

Notes on Randomized Algorithms

James Aspnes

2020-10-29 10:43

Copyright © 2009–2021 by James Aspnes. Distributed under a Creative Commons Attribution-ShareAlike 4.0 International license: <https://creativecommons.org/licenses/by-sa/4.0/>.

Contents

Table of contents	ii
List of figures	xiv
List of tables	xv
List of algorithms	xvi
Preface	xvii
Syllabus	xviii
Lecture schedule	xx
1 Randomized algorithms	1
1.1 A trivial example	2
1.2 Verifying polynomial identities	3
1.3 Randomized QuickSort	5
1.3.1 Brute force method: solve the recurrence	5
1.3.2 Clever method: use linearity of expectation	6
1.4 Where does the randomness come from?	8
1.5 Classifying randomized algorithms	9
1.5.1 Las Vegas vs Monte Carlo	10
1.5.2 Randomized complexity classes	11
1.6 Classifying randomized algorithms by their methods	12
2 Probability theory	14
2.1 Probability spaces and events	15
2.1.1 General probability spaces	15
2.2 Boolean combinations of events	17

2.3	Conditional probability	20
2.3.1	Conditional probability and independence	20
2.3.2	Conditional probability and the law of total probability	20
2.3.3	Examples	22
2.3.3.1	Racing coin-flips	22
2.3.3.2	Karger's min-cut algorithm	24
3	Random variables	27
3.1	Operations on random variables	28
3.2	Random variables and events	28
3.3	Measurability	30
3.4	Expectation	31
3.4.1	Linearity of expectation	31
3.4.1.1	Linearity of expectation for infinite sequences	33
3.4.2	Expectation and inequalities	33
3.4.3	Expectation of a product	34
3.4.3.1	Wald's equation (simple version)	34
3.5	Conditional expectation	36
3.5.1	Expectation conditioned on an event	36
3.5.2	Expectation conditioned on a random variable	37
3.5.2.1	Calculating conditional expectations	39
3.5.2.2	The law of iterated expectation	40
3.5.2.3	Conditional expectation as orthogonal projection (optional)	41
3.5.3	Expectation conditioned on a σ -algebra	42
3.5.4	Examples	44
3.6	Applications	44
3.6.1	Yao's lemma	44
3.6.2	Geometric random variables	46
3.6.3	Coupon collector	48
3.6.4	Hoare's FIND	48
4	Basic probabilistic inequalities	50
4.1	Union bound (Boole's inequality)	50
4.1.1	Applications	51
4.1.1.1	Balls in bins	51
4.2	Markov's inequality	52
4.2.1	Applications	53
4.2.1.1	The union bound	53
4.2.1.2	Fair coins	53

4.2.1.3	Randomized QuickSort	53
4.2.1.4	Balls in bins	53
4.3	Jensen's inequality	54
4.3.1	Proof	54
4.3.2	Applications	55
4.3.2.1	Fair coins: lower bound	55
4.3.2.2	Fair coins: upper bound	56
4.3.2.3	Sifters	56
5	Concentration bounds	58
5.1	Chebyshev's inequality	58
5.1.1	Computing variance	59
5.1.1.1	Alternative formula	59
5.1.1.2	Variance of a Bernoulli random variable	60
5.1.1.3	Variance of a sum	60
5.1.1.4	Variance of a geometric random variable	61
5.1.2	More examples	65
5.1.2.1	Flipping coins	65
5.1.2.2	Balls in bins	65
5.1.2.3	Lazy select	66
5.2	Chernoff bounds	67
5.2.1	The classic Chernoff bound	68
5.2.2	Easier variants	70
5.2.3	Lower bound version	71
5.2.4	Two-sided version	71
5.2.5	Almost-independent variables	72
5.2.6	Other tail bounds for the binomial distribution	73
5.2.7	Applications	73
5.2.7.1	Flipping coins	73
5.2.7.2	Balls in bins again	74
5.2.7.3	Flipping coins, central behavior	74
5.2.7.4	Permutation routing on a hypercube	75
5.3	The Azuma-Hoeffding inequality	78
5.3.1	Hoeffding's inequality	78
5.3.1.1	Hoeffding vs Chernoff	80
5.3.1.2	Asymmetric version	81
5.3.2	Azuma's inequality	81
5.3.3	The method of bounded differences	86
5.3.4	Applications	88
5.3.4.1	Sprinkling points on a hypercube	88

5.3.4.2	Chromatic number of a random graph	88
5.3.4.3	Balls in bins	89
5.3.4.4	Probabilistic recurrence relations	89
5.3.4.5	Multi-armed bandits	91
	The UCB1 algorithm	92
	Analysis of UCB1	93
5.4	Relation to limit theorems	96
5.5	Anti-concentration bounds	96
5.5.1	The Berry-Esseen theorem	97
5.5.2	The Littlewood-Offord problem	97
6	Randomized search trees	99
6.1	Binary search trees	99
6.1.1	Rebalancing and rotations	100
6.2	Random insertions	100
6.3	Treaps	102
6.3.1	Assumption of an oblivious adversary	104
6.3.2	Analysis	104
6.3.2.1	Searches	106
6.3.2.2	Insertions and deletions	106
6.3.2.3	Other operations	108
6.4	Skip lists	108
7	Hashing	110
7.1	Hash tables	110
7.2	Universal hash families	112
7.2.1	Linear congruential hashing	114
7.2.2	Tabulation hashing	115
7.3	FKS hashing	116
7.4	Cuckoo hashing	118
7.4.1	Structure	118
7.4.2	Analysis	120
7.5	Practical issues	122
7.6	Bloom filters	123
7.6.1	Construction	123
7.6.2	False positives	123
7.6.3	Comparison to optimal space	126
7.6.4	Applications	127
7.6.5	Counting Bloom filters	128
7.6.6	Count-min sketches	129

7.6.6.1	Initialization and updates	129
7.6.6.2	Queries	129
7.6.6.3	Finding heavy hitters	132
7.7	Locality-sensitive hashing	133
7.7.1	Approximate nearest neighbor search	133
7.7.1.1	Locality-sensitive hash functions	134
7.7.1.2	Constructing an (r_1, r_2) -PLEB	135
7.7.1.3	Hash functions for Hamming distance	136
7.7.1.4	Hash functions for ℓ_1 distance	138
8	Martingales and stopping times	140
8.1	Definitions	140
8.2	Submartingales and supermartingales	141
8.3	The optional stopping theorem	142
8.4	Proof of the optional stopping theorem (optional)	143
8.5	Applications	145
8.5.1	Random walks	145
8.5.2	Wald's equation	149
8.5.3	Maximal inequalities	150
8.5.4	Waiting times for patterns	150
9	Markov chains	152
9.1	Basic definitions and properties	153
9.1.1	Examples	155
9.2	Convergence of Markov chains	156
9.2.1	Stationary distributions	157
9.2.2	Total variation distance	158
9.2.2.1	Total variation distance and expectation	159
9.2.3	Mixing time	160
9.2.4	Coupling of Markov chains	160
9.2.5	Irreducible and aperiodic chains	162
9.2.6	Convergence of finite irreducible aperiodic Markov chains	163
9.3	Reversible chains	164
9.3.1	Stationary distributions	165
9.3.2	Examples	166
9.3.3	Time-reversed chains	166
9.3.4	Adjusting stationary distributions with the Metropolis-Hastings algorithm	168
9.4	The coupling method	169
9.4.1	Random walk on a cycle	170

9.4.2	Random walk on a hypercube	171
9.4.3	Various shuffling algorithms	172
9.4.3.1	Move-to-top	172
9.4.3.2	Random exchange of arbitrary cards	173
9.4.3.3	Random exchange of adjacent cards	174
9.4.3.4	Real-world shuffling	175
9.4.4	Path coupling	175
9.4.5	Random walk on a hypercube	176
9.4.5.1	Sampling graph colorings	177
9.4.5.2	Simulated annealing	180
	Single peak	180
	Somewhat smooth functions	181
9.5	Spectral methods for reversible chains	182
9.5.1	Spectral properties of a reversible chain	182
9.5.2	Analysis of symmetric chains	183
9.5.3	Analysis of asymmetric chains	185
9.5.4	Conductance	186
9.5.5	Easy cases for conductance	187
9.5.6	Edge expansion using canonical paths	188
9.5.7	Congestion	190
9.5.8	Examples	191
9.5.8.1	Lazy random walk on a line	191
9.5.8.2	Random walk on a hypercube	191
9.5.8.3	Matchings in a graph	192
9.5.8.4	Perfect matchings in dense bipartite graphs	195
10	Approximate counting	197
10.1	Exact counting	197
10.2	Counting by sampling	198
10.2.1	Generating samples	199
10.3	Approximating #DNF	200
10.4	Approximating #KNAPSACK	202
10.5	Approximating exponentially improbable events	204
10.5.1	Matchings	205
10.5.2	Other problems	206
11	The probabilistic method	207
11.1	Randomized constructions and existence proofs	207
11.1.1	Unique hats	208
11.1.2	Ramsey numbers	209

11.1.3 Directed cycles in tournaments	211
11.2 Approximation algorithms	212
11.2.1 MAX CUT	212
11.2.2 MAX SAT	213
11.3 The Lovász Local Lemma	217
11.3.1 General version	217
11.3.2 Symmetric version	218
11.3.3 Applications	219
11.3.3.1 Graph coloring	219
11.3.3.2 Satisfiability of k -CNF formulas	219
11.3.3.3 Hypergraph 2-colorability	220
11.3.4 Non-constructive proof	220
11.3.5 Constructive proof	223
12 Derandomization	228
12.1 Deterministic vs. randomized algorithms	229
12.2 Adleman's theorem	230
12.3 Limited independence	231
12.3.1 MAX CUT	231
12.4 The method of conditional probabilities	232
12.4.1 A trivial example	233
12.4.2 Deterministic construction of Ramsey graphs	233
12.4.3 MAX CUT using conditional probabilities	234
12.4.4 Set balancing	235
13 Quantum computing	236
13.1 Random circuits	236
13.2 Bra-ket notation	239
13.2.1 States as kets	240
13.2.2 Operators as sums of kets times bras	240
13.3 Quantum circuits	241
13.3.1 Quantum operations	242
13.3.2 Quantum implementations of classical operations	243
13.3.3 Representing Boolean functions	245
13.3.4 Practical issues (which we will ignore)	245
13.3.5 Quantum computations	246
13.4 Deutsch's algorithm	246
13.5 Grover's algorithm	247
13.5.1 Initial superposition	248
13.5.2 The Grover diffusion operator	248

13.5.3	Effect of the iteration	249
14	Randomized distributed algorithms	251
14.1	Consensus	252
14.1.1	Impossibility of deterministic algorithms	253
14.2	Leader election	254
14.3	How randomness helps	254
14.4	Building a weak shared coin	255
14.5	Leader election with sifters	257
14.6	Consensus with sifters	258
A	Assignments	261
A.1	Assignment 1: due Wednesday, 2021-02-24, at 23:00	261
A.2	Assignment 2: due Friday, 2021-03-12, at 23:00	261
A.3	Assignment 3: due Friday, 2021-03-26, at 23:00	262
A.4	Assignment 4: due Wednesday, 2021-04-07, at 23:00	262
A.5	Assignment 5: due Wednesday, 2021-04-21, at 23:00	262
A.6	Assignment 6: due Wednesday, 2021-05-05, at 23:00	262
B	Sample assignments from Fall 2019	263
B.1	Assignment 1: due Thursday, 2019-09-12, at 23:00	263
B.1.1	The golden ticket	263
B.1.2	Exploding computers	265
B.2	Assignment 2: due Thursday, 2019-09-26, at 23:00	267
B.2.1	A logging problem	267
B.2.2	Return of the exploding computers	268
B.3	Assignment 3: due Thursday, 2019-10-10, at 23:00	269
B.3.1	Two data plans	269
B.3.2	A randomly-indexed list	271
B.4	Assignment 4: due Thursday, 2019-10-31, at 23:00	272
B.4.1	A hash tree	272
B.4.2	Randomized robot rendezvous on a ring	273
B.5	Assignment 5: due Thursday, 2019-11-14, at 23:00	275
B.5.1	Non-exploding computers	275
B.5.2	A wordy walk	277
B.6	Assignment 6: due Monday, 2019-12-09, at 23:00	279
B.6.1	Randomized colorings	279
B.6.2	No long paths	280
B.7	Final exam	282
B.7.1	A uniform ring	282

B.7.2	Forbidden runs	284
B.7.3	A derandomized balancing scheme	285
C	Sample assignments from Fall 2016	287
C.1	Assignment 1: due Sunday, 2016-09-18, at 17:00	287
C.1.1	Bubble sort	287
C.1.2	Finding seats	289
C.2	Assignment 2: due Thursday, 2016-09-29, at 23:00	291
C.2.1	Technical analysis	291
C.2.2	Faulty comparisons	295
C.3	Assignment 3: due Thursday, 2016-10-13, at 23:00	296
C.3.1	Painting with sprites	296
C.3.2	Dynamic load balancing	298
C.4	Assignment 4: due Thursday, 2016-11-03, at 23:00	299
C.4.1	Re-rolling a random treap	299
C.4.2	A defective hash table	302
C.5	Assignment 5: due Thursday, 2016-11-17, at 23:00	304
C.5.1	A spectre is haunting Halloween	304
C.5.2	Colliding robots on a line	305
C.6	Assignment 6: due Thursday, 2016-12-08, at 23:00	309
C.6.1	Another colliding robot	309
C.6.2	Return of the sprites	312
C.7	Final exam	314
C.7.1	Virus eradication (20 points)	314
C.7.2	Parallel bubblesort (20 points)	315
C.7.3	Rolling a die (20 points)	316
D	Sample assignments from Spring 2014	318
D.1	Assignment 1: due Wednesday, 2014-09-10, at 17:00	318
D.1.1	Bureaucratic part	318
D.1.2	Two terrible data structures	318
D.1.3	Parallel graph coloring	322
D.2	Assignment 2: due Wednesday, 2014-09-24, at 17:00	324
D.2.1	Load balancing	324
D.2.2	A missing hash function	324
D.3	Assignment 3: due Wednesday, 2014-10-08, at 17:00	325
D.3.1	Tree contraction	325
D.3.2	Part testing	328
	Using McDiarmid's inequality and some cleverness	329
D.4	Assignment 4: due Wednesday, 2014-10-29, at 17:00	330

D.4.1	A doubling strategy	330
D.4.2	Hash treaps	331
D.5	Assignment 5: due Wednesday, 2014-11-12, at 17:00	332
D.5.1	Agreement on a ring	332
D.5.2	Shuffling a two-dimensional array	335
D.6	Assignment 6: due Wednesday, 2014-12-03, at 17:00	336
D.6.1	Sampling colorings on a cycle	336
D.6.2	A hedging problem	337
D.7	Final exam	338
D.7.1	Double records (20 points)	339
D.7.2	Hipster graphs (20 points)	340
	Using the method of conditional probabilities	340
	Using hill climbing	341
D.7.3	Storage allocation (20 points)	341
D.7.4	Fault detectors in a grid (20 points)	342
E	Sample assignments from Spring 2013	345
E.1	Assignment 1: due Wednesday, 2013-01-30, at 17:00	345
E.1.1	Bureaucratic part	345
E.1.2	Balls in bins	345
E.1.3	A labeled graph	346
E.1.4	Negative progress	346
E.2	Assignment 2: due Thursday, 2013-02-14, at 17:00	348
E.2.1	A local load-balancing algorithm	348
E.2.2	An assignment problem	350
E.2.3	Detecting excessive collusion	350
E.3	Assignment 3: due Wednesday, 2013-02-27, at 17:00	352
E.3.1	Going bowling	352
E.3.2	Unbalanced treaps	353
E.3.3	Random radix trees	355
E.4	Assignment 4: due Wednesday, 2013-03-27, at 17:00	356
E.4.1	Flajolet-Martin sketches with deletion	356
E.4.2	An adaptive hash table	358
E.4.3	An odd locality-sensitive hash function	360
E.5	Assignment 5: due Friday, 2013-04-12, at 17:00	361
E.5.1	Choosing a random direction	361
E.5.2	Random walk on a tree	362
E.5.3	Sampling from a tree	363
E.6	Assignment 6: due Friday, 2013-04-26, at 17:00	364
E.6.1	Increasing subsequences	364

E.6.2	Futile word searches	365
E.6.3	Balance of power	367
E.7	Final exam	368
E.7.1	Dominating sets	368
E.7.2	Tricolor triangles	369
E.7.3	The n rooks problem	370
E.7.4	Pursuing an invisible target on a ring	370
F	Sample assignments from Spring 2011	372
F.1	Assignment 1: due Wednesday, 2011-01-26, at 17:00	372
F.1.1	Bureaucratic part	372
F.1.2	Rolling a die	372
F.1.3	Rolling many dice	374
F.1.4	All must have candy	374
F.2	Assignment 2: due Wednesday, 2011-02-09, at 17:00	375
F.2.1	Randomized dominating set	375
F.2.2	Chernoff bounds with variable probabilities	377
F.2.3	Long runs	378
F.3	Assignment 3: due Wednesday, 2011-02-23, at 17:00	380
F.3.1	Longest common subsequence	380
F.3.2	A strange error-correcting code	382
F.3.3	A multiway cut	383
F.4	Assignment 4: due Wednesday, 2011-03-23, at 17:00	384
F.4.1	Sometimes successful betting strategies are possible	384
F.4.2	Random walk with reset	386
F.4.3	Yet another shuffling algorithm	388
F.5	Assignment 5: due Thursday, 2011-04-07, at 23:59	389
F.5.1	A reversible chain	389
F.5.2	Toggling bits	390
F.5.3	Spanning trees	392
F.6	Assignment 6: due Monday, 2011-04-25, at 17:00	393
F.6.1	Sparse satisfying assignments to DNFs	393
F.6.2	Detecting duplicates	394
F.6.3	Balanced Bloom filters	395
F.7	Final exam	398
F.7.1	Leader election	398
F.7.2	Two-coloring an even cycle	399
F.7.3	Finding the maximum	400
F.7.4	Random graph coloring	401

G Sample assignments from Spring 2009	402
G.1 Final exam, Spring 2009	402
G.1.1 Randomized mergesort (20 points)	402
G.1.2 A search problem (20 points)	403
G.1.3 Support your local police (20 points)	404
G.1.4 Overloaded machines (20 points)	405
H Probabilistic recurrences	406
H.1 Recurrences with constant cost functions	406
H.2 Examples	406
H.3 The Karp-Upfal-Wigderson bound	407
H.3.1 Waiting for heads	409
H.3.2 Quickselect	409
H.3.3 Tossing coins	409
H.3.4 Coupon collector	410
H.3.5 Chutes and ladders	410
H.4 High-probability bounds	411
H.4.1 High-probability bounds from expectation bounds	412
H.4.2 Detailed analysis of the recurrence	412
H.5 More general recurrences	413
Bibliography	414
Index	429

List of Figures

2.1	Karger's min-cut algorithm	25
4.1	Proof of Jensen's inequality	55
5.1	Comparison of Chernoff bound variants	71
5.2	Hypercube network with $n = 3$	75
6.1	Tree rotations	100
6.2	Balanced and unbalanced binary search trees	101
6.3	Binary search tree after inserting 5 1 7 3 4 6 2	102
6.4	Inserting values into a treap	103
6.5	Tree rotation shortens spines	107
6.6	Skip list	108
11.1	Tricky step in MAX SAT argument	216
C.1	Filling a screen with Space Invaders	297
C.2	Hidden Space Invaders	313
D.1	Example of tree contraction for Problem D.3.1	326
E.1	Radix tree	355
E.2	Word searches	366

List of Tables

3.1	Sum of two dice	29
3.2	Various conditional expectations on two independent dice . . .	45
5.1	Concentration bounds	59
7.1	Hash table parameters	112

List of Algorithms

7.1	Insertion procedure for cuckoo hashing	119
14.1	Attiya-Censor weak shared coin [AC08]	256
14.2	Giakkoupis-Woelfel sifter [GW12]	257
14.3	Sifter using max registers [AE19]	260
C.1	One pass of bubble sort	288
E.1	Adaptive hash table insertion	359
F.1	Dubious duplicate detector	394
F.2	Randomized max-finding algorithm	400

Preface

These are notes for the Yale course CPSC 469/569 Randomized Algorithms. This document also incorporates the lecture schedule and assignments, as well as some sample assignments from previous semesters. Because this is a work in progress, it will be updated frequently over the course of the semester.

Much of the structure of the course follows Mitzenmacher and Upfal's *Probability and Computing: Randomized Algorithms and Probabilistic Analysis* [MU05], with some material from Motwani and Raghavan's *Randomized Algorithms* [MR95]. In most cases you'll find these textbooks contain much more detail than what is presented here, so it is probably better to consider this document a supplement to them than to treat it as your primary source of information.

The most recent version of these notes will be available at <http://www.cs.yale.edu/homes/aspnes/classes/469/notes.pdf>. More stable archival versions may be found at <https://arxiv.org/abs/2003.01902>.

I would like to thank my many students and teaching fellows over the years for their help in pointing out errors and omissions in earlier drafts of these notes.

Syllabus

Description

A study of randomized algorithms from several areas: graph algorithms, algorithms in algebra, approximate counting, probabilistically checkable proofs, and matrix algorithms. Topics include an introduction to tools from probability theory, including some inequalities such as Chernoff bounds.

Meeting times

Lectures are Tuesdays and Thursdays, from 14:30 to 15:45 Eastern US time. Zoom link: <https://yale.zoom.us/j/220040380>. Recorded lectures will be available through Canvas if you can't make it to the live lectures.

On-line course information

The lecture schedule, course notes, and all assignments can be found in a single gigantic PDF file at <http://www.cs.yale.edu/homes/aspnes/classes/469/notes.pdf>. You should probably bookmark this file, as it will be updated frequently.

Office hours can be found in the [course calendar at Google Calendar](#), which can also be reached through [James Aspnes's web page](#).

Staff

The instructor for the course is James Aspnes. Office: AKW 401. Email: james.aspnes@gmail.com. URL: <http://www.cs.yale.edu/homes/aspnes/>.

Textbook

The primary text for this class will be the lecture notes. We will also make reference as needed to some of the reserved books below.

Course requirements

Six homework assignments (100% of the semester grade).

Use of outside help

Students are free to discuss homework problems and course material with each other, and to consult with the instructor or a TA. Solutions handed in, however, should be the student's own work. If a student benefits substantially from hints or solutions received from fellow students or from outside sources, then the student should hand in their solution but acknowledge the outside sources, and we will apportion credit accordingly. Using outside resources in solving a problem is acceptable but plagiarism is not.

Clarifications for homework assignments

From time to time, ambiguities and errors may creep into homework assignments. Questions about the interpretation of homework assignments can be sent to the instructor at james.aspnes@gmail.com. Clarifications will appear in an updated version of the assignment.

In some circumstances, you may be able to get a faster response using Piazza, at piazza.com/yale/spring2021/cpsc469. Note that questions you ask there are visible to other students if not specifically marked private, so be careful about broadcasting your draft solutions.

Late assignments

Late assignments will not be accepted without a Dean's Excuse.

Lecture schedule

As always, the future is uncertain, so you should take parts of the schedule that haven't happened yet with a grain of salt.

2021-02-02 [[[To be announced.]]

2021-02-04 [[[To be announced.]]

2021-02-09 [[[To be announced.]]

2021-02-11 [[[To be announced.]]

2021-02-16 [[[To be announced.]]

2021-02-18 [[[To be announced.]]

2021-02-23 [[[To be announced.]]

2021-02-25 [[[To be announced.]]

2021-03-02 [[[To be announced.]]

2021-03-04 [[[To be announced.]]

2021-03-11 [[[To be announced.]]

2021-03-16 [[[To be announced.]]

2021-03-18 [[[To be announced.]]

2021-03-23 [[[To be announced.]]

2021-03-25 [[[To be announced.]]

2021-03-30 [[[To be announced.]]

2021-04-01 [[[To be announced.]]

LECTURE SCHEDULE

xxi

2021-04-06 [[[To be announced.]]

2021-04-13 [[[To be announced.]]

2021-04-15 [[[To be announced.]]

2021-04-20 [[[To be announced.]]

2021-04-22 [[[To be announced.]]

2021-04-27 [[[To be announced.]]

2021-04-29 [[[To be announced.]]

2021-05-04 [[[To be announced.]]

2021-05-06 [[[To be announced.]]

Chapter 1

Randomized algorithms

Last updated 2019. Some material may be out of date.

A randomized algorithm flips coins during its execution to determine what to do next. When considering a randomized algorithm, we usually care about its **expected worst-case** performance, which is the average amount of time it takes on the worst input of a given size. This average is computed over all the possible outcomes of the coin flips during the execution of the algorithm. We may also ask for a **high-probability bound**, showing that the algorithm doesn't consume too much resources most of the time.

In studying randomized algorithms, we consider pretty much the same issues as for deterministic algorithms: how to design a good randomized algorithm, and how to prove that it works within given time or error bounds. The main difference is that it is often easier to design a randomized algorithm—randomness turns out to be a good substitute for cleverness more often than one might expect—but harder to analyze it. So much of what one does is develop good techniques for analyzing the often very complex random processes that arise in the execution of an algorithm. Fortunately, in doing so we can often use techniques already developed by probabilists and statisticians for analyzing less overtly algorithmic processes.

Formally, we think of a randomized algorithm as a machine M that computes $M(x, r)$, where x is the problem input and r is the sequence of random bits. Our machine model is the usual **random-access machine** or **RAM** model, where we have a memory space that is typically polynomial in the size of the input n , and in constant time we can read a memory location, write a memory location, or perform arithmetic operations on integers of

up to $O(\log n)$ bits.¹ In this model, we may find it easier to think of the random bits as supplied as needed by some subroutine, where generating a random integer of size $O(\log n)$ takes constant time; the justification for this assumption is that it takes constant time to read the next $O(\log n)$ -sized value from the random input.

Because the number of these various constant-time operations, and thus the running time for the algorithm as a whole, may depend on the random bits, it is now a **random variable**—a function on points in some probability space. The probability space Ω consists of all possible sequences r , each of which is assigned a probability $\Pr[r]$ (typically $2^{-|r|}$), and the running time for M on some input x is generally given as an **expected value**² $E_r[\text{time}(M(x, r))]$, where for any X ,

$$E_r[X] = \sum_{r \in \Omega} X(r) \Pr[r]. \quad (1.0.1)$$

We can now quote the performance of M in terms of this expected value: where we would say that a deterministic algorithm runs in time $O(f(n))$, where $n = |x|$ is the size of the input, we instead say that our randomized algorithm runs in **expected time** $O(f(n))$, which means that $E_r[\text{time}(M(x, r))] = O(f(|x|))$ for all inputs x .

This is distinct from traditional **worst-case analysis**, where there is no r and no expectation, and **average-case analysis**, where there is again no r and the value reported is not a maximum but an expectation over some distribution on x . The following trivial example shows the distinction.

1.1 A trivial example

Let us consider a variant of the classic card game **Find the Lady**³ Here a dealer puts down three cards and we want to find a specific card among the

¹This model is unrealistic in several ways: the assumption that we can perform arithmetic on $O(\log n)$ -bit quantities in constant time omits at least a factor of $\Omega(\log \log n)$ for addition and probably more for multiplication in any realistic implementation, while the assumption that we can address n^c distinct locations in memory in anything less than $n^{c/3}$ time in the worst case requires exceeding the speed of light. But for reasonably small n , this gives a pretty good approximation of the performance of real computers, which do in fact perform arithmetic and access memory in a fixed amount of time, although with fixed bounds on the size of both arithmetic operands and memory.

²We'll see more details of these and other concepts from probability theory in Chapters 2 and 3.

³Often called **Three-card Monte**, but that will get confusing when we talk about *Monte Carlo* algorithms later.

three (say, the Queen of Spades). In this version, the dealer will let us turn over as many cards as we want, but each card we turn over will cost us a dollar. If we find the queen, we get two dollars.

Because this is a toy example, we assume that the dealer is not cheating. The author is not responsible for the results of applying the analysis below to real-world games where this assumption does not hold.

A deterministic algorithm tries the cards in some fixed order. A clever dealer will place the Queen in the last place we look: so the worst-case payoff is a loss of a dollar.

In the average case, if we assume that all three positions are equally likely to hold the target card, then we turn over one card a third of the time, two cards a third of the time, and all three cards a third of the time; this gives an expected payoff of

$$\frac{1}{3}(1 + 2 + 3) - 2 = 0.$$

But this requires making assumptions about the distribution of the cards, and we are no longer doing worst-case analysis.

The trick to randomized algorithms is that we can obtain the same expected payoff even in the worst case by supplying the randomness ourselves. If we turn over cards in a random order, then the same analysis for the average case tells us we get the same expected payoff of 0—but unlike the average case, we get this expected performance no matter where the dealer places the cards.

1.2 Verifying polynomial identities

A less trivial example is described in [MU05, §1.1]. Here we are given two products of polynomials and we want to determine if they compute the same function. For example, we might have

$$\begin{aligned} p(x) &= (x - 7)(x - 3)(x - 1)(x + 2)(2x + 5) \\ q(x) &= 2x^5 - 13x^4 - 21x^3 + 127x^2 + 121x - 210 \end{aligned}$$

These expressions both represent degree-5 polynomials, and it is not obvious without multiplying out the factors of p whether they are equal or not. Multiplying out all the factors of p may take as much as $O(d^2)$ time if we assume integer multiplication takes unit time and do it the straightforward way.⁴ We can do better than this using randomization.

⁴It can be faster if we do something sneaky like use fast Fourier transforms [SS71].

The trick is that evaluating $p(x)$ and $q(x)$ takes only $O(d)$ integer operations, and we will find $p(x) = q(x)$ only if either (a) $p(x)$ and $q(x)$ are really the same polynomial, or (b) x is a **root** of $p(x) - q(x)$. Since $p(x) - q(x)$ has degree at most d , it can't have more than d roots. So if we choose x uniformly at random from some much larger space, it's likely that we will not get a root. Indeed, evaluating $p(11) = 112320$ and $q(11) = 120306$ quickly shows that p and q are not in fact the same.

This is an example of a **Monte Carlo algorithm**, which is an algorithm that runs in a fixed amount of time but only gives the right answer some of the time. (In this case, with probability $1 - d/r$, where r is the size of the range of random integers we choose x from.) Monte Carlo algorithms have the unnerving property of not indicating when their results are incorrect, but we can make the probability of error as small as we like by running the algorithm repeatedly. For this particular algorithm, the probability of error after k trials is only $(d/r)^k$, which means that for fixed d/r we need $O(\log(1/\epsilon))$ iterations to get the error bound down to any given ϵ . If we are really paranoid, we could get the error down to 0 by testing $d + 1$ distinct values, but now the cost is as high as multiplying out p again.

The error for this algorithm is one-sided: if we find a **witness** to the fact that $p \neq q$, we are done, but if we don't, then all we know is that we haven't found a witness yet. We also have the property that if we check enough possible witnesses, we are guaranteed to find one.

A similar property holds in the classic **Miller-Rabin primality test**, a randomized algorithm for determining whether a large integer is prime or not.⁵ The original version, due to Gary Miller [Mil76] showed that, as in polynomial identity testing, it might be sufficient to pick a particular set of deterministic candidate witnesses. Unfortunately, this result depends on the truth of the extended Riemann hypothesis, a notoriously difficult open problem in number theory. Michael Rabin [Rab80] demonstrated that choosing random witnesses was enough, if we were willing to accept a small probability of incorrectly identifying a composite number as prime.

For many years it was open whether it was possible to test primality deterministically in polynomial time without unproven number-theoretic assumptions, and the randomized Miller-Rabin algorithm was one of the most widely-used randomized algorithms for which no good deterministic alternative was known. Eventually, Agrawal *et al.* [AKS04] demonstrated how to test primality deterministically using a different technique, although the cost of their algorithm is high enough that Miller-Rabin is still used in

⁵We will not describe this algorithm here.

practice.

1.3 Randomized QuickSort

The **QuickSort** algorithm [Hoa61a] works as follows. For simplicity, we assume that no two elements of the array being sorted are equal.

- If the array has > 1 elements,
 - Pick a **pivot** p uniformly at random from the elements of the array.
 - Split the array into A_1 and A_2 , where A_1 contains all elements $< p$ elements $> p$.
 - Sort A_1 and A_2 recursively and return the sequence A_1, p, A_2 .
- Otherwise return the array.

The splitting step takes exactly $n - 1$ comparisons, since we have to check each non-pivot against the pivot. We assume all other costs are dominated by the cost of comparisons. How many comparisons does randomized QuickSort do on average?

There are two ways to solve this problem: the dumb way and the smart way. We'll do it the dumb way now and save the smart way for §1.3.2.

1.3.1 Brute force method: solve the recurrence

Let $T(n)$ be the expected number of comparisons done on an array of n elements. We have $T(0) = T(1) = 0$ and for larger n ,

$$T(n) = \frac{1}{n} \sum_{k=0}^{n-1} (T(k) + T(n-1-k)). \quad (1.3.1)$$

Why? Because there are n equally-likely choices for our pivot (hence the $1/n$), and for each choice the expected cost is $T(k) + T(n-1-k)$, where k is the number of elements that land in A_1 . Formally, we are using here the **law of total probability**, which says that for any random variable X and partition of the probability space into events $B_1 \dots B_n$, then

$$E[X] = \sum \Pr[B_i] E[X | B_i],$$

where

$$E[X \mid B_i] = \sum_{\omega \in B_i} X(\omega) \frac{\Pr[\omega]}{\Pr[B_i]}$$

is the **conditional expectation** of X conditioned on B_i , which we can think of as just the average value of X if we know that B_i occurred. (See §2.3.2 for more details.)

So now we just have to solve this ugly recurrence. We can reasonably guess that when $n \geq 1$, $T(n) \leq an \log n$ for some constant a . Clearly this holds for $n = 1$. Now apply induction on larger n to get

$$\begin{aligned} T(n) &= (n-1) + \frac{1}{n} \sum_{k=0}^{n-1} (T(k) + T(n-1-k)) \\ &= (n-1) + \frac{2}{n} \sum_{k=0}^{n-1} T(k) \\ &= (n-1) + \frac{2}{n} \sum_{k=1}^{n-1} T(k) \\ &\leq (n-1) + \frac{2}{n} \sum_{k=1}^{n-1} ak \log k \\ &\leq (n-1) + \frac{2}{n} \int_{k=1}^n ak \log k \\ &= (n-1) + \frac{2a}{n} \left(\frac{n^2 \log n}{2} - \frac{n^2}{4} + \frac{1}{4} \right) \\ &= (n-1) + \frac{2a}{n} \left(\frac{n^2 \log n}{2} - \frac{n^2}{4} + \frac{1}{4} \right) \\ &= (n-1) + an \log n - \frac{an}{2} + \frac{a}{2n}. \end{aligned}$$

If we squint carefully at this recurrence for a while we notice that setting $a = 2$ makes this less than or equal to $an \log n$, since the remaining terms become $(n-1) - n + 1/n = 1/n - 1$, which is negative for $n \geq 1$. We can thus confidently conclude that $T(n) \leq 2n \log n$ (for $n \geq 1$).

1.3.2 Clever method: use linearity of expectation

Alternatively, we can use **linearity of expectation** (which we'll discuss further in §3.4.1 to compute the expected number of comparisons used by randomized QuickSort.

Imagine we use the following method for choosing pivots: we generate a random permutation of all the elements in the array, and when asked to sort some subarray A' , we use as pivot the first element of A' that appears in our list. Since each element is equally likely to be first, this is equivalent to the actual algorithm. Pretend that we are always sorting the numbers $1 \dots n$ and define for each pair of elements $i < j$ the **indicator variable** X_{ij} to be 1 if i is compared to j at some point during the execution of the algorithm and 0 otherwise. Amazingly, we can actually compute the probability of this event (and thus $E[X_{ij}]$): the only time i and j are compared is if one of them is chosen as a pivot before they are split up into different arrays. How do they get split up into different arrays? This happens if some intermediate element k is chosen as pivot first, that is, if some k with $i < k < j$ appears in the permutation before both i and j . Occurrences of other elements don't affect the outcome, so we can concentrate on the restriction of the permutations to just the numbers i through j , and we win if this restricted permutation starts with either i or j . This event occurs with probability $2/(j - i + 1)$, so we have $E[X_{ij}] = 2/(j - i + 1)$. Summing over all pairs $i < j$ gives:

$$\begin{aligned}
E \left[\sum_{i < j} X_{ij} \right] &= \sum_{i < j} E[X_{ij}] \\
&= \sum_{i < j} \frac{2}{j - i + 1} \\
&= \sum_{i=1}^{n-1} \sum_{k=2}^{n-i+1} \frac{2}{k} \\
&= \sum_{i=2}^n \sum_{k=2}^i \frac{2}{k} \\
&= \sum_{k=2}^n \frac{2(n - k + 1)}{k} \\
&= \sum_{k=2}^n \left(\frac{2(n + 1)}{k} - 2 \right) \\
&= \sum_{k=2}^n \frac{2(n + 1)}{k} - 2(n - 1) \\
&= 2(n + 1)(H_n - 1) - 2(n - 1) \\
&= 2(n + 1)H_n - 4n.
\end{aligned}$$

Here $H_n = \sum_{i=1}^n \frac{1}{i}$ is the n -th **harmonic number**, equal to $\ln n + \gamma +$

$O(n^{-1})$, where $\gamma \approx 0.5772$ is the **Euler-Mascheroni constant** (whose exact value is unknown!). For asymptotic purposes we only need $H_n = \Theta(\log n)$.

For the first step we are taking advantage of the fact that linearity of expectation doesn't care about the variables not being independent. The rest is just algebra.

This is pretty close to the bound of $2n \log n$ we computed using the recurrence in §1.3.1. Given that we now know the exact answer, we could in principle go back and use it to solve the recurrence exactly.⁶

Which way is better? Solving the recurrence requires less probabilistic **handwaving** (a more polite term might be “insight”) but more grinding out inequalities, which is a pretty common trade-off. Since I am personally not very clever I would try the brute-force approach first. But it's worth knowing about better methods so you can try them in other situations.

1.4 Where does the randomness come from?

Typically we assume in analyzing a randomized algorithm that we have access to genuinely random bits r , and don't ask too carefully how we can get these random bits. For practical applications, there are basically three choices, from strongest (and most expensive) to weakest (and cheapest):

1. **Physical randomness.** Lock a cat in a box with a radioactive source and a Geiger counter attached to a solenoid aimed at a vial of prussic acid. Come back in an hour and check if the cat is still breathing. With an appropriately-tuned radioactive source, this generates one very unpredictable bit of randomness at the cost of one half of a cat on average.⁷

There are cheaper variants, mostly involving amplified quantum noise at the high end, or monitoring physical processes that are expected to be somewhat random (like intervals between keyboard presses or seek times of hard drive heads) at the low end. In each case you get a sequence of random bits that, under plausible physical assumptions, are effectively unpredictable.

The `/dev/random` device in Linux systems gives you access to the cheap kind of physical randomness, and will block waiting for you to wave your mouse around if it runs out.

⁶We won't.

⁷Expected cost may be higher if the observer forgets their gas mask.

2. **Cryptographically-secure pseudorandomness.** Find some function that spits out a sequence of random-looking values given a **seed**, such that if the seed is chosen uniformly at random (say using a physical random number generator), no polynomial-time program can distinguish the sequence from an actual random sequence with non-trivial probability. Usually expensive, but if your polynomial-time randomized algorithm fails using a CPRNG, you've broken the CPRNG.

The `/dev/urandom` device in Linux systems gives you this, based on a seed derived from the same sources as `/dev/random`.

3. **Statistical pseudorandomness.** As above, but choose a function that is not cryptographically secure but merely passes common statistical tests for things like k -wise independence of consecutive outputs. Which function to choose is largely a matter of convenience and fashion (although PRNGs in many standard libraries tend to be very, very bad). The advantage is that many PRNGs are very fast and will not slow your program down. The disadvantage is that you are relying on your program not doing anything that exposes the weakness of the PRNG.

The `random` function in the standard C library is an example of this, although maybe not a good example. The cool kids mostly use Mersenne Twister.

One advantage of pseudorandom generators is that they allow for debugging: if you run your program twice with the same key, it will do the same thing. This was in fact an argument made by von Neumann *against* using physical randomness back in the old days.

In practice, most people just use whatever PRNG is ready to hand, and hope it works. As theorists, we will ignore this issue, and assume that our random inputs are actually random.

1.5 Classifying randomized algorithms

Different random algorithms make different guarantees about the likelihood of getting a correct output or even the possibility of recognizing a correct output. These different guarantees have established names in the randomized algorithms literature and correspond to various complexity classes in complexity theory.

1.5.1 Las Vegas vs Monte Carlo

One difference between QuickSort and polynomial equality testing that QuickSort always succeeds, but may run for longer than you expect; while the polynomial equality tester always runs in a fixed amount of time, but may produce the wrong answer. These are examples of two classes of randomized algorithms, which were originally named by László Babai [Bab79]:⁸

- A **Las Vegas algorithm** fails with some probability, but we can tell when it fails. In particular, we can run it again until it succeeds, which means that we can eventually succeed with probability 1 (but with a potentially unbounded running time). Alternatively, we can think of a Las Vegas algorithm as an algorithm that runs for an unpredictable amount of time but always succeeds (we can convert such an algorithm back into one that runs in bounded time by declaring that it fails if it runs too long—a condition we can detect). QuickSort is an example of a Las Vegas algorithm.
- A **Monte Carlo algorithm** fails with some probability, but we can't tell when it fails. If the algorithm produces a yes/no answer and the failure probability is significantly less than 1/2, we can reduce the probability of failure by running it many times and taking a majority of the answers. The polynomial equality-testing algorithm is an example of a Monte Carlo algorithm.

The heuristic for remembering which class is which is that the names were chosen to appeal to English speakers: in Las Vegas, the dealer can tell you whether you've won or lost, but in Monte Carlo, *le croupier ne parle que Français*, so you have no idea what he's saying.

Generally, we prefer Las Vegas algorithms, because we like knowing when we have succeeded. But sometimes we have to settle for Monte Carlo algorithms, which can still be useful if we can get the probability of failure small enough. For example, any time we try to estimate an average by **sampling** (say, inputs to a function we are trying to integrate or political views of voters we are trying to win over) we are running a Monte Carlo algorithm: there is always some possibility that our sample is badly non-representative, but we can't tell if we got a bad sample unless we already know the answer we are looking for.

⁸To be more precise, Babai defined *Monte Carlo algorithms* based on the properties of **Monte Carlo simulation**, a technique dating back to the Manhattan project. The term *Las Vegas algorithm* was new.

1.5.2 Randomized complexity classes

Las Vegas vs Monte Carlo is the typical distinction made by algorithm designers, but complexity theorists have developed more elaborate classifications. These include algorithms with “one-sided” failure properties. For these algorithms, we never get a bogus “yes” answer but may get a bogus “no” answer (or vice versa). This gives us several complexity classes that act like randomized versions of **NP**, **co-NP**, etc.:

- The class **R** or **RP** (randomized **P**) consists of all languages L for which a polynomial-time Turing machine M exists such that if $x \in L$, then $\Pr[M(x, r) = 1] \geq 1/2$ and if $x \notin L$, then $\Pr[M(x, r) = 1] = 0$. In other words, we can find a witness that $x \in L$ with constant probability. This is the randomized analog of **NP** (but it’s much more practical, since with **NP** the probability of finding a winning witness may be exponentially small).
- The class **co-R** consists of all languages L for which a poly-time Turing machine M exists such that if $x \notin L$, then $\Pr[M(x, r) = 1] \geq 1/2$ and if $x \in L$, then $\Pr[M(x, r) = 1] = 0$. This is the randomized analog of **co-NP**.
- The class **ZPP** (zero-error probabilistic P) is defined as **RP** \cap **co-RP**. If we run both our **RP** and **co-RP** machines for polynomial time, we learn the correct classification of x with probability at least $1/2$. The rest of the time we learn only that we’ve failed (because both machines return 0, telling us nothing). This is the class of (polynomial-time) Las Vegas algorithms. The reason it is called “zero-error” is that we can equivalently define it as the problems solvable by machines that always output the correct answer eventually, but only run in *expected* polynomial time.
- The class **BPP** (bounded-error probabilistic **P**) consists of all languages L for which a poly-time Turing machine exists such that if $x \notin L$, then $\Pr[M(x, r) = 1] \leq 1/3$, and if $x \in L$, then $\Pr[M(x, r) = 1] \geq 2/3$. These are the (polynomial-time) Monte Carlo algorithms: if our machine answers 0 or 1, we can guess whether $x \in L$ or not, but we can’t be sure.
- The class **PP** (probabilistic **P**) consists of all languages L for which a poly-time Turing machine exists such that if $x \notin L$, then $\Pr[M(x, r) = 1] \geq 1/2$, and if $x \in L$, then $\Pr[M(x, r) = 1] < 1/2$. Since there is only an

exponentially small gap between the two probabilities, such algorithms are not really useful in practice; **PP** is mostly of interest to complexity theorists.

Assuming we have a source of random bits, any algorithm in **RP**, **co-RP**, **ZPP**, or **BPP** is good enough for practical use. We can usually even get away with using a pseudorandom number generator, and there are good reasons to suspect that in fact every one of these classes is equal to **P**.

1.6 Classifying randomized algorithms by their methods

We can also classify randomized algorithms by how they use their randomness to solve a problem. Some very broad categories:⁹

- **Avoiding worst-case inputs**, by hiding the details of the algorithm from the adversary. Typically we assume that an adversary supplies our input. If the adversary can see what our algorithm is going to do (for example, he knows which door we will open first), he can use this information against us. By using randomness, we can replace our predictable deterministic algorithm by what is effectively a random choice of many different deterministic algorithms. Since the adversary doesn't know which algorithm we are using, he can't (we hope) pick an input that is bad for all of them.
- **Sampling**. Here we use randomness to find an example or examples of objects that are likely to be typical of the population they are drawn from, either to estimate some average value (pretty much the basis of all of statistics) or because a typical element is useful in our algorithm (for example, when picking the pivot in QuickSort). Randomization means that the adversary can't direct us to non-representative samples.
- **Hashing**. Hashing is the process of assigning a large object x a small name $h(x)$ by feeding it to a **hash function** h . Because the names are small, the Pigeonhole Principle implies that many large objects hash to the same name (a **collision**). If we have few objects that we actually care about, we can avoid collisions by choosing a hash function that happens to map them to different places. Randomization helps here

⁹These are largely adapted from the introduction to [MR95].

by keeping the adversary from choosing the objects after seeing what our hash function is.

Hashing techniques are used both in **load balancing** (e.g., insuring that most cells in a **hash table** hold only a few objects) and in **fingerprinting** (e.g., using a **cryptographic hash function** to record a **fingerprint** of a file, so that we can detect when it has been modified).

- **Building random structures.** The **probabilistic method** shows the existence of structures with some desired property (often graphs with interesting properties, but there are other places where it can be used) by showing that a randomly-generated structure in some class has a nonzero probability of having the property we want. If we can beef the probability up to something substantial, we get a randomized algorithm for generating these structures.
- **Symmetry breaking.** In **distributed algorithms** involving multiple processes, progress may be stymied by all the processes trying to do the same thing at the same time (this is an obstacle, for example, in **leader election**, where we want only one process to declare itself the leader). Randomization can break these deadlocks.

Chapter 2

Probability theory

Last updated 2019. Some material may be out of date.

In this chapter, we summarize the parts of **probability theory** that we need for the course. This is not really a substitute for reading an actual probability theory book like Feller [Fel68] or Grimmett and Stirzaker [GS01], but the hope is that it's enough to get by.

The basic idea of probability theory is that we want to model all possible outcomes of whatever process we are studying simultaneously. This gives the notion of a **probability space**, which is the set of all possible outcomes; for example, if we roll two dice, the probability space would consist of all 36 possible combinations of values. Subsets of this space are called **events**; an example in the two-dice space would be the event that the sum of the two dice is 11, given by the set $A = \{\langle 5, 6 \rangle, \langle 6, 5 \rangle\}$. The probability of an event A is given by a **probability measure** $\Pr[A]$; for simple probability spaces, this is just the sum of the probabilities of the individual outcomes contained in A , while for more general spaces, we define the measure on events first and the probabilities of individual outcomes are derived from this. Formal definitions of all of these concepts are given later in this chapter.

When analyzing a randomized algorithm, the probability space describes all choices of the random bits used by the algorithm, and we can think of the possible executions of an algorithm as living within this probability space. More formally, the sequence of operations carried out by an algorithm and the output it ultimately produces are examples of **random variables**—functions from a probability space to some other set—which we will discuss in detail in Chapter 3.

2.1 Probability spaces and events

A **discrete probability space** is a countable set Ω of **points** or **outcomes** ω . Each ω in Ω has a **probability** $\Pr[\omega]$, which is a real value with $0 \leq \Pr[\omega] \leq 1$. It is required that $\sum_{\omega \in \Omega} \Pr[\omega] = 1$.

An **event** A is a subset of Ω ; its probability is $\Pr[A] = \sum_{\omega \in A} \Pr[\omega]$. We require that $\Pr[\Omega] = 1$, and it is immediate from the definition that $\Pr[\emptyset] = 0$.

The **complement** \bar{A} or $\neg A$ of an event A is the event $\Omega - A$. It is always the case that $\Pr[\neg A] = 1 - \Pr[A]$.

This fact is a special case of the general principle that if A_1, A_2, \dots forms a **partition** of Ω —that is, if $A_i \cap A_j = \emptyset$ when $i \neq j$ and $\bigcup A_i = \Omega$ —then $\sum \Pr[A_i] = \Pr[\Omega] = 1$. It happens that $\neg A$ and A form a partition of Ω consisting of exactly two elements.

Even more generally, whenever A_1, A_2, \dots are disjoint events (that is, when $A_i \cap A_j = \emptyset$ for all $i \neq j$), $\Pr[\bigcup A_i] = \sum \Pr[A_i]$. This fact does not hold in general for events that are not disjoint.

For discrete probability spaces, all of these facts can be proven directly from the definition of probabilities for events. For more general probability spaces, it's no longer possible to express the probability of an event as the sum of the probabilities of its elements, and we adopt an axiomatic approach instead.

2.1.1 General probability spaces

More general probability spaces consist of a triple $(\Omega, \mathcal{F}, \Pr)$ where Ω is a set of points, \mathcal{F} is a **σ -algebra** (a family of subsets of Ω that contains Ω and is closed under complement and countable unions) of **measurable sets**, and \Pr is a function from \mathcal{F} to $[0, 1]$ that gives $\Pr[\Omega] = 1$ and satisfies **countable additivity**: when A_1, \dots are disjoint, $\Pr[\bigcup A_i] = \sum \Pr[A_i]$. This definition is needed for uncountable spaces, because (under certain set-theoretic assumptions) we may not be able to assign a meaningful probability to all subsets of Ω .

Formally, this definition is often presented as three **axioms of probability**, due to Kolmogorov [Kol33]:

1. $\Pr[A] \geq 0$ for all $A \in \mathcal{F}$.
2. $\Pr[\Omega] = 1$.

3. For any countable collection of disjoint events A_1, A_2, \dots ,

$$\Pr \left[\bigcup_i A_i \right] = \sum_i \Pr [A_i].$$

It's not hard to see that the discrete probability spaces defined in the preceding section satisfy these axioms.

General probability spaces arise in randomized algorithms when we have an algorithm that might consume an unbounded number of random bits. The problem now is that an outcome consists of countable sequence of bits, and there are uncountably many such outcomes. The solution is to consider as measurable events only those sets with the property that membership in them can be determined after a finite amount of time. Formally, the probability space Ω is the set $\{0, 1\}^{\mathbb{N}}$ of all countably infinite sequences of 0 and 1 values indexed by the natural numbers, and the measurable sets \mathcal{F} are all sets that can be generated by countable unions¹ of **cylinder sets**, where a cylinder set consists of all extensions xy of some finite prefix x . The probability measure itself is obtained by assigning the set of all points that start with x the probability $2^{-|x|}$, and computing the probabilities of other sets from the axioms.²

An oddity that arises in general probability spaces is it may be that every particular outcome has probability zero but their union has probability 1. For example, the probability of any particular infinite string of bits is 0, but the set containing all such strings is the entire space and has probability 1. This is where the fact that probabilities only add over *countable* unions comes in.

Most randomized algorithms books gloss over general probability spaces, with three good reasons. The first is that if we truncate an algorithm after a finite number of steps, we are usually get back to a discrete probability space, which avoids a lot of worrying about measurability and convergence. The second is that we are often implicitly working in a probability space that is either discrete or well-understood (like the space of bit-vectors described above). The last is that the **Kolmogorov extension theorem** says that

¹As well as complements and countable intersections. However, it is not hard to show that that sets defined using these operations can be reduced to countable unions of cylinder sets.

²This turns out to give the same probabilities as if we consider each outcome as a real number in the interval $[0, 1]$ and use Lebesgue measure to compute the probability of events. For some applications, thinking of our random values as real numbers (or even sequences of real numbers) can make things easier: consider for example what happens when we want to choose one of three outcomes with equal probability.

if we specify $\Pr[A_1 \cap A_2 \cap \dots \cap A_k]$ consistently for all finite sets of events $\{A_1 \dots A_k\}$, then there exists some probability space that makes these probabilities work, even if we have uncountably many such events. So it's usually enough to specify how the events we care about interact, without worrying about the details of the underlying space.

2.2 Boolean combinations of events

Even though events are defined as sets, we often think of them as representing propositions that we can combine using the usual Boolean operations of NOT (\neg), AND (\wedge), and OR (\vee). In terms of sets, these correspond to taking a complement $\bar{A} = \Omega \setminus A$, an intersection $A \cap B$, or a union $A \cup B$.

We can use the axioms of probability to calculate the probability of \bar{A} :

Lemma 2.2.1.

$$\Pr[\bar{A}] = 1 - \Pr[A].$$

Proof. First note that $A \cap \bar{A} = \emptyset$, so $A \cup \bar{A} = \Omega$ is a disjoint union of countably many³ events. This gives $\Pr[A] + \Pr[\bar{A}] = \Pr[\Omega] = 1$. \square

For example, if our probability space consists of the six outcomes of a fair die roll, and $A = [\text{outcome is 3}]$ with $\Pr[A] = 1/6$, then $\Pr[\text{outcome is not 3}] = \Pr[\bar{A}] = 1 - 1/6 = 5/6$. Though this example is trivial, using the formula does save us from having to add up the five cases where we don't get 3.

If we want to know the probability of $A \cap B$, we need to know more about the relationship between A and B . For example, it could be that A and B are both events representing a fair coin coming up heads, with $\Pr[A] = \Pr[B] = 1/2$. The probability of $A \cap B$ could be anywhere between $1/2$ and 0:

- For ordinary fair coins, we'd expect that half the time that A happens, B also happens. This gives $\Pr[A \cap B] = (1/2) \cdot (1/2) = 1/4$. To make this formal, we might define our probability space Ω as having four outcomes HH, HT, TH, and TT, each of which occurs with equal probability.
- But maybe A and B represent the same fair coin: then $A \cap B = A$ and $\Pr[A \cap B] = \Pr[A] = 1/2$.

³Countable need not be infinite, so 2 is countable.

- At the other extreme, maybe A and B represent two fair coins welded together so that if one comes up heads the other comes up tails. Now $\Pr[A \cap B] = 0$.
- With a little bit of tinkering, we could also find probabilities for the outcomes in our four-outcome space to make $\Pr[A] = \Pr[\text{HH}] + \Pr[\text{HT}] = 1/2$ and $\Pr[B] = \Pr[\text{HH}] + \Pr[\text{TH}] = 1/2$ while setting $\Pr[A \cap B] = \Pr[\text{HH}]$ to any value between 0 and 1/2.

The difference between the nice case where $\Pr[A \cap B] = 1/4$ and the other, more annoying cases where it doesn't is that in the first case we have assumed that A and B are **independent**, which is *defined* to mean that $\Pr[A \cap B] = \Pr[A] \Pr[B]$.

In the real world, we expect events to be independent if they refer to parts of the universe that are not causally related: if we flip two coins that aren't glued together somehow, then we assume that the outcomes of the coins are independent. But we can also get independence from events that are not causally disconnected in this way. An example would be if we rolled a fair four-sided die labeled $\text{HH}, \text{HT}, \text{TH}, \text{TT}$, where we take the first letter as representing A and the second as B .

There's no simple formula for $\Pr[A \cup B]$ when A and B are not disjoint, even for independent events, but we can compute the probability by splitting up into smaller, disjoint events and using countable additivity:

$$\begin{aligned}
 \Pr[A \cup B] &= \Pr[(A \cap B) \cup (A \cap \bar{B}) \cup (\bar{A} \cap B)] \\
 &= \Pr[A \cap B] + \Pr[A \cap \bar{B}] + \Pr[\bar{A} \cap B] \\
 &= (\Pr[A \cap B] + \Pr[A \cap \bar{B}]) + (\Pr[\bar{A} \cap B] + \Pr[A \cap B]) - \Pr[A \cap B] \\
 &= \Pr[A] + \Pr[B] - \Pr[A \cap B].
 \end{aligned}$$

The idea is that we can compute $\Pr[A \cup B]$ by adding up the individual probabilities and then subtracting off the part where the counted the event twice.

This is a special case of the general **inclusion-exclusion formula**, which says:

Lemma 2.2.2. For any finite sequence of events $A_1 \dots A_n$,

$$\begin{aligned} \Pr \left[\bigcup_{i=1}^n A_i \right] &= \sum_i \Pr [A_i] - \sum_{i<j} \Pr [A_i \cap A_j] + \sum_{i<j<k} \Pr [A_i \cap A_j \cap A_k] - \dots \\ &= \sum_{S \subseteq \{1 \dots n\}, S \neq \emptyset} (-1)^{|S|+1} \Pr \left[\bigcap_{i \in S} A_i \right]. \end{aligned} \quad (2.2.1)$$

Proof. Partition Ω into 2^n disjoint events B_T , where $B_T = (\bigcap_{i \in T} A_i) \cap (\bigcap_{i \notin T} \bar{A}_i)$ is the event that all A_i occur for i in T and no A_i occurs for i not in T . Then A_i is the union of all B_T with $T \ni i$ and $\bigcup A_i$ is the union of all B_T with $T \neq \emptyset$.

That the right-hand side gives the probability of this event is a sneaky consequence of the binomial theorem, and in particular the fact that $\sum_{i=1}^n (-1)^i = \sum_{i=0}^n (-1)^i - 1 = (1-1)^n - 1$ is -1 if $n > 0$ and 0 if $n = 0$. Using this fact after rewriting the right-hand side using the B_T events gives

$$\begin{aligned} \sum_{S \subseteq \{1 \dots n\}, S \neq \emptyset} (-1)^{|S|+1} \Pr \left[\bigcap_{i \in S} A_i \right] &= \sum_{S \subseteq \{1 \dots n\}, S \neq \emptyset} (-1)^{|S|+1} \sum_{T \supseteq S} \Pr [B_T] \\ &= \sum_{T \subseteq \{1 \dots n\}} \left(\Pr [B_T] \sum_{S \subseteq T, S \neq \emptyset} (-1)^{|S|+1} \right) \\ &= \sum_{T \subseteq \{1 \dots n\}} \left(-\Pr [B_T] \sum_{i=1}^n (-1)^i \binom{|T|}{i} \right) \\ &= \sum_{T \subseteq \{1 \dots n\}} \left(-\Pr [B_T] ((1-1)^{|T|} - 1) \right) \\ &= \sum_{T \subseteq \{1 \dots n\}} \Pr [B_T] ((1-0)^{|T|}) \\ &= \sum_{T \subseteq \{1 \dots n\}, T \neq \emptyset} \Pr [B_T] \\ &= \Pr \left[\bigcup_{i=1}^n A_i \right]. \end{aligned}$$

□

2.3 Conditional probability

The **probability of A conditioned on B** or **probability of A given B** , written $\Pr[A \mid B]$, is defined by

$$\Pr[A \mid B] = \frac{\Pr[A \cap B]}{\Pr[B]}, \quad (2.3.1)$$

provided $\Pr[B] \neq 0$. If $\Pr[B] = 0$, we can't condition on B .

Such conditional probabilities represent the effect of restricting our probability space to just B , which can think of as computing the probability of each event if we know that B occurs. The intersection in the numerator limits A to circumstances where B occurs, while the denominator normalizes the probabilities so that, for example, $\Pr[\Omega \mid B] = \Pr[B \mid B] = 1$.

2.3.1 Conditional probability and independence

Rearranging (2.3.1) gives $\Pr[A \cap B] = \Pr[B] \Pr[A \mid B] = \Pr[A] \Pr[B \mid A]$. In many cases, knowing that B occurs tells us nothing about whether A occurs; if so, we have $\Pr[A \mid B] = \Pr[A]$, which implies that $\Pr[A \cap B] = \Pr[A] \Pr[B] = \Pr[A] \Pr[B]$ —events A and B are **independent**. So $\Pr[A \mid B] = \Pr[A]$ gives an alternative criterion for independence when $\Pr[B]$ is nonzero.⁴

A *set* of events A_1, A_2, \dots is independent if A_i is independent of B when B is any Boolean formula of the A_j for $j \neq i$. The idea is that you can't predict A_i by knowing anything about the rest of the events.

A set of events A_1, A_2, \dots is **pairwise independent** if each A_i and A_j , $i \neq j$ are independent. It is possible for a set of events to be pairwise independent but not independent; a simple example is when A_1 and A_2 are the events that two independent coins come up heads and A_3 is the event that both coins come up with the same value. The general version of pairwise independence is **k -wise independence**, which means that any subset of k (or fewer) events are independent.

2.3.2 Conditional probability and the law of total probability

The reason we like conditional probability in algorithm analysis is that it gives us a natural way to model the kind of case analysis that we are used to applying to deterministic algorithms. Suppose we are trying to prove that a randomized algorithm works (event A) with a certain probability. Most

⁴If $\Pr[B]$ is zero, then A and B are always independent.

likely, the first random thing the algorithm does is flip a coin, giving two possible outcomes B and \bar{B} . Countable additivity tells us that $\Pr[A] = \Pr[A \cap B] + \Pr[A \cap \bar{B}]$, which we can rewrite using conditional probability as

$$\Pr[A] = \Pr[A | B] \Pr[B] + \Pr[A | \bar{B}] \Pr[\bar{B}], \quad (2.3.2)$$

a special case of the **law of total probability**.

What's nice about this expression is that we can often compute $\Pr[A | B]$ and $\Pr[A | \bar{B}]$ by looking at what the algorithm does starting from the point where it has just gotten heads (B) or tails (\bar{B}), and use the formula to combine these values to get the overall probability of success.

For example, if

$$\Pr[\text{class occurs} | \text{snow}] = 3/5,$$

$$\Pr[\text{class occurs} | \text{no snow}] = 99/100, \text{ and}$$

$$\Pr[\text{snow}] = 1/10,$$

then

$$\Pr[\text{class occurs}] = (3/5) \cdot (1/10) + (99/100) \cdot (1 - 1/10) = 0.951.$$

More generally, we can do the same computation for any partition of Ω into countably many disjoint events B_i :

$$\begin{aligned} \Pr[A] &= \Pr\left[\bigcup_i (A \cap B_i)\right] \\ &= \sum_i \Pr[A \cap B_i] \\ &= \sum_{i, \Pr[B_i] \neq 0} \Pr[A | B_i] \Pr[B_i], \end{aligned} \quad (2.3.3)$$

which is the **law of total probability**. Note that the last step works for each term only if $\Pr[A | B_i]$ is well-defined, meaning that $\Pr[B_i] \neq 0$. But any such case contributes nothing to the previous sum, so we get the correct answer if we simply omit any terms from the sum for which $\Pr[B_i] = 0$.

A special case arises when $\Pr[A | \bar{B}] = 0$, which occurs, for example, if $A \subseteq B$. Then we just have $\Pr[A] = \Pr[A | B] \Pr[B]$. If we consider an

event $A = A_1 \cap A_2 \cap \dots \cap A_k$, then we can iterate this expansion to get

$$\begin{aligned}
 \Pr[A_1 \cap A_2 \cap \dots \cap A_k] &= \Pr[A_1 \cap \dots \cap A_{k-1}] \Pr[A_k \mid A_1, \dots, A_{k-1}] \\
 &= \Pr[A_1 \cap \dots \cap A_{k-2}] \Pr[A_{k-1} \mid A_1, \dots, A_{k-2}] \Pr[A_k \mid A_1, \dots, A_{k-1}] \\
 &= \dots \\
 &= \prod_{i=1}^k \Pr[A_i \mid A_1, \dots, A_i].
 \end{aligned} \tag{2.3.4}$$

Here $\Pr[A \mid B, C, \dots]$ is short-hand for $\Pr[B \cap C \cap \dots]$, the probability that A occurs given that all of B, C , etc., occur.

2.3.3 Examples

Here we have some examples of applying conditional probability to algorithm analysis. Mostly we will be using some form of the law of total probability.

2.3.3.1 Racing coin-flips

Suppose that I flip coins and allocate a space for each heads that I get before the coin comes up tails. Suppose that you then supply me with objects (each of which takes up one unit of space), one for each heads that *you* get before you get tails. What are my chances of allocating enough space?

Let's start by solving this directly using the law of total probability. Let A_i be the event that I allocate i spaces. The event A_i is the intersection of i independent events that I get heads in the first i positions and the event that I get tails in position $i + 1$; this multiplies out to $(1/2)^{i+1}$. Let B_i be the similar event that you supply i objects. Let W be the event that I win. To make the A_i partition the space, we must also add an extra event A_∞ equal to the singleton set $\{\text{HHHHHHH} \dots\}$ consisting of the all-H sequence; this has probability 0 (so it won't have much of an effect), but we need to include it since $\text{HHHHHHH} \dots$ is not contained in any of the other A_i .

We can compute

$$\begin{aligned}
 \Pr[W \mid A_i] &= \Pr[B_0 \cap B_1 \cap \cdots \cap B_i \mid A_i] \\
 &= \Pr[B_0 \cap B_1 \cap \cdots \cap B_i] \\
 &= \Pr[B_0] + \Pr[B_1] + \cdots + \Pr[B_i] \\
 &= \sum_{j=0}^i (1/2)^j \\
 &= (1/2) \cdot \frac{1 - (1/2)^{i+1}}{1 - 1/2} \\
 &= 1 - (1/2)^{i+1}. \tag{2.3.5}
 \end{aligned}$$

The clean form of this expression suggests strongly that there is a better way to get it, and that this way involves taking the negation of the intersection of $i + 1$ independent events that occur with probability $1/2$ each. With a little reflection, we can see that the probability that your objects *don't* fit in my buffer is exactly $(1/2)^{i+1}$

From the law of total probability (2.3.3),

$$\begin{aligned}
 \Pr[W] &= \sum_{i=0}^{\infty} (1 - (1/2)^{i+1})(1/2)^{i+1} \\
 &= 1 - \sum_{i=0}^{\infty} (1/4)^{i+1} \\
 &= 1 - \frac{1}{4} \cdot 11 - 1/4 \\
 &= 2/3.
 \end{aligned}$$

This gives us our answer. However, we again see an answer that is suspiciously simple, which suggests looking for another way to find it. We can do this using conditional probability by defining new events C_i , where C_i contains all sequences of coin-flips for both players where get i heads in a row but at least one gets tails on the $(i + 1)$ -th coin. These events plus the probability-zero event $C_{\infty} = \{\text{HHHHHHHH} \dots, \text{TTTTTTT} \dots\}$ partition the space, so $\Pr[W] = \sum_{i=0}^{\infty} \Pr[W \mid C_i] \Pr[C_i]$.

Now we ask, what is $\Pr[W \mid C_i]$? Here we only need to consider three cases, depending on the outcomes of our $(i + 1)$ -th coin-flips. The cases $\langle \text{H}, \text{T} \rangle$ and $\langle \text{T}, \text{T} \rangle$ cause me to win, while the case $\langle \text{T}, \text{H} \rangle$ causes me to lose, and each case occurs with equal probability conditioned on C_i (which excludes $\langle \text{H}, \text{H} \rangle$). So I win $2/3$ of the time conditioned on C_i , and summing

$\Pr[W] = \sum_{i=0}^{\infty} (2/3) \Pr[C_i] = 2/3$ since $\Pr[C_i]$ sums to 1, because the union of these events includes the entire space except for the probability-zero event C_{∞} .

Still another approach is to compute the probability that our runs have exactly the same length ($\sum_{i=1}^{\infty} 2^{-i} \cdot 2^{-i} = 1/3$), and argue by symmetry that the remaining $2/3$ probability is equally split between my run being longer ($1/3$) and your run being longer ($1/3$). Since W occurs if my run is just as long or longer, $\Pr[W] = 1/3 + 1/3 = 2/3$. A nice property of this approach is that the only summation involved is over disjoint events, so we get to avoid using conditional probability entirely.

2.3.3.2 Karger's min-cut algorithm

Here we'll give a simple algorithm for finding a global **min-cut** in a **multigraph**,⁵ due to David Karger [Kar93].

The idea is that we are given a multigraph G , and we want to partition the vertices into nonempty sets S and T such that the number of edges with one endpoint in S and one endpoint in T is as small as possible. There are many efficient ways to do this, most of which are quite sophisticated. There is also the algorithm we will now present, which solves the problem with reasonable efficiency using almost no sophistication at all (at least in the algorithm itself).

The main idea is that given an edge uv , we can construct a new multigraph G_1 by **contracting** the edge: in G_1 , u and v are replaced by a single vertex, and any edge that used to have either vertex as an endpoint now goes to the combined vertex (edges with both endpoints in $\{u, v\}$ are deleted). Karger's algorithm is to contract edges chosen uniformly at random until only two vertices remain. All the vertices that got packed into one of these become S , the others become T . It turns out that this finds a minimum cut with probability at least $1/\binom{n}{2}$.

An example of the algorithm in action is given in Figure 2.1.

Theorem 2.3.1. *Given any min cut (S, T) of a graph G on n vertices, Karger's algorithm outputs (S, T) with probability at least $1/\binom{n}{2}$.*

Proof. Let (S, T) be a min cut of size k . Then the degree of each vertex v is at least k (otherwise $(v, G - v)$ would be a smaller cut), and G contains at least $kn/2$ edges. The probability that we contract an S - T edge is thus at most $k/(kn/2) = 2/n$, and the probability that we don't contract one is

⁵Unlike ordinary graphs, multigraphs can have more than one edge between two vertices.

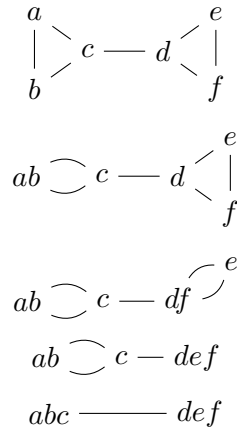


Figure 2.1: Karger’s min-cut algorithm. Initial graph (at top) has min cut $\langle \{a, b, c\}, \{d, e, f\} \rangle$. We find this cut by getting lucky and contracting edges ab , df , de , and ac in that order. The final graph (at bottom) gives the cut.

$1 - 2/n = (n - 2)/n$. Assuming we missed collapsing (S, T) the first time, we now have a new graph G_1 with $n - 1$ vertices in which the min cut is still of size k . So now the chance that we miss (S, T) is $(n - 3)/(n - 1)$. We stop when we have two vertices left, so the last step succeeds with probability $1/3$.

We can compute the probability that the S - T cut is never contracted by applying (2.3.4), which just tells us to multiply all the conditional probabilities together:

$$\prod_{i=3}^n \frac{i - 2}{i} = \frac{2}{n(n - 1)}.$$

□

This tells us what happens when we are considering a particular min cut. If the graph has more than one min cut, this only makes our life easier. Note that since each min cut turns up with probability at least $1/\binom{n}{2}$, there can’t be more than $\binom{n}{2}$ of them.⁶ But even if there is only one, we have a good

⁶The suspiciously combinatorial appearance of the $1/\binom{n}{2}$ suggests that there should be some way of associating minimum cuts with particular pairs of vertices, but I’m not aware of any natural way to do this. It may be that sometimes the appearance of a simple expression in a surprising context may just stem from the fact that there aren’t very many distinct simple expressions.

chance of finding it if we simply re-run the algorithm substantially more than n^2 times.

Chapter 3

Random variables

Last updated 2019. Some material may be out of date.

A **random variable** on a probability space Ω is a function with domain Ω .¹ Rather than writing a random variable as $f(\omega)$ everywhere, the convention is to write a random variable as a capital letter (X , Y , S , etc.) and make the argument implicit: X is really $X(\omega)$. Variables that aren't random (or aren't variable) are written in lowercase.

Most of the random variables we will consider will be **discrete random variables**. A discrete random variable takes on only countably many values.

For example, consider the probability space corresponding to rolling two independent fair six-sided dice. There are 36 possible outcomes in this space, corresponding to the 6×6 pairs of values $\langle x, y \rangle$ we might see on the two dice. We could represent the value of each die as a random variable X or Y given by $X(\langle x, y \rangle) = x$ or $Y(\langle x, y \rangle) = y$, but for many applications, we don't care so much about the specific values on each die. Instead, we want to know the sum $S = X + Y$ of the dice. This value S is also random variable; as a function on Ω , it's defined by $S(\langle x, y \rangle) = x + y$.

Random variables need not be real-valued. There's no reason why we can't think of the pair $\langle x, y \rangle$ itself a random variable, whose range is the set $[1 \dots 6] \times [1 \dots 6]$. Similarly, if we imagine choosing a point uniformly at random in the unit square $[0, 1]^2$, its coordinates are a random variable. For a more exotic example, the **random graph** $G_{n,p}$ obtained by starting with

¹Technically, this only works for discrete spaces. In general, a random variable is a **measurable function** from a probability space (Ω, \mathcal{F}) to some other set S equipped with its own σ -algebra \mathcal{F}' . What makes a function measurable in this sense is that that for any set A in \mathcal{F}' , the inverse image $f^{-1}(A)$ must be in \mathcal{F} . See §3.3 for more details.

n vertices and including each possible edge with independent probability p is a random variable whose range is the set of all graphs on n vertices.

3.1 Operations on random variables

Random variables may be combined using standard arithmetic operators, have functions applied to them, etc., to get new random variables. For example, the random variable X/Y is a function from Ω that takes on the value $X(\omega)/Y(\omega)$ on each point ω .

3.2 Random variables and events

Any random variable X allows us to define events based on its possible values. Typically these are expressed by writing a predicate involving the random variable in square brackets. An example would be the probability that the sum of two dice is exactly 11: $[S = 11]$; or that the sum of the dice is less than 5: $[S < 5]$. These are both sets of outcomes; we could expand $[S = 11] = \{\langle 5, 6 \rangle, \langle 6, 5 \rangle\}$ or $[S < 5] = \{\langle 1, 1 \rangle, \langle 1, 2 \rangle, \langle 1, 3 \rangle, \langle 2, 1 \rangle, \langle 2, 2 \rangle, \langle 3, 1 \rangle\}$. This allows us to calculate the probability that a random variable has particular properties: $\Pr[S = 11] = \frac{2}{36} = \frac{1}{18}$ and $\Pr[S < 5] = \frac{6}{36} = \frac{1}{6}$.

Conversely, given any event A , we can define an **indicator random variable** 1_A that is 1 when A occurs and 0 when it doesn't.² Formally, $1_A(\omega) = 1$ for ω in A and $1_A(\omega) = 0$ for ω not in A .

Indicator variables are mostly useful when combined with other random variables. For example, if you roll two dice and normally collect the sum of the values but get nothing if it is 7, we could write your payoff as $S \cdot 1_{[S \neq 7]}$.

The **probability mass function** of a random variable gives $\Pr[X = x]$ for each possible value x . For example, our random variable S has the probability mass function show in Table 3.1. For a discrete random variable X , the probability mass function gives enough information to calculate the probability of any event involving X , since we can just sum up cases using countable additivity. This gives us another way to compute $\Pr[S < 5] = \Pr[S = 2] + \Pr[S = 3] + \Pr[S = 4] = \frac{1+2+3}{36} = \frac{1}{6}$.

²Some people like writing χ_A for these.

You may also see $[P]$ where P is some predicate, a convention known as **Iverson notation** or the **Iverson bracket** that was invented by Iverson for the programming language APL, appears in later languages like *C* where the convention is that true predicates evaluate to 1 and false ones to 0, and ultimately popularized for use in mathematics—with the specific choice of square brackets to set off the predicate—by Graham *et al.* [GKP88].

S	Probability
2	$1/36$
3	$2/36$
4	$3/36$
5	$4/36$
6	$5/36$
7	$6/36$
8	$5/36$
9	$4/36$
10	$3/36$
11	$2/36$
12	$1/36$

Table 3.1: Probability mass function for the sum of two independent fair six-sided dice

For two random variables, the **joint probability mass function** gives $\Pr[X = x \wedge Y = y]$ for each pair of values x and y (this generalizes in the obvious way for more than two variables).

We will often refer to the probability mass function as giving the **distribution** or **joint distribution** of a random variable or collection of random variables, even though distribution (for real-valued variables) technically refers to the **cumulative distribution function** $F(x) = \Pr[X \leq x]$, which is generally not directly computable from the probability mass function for **continuous random variables** that take on uncountably many values. To the extent that we can, we will try to avoid continuous random variables, and the rather messy integration theory needed to handle them.

Two or more random variables are **independent** if all sets of events involving different random variables are independent. In terms of probability mass functions, X and Y are independent if $\Pr[X = x \wedge Y = y] = \Pr[X = x] \cdot \Pr[Y = y]$. In terms of cumulative distribution functions, X and Y are independent if $\Pr[X = x \wedge Y = y] = \Pr[X = x] \cdot \Pr[Y = y]$. As with events, we generally assume that random variables associated with causally disconnected processes are independent, but this is not the only way we might have independence.

It's not hard to see that the individual die values X and Y in our two-dice example are independent, because every possible combination of values x and y has the same probability $1/36 = \Pr[X = x] \Pr[Y = y]$. If we chose a different probability distribution on the space, we might not have independence.

3.3 Measurability

For discrete probability spaces, any function on outcomes can be a random variable. The reason is that any event in a discrete probability space has a well-defined probability. For more general spaces, in order to be useful, events involving a random variable should have well-defined probabilities. For **discrete random variables** that take on only countably many values (e.g., integers or rationals), it's enough for the event $[X = x]$ (that is, the set $\{\omega \mid X(\omega) = x\}$) to be in \mathcal{F} for all x . For real-valued random variables, we ask that the event $[X \leq x]$ be in \mathcal{F} . In these cases, we say that X is **measurable** with respect to \mathcal{F} , or just **measurable** \mathcal{F} . More exotic random variables use a definition of measurability that generalizes the real-valued

Out of these alternatives, I personally find 1_A to be the least confusing.

version, which we probably won't need.³ Since we usually just assume that all of our random variables are measurable unless we are doing something funny with \mathcal{F} to represent ignorance, this issue won't come up much.

3.4 Expectation

The **expectation** or **expected value** of a random variable X is given by $E[X] = \sum_x x \Pr[X = x]$. This is essentially an average value of X weighted by probability, and it only makes sense if X takes on values that can be summed in this way (e.g., real or complex values, or vectors in a real- or complex-valued vector space). Even if the expectation makes sense, it may be that a particular random variable X doesn't have an expectation, because the sum fails to converge.⁴

For an example that does work, if X and Y are independent fair six-sided dice, then $E[X] = E[Y] = \sum_{i=1}^6 i \left(\frac{1}{6}\right) = \frac{21}{6} = \frac{7}{2}$, while $E[X + Y]$ is the rather horrific

$$\begin{aligned} \sum_{i=2}^{12} i \Pr[X + Y = i] &= \frac{2 \cdot 1 + 3 \cdot 2 + 4 \cdot 3 + 5 \cdot 4 + 6 \cdot 5 + 7 \cdot 6 + 8 \cdot 5 + 9 \cdot 4 + 10 \cdot 3 + 11 \cdot 2 + 12 \cdot 1}{36} \\ &= \frac{252}{36} = 7. \end{aligned}$$

The fact that $7 = \frac{7}{2} + \frac{7}{2}$ here is not a coincidence.

3.4.1 Linearity of expectation

The main reason we like expressing the run times of algorithms in terms of expectation is **linearity of expectation**: $E[aX + bY] = E[aX] + E[bY]$ for all random variables X and Y for which $E[X]$ and $E[Y]$ are defined, and all constants a and b . This means that we can compute the running time for different parts of our algorithm separately and then add them together, *even if the costs of different parts of the algorithm are not independent*.

³The general version is that if X takes on values on another measure space (Ω', \mathcal{F}') , then the inverse image $X^{-1}(A) = \{\omega \in \Omega \mid X(\omega) \in A\}$ of any set A in \mathcal{F}' is in \mathcal{F} . This means in particular that \Pr_Ω maps through X to give a probability measure on Ω' by $\Pr_{\Omega'}[A] = \Pr_\Omega[X^{-1}(A)]$, and the condition on $X^{-1}(A)$ being in \mathcal{F} makes this work.

⁴Example: Let X be the number of times you flip a fair coin until it comes up heads. We'll see later that $E[X] = 1/(1/2) = 2$. But $E[2^X] = \sum_{n=1}^{\infty} 2^n 2^{-n} = \sum_{n=1}^{\infty} 1$, which diverges. With some tinkering it is possible to come up with even uglier cases, like an array that contains 1 element on average but requires infinite expected time to sort using a $\Theta(n \log n)$ algorithm.

The general version is $E[\sum a_i X_i] = \sum a_i E[X_i]$ for any *finite* collection of random variables X_i and constants a_i , which follows by applying induction to the two-variable case. A special case is $E[cX] = cE[X]$ when c is constant.

For discrete random variables, linearity of expectation follows immediately from the definition of expectation and the fact that the event $[X = x]$ is the disjoint union of the events $[X = x, Y = y]$ for all y :

$$\begin{aligned} E[aX + bY] &= \sum_{x,y} (ax + by) \Pr[X = x \wedge Y = y] \\ &= a \sum_{x,y} x \Pr[X = x, Y = y] + b \sum_{x,y} y \Pr[X = x, Y = y] \\ &= a \sum_x x \sum_y \Pr[X = x, Y = y] + b \sum_y y \sum_x \Pr[X = x, Y = y] \\ &= a \sum_x x \Pr[X = x] + b \sum_y y \Pr[Y = y] \\ &= aE[X] + bE[Y]. \end{aligned}$$

Note that this proof does *not* require that X and Y be independent. The sum of two fair six-sided dice always has expectation $\frac{7}{2} + \frac{7}{2} = 7$, whether they are independent dice, the same die counted twice, or one die X and its complement $7 - X$.

Linearity of expectation makes it easy to compute the expectations of random variables that are expressed as sums of other random variables. One example that will come up a lot is a **binomial random variable**, which is a sum $S = \sum_{i=1}^n X_i$ of n independent (Bernoulli random variables, each of which is 1 with probability p and 0 with probability $q = 1 - p$). These are called binomial random variables because the probability that S is equal to k is given by

$$\Pr[S = k] = \binom{n}{k} p^k q^{n-k}, \quad (3.4.1)$$

which is the k -th term in the binomial expansion of $(p + q)^n$. In this case each X_i has $E[X_i] = p$, so $E[S]$ is just np . It is possible to calculate this fact directly from (3.4.1), but it's a lot more work.⁵

⁵One way is to use the **probability generating function** $F(z) = \sum_{k=0}^{\infty} \Pr[S = k] z^k = \sum_{k=0}^{\infty} \binom{n}{k} p^k q^{n-k} z^k = (pz + q)^n$. Then take the derivative $F'(z) = \sum_{k=0}^{\infty} \Pr[S = k] k z^{k-1}$ and observe $F'(1) = \sum_{k=0}^{\infty} \Pr[S = k] k = E[S]$. But we can also write $F'(z)$ as $n(pz + q)^{n-1} p$, which lets us compute $F'(1) = n(p + q)^{n-1} p = np$.

3.4.1.1 Linearity of expectation for infinite sequences

For infinite sequences of random variables, linearity of expectation may break down. This is true even if the sequence is countable. An example is the **St. Petersburg paradox**, in which a gambler bets \$1 on a double-or-nothing game, then bets \$2 if she loses, then \$4, and so on, until she eventually wins and stops, up \$1. If we represent the gambler's gain or loss at stage i as a random variable X_i , it's easy to show that $E[X_i] = 0$, because the gambler either wins $\pm 2^i$ with equal probability, or doesn't play at all. So $\sum_{i=0}^{\infty} E[X_i] = 0$. But $E[\sum_{i=0}^{\infty} X_i] = 1$, because the probability that the gambler doesn't eventually win is zero.⁶

Fortunately, these pathological cases don't come up often in algorithm analysis, and with some additional side constraints we can apply linearity of expectation even to infinite sums of random variables. The simplest is when $X_i \geq 0$ for all i ; then $E[\sum_{i=0}^{\infty} X_i]$ exists and is equal to $\sum_{i=0}^{\infty} E[X_i]$ whenever the sum of the expectations converges (this is a consequence of the monotone convergence theorem). Another condition that works is if $|\sum_{i=0}^n X_i| \leq Y$ for all n , where Y is a random variable with finite expectation; the simplest version of this is when Y is constant. See [GS92, §5.6.12] or [Fel71, §IV.2] for more details.

3.4.2 Expectation and inequalities

If $X \leq Y$ (that is, if the event $[X \leq Y]$ holds with probability 1), then $E[X] \leq E[Y]$. For finite discrete spaces the proof is trivial:

$$\begin{aligned} E[X] &= \sum_{\omega \in \Omega} \Pr[\omega] X(\omega) \\ &\leq \sum_{\omega \in \Omega} \Pr[\omega] Y(\omega) \\ &= E[Y]. \end{aligned}$$

(The claim continues to hold even in more general cases, but the proof is more work.)

⁶The trick here is that we are trading a probability-1 gain of 1 against a probability-0 loss of ∞ . So we could declare that $E[\sum_{i=0}^{\infty} X_i]$ involves $0 \cdot (-\infty)$ and is undefined. But this would lose the useful property that expectation isn't affected by probability-0 outcomes. As often happens in mathematics, we are forced to choose between candidate definitions based on which bad consequences we most want to avoid, with no way to avoid all of them. So the standard definition of expectation allows the St. Petersburg paradox because the alternatives are worse.

One special case of this that comes up a lot is that $X \geq 0$ implies $E[X] \geq 0$.

3.4.3 Expectation of a product

When two random variables X and Y are independent, it also holds that $E[XY] = E[X]E[Y]$. The proof (at least for discrete random variables) is straightforward:

$$\begin{aligned} E[XY] &= \sum_x \sum_y xy \Pr[X = x, Y = y] \\ &= \sum_x \sum_y xy \Pr[X = x] \Pr[Y = y] \\ &= \left(\sum_x x \Pr[X = x] \right) \left(\sum_y \Pr[Y = y] \right) \\ &= E[X]E[Y]. \end{aligned}$$

For example, the expectation of the product of two independent fair six-sided dice is $\left(\frac{7}{2}\right)^2 = \frac{49}{4}$.

This is not true for arbitrary random variables. If we compute the expectation of the product of a single fair six-sided die with itself, we get $\frac{1 \cdot 1 + 2 \cdot 2 + 3 \cdot 3 + 4 \cdot 4 + 5 \cdot 5 + 6 \cdot 6}{6} = \frac{91}{6}$ which is much larger.

The difference $E[XY] - E[X] \cdot E[Y]$ is called the **covariance** of X and Y , written $\text{Cov}[X, Y]$. It will come back later when we look at concentration bounds in Chapter 5.

3.4.3.1 Wald's equation (simple version)

Computing the expectation of a product does not often come up directly in the analysis of a randomized algorithm. Where we might expect to do it is when we have a loop: one random variable N tells us the number of times we execute the loop, while another random variable X tells us the cost of each iteration. The problem is that if each iteration is randomized, then we really have a sequence of random variables X_1, X_2, \dots , and what we want to calculate is

$$E \left[\sum_{i=1}^N X_i \right], \tag{3.4.2}$$

where we can't use the sum formula directly because N is a random variable and we can't use the product formula because the X_i are all different random variables.

If N and the X_i are all independent,⁷ and N is bounded by some fixed maximum n , then we can apply the product rule to get the value of (3.4.2) by throwing in a few indicator variables. The idea is that the contribution of X_i to the sum is given by $X_i 1_{[N \geq i]}$, and because we assume that N is independent of the X_i , if we need to compute $E[X_i 1_{[N \geq i]}]$, we can do so by computing $E[X_i] E[1_{[N \geq i]}]$.

So we get

$$\begin{aligned} E\left[\sum_{i=1}^N X_i\right] &= E\left[\sum_{i=1}^n X_i 1_{[N \geq i]}\right] \\ &= \sum_{i=1}^n E[X_i 1_{[N \geq i]}] \\ &= \sum_{i=1}^n E[X_i] E[1_{[N \geq i]}]. \end{aligned}$$

For general X_i we have to stop here. But if we also know that the X_i all have the same expectation μ , then $E[X_i]$ doesn't depend on i and we can bring it out of the sum. This gives

$$\begin{aligned} \sum_{i=1}^n E[X_i] E[1_{[N \geq i]}] &= \mu \sum_{i=1}^n E[1_{[N \geq i]}] \\ &= \mu E[N]. \end{aligned} \tag{3.4.3}$$

This equation is a special case of **Wald's equation**, which we will see again in §8.5.2. The main difference between this version and the general version is that here we had to assume that N was independent of the X_i , which may not be true if our loop is a **while** loop, and termination is correlated with the time taken by the last iteration.

⁷When I presented this in lecture, an objection was raised for the randomized loop example that N is most likely not independent of the X_i , because a “successful” iteration of the loop that causes the algorithm to terminate may have a different distribution on cost than a “failed” iteration. This turns out not to be a problem, because this difference ends up being counted in the expectations of the X_i . For the specific case of the loop, we can argue that if each attempt succeeds with probability p after s steps on average, and fails with probability $q = 1 - p$ after f steps on average, then the expected cost is $s + f(1/p - 1)$, since on average we get $1/p - 1$ failures before our first success. Alternatively, we can compute $\mu = E[X_i] = ps + (1 - p)f$ and $E[N] = 1/p$, giving $\mu E[N] = (ps + (1 - p)f)/p = s + (1/p - 1)f$, which happens to be the same quantity. This coincidence turns out to be a consequence of the full version of Wald's equation, described in §8.5.2.

But for simple cases, (3.4.3) can still be useful. For example, if we throw one six-sided die to get N , and then throw N six-sided dice and add them up, we get the same expected total $\frac{7}{2} \cdot \frac{7}{2} = \frac{49}{4}$ as if we just multiply two six-sided dice. This is true even though the actual distribution of values is very different in the two cases.

3.5 Conditional expectation

We can also define a notion of **conditional expectation**, analogous to conditional probability. There are three versions of this, depending on how fancy we want to get about specifying what information we are conditioning on.

3.5.1 Expectation conditioned on an event

The conditional expectation of X conditioned on an *event* A is written $E[X | A]$ and defined by

$$E[X | A] = \sum_x x \Pr[X = x | A] = \sum_x \frac{\Pr[X = x \wedge A]}{\Pr[A]}. \quad (3.5.1)$$

This is essentially the weighted average value of X if we know that A occurs.

Most of the properties that we see with ordinary expectations continue to hold for conditional expectation. For example, linearity of expectation

$$E[aX + bY | A] = aE[X | A] + bE[Y | A] \quad (3.5.2)$$

holds whenever a and b are constant on A .

Similarly if $\Pr[X \geq Y | A] = 1$, $E[X | A] \geq E[Y | A]$.

Conditional expectation is handy because we can use it to compute expectations by case analysis the same way we use conditional probabilities using the law of total probability (see §2.3.2). If A_1, A_2, \dots are a countable

partition of Ω , then

$$\begin{aligned}
 \sum_i \Pr[A_i] E[X | A_i] &= \sum_i \Pr[A_i] \left(\sum_x x \Pr[X = x | A_i] \right) \\
 &= \sum_i \Pr[A_i] \left(\sum_x x \frac{\Pr[X = x \wedge A_i]}{\Pr[A_i]} \right) \\
 &= \sum_i \Pr[A_i] \left(\sum_x x \frac{\Pr[X = x \wedge A_i]}{\Pr[A_i]} \right) \\
 &= \sum_i \sum_x x \Pr[X = x \wedge A_i] \\
 &= \sum_x x \left(\sum_i \Pr[X = x \wedge A_i] \right) \\
 &= \sum_x x \Pr[X = x] \\
 &= E[X].
 \end{aligned} \tag{3.5.3}$$

This is actually a special case of the law of iterated expectation, which we will see in the next section.

3.5.2 Expectation conditioned on a random variable

In the previous section, we considered computing $E[X]$ by breaking it up into disjoint cases $E[X | A_1]$, $E[X | A_2]$, etc. But keeping track of all the events in our partition of Ω is a lot of book-keeping. Conditioning on a random variable lets us combine all these conditional probabilities into a single expression

$$E[X | Y],$$

the **expectation of X conditioned on Y** which is defined to have the value $E[X | Y = y]$ whenever $Y = y$.⁸

Note that $E[X | Y]$ is generally a function of Y , unlike $E[X]$ which is a constant. This also means that $E[X | Y]$ is a random variable, and its value can depend on which outcome ω we picked from our probability space Ω . The intuition behind the definition is that $E[X | Y]$ is the best estimate we can make of X given that we know the value of Y but nothing else.

⁸If Y is not discrete, the situation is more complicated. See [Fel71, §§III.2 and V.9–V.11] or [GS92, §7.9].

If we want to be formal about the definition, we can specify the value of $E[X | Y]$ explicitly for each point $\omega \in \Omega$:

$$E[X | Y](\omega) = E[X | Y = Y(\omega)]. \quad (3.5.4)$$

This is just another way of saying what we said already: if you want to know what the expectation of X is conditioned on Y when you get outcome ω , find the value of Y at ω and condition on seeing that.

Here is a simple example. Suppose that X and Y are independent fair coin-flips that take on the values 0 and 1 with equal probability. Then our probability space Ω has four elements, and looks like this:

$$\begin{array}{cc} \langle 0, 0 \rangle & \langle 0, 1 \rangle \\ \langle 1, 0 \rangle & \langle 1, 1 \rangle \end{array}$$

where each tuple $\langle x, y \rangle$ gives the values of X and Y .

We can also define the total number of heads as $Z = X + Y$. If we label all the points ω in our probability space with $Z(\omega)$, we get a picture that looks like this:

$$\begin{array}{cc} 0 & 1 \\ 1 & 2 \end{array}$$

This is what we see if we know the exact value of both coin-flips (or at least the exact value of Z).

But now suppose we only know X , and want to compute $E[Z | X]$. When $X = 0$, $E[Z | X = 0] = \frac{1}{2} \cdot 0 + \frac{1}{2} \cdot 1 = \frac{1}{2}$; and when $X = 1$, $E[Z | X = 1] = \frac{1}{2} \cdot 1 + \frac{1}{2} \cdot 2 = \frac{3}{2}$. So drawing $E[Z | X]$ over our probability space gives

$$\begin{array}{cc} \frac{1}{2} & \frac{1}{2} \\ \frac{3}{2} & \frac{3}{2} \end{array}$$

We've averaged the value of Z across each row, since each row corresponds to one of the possible values of X .

If instead we compute $E[Z | Y]$, we get this picture instead:

$$\begin{array}{cc} \frac{1}{2} & \frac{3}{2} \\ \frac{1}{2} & \frac{3}{2} \end{array}$$

Now instead of averaging across rows (values of X) we average across columns (values of Y). So the left column shows $E[Z | Y = 0] = \frac{1}{2}$ and the right column shows $E[Z | Y = 1] = \frac{3}{2}$, which is pretty much what we'd expect.

Nothing says that we can only condition on X and Y . What happens if we condition on Z ?

Now we are going to get fixed values for each possible value of Z . If we compute $E[X | Z]$, then when $Z = 0$ this will be 0 (because $Z = 0$ implies $X = 0$), and when $Z = 2$ this will be 1 (because $Z = 2$ implies $X = 1$). The middle case is $E[X | Z = 1] = \frac{1}{2} \cdot 0 + \frac{1}{2} \cdot 1 = \frac{1}{2}$, because the two outcomes $\langle 0, 1 \rangle$ and $\langle 1, 0 \rangle$ that give $Z = 1$ are equally likely. The picture is

$$\begin{array}{cc} 0 & \frac{1}{2} \\ \frac{1}{2} & 1 \end{array}$$

3.5.2.1 Calculating conditional expectations

Usually we will not try to $E[X | Y]$ individually for each possible ω or even each possible value y of Y . Instead, we can use various basic facts to compute $E[X | Y]$ by applying arithmetic to random variables.

The two basic facts to start with are:

1. If X is a function of Y , then $E[X | Y] = X$. Proof: Suppose $X = f(Y)$. From (3.5.4), for each outcome ω , we have $E[X | Y](\omega) = E[X | Y = Y(\omega)] = E[f(Y(\omega)) | Y = Y(\omega)] = f(Y(\omega)) = X(\omega)$.
2. If X is independent of Y , then $E[X | Y] = E[X]$. Proof: Now for each ω , we have $E[X | Y](\omega) = E[X | Y = Y(\omega)] = E[X]$.

We also have a rather strong version of linearity of expectation. If A and B are both functions of Z , then

$$E[AX + BY | Z] = A E[X | Z] + B E[Y | Z]. \quad (3.5.5)$$

Here is a proof for discrete probability spaces. For each value z of Z , we have

$$\begin{aligned} E[A(Z)X + B(Z)Y | Z = z] &= \sum_{\omega \in Z^{-1}(z)} \frac{\Pr[\omega]}{\Pr[Z = z]} (A(z)X(\omega) + B(z)Y(\omega)) \\ &= A(z) \sum_{\omega \in Z^{-1}(z)} \frac{\Pr[\omega]}{\Pr[Z = z]} X(\omega) + B(z) \sum_{\omega \in Z^{-1}(z)} \frac{\Pr[\omega]}{\Pr[Z = z]} Y(\omega) \\ &= A(z) E[X | Z = z] + B(z) E[Y | Z = z], \end{aligned}$$

which is the value when $Z = z$ of $A E[X | Z] + B E[Y | Z]$.

This means that we can quickly simplify many conditional expectations. If we go back to the example of the previous section, where $Z = X + Y$ is

the sum of two independent fair coin-flips X and Y , then we can compute

$$\begin{aligned} \mathbb{E}[Z | X] &= \mathbb{E}[X + Y | X] \\ &= \mathbb{E}[X | X] + \mathbb{E}[Y | X] \\ &= X + \mathbb{E}[Y] \\ &= X + \frac{1}{2}. \end{aligned}$$

Similarly, $\mathbb{E}[Z | Y] = \mathbb{E}[X + Y | Y] = \frac{1}{2} + Y$.

In some cases we have enough additional information to run this in reverse. If we know Z and want to estimate X , we can use the fact that X and Y are symmetric to argue that $\mathbb{E}[X | Z] = \mathbb{E}[Y | Z]$. But then $Z = \mathbb{E}[X + Y | Z] = 2\mathbb{E}[X | Z]$, so $\mathbb{E}[X | Z] = Z/2$. Note that this works in general only if the events $[X = a, Y = b]$ and $[X = b, Y = a]$ have the same probabilities for all a and b even if we condition on Z , which in this case follows from the fact that X and Y are independent and identically distributed and that addition is commutative. Other cases may be messier.⁹

Other facts, like $X \geq Y$ implies $\mathbb{E}[X | Z] \geq \mathbb{E}[Y | Z]$, can be proved using similar techniques.

3.5.2.2 The law of iterated expectation

The **law of iterated expectation** says that

$$\mathbb{E}[X] = \mathbb{E}[\mathbb{E}[X | Y]]. \quad (3.5.6)$$

When Y is discrete, this is just (3.5.3) in disguise. For each of the (countably many) values of Y that occurs with nonzero probability, let A_y be the event $[Y = y]$. Then these events are a countable partition of Ω , and

$$\begin{aligned} \mathbb{E}[\mathbb{E}[X | Y]] &= \sum_y \Pr[Y = y] \mathbb{E}[\mathbb{E}[X | Y] | Y = y] \\ &= \sum_y \Pr[Y = y] \mathbb{E}[X | Y = y] \\ &= \mathbb{E}[X]. \end{aligned}$$

The trick here is that we use (3.5.3) to expand out the original expression in terms of the events A_y , then notice that $\mathbb{E}[X | Y]$ is equal to $\mathbb{E}[X | Y = y]$ whenever $Y = y$.

⁹An example with X and Y i.d. but not independent is to imagine that we roll a six-sided die to get X , and let $Y = X + 1$ if $X < 6$ and $Y = 1$ if $X = 6$. Now knowing $Z = X + Y = 3$ tells me that $X = 1$ and $Y = 2$ exactly, neither of which is $Z/2$.

So as claimed, conditioning on a variable gives a way to write averaging over cases very compactly.

It's also not too hard to show that iterated expectation works with partial conditioning:

$$E[E[X | Y, Z] | Y] = E[X | Y]. \quad (3.5.7)$$

3.5.2.3 Conditional expectation as orthogonal projection (optional)

If you are comfortable with linear algebra, it may be helpful to think about expectation conditioned on a random variable as a form of projection onto a subspace. In this section, we'll give a brief description of how this works for a finite, discrete probability space. For a more general version, see for example [GS92, §7.9].

Consider the set of all real-valued random variables on the probability space $\{\text{TT}, \text{TH}, \text{HT}, \text{HH}\}$ corresponding to flipping two independent fair coins. We can think of each such random variable as a vector in \mathbb{R}^4 , where the four coordinates give the value of the variable on the four possible outcomes. For example, the indicator variable for the event that the first coin is heads would look like $X = \langle 0, 0, 1, 1 \rangle$ and the indicator variable for the event that the second coin is heads would look like $Y = \langle 0, 1, 0, 1 \rangle$.

When we add two random variables together, we get a new random variable. This corresponds to vector addition: $X + Y = \langle 0, 0, 1, 1 \rangle + \langle 0, 1, 0, 1 \rangle = \langle 0, 1, 1, 2 \rangle$. Multiplying a random variable by a constant looks like scalar multiplication $2X = 2 \cdot \langle 0, 0, 1, 1 \rangle = \langle 0, 0, 2, 2 \rangle$. Because random variables support both addition and scalar multiplication, and because these operations obey the axioms of a **vector space**, we can treat the set of all real-valued random variables defined on a given probability space as a vector space, and apply all the usual tools from linear algebra to this vector space.

One thing in particular we can look at is subspaces of this vector space. Consider the set of all random variables that are functions of X . These are vectors of the form $\langle a, a, b, b \rangle$, and adding any two such vectors or multiplying any such vector by a scalar yields another vector of this form. So the functions of X form a two-dimensional subspace of the four-dimensional space of all random variables. An even lower-dimensional subspace is the one-dimensional subspace of constants: vectors of the form $\langle a, a, a, a \rangle$. As with functions of X , this set is closed under addition and multiplication by a constant.

When we take the expectation of X , we are looking for a constant that gives us the average value of X . In vector terms, this means that $E[\langle 0, 0, 1, 1 \rangle] = \langle 1/2, 1/2, 1/2, 1/2 \rangle$. This expectation vector is in fact the orthogonal projection of $\langle 0, 0, 1, 1 \rangle$ onto the subspace generated by $\mathbf{1} =$

$\langle 1, 1, 1, 1 \rangle$; we can tell this because the dot-product of $X - E[X]$ with $\mathbf{1}$ is $\langle -1/2, -1/2, 1/2, 1/2 \rangle \cdot \langle 1, 1, 1, 1 \rangle = 0$. If instead we take a conditional expectation, we are again doing an orthogonal projection, but now onto a higher-dimensional subspace. So $E[X + Y | X] = \langle 1/2, 1/2, 3/2, 3/2 \rangle$ is the orthogonal projection of $X + Y = \langle 0, 1, 1, 2 \rangle$ onto the space of all functions of X , which is generated by the vectors $\langle 0, 0, 1, 1 \rangle$ and $\langle 1, 1, 0, 0 \rangle$. As in the simple expectation, the dot-product of $(X + Y) - E[X + Y | X]$ with either of these basis vectors is 0.

Many facts about conditional expectation translate in a straightforward way to facts about projections. Linearity of expectation is equivalent to linearity of projection onto a subspace. The law of iterated expectation $E[E[X | Y]] = E[X]$ says that projecting onto the subspace of function of Y and then onto the subspace of constants is equivalent to projection directly down to the subspace of constants; this is true in general for projection operations. It's also possible to represent other features of probability spaces in terms of expectations; for example, $E[XY]$ acts like an inner product for random variables, $E[X^2]$ acts like the square of the Euclidean distance, and the fact that $E[X]$ is an orthogonal projection of X means that $E[X]$ is precisely the constant value μ that minimizes the distance $E[(X - \mu)^2]$. We won't actually use any of these facts in the following, but having another way to look at conditional expectation may be helpful in understanding how it works.

3.5.3 Expectation conditioned on a σ -algebra

Expectation conditioned on a random variable is actually a special case of the expectation of X conditioned on a σ -algebra \mathcal{F} . Recall that a σ -algebra is a family of subsets of Ω that includes Ω and is closed under complement and countable union; for discrete probability spaces, this turns out to be the set of all unions of equivalence classes for some equivalence relation on Ω ,¹⁰ and we think of \mathcal{F} as representing knowledge of which equivalence class we are in, but not which point in the equivalence class we land on. An example would be if Ω consists of all values (X_1, X_2) obtained from two die rolls, and \mathcal{F} consists of all sets A such that whenever one point ω with $X_1(\omega) + X_2(\omega) = s$

¹⁰Proof: Let \mathcal{F} be a σ -algebra over a countable set Ω . Let $\omega \sim \omega'$ if, for all A in \mathcal{F} , $\omega \in A$ if and only if $\omega' \in A$; this is an equivalence relation on Ω . To show that the equivalence classes of \sim are elements of \mathcal{F} , for each $\omega'' \not\sim \omega$, let $A_{\omega''}$ be some element of \mathcal{F} that contains ω but not ω'' . Then $\bigcap_{\omega''} A_{\omega''}$ (a countable intersection of elements of \mathcal{F}) contains ω and all points $\omega' \sim \omega$ but no points $\omega'' \not\sim \omega$; in other words, it's the equivalence class of ω . Since there are only countably many such equivalence classes, we can construct all the elements of \mathcal{F} by taking all possible unions of them.

is in A , so is every other point ω' with $X_1(\omega') + X_2(\omega') = s$. (This is the σ -algebra **generated by** the random variable $X_1 + X_2$.)

A discrete random variable X is **measurable** with respect to \mathcal{F} , or **\mathcal{F} -measurable**, if every event $[X = x]$ is contained in \mathcal{F} ; in other words, knowing only where we are in \mathcal{F} , we can compute exactly the value of X . This gives a formal way to define $\sigma(X)$: it is the smallest σ -algebra \mathcal{F} such that X is \mathcal{F} -measurable.

If X is not \mathcal{F} -measurable, the best approximation we can make to it given that we only know where we are in \mathcal{F} is $E[X | \mathcal{F}]$, which is defined as a random variable Q that is (a) \mathcal{F} -measurable; and (b) satisfies $E[Q | A] = E[X | A]$ for any event $A \in \mathcal{F}$ with $\Pr[A] \neq 0$.

For discrete probability spaces, this just means that we replace X with its average value across each equivalence class: property (a) is satisfied because $E[X | \mathcal{F}]$ is constant across each equivalence class, meaning that $[E[X | \mathcal{F}] = x]$ is a union of equivalence classes, and property (b) is satisfied because we define $E[E[X | \mathcal{F}] | A] = E[X | A]$ for each equivalence class A , and the same holds for unions of equivalence classes by a simple calculation.

This gives the same result as $E[X | Y]$ if \mathcal{F} is generated by Y , or more generally as $E[X | Y_1, Y_2, \dots]$ if \mathcal{F} is generated by Y_1, Y_2, \dots . In each case the intuition is that we are getting the best estimate we can for X given the information we have. It is also possible to define $E[X | \mathcal{F}]$ as a projection onto the subspace of all random variables that are \mathcal{F} -measurable, analogously to the special case for $E[X | Y]$ described in §3.5.2.3.

Sometimes it is convenient to use more than one σ -algebra to represent increasing knowledge over time. A **filtration** is a sequence of σ -algebras $\mathcal{F}_0 \subseteq \mathcal{F}_1 \subseteq \mathcal{F}_2 \subseteq \dots$, where each \mathcal{F}_t represents the information we have available at time t . That each \mathcal{F}_t is a subset of \mathcal{F}_{t+1} means that any event we can determine at time t we can also determine at all future times $t' > t$: though we may learn more information over time, we never forget what we already know. A common example of a filtration is when we have a sequence of random variables X_1, X_2, \dots , and define \mathcal{F}_t as the σ -algebra $\langle X_1, X_2, \dots, X_t \rangle$ generated by X_1, X_2, \dots, X_t .

When one σ -algebra is a subset of another, a version of the law of iterated expectation applies: $\mathcal{F} \in \mathcal{F}'$ implies $E[E[X | \mathcal{F}'] | \mathcal{F}] = E[X | \mathcal{F}]$. One way to think about this is that if we forget everything about X we can't predict from \mathcal{F}' and then forget everything that's left that we can't predict from \mathcal{F} , we get to the same place as if we just forget everything except \mathcal{F} to begin with. The simplest version $E[X | \mathcal{F}'] = E[X]$ is just what happens when \mathcal{F} is the trivial σ -algebra $\{\emptyset, \Omega\}$, where all we know is that something happened, but we don't know what.

3.5.4 Examples

- Let X be the value of a six-sided die. Let A be the event that X is even. Then

$$\begin{aligned} E[X | A] &= \sum_x x \Pr[X = x | A] \\ &= (2 + 4 + 6) \cdot \frac{1}{3} \\ &= 4. \end{aligned}$$

- Let X and Y be independent six-sided dice, and let $Z = X + Y$. Then $E[Z | X]$ is a random variable whose value is $1 + 7/2$ when $X = 1$, $2 + 7/2$ when $X = 2$, etc. We can write this succinctly by writing $E[Z | X] = X + 7/2$.
- Conversely, if X , Y , and Z are as above, we can also compute $E[X | Z]$. Here we are told what Z is and must make an estimate of X .

For some values of Z , this nails down X completely: $E[X | Z = 2] = 1$ because X you can only make 2 in this model as $1 + 1$. For other values, we don't know much about X , but can still compute the expectation. For example, to compute $E[X | Z = 5]$, we have to average X over all the pairs (X, Y) that sum to 5. This gives $E[X | Z = 5] = \frac{1}{4}(1 + 2 + 3 + 4) = \frac{5}{2}$. (This is not terribly surprising, since by symmetry $E[Y | Z = 5]$ should equal $E[X | Z = 5]$, and since conditional expectations add just like regular expectations, we'd expect that the sum of these two expectations would be 5.)

The actual random variable $E[X | Z]$ summarizes these conditional expectations for all events of the form $[Z = z]$. Because of the symmetry argument above, we can write it succinctly as $E[X | Z] = \frac{Z}{2}$. Or we could list its value for every ω in our underlying probability space, as in done in Table 3.2 for this and various other conditional expectations on the two-independent-dice space.

3.6 Applications

3.6.1 Yao's lemma

In Section 1.1, we considered a special case of the unordered search problem, where we have an unordered array $A[1..n]$ and want to find the location

ω	X	Y	$Z = X + Y$	$E[X]$	$E[X Y = 3]$	$E[X Y]$	$E[Z X]$	$E[X Z]$	$E[X X]$
(1,1)	1	1	2	7/2	7/2	7/2	1 + 7/2	2/2	1
(1,2)	1	2	3	7/2	7/2	7/2	1 + 7/2	3/2	1
(1,3)	1	3	4	7/2	7/2	7/2	1 + 7/2	4/2	1
(1,4)	1	4	5	7/2	7/2	7/2	1 + 7/2	5/2	1
(1,5)	1	5	6	7/2	7/2	7/2	1 + 7/2	6/2	1
(1,6)	1	6	7	7/2	7/2	7/2	1 + 7/2	7/2	1
(2,1)	2	1	3	7/2	7/2	7/2	2 + 7/2	3/2	2
(2,2)	2	2	4	7/2	7/2	7/2	2 + 7/2	4/2	2
(2,3)	2	3	5	7/2	7/2	7/2	2 + 7/2	5/2	2
(2,4)	2	4	6	7/2	7/2	7/2	2 + 7/2	6/2	2
(2,5)	2	5	7	7/2	7/2	7/2	2 + 7/2	7/2	2
(2,6)	2	6	8	7/2	7/2	7/2	2 + 7/2	8/2	2
(3,1)	3	1	4	7/2	7/2	7/2	3 + 7/2	4/2	3
(3,2)	3	2	5	7/2	7/2	7/2	3 + 7/2	5/2	3
(3,3)	3	3	6	7/2	7/2	7/2	3 + 7/2	6/2	3
(3,4)	3	4	7	7/2	7/2	7/2	3 + 7/2	7/2	3
(3,5)	3	5	8	7/2	7/2	7/2	3 + 7/2	8/2	3
(3,6)	3	6	9	7/2	7/2	7/2	3 + 7/2	9/2	3
(4,1)	4	1	5	7/2	7/2	7/2	4 + 7/2	5/2	4
(4,2)	4	2	6	7/2	7/2	7/2	4 + 7/2	6/2	4
(4,3)	4	3	7	7/2	7/2	7/2	4 + 7/2	7/2	4
(4,4)	4	4	8	7/2	7/2	7/2	4 + 7/2	8/2	4
(4,5)	4	5	9	7/2	7/2	7/2	4 + 7/2	9/2	4
(4,6)	4	6	10	7/2	7/2	7/2	4 + 7/2	10/2	4
(5,1)	5	1	6	7/2	7/2	7/2	5 + 7/2	6/2	5
(5,2)	5	2	7	7/2	7/2	7/2	5 + 7/2	7/2	5
(5,3)	5	3	8	7/2	7/2	7/2	5 + 7/2	8/2	5
(5,4)	5	4	9	7/2	7/2	7/2	5 + 7/2	9/2	5
(5,5)	5	5	10	7/2	7/2	7/2	5 + 7/2	10/2	5
(5,6)	5	6	11	7/2	7/2	7/2	5 + 7/2	11/2	5
(6,1)	6	1	7	7/2	7/2	7/2	6 + 7/2	7/2	6
(6,2)	6	2	8	7/2	7/2	7/2	6 + 7/2	8/2	6
(6,3)	6	3	9	7/2	7/2	7/2	6 + 7/2	9/2	6
(6,4)	6	4	10	7/2	7/2	7/2	6 + 7/2	10/2	6
(6,5)	6	5	11	7/2	7/2	7/2	6 + 7/2	11/2	6
(6,6)	6	6	12	7/2	7/2	7/2	6 + 7/2	12/2	6

Table 3.2: Various conditional expectations on two independent dice

of a specific element x . For deterministic algorithms, this requires probing n array locations in the worst case, because the adversary can place x in the last place we look. Using a randomized algorithm, we can reduce this to $(n + 1)/2$ probes on average, either by probing according to a uniform random permutation or just by probing from left-to-right or right-to-left with equal probability.

Can we do better? Proving lower bounds is a nuisance even for deterministic algorithms, and for randomized algorithms we have even more to keep track of. But there is a sneaky trick that allows us to reduce randomized lower bounds to deterministic lower bounds in many cases.

The idea is that if we have a randomized algorithm that runs in time $T(x, r)$ on input x with random bits r , then for any fixed choice of r we have a deterministic algorithm. So for each n , we find some random X with $|X| = n$ and show that, for any deterministic algorithm that runs in time $T'(x)$, $E[T'(X)] \geq f(n)$. But then $E[T(X, R)] = E[E[T(X, R) | R]] = E[E[T_R(X) | R]] \geq f(n)$.

This gives us **Yao's lemma**:

Lemma 3.6.1. *Yao's lemma (informal version)[Yao77]* lemma-yao Fix some problem. Suppose there is a random distribution on inputs X of size n such that every deterministic algorithm for the problem has expected cost $T(n)$.

Then the worst-case expected cost of any randomized algorithm is at least $T(n)$.

For unordered search, putting x in a uniform random array location makes any deterministic algorithm take at least $(n + 1)/2$ probes on average. So randomized algorithms take at least $(n + 1)/2$ probes as well.

3.6.2 Geometric random variables

Suppose that we are running a Las Vegas algorithm that takes a fixed amount of time T , but succeeds only with probability p (which we take to be independent of the outcome of any other run of the algorithm). If the algorithm fails, we run it again. How long does it take on average to get the algorithm to work?

We can reduce the problem to computing $E[TX] = T E[X]$, where X is the number of times the algorithm runs. The probability that $X = n$ is exactly $(1 - p)^{n-1}p$, because we need to get $n - 1$ failures with probability $1 - p$ each followed by a single success with probability p , and by assumption all of these probabilities are independent. A variable with this kind of distribution is called a **geometric random variable**. We saw a special case of this

distribution earlier (§2.3.3.1) when we were looking at how long it took to get a tails out of a fair coin (in that case, p was $1/2$).

Using conditional expectation, it's straightforward to compute $E[X]$. Let A be the event that the algorithm succeeds on the first run, i.e., then event $[X = 1]$. Then

$$\begin{aligned} E[X] &= E[X \mid A] \Pr[A] + E[X \mid \bar{A}] \Pr[\bar{A}] \\ &= 1 \cdot p + E[X \mid \bar{A}] \cdot (1 - p). \end{aligned}$$

The tricky part here is to evaluate $E[X \mid \bar{A}]$. Intuitively, if we don't succeed the first time, we've wasted one step and are back where we started, so it should be the case that $E[X \mid \bar{A}] = 1 + E[X]$. If we want to be really careful, we can calculate this out formally (no sensible person would ever do this):

$$\begin{aligned} E[X \mid \bar{A}] &= \sum_{n=1}^{\infty} n \Pr[X = n \mid X \neq 1] \\ &= \sum_{n=2}^{\infty} n \frac{\Pr[X = n]}{\Pr[X \neq 1]} \\ &= \sum_{n=2}^{\infty} n \frac{(1-p)^{n-1}p}{1-p} \\ &= \sum_{n=2}^{\infty} n(1-p)^{n-2}p \\ &= \sum_{n=1}^{\infty} (n+1)(1-p)^{n-1}p \\ &= 1 + \sum_{n=1}^{\infty} n(1-p)^{n-1}p \\ &= 1 + E[X]. \end{aligned}$$

Since we know that $E[X] = p + (1 + E[X])(1 - p)$, a bit of algebra gives $E[X] = 1/p$, which is about what we'd expect.

There are more direct ways to get the same result. If we don't have conditional expectation to work with, we can try computing the sum $E[X] = \sum_{n=1}^{\infty} n(1-p)^{n-1}p$ directly. The easiest way to do this is probably to use generating functions (see, for example, [GKP88, Chapter 7] or [Wil06]). An alternative argument is given in [MU05, §2.4]; this uses the fact that $E[X] = \sum_{n=1}^{\infty} \Pr[X \geq n]$, which holds when X takes on only non-negative integer values.

3.6.3 Coupon collector

In the **coupon collector problem**, we throw balls uniformly and independently into n bins until every bin has at least one ball. When this happens, how many balls have we used on average?¹¹

Let X_i be the number of balls needed to go from $i - 1$ nonempty bins to i nonempty bins. It's easy to see that $X_1 = 1$ always. For larger i , each time we throw a ball, it lands in an empty bin with probability $\frac{n-i+1}{n}$. This means that X_i has a geometric distribution with probability $\frac{n-i+1}{n}$, giving $E[X_i] = \frac{n}{n-i+1}$ from the analysis in §3.6.2.

To get the total expected number of balls, take the sum

$$\begin{aligned} E\left[\sum_{i=1}^n X_i\right] &= \sum_{i=1}^n E[X_i] \\ &= \sum_{i=1}^n \frac{n}{n-i+1} \\ &= n \sum_{i=1}^n \frac{1}{i} \\ &= nH_n. \end{aligned}$$

In asymptotic terms, this is $\Theta(n \log n)$.

3.6.4 Hoare's FIND

Hoare's FIND [Hoa61b], often called **QuickSelect**, is an algorithm for finding the k -th smallest element of an unsorted array that works like QuickSort, only after partitioning the array around a random pivot we throw away the part that doesn't contain our target and recurse only on the surviving piece. As with QuickSort, we'd like to compute the expected number of comparisons used by this algorithm, on the assumption that the cost of the comparisons dominates the rest of the costs of the algorithm.

Here the indicator-variable trick gets painful fast. It turns out to be easier to get an upper bound by computing the expected number of elements that are left after each split.

First, let's analyze the pivot step. If the pivot is chosen uniformly, the number of elements X smaller than the pivot is uniformly distributed in the

¹¹The name comes from the problem of collecting coupons at random until you have all of them. A typical algorithmic application is having a cluster of machines choose jobs to finish at random from some list until all are done. The expected number of job executions to complete n jobs is given exactly by the solution to the coupon collector problem.

range 0 to $n-1$. The number of elements larger than the pivot will be $n-X-1$. In the worst case, we find ourselves recursing on the large pile always, giving a bound on the number of survivors Y of $Y \leq \max(X, n-X+1)$.

What is the expected value of Y ? By considering both ways the max can go, we get $E[Y] = E[X \mid X > n-X+1] \Pr[X > n-X+1] + E[n-X+1 \mid n-X+1 \geq X]$. For both conditional expectations we are choosing a value uniformly in either the range $[\frac{n-1}{2}]$ to $n-1$ or $[\frac{n-1}{2}] + 1$ to $n-1$, and in either case the expectation will be equal to the average of the two endpoints by symmetry. So we get

$$\begin{aligned} E[Y] &\leq \frac{n/2 + n - 1}{2} \Pr[X > n - X + 1] + \frac{n/2 + n}{2} \Pr[n - X + 1 \geq X] \\ &= \left(\frac{3}{4}n - \frac{1}{2}\right) \Pr[X > n - X + 1] + \frac{3}{4}n \Pr[n - X + 1 \geq X] \\ &\leq \frac{3}{4}n. \end{aligned}$$

Now let X_i be the number of survivors after i pivot steps. Note that $\max(0, X_i - 1)$ gives the number of comparisons at the following pivot step, so that $\sum_{i=0}^{\infty} X_i$ is an upper bound on the number of comparisons.

We have $X_0 = n$, and from the preceding argument $E[X_1] \leq (3/4)n$. But more generally, we can use the same argument to show that $E[X_{i+1} \mid X_i] \leq (3/4)X_i$, and by induction $E[X_i] \leq (3/4)^i n$. We also have that $X_j = 0$ for all $j \geq n$, because we lose at least one element (the pivot) at each pivoting step. This saves us from having to deal with an infinite sum.

Using linearity of expectation,

$$\begin{aligned} E\left[\sum_{i=0}^{\infty} X_i\right] &= E\left[\sum_{i=0}^n X_i\right] \\ &= \sum_{i=0}^n E[X_i] \\ &\leq \sum_{i=0}^n (3/4)^i n \\ &\leq 4n. \end{aligned}$$

Chapter 4

Basic probabilistic inequalities

Last updated 2019. Some material may be out of date.

Here we're going to look at some inequalities useful for proving properties of randomized algorithms. These come in two flavors: inequalities involving probabilities, which are useful for bounding the probability that something bad happens, and inequalities involving expectations, which are used to bound expected running times. Later, in Chapter 5, we'll be doing both, by looking at inequalities that show that a random variable is close to its expectation with high probability.¹

4.1 Union bound (Boole's inequality)

For any countable collection of events $\{A_i\}$,

$$\Pr \left[\bigcup A_i \right] \leq \sum \Pr [A_i]. \quad (4.1.1)$$

The direct way to prove this is to replace A_i with $B_i = A_i \setminus \bigcup_{j=1}^{i-1} A_j$. Then $\bigcup A_i = \bigcup B_i$, but since the B_i are disjoint and each B_i is a subset of the corresponding A_i , we get $\Pr [\bigcup A_i] = \Pr [\bigcup B_i] = \sum \Pr [B_i] \leq \sum \Pr [A_i]$.

¹Often, the phrase **with high probability** is used in algorithm analysis to mean specifically with probability at least $1 - n^{-c}$ for any fixed c . The idea is that if an algorithm works with high probability in this sense, then the probability that it fails each time you run it is at most n^{-c} , which means that if you run it as a subroutine in a polynomial-time algorithm that calls it at most $n^{c'}$ times, the total probability of failure is at most $n^{c'} n^{-c} = n^{c'-c}$ by the union bound. Assuming we can pick c to be much larger than c' , this makes the outer algorithm also work with high probability.

The typical use of the union bound is to show that if an algorithm can fail only if various improbable events occur, then the probability of failure is no greater than the sum of the probabilities of these events. This reduces the problem of showing that an algorithm works with probability $1 - \epsilon$ to constructing an **error budget** that divides the ϵ probability of failure among all the bad outcomes.

4.1.1 Applications

(See also the proof of Adleman's Theorem in Chapter 12.)

4.1.1.1 Balls in bins

Suppose we toss n balls into n bins. What is a reasonable estimate of the maximum number of balls in any one bin?²

Consider all $\binom{n}{k}$ sets S of k balls. If we get at least k balls in some bin, then one of these sets must all land in the same bin. Call the event that all balls in S choose the same bin A_S . The probability that A_S occurs is exactly n^{-k+1} .

So the union bound says

$$\begin{aligned} \Pr \left[\bigcup_S A_S \right] &\leq \sum_S \Pr [A_S] \\ &= \binom{n}{k} n^{-k+1} \\ &\leq \frac{n^k}{k!} n^{-k+1} \\ &= \frac{n}{k!}. \end{aligned}$$

If we want this probability to be low, we should choose k so that $k! \gg n$. Stirling's formula says that $k! \geq \sqrt{2\pi k} (k/e)^k \geq (k/e)^k$, which gives $\ln(k!) \geq k(\ln k - 1)$. If we set $k = c \ln n / \ln \ln n$, we get

$$\begin{aligned} \ln(k!) &\geq \frac{c \ln n}{\ln \ln n} (\ln c + \ln \ln n - \ln \ln \ln n - 1) \\ &\geq \frac{c \ln n}{2} \end{aligned}$$

²Algorithmic version: we insert n elements into a hash table with n positions using a random hash function. What is the maximum number of elements in any one position?

when n is sufficiently large.

It follows that the bound $n/k!$ in this case is less than $n/\exp(c \ln n/2) = n \cdot n^{-c/2} = n^{1-c/2}$. For suitable choice of c we get a high probability that every bin gets at most $O(\log n / \log \log n)$ balls.

4.2 Markov's inequality

This is the key tool for turning expectations of non-negative random variables into (upper) bounds on probabilities. Used directly, it generally doesn't give very good bounds, but it can work well if we apply it to $E[f(X)]$ for a fast-growing function f ; see Chebyshev's inequality (§5.1) or Chernoff bounds (§5.2).

If $X \geq 0$, then

$$\Pr[X \geq \alpha] \leq \frac{E[X]}{\alpha}. \quad (4.2.1)$$

The proof is immediate from the law of total probability (2.3.3). We have

$$\begin{aligned} E[X] &= E[X \mid X \geq \alpha] \Pr[X \geq \alpha] + E[X \mid X < \alpha] \Pr[X < \alpha] \\ &\geq \alpha \cdot \Pr[X \geq \alpha] + 0 \cdot \Pr[X < \alpha] \\ &= \alpha \cdot \Pr[X \geq \alpha]; \end{aligned}$$

now solve for $\Pr[X \geq \alpha]$.

Markov's inequality doesn't work in reverse. For example, consider the following game: for each integer $k > 0$, with probability 2^{-k} , I give you 2^k dollars. Letting X be your payoff from the game, we have $\Pr[X \geq 2^k] = \sum_{j=k}^{\infty} 2^{-j} = 2^{-k} \sum_{\ell=0}^{\infty} 2^{-\ell} = \frac{2}{2^k}$. The right-hand side here is exactly what we would get from Markov's inequality if $E[X] = 2$. But in this case, $E[X] \neq 2$; in fact, the expectation of X is given by $\sum_{k=1}^{\infty} 2^k 2^{-k}$, which diverges.

4.2.1 Applications

4.2.1.1 The union bound

Combining Markov's inequality with linearity of expectation and indicator variables gives a succinct proof of the union bound:

$$\begin{aligned} \Pr \left[\bigcup A_i \right] &= \Pr \left[\sum 1_{A_i} \geq 1 \right] \\ &\leq \mathbb{E} \left[\sum 1_{A_i} \right] \\ &= \sum \mathbb{E} [1_{A_i}] \\ &= \sum \Pr [A_i]. \end{aligned}$$

Note that for this to work for infinitely many events we need to use the fact that 1_{A_i} is always non-negative.

4.2.1.2 Fair coins

Flip n independent fair coins, and let S be the number of heads we get. Since $\mathbb{E}[S] = n/2$, we get $\Pr[S = n] = 1/2$. This is much larger than the actual value 2^{-n} , but it's the best we can hope for if we only know $\mathbb{E}[S]$: if we let S be 0 or n with equal probability, we also get $\mathbb{E}[S] = n/2$.

4.2.1.3 Randomized QuickSort

The expected running time for randomized QuickSort is $O(n \log n)$. It follows that the probability that randomized QuickSort takes more than $f(n)$ time is $O(n \log n / f(n))$. For example, the probability that it performs the maximum $\binom{n}{2} = O(n^2)$ comparisons is $O(\log n / n)$. (It's possible to do much better than this.)

4.2.1.4 Balls in bins

Suppose we toss m balls in n bins, uniformly and independently. What is the probability that some particular bin contains at least k balls? The probability that a particular ball lands in a particular bin is $1/n$, so the expected number of balls in the bin is m/n . This gives a bound of m/nk that a bin contains k or more balls. We can combine this with the union bound to show that the probability that any bin contains k or more balls is at most m/k . Unfortunately this is not a very good bound.

4.3 Jensen's inequality

This is mostly useful if we can calculate $E[X]$ easily for some X , but what we really care about is some other random variable $Y = f(X)$.

Jensen's inequality applies when f is a **convex function**, which means that for any x, y , and $0 \leq \mu \leq 1$, $f(\lambda x + (1 - \lambda)y) \leq \lambda f(x) + (1 - \lambda)f(y)$. Geometrically, this means that the line segment between any two points on the graph of f never goes below f ; i.e., that the set of points $\{(x, y) \mid y \geq f(x)\}$ is convex. If we want to show that f is convex, it's enough to show that $f\left(\frac{x+y}{2}\right) \leq \frac{f(x)+f(y)}{2}$ for all x and y (in effect, we only need to prove it for the case $\lambda = 1/2$). If f is twice-differentiable, an even easier way is to show that $f''(x) \geq 0$ for all x .

The inequality says that if X is a random variable and f is convex then

$$f(E[X]) \leq E[f(X)]. \quad (4.3.1)$$

Alternatively, if f is **concave** (which means that $f(\lambda x + (1 - \lambda)y) \geq \lambda f(x) + (1 - \lambda)f(y)$, or equivalently that $-f$ is convex), the reverse inequality holds:

$$f(E[X]) \geq E[f(X)]. \quad (4.3.2)$$

In both cases, the direction of Jensen's inequality matches the direction of the inequality in the definition of convexity or concavity. This is not surprising because convexity or concavity is just Jensen's inequality for the random variable X that equals x with probability λ and y with probability $1 - \lambda$. Jensen's inequality just says that this continues to work for any random variable for which the expectations exist.

4.3.1 Proof

Here is a proof for the case that f is convex and differentiable. The idea is that if f is convex, then it lies above the tangent line at $E[X]$. So we can define a linear function g that represents this tangent line, and get, for all x :

$$f(x) \geq g(x) = f(E[X]) + (x - E[X])f'(E[X]). \quad (4.3.3)$$

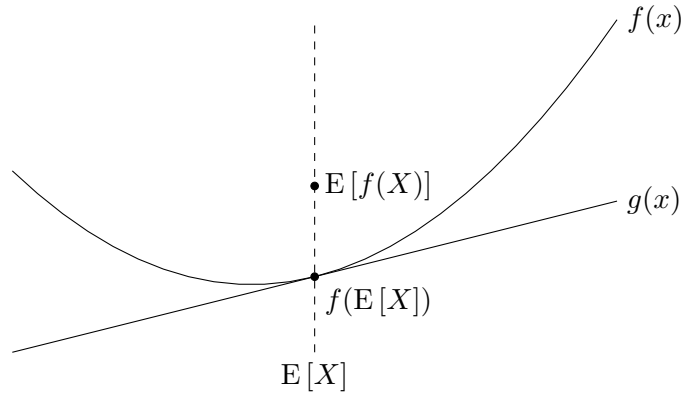


Figure 4.1: Proof of Jensen's inequality

But then

$$\begin{aligned}
 \mathbf{E}[f(X)] &\geq \mathbf{E}[g(X)] \\
 &= \mathbf{E}[f(\mathbf{E}[X]) + (X - \mathbf{E}[X])f'(\mathbf{E}[X])] \\
 &= f(\mathbf{E}[X]) + \mathbf{E}[(X - \mathbf{E}[X])]f'(\mathbf{E}[X]) \\
 &= f(\mathbf{E}[X]) + (\mathbf{E}[X] - \mathbf{E}[X])f'(\mathbf{E}[X]) \\
 &= f(\mathbf{E}[X]).
 \end{aligned}$$

Figure 4.1 shows what this looks like for a particular convex f .

This is pretty much all linearity of expectation in action: $\mathbf{E}[X]$, $f(\mathbf{E}[X])$, and $f'(\mathbf{E}[X])$ are all constants, so we can pull them out of expectations whenever we see them.

The proof of the general case is similar, but for a non-differentiable convex function it takes a bit more work to show that the bounding linear function g exists.

4.3.2 Applications

4.3.2.1 Fair coins: lower bound

Suppose we flip n fair coins, and we want to get a lower bound on $\mathbf{E}[X^2]$, where X is the number of heads. The function $f : x \mapsto x^2$ is convex (take its second derivative), so (4.3.1) gives $\mathbf{E}[X^2] \geq (\mathbf{E}[X])^2 = \frac{n^2}{4}$. (The actual value for $\mathbf{E}[X^2]$ is $\frac{n^2}{4} + \frac{n}{4}$, which can be found using generating functions.³)

³Here's how: The **probability generating function** for X is $F(z) = \mathbf{E}[z^k] = \sum_k z^k \Pr[X = k] = 2^{-n}(1+z)^n$. Then $zF'(z) = 2^{-n}nz(1+z)^{n-1} = \sum_k kz^k \Pr[X = k]$.

4.3.2.2 Fair coins: upper bound

For an upper bound, we can choose a concave f . For example, if X is as in the previous example, $E[\lg X] \leq \lg E[X] = \lg \frac{n}{2} = \lg n - 1$. This is probably pretty close to the exact value, as we will see later that X will almost always be within a factor of $1 + o(1)$ of $n/2$. It's not a terribly useful upper bound, because if we use it with (say) Markov's inequality, the most we can prove is that $\Pr[X = n] = \Pr[\lg X = \lg n] \leq \frac{\lg n - 1}{\lg n} = 1 - \frac{1}{\lg n}$, which is an even worse bound than the $1/2$ we can get from applying Markov's inequality to X directly.

4.3.2.3 Sifters

Here's an example of Jensen's inequality in action in the analysis of an actual distributed algorithm. For some problems in distributed computing, it's useful to reduce coordinating a large number of processes to coordinating a smaller number. A **sifter** [AA11] is a randomized mechanism for an asynchronous shared-memory system that sorts the processes into "winners" and "losers," guaranteeing that there is at least one winner. The goal is to make the expected number of winners as small as possible. The problem is tricky, because processes can only communicate by reading and writing shared variables, and an adversary gets to choose which processes participate and fix the schedule of when each of these processes perform their operations.

The current best known sifter is due to Giakkoupis and Woelfel [GW12]. For n processes, it uses an array A of $\lceil \lg n \rceil$ bits, each of which can be read or written by any of the processes. When a process executes the sifter, it chooses a random index $r \in 1 \dots \lceil \lg n \rceil$ with probability 2^{-r-1} (this doesn't exactly sum to 1, so the excess probability gets added to $r = \lceil \lg n \rceil$). The process then writes a 1 to $A[r]$ and reads $A[r+1]$. If it sees a 0 in its read (or chooses $r = \lceil \lg n \rceil$), it wins; otherwise it loses.

This works as a sifter, because no matter how many processes participate, some process chooses a value of r at least as large as any other process's value, and this process wins. To bound the expected number of winners, take the sum over all r over the random variable W_r representing the winners who chose this particular value r . A process that chooses r wins if it carries

Taking the derivative of this a second time gives $2^{-n}n(1+z)^{n-1} + 2^{-n}n(n-1)z(1+z)^{n-2} = \sum_k k^2 z^{k-1} \Pr[X=k]$. Evaluate this monstrosity at $z=1$ to get $E[X^2] = \sum_k k^2 \Pr[X=k] = 2^{-n}n2^{n-1} + 2^{-n}n(n-1)2^{n-2} = n/2 + n(n-1)/4 = \frac{2n+n^2-n}{4} = n^2/4 + n/4$. This is pretty close to the lower bound we got out of Jensen's inequality, but we can't count on this happening in general.

out its read operation before any process writes $r + 1$. If the adversary wants to maximize the number of winners, it should let each process read as soon as possible; this effectively means that a process that choose r wins if no process previously chooses $r + 1$. Since r is twice as likely to be chosen as $r + 1$, conditioning on a process picking r or $r + 1$, there is only a $1/3$ chance that it chooses $r + 1$. So at most $1/(1/3) - 1 = 2 = O(1)$ process on average choose r before some process chooses $r + 1$. (A simpler argument shows that the expected number of processes that win because they choose $r = \lceil \lg n \rceil$ is at most 2 as well.)

Summing $W_r \leq 2$ over all r gives at most $2\lceil \lg n \rceil$ winners on average. Furthermore, if $k < n$ processes participate, essentially the same analysis shows that only $2\lceil \lg k \rceil$ processes win on average. So this is a pretty effective tool for getting rid of excess processes.

But it gets better. Suppose that we take the winners of one sifter and feed them into a second sifter. Let X_k be the number of processes left after k sifters. We have that $X_0 = n$ and $E[X_1] \leq 2\lceil \lg n \rceil$, but what can we say about $E[X_2]$? We can calculate $E[X_2] = E[E[X_2 | X_1]] \leq 2\lceil \lg X_1 \rceil$. Unfortunately, the ceiling means that $2\lceil \lg x \rceil$ is not a concave function, but $f(x) = 2(\lg x + 1) \geq 2\lceil \lg x \rceil$ is. So $E[X_2] \leq f(f(n))$, and in general $E[X_i] \leq f^{(i)}(n)$, where $f^{(i)}$ is the i -fold composition of f . All the extra constants obscure what is going on a bit, but with a little bit of algebra it is not too hard to show that $f^{(i)}(n) = O(1)$ for $i = O(\log^* n)$.⁴ So this gets rid of all but a constant number of processes very quickly.

⁴The \log^* function counts how many times you need to hit n with \lg to reduce it to one or less. So $\log^* 1 = 0$, $\log^* 2 = 1$, $\log^* 4 = 2$, $\log^* 16 = 3$, $\log^* 65536 = 4$, $\log^* 2^{65536} = 5$, and after that it starts getting silly.

Chapter 5

Concentration bounds

Last updated 2019. Some material may be out of date.

If we really want to get tight bounds on a random variable X , the trick will turn out to be picking some non-negative function $f(X)$ where (a) we can calculate $E[f(X)]$, and (b) f grows fast enough that merely large values of X produce huge values of $f(X)$, allowing us to get small probability bounds by applying Markov's inequality to $f(X)$. This approach is often used to show that X lies close to $E[X]$ with reasonably high probability, what is known as a **concentration bound**.

Typically concentration bounds are applied to sums of random variables, which may or may not be fully independent. Which bound you may want to use often depends on the structure of your sum. A quick summary of the bounds in this chapter is given in Table 5.1. The rule of thumb is to use Chernoff bounds (§5.2) if you have a sum of independent 0–1 random variables; the Azuma-Hoeffding inequality (§5.3) if you have bounded variables with a more complicated distribution that may be less independent; and Chebyshev's inequality (§5.1) if nothing else works but you can somehow compute the variance of your sum (e.g., if the X_i are independent or have easily computed covariance). In the case of Chernoff bounds, you will almost always end up using one of the weaker but cleaner versions in §5.2.2 rather than the general version in §5.2.1.

5.1 Chebyshev's inequality

Chebyshev's inequality allows us to show that a random variable is close to its mean, by applying Markov's inequality to the **variance** of X , defined

Chernoff	$X_i \in \{0, 1\}$, independent	$\Pr[S \geq (1 + \delta) \mathbb{E}[S]] \leq \left(\frac{e^\delta}{(1+\delta)^{1+\delta}}\right)^{\mathbb{E}[S]}$
Azuma-Hoeffding	$ X_i \leq c_i$, martingale	$\Pr[S \geq t] \leq \exp\left(-\frac{t^2}{2 \sum c_i^2}\right)$
Chebyshev		$\Pr[S - \mathbb{E}[S] \geq \alpha] \leq \frac{\text{Var}[S]}{\alpha^2}$

Table 5.1: Concentration bounds for $S = \sum X_i$ (strongest to weakest)

as $\text{Var}[X] = \mathbb{E}[(X - \mathbb{E}[X])^2]$. It's a fairly weak concentration bound, that is most useful when X is a sum of random variables with limited independence.

Using Markov's inequality, calculate

$$\begin{aligned} \Pr[|X - \mathbb{E}[X]| \geq \alpha] &= \Pr[(X - \mathbb{E}[X])^2 \geq \alpha^2] \\ &\leq \frac{\mathbb{E}[(X - \mathbb{E}[X])^2]}{\alpha^2} \\ &= \frac{\text{Var}[X]}{\alpha^2}. \end{aligned} \tag{5.1.1}$$

5.1.1 Computing variance

At this point it would be reasonable to ask why we are going through $\text{Var}[X] = \mathbb{E}[(X - \mathbb{E}[X])^2]$ rather than just using $\mathbb{E}[|X - \mathbb{E}[X]|]$. The reason is that $\text{Var}[X]$ is usually easier to compute, especially if X is a sum. In this section, we'll give some examples of computing variance, including for various standard random variables that come up a lot in randomized algorithms.

5.1.1.1 Alternative formula

The first step is to give an alternative formula for the variance that is more convenient in some cases.

Expand

$$\begin{aligned} \mathbb{E}[(X - \mathbb{E}[X])^2] &= \mathbb{E}[X^2 - 2X \cdot \mathbb{E}[X] + (\mathbb{E}[X])^2] \\ &= \mathbb{E}[X^2] - 2\mathbb{E}[X] \cdot \mathbb{E}[X] + (\mathbb{E}[X])^2 \\ &= \mathbb{E}[X^2] - (\mathbb{E}[X])^2. \end{aligned} \tag{5.1.2}$$

This formula is easier to use if you are estimating the variance from a sequence of samples; by tracking $\sum x_i^2$ and $\sum x_i$, you can estimate $E[X^2]$ and $E[X]$ in a single pass, without having to estimate $E[X]$ first and then go back for a second pass to calculate $(x_i - E[X])^2$ for each sample. We won't use this particular application much, but this explains why the formula is popular with statisticians.

5.1.1.2 Variance of a Bernoulli random variable

Recall that a Bernoulli random variable is 1 with probability p and 0 with probability $q = 1 - p$; in particular, any indicator variable is a Bernoulli random variable.

The variance of a Bernoulli random variable is easily calculated from (5.1.2):

$$\begin{aligned}\text{Var}[X] &= E[X^2] - (E[X])^2 \\ &= p - p^2 \\ &= p(1 - p) \\ &= pq.\end{aligned}$$

5.1.1.3 Variance of a sum

If $S = \sum_i X_i$, then we can calculate

$$\begin{aligned}\text{Var}[S] &= E\left[\left(\sum_i X_i\right)^2\right] - \left(E\left[\sum_i X_i\right]\right)^2 \\ &= E\left[\sum_i \sum_j X_i X_j\right] - \sum_i \sum_j E[X_i] E[X_j] \\ &= \sum_i \sum_j (E[X_i X_j] - E[X_i] E[X_j]).\end{aligned}$$

For any two random variables X and Y , the quantity $E[XY] - E[X]E[Y]$ is called the **covariance** of X and Y , written $\text{Cov}[X, Y]$. If we take the covariance of a variable and itself, covariance becomes variance: $\text{Cov}[X, X] = \text{Var}[X]$.

We can use $\text{Cov}[X, Y]$ to rewrite the above expansion as

$$\text{Var} \left[\sum_i X_i \right] = \sum_{i,j} \text{Cov} [X_i, X_j] \quad (5.1.3)$$

$$= \sum_i \text{Var} [X_i] + \sum_{i \neq j} \text{Cov} [X_i, X_j] \quad (5.1.4)$$

$$= \sum_i \text{Var} [X_i] + 2 \sum_{i < j} \text{Cov} [X_i, X_j] \quad (5.1.5)$$

Note that $\text{Cov}[X, Y] = 0$ when X and Y are independent; this makes Chebyshev's inequality particularly useful for **pairwise-independent** random variables, because then we can just sum up the variances of the individual variables.

A typical application is when we have a sum $S = \sum X_i$ of non-negative random variables with small covariance; here applying Chebyshev's inequality to S can often be used to show that S is not likely to be much smaller than $E[S]$, which can be handy if we want to show that some lower bound holds on S with some probability. This complements Markov's inequality, which can only be used to get upper bounds.

For example, suppose $S = \sum_{i=1}^n X_i$, where the X_i are independent Bernoulli random variables with $E[X_i] = p$ for all i . Then $E[S] = np$, and $\text{Var}[S] = \sum_i \text{Var}[X_i] = npq$ (because the X_i are independent). Chebyshev's inequality then says

$$\Pr[|S - E[S]| \geq \alpha] \leq \frac{npq}{\alpha^2}.$$

The highest variance is when $p = 1/2$. In this case, the probability that S is more than $\beta\sqrt{n}$ away from its expected value $n/2$ is bounded by $\frac{1}{4\beta^2}$. We'll see better bounds on this problem later, but this may already be good enough for many purposes.

More generally, the approach of bounding S from below by estimating $E[S]$ and either $E[S^2]$ or $\text{Var}[S]$ is known as the **second-moment method**. In some cases, tighter bounds can be obtained by more careful analysis.

5.1.1.4 Variance of a geometric random variable

Let X be a geometric random variable with parameter p as defined in §3.6.2, so that X takes on the values $1, 2, \dots$ and $\Pr[X = n] = q^{n-1}p$, where $q = 1 - p$ as usual. What is $\text{Var}[X]$?

We know that $E[X] = 1/p$, so $(E[X])^2 = 1/p^2$. Computing $E[X^2]$ is trickier. Rather than do this directly from the definition of expectation, we

can exploit the memorylessness of geometric random variables to get it using conditional expectations, just like we did for $E[X]$ in §3.6.2.

Conditioning on the two possible outcomes of the first trial, we have

$$E[X^2] = p + qE[X^2 \mid X > 1]. \quad (5.1.6)$$

We now argue that $E[X^2 \mid X > 1] = E[(X + 1)^2]$. The intuition is that once we have flipped one coin the wrong way, we are back where we started, except that now we have to add that extra coin to X . More formally, we have, for $n > 1$, $\Pr[X^2 = n \mid X > 1] = \frac{\Pr[X^2 = n]}{\Pr[X > 1]} = \frac{q^{n-1}p}{q} = q^{n-2}p = \Pr[X = n - 1] = \Pr[X + 1 = n]$. So we get the same probability mass function for X conditioned on $X > 1$ as for $X + 1$ with no conditioning.

Applying this observation to the right-hand side of (5.1.6) gives

$$\begin{aligned} E[X^2] &= p + qE[(X + 1)^2] \\ &= p + q(E[X^2] + 2E[X] + 1) \\ &= p + qE[X^2] + \frac{2q}{p} + q \\ &= 1 + qE[X^2] + \frac{2q}{p}. \end{aligned}$$

A bit of algebra turns this into

$$\begin{aligned} E[X^2] &= \frac{1 + 2q/p}{1 - q} \\ &= \frac{1 + 2q/p}{p} \\ &= \frac{p + 2q}{p^2} \\ &= \frac{2 - p}{p^2}. \end{aligned}$$

Now subtract $(E[X])^2 = \frac{1}{p^2}$ to get

$$\text{Var}[X] = \frac{1 - p}{p^2} = \frac{q}{p^2}. \quad (5.1.7)$$

By itself, this doesn't give very good bounds on X . For example, if we

want to bound the probability that $X = 1$, we get

$$\begin{aligned}
 \Pr[X = 1] &= \Pr\left[X - \mathbb{E}[X] = 1 - \frac{1}{p}\right] \\
 &\leq \Pr\left[|X - \mathbb{E}[X]| \geq \frac{1}{p} - 1\right] \\
 &\leq \frac{\text{Var}[X]}{\left(\frac{1}{p} - 1\right)^2} \\
 &= \frac{q/p^2}{\left(\frac{1}{p} - 1\right)^2} \\
 &= \frac{q}{(1-p)^2} \\
 &= \frac{1}{q}.
 \end{aligned}$$

Since $\frac{1}{q} \geq 1$, we could have gotten this bound with a lot less work.

The other direction is not much better. We can easily calculate that $\Pr[X \geq n]$ is exactly q^{n-1} (because this corresponds to flipping n coins the wrong way, no matter what happens with subsequent coins). Using Chebyshev's inequality gives

$$\begin{aligned}
 \Pr[X \geq n] &\leq \Pr\left[\left|X - \frac{1}{p}\right| \geq n - \frac{1}{p}\right] \\
 &\leq \frac{q/p^2}{\left(n - \frac{1}{p}\right)^2} \\
 &= \frac{q}{(np - 1)^2}.
 \end{aligned}$$

This at least has the advantage of dropping below 1 when n gets large enough, but it's only polynomial in n while the true value is exponential.

Where this might be useful is in analyzing the sum of a bunch of geometric random variables, as occurs in the Coupon Collector problem discussed in §3.6.3.¹ Letting X_i be the number of balls to take us from $i - 1$ to i empty bins, we have previously argued that X_i has a geometric distribution with

¹We are following here a similar analysis in [MU05, §3.3.1].

$p = \frac{n-i-1}{n}$, so

$$\begin{aligned}\text{Var}[X_i] &= \frac{i-1}{n} \left/ \left(\frac{n-i-1}{n} \right)^2 \right. \\ &= n \frac{i-1}{(n-i-1)^2},\end{aligned}$$

and

$$\begin{aligned}\text{Var} \left[\sum_{i=1}^n X_i \right] &= \sum_{i=1}^n \text{Var}[X_i] \\ &= \sum_{i=1}^n n \frac{i-1}{(n-i-1)^2}.\end{aligned}$$

Having the numerator go up while the denominator goes down makes this a rather unpleasant sum to try to solve directly. So we will follow the lead of Mitzenmacher and Upfal and bound the numerator by n , giving

$$\begin{aligned}\text{Var} \left[\sum_{i=1}^n X_i \right] &\leq \sum_{i=1}^n n \frac{n}{(n-i-1)^2} \\ &= n^2 \sum_{i=1}^n \frac{1}{i^2} \\ &\leq n^2 \sum_{i=1}^{\infty} \frac{1}{i^2} \\ &= n^2 \frac{\pi^2}{6}.\end{aligned}$$

The fact that $\sum_{i=1}^{\infty} \frac{1}{i^2}$ converges to $\frac{\pi^2}{6}$ is not trivial to prove, and was first shown by Leonhard Euler in 1735 some ninety years after the question was first proposed.² But it's easy to show that the series converges to something, so even if we didn't have Euler's help, we'd know that the variance is $O(n^2)$.

Since the expected value of the sum is $\Theta(n \log n)$, this tells us that we are likely to see a total waiting time reasonably close to this; with at least constant probability, it will be within $\Theta(n)$ of the expectation. In fact, the distribution is much more sharply concentrated (see [MU05, §5.4.1] or [MR95, §3.6.3]), but this bound at least gives us something.

²See http://en.wikipedia.org/wiki/Basel_Problem for a history, or Euler's original paper [Eul68], available at <http://eulerarchive.maa.org/docs/originals/E352.pdf>, for the actual proof in the full glory of its original 18th-century typesetting. Curiously, though Euler announced his result in 1735, he didn't submit the journal version until 1749, and it didn't see print until 1768. Things moved more slowly in those days.

5.1.2 More examples

Here are some more examples of Chebyshev's inequality in action. Some of these repeat examples for which we previously got crummier bounds in §4.2.1.

5.1.2.1 Flipping coins

Let X be the sum of n independent fair coins. Let X_i be the indicator variable for the event that the i -th coin comes up heads. Then $\text{Var}[X_i] = 1/4$ and $\text{Var}[X] = \sum \text{Var}[X_i] = n/4$. Chebyshev's inequality gives $\Pr[X = n] \leq \Pr[|X - n/2| \geq n/2] \leq \frac{n/4}{(n/2)^2} = \frac{1}{n}$. This is still not very good, but it's getting better. It's also about the best we can do given only the assumption of pairwise independence.

To see this, let $n = 2^m - 1$ for some m , and let $Y_1 \dots Y_m$ be independent, fair 0–1 random variables. For each non-empty subset S of $\{1 \dots m\}$, let X_S be the exclusive OR of all Y_i for $i \in S$. Then (a) the X_i are pairwise independent; (b) each X_i has variance $1/4$; and thus (c) the same Chebyshev's inequality analysis for independent coin flips above applies to $X = \sum_S X_S$, giving $\Pr[|X - n/2| = n/2] \leq \frac{1}{n}$. In this case it is not actually possible for X to equal n , but we can have $X = 0$ if all the Y_i are 0, which occurs with probability $2^{-m} = \frac{1}{n+1}$. So the Chebyshev's inequality is almost tight in this case.

5.1.2.2 Balls in bins

Let X_i be the indicator that the i -th of m balls lands in a particular bin. Then $\mathbb{E}[X_i] = 1/n$, giving $\mathbb{E}[\sum X_i] = m/n$, and $\text{Var}[X_i] = 1/n - 1/n^2$, giving $\text{Var}[\sum X_i] = m/n - m/n^2$. So the probability that we get $k + m/n$ or more balls in a particular bin is at most $(m/n - m/n^2)/k^2 < m/nk^2$, and applying the union bound, the probability that we get $k + m/n$ or more balls in any of the n bins is less than m/k^2 . Setting this equal to ϵ and solving for k gives a probability of at most ϵ of getting more than $m/n + \sqrt{m/\epsilon}$ balls in any of the bins. This is not as good a bound as we will be able to prove later, but it's at least non-trivial.

5.1.2.3 Lazy select

This example comes from [MR95, §3.3]; essentially the same example, specialized to finding the median, also appears in [MU05, §3.4].³

We want to find the k -th smallest element $S_{(k)}$ of a set S of size n . (The parentheses around the index indicate that we are considering the sorted version of the set $S_{(1)} < S_{(2)} \cdots < S_{(n)}$.) The idea is to:

1. Sample a multiset R of $n^{3/4}$ elements of S with replacement and sort them. This takes $O(n^{3/4} \log n^{3/4}) = o(n)$ comparisons so far.
2. Use our sample to find an interval that is likely to contain $S_{(k)}$. The idea is to pick indices $\ell = (k - n^{3/4})n^{-1/4}$ and $r = (k + n^{3/4})n^{-1/4}$ and use $R_{(\ell)}$ and $R_{(r)}$ as endpoints (we are omitting some floors and maxes here to simplify the notation; for a more rigorous presentation see [MR95]). The hope is that the interval $P = [R_{(\ell)}, R_{(r)}]$ in S will both contain $S_{(k)}$, and be small, with $|P| \leq 4n^{3/4} + 2$. We can compute the elements of P in $2n$ comparisons exactly by comparing every element with both $R_{(\ell)}$ and $R_{(r)}$.
3. If both these conditions hold, sort P ($o(n)$ comparisons) and return $S_{(k)}$. If not, try again.

We want to get a bound on how likely it is that P either misses $S_{(k)}$ or is too big.

For any fixed k , the probability that one sample in R is less than or equal to $S_{(k)}$ is exactly k/n , so the expected number X of samples $\leq S_{(k)}$ is exactly $kn^{-1/4}$. The variance on X can be computed by summing the variances of the indicator variables that each sample is $\leq S_{(k)}$, which gives a bound $\text{Var}[X] = n^{3/4}((k/n)(1 - k/n)) \leq n^{3/4}/4$. Applying Chebyshev's inequality gives $\Pr\left[|X - kn^{-1/4}| \geq \sqrt{n}\right] \leq n^{3/4}/4n = n^{-1/4}/4$.

Now let's look at the probability that P misses $S_{(k)}$ because $R_{(\ell)}$ is too big, where $\ell = kn^{-1/4} - \sqrt{n}$. This is

$$\begin{aligned} \Pr\left[R_{(\ell)} > S_{(k)}\right] &= \Pr\left[X < kn^{-1/4} - \sqrt{n}\right] \\ &\leq n^{-1/4}/4. \end{aligned}$$

³ The history is that Motwani and Raghavan adapted this algorithm from a similar algorithm by Floyd and Rivest [FR75]. Mitzenmacher and Upfal give a version that also includes the adaptations appearing Motwani and Raghavan, although they don't say where they got it from, and it may be that both textbook versions come from a common folklore source.

(with the caveat that we are being sloppy about round-off errors).

Similarly,

$$\begin{aligned} \Pr \left[R_{(h)} < S_{(k)} \right] &= \Pr \left[X > kn^{-1/4} + \sqrt{n} \right] \\ &\leq n^{-1/4}/4. \end{aligned}$$

So the total probability that P misses $S_{(k)}$ is at most $n^{-1/4}/2$.

Now we want to show that $|P|$ is small. We will do so by showing that it is likely that $R_{(\ell)} \geq S_{(k-2n^{3/4})}$ and $R_{(h)} \leq S_{(k+2n^{3/4})}$. Let X_ℓ be the number of samples in R that are $\leq S_{(k-2n^{3/4})}$ and X_r be the number of samples in R that are $\leq S_{(k+2n^{3/4})}$. Then we have $E[X_\ell] = kn^{-1/4} - 2\sqrt{n}$ and $E[X_r] = kn^{-1/4} + 2\sqrt{n}$, and $\text{Var}[X_\ell]$ and $\text{Var}[X_r]$ are both bounded by $n^{3/4}/4$.

We can now compute

$$\Pr \left[R_{(\ell)} < S_{(k-2n^{3/4})} \right] = \Pr[X_\ell > kn^{-1/4} - \sqrt{n}] < n^{-1/4}/4$$

by the same Chebyshev's inequality argument as before, and get the symmetric bound on the other side for $\Pr \left[R_{(r)} > S_{(k+2n^{3/4})} \right]$. This gives a total bound of $n^{-1/4}/2$ that P is too big, for a bound of $n^{-1/4} = o(n)$ that the algorithm fails on its first attempt.

The total expected number of comparisons is thus given by $T(n) = 2n + o(n) + O(n^{-1/4}T(n)) = 2n + o(n)$.

5.2 Chernoff bounds

To get really tight bounds, we apply Markov's inequality to $\exp(\alpha S)$, where $S = \sum_i X_i$. This works best when the X_i are independent: if this is the case, so are the variables $\exp(\alpha X_i)$, and so we can easily calculate $E[\exp(\alpha S)] = E[\prod_i \exp(\alpha X_i)] = \prod_i E[\exp(\alpha X_i)]$.

The quantity $E[\exp(\alpha S)]$, treated as a function of α , is called the **moment generating function** of S , because it expands formally into $\sum_{k=0}^{\infty} E[X^k] \frac{\alpha^k}{k!}$, the **exponential generating function** for the series of k -th **moments** $E[X^k]$. Note that it may not converge for all S and α ;⁴ we will be careful to choose α for which it does converge and for which Markov's inequality gives us good bounds.

⁴For example, the moment generating function for our earlier bad X with $\Pr[X = 2^k] = 2^{-k}$ is equal to $\sum_k 2^{-k} e^{\alpha k}$, which diverges unless $e^\alpha/2 < 1$.

5.2.1 The classic Chernoff bound

The basic Chernoff bound applies to sums of independent 0–1 random variables, which need not be identically distributed. For identically distributed random variables, the sum has a binomial distribution, which we can either compute exactly or bound more tightly using approximations specific to binomial tails; for sums of bounded random variables that aren't necessarily 0–1, we can use Hoeffding's inequality instead (see §5.3).

Let each X_i for $i = 1 \dots n$ be a 0–1 random variable with expectation p_i , so that $E[S] = \mu = \sum_i p_i$. The plan is to show $\Pr[S \geq (1 + \delta)\mu]$ is small when δ and μ are large, by applying Markov's inequality to $E[e^{\alpha S}]$, where α will be chosen to make the bound as tight as possible for some specific δ . The first step is to get an upper bound on $E[e^{\alpha S}]$.

Compute

$$\begin{aligned} E[e^{\alpha S}] &= E[e^{\alpha \sum X_i}] \\ &= \prod_i E[e^{\alpha X_i}] \\ &= \prod_i (p_i e^{\alpha} + (1 - p_i)e^0) \\ &= \prod_i (p_i e^{\alpha} + 1 - p_i) \\ &= \prod_i (1 + (e^{\alpha} - 1)p_i) \\ &\leq \prod_i e^{(e^{\alpha} - 1)p_i} \\ &= e^{(e^{\alpha} - 1) \sum_i p_i} \\ &= e^{(e^{\alpha} - 1)\mu}. \end{aligned}$$

The sneaky inequality step in the middle uses the fact that $(1 + x) \leq e^x$ for all x , which itself is one of the most useful inequalities you can memorize.⁵

What's nice about this derivation is that at the end, the p_i have vanished. We don't care what random variables we started with or how many of them there were, but only about their expected sum μ .

Now that we have an upper bound on $E[e^{\alpha S}]$, we can throw it into

⁵For a proof of this inequality, observe that the function $f(x) = e^x - (1 + x)$ has the derivative $e^x - 1$, which is positive for $x > 0$ and negative for $x < 0$. It follows that $x = 1$ is the unique minimum of f , at which $f(1) = 0$.

Markov's inequality to get the bound we really want:

$$\begin{aligned}
 \Pr[S \geq (1 + \delta)\mu] &= \Pr[e^{\alpha S} \geq e^{\alpha(1+\delta)\mu}] \\
 &\leq \frac{\mathbb{E}[e^{\alpha S}]}{e^{\alpha(1+\delta)\mu}} \\
 &\leq \frac{e^{(e^\alpha - 1)\mu}}{e^{\alpha(1+\delta)\mu}} \\
 &= \left(\frac{e^{e^\alpha - 1}}{e^{\alpha(1+\delta)}} \right)^\mu \\
 &= \left(e^{e^\alpha - 1 - \alpha(1+\delta)} \right)^\mu.
 \end{aligned}$$

We now choose α to minimize the base in the last expression, by minimizing its exponent $e^\alpha - 1 - \alpha(1 + \delta)$. Setting the derivative of this expression with respect to α to zero gives $e^\alpha = (1 + \delta)$ or $\alpha = \ln(1 + \delta)$; luckily, this value of α is indeed greater than 0 as we have been assuming. Plugging this value in gives

$$\begin{aligned}
 \Pr[S \geq (1 + \delta)\mu] &\leq \left(e^{(1+\delta) - 1 - (1+\delta)\ln(1+\delta)} \right)^\mu \\
 &= \left(\frac{e^\delta}{(1 + \delta)^{1+\delta}} \right)^\mu. \tag{5.2.1}
 \end{aligned}$$

The base of this rather atrocious quantity is $e^0/1^1 = 1$ at $\delta = 0$, and its derivative is negative for $\delta \geq 0$ (the easiest way to show this is to substitute $\delta = x - 1$ first). So the bound is never greater than 1 and is both decreasing and less than 1 as soon as $\delta > 0$. We also have that the bound is exponential in μ for any fixed δ .

If we look at the shape of the base as a function of δ , we can observe that when δ is very large, we can replace $(1 + \delta)^{1+\delta}$ with δ^δ without changing the bound much (and to the extent that we change it, it's an increase, so it still works as a bound). This turns the base into $\frac{e^\delta}{\delta^\delta} = (e/\delta)^\delta = 1/(\delta/e)^\delta$. This is pretty close to Stirling's formula for $1/\delta!$ (there is a $\sqrt{2\pi\delta}$ factor missing).

For very small δ , we have that $1 + \delta \approx e^\delta$, so the base becomes approximately $\frac{e^\delta}{e^{\delta(1+\delta)}} = e^{-\delta^2}$. This approximation goes in the wrong direction (it's smaller than the actual value) but with some fudging we can show bounds of the form $e^{-\mu\delta^2/c}$ for various constants c , as long as δ is not too big.

5.2.2 Easier variants

The full Chernoff bound can be difficult to work with, especially since it's hard to invert (5.2.1) to find a good δ that gives a particular ϵ bound. Fortunately, there are approximate variants that substitute a weaker but less intimidating bound. Some of the more useful are:

- For $0 \leq \delta \leq 1.81$,

$$\Pr[X \geq (1 + \delta)\mu] \leq e^{-\mu\delta^2/3}. \quad (5.2.2)$$

(The actual upper limit is slightly higher.) Useful for small values of δ , especially because the bound can be inverted: if we want $\Pr[X \geq (1 + \delta)\mu] \leq \exp(-\mu\delta^2/3) \leq \epsilon$, we can use any δ with $\sqrt{3 \ln(1/\epsilon)/\mu} \leq \delta \leq 1.81$. The essential idea to the proof is to show that, in the given range, $e^\delta/(1 + \delta)^{1+\delta} \leq \exp(-\delta^2/3)$. This is easiest to do numerically; a somewhat more formal argument that the bound holds in the range $0 \leq \delta \leq 1$ can be found in [MU05, Theorem 4.4].

- For $0 \leq \delta \leq 4.11$,

$$\Pr[X \geq (1 + \delta)\mu] \leq e^{-\mu\delta^2/4}. \quad (5.2.3)$$

This is a slightly weaker bound than the previous that holds over a larger range. It gives $\Pr[X \geq (1 + \delta)\mu] \leq \epsilon$ if $\sqrt{4 \ln(1/\epsilon)/\mu} \leq \delta \leq 4.11$. Note that the version given on page 72 of [MR95] is *not correct*; it claims that the bound holds up to $\delta = 2e - 1 \approx 4.44$, but it fails somewhat short of this value.

- For $R \geq 2e\mu$,

$$\Pr[X \geq R] \leq 2^{-R}. \quad (5.2.4)$$

Sometimes the assumption is replaced with the stronger $R \geq 6\mu$ (this is the version given in [MU05, Theorem 4.4], for example); one can also verify numerically that $R \geq 5\mu$ (i.e., $\delta \geq 4$) is enough. The proof of the $2e\mu$ bound is that $e^\delta/(1 + \delta)^{(1+\delta)} < e^{1+\delta}/(1 + \delta)^{(1+\delta)} = (e/(1 + \delta))^{1+\delta} \leq 2^{-(1+\delta)}$ when $e/(1 + \delta) \leq 1/2$ or $\delta \geq 2e - 1$. Raising this to μ gives $\Pr[X \geq (1 + \delta)\mu] \leq 2^{-(1+\delta)\mu}$ for $\delta \geq 2e - 1$. Now substitute R for $(1 + \delta)\mu$ (giving $R \geq 2e\mu$) to get the full result. Inverting this one gives $\Pr[X \geq R] \leq \epsilon$ when $R \geq \min(2e\mu, \lg(1/\epsilon))$.

Figure 5.1 shows the relation between the various bounds, in the region where they cross each other.

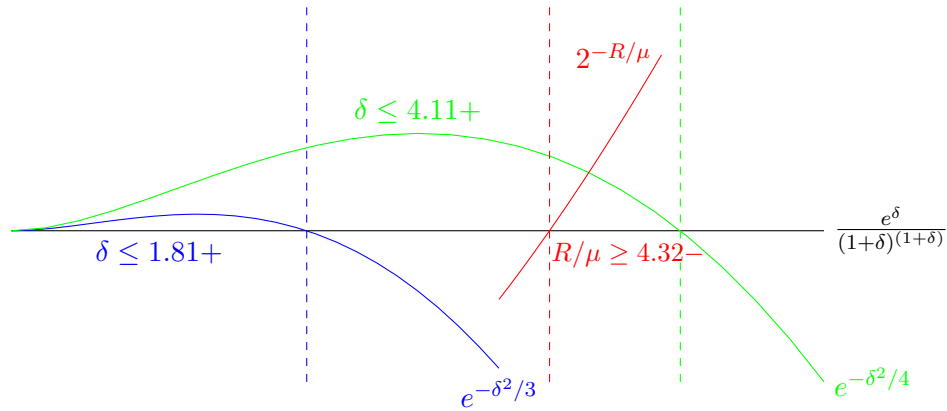


Figure 5.1: Comparison of Chernoff bound variants with exponent μ omitted, plotted in logarithmic scale relative to the standard bound. Each bound is valid only in the region where it exceeds $e^\delta/(1+\delta)^{1+\delta}$.

5.2.3 Lower bound version

We can also use Chernoff bounds to show that a sum of independent 0–1 random variables isn’t too small. The essential idea is to repeat the upper bound argument with a negative value of α , which makes $e^{\alpha(1-\delta)\mu}$ an increasing function in δ . The resulting bound is:

$$\Pr[S \leq (1 - \delta)\mu] \leq \left(\frac{e^{-\delta}}{(1 - \delta)^{1-\delta}} \right)^\mu. \quad (5.2.5)$$

A simpler but weaker version of this bound is

$$\Pr[S \leq (1 - \delta)\mu] \leq e^{-\mu\delta^2/2}. \quad (5.2.6)$$

Both bounds hold for all δ with $0 \leq \delta \leq 1$.

5.2.4 Two-sided version

If we combine (5.2.2) with (5.2.6), we get

$$\Pr[|S - \mu| \geq \delta\mu] \leq 2e^{-\mu\delta^2/3}, \quad (5.2.7)$$

for $0 \leq \delta \leq 1.81$.

Suppose that we want this bound to be less than ϵ . Then we need $2e^{-\delta^2/3} \leq \epsilon$ or $\delta \geq \sqrt{\frac{3 \ln(2/\epsilon)}{\mu}}$. Setting δ to exactly this quantity, (5.2.7)

becomes

$$\Pr \left[|S - \mu| \geq \sqrt{3\mu \ln(2/\epsilon)} \right] \leq \epsilon, \quad (5.2.8)$$

provided $\epsilon \geq 2e^{-\mu/3}$.

For asymptotic purposes, we can omit the constants, giving

Lemma 5.2.1. *Let S be a sum of independent 0–1 variables with $E[S] = \mu$. Then for any $0 < \epsilon \leq 2e^{-\mu/3}$, S lies within $O\left(\sqrt{\mu \log(1/\epsilon)}\right)$ of μ , with probability at least $1 - \epsilon$.*

5.2.5 Almost-independent variables

Chernoff bounds generally don't work very well for variables that are not independent, and in most such cases we must use Chebyshev's inequality (§5.1) or the Azuma-Hoeffding inequality (§5.3) instead. But there is one special case that comes up occasionally where it helps to be able to apply the Chernoff bound to variables that are *almost* independent in a particular technical sense.

Lemma 5.2.2. *Let X_1, \dots, X_n be 0–1 random variables with the property that $E[X_i | X_1, \dots, X_{i-1}] \leq p_i \leq 1$ for all i . Let $\mu = \sum_{i=1}^n p_i$ and $S = \sum_{i=1}^n X_i$. Then (5.2.1) holds.*

Alternatively, let $E[X_i | X_1, \dots, X_{i-1}] \geq p_i \geq 0$ for all i , and let $\mu = \sum_{i=1}^n p_i$ and $S = \sum_{i=1}^n X_i$ as before. Then (5.2.5) holds.

Proof. Rather than repeat the argument for independent variables, we will employ a coupling, where we replace the X_i with independent Y_i so that $\sum_{i=1}^n Y_i$ gives a bound on $\sum_{i=1}^n X_i$.

For the upper bound, let each $Y_i = 1$ with independent probability p_i . Use the following process to generate a new X'_i in increasing order of i : if $Y_i = 0$, set $X'_i = 0$. Otherwise set $X'_i = 1$ with probability $\Pr[X_i = 1 | X_1 = X'_1, \dots, X_{i-1} = X'_{i-1}] / p_i$. Then $X'_i \leq Y_i$, and

$$\begin{aligned} \Pr[X'_i = 1 | X'_1, \dots, X'_i] &= (\Pr[X_i = 1 | X_1 = X'_1, \dots, X_{i-1} = X'_{i-1}] / p_i) \Pr[Y_i = 1] \\ &= \Pr[X_i = 1 | X_1 = X'_1, \dots, X_{i-1} = X'_{i-1}]. \end{aligned}$$

It follows that the X'_i have the same joint distribution as the X_i , and so

$$\begin{aligned} \Pr \left[\sum_{i=1}^n X'_i \geq \mu(1 + \delta) \right] &= \Pr \left[\sum_{i=1}^n X_i \geq \mu(1 + \delta) \right] \\ &\leq \Pr \left[\sum_{i=1}^n Y_i \geq \mu(1 + \delta) \right] \\ &\leq \left(\frac{e^\delta}{(1 + \delta)^{1+\delta}} \right)^\mu. \end{aligned}$$

For the other direction, generate the X_i first and generate the Y_i using the same rejection sampling trick. Now the Y_i are independent (because their joint distribution is) and each Y_i is a lower bound on the corresponding X_i . \square

The lemma is stated for the general Chernoff bounds (5.2.1) and (5.2.5), but the easier versions follow from these, so they hold as well, as long as we are careful to remember that the μ in the upper bound is not necessarily the same μ as in the lower bound.

5.2.6 Other tail bounds for the binomial distribution

The random graph literature can be a good source for bounds on the binomial distribution. See for example [Bol01, §1.3], which uses normal approximation to get bounds that are slightly tighter than Chernoff bounds in some cases, and [JLR00, Chapter 2], which describes several variants of Chernoff bounds as well as tools for dealing with sums of random variables that aren't fully independent.

5.2.7 Applications

5.2.7.1 Flipping coins

Suppose S is the sum of n independent fair coin-flips. Then $E[S] = n/2$ and $\Pr[S = n] = \Pr[S \geq 2E[S]]$ is bounded using (5.2.1) by setting $\mu = n/2$, $\delta = 1$ to get $\Pr[S = n] \leq (e/4)^{n/2} = (2/\sqrt{e})^{-n}$. This is not quite as good as the real answer 2^{-n} (the quantity $2/\sqrt{e}$ is about 1.213...), but it's at least exponentially small.

5.2.7.2 Balls in bins again

Let's try applying the Chernoff bound to the balls-in-bins problem. Here we let $S = \sum_{i=1}^m X_i$ be the number of balls in a particular bin, with X_i the indicator that the i -th ball lands in the bin, $E[X_i] = p_i = 1/n$, and $E[S] = \mu = m/n$. To get a bound on $\Pr[S \geq m/n + k]$, apply the Chernoff bound with $\delta = kn/m$ to get

$$\begin{aligned} \Pr[S \geq m/n + k] &= \Pr[S \geq (m/n)(1 + kn/m)] \\ &\leq \frac{e^k}{(1 + kn/m)^{1+kn/m}}. \end{aligned}$$

For $m = n$, this collapses to the somewhat nicer but still pretty horrifying $e^k/(k+1)^{k+1}$.

Staying with $m = n$, if we are bounding the probability of having large bins, we can use the 2^{-R} variant to show that the probability that any particular bin has more than $2 \lg n$ balls (for example), is at most n^{-2} , giving the probability that there exists such a bin of at most $1/n$. This is not as strong as what we can get out of the full Chernoff bound. If we take the logarithm of $e^k/(k+1)^{k+1}$, we get $k - (k+1) \ln(k+1)$; if we then substitute $k = \frac{c \ln n}{\ln \ln n} - 1$, we get

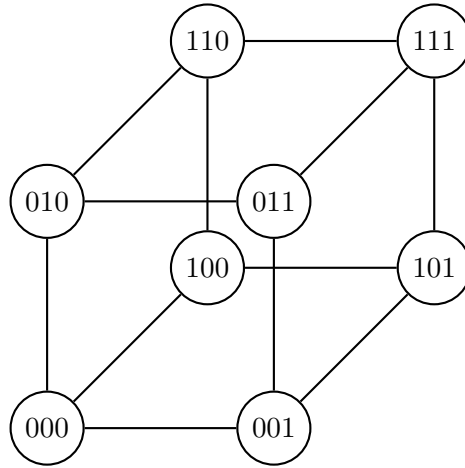
$$\begin{aligned} &\frac{c \ln n}{\ln \ln n} - 1 - \frac{c \ln n}{\ln \ln n} \ln \frac{c \ln n}{\ln \ln n} \\ &= (\ln n) \left(\frac{c}{\ln \ln n} - \frac{1}{\ln n} - \frac{c}{\ln \ln n} (\ln c + \ln \ln n - \ln \ln \ln n) \right) \\ &= (\ln n) \left(\frac{c}{\ln \ln n} - \frac{1}{\ln n} - \frac{c \ln c}{\ln \ln n} - c + \frac{c \ln \ln \ln n}{\ln \ln n} \right) \\ &= (\ln n)(-c + o(1)). \end{aligned}$$

So the probability of getting more than $c \ln n / \ln \ln n$ balls in any one bin is bounded by $\exp((\ln n)(-c + o(1))) = n^{-c+o(1)}$. This gives a maximum bin size of $O(\log n / \log \log n)$ with any fixed probability bound n^{-a} for sufficiently large n .

5.2.7.3 Flipping coins, central behavior

Suppose we flip n fair coins, and let S be the number that come up heads. We expect $\mu = n/2$ heads on average. How many extra heads can we get, if we want to stay within a probability bound of n^{-c} ?

Here we use the small- δ approximation, which gives $\Pr[S \geq (1 + \delta)(n/2)] \leq \exp(-\delta^2 n/6)$. Setting $\exp(-\delta^2 n/6) = n^{-c}$ gives $\delta = \sqrt{6 \ln n^c / n} = \sqrt{6c \ln n / n}$.

Figure 5.2: Hypercube network with $n = 3$

The actual excess over the mean is $\delta(n/2) = (n/2)\sqrt{6c \ln n/n} = \sqrt{\frac{3}{2}cn \ln n}$. By symmetry, the same bound applies to extra tails. So if we flip 1000 coins and see more than 676 heads (roughly the bound when $c=3$), we can reasonably conclude that either (a) our coin is biased, or (b) we just hit a rare one-in-a-billion jackpot.

In algorithm analysis, the $\sqrt{(3/2)c}$ part usually gets absorbed into the asymptotic notation, and we just say that with probability at least $1 - n^{-c}$, the sum of n random bits is within $O(\sqrt{n \log n})$ of $n/2$.

5.2.7.4 Permutation routing on a hypercube

Here we use Chernoff bounds to show bounds on a classic **permutation-routing** algorithm for **hypercube networks** due to Valiant [Val82]. The presentation here is based on §4.2 of [MR95], which in turn is based on an improved version of Valiant's original analysis that appeared in a follow-up paper with Brebner [VB81]. There's also a write-up of this in [MU05, §4.5.1].

The basic idea of a hypercube architecture is that we have a collection of $N = 2^n$ processors, each with an n -bit address. Two nodes are adjacent if their addresses differ by one bit (see Figure 5.2 for an example). Though now mostly of theoretical interest, these things were the cat's pajamas back in the 1980s: see http://en.wikipedia.org/wiki/Connection_Machine.

Suppose that at some point in a computation, each processor i wants to send a packet of data to some processor $\pi(i)$, where π is a permutation of the addresses. But we can only send one packet per time unit along each of

the n edges leaving a processor.⁶ How do we route the packets so that all of them arrive in the minimum amount of time?

We could try to be smart about this, or we could use randomization. Valiant's idea is to first route each process i 's packet to some random intermediate destination $\sigma(i)$, then in the second phase, we route it from $\sigma(i)$ to its ultimate destination $\pi(i)$. Unlike π , σ is not necessarily a permutation; instead, $\sigma(i)$ is chosen uniformly at random independently of all the other $\sigma(j)$. This makes the choice of paths for different packets independent of each other, which we will need later to apply Chernoff bounds.

Routing is done by a **bit-fixing**: if a packet is currently at node x and heading for node y , find the leftmost bit j where $x_j \neq y_j$ and fix it, by sending the packet on to $x[x_j/y_j]$. In the absence of contention, bit-fixing routes a packet to its destination in at most n steps. The hope is that the randomization will tend to spread the packets evenly across the network, reducing the contention for edges enough that the actual time will not be much more than this.

The first step is to argue that, during the first phase, any particular packet is delayed at most one time unit by any other packet whose path overlaps with it. Suppose packet i is delayed by contention on some edge uv . Then there must be some other packet j that crosses uv during this phase. From this point on, j remains one step ahead of i (until its path diverges), so it can't block i again unless both are blocked by some third packet k (in which case we charge i 's further delay to k). This means that we can bound the delays for packet i by counting how many other packets cross its path.⁷ So now we just need a high-probability bound on the number of packets that get in a particular packet's way.

Following the presentation in [MR95], define H_{ij} to be the indicator variable for the event that packets i and j cross paths during the first phase. Because each j chooses its destination independently, once we fix i 's path, the H_{ij} are all independent. So we can bound $S = \sum_{j \neq i} H_{ij}$ using Chernoff bounds. To do so, we must first calculate an upper bound on $\mu = \mathbb{E}[S]$.

The trick here is to observe that any path that crosses i 's path must cross one of its edges, and we can bound the number of such paths by bounding how many paths cross each edge. For each edge e , let T_e be the number of paths that cross edge e , and for each j , let X_j be the number of edges that path j crosses. Counting two ways, we have $\sum_e T_e = \sum_j X_j$, and so

⁶Formally, we have a synchronous routing model with unbounded buffers at each node, with a maximum capacity of one packet per edge per round.

⁷A much more formal version of this argument is given as [MR95, Lemma 4.5].

$E[\sum_e T_e] = E[\sum_j X_j] \leq N(n/2)$. By symmetry, all the T_e have the same expectation, so we get $E[T_e] \leq \frac{N(n/2)}{Nn} = 1/2$.

Now fix $\sigma(i)$. This determines some path $e_1 e_2 \dots e_k$ for packet i . In general we do not expect $E[T_{e_\ell} \mid \sigma(i)]$ to equal $E[T_{e_\ell}]$, because conditioning on i 's path crossing e_ℓ guarantees that at least one path crosses this edge that might not have. However, if we let T'_e be the number of packets $j \neq i$ that cross e , then we have $T'_e \leq T_e$ always, giving $E[T'_e] \leq E[T_e]$, and because T'_e does not depend on i 's path, $E[T'_e \mid \sigma(i)] = E[T'_e] \leq E[T_e] \leq 1/2$. Summing this bound over all $k \leq n$ edges on i 's path gives $E[H_{ij} \mid \sigma(i)] \leq n/2$, which implies $E[H_{ij}] \leq n/2$ after removing the conditioning on $\sigma(i)$.

Inequality (5.2.4) says that $\Pr[X \geq R] \leq 2^{-R}$ when $R \geq 2e\mu$. Letting $X = \sum_{j \neq i} H_{ij}$ and setting $R = 3n$ gives $R = 6(n/2) \geq 6\mu > 2e\mu$, so $\Pr[\sum_j H_{ij} \geq 3n] \leq 2^{-3n} = N^{-3}$. This says that any one packet reaches its random destination with at most $3n$ added delay (thus, in at most $4n$ time units) with probability at least $1 - N^{-3}$. If we consider all N packets, the total probability that any of them fail to reach their random destinations in $4n$ time units is at most $N \cdot N^{-3} = N^{-2}$. Note that because we are using the union bound, we don't need independence for this step—which is good, because we don't have it.

What about the second phase? Here, routing the packets from the random destinations back to the real destinations is just the reverse of routing them from the real destinations to the random destinations. So the same bound applies, and with probability at most N^{-2} some packet takes more than $4n$ time units to get back (this assumes that we hold all the packets before sending them back out, so there are no collisions between packets from different phases).

Adding up the failure probabilities and costs for both stages gives a probability of at most $2/N^2$ that any packet takes more than $8n$ time units to reach its destination.

The structure of this argument is pretty typical for applications of Chernoff bounds: we get a very small bound on the probability that something bad happens by applying Chernoff bounds to a part of the problem where we have independence, then use the union bound to extend this to the full problem where we don't.

5.3 The Azuma-Hoeffding inequality

The problem with Chernoff bounds is that they only work for Bernoulli random variables. **Hoeffding's inequality** is another concentration bound based on the moment generating function that applies to any sum of bounded independent random variables with mean 0.⁸ It has the additional useful feature that it generalizes nicely to some collections of random variables that are not mutually independent, as we will see in §5.3.2. This more general version is known as **Azuma's inequality** or the **Azuma-Hoeffding inequality**.⁹

5.3.1 Hoeffding's inequality

This is the version for sums of bounded independent random variables. We will consider the symmetric case, where each variable X_i satisfies $|X_i| \leq c_i$ for some constant c_i . Hoeffding's original result considered bounds of the form $a_i \leq X_i \leq b_i$, and is equivalent when $a_i = -b_i$.

The main tool is **Hoeffding's lemma**, which states

Lemma 5.3.1. *Let $E[X] = 0$ and $|X| \leq c$ with probability 1. Then*

$$E[e^{\alpha X}] \leq e^{(\alpha c)^2/2}.$$

Proof. The basic idea is that, for any α , $e^{\alpha x}$ is a convex function. Since we want an upper bound, we can't use Jensen's inequality (4.3.1), but we can use the fact that X is bounded and we know its expectation. Convexity of $e^{\alpha x}$ means that, for any x with $-c \leq x \leq c$, $e^{\alpha x} \leq \lambda e^{-\alpha c} + (1 - \lambda)e^{\alpha c}$, where $x = \lambda(-c) + (1 - \lambda)c$. Solving for λ in terms of x gives $\lambda = \frac{1}{2}(1 - \frac{x}{c})$ and $1 - \lambda = \frac{1}{2}(1 + \frac{x}{c})$. So

$$\begin{aligned} E[e^{\alpha X}] &\leq E\left[\frac{1}{2}\left(1 - \frac{X}{c}\right)e^{-\alpha c} + \frac{1}{2}\left(1 + \frac{X}{c}\right)e^{\alpha c}\right] \\ &= \frac{e^{-\alpha c} + e^{\alpha c}}{2} - \frac{e^{-\alpha c}}{2c}E[X] + \frac{e^{\alpha c}}{2c}E[X] \\ &= \frac{e^{-\alpha c} + e^{\alpha c}}{2} \\ &= \cosh(\alpha c). \end{aligned}$$

⁸Note that the requirement that $E[X_i] = 0$ can always be satisfied by considering instead $Y_i = X_i - E[X_i]$.

⁹The history of this is that Hoeffding [Hoe63] proved it for independent random variables, and observed that the proof was easily extended to martingales, while Azuma [Azu67] actually went and did the work of proving it for martingales.

In other words, the worst possible X is a fair choice between $\pm c$, and in this case we get the hyperbolic cosine of αc as its moment generating function.

We don't like hyperbolic cosines much, because we are going to want to take products of our bounds, and hyperbolic cosines don't multiply very nicely. As before with $1 + x$, we'd be much happier if we could replace the cosh with a nice exponential. The Taylor series expansion of $\cosh x$ starts with $1 + x^2/2 + \dots$, suggesting that we should approximate it with $\exp(x^2/2)$, and indeed it is the case that for all x , $\cosh x \leq e^{x^2/2}$. This can be shown by comparing the rest of the Taylor series expansions:

$$\begin{aligned} \cosh x &= \frac{e^x + e^{-x}}{2} \\ &= \frac{1}{2} \left(\sum_{n=0}^{\infty} \frac{x^n}{n!} + \sum_{n=0}^{\infty} \frac{(-x)^n}{n!} \right) \\ &= \sum_{n=0}^{\infty} \frac{x^{2n}}{(2n)!} \\ &\leq \sum_{n=0}^{\infty} \frac{x^{2n}}{2^n n!} \\ &= \sum_{n=0}^{\infty} \frac{(x^2/2)^n}{n!} \\ &= e^{x^2/2}. \end{aligned}$$

This gives the claimed bound

$$\mathbb{E} \left[e^{\alpha X} \right] \leq \cosh(\alpha c) \leq e^{(\alpha c)^2/2}.$$

□

Theorem 5.3.2. *Let $X_1 \dots X_n$ be independent random variables with $\mathbb{E}[X_i] = 0$ and $|X_i| \leq c_i$ for all i . Then for all t ,*

$$\Pr \left[\sum_{i=1}^n X_i \geq t \right] \leq \exp \left(-\frac{t^2}{2 \sum_{i=1}^n c_i^2} \right). \quad (5.3.1)$$

Proof. Let $S = \sum_{i=1}^n X_i$. As with Chernoff bounds, we'll first calculate a bound on the moment generating function $\mathbb{E} \left[e^{\alpha S} \right]$ and then apply Markov's inequality with a carefully-chosen α .

From (5.3.1), we have $\mathbb{E} \left[e^{\alpha X_i} \right] \leq e^{-(\alpha c_i)^2/2}$ for all i . Using this bound and the independence of the X_i , we compute

$$\begin{aligned} \mathbb{E} \left[e^{\alpha S} \right] &= \mathbb{E} \left[\exp \left(\alpha \sum_{i=1}^n X_i \right) \right] \\ &= \mathbb{E} \left[\prod_{i=1}^n e^{\alpha X_i} \right] \\ &= \prod_{i=1}^n \mathbb{E} \left[e^{\alpha X_i} \right] \\ &\leq \prod_{i=1}^n e^{(\alpha c_i)^2/2} \\ &= \exp \left(\sum_{i=1}^n \frac{\alpha^2 c_i^2}{2} \right) \\ &= \exp \left(\frac{\alpha^2}{2} \sum_{i=1}^n c_i^2 \right). \end{aligned}$$

Applying Markov's inequality then gives (when $\alpha > 0$):

$$\begin{aligned} \Pr [S \geq t] &= \Pr \left[e^{\alpha S} \geq e^{\alpha t} \right] \\ &\leq \exp \left(\frac{\alpha^2}{2} \sum_{i=1}^n c_i^2 - \alpha t \right). \end{aligned} \tag{5.3.2}$$

Now we do the same trick as in Chernoff bounds and choose α to minimize the bound. If we write C for $\sum_{i=1}^n c_i^2$, this is done by minimizing the exponent $\frac{\alpha^2}{2} C - \alpha t$, which we do by taking the derivative with respect to α and setting it to zero: $\alpha C - t = 0$, or $\alpha = t/C$. At this point, the exponent becomes $\frac{(t/C)^2}{2} C - (t/C)t = -\frac{t^2}{2C}$.

Plugging this into (5.3.2) gives the bound (5.3.1) claimed in the theorem. \square

5.3.1.1 Hoeffding vs Chernoff

Let's see how good a bound this gets us for our usual test problem of bounding $\Pr [S = n]$ where $S = \sum_{i=1}^n X_i$ is the sum of n independent fair coin-flips. To make the problem fit the theorem, we replace each X_i by a rescaled version $Y_i = 2X_i - 1 = \pm 1$ with equal probability; this makes $\mathbb{E} [Y_i] = 0$ as needed,

with $|Y_i| \leq c_i = 1$. Hoeffding's inequality (5.3.1) then gives

$$\begin{aligned} \Pr \left[\sum_{i=1}^n Y_i \geq n \right] &\leq \exp \left(-\frac{n^2}{2n} \right) \\ &= e^{-n/2} = (\sqrt{e})^{-n}. \end{aligned}$$

Since $\sqrt{e} \approx 1.649\dots$, this is actually slightly better than the $(2/\sqrt{e})^{-n}$ bound we get using Chernoff bounds.

On the other hand, Chernoff bounds work better if we have a more skewed distribution on the X_i ; for example, in the balls-in-bins case, each X_i is a Bernoulli random variable with $\mathbb{E}[X_i] = 1/n$. Using Hoeffding's inequality, we get a bound c_i on $|X_i - \mathbb{E}[X_i]|$ of only $1 - 1/n$, which puts $\sum_{i=1}^n c_i^2$ very close to n , requiring $t = \Omega(\sqrt{n})$ before we get any non-trivial bound out of (5.3.1), pretty much the same as in the fair-coin case (which is not surprising, since Hoeffding's inequality doesn't know anything about the distribution of the X_i). But we've already seen that Chernoff gives us that $\sum X_i = O(\log n / \log \log n)$ with high probability in this case.

5.3.1.2 Asymmetric version

The original version of Hoeffding's inequality [Hoe63] assumes $a_i \leq X_i \leq b_i$, but $\mathbb{E}[X_i]$ is still zero for all X_i . In this version, the bound is

$$\Pr \left[\sum_{i=1}^n X_i \geq t \right] \leq \exp \left(-\frac{2t^2}{\sum_{i=1}^n (b_i - a_i)^2} \right). \quad (5.3.3)$$

This reduces to (5.3.1) when $a_i = -c_i$ and $b_i = c_i$. The proof is essentially the same, but a little more analytic sneakery is required to show that $\mathbb{E} \left[e^{\alpha X_i} \right] \leq e^{\alpha^2 (b_i - a_i)^2 / 8}$; see [McD89] for a proof of this that is a little more approachable than Hoeffding's original paper. For most applications, the only difference between the symmetric version (5.3.1) and the asymmetric version (5.3.3) is a small constant factor on the resulting bound on t .

Hoeffding's inequality is not the tightest possible inequality that can be obtained from the conditions under which it applies, but is relatively simple and easy to work with. For a particularly strong version of Hoeffding's inequality and a discussion of some variants, see [FGQ12].

5.3.2 Azuma's inequality

A general rule of thumb is that most things that work for sums of independent random variables also work for martingales, which are sequences of random variables that have similar behavior but allow for more dependence.

Formally, a **martingale** is a sequence of random variables S_0, S_1, S_2, \dots , where $E[S_t | S_1, \dots, S_{t-1}] = S_{t-1}$. In other words, given everything you know up until time $t - 1$, your best guess of the expected value at time t is just wherever you are now.

Another way to describe a martingale is to take the partial sums $S_t = \sum_{i=1}^t X_i$ of a **martingale difference sequence**, which is a sequence of random variables X_1, X_2, \dots where $E[X_t | X_1 \dots X_{t-1}] = 0$. So in this version, your expected change from time $t - 1$ to t averages out to zero, even if you try to predict it using all the information you have at time $t - 1$.

In some cases it makes sense to allow extra information to sneak in. We can represent this using σ -algebras, in particular by using a filtration of the form $\mathcal{F}_0 \subseteq \mathcal{F}_1 \subseteq \mathcal{F}_2 \subseteq \dots$, where each \mathcal{F}_t is a σ -algebra (see §3.5.3). A sequence S_0, S_1, S_2, \dots is **adapted** to a filtration $\mathcal{F}_0 \subseteq \mathcal{F}_1 \subseteq \mathcal{F}_2 \subseteq \dots$ if each S_t is \mathcal{F}_t -measurable. This means that at time t the sum of our knowledge (\mathcal{F}_t) is enough to predict exactly the value of S_t . The subset relations also mean that we remember everything there is to know about $S_{t'}$ for $t' < t$.

The general definition of a martingale is a collection $\{(S_t, \mathcal{F}_t) | t \in \mathbb{N}\}$ where

1. Each S_t is \mathcal{F}_t -measurable; and
2. $E[S_{t+1} | \mathcal{F}_t] = S_t$.

This means that even if we include any extra information we might have at time t , we still can't predict S_{t+1} any better than by guessing the current value S_t . This alternative definition will be important in some special cases, as when S_t is a function of some other collection of random variables that we use to define the \mathcal{F}_t . Because \mathcal{F}_t includes at least as much information as S_0, \dots, S_t , it will always be the case that any sequence $\{(S_t, \mathcal{F}_t)\}$ that is a martingale in the general sense gives a sequence $\{S_t\}$ that is a martingale in the more specialized $E[S_{t+1} | S_0, \dots, S_t] = S_t$ sense.

Martingales were invented to analyze fair gambling games, where your return over some time interval is not independent of previous outcomes (for example, you may change your bet or what game you are playing depending on how things have been going for you), but it is always zero on average given previous information.¹⁰ The nice thing about martingales is they allow

¹⁰Real casinos give negative expected return, so your winnings in a real casino form a **supermartingale** with $S_t \geq E[S_{t+1} | S_0 \dots S_t]$. On the other hand, the casino's take, in a well-run casino, is a **submartingale**, a process with $S_t \leq E[S_{t+1} | S_0 \dots S_t]$. These definitions also generalize in the obvious way to the $\{(S_t, \mathcal{F}_t)\}$ case.

for a bit of dependence while still acting very much like sums of independent random variables.

Where this comes up with Hoeffding's inequality is that we might have a process that is reasonably well-behaved, but its increments are not technically independent. For example, suppose that a gambler plays a game where she bets x units $0 \leq x \leq 1$ at each round, and receives $\pm x$ with equal probability. Suppose also that her bet at each round may depend on the outcome of previous rounds (for example, she might stop betting entirely if she loses too much money). If X_i is her take at round i , we have that $\mathbb{E}[X_i | X_1 \dots X_{i-1}] = 0$ and that $|X_i| \leq 1$. This is enough to apply the martingale version of Hoeffding's inequality, often called Azuma's inequality.

Theorem 5.3.3. *Let $\{S_k\}$ be a martingale with $S_k = \sum_{i=1}^k X_i$ and $|X_i| \leq c_i$ for all i . Then for all n and all $t \geq 0$:*

$$\Pr[S_n \geq t] \leq \exp\left(\frac{-t^2}{2 \sum_{i=1}^n c_i^2}\right). \quad (5.3.4)$$

Proof. Basically, we just show that $\mathbb{E}[e^{\alpha S_n}] \leq \exp\left(\frac{\alpha^2}{2} \sum_{i=1}^n c_i^2\right)$ —just like in the proof of Theorem 5.3.2—and the rest follows using the same argument. The only tricky part is we can no longer use independence to transform $\mathbb{E}\left[\prod_{i=1}^n e^{\alpha X_i}\right]$ into $\prod_{i=1}^n \mathbb{E}[e^{\alpha X_i}]$.

Instead, we use the martingale property. For each X_i , we have $\mathbb{E}[X_i | X_1 \dots X_{i-1}] = 0$ and $|X_i| \leq c_i$ always. Recall that $\mathbb{E}[e^{\alpha X_i} | X_1 \dots X_{i-1}]$ is a random variable that takes on the average value of $e^{\alpha X_i}$ for each setting of $X_1 \dots X_{i-1}$. We can apply the same analysis as in the proof of 5.3.2 to show that this means that $\mathbb{E}[e^{\alpha X_i} | X_1 \dots X_{i-1}] \leq e^{(\alpha c_i)^2/2}$ always.

The trick is to use the fact that, for any random variables X and Y , $\mathbb{E}[XY] = \mathbb{E}[\mathbb{E}[XY | X]] = \mathbb{E}[X \mathbb{E}[Y | X]]$.

We argue by induction on n that $\mathbb{E}\left[\prod_{i=1}^n e^{\alpha X_i}\right] \leq \prod_{i=1}^n e^{(\alpha c_i)^2/2}$. The

base case is when $n = 0$. For the induction step, compute

$$\begin{aligned}
\mathbb{E} \left[\prod_{i=1}^n e^{\alpha X_i} \right] &= \mathbb{E} \left[\mathbb{E} \left[\prod_{i=1}^n e^{\alpha X_i} \mid X_1 \dots X_{n-1} \right] \right] \\
&= \mathbb{E} \left[\left(\prod_{i=1}^{n-1} e^{\alpha X_i} \right) \mathbb{E} \left[e^{\alpha X_n} \mid X_1 \dots X_{n-1} \right] \right] \\
&\leq \mathbb{E} \left[\left(\prod_{i=1}^{n-1} e^{\alpha X_i} \right) e^{(\alpha c_n)^2/2} \right] \\
&= \mathbb{E} \left[\prod_{i=1}^{n-1} e^{\alpha X_i} \right] e^{(\alpha c_n)^2/2} \\
&\leq \left(\prod_{i=1}^{n-1} e^{(\alpha c_i)^2/2} \right) e^{(\alpha c_n)^2/2} \\
&= \prod_{i=1}^n e^{(\alpha c_i)^2/2}. \\
&= \exp \left(\frac{\alpha^2}{2} \sum_{i=1}^n c_i^2 \right).
\end{aligned}$$

The rest of the proof goes through as before. \square

Some extensions:

- The asymmetric version of Hoeffding's inequality (5.3.3) also holds for martingales. So if each increment X_i satisfies $a_i \leq X_i \leq b_i$ always,

$$\Pr \left[\sum_{i=1}^n X_i \geq t \right] \leq \exp \left(- \frac{2t^2}{\sum_{i=1}^n (b_i - a_i)^2} \right). \quad (5.3.5)$$

- The same bound works for bounded-difference **supermartingales**. A supermartingale is a process where $\mathbb{E}[X_i \mid X_1 \dots X_{i-1}] \leq 0$; the idea is that my expected gain at any step is non-positive, so my present wealth is always superior to my future wealth.¹¹ If $\mathbb{E}[X_i \mid X_1 \dots X_{i-1}] \leq 0$ and $|X_i| \leq c_i$, then we can write $X_i = Y_i + Z_i$ where $Y_i = \mathbb{E}[X_i \mid X_1 \dots X_{i-1}] \leq 0$ is predictable from $X_1 \dots X_{i-1}$ and $\mathbb{E}[Z_i \mid X_1 \dots X_{i-1}] = 0$.¹² Then we can bound $\sum_{i=1}^n X_i$ by observing that it is no greater than $\sum_{i=1}^n Z_i$.

¹¹The corresponding notion in the other direction is a **submartingale**. See §8.2.

¹²This is known as a **Doob decomposition** and can be used to extract a martingale $\{Z_i\}$ from any stochastic process $\{X_i\}$. For general processes, $Y_i = X_i - Z_i$ will still be predictable, but may not satisfy $\mathbb{E}[Y_i \mid X_1, \dots, X_{i-1}] \leq 0$.

A complication is that we no longer have $|Z_i| \leq c_i$; instead, $|Z_i| \leq 2c_i$ (since leaving out Y_i may shift Z_i up). But with this revised bound, (5.3.4) gives

$$\begin{aligned} \Pr \left[\sum_{i=1}^n X_i \geq t \right] &\leq \Pr \left[\sum_{i=1}^n Z_i \geq t \right] \\ &\leq \exp \left(-\frac{t^2}{8 \sum_{i=1}^n c_i^2} \right). \end{aligned} \quad (5.3.6)$$

- Suppose that we stop the process after the first time τ with $S_\tau = \sum_{i=1}^\tau X_i \geq t$. This is equivalent to making a new variable Y_i that is zero whenever $S_{i-1} \geq t$ and equal to X_i otherwise. This doesn't affect the conditions $E[Y_i | Y_1 \dots Y_{i-1}] = 0$ or $|Y_i| \leq c_i$, but it makes it so $\sum_{i=1}^n Y_i \geq t$ if and only if $\max_{k \leq n} \sum_{i=1}^k X_i \geq t$. Applying (5.3.4) to $\sum Y_i$ then gives

$$\Pr \left[\max_{k \leq n} \sum_{i=1}^k X_i \geq t \right] \leq \exp \left(-\frac{t^2}{2 \sum_{i=1}^n c_i^2} \right). \quad (5.3.7)$$

- Since the conditions on X_i in Theorem 5.3.3 apply equally well to $-X_i$, we have

$$\Pr \left[\sum_{i=1}^n X_i \leq -t \right] \leq \exp \left(-\frac{t^2}{2 \sum_{i=1}^n c_i^2} \right). \quad (5.3.8)$$

which we can combine with (5.3.4) to get the two-sided bound

$$\Pr \left[\left| \sum_{i=1}^n X_i \right| \geq t \right] \leq 2 \exp \left(-\frac{t^2}{2 \sum_{i=1}^n c_i^2} \right). \quad (5.3.9)$$

- The extension of Hoeffding's inequality to the case $a_i \leq X_i \leq b_i$ works equally well for Azuma's inequality, giving the same bound as in (5.3.3).
- Finally, one can replace the requirement that each c_i be a constant with a requirement that c_i be predictable from $X_1 \dots X_{i-1}$ and that $\sum_{i=1}^n c_i^2 \leq C$ always and get $\Pr [\sum_{i=1}^n X_i \geq t] \leq e^{-t^2/2C}$. This generally doesn't come up unless you have an algorithm that explicitly cuts off the process if $\sum c_i^2$ gets too big, but there is at least one example of this in the literature [AW96].

5.3.3 The method of bounded differences

To use Azuma's inequality, we need a bounded-difference martingale. The easiest way to get such martingales is through the **method of bounded differences**, which was popularized by a survey paper by McDiarmid [McD89]. For this reason the key result is often referred to as **McDiarmid's inequality**.

The basic idea of the method is to structure a problem so that we are computing a function $f(X_1, \dots, X_n)$ of a sequence of independent random variables X_1, \dots, X_n . To get our martingale, we'll imagine we reveal the X_i one at a time, and compute at each step the expectation of the final value of f based on just the inputs we've seen so far.

Formally, let $\mathcal{F}_t = \langle X_1, \dots, X_t \rangle$, the σ -algebra generated by X_1 through X_t . This represents all the information we have at time t . Let $Y_t = \mathbb{E}[f \mid \mathcal{F}_t]$, the expected value of f given the values of the first t variables. Then $\{\langle Y_t, \mathcal{F}_t \rangle\}$ forms a martingale, with $Y_0 = \mathbb{E}[f]$ and $Y_n = \mathbb{E}[f \mid X_1, \dots, X_n] = f$.¹³ So if we can find a bound c_t on $|Y_t - Y_{t-1}|$, we can apply Azuma's inequality to get bounds on $Y_n - Y_0 = f - \mathbb{E}[f]$.

We do this by assuming that f has the **bounded difference property**, which says that there are bounds c_t such that for any $x_1 \dots x_n$ and any x'_t , we have

$$|f(x_1 \dots x_t \dots x_n) - f(x_1 \dots x'_t \dots x_n)| \leq c_t. \quad (5.3.10)$$

We want to bound $|Y_{t+1} - Y_t| = |\mathbb{E}[f \mid \mathcal{F}_{t+1}] - \mathbb{E}[f \mid \mathcal{F}_t]|$. We can do this showing $Y_{t+1} - Y_t \leq c_{t+1}$. Because f can be replaced by $-f$ without changing the bounded difference property, essentially the same argument will show $Y_{t+1} - Y_t \geq -c_{t+1}$.

Fix some possible value x_{t+1} for X_{t+1} . The bounded difference property says that

$$f(X_1, \dots, X_t, x_{t+1}, X_{t+2}, \dots, X_n) - f(X_1, \dots, X_t, X_{t+1}, X_{t+2}, \dots, X_n)$$

so

$$\mathbb{E}[f(X_1, \dots, X_t, x_{t+1}, X_{t+2}, \dots, X_n) \mid X_1, \dots, X_t] - \mathbb{E}[f(X_1, \dots, X_t, X_{t+1}, X_{t+2}, \dots, X_n) \mid X_1, \dots, X_t] \quad (5.3.11)$$

¹³ A sequence of random variables of the form $Y_t = \mathbb{E}[Z \mid \mathcal{F}_t]$, where $\mathcal{F}_0 \subseteq \mathcal{F}_1 \subseteq \mathcal{F}_2 \subseteq \dots$ is called a **Doob martingale**. The proof that it is a martingale is immediate from the general version of the law of iterated expectation $\mathbb{E}[Y_{t+1} \mid \mathcal{F}_t] = \mathbb{E}[\mathbb{E}[Z \mid \mathcal{F}_{t+1}] \mid \mathcal{F}_t] = \mathbb{E}[Z \mid \mathcal{F}_t] = Y_t$.

The second conditional expectation is just Y_t . What is the first one? Recall

$$\begin{aligned} Y_{t+1} &= \mathbb{E}[f(X_1, \dots, X_n \mid X_1, \dots, X_{t+1})] \\ &= \mathbb{E}[f(x_1, \dots, x_{t+1}, X_1, \dots, X_{t+1})] \end{aligned}$$

when $X_i = x_i$ for all $i \leq t + 1$, since conditioning on X_1, \dots, X_{t+1} just replaces the random variables with their actual values and then averages over the rest. This is the same as

$$\begin{aligned} \mathbb{E}[f(X_1, \dots, X_t, x_{t+1}, X_{t+2}, \dots, X_n \mid X_1, \dots, X_t)] \\ = \mathbb{E}[f(x_1, \dots, x_{t+1}, X_1, \dots, X_{t+1})] \end{aligned}$$

when x_{t+1} happens to be the value of X_{t+1} . So the first conditional expectation in (5.3.11) is just Y_{t+1} , giving

$$Y_{t+1} - Y_t \leq c_{t+1}.$$

Now we can apply Azuma-Hoeffding to get $\Pr[Y_n - Y_0 \geq t] \leq \exp\left(-\frac{t^2}{2\sum_{i=1}^n c_i^2}\right)$.

This turns out to overestimate the possible range of Y_{t+1} . With a more sophisticated argument, it can be shown that for any fixed x_1, \dots, x_t , there exist bounds $a_{t+1} \leq Y_{t+1} - Y_t \leq b_{t+1}$ such that $b_{t+1} - a_{t+1} = c_{t+1}$. We would like to use this to apply the asymmetric version of Azuma-Hoeffding given in (5.3.5). A complication is that the specific values of a_{t+1} and b_{t+1} depend on the previous values x_1, \dots, x_t , even though their difference $b_{t+1} - a_{t+1}$ does not. Fortunately, McDiarmid shows that the inequality works anyway, giving:

Theorem 5.3.4 (McDiarmid's inequality [McD89]). *Let X_1, \dots, X_n be independent random variables and let $f(X_1, \dots, X_n)$ have the bounded difference property with bounds c_i . Then*

$$\Pr[f(X_1, \dots, X_n) - \mathbb{E}[f(X_1, \dots, X_n)] \geq t] \leq \exp\left(-\frac{2t^2}{\sum_{i=1}^n c_i^2}\right). \quad (5.3.12)$$

The main difference between this and a direct application of Azuma-Hoeffding is that the constant factor in the exponent is better by a factor of 4.

Since $-f$ satisfies the same constraints as f , the same bound holds for $\Pr[f(X_1, \dots, X_n) - \mathbb{E}[f(X_1, \dots, X_n)] \leq -t]$. For some applications it may

make sense to apply the union bound to get a two-sided version

$$\Pr [|f(X_1, \dots, X_n) - \mathbb{E}[f(X_1, \dots, X_n)]| \geq t] \leq 2 \exp\left(-\frac{2t^2}{\sum_{i=1}^n c_i^2}\right). \quad (5.3.13)$$

5.3.4 Applications

Here are some applications of the preceding inequalities. Most of these are examples of the method of bounded differences.

5.3.4.1 Sprinkling points on a hypercube

Suppose you live in a hypercube, and the local government has conveniently placed mailboxes on some subset A of the nodes. If you start at a random location, how likely is it that your distance to the nearest mailbox deviates substantially from the average distance?

We can describe your position as a bit vector X_1, \dots, X_n , where each X_i is an independent random bit. Let $f(X_1, \dots, X_n)$ be the distance from X_1, \dots, X_n to the nearest element of A . Then changing one of the bits changes this function by at most 1. So we have $\Pr [|f - \mathbb{E}[f]| \geq t] \leq 2e^{-2t^2/n}$ by (5.3.12), giving a range of possible distances that is $O(\sqrt{n \log n})$ with probability at least $1 - n^{-c}$ for any fixed $c > 0$.¹⁴ Of course, without knowing what A is, we don't know what $\mathbb{E}[f]$ is; but at least we can be assured that (unless A is very big) the distance we have to walk to send our mail will be pretty much the same pretty much wherever we start.

5.3.4.2 Chromatic number of a random graph

Consider a **random graph** $G(n, p)$ consisting of n vertices, where each possible edge appears with independent probability p . Let χ be the **chromatic number** of this graph, the minimum number of colors necessary if we want to assign a color to each vertex that is distinct for the colors of all of its neighbors. The **vertex exposure martingale** shows us the vertices of the graph one at a time, along with all the edges between vertices that have been exposed so far. We define X_t to be the expected value of χ given this information for vertices $1 \dots t$.

If Z_i is a random variable describing which edges are present between i and vertices less than i , then the Z_i are all independent, and we can write

¹⁴Proof: Let $t = \sqrt{\frac{1}{2}(c+1)n \ln n} = O(\sqrt{n \log n})$. Then $2e^{-2t^2/n} = 2e^{-(c+1) \ln n} = 2n^{-c-1} < n^{-c}$ when n is sufficiently large.

$\chi = f(Z_1, \dots, Z_n)$ for some function f (this function may not be very easy to compute, but it exists). Then X_t as defined above is just $E[f \mid Z_1, \dots, Z_t]$.

Now observe that f has the bounded difference property with $c_t = 1$: if I change the edges for some vertex v_t , I can't increase the number of colors I need by more than 1, since in the worst case I can always take whatever coloring I previously had for all the other vertices and add a new color for v_t . This implies that the difference between any two graphs G and G' that differ only in the value of some Z_t is at most one, because going between them is an increase on one direction or the other.

McDiarmid's inequality (5.3.12) then says that $\Pr[|\chi - E[\chi]| \geq t] \leq 2e^{-2t^2/n}$; in other words, the chromatic number of a random graph is tightly concentrated around its mean, even if we don't know what that mean is. (This proof is due to Shamir and Spencer [SS87].)

5.3.4.3 Balls in bins

Suppose we toss m balls into n bins. How many empty bins do we get? The probability that each bin individually is empty is exactly $(1 - 1/n)^m$, which is approximately $e^{-m/n}$ when n is large. So the expected number of empty bins is exactly $n(1 - 1/n)^m$. If we let X_i be the bin that ball i gets tossed into, and let $Y = f(X_1, \dots, X_m)$ be the number of empty bins, then changing a single X_i can change f by at most 1. So from (5.3.12) we have $\Pr[Y \geq n(1 - 1/n)^m + t] \leq e^{-2t^2/m}$.

5.3.4.4 Probabilistic recurrence relations

Most probabilistic recurrence arguments (as in Appendix H) can be interpreted as supermartingales: the current estimate of $T(n)$ is always at least the expected estimate after doing one stage of the recurrence. This fact can be used to get concentration bounds using (5.3.6).

For example, let's take the recurrence (1.3.1) for the expected number of comparisons for QuickSort:

$$T(n) = \frac{1}{n} \sum_{k=0}^{n-1} (T(k) + T(n-1-k)).$$

We showed in §1.3.1 that the solution to this recurrence satisfies $T(n) \leq 2n \ln n$.

To turn this into a supermartingale, imagine that we carry out a process where we keep around at each step t a set of unsorted blocks of size

$n_1^t, n_2^t, \dots, n_{k_t}^t$ for some k_t (note that the superscripts on n_i^t are not exponents). One step of the process involves choosing one of the blocks (we can do this arbitrarily without affecting the argument) and then splitting that block around a uniformly-chosen pivot. We will track a random variable X_t equal to $C_t + \sum_{i=1}^{k_t} 2n_i^t \ln n_i^t$, where C_t is the number of comparisons done so far and the summation gives an upper bound on the expected number of comparisons remaining.

To show that this is in fact a supermartingale, observe that if we partition a block of size n we add n to C_t but replace the cost bound $2n \ln n$ by an expected

$$\begin{aligned} 2 \cdot \frac{1}{n} \sum_{k=0}^{n-1} 2k \ln k &\leq \frac{4}{n} \int_2^n n \ln n \\ &= \frac{4}{n} \left(\frac{n^2 \ln n}{2} - \frac{n^2}{4} - \ln 2 + 1 \right) \\ &= 2n \ln n - n - \ln 2 + 1 \\ &< 2n \ln n - n. \end{aligned}$$

The net change is less than $-\ln 2$. The fact that it's not zero suggests that we could improve the $2n \ln n$ bound slightly, but since it's going down, we have a supermartingale.

Let's try to get a bound on how much X_t changes at each step. The C_t part goes up by at most $n - 1$. The summation can only go down; if we split a block of size n_i , the biggest drop we get is if we split it evenly,¹⁵ This gives a drop of

$$\begin{aligned} 2n \ln n - 2 \left(2 \frac{n-1}{2} \ln \frac{n-1}{2} \right) &= 2n \ln n - 2(n-1) \ln \left(n \frac{n-1}{2n} \right) \\ &= 2n \ln n - 2(n-1) \left(\ln n - \ln \frac{2n}{n-1} \right) \\ &= 2n \ln n - 2n \ln n + 2n \ln \frac{2n}{n-1} + 2 \ln n - 2 \ln \frac{2n}{n-1} \\ &= 2n \cdot O(1) + O(\log n) \\ &= O(n). \end{aligned}$$

(with a constant tending to 2 in the limit).

¹⁵This can be proven most easily using convexity of $n \ln n$.

So we can apply (5.3.6) with $c_t = O(n)$ to the at most n steps of the algorithm, and get

$$\Pr [C_n - 2n \ln n \geq t] \leq e^{-t^2/O(n^3)}.$$

This gives $C_n = O(n^{3/2})$ with constant probability or $O(n^{3/2}\sqrt{\log n})$ with all but polynomial probability. This is a rather terrible bound, but it's a lot better than $O(n^2)$.

Much tighter bounds are known: QuickSort in fact uses $\Theta(n \log n)$ comparisons with high probability [MH92]. If we aren't too worried about constants, an easy way to see the upper bound side of this is to adapt the analysis of Hoare's FIND (§3.6.4). For each element, the number of other elements in the same block is multiplied by a factor of at most $3/4$ on average each time the element is compared, so the chance that the element is not by itself is at most $(3/4)^k n$ after k comparisons. Setting $k = \log_{4/3}(n^2/\epsilon)$ gives that any particular element is compared k or times with probability at most ϵ/n . The union bound then gives a probability of at most ϵ that the most-compared element is compared k or more times. So the total number of comparisons is $O(\log(n/\epsilon))$ with probability $1 - \epsilon$, which becomes $O(\log n)$ with probability $1 - n^{-c}$ if we set $\epsilon = n^{-c}$ for a fixed c .

5.3.4.5 Multi-armed bandits

In the **multi-armed bandit** problem, we must choose at each time step one of a fixed set of k **arms** to pull. Pulling arm i at time t yields a return of X_i^t , a random payoff typically assumed to be between 0 and 1. Suppose that all the X_i^t are independent, and that for each fixed i , all X_i^t have the same distribution, and thus the same expected payoff. Suppose also that we initially know nothing about these distributions. What strategy can we use to maximize our expected payoff over a large number of pulls?

More specifically, how can we minimize our **regret**, defined as

$$T\mu^* - \sum_{i=1}^k \mathbb{E}[X_i] T_i, \quad (5.3.14)$$

where μ^* is the expected payoff of the best arm, and T_i counts the number of times we pull arm i ?

The tricky part here is that when we pull an arm and get a bad return, we don't know if we were just unlucky this time or it's actually a bad arm. So we have an incentive to try lots of different arms. On the other hand, the more we pull a genuinely inferior arm, the worse our overall return. We'd like

to adopt a strategy that trades off between exploration (trying new arms) and exploitation (collecting on the best arm so far) to do as best we can in comparison to a strategy that always pulls the best arm.

The UCB1 algorithm Fortunately, there is a simple algorithm due to Auer *et al.* [ACBF02] that solves this problem for us.¹⁶ To start with, we pull each arm once. For any subsequent pull, suppose that for each i , we have pulled the i -th arm n_i times so far. Let $n = \sum n_i$, and let \bar{x}_i be the average payoff from arm i so far. Then the **UCB1** algorithm pulls the arm that maximizes

$$\bar{x}_i + \sqrt{\frac{2 \ln n}{n_i}}. \quad (5.3.15)$$

UCB stands for **upper confidence bound**. (The “1” is because it is the first, and simplest, of several algorithms of this general structure given in the paper.) The idea is that we give arms that we haven’t tried a lot the benefit of the doubt, and assume that their actual average payout lies at the upper end of some plausible range of values.¹⁷

The quantity $\sqrt{\frac{2 \ln n}{n_i}}$ is a bit mysterious, but it arises in a fairly natural way from the asymmetric version of Hoeffding’s inequality. With a small adjustment to deal with non-zero-mean variables, (5.3.3) says that, if S is a sum of n random variables bounded between a_i and b_i , then

$$\Pr \left[\sum_{i=1}^n (X_i - \mathbb{E}[X_i]) \geq t \right] \leq e^{-2t^2 / \sum_{i=1}^n (b_i - a_i)^2}. \quad (5.3.16)$$

By applying (5.3.3) to $-X_i$, we also get

$$\Pr \left[\sum_{i=1}^n (X_i - \mathbb{E}[X_i]) \leq -t \right] \leq e^{-2t^2 / \sum_{i=1}^n (b_i - a_i)^2}. \quad (5.3.17)$$

Now consider $\bar{x}_i = \frac{1}{n_i} \sum_{j=1}^{n_i} X_j$ where each X_j lies between 0 and 1. This is equivalent to having $\bar{x}_i = \sum_{j=1}^{n_i} Y_j$ where $Y_j = X_j/n_i$ lies between 0 and

¹⁶This is not the only algorithm for solving multi-armed bandit problems, and it’s not even the only algorithm in the Auer *et al.* paper. But it has the advantage of being relatively straightforward to analyze. For a more general survey of multi-armed bandit algorithms for the regret model, see [BCB12].

¹⁷The “bound” part is because we don’t attempt to compute the exact upper end of this confidence interval, which may be difficult, but instead use an upper bound derived from Hoeffding’s inequality. This distinguishes the UCB1 algorithm of [ACBF02] from the *upper confidence interval* approach of Lai and Robbins [LR85] that it builds on.

$1/n_i$. So (5.3.16) says that

$$\begin{aligned} \Pr \left[\bar{x}_i - \mathbb{E}[\bar{x}_i] \geq \sqrt{\frac{2 \ln n}{n_i}} \right] &\leq e^{-2(\sqrt{(2 \ln n)/n_i})^2 / (n_i(1/n_i)^2)} \\ &= e^{-4 \ln n} \\ &= n^{-4}. \end{aligned} \tag{5.3.18}$$

We also get a similar lower bound using (5.3.17).

So the bonus applied to \bar{x}_i is really a high-probability bound on how big the difference between the observed payoff and expected payoff might be. The $\sqrt{\ln n}$ part is there to make the error probability be small as a function of n , since we will be summing over a number of bad cases polynomial in n and not a particular n_i . Applying the bonus to all arms make it likely that the observed payoff of the best arm stays above its actual payoff, so we won't forget to pull it. The hope is that over time the same bonus applied to other arms will not boost them up so much that we pull them any more than we have to.

However, in an infinite execution of the algorithm, even a bad arm will be pulled infinitely often, as $\ln n$ rises enough to compensate for the last increase in n_i . This accounts for an $O(\log n)$ term in the regret, as we will see below. It also prevents us from getting stuck refusing to give a good arm a second chance just because we had an initial run of bad luck.

Analysis of UCB1 The following theorem is a direct quote of [ACBF02, Theorem 1]:

Theorem 5.3.5 ([ACBF02]). *For all $K > 1$, if policy UCB1 is run on K machines having arbitrary reward distributions P_1, \dots, P_K with support in $[0, 1]$, then its expected regret after any number n of plays is at most*

$$\left[8 \sum_{i: \mu_i < \mu^*} \frac{\ln n}{\Delta_i} \right] + \left(1 + \frac{\pi^2}{3} \right) \left(\sum_{j=1}^K \Delta_j \right). \tag{5.3.19}$$

Here $\mu_i = \mathbb{E}[P_i]$ is the expected payoff for arm i , μ^* as before is $\max_i \mu_i$, and $\Delta_i = \mu^* - \mu_i$ is the regret for pulling arm i . The theorem states that our expected regret is bounded by a term for each arm worse than μ^* that grows logarithmically in n and is inversely proportional to how close the arm is to optimal, plus a constant additional loss corresponding to pulling every arm a little more than 4 times on average. The logarithmic regret in n is

a bit of a nuisance, but an earlier lower bound of Lai and Robbins [LR85] shows that something like this is necessary in the limit.

To prove Theorem 5.3.5, we need to get an upper bound on the number of times each suboptimal arm is pulled during the first n pulls. Define

$$c_{t,s} = \sqrt{\frac{2 \ln t}{s}}, \quad (5.3.20)$$

the bonus given to an arm that has been pulled s times in the first t pulls.

Fix some optimal arm. Let $\bar{X}_{i,s}$ be the average return on arm i after s pulls and \bar{X}_s^* be the average return on this optimal arm after s pulls.

If we pull arm i after t total pulls, when arm i has previously been pulled s_i times and our optimal arm has been pulled s^* times, then we must have

$$\bar{X}_{i,s_i} + c_{t,s_i} \geq \bar{X}_{s^*}^* + C_{t,s^*}. \quad (5.3.21)$$

This just says that arm i with its bonus looks better than the optimal arm with its bonus.

To show that this bad event doesn't happen, we need three things:

1. The value of $\bar{X}_{s^*}^* + c_{t,s^*}$ should be at least μ^* . Were it to be smaller, the observed value $\bar{X}_{s^*}^*$ would be more than c_{t,s^*} away from its expectation. Hoeffding's inequality implies this doesn't happen too often.
2. The value of $\bar{X}_{i,s_i} + c_{t,s_i}$ should not be too much bigger than μ_i . We'll again use Hoeffding's inequality to show that \bar{X}_{i,s_i} is likely to be at most $\mu_i + c_{t,s_i}$, making $\bar{X}_{i,s_i} + c_{t,s_i}$ at most $\mu_i + 2c_{t,s_i}$.
3. The bonus c_{t,s_i} should be small enough that even adding $2c_{t,s_i}$ to μ_i is not enough to beat μ^* . This means that we need to pick s_i large enough that $2c_{t,s_i} \leq \Delta_i$. For smaller values of s_i , we will just accept that we need to pull arm i a few more times before we wise up.

More formally, if none of the following events hold:

$$\bar{X}_{s^*}^* + c_{t,s^*} \leq \mu^*. \quad (5.3.22)$$

$$\bar{X}_{i,s_i} \geq \mu_i + c_{t,s_i} \quad (5.3.23)$$

$$\mu^* - \mu_i < 2c_{t,s_i}, \quad (5.3.24)$$

then $\bar{X}_{s^*}^* + c_{t,s^*} > \mu^* > \mu_i + 2c_{t,s_i} > \bar{X}_{i,s_i}$, and we don't pull arm i because the optimal arm is better. (We don't necessarily pull the optimal arm, but if we don't, it's because we pull some other arm that still isn't arm i .)

For (5.3.22) and (5.3.23), we repeat the argument in (5.3.18), plugging in t for n and s_i or s^* for n_i . This gives a probability of at most $2t^{-4}$ that either or both of these bad events occur.

For (5.3.24), we need to do something a little sneaky, because the statement is not actually true when s_i is small. So we will give ℓ free pulls to arm i , and only start comparing arm i to the optimal arm after we have done at least this many pulls. The value of ℓ is chosen so that, when $t \leq n$ and $s_i > \ell$,

$$2c_{t,s_i} \leq \mu^* - \mu_i,$$

which expands to,

$$2\sqrt{\frac{2 \ln t}{s_i}} \leq \Delta_i,$$

giving

$$s_i \geq \frac{8 \ln t}{\Delta_i^2}.$$

So we must set ℓ to be at least

$$\frac{8 \ln n}{\Delta_i^2} \geq \frac{8 \ln t}{\Delta_i^2}.$$

Because ℓ must be an integer, we actually get

$$\ell = \lceil \frac{8 \ln n}{\Delta_i^2} \rceil \leq 1 + \frac{8 \ln n}{\Delta_i^2}.$$

This explains (after multiplying by the regret Δ_i) the first term in (5.3.19).

For the other sources of bad pulls, apply the union bound to the $2t^{-4}$ error probabilities we previously computed for all choices of $t \leq n$, $s^* \geq 1$, and $s_i > \ell$. This gives

$$\begin{aligned} \sum_{t=1}^n \sum_{s^*=1}^{t-1} \sum_{s_i=\ell+1}^{t-1} 2t^{-4} &< 2 \sum_{t=1}^{\infty} t^2 \cdot t^{-4} \\ &= 2 \cdot \frac{\pi^2}{6} \\ &= \frac{\pi^2}{3}. \end{aligned}$$

Again we have to multiply by the regret Δ_i for pulling the i -th arm, which gives the second term in (5.3.19).

5.4 Relation to limit theorems

Since in many cases we are working with sums $S_n = \sum_{i=1}^n X_i$ of independent, identically distributed random variables X_i , classical limit theorems apply. These relate the behavior of S_n in the limit to the common expectation $\mu = E[X_i]$ and variance $\sigma^2 = \text{Var}[X_i]$.¹⁸

These include the **strong law of large numbers**

$$\Pr \lim_{n \rightarrow \infty} S_n/n = \mu = 1 \quad (5.4.1)$$

and the **central limit theorem (CLT)**

$$\lim_{n \rightarrow \infty} \Pr \left[\frac{S_n - \mu n}{\sigma \sqrt{n}} \leq t \right] = \Phi(t), \quad (5.4.2)$$

where Φ is the normal distribution function.

These can sometimes be useful in the analysis of randomized algorithms, but often are not strong enough to get the results we want. The main problem is that the standard versions of both the strong law and the central limit theorem say nothing about rate of convergence. So if we want (for example) to use the CLT to show that S_n is exponentially unlikely to be $n\sigma$ away from the mean, we can't do it directly, because $n\sigma/\sigma$ is not a fixed constant t , and for any fixed constant t , we don't know when the limit behavior actually starts working.

But there are variants of these theorems that do bound rate of convergence that can be useful in some cases. An example is given in §5.5.1.

5.5 Anti-concentration bounds

It may be that for some problem you want to show that a sum of random variables is far from its mean at least some of the time: this would be an **anti-concentration bound**. Anti-concentration bounds are much less well-understood than concentration bounds, but there are known results that can help in some cases.

For variables where we know the distribution of the sum exactly (e.g., sums with binomial distributions, or sums we can attack with generating functions), we don't need these. But they may be useful if computing the distribution of the sum directly is hard.

¹⁸The quantity $\sigma = \sqrt{\text{Var}[X]}$ is called the **standard deviation** of X , and informally gives a measure of the typical distance of X from its mean, measured in the same units as X .

5.5.1 The Berry-Esseen theorem

The **Berry-Esseen theorem**¹⁹ is an extension of the central limit theorem that characterizes how quickly a sum of independent identically-distributed random variables converges to a normal distribution, as a function of the **third moment** of the random variables. Its simplest version says that if we have n independent, identically-distributed random variables $X_1 \dots X_n$, with $E[X_i] = 0$, $\text{Var}[X_i] = E[X_i^2] = \sigma^2$, and $E[|X_i|^3] \leq \rho$, then

$$\sup_{-\infty < x < \infty} \left| \Pr \left[\frac{1}{\sqrt{n}} \sum_{i=1}^n X_i \leq x \right] - \Phi(x) \right| \leq \frac{C\rho}{\sigma^3\sqrt{n}}, \quad (5.5.1)$$

were C is an absolute constant and Φ is the normal distribution function. Note that the σ^3 in the denominator is really $\text{Var}[X_i]^{3/2}$. Since the probability bound doesn't depend on x , it's more useful toward the middle of the distribution than in the tails.

A classic proof of this result with $C = 3$ can be found in [Fel71, §XVI.5]. More recent work has tightened the bound on C : the best currently known constant is $C < 0.4784 < 1/2$, due to Shevtsova [She11].

As with most results involving sums of random variables, there are generalizations to martingales. These are too involved to describe here, but see [HH80, §3.6].

5.5.2 The Littlewood-Offord problem

The **Littlewood-Offord problem** asks, given a set of n complex numbers $x_1 \dots x_n$ with $|x_i| \geq 1$, for how many assignments of ± 1 to coefficients $\epsilon_1 \dots \epsilon_n$ it holds that $|\sum_{i=1}^n \epsilon_i x_i| \leq r$. Paul Erdős showed [Erd45] that this quantity was at most $cr2^n/\sqrt{n}$, where c is a constant. The quantity $c2^n/\sqrt{n}$ here is really $\frac{1}{2} \binom{n}{\lfloor n/2 \rfloor}$: Erdős's proof shows that for each interval of length $2r$, the number of assignments that give a sum in the interior of the interval is bounded by at most the sum of the r largest binomial coefficients.

In random-variable terms, this means that if we are looking at $\sum_{i=1}^n \epsilon_i x_i$, where the x_i are constants with $|x_i| \geq 1$ and the ϵ_i are independent ± 1 fair coin-flips, then $\Pr[|\sum_{i=1}^n \epsilon_i x_i| \leq r]$ is maximized by making all the x_i equal to 1. This shows that any distribution where the x_i are all reasonably large will not be any more concentrated than a binomial distribution.

There has been a lot of more recent work on variants of the Littlewood-Offord problem, much of it by Terry Tao and Van Vu. See <http://terrytao>.

¹⁹Sometimes written *Berry-Esséen theorem* to help with the pronunciation of Esseen's last name.

wordpress.com/2009/02/16/a-sharp-inverse-littlewood-offord-theorem/
for a summary of much of this work.

Chapter 6

Randomized search trees

Last updated 2019. Some material may be out of date.

These are data structures that are either trees or equivalent to trees, and use randomization to maintain balance. We'll start by reviewing deterministic binary search trees and then add in the randomization.

6.1 Binary search trees

A **binary search tree** is a standard data structure for holding sorted data. A **binary tree** is either empty, or it consists of a **root** node containing a **key** and pointers to left and right **subtrees**. What makes a binary tree a binary search tree is the invariant that, both for the tree as a whole and any subtree, all keys in the left subtree are less than the key in the root, while all keys in the right subtree are greater than the key in the root. This ordering property means that we can search for a particular key by doing binary search: if the key is not at the root, we can recurse into the left or right subtree depending on whether it is smaller or bigger than the key at the root.

The efficiency of this operation depends on the tree being **balanced**. If each subtree always holds a constant fraction of the nodes in the tree, then each recursive step throws away a constant fraction of the remaining nodes. So after $O(\log n)$ steps, we find the key we are looking for (or find that the key is not in the tree). But the definition of a binary search tree does not by itself guarantee balance, and in the worst case a binary search tree degenerates into a linked list with $O(n)$ cost for all operations (see Figure 6.2).

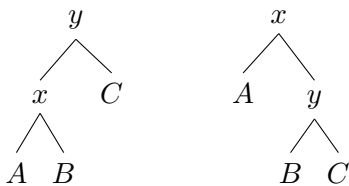


Figure 6.1: Tree rotations

6.1.1 Rebalancing and rotations

Deterministic binary search tree implementations include sophisticated rebalancing mechanisms to adjust the structure of the tree to preserve balance as nodes are inserted or delete. Typically this is done using **rotations**, which are operations that change the position of a parent and a child while preserving the left-to-right ordering of keys (see Figure 6.1).

Examples include **AVL trees** [AVL62], where the left and right subtrees of any node have heights that differ by at most 1; **red-black trees** [GS78], where a coloring scheme is used to maintain balance; and **scapegoat trees** [GR93], where no information is stored at a node but part of the tree is rebuilt from scratch whenever an operation takes too long. These all give $O(\log n)$ cost per operation (amortized in the case of scapegoat trees), and vary in how much work is needed in rebalancing. Both AVL trees and red-black trees perform more rotations than randomized rebalancing does on average.

6.2 Random insertions

Suppose we insert n keys into an initially-empty binary search tree in random order with no rebalancing. This means that for each insertion, we follow the same path that we would when searching for the key, and when we reach an empty tree, we replace it with a tree consisting solely of the key at the root.¹

Since we chose a random order, each element is equally likely to be the root, and all the elements less than the root end up in the left subtree, while all the elements greater than the root end up in the right subtree, where they are further partitioned recursively. This is exactly what happens in

¹This is not the only way to generate a binary search tree at random. For example, we could instead choose uniformly from the set of all C_n binary search trees with n nodes, where $C_n = \frac{1}{n+1} \binom{2n}{n}$ is the n -th **Catalan number**. For $n \geq 3$, this gives a different distribution that we don't care about.

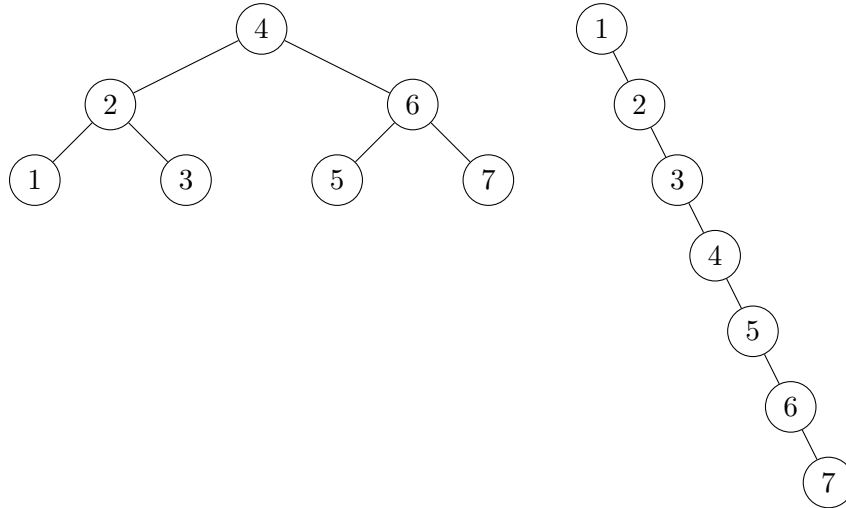


Figure 6.2: Balanced and unbalanced binary search trees

randomized QuickSort (see §1.3.1), so the structure of the tree will exactly mirror the structure of an execution of QuickSort. So, for example, we can immediately observe from our previous analysis of QuickSort that the **total path length**—the sum of the depths of the nodes—is $\Theta(n \log n)$, since the depth of each node is equal to 1 plus the number of comparisons it participates in as a non-pivot, and (using the same argument as for Hoare’s FIND in §3.6.4) that the height of the tree is $O(\log n)$ with high probability.²

When n is small, randomized binary search trees can look pretty scraggly. Figure 6.3 shows a typical example.

The problem with this approach in general is that we don’t have any guarantees that the input will be supplied in random order, and in the worst case we end up with a linked list, giving $O(n)$ worst-case cost for all operations.

²The argument for Hoare’s FIND is that any node has at most $3/4$ of the descendants of its parent on average; this gives for any node x that $\Pr[\text{depth}(x) > d] \leq (3/4)^{d-1}$, or a probability of at most n^{-c} that $\text{depth}(x) > 1 + (c+1) \log(n) / \log(4/3) \approx 1 + 6.952 \ln n$ for $c = 1$, which we need to apply the union bound. The right answer for the actual height of a randomly-generated search tree in the limit is $4.31107 \ln n$ [Dev88] so this bound is actually pretty close. The real height is still nearly a factor of three worse than for a completely balanced tree, which has max depth bounded by $1 + \lg n \approx 1 + 1.44269 \ln n$.

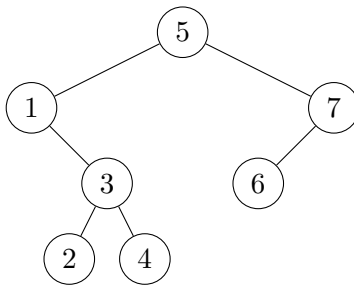


Figure 6.3: Binary search tree after inserting 5 1 7 3 4 6 2

6.3 Treaps

The solution to bad inputs is the same as for QuickSort: instead of assuming that the input is permuted randomly, we assign random priorities to each element and organize the tree so that elements with higher priorities rise to the top. The resulting structure is known as a **treap** [SA96], because it satisfies the binary search tree property with respect to keys and the **heap** property with respect to priorities.³

There’s an extensive page of information on treaps at <http://faculty.washington.edu/aragon/treaps.html>, maintained by Cecilia Aragon, the co-inventor of treaps; they are also discussed at length in [MR95, §8.2]. We’ll give a brief description here.

To insert a new node in a treap, first walk down the tree according to the key and insert the node as a new leaf. Then go back up fixing the heap property by rotating the new element up until it reaches an ancestor with the same or higher priority. (See Figure 6.4 for an example.) Deletion is the reverse of insertion: rotate a node until it has 0 or 1 children (by swapping with its higher-priority child at each step), and then prune it out, connecting its child, if any, directly to its parent.

Because of the heap property, the root of each subtree is always the element in that subtree with the highest priority. This means that the structure of a treap is completely determined by the priorities and the keys, no matter what order the elements arrive in. We can imagine in retrospect

³The name “treap” for this data structure is now standard but the history is a little tricky. According to Seidel and Aragon, essentially the same data structure (though with non-random priorities) was previously called a cartesian tree by Vuillemin [Vui80], and the word “treap” was initially applied by McCreight to a different data structure—designed for storing two-dimensional data—that was called a “priority search tree” in its published form. [McC85].

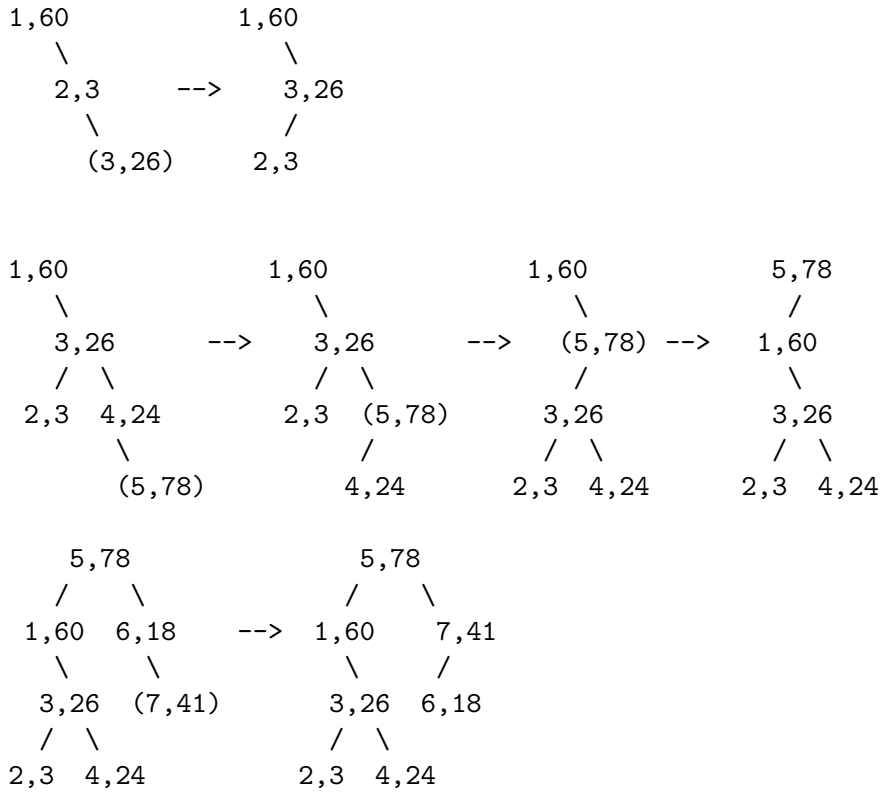


Figure 6.4: Inserting values into a treap. Each node is labeled with k, p where k is the key and p the priority. Insertion of values not requiring rotations are not shown.

that the treap is constructed recursively by choosing the highest-priority element as the root, then organizing all smaller-index and all larger-index nodes into the left and right subtrees by the same rule.

If we assign the priorities independently and uniformly at random from a sufficiently large set ($\omega(n^2)$ is enough in the limit), then we get no duplicates, and by symmetry all $n!$ orderings are equally likely. So the analysis of the depth of a treap with random priorities is identical to the analysis of a binary search tree with random insertion order. It's not hard to see that the costs of search insertion, and deletion operations are all linear in the depth of the tree, so the expected cost of each of these operations is $O(\log n)$.

6.3.1 Assumption of an oblivious adversary

One caveat is that this only works if the priorities of the elements of the tree are in fact independent. If operations on the tree are chosen by an **adaptive adversary**, this assumption may not work. An adaptive adversary is one that can observe the choice made by the algorithm and react to them: in this case, a simple strategy would be to insert elements 1, 2, 3, 4, etc., in order, deleting each one and reinserting it until it has a lower priority value than all the smaller elements. This might take a while for the later elements, but the end result is the linked list again. For this reason it is standard to assume in randomized data structures that the adversary is **oblivious**, meaning that it has to specify a sequence of operations without knowing what choices are made by the algorithm. Under this assumption, whatever insert or delete operations the adversary chooses, at the end of any particular sequence of operations we still have independent priorities on all the remaining elements, and the $O(\log n)$ analysis goes through.

6.3.2 Analysis

The analysis of treaps as carried out by Seidel and Aragon [SA96] is a nice example of how to decompose a messy process into simple variables, much like the linearity-of-expectation argument for QuickSort (§1.3.2). The key observation is that it's possible to bound both the expected depth of any node and the number of rotations needed for an insert or delete operation directly from information about the ancestor-descendant relationship between nodes.

Define two classes of indicator variables. For simplicity, we assume that the elements have keys 1 through n , which we also use as indices.

1. $A_{i,j}$ indicates the event that i is an **ancestor** of j , where i is an ancestor of j if it appears on the path from the root to j . Note that every node

is an ancestor of itself.

2. $C_{i;\ell,m}$ indicates the event that i is a **common ancestor** of both ℓ and m ; formally, $C_{i;\ell,m} = A_{i,\ell}A_{i,m}$.

The nice thing about these indicator variables is that it's easy to compute their expectations.

For $A_{i,j}$, i will be the ancestor of j if and only if i has a higher priority than j and there is no k between i and j that has an even higher priority: in other words, if i has the highest priority of all keys in the interval $[\min(i, j), \max(i, j)]$. To see this, imagine that we are constructing the treap recursively, by starting with all elements in a single interval and partitioning each interval by its highest-priority element. Consider the last interval in this process that contains both i and j , and suppose $i < j$ (the $j > i$ case is symmetric). If the highest-priority element is some k with $i < k < j$, then i and j are separated into distinct intervals and neither is the ancestor of the other. If the highest-priority element is j , then j becomes the ancestor of i . The highest-priority element can't be less than i or greater than j , because then we get a smaller interval that contains both i and j . So the only case where i becomes an ancestor of j is when i has the highest priority.

It follows that $E[A_{i,j}] = \frac{1}{|i-j|+1}$, where the denominator is just the number of elements in the range $[\min(i, j), \max(i, j)]$.

For $C_{i;\ell,m}$, i is the common ancestor of both ℓ and m if and only if it has the highest priority in both $[\min(i, \ell), \max(i, \ell)]$ and $[\min(i, m), \max(i, m)]$. It turns out that no matter what order i , ℓ , and m come in, these intervals overlap so that i must have the highest priority in $[\min(i, \ell, m), \max(i, \ell, m)]$. This gives $E[C_{i;\ell,m}] = \frac{1}{\max(i,\ell,m) - \min(i,\ell,m) + 1}$.

6.3.2.1 Searches

From the $A_{i,j}$ variables we can compute $\text{depth}(j) = \sum_i A_{i,j} - 1$.⁴ So

$$\begin{aligned} \mathbb{E}[\text{depth}(j)] &= \left(\sum_{i=1}^n \frac{1}{|i-j|+1} \right) - 1 \\ &= \left(\sum_{i=1}^j \frac{1}{j-i+1} \right) + \left(\sum_{i=j+1}^n \frac{1}{i-j+1} \right) - 1 \\ &= \left(\sum_{k=1}^j \frac{1}{k} \right) + \left(\sum_{k=2}^{n-j+1} \frac{1}{k} \right) - 1 \\ &= H_j + H_{n-j+1} - 2. \end{aligned}$$

This is maximized at $j = (n+1)/2$, giving $2H_{(n+1)/2} - 2 = 2 \ln n + O(1)$. So we get the same $2 \ln n + O(1)$ bound on the expected depth of any one node that we got for QuickSort. We can also sum over all j to get the exact value of the expected total path length (but we won't). These quantities bound the expected cost of searches.

6.3.2.2 Insertions and deletions

For insertions and deletions, the question is how many rotations we have to perform to float a new leaf up to its proper location (after an insertion) or to float a deleted node down to a leaf (before a deletion). Since insertion is just the reverse of deletion, we can get a bound on both by concentrating on deletion. The trick is to find some metric for each node that (a) bounds the number of rotations needed to move a node to the bottom of the tree and (b) is easy to compute based on the A and C variables

The **left spine** of a subtree is the set of all nodes obtained by starting at the root and following left pointers; similarly the **right spine** is what we get if we follow the right pointers instead.

When we rotate an element down, we are rotating either its left or right child up. This removes one element from either the right spine of the left subtree or the left spine of the right subtree, but the rest of the spines are left intact (see Figure 6.5). Subsequent rotations will eventually remove all these elements by rotating them above the target, while other elements in

⁴We need the -1 because of the convention that the root has depth 0, making the depth of a node one less than the number of its ancestors. Equivalently, we could exclude j from the sum and count only proper ancestors.

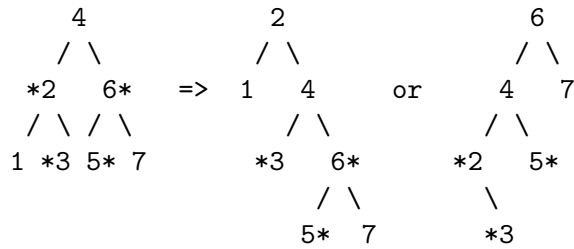


Figure 6.5: Rotating 4 right shortens the right spine of its left subtree by removing 2; rotating left shortens the left spine of the right subtree by removing 6.

the subtree will be carried out from under the target without ever appearing as a child or parent of the target. Because each rotation removes exactly one element from one or the other of the two spines, and we finish when both are empty, the sum of the length of the spines gives the number of rotations.

To calculate the length of the right spine of the left subtree of some element ℓ , start with the predecessor $\ell - 1$ of ℓ . Because there is no element between them, either $\ell - 1$ is a descendant of ℓ or an ancestor of ℓ . In the former case (for example, when ℓ is 4 in Figure 6.5), we want to include all ancestors of $\ell - 1$ up to ℓ itself. Starting with $\sum_i A_{i,\ell-1}$ gets all the ancestors of $\ell - 1$, and subtracting off $\sum_i C_{i;\ell-1,\ell}$ removes any common ancestors of $\ell - 1$ and ℓ . Alternatively, if $\ell - 1$ is an ancestor of ℓ , every ancestor of $\ell - 1$ is also an ancestor of ℓ , so the same expression $\sum_i A_{i,\ell-1} - \sum_i C_{i;\ell-1,\ell}$ evaluates to zero.

It follows that the expected length of the right spine of the left subtree is exactly

$$\begin{aligned}
 \mathbb{E} \left[\sum_{i=1}^n A_{i,\ell-1} - \sum_{i=1}^n C_{i;\ell-1,\ell} \right] &= \sum_{i=1}^n \frac{1}{|i - (\ell - 1)| + 1} - \sum_{i=1}^n \frac{1}{\max(i, \ell) - \min(i, \ell - 1) + 1} \\
 &= \sum_{i=1}^{\ell-1} \frac{1}{\ell - i} + \sum_{i=\ell}^n \frac{1}{i - \ell + 2} - \sum_{i=1}^{\ell-1} \frac{1}{\ell - i + 1} - \sum_{i=\ell}^n \frac{1}{i - (\ell - 1) + 1} \\
 &= \sum_{j=1}^{\ell-1} \frac{1}{j} + \sum_{j=2}^{n-\ell+2} \frac{1}{j} - \sum_{j=2}^{\ell} \frac{1}{j} - \sum_{j=2}^{n-\ell+2} \frac{1}{j} \\
 &= 1 - \frac{1}{\ell}.
 \end{aligned}$$

By symmetry, the expected length of the left spine of the right subtree is $1 - \frac{1}{n-\ell+1}$. So the total expected number of rotations needed to delete the

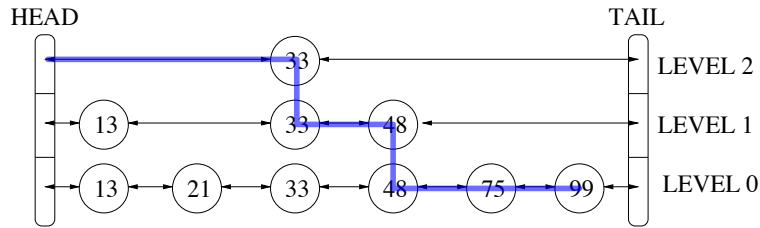


Figure 6.6: A skip list. The blue search path for 99 is superimposed on an original image from [AS07].

ℓ -th element is

$$2 - \frac{1}{\ell} - \frac{1}{n - \ell + 1} \leq 2.$$

6.3.2.3 Other operations

Treaps support some other useful operations: for example, we can split a treap into two treaps consisting of all elements less than and all elements greater than a chosen pivot by rotating the pivot to the root ($O(\log n)$ rotations on average, equal to the pivot's expected depth as calculated in §6.3.2.1) and splitting off the left and right subtrees. Merging two treaps with non-overlapping keys is the reverse of this and so it has the same expected complexity.

6.4 Skip lists

A skip list [Pug90] is a randomized tree-like data structure based on linked lists. It consists of a level 0 list that is an ordinary sorted linked list, together with higher-level lists that contain a random sampling of the elements at lower levels. When inserted into the level i list, an element flips a coin that tells it with probability p to insert itself in the level $i + 1$ list as well. The result is that the element is represented by a tower of nodes, one in each of the bottom $1 + X$ many layers, where X is a geometrically-distributed random variable. An example of a small skip list is shown in Figure 6.6.

Searches in a skip list are done by starting in the highest-level list and searching forward for the last node whose key is smaller than the target; the search then continues in the same way on the next level down. To bound the expected running time of a search, it helps to look at this process backwards; the reversed search path starts at level 0 and continues going backwards

until it reaches the first element that is also in a higher level; it then jumps to the next level up and repeats the process. The nice thing about this reversed process is that it has a simple recursive structure: if we restrict a skip list to only those nodes to the left of and at the same level or higher of a particular node, we again get a skip list. Furthermore, the structure of this restricted skip list depends only on coin-flips taken at nodes within it, so it's independent of anything that happens elsewhere in the full skip list.

We can analyze this process by tracking the number of nodes in the restricted skip list described above, which is just the number of nodes in the current level that are earlier than the current node. If we move left, this drops by 1; if up, this drops to p times its previous value on average. So the number of such nodes X_k after k steps satisfies the probabilistic recurrence $E[X_{k+1} | X_k] = (1-p)(X_k - 1) + p(pX_k) = (1-p+p^2)X_k - (1-p) \leq (1-p+p^2)X_k$, with $X_0 = n$. Wrapping expectations around both sides gives $E[X_{k+1}] = E[E[X_{k+1} | X_k]] \leq (1-p+p^2)E[X_k]$, and in general we get $E[X_k] \leq (1-p+p^2)^k X_0 = (1-p+p^2)^k n$. (We can substitute the rank of the starting node for n if we want a slightly tighter bound.) This is minimized at $p = 1/2$, giving $E[X_k] \leq (3/4)^k n$, suspiciously similar to the bound we computed before for random binary search trees.

When $X_k = 0$, our search is done, so if T is the time to search, we have $\Pr[T \geq k] = \Pr[X_k \geq 1] \leq (3/4)^k n$, by Markov's inequality. In particular, if we want to guarantee that we finish with probability $1 - \epsilon$, we need to run for $\log_{4/3}(n/\epsilon)$ steps. This translates into an $O(\log n)$ bound on the expected search time, and the constant is even the same as our (somewhat loose) bound for treaps.

The space per element of a skip list also depends on p . Every element needs one pointer for each level it appears in. The number of levels each element appears in is a geometric random variable where we are waiting for an event of probability $1 - p$, so the expected number of pointers is $\frac{1}{1-p}$. For constant p this is $O(1)$. However, the space cost can be reduced (at the cost of increasing search time) by adjusting p . For example, if space is at a premium, setting $p = 1/10$ produces $10/9$ pointers per node on average—not much more than in a linked list—but still gives $O(\log n)$ search time.

In general the trade-off is between $n \left(\frac{1}{1-p}\right)$ total expected pointers and $\log_{1/(1-p+p^2)}(n/\epsilon)$ search time. For small p , the number of pointers scales as $1+O(p)$, while the constant factor in the search time is $\frac{1}{-\log(1-p+p^2)} = O(1/p)$.

Skip lists are even easier to split or merge than treaps. It's enough to cut (or recreate) all the pointers crossing the boundary, without changing the structure of the rest of the list.

Chapter 7

Hashing

Last updated 2019. Some material may be out of date.

The basic idea of hashing is that we have keys from a large set U , and we'd like to pack them in a small set M by passing them through some function $h : U \rightarrow M$, without getting too many **collisions**, pairs of distinct keys x and y with $h(x) = h(y)$. Where randomization comes in is that we want this to be true even if the adversary picks the keys to hash. At one extreme, we could use a random function, but this will take a lot of space to store.¹ So our goal will be to find functions with succinct descriptions that are still random enough to do what we want.

The presentation in this chapter is based largely on [MR95, §§8.4-8.5] (which is in turn based on work of Carter and Wegman [CW77] on universal hashing and Fredman, Komlós, and Szemerédi [FKS84] on $O(1)$ worst-case hashing); on [PR04] and [Pag06] for cuckoo hashing; and [MU05, §5.5.3] for Bloom filters.

7.1 Hash tables

Here we review the basic idea of **hash tables**, which are implementations of the **dictionary** data type mapping keys to values. The basic idea of hash tables is usually attributed to Dumey [Dum56].²

¹An easy counting argument shows that almost all functions from U to M take $|U| \log |M|$ bits to represent, no matter how clever you are about choosing your representation. This forms the basis for **algorithmic information theory**, which *defines* an object as random if there is no way to reduce the number of bits used to express it.

²Caveat: This article is pretty hard to find, so I am basing this citation on its frequent appearance in later sources. This is generally a bad idea that would not really be acceptable

Suppose we want to store n elements from a universe U of in a table with **keys** or **indices** drawn from an index space M of size m . Typically we assume $U = [|U|] = \{0 \dots |U| - 1\}$ and $M = [m] = \{0 \dots m - 1\}$.

If $|U| \leq m$, we can just use an array. Otherwise, we can map keys to positions in the array using a **hash function** $h : U \rightarrow M$. This necessarily produces **collisions**: pairs (x, y) with $h(x) = h(y)$, and any design of a hash table must include some mechanism for handling keys that hash to the same place. Typically this is a secondary data structure in each bin, but we may also place excess values in some other place. Typical choices for data structures are linked lists (**separate chaining** or just **chaining**) or secondary hash tables (see §7.3 below). Alternatively, we can push excess values into other positions in the same hash table according to some deterministic rule (**open addressing** or **probing**) or a second hash function (see §7.4).

For most of these techniques, the costs of insertions and searches will depend on how likely it is that we get collisions. An adversary that knows our hash function can always choose keys with the same hash value, but we can avoid that by choosing our hash function randomly.³ Our ultimate goal is to do each search in $O(1 + n/m)$ expected time, which for $n \leq m$ will be much better than the $\Theta(\log n)$ time for pointer-based data structures like balanced trees or skip lists. The quantity n/m is called the **load factor** of the hash table and is often abbreviated as α .

All of this only works if we are working in a RAM (random-access machine model), where we can access arbitrary memory locations in time $O(1)$ and similarly compute arithmetic operations on $O(\log|U|)$ -bit values in time $O(1)$. There is an argument that in reality any actual RAM machine requires either $\Omega(\log m)$ time to read one of m memory locations (routing costs) or, if one is particularly pedantic, $\Omega(m^{1/3})$ time (speed of light + finite volume for each location). We will ignore this argument.

We will try to be consistent in our use of variables to refer to the different parameters of a hash table. Table 7.1 summarizes the meaning of these variable names.

in an actual scholarly publication.

³In practice, hardly anybody every does this. Hash functions are instead chosen based on fashion and occasional experiments, often with additional goals like cryptographic security or speed. For cryptographic security, the **SHA** family is standard. For speed, **xxHash** (see <https://cyan4973.github.io/xxHash/>) is probably the fastest widely-used hash function with decent statistical properties.

U	Universe of all keys
$S \subseteq U$	Set of keys stored in the table
$n = S $	Number of keys stored in the table
M	Set of table positions
$m = M $	Number of table positions
$\alpha = n/m$	Load factor

Table 7.1: Hash table parameters

7.2 Universal hash families

A family of hash functions H is **2-universal** if for any $x \neq y$, $\Pr[h(x) = h(y)] \leq 1/m$ for a uniform random $h \in H$. It's **strongly 2-universal** if for any $x_1 \neq x_2 \in U$, $y_1, y_2 \in M$, $\Pr[h(x_1) = y_1 \wedge h(x_2) = y_2] = 1/m^2$ for a uniform random $h \in H$. Another way to describe strong 2-universality is that the values of the hash function are uniformly distributed and pairwise-independent.

For $k > 2$, **k -universal** usually means **strongly k -universal**: Given distinct $x_1 \dots x_k$, and any $y_1 \dots y_k$, $\Pr[h(x_i) = y_i \forall i] = m^{-k}$. This is equivalent to the $h(x_i)$ values for distinct x_i and randomly-chosen h having a uniform distribution and k -wise independence. It is possible to generalize the weak version of 2-universality to get a weak version of k -universality ($\Pr[h(x_i)$ are all equal] $\leq m^{-(k-1)}$), but this generalization is not as useful as strong k -universality.

To analyze universal hash families, it is helpful to have some notation for counting collisions. We'll mostly be doing counting rather than probabilities because it saves carrying around a lot of denominators. Since we are assuming uniform choices of h we can always get back probabilities by dividing by $|H|$.

Let $\delta(x, y, h) = 1$ if $x \neq y$ and $h(x) = h(y)$, 0 otherwise. Abusing notation, we also define, for sets X, Y , and H , $\delta(X, Y, H) = \sum_{x \in X, y \in Y, h \in H} \delta(x, y, h)$; and allow lowercase variables to stand in for singleton sets, as in $\delta(x, Y, h) = \delta(\{x\}, Y, \{h\})$. Now the statement that H is 2-universal becomes $\forall x, y : \delta(x, y, H) \leq |H|/m$; this says that only $1/m$ of the functions in H cause any particular distinct x and y to collide.

If H includes all functions $U \rightarrow M$, we get equality: a random function gives $h(x) = h(y)$ with probability exactly $1/m$. But we might do better if each h tends to map distinct values to distinct places. The following lemma shows we can't do too much better:

Lemma 7.2.1. *For any family H , there exist x, y such that $\delta(x, y, H) \geq \frac{|H|}{m} \left(1 - \frac{m-1}{|U|-1}\right)$.*

Proof. We'll count collisions in the inverse image of each element z . Since all distinct pairs of elements of $h^{-1}(z)$ collide with each other, we have

$$\delta(h^{-1}(z), h^{-1}(z), h) = |h^{-1}(z)| \cdot \left(|h^{-1}(z)| - 1 \right).$$

Summing over all $z \in M$ gets all collisions, giving

$$\delta(U, U, h) = \sum_{z \in M} \left(|h^{-1}(z)| \cdot \left(|h^{-1}(z)| - 1 \right) \right).$$

Use convexity or Lagrange multipliers to argue that the right-hand side is minimized subject to $\sum_z |h^{-1}(z)| = |U|$ when all pre-images are the same size $|U|/m$. It follows that

$$\begin{aligned} \delta(U, U, h) &\geq \sum_{z \in M} \frac{|U|}{m} \left(\frac{|U|}{m} - 1 \right) \\ &= m \frac{|U|}{m} \left(\frac{|U|}{m} - 1 \right) \\ &= \frac{|U|}{m} (|U| - m). \end{aligned}$$

If we now sum over all h , we get

$$\delta(U, U, H) \geq \frac{|H|}{m} |U| (|U| - m).$$

There are exactly $|U|(|U| - 1)$ ordered pairs x, y for which $\delta(x, y, H)$ might not be zero; so the Pigeonhole principle says some pair x, y has

$$\begin{aligned} \delta(x, y, H) &\geq \frac{|H|}{m} \left(\frac{|U|(|U| - m)}{|U|(|U| - 1)} \right) \\ &= \frac{|H|}{m} \left(1 - \frac{m - 1}{|U| - 1} \right). \end{aligned}$$

□

Since $1 - \frac{m-1}{|U|-1}$ is likely to be very close to 1, we are happy if we get the 2-universal upper bound of $|H|/m$.

Why we care about this: With a 2-universal hash family, chaining using linked lists costs $O(1 + s/n)$ expected time per operation. The reason is that the expected cost of an operation on some key x is proportional to the size of the linked list at $h(x)$ (plus $O(1)$ for the cost of hashing itself). But the expected size of this linked list is just the expected number of keys y in the dictionary that collide with x , which is exactly $s\delta(x, y, H) \leq s/n$.

7.2.1 Linear congruential hashing

Universal hash families often look suspiciously like classic pseudorandom number generators. Here is a 2-universal hash family based on taking remainders. It is assumed that the universe U is a subset of \mathbb{Z}_p , the integers mod p ; effectively, this just means that every element x of U satisfies $0 \leq x \leq p - 1$.

Lemma 7.2.2. *Let $h_{ab}(x) = (ax + b \bmod p) \bmod m$, where $a \in \mathbb{Z}_p \setminus \{0\}$, $b \in \mathbb{Z}_p$, and p is a prime $\geq m$. Then $\{h_{ab}\}$ is 2-universal.*

Proof. Again, we count collisions. Split $h_{ab}(x)$ as $g(f_{ab}(x))$ where $f_{ab}(x) = ax + b \bmod p$ and $g(x) = x \bmod m$.

The intuition is that if we fix x and y and consider all pairs a, b , all distinct pairs of values $r = f_{ab}(x)$ and $s = f_{ab}(y)$ are equally likely. We then show that feeding these values to g produces no more collisions than expected.

The formal statement of the intuition is that for any $0 \leq x, y \leq p - 1$ with $x \neq y$, $\delta(x, y, H) = \delta(\mathbb{Z}_p, \mathbb{Z}_p, g)$.

To prove this, fix x and y , and consider some pair $r \neq s \in \mathbb{Z}_p$. Then the equations $ax + b = r$ and $ay + b = s$ have a unique solution for a and $b \bmod p$ (because \mathbb{Z}_p is a finite field). Furthermore this solution has $a \neq 0$ since otherwise $f_{ab}(x) = f_{ab}(y) = b$. So the function $q(a, b) = \langle f_{ab}(x), f_{ab}(y) \rangle$ is a bijection between pairs a, b and pairs r, s . Any collisions will arise from applying g , giving $\delta(x, y, H) = \sum_{a,b} \delta(x, y, h_{ab}) = \sum_{r \neq s} \delta(r, s, g) = \delta(\mathbb{Z}_p, \mathbb{Z}_p, g)$.

Now we just need to count how many distinct r and s collide. There are p choices for r . For each r , there are at most $\lceil p/m \rceil$ values that map to the same remainder mod m , but one of those values is equal to r , so this leaves only $\lceil p/m \rceil - 1$ choices for s . But $\lceil p/m \rceil - 1 \leq (p + (m - 1))/m - 1 = (p - 1)/m$. Multiplying by the p choices for r gives $\delta(x, y, H) = \delta(\mathbb{Z}_p, \mathbb{Z}_p, g) \leq p(p - 1)/m$.

Since each choice of $a \neq 0$ and b occurs with probability $\frac{1}{p(p-1)}$, this gives a probability of collision of at most $1/m$. \square

A difficulty with this hash family is that it requires doing modular arithmetic. A faster hash is given by Dietzfelbinger *et al.* [DHKP97], although it requires a slight weakening of the notion of 2-universality. For each k and ℓ they define a class $H_{k,\ell}$ of functions from $[2^k]$ to $[2^\ell]$ by defining

$$h_a(x) = (ax \bmod 2^k) \operatorname{div} 2^{k-\ell},$$

where $x \operatorname{div} y = \lfloor x/y \rfloor$. They prove [DHKP97, Lemma 2.4] that if a is a random *odd* integer with $0 < a < 2^\ell$, and $x \neq y$, $\Pr [h_a(x) = h_a(y)] \leq 2^{-\ell+1}$. This increases by a factor of 2 the likelihood of a collision, but any extra costs from this can often be justified in practice by the reduction in costs from working with powers of 2.

If we are willing to use more randomness (and more space), a method called **tabulation hashing** (§7.2.2) gives a simpler alternative that is 3-universal.

7.2.2 Tabulation hashing

Tabulation hashing [CW77] is a method for hashing fixed-length strings (or things that can be represented as fixed-length strings) into bit-vectors. The description here follows Patrascu and Thorup [PT12].

Let c be the length of each string in characters, and let s be the size of the alphabet. Initialize the hash function by constructing tables $T_1 \dots T_c$ mapping characters to independent random bit-vectors of size $\lg m$. Define

$$h(x) = T_1[x_1] \oplus T_2[x_2] \oplus \dots T_c[x_c],$$

where \oplus represents bitwise exclusive OR (what \wedge does in C-like languages).⁴ This gives a family of hash functions that is 3-wise independent but not 4-wise independent.

The intuition for why the hash values might be independent is that if we have a collection of strings, and each string brings in an element of T that doesn't appear in the other strings, then that element is independent of the hash values for the other strings and XORing it with the rest of the hash value gives a random bit string that is independent of the hash values of the other strings. In fact, we don't even need each string to include a unique value; it's enough if we can order the strings so that each string gets a value that isn't represented among its predecessors.

More formally, suppose we can order the strings x^1, x^2, \dots, x^n that we are hashing so that each has a position i_j such that $x_{i_j}^j \neq x_{i_j}^{j'}$ for any $j' < j$, then we have, for each value v , $\Pr [h(x^j) = v \mid h(x^{j'}) = v_{j'}, \forall j' < j] = 1/m$.

⁴Letting m be a power of 2 and using exclusive OR is convenient on real computers. If for some reason we don't like this approach, the same technique, with essentially the same analysis, works for arbitrary m if we replace bitwise XOR with addition mod m .

It follows that the hash values are independent:

$$\begin{aligned} \Pr \left[h(x^1) = v_1, h(x^2) = v_2, \dots, h(x^n) = v_n \right] &= \prod_{j=1}^n \Pr \left[h(x^j) = v_j \mid h(x^1) = v_1 \dots h(x^{j-1}) = v_{j-1} \right] \\ &= \frac{1}{m^n} \\ &= \prod_{j=1}^n \Pr \left[h(x^j) = v_j \right]. \end{aligned}$$

Now we want to show that when $n = 3$, this actually works for all possible distinct strings x , y , and z . Let S be the set of indices i such that $y_i \neq x_i$, and similarly let T be the set of indices i such that $z_i \neq x_i$; note that both sets must be non-empty, since $y \neq x$ and $z \neq x$. If $S \setminus T$ is nonempty, then (a) there is some index i in T where $z_i \neq x_i$, and (b) there is some index j in $S \setminus T$ where $y_j \neq x_j = z_j$; in this case, ordering the strings as x, z, y gives the independence property above. If $T \setminus S$ is nonempty, order them as x, z, y instead. Alternatively, if $S = T$, then $y_i \neq z_i$ for some i in S (otherwise $y = z$, since they both equal x on all positions outside S). In this case, x_i, y_i , and z_i are all distinct.

For $n = 4$, we can have strings **aa**, **ab**, **ba**, and **bb**. If we take the bitwise exclusive OR of all four hash values, we get zero, because each character is included exactly twice in each position. So the hash values are not independent, and we do not get 4-independence in general.

However, even though tabulation hashing is not 4-independent, most reasonably small sets of inputs do give independence. This can be used to show various miraculous properties like working well for the cuckoo hashing algorithm described in §7.4.

7.3 FKS hashing

The FKS hash table, named for Fredman, Komlós, and Szemerédi [FKS84], is a method for storing a static set S so that we never pay more than constant time for search (not just in expectation), while at the same time not consuming too much space. The assumption that S is static is critical, because FKS chooses hash functions based on the elements of S .

If we were lucky in our choice of S , we might be able to do this with standard hashing. A **perfect hash function** for a set $S \subseteq U$ is a hash function $h : U \rightarrow M$ that is injective on S (that is, $x \neq y \rightarrow h(x) \neq h(y)$ when $x, y \in S$). Unfortunately, we can only count on finding a perfect hash function if m is large:

Lemma 7.3.1. *If H is 2-universal and $|S| = n$ with $\binom{n}{2} \leq m$, then there is a perfect $h \in H$ for S .*

Proof. Let h be chosen uniformly at random from H . For each unordered pair $x \neq y$ in S , let X_{xy} be the indicator variable for the event that $h(x) = h(y)$, and let $C = \sum_{x \neq y} X_{xy}$ be the total number of collisions in S . Each X_{xy} has expectation at most $1/m$, so $\mathbb{E}[C] \leq \binom{n}{2}/m < 1$. But we can write $\mathbb{E}[C]$ as $\mathbb{E}[C \mid C = 0] \Pr[C = 0] + \mathbb{E}[C \mid C \geq 1] \Pr[C \neq 0] \geq \Pr[C \neq 0]$. So $\Pr[C \neq 0] \leq \binom{n}{2}/m < 1$, giving $\Pr[C = 0] > 0$. But if C is zero with nonzero probability, there must be some h that makes it 0. That h is perfect for S . \square

Essentially the same argument shows that if $\alpha \binom{n}{2} \leq m$, then $\Pr[h \text{ is perfect for } S] \geq 1 - \alpha$. This can be handy if we want to find a perfect hash function and not just demonstrate that it exists.

Using a perfect hash function, we get $O(1)$ search time using $O(n^2)$ space. But we can do better by using perfect hash functions only at the second level of our data structure, which at top level will just be an ordinary hash table. This is the idea behind the Fredman-Komlós-Szemerédi (FKS) hash table [FKS84].

The short version is that we hash to $n = |S|$ bins, then rehash perfectly within each bin. The top-level hash table stores a pointer to a header for each bin, which gives the size of the bin and the hash function used within it. The i -th bin, containing n_i elements, uses $O(n_i^2)$ space to allow perfect hashing. The total size is $O(n)$ as long as we can show that $\sum_{i=1}^n n_i^2 = O(n)$. The time to do a search is $O(1)$ in the worst case: $O(1)$ for the outer hash plus $O(1)$ for the inner hash.

Theorem 7.3.2. *The FKS hash table uses $O(n)$ space.*

Proof. Suppose we choose $h \in H$ as the outer hash function, where H is some 2-universal family of hash functions. Compute:

$$\begin{aligned} \sum_{i=1}^n n_i^2 &= \sum_{i=1}^n (n_i + n_i(n_i - 1)) \\ &= n + \delta(S, S, h). \end{aligned}$$

The last equality holds because each ordered pair of distinct values in S that map to the same bucket i corresponds to exactly one collision in $\delta(S, S, h)$.

Since H is 2-universal, we have $\delta(S, S, H) \leq |H| \frac{|S|(|S|-1)}{n} = |H| \frac{n(n-1)}{n} = |H|(n-1)$. But then the Pigeonhole principle says there exists some $h \in H$

with $\delta(S, S, h) \leq \frac{1}{|H|} \delta(S, S, H) = n - 1$. Choosing this h gives $\sum_{i=1}^n n_i^2 \leq n + (n - 1) = 2n - 1 = O(n)$. \square

If we want to find a good h quickly, increasing the size of the outer table to n/α gives us a probability of at least $1 - \alpha$ of getting a good one, using essentially the same argument as for perfect hash functions.

7.4 Cuckoo hashing

The FKS hash guarantees that any search takes only two probes, but it only works for static data. **Cuckoo hashing** [PR04] gives the same guarantee for dynamic data.

The name comes from the **cuckoo**, a family of birds notorious for stealing space for their own eggs in other birds' nests. In cuckoo hashing, newly-inserted elements may steal slots from other elements, forcing those elements to find an alternate nest.

The formal mechanism is to use two hash functions h_1 and h_2 , and store each element x in one of the two positions $h_1(x)$ or $h_2(x)$. This may require moving other elements to their alternate locations to make room. But the payoff is that each search takes only two reads, which can even be done in parallel. This is optimal by a lower bound of Pagh [Pag01], which also shows a matching upper bound for static dictionaries using a different technique.

Cuckoo hashing was invented by Pagh and Rodler [PR04]. The version described here is based on a simplified version from notes of Pagh [Pag06]. The main difference is that it uses just one table instead of the two tables—one for each hash function—in [PR04].

7.4.1 Structure

We have a table T of size n , with two separate, independent hash functions h_1 and h_2 . These functions are assumed to be k -universal for some sufficiently large value k ; as long as we never look at more than k values at once, this means we can treat them effectively as random functions. In practice, using crummy hash functions seems to work just fine, a common property of hash tables. There are also specific hash functions that have been shown to work with particular variants of cuckoo hashing [PR04, PT12]. We will avoid these issues by assuming that our hash functions are actually random.

Every key x is stored either in $T[h_1(x)]$ or $T[h_2(x)]$. So the search procedure just looks at both of these locations and returns whichever one contains x (or fails if neither contains x).

To insert a value $x_1 = x$, we must put it in $T[h_1(x_1)]$ or $T[h_2(x_1)]$. If one or both of these locations is empty, we put it there. Otherwise we have to kick out some value that is in the way (this is the “cuckoo” part of cuckoo hashing, named after the bird that leaves its eggs in other birds’ nests). We do this by letting $x_2 = T[h_1(x_1)]$ and writing x_1 to $T[h_1(x_1)]$. We now have a new “nestless” value x_2 , which we swap with whatever is in $T[h_2(x_2)]$. If that location was empty, we are done; otherwise, we get a new value x_3 that we have to put in $T[h_1(x_3)]$ and so on. The procedure terminates when we find an empty spot or if enough iterations have passed that we don’t expect to find an empty spot, in which case we rehash the entire table. This process can be implemented succinctly as shown in Algorithm 7.1.

```

1 procedure insert( $x$ )
2   if  $T[h_1(x) = x$  or  $T[h_2(x) = x$  then
3     | return
4    $\text{pos} \leftarrow h_1(x)$ 
5   for  $i \leftarrow 1 \dots n$  do
6     | if  $T[\text{pos}] = \perp$  then
7       |  $T[\text{pos}] \leftarrow x$ 
8       | return
9     |  $x \rightleftharpoons T[\text{pos}]$ 
10    | if  $\text{pos} = h_1(x)$  then
11      |  $\text{pos} \leftarrow h_2(x)$ 
12    | else
13      |  $\text{pos} \leftarrow h_1(x)$ 
14  | If we got here, rehash the table and reinsert  $x$ .

```

Algorithm 7.1: Insertion procedure for cuckoo hashing. Adapted from [Pag06]

A detail not included in the above code is that we always rehash (in theory) after m^2 insertions; this avoids potential problems with the hash functions used in the paper not being universal enough. We will avoid this issue by assuming that our hash functions are actually random (instead of being approximately n -universal with reasonably high probability). For a more principled analysis of where the hash functions come from, see [PR04]. An alternative hash family that is known to work for a slightly different variant of cuckoo hashing is tabulation hashing, as described in §7.2.2; the proof that this works is found in [PT12].

7.4.2 Analysis

The main question is how long it takes the insertion procedure to terminate, assuming the table is not too full.

First let's look at what happens during an insert if we have a lot of nestless values. We have a sequence of values x_1, x_2, \dots , where each pair of values x_i, x_{i+1} collides in h_1 or h_2 . Assuming we don't reach the loop limit, there are three main possibilities (the leaves of the tree of cases below):

1. Eventually we reach an empty position without seeing the same key twice.
2. Eventually we see the same key twice; there is some i and $j > i$ such that $x_j = x_i$. Since x_i was already moved once, when we reach it the second time we will try to move it back, displacing x_{i-1} . This process continues until we have restored x_2 to $T[h_1(x_1)]$, displacing x_1 to $T[h_2(x_1)]$ and possibly creating a new sequence of nestless values. Two outcomes are now possible:
 - (a) Some x_ℓ is moved to an empty location. We win!
 - (b) Some x_ℓ is moved to a location we've already looked at. We lose! We find we are playing musical chairs with more players than chairs, and have to rehash.

Let's look at the probability that we get the last case, a **closed loop**. Following the argument of Pagh and Rodler, we let v be the number of distinct nestless keys in the loop. Since v includes x_1 , v is at least 1. We can now count how many different ways such a loop can form, and argue that in each case we include enough information to reconstruct $h_1(u_i)$ and $h_2(u_i)$ for each of a specific set of unique elements u_1, \dots, u_v .

We'll specify this information:

- The v elements u_1, \dots, u_v . Since we can fix $u_1 = x_1$, this leaves $v - 1$ choices from S , giving $n^{(v-1)}$ possibilities. We'll assume that the other u_i for $i > 1$ appear in the list in the same order they first appear in the sequence x_1, x_2, \dots .
- The $v - 1$ locations we are trying to fit these elements into. There are $m^{(v-1)}$ choices for these. Again we order these by order of first appearance.

- The values of i , j , and ℓ . These allow us to identify which segments of the sequence x_1, x_2, \dots correspond to new values u_i and which are old values repeated (possibly in reverse order).

There are at most v choices for i and j (because we are still in the initial segment with no repeats), and at most $2v$ choices for ℓ if we count carefully (because we either land on either the initial no-duplicate sequence starting with x_1 or the second no-duplicate sequence starting with the second occurrence of x_1).

Altogether these give $2v^3$ choices.

- For each $i \neq 1$, whether the first occurrence of u_i appears in $h_1(u_i)$ or $h_2(u_i)$. This gives 2^{v-1} choices, and allows us to correctly identify $h_1(u_i)$ or $h_2(u_1)$ from the value of u_i and its first location and the other hash value for u_i given the next location in the list.⁵

Multiplying everything out gives at most $2v^3(2nm)^{(v-1)}$ choices of closed loops with v unique elements. Since each particular loop allows us to determine both h_1 and h_2 for all v of its elements, the probability that we get exactly these hash values (so that the loop occurs) is m^{-2v} . Summing over all closed loops with v elements gives a total probability of

$$\begin{aligned} 2v^3(2nm)^{v-1}m^{-2v} &= 2v^3(2n)^{v-1}m^{-v-1} \\ &= 2v^3(2n/m)^{v-1}m^{-2}. \end{aligned}$$

Now sum over all $v \geq 1$. We get

$$m^{-2} \sum_{v=1}^n 2v^3(2n/m)^{v-1} < m^{-2} \sum_{v=1}^{\infty} 2v^3(2n/m)^{v-1}.$$

The series converges if $2n/m < 1$, so for any fixed $\alpha < 1/2$, the probability of any closed loop forming is $O(m^{-2})$.

If we do hit a closed loop, then we pay $O(m)$ time to scan the existing table and create a new empty table, and $O(n) = O(m)$ time on average to reinsert all the elements into the new table, assuming that this reinsertion process doesn't generate any more closed loops and that the average cost of an insertion that doesn't produce a closed loop is $O(1)$, which we will show below. But the rehashing step only fails with probability $O(nm^{-2}) = O(m^{-1})$, so if it does fail we can just try again until it works, and the expected total cost

⁵The original analysis in [PR04] avoids this by alternating between two tables, so that we can determine which of h_1 or h_2 is used at each step by parity.

is still $O(m)$. Since we pay this $O(m)$ for each insertion with probability $O(m^{-2})$, this adds only $O(m^{-1})$ to the expected cost of a single insertion.

Now we look at what happens if we don't get a closed loop. This doesn't force us to rehash, but if the path is long enough, we may still pay a lot to do an insertion.

It's a little messy to analyze the behavior of keys that appear more than once in the sequence, so the trick used in the paper is to observe that for any sequence of nestless keys $x_1 \dots x_p$, there is a subsequence of size $p/3$ with no repetitions that starts with x_1 . This will be either the sequence S_1 given by $x_1 \dots x_{j-1}$ —the sequence starting with the first place we try to insert x_1 —or S_2 given by $x_1 = x_{i+j-1} \dots x_p$, the sequence starting with the second place we try to insert x_1 . Between these we have a third sequence T where we undo some of the moves made in S_1 . Because $|S_1| + |T| + |S_2| \geq p$, at least one of these three subsequences has size $p/3$. But $|T| \leq |S_1|$, so it must be either S_1 or S_2 .

We can then argue that the probability that we get a sequence of v distinct keys in either S_1 or S_2 is most $2(n/m)^{v-1}$. The $(n/m)^{v-1}$ is because we need to hit a nonempty spot (which happens with probability at most n/m) for the first $v-1$ elements in the path, and since we assume that our hash functions are random, the choices of these $v-1$ spots are all independent. The 2 is from the union bound over S_1 and S_2 . If T is the insertion time, we get $E[T] = \sum_{v=1}^{\infty} \Pr[T \geq v] \leq \sum_{v=1}^{\infty} 2(n/m)^{v-1} = O(1)$, assuming n/m is bounded by a constant less than 1. Since we already need $n/m \leq 1/2$ to avoid the bad closed-loop case, we can use this here as well.

An annoyance with cuckoo hashing is that it has high space overhead compared to more traditional hash tables: in order for the first part of the analysis above to work, the table must be at least half empty. This can be avoided at the cost of increasing the time complexity by choosing between d locations instead of 2. This technique, due to Fotakis *et al.* [FPSS03], is known as **d -ary cuckoo hashing**. For a suitable choice of d , it uses $(1+\epsilon)n$ space and guarantees that a lookup takes $O(1/\epsilon)$ probes, while insertion takes $(1/\epsilon)^{O(\log \log(1/\epsilon))}$ steps in theory and appears to take $O(1/\epsilon)$ steps in practice according to experiments done by the authors.

7.5 Practical issues

Most hash functions used in practice do not have very good theoretical guarantees, and indeed we have assumed in several places in this chapter that we are using genuinely random hash functions when we would expect our

actual hash functions to be at most 2-universal. There is some justification for doing this if there is enough entropy in the set of keys S . A proof of this for many common applications of hash functions is given by Chung *et al.* [CMV13].

Even taking into account these results, hash tables that depend on strong properties of the hash function may behave badly if the user supplies a crummy hash function. For this reason, many library implementations of hash tables are written defensively, using algorithms that respond better in bad cases. See <https://svn.python.org/projects/python/trunk/Objects/dictobject.c> for an example of a widely-used hash table implementation chosen specifically because of its poor theoretical characteristics.

For large hash tables, local probing schemes are often faster than chaining or cuckoo hashing, because it is likely that all of the locations probed to find a particular value will be on the same virtual memory page. This means that a search for a new value usually requires one cache miss instead of two. **Hopscotch hashing** [HST08] combines ideas from linear probing and cuckoo hashing to get better performance than both in practice.

7.6 Bloom filters

See [MU05, §5.5.3] for basics and a formal analysis or http://en.wikipedia.org/wiki/Bloom_filter for many variations and the collective wisdom of the unwashed masses. The presentation here mostly follows [MU05].

7.6.1 Construction

Bloom filters are a highly space-efficient randomized data structure invented by Burton H. Bloom [Blo70] that store sets of data, with a small probability that elements not in the set will be erroneously reported as being in the set.

Suppose we have k independent hash functions h_1, h_2, \dots, h_k . Our memory store A is a vector of m bits, all initially zero. To store a key x , set $A[h_i(x)] = 1$ for all i . To test membership for x , see if $A[h_i(x)] = 1$ for all i . The membership test always gives the right answer if x is in fact in the Bloom filter. If not, we might decide that x is in the Bloom filter anyway.

7.6.2 False positives

The probability of such **false positives** can be computed in two steps: first, we estimate how many of the bits in the Bloom filter are set after inserting

n values, and then we use this estimate to compute a probability that any fixed x shows up when it shouldn't.

If the h_i are close to being independent random functions,⁶ then with n entries in the filter we have $\Pr[A[i] = 1] = 1 - (1 - 1/m)^{kn}$, since each of the kn bits that we set while inserting the n values has one chance in m of hitting position i .

We'd like to simplify this using the inequality $1 + x \leq e^x$, but it goes in the wrong direction; instead, we'll use $1 - x \geq e^{-x-x^2}$, which holds for $0 \leq x \leq 0.683803$ and in our application holds for $m \geq 2$. This gives

$$\begin{aligned} \Pr[A[i] = 1] &\leq 1 - (1 - 1/m)^{kn} \\ &\leq 1 - e^{-k(n/m)(1+1/m)} \\ &= 1 - e^{-k\alpha(1+1/m)} \\ &= 1 - e^{-k\alpha'} \end{aligned}$$

where $\alpha = n/m$ is the load factor and $\alpha' = \alpha(1 + 1/m)$ is the load factor fudged upward by a factor of $1 + 1/m$ to make the inequality work.

Suppose now that we check to see if some value x that we never inserted in the Bloom filter appears to be present anyway. This occurs if $A[h_i(x)] = 1$ for all i . For $k \ll m$, it is likely that the $h_i(x)$ values are all distinct. In this case, the probability that they all come up 1 conditioned on A is

$$\left(\frac{\sum A[i]}{m}\right)^k. \tag{7.6.1}$$

We have an upper bound $E[\sum A[i]] \leq m(1 - e^{-k\alpha'})$, and if we were born luckier, we might be able to get an upper bound on the expectation of (7.6.1) by applying Jensen's inequality to the function $f(x) = x^k$. But sadly this inequality also goes in the wrong direction, because f is convex for $k > 1$. So instead we will prove a concentration bound on $S = \sum A[i]$.

Because the $A[i]$ are not independent, we can't use off-the-shelf Chernoff bounds. Instead, we rely on McDiarmid's inequality. Our assumption is that the locations of the kn ones that get written to A are independent. Furthermore, changing the location of one of these writes changes S by at most 1. So McDiarmid's inequality (5.3.12) gives $\Pr[S \geq E[S] + t] \leq e^{-2t^2/kn}$,

⁶We are going to sidestep the rather deep swamp of how plausible this assumption is and what assumption we should be making instead. However, it is known [KM08] that starting with two sufficiently random-looking hash functions h and h' and setting $h_i(x) = h(x) + ih'(x)$ works.

which is bounded by n^{-c} for $t \geq \sqrt{\frac{1}{2}ckn \log n}$. So as long as a reasonably large fraction of the array is likely to be full, the relative error from assuming $S = E[S]$ is likely to be small. Alternatively, if the array is mostly empty, then we don't care about the relative error so much because the probability of getting a false positive will already be exponentially small as a function of k .

So let's assume for simplicity that our false positive probability is exactly $(1 - e^{-k\alpha'})^k$. We can choose k to minimize this quantity for fixed α' by doing the usual trick of taking a derivative and setting it to zero; to avoid weirdness with the k in the exponent, it helps to take the logarithm first (which doesn't affect the location of the minimum), and it further helps to take the derivative with respect to $x = e^{-\alpha'k}$ instead of k itself. Note that when we do this, $k = -\frac{1}{\alpha'} \ln x$ still depends on x , and we will deal with this by applying this substitution at an appropriate point.

Compute

$$\begin{aligned} \frac{d}{dx} \ln \left((1-x)^k \right) &= \frac{d}{dx} k \ln(1-x) \\ &= \frac{d}{dx} \left(-\frac{1}{\alpha'} \ln x \right) \ln(1-x) \\ &= -\frac{1}{\alpha'} \left(\frac{\ln(1-x)}{x} - \frac{\ln x}{1-x} \right). \end{aligned}$$

Setting this to zero gives $(1-x) \ln(1-x) = x \ln x$, which by symmetry has the unique solution $x = 1/2$, giving $k = \frac{1}{\alpha'} \ln 2$.

In other words, to minimize the false positive rate for a known load factor α , we want to choose $k = \frac{1}{\alpha'} \ln 2 = \frac{1}{\alpha(1+1/m)} \ln 2$, which makes each bit one with probability approximately $1 - e^{-\ln 2} = \frac{1}{2}$. This makes intuitive sense, since having each bit be one or zero with equal probability maximizes the entropy of the data.

The probability of a false positive is then $2^{-k} = 2^{-\ln 2/\alpha'}$. For a given maximum false positive rate ϵ , and assuming optimal choice of k , we need to keep $\alpha' \leq \frac{\ln^2 2}{\ln(1/\epsilon)}$ or $\alpha \leq \frac{\ln^2 2}{(1+1/m)\ln(1/\epsilon)}$.

Alternatively, if we fix ϵ and n , we need $m/(1+1/m) \geq n \cdot \frac{\ln(1/\epsilon)}{\ln^2 2} \approx 1.442n \lg(1/\epsilon)$, which works out to $m \geq 1.442n \lg(1/\epsilon) + O(1)$. This is very good for constant ϵ .

Note that for this choice of m , we have $\alpha = O(1/\ln(1/\epsilon))$, giving $k = O(\log(1/\epsilon))$. So for polynomial ϵ , we get $k = O(\log n)$. This is closer to the complexity of tree lookups than hash table lookups, so the main payoff for a sequential implementation is that we don't have to store full keys.

7.6.3 Comparison to optimal space

If we wanted to design a Bloom-filter-like data structure from scratch and had no constraints on processing power, we'd be looking for something that stored an index of size $\lg M$ into a family of subsets S_1, S_2, \dots, S_M of our universe of keys U , where $|S_i| \leq \epsilon|U|$ for each i (giving the upper bound on the false positive rate)⁷ and for any set $A \subseteq U$ of size n , $A \subseteq S_i$ for at least one S_i (allowing us to store A).

Let $N = |U|$. Then each set S_i covers $\binom{\epsilon N}{n}$ of the $\binom{N}{n}$ subsets of size n . If we could get them to overlap optimally (we can't), we'd still need a minimum of $\binom{N}{n} / \binom{\epsilon N}{n} = (N)_n / (\epsilon N)_n \approx (1/\epsilon)^n$ sets to cover everybody, where the approximation assumes $N \gg n$. Taking the log gives $\lg M \approx n \lg(1/\epsilon)$, meaning we need about $\lg(1/\epsilon)$ bits per key for the data structure. Bloom filters use $1/\ln 2$ times this.

There are known data structures that approach this bound asymptotically. The first of these, due to Pagh *et al.* [PPR05] also has other desirable properties, like supporting deletions and faster lookups if we can't look up bits in parallel.

More recently, Fan *et al.* [FAKM14] have described a variant of cuckoo hashing (see §7.4) called a **cuckoo filter**. This is a cuckoo hash table that, instead of storing full keys x , stores **fingerprints** $f(x)$, where f is a hash function with ℓ -bit outputs. False positives now arise if we happen to hash a value x' with $f(x') = f(x)$ to the same location as x . If f is drawn from a 2-universal family, this occurs with probability at most $2^{-\ell}$. So the idea is that by accepting an ϵ small rate of false positives, we can shrink the space needed to store each key from the full key length to $\lg(1/\epsilon) = \ln(1/\epsilon)/\ln 2$, the asymptotic minimum.

One complication is that, since we are throwing away the original key x , when we displace a key from $h_1(x)$ to $h_2(x)$ or vice versa, we can't recompute $h_1(x)$ and $h_2(x)$ for arbitrary h_1 and h_2 . The solution proposed by Fan *et al.* is to let $h_2(x) = h_1(x) \oplus g(f(x))$, where g is a hash function that depends

⁷Technically, this gives a weaker bound on false positives. For standard Bloom filters, assuming random hash functions, each key individually has at most an ϵ probability of appearing as a false positive. The hypothetical data structure we are considering here—which is effectively deterministic—allows the set of false positives to depend directly on the set of keys actually inserted in the data structure, meaning that the adversary could arrange for a specific key to appear as a false positive with probability 1 by choosing appropriate keys to insert. So this argument may underestimate the space needed to get make the false positives less predictable. On the other hand, we aren't charging the Bloom filter for the space needed to store the hash functions, which could be quite a bit if they are genuine random functions.

only on the fingerprint. This means that when looking at a fingerprint $f(x)$ stored in position i , we don't need to know whether i is $h_1(x)$ or $h_2(x)$, since whichever it is, the other location will be $i \oplus g(f(x))$. Unfortunately, this technique and some other techniques used in the paper to crunch out excess empty space break the standard analysis of cuckoo hashing, so the authors can only point to experimental evidence that their data structure actually works. However, a variant of this data structure has been shown to work by Eppstein [Epp16].

7.6.4 Applications

Historically, Bloom filters were invented to act as a way of filtering queries to a database table through fast but expensive⁸ RAM before looking up the actual values on a slow but cheap tape drive. Nowadays the cost of RAM is low enough that this is less of an issue in most cases, but Bloom filters are still popular in networking and in distributed databases.

In networking, Bloom filters are useful in building network switches, where incoming packets need to be matched against routing tables in fractions of a nanosecond. Bloom filters work particularly well for this when implemented in hardware, since the k hash functions can be computed in parallel. False positives, if infrequent enough, can be handled by some slower backup mechanism.

In distributed databases, Bloom filters are used in the **Bloomjoin** algorithm [ML86]. Here we want to do a join on two tables stored on different machines (a join is an operation where we find all pairs of rows, one in each table, that match on some common key). A straightforward but expensive way to do this is to send the list of keys from the smaller table across the network, then match them against the corresponding keys from the larger table. If there are n_s rows in the smaller table, n_b rows in the larger table, and j matching rows in the larger table, this requires sending n_s keys plus j rows. If instead we send a Bloom filter representing the set of keys in the smaller table, we only need to send $\lg(1/\epsilon)/\ln 2$ bits for the Bloom filter plus an extra ϵn_b rows on average for the false positives. This can be cheaper than sending full keys across if the number of false positives is reasonably small.

⁸As much as \$0.10/bit in 1970.

7.6.5 Counting Bloom filters

It's not hard to modify a Bloom filter to support deletion. The basic trick is to replace each bit with a counter, so that whenever a value x is inserted, we increment $A[h_i(x)]$ for all i and when it is deleted, we decrement the same locations. The search procedure now returns $\min_i A[h_i(x)]$ (which means that in principle it can even report back multiplicities, though with some probability of reporting a value that is too high). To avoid too much space overhead, each array location is capped at some small maximum value c ; once it reaches this value, further increments have no effect. The resulting structure is called a **counting Bloom filter**, due to Fan *et al.* [FCAB00].

We can only expect this to work if our chance of hitting the cap is small. Fan *et al.* observe that the probability that the m table entries include one that is at least c after n insertions is bounded by

$$\begin{aligned} m \binom{nk}{c} \frac{1}{m^c} &\leq m \left(\frac{enk}{c} \right)^c \frac{1}{m^c} \\ &= m \left(\frac{enk}{cm} \right)^c \\ &= m(ek\alpha/c)^c. \end{aligned}$$

(This uses the bound $\binom{n}{k} \leq \left(\frac{en}{k}\right)^k$, which follows from Stirling's formula.)

For $k = \frac{1}{\alpha} \ln 2$, this is $m(e \ln 2/c)^c$. For the specific value of $c = 16$ (corresponding to 4 bits per entry), they compute a bound of $1.37 \times 10^{-15}m$, which they argue is minuscule for all reasonable values of m (it's a systems paper).

The possibility that a long chain of alternating insertions and deletions might produce a false negative due to overflow is considered in the paper, but the authors state that “the probability of such a chain of events is so low that it is much more likely that the proxy server would be rebooted in the meantime and the entire structure reconstructed.” An alternative way of dealing with this problem is to never decrement a maxed-out register. This never produces a false negative, but may cause the filter to slowly fill up with maxed-out registers, producing a higher false-positive rate.

A fancier variant of this idea is the **spectral Bloom filter** of Cohen and Matias [CM03], which uses larger counters to track multiplicities of items. The essential idea here is that we can guess that the number of times a particular value x was inserted is equal to $\min_{i=1}^m A[h_i(x)]$, with some extra tinkering to detect errors based on deviations from the typical joint distribution of the $A[h_i(x)]$ values. An even more sophisticated approach gives the count-min sketches of the next section.

7.6.6 Count-min sketches

Count-min sketches are designed for use in **data stream computation**. In this model, we are given a huge flood of data—far too big to store—in a single pass, and want to incrementally build a small data structure, called a **sketch**, that will allow us to answer statistical questions about the data after we’ve processed it all. The motivation is the existence of data sets that are too large to store at all (network traffic statistics), or too large to store in fast memory (very large database tables). By building a sketch we can make one pass through the data set but answer queries after the fact, with some loss of accuracy.

An example of a problem in this model is that we are presented with a sequence of pairs (i_t, c_t) where $1 \leq i_t \leq n$ is an *index* and c_t is a *count*, and we want to construct a sketch that will allow us to approximately answer statistical queries about the vector a given by $a_i = \sum_{t, i[t]=i} c_t$. The size of the sketch should be polylogarithmic in the size of a and the length of the stream, and polynomial in the error bounds. Updating the sketch given a new data point should be cheap.

A solution to this problem is given by the **count-min sketch** of Cormode and Muthukrishnan [CM05] (see also [MU05, §13.4]). This gives approximations of a_i , $\sum_{i=\ell}^r a_i$, and $a \cdot b$ (for any fixed b), and can be used for more complex tasks like finding **heavy hitters**—indices with high weight. The easiest case is approximating a_i when all the c_t are non-negative, so we’ll start with that.

7.6.6.1 Initialization and updates

To construct a count-min sketch, build a two-dimensional array c with depth $d = \lceil \ln(1/\delta) \rceil$ and width $w = \lceil e/\epsilon \rceil$, where ϵ is the error bound and δ is the probability of exceeding the error bound. Choose d independent hash functions from some 2-universal hash family; we’ll use one of these hash functions for each row of the array. Initialize c to all zeros.

The update rule: Given an update (i_t, c_t) , increment $c[j, h_j(i_t)]$ by c_t for $j = 1 \dots d$. (This is the *count* part of count-min.)

7.6.6.2 Queries

Let’s start with **point queries**. Here we want to estimate a_i for some fixed i . There are two cases; the first handles non-negative increments only, while the second handles arbitrary increments. In both cases we will get an estimate whose error is linear in both the error parameter ϵ and the ℓ_1 -norm

$\|a\|_1 = \sum_i |a_i|$ of a . It follows that the relative error will be low for heavy points, but we may get a large relative error for light points (and especially large for points that don't appear in the data set at all).

For the non-negative case, to estimate a_i , compute $\hat{a}_i = \min_j c[j, h_j(i)]$. (This is the *min* part of coin-min.) Then:

Lemma 7.6.1. *When all c_t are non-negative, for \hat{a}_i as defined above:*

$$\hat{a}_i \geq a_i, \quad (7.6.2)$$

and

$$\Pr[\hat{a}_i \leq a_i + \epsilon \|a\|_1] \geq 1 - \delta. \quad (7.6.3)$$

Proof. The lower bound is easy. Since for each pair (i, c_t) we increment each $c[j, h_j(i)]$ by c_t , we have an invariant that $a_i \leq c[j, h_j(i)]$ for all j throughout the computation, which gives $a_i \leq \hat{a}_i = \min_j c[j, h_j(i)]$.

For the upper bound, let I_{ijk} be the indicator for the event that $(i \neq k) \wedge (h_j(i) = h_j(k))$, i.e., that we get a collision between i and k using h_j . The 2-universality property of the h_j gives $\mathbb{E}[I_{ijk}] \leq 1/w \leq \epsilon/e$.

Now let $X_{ij} = \sum_{k=1}^n I_{ijk} a_k$. Then $c[j, h_j(i)] = a_i + X_{ij}$. (The fact that $X_{ij} \geq 0$ gives an alternate proof of the lower bound.) Now use linearity of expectation to get

$$\begin{aligned} \mathbb{E}[X_{ij}] &= \mathbb{E}\left[\sum_{k=1}^n I_{ijk} a_k\right] \\ &= \sum_{k=1}^n a_k \mathbb{E}[I_{ijk}] \\ &\leq \sum_{k=1}^n a_k (\epsilon/e) \\ &= (\epsilon/e) \|a\|_1. \end{aligned}$$

So $\Pr[c[j, h_j(i)] > a_i + \epsilon \|a\|_1] = \Pr[X_{ij} > e \mathbb{E}[X_{ij}]] < 1/e$, by Markov's inequality. With d choices for j , and each h_j chosen independently, the probability that every count is too big is at most $(1/e)^{-d} = e^{-d} \leq \exp(-\ln(1/\delta)) = \delta$. \square

Now let's consider the general case, where the increments c_t might be negative. We still initialize and update the data structure as described in §7.6.6.1, but now when computing \hat{a}_i , we use the median count instead of the minimum count: $\hat{a}_i = \text{median}\{c[j, h_j(i)] \mid j = 1 \dots n\}$. Now we get:

Lemma 7.6.2. For \hat{a}_i as defined above,

$$\Pr [a_i - 3\epsilon\|a\|_1 \leq \hat{a}_i \leq a_i + 3\epsilon\|a\|_1] > 1 - \delta^{1/4}. \quad (7.6.4)$$

Proof. The basic idea is that for the median to be off by t , at least $d/2$ rows must give values that are off by t . We'll show that for $t = 3\epsilon\|a\|_1$, the expected number of rows that are off by t is at most $d/8$. Since the hash functions for the rows are chosen independently, we can use Chernoff bounds to show that with a mean of $d/8$, the chances of getting all the way to $d/2$ are small.

In detail, we again define the error term X_{ij} as above, and observe that

$$\begin{aligned} \mathbb{E}[|X_{ij}|] &= \mathbb{E}\left[\left|\sum_k I_{ijk}a_k\right|\right] \\ &\leq \sum_{k=1}^n |a_k| \mathbb{E}[I_{ijk}] \\ &\leq \sum_{k=1}^n |a_k|(\epsilon/e) \\ &= (\epsilon/e)\|a\|_1. \end{aligned}$$

Using Markov's inequality, we get $\Pr[|X_{ij}| > 3\epsilon\|a\|_1] = \Pr[|X_{ij}| > 3e\mathbb{E}[|X_{ij}|]] < 1/3e < 1/8$. In order for the median to be off by more than $3\epsilon\|a\|_1$, we need $d/2$ of these low-probability events to occur. The expected number that occur is $\mu = d/8$, so applying the standard Chernoff bound (5.2.1) with $\delta = 3$ we are looking at

$$\begin{aligned} \Pr[S \geq d/2] &= \Pr[S \geq (1+3)\mu] \\ &\leq (e^3/4^4)^{d/8} \\ &\leq (e^{3/8}/2)^{\ln(1/\delta)} \\ &= \delta^{\ln 2 - 3/8} \\ &< \delta^{1/4} \end{aligned}$$

(the actual exponent is about 0.31, but $1/4$ is easier to deal with). This immediately gives (7.6.4). \square

One way to think about this is that getting an estimate within $\epsilon\|a\|_1$ of the right value with probability at least $1 - \delta$ requires 3 times the width and 4 times the depth—or 12 times the space and 4 times the time—when we aren't assuming increments are non-negative.

Next, we consider inner products. Here we want to estimate $a \cdot b$, where a and b are both stored as count-min sketches using the same hash functions. The paper concentrates on the case where a and b are both non-negative, which has applications in estimating the size of a join in a database. The method is to estimate $a \cdot b$ as $\min_j \sum_{k=1}^w c_a[j, k] \cdot c_b[j, k]$.

For a single j , the sum consists of both good values and bad collisions; we have $\sum_{k=1}^w c_a[j, k] \cdot c_b[j, k] = \sum_{k=1}^n a_i b_i + \sum_{p \neq q, h_j(p)=h_j(q)} a_p b_q$. The second term has expectation

$$\begin{aligned} \sum_{p \neq q} \Pr [h_j(p) = h_j(q)] a_p b_q &\leq \sum_{p \neq q} (\epsilon/e) a_p b_q \\ &\leq \sum_{p, q} (\epsilon/e) a_p b_q \\ &\leq (\epsilon/e) \|a\|_1 \|b\|_1. \end{aligned}$$

As in the point-query case, we get probability at most $1/e$ that a single j gives a value that is too high by more than $\epsilon \|a\|_1 \|b\|_1$, so the probability that the minimum value is too high is at most $e^{-d} \leq \delta$.

7.6.6.3 Finding heavy hitters

Here we want to find the heaviest elements in the set: those indices i for which a_i exceeds $\phi \|a\|_1$ for some constant threshold $0 < \phi \leq 1$.

The easy case is when increments are non-negative (for the general case, see the paper), and uses a method from a previous paper by Charikar *et al.* [CCFC04]. Because $\|a\|_1 = \sum_i a_i$, we know that there will be at most $1/\phi$ heavy hitters. But the tricky part is figuring out which elements they are.

Instead of trying to find the elements after the fact, we extend the data structure and update procedure to track all the heavy elements found so far (stored in a heap), as well as $\|a\|_1 = \sum c_t$. When a new increment (i, c) comes in, we first update the count-min structure and then do a point query on a_i ; if $\hat{a}_i \geq \phi \|a\|_1$, we insert i into the heap, and if not, we delete i along with any other value whose stored point-query estimate has dropped below threshold.

The trick here is that the threshold $\phi \|a\|_1$ only increases over time (remember that we are assuming non-negative increments). So if some element i is below threshold at time t , it can only go above threshold if it shows up again, and we have a probability of at least $1 - \delta$ of including it then. This means that every heavy hitter appears in the heap with probability at least $1 - \delta$.

The total space cost for this data structure is the cost of the count-min structure plus the cost of the heap; this last part will be $O((1 + \epsilon)/\phi)$ with reasonably high probability, since this is the maximum number of elements that have weight at least $\phi\|a\|_1/(1 + \epsilon)$, the minimum needed to get an apparent weight of $\phi\|a\|_1$ even after taking into account the error in the count-min structure.

7.7 Locality-sensitive hashing

Locality-sensitive hashing was invented by Indyk and Motwani [IM98] to solve the problem of designing a data structure that finds approximate nearest neighbors to query points in high dimension. We'll mostly be following this paper in this section, concentrating on the hashing parts.

7.7.1 Approximate nearest neighbor search

In the **nearest neighbor search** problem (NNS for short), we are given a set of n points P in a metric space with distance function d , and we want to construct a data structure that allows us to quickly find the closet point p in P to any given query point q . We could always compute the distance between q and each possible p , but this takes time $O(n)$, and we'd like to get lookups to be sublinear in n .

Indyk and Motwani were particularly interested in what happens in \mathbb{R}^d for high dimension d under various natural metrics. Because the volume of a ball in a high-dimensional space grows exponentially with the dimension, this problem suffers from the **curse of dimensionality** [Bel57]: simple techniques based on, for example, assigning points in P to nearby locations in a grid may require searching exponentially many grid locations. Indyk and Motwani deal with this through a combination of randomization and solving the weaker problem of **ϵ -nearest neighbor search** (ϵ -NNS), where it's OK to return a different point p' as long as $d(q, p') \leq (1 + \epsilon) \min_{p \in P} d(q, p)$.

This problem can be solved by reduction to a simpler problem called **ϵ -point location in equal balls** or ϵ -PLEB. In this problem, we are given n radius- r balls centered on points c in a set C , and we want a data structure that returns a point $c' \in C$ with $d(q, c') \leq (1 + \epsilon)r$ if there is at least one point c with $d(q, c) \leq r$. If there is no such point, the data structure may or may not return a point (it might say no, or it might just return a point that is too far away, which we can discard). The difference between an ϵ -PLEB and NNS is that an ϵ -PLEB isn't picky about returning the closest point to

q if there are multiple points that are all good enough. Still, we can reduce NNS to ϵ -PLEB.

The easy reduction is to use binary search. Let $R = \frac{\max_{x,y \in P} d(x,y)}{\min_{x,y \in P, x \neq y} d(x,y)}$. Given a point q , look for the minimum $\ell \in \{(1+\epsilon)^0, (1+\epsilon)^1, \dots, R\}$ for which an ϵ -PLEB data structure with radius ℓ and centers P returns a point p with $d(q,p) \leq (1+\epsilon)\ell$; then return this point as the approximate nearest neighbor.

This requires $O(\log_{1+\epsilon} R)$ instances of the ϵ -PLEB data structure and $O(\log \log_{1+\epsilon} R)$ queries. The blowup as a function of R can be avoided using a more sophisticated data structure called a **ring-cover tree**, defined in the paper. We won't talk about ring-cover trees because they are (a) complicated and (b) not randomized. Instead, we'll move directly to the question of how we solve ϵ -PLEB.

7.7.1.1 Locality-sensitive hash functions

Definition 7.7.1 ([IM98]). *A family of hash functions H is (r_1, r_2, p_1, p_2) -sensitive for d if, for any points p and q , if h is chosen uniformly from H ,*

1. *If $d(p, q) \leq r_1$, then $\Pr[h(p) = h(q)] \geq p_1$, and*
2. *If $d(p, q) > r_2$, then $\Pr[h(p) = h(q)] \leq p_2$.*

These are useful if $p_1 > p_2$ and $r_1 < r_2$; that is, we are more likely to hash inputs together if they are closer. Ideally, we can choose r_1 and r_2 to build ϵ -PLEB data structures for a range of radii sufficient to do binary search as described above (or build a ring-cover tree if we are doing it right). For the moment, we will aim for an (r_1, r_2) -PLEB data structure, which returns a point within r_1 with high probability if one exists, and never returns a point farther away than r_2 .

There is some similarity between locality-sensitive hashing and a more general dimension-reduction technique known as the **Johnson-Lindenstrauss lemma** [JL84]; this says that projecting n points in a high-dimensional space to $O(\epsilon^{-2} \log n)$ dimensions using an appropriate random matrix preserves ℓ_2 distances between the points to within relative error ϵ (in fact, even a random matrix with ± 1 entries is enough [Ach03]). Unfortunately, dimension reduction by itself is not enough to solve approximate nearest neighbors in sublinear time, because we may still need to search a number of boxes exponential in $O(\epsilon^{-2} \log n)$, which will be polynomial in n .

7.7.1.2 Constructing an (r_1, r_2) -PLEB

The first trick is to amplify the difference between p_1 and p_2 so that we can find a point within r_1 of our query point q if one exists. This is done in three stages: First, we concatenate multiple hash functions to drive the probability that distant points hash together down until we get few collisions: the idea here is that we are taking the AND of the events that we get collisions in the original hash function. Second, we hash our query point and target points multiple times to bring the probability that nearby points hash together up: this is an OR. Finally, we iterate the procedure to drive down any remaining probability of failure below a target probability δ : another AND.

For the first stage, let $k = \log_{1/p_2} n$ and define a composite hash function $g(p) = (h_1(p) \dots h_k(p))$. If $d(p, q) > r_2$, $\Pr [g(p) = g(q)] \leq p_2^k = p_2^{\log_{1/p_2} n} = 1/n$. Adding this up over all n points in our data structure gives us one false match for q on average.

However, we may not be able to find the correct match for q , since p_1 may not be all that much larger than p_2 . For this, we do a second round of amplification, where now we are taking the OR of events we want instead of the AND of events we don't want.

Let $\ell = n^\rho$, where $\rho = \frac{\log(1/p_1)}{\log(1/p_2)} = \frac{\log p_1}{\log p_2} < 1$, and choose hash functions $g_1 \dots g_\ell$ independently as above. To store a point p , put it in a bucket for $g_j(p)$ for each j ; these buckets are themselves stored in a hash table (by hashing the value of $g_j(p)$ down further) so that they fit in $O(n)$ space. Suppose now that $d(p, q) \leq r_1$ for some p . Then

$$\begin{aligned} \Pr [g_j(p) = g_j(q)] &\geq p_1^k \\ &= p_1^{\log_{1/p_2} n} \\ &= n^{-\frac{\log 1/p_1}{\log 1/p_2}} \\ &= n^{-\rho} \\ &= 1/\ell. \end{aligned}$$

So by searching through ℓ independent buckets we find p with probability at least $1 - (1 - 1/\ell)^\ell = 1 - 1/e + o(1)$. We'd like to guarantee that we only have to look at $O(n^\rho)$ points (most of which we may reject) during this process; but we can do this by stopping if we see more than 2ℓ points. Since we only expect to see ℓ bad points in all ℓ buckets, this event only happens with probability $1/2$. So even adding it to the probability of failure from the hash functions not working right we still have only a constant probability of failure $1/e + 1/2 + o(1)$.

Iterating the entire process $O(\log(1/\delta))$ times then gives the desired bound δ on the probability that this process fails to find a good point if one exists.

Multiplying out all the costs gives a cost of a query of $O(k\ell \log(1/\delta)) = O\left(n^\rho \log_{1/p_2} n \log(1/\delta)\right)$ hash function evaluations and $O(n^\rho \log(1/\delta))$ distance computations. The cost to insert a point is just $O(k\ell \log(1/\delta)) = O\left(n^\rho \log_{1/p_2} n \log(1/\delta)\right)$ hash function evaluations, the same number as for a query.

7.7.1.3 Hash functions for Hamming distance

Suppose that our points are d -bit vectors and that we use Hamming distance for our metric. In this case, using the family of one-bit projections $\{h_i \mid h_i(x) = x_i\}$ gives a locality-sensitive hash family [ABMRT96].

Specifically, we can show this family is $(r, r(1 + \epsilon), 1 - \frac{r}{d}, 1 - \frac{r(1+\epsilon)}{d})$ -sensitive. The argument is trivial: if two points p and q are at distance r or less, they differ in at most r places, so the probability that they hash together is just the probability that we don't pick one of these places, which is at least $1 - \frac{r}{d}$. Essentially the same argument works when p and q are far away.

These are not particularly clever hash functions, so the heavy lifting will be done by the (r_1, r_2) -PLEB construction. Our goal is to build an ϵ -PLEB for any fixed r , which will correspond to an $(r, r(1 + \epsilon))$ -PLEB. The main thing we need to do, following [IM98] as always, is compute a reasonable bound on $\rho = \frac{\log p_1}{\log p_2} = \frac{\ln(1-r/d)}{\ln(1-(1+\epsilon)r/d)}$. This is essentially just a matter of hitting it with enough inequalities, although there are a couple of tricks in the middle.

Compute

$$\begin{aligned}
 \rho &= \frac{\ln(1 - r/d)}{\ln(1 - (1 + \epsilon)r/d)} \\
 &= \frac{(d/r) \ln(1 - r/d)}{(d/r) \ln(1 - (1 + \epsilon)r/d)} \\
 &= \frac{\ln((1 - r/d)^{d/r})}{\ln((1 - (1 + \epsilon)r/d)^{d/r})} \\
 &\leq \frac{\ln(e^{-1}(1 - r/d))}{\ln e^{-(1+\epsilon)}} \\
 &= \frac{-1 + \ln(1 - r/d)}{-(1 + \epsilon)} \\
 &= \frac{1}{1 + \epsilon} - \frac{\ln(1 - r/d)}{1 + \epsilon}. \tag{7.7.1}
 \end{aligned}$$

Note that we used the fact that $1 + x \leq e^x$ for all x in the denominator and $(1 - x)^{1/x} \geq e^{-1}(1 - x)$ for $x \in [0, 1]$ in the numerator. The first fact is our usual favorite inequality.

The second can be proved in a number of ways. The most visually intuitive is that $(1 - x)^{1/x}$ and $e^{-1}(1 - x)$ are equal at $x = 1$ and equal in the limit as x goes to 0, while $(1 - x)^{1/x}$ is concave in between 0 and 1 and $e^{-1}(1 - x)$ is linear. Unfortunately it is rather painful to show that $(1 - x)^{1/x}$ is in fact concave. An alternative is to rewrite the inequality $(1 - x)^{1-x} \geq e^{-1}(1 - x)$ as $(1 - x)^{1/x-1} \geq e^{-1}$, apply a change of variables $y = 1/x$ to get $(1 - 1/y)^{y-1} \geq e^{-1}$ for $y \in [1, \infty)$, and then argue that (a) equality holds in the limit as y goes to infinity, and (b) the left-hand-side is a nonincreasing function, since

$$\begin{aligned}
 \frac{d}{dy} \ln \left((1 - 1/y)^{y-1} \right) &= \frac{d}{dy} [(y - 1) (\ln(y - 1) - \ln y)] \\
 &= \ln(1 - 1/y) + (y - 1) \left(\frac{1}{y - 1} - \frac{1}{y} \right) \\
 &= \ln(1 - 1/y) + 1 - (1 - 1/y) \\
 &= \ln(1 - 1/y) + 1/y \\
 &\leq -1/y + 1/y \\
 &= 0.
 \end{aligned}$$

We now return to (7.7.1). We'd really like the second term to be small enough that we can just write n^ρ as $n^{1/(1+\epsilon)}$. (Note that even though it looks

negative, it isn't, because $\ln(1 - r/d)$ is negative.) So we pull a rabbit out of a hat by assuming that $r/d < 1/\ln n$.⁹ This assumption can be justified by modifying the algorithm so that d is padded out with up to $d \ln n$ unused junk bits if necessary. Using this assumption, we get

$$\begin{aligned} n^\rho &< n^{1/(1+\epsilon)} n^{-\ln(1-1/\ln n)/(1+\epsilon)} \\ &= n^{1/(1+\epsilon)} (1 - 1/\ln n)^{-\ln n} \\ &\leq e n^{1/(1+\epsilon)}. \end{aligned}$$

Plugging into the formula for (r_1, r_2) -PLEB gives $O(n^{1/(1+\epsilon)} \log n \log(1/\delta))$ hash function evaluations per query, each of which costs $O(1)$ time, plus $O(n^{1/(1+\epsilon)} \log(1/\delta))$ distance computations, which will take $O(d)$ time each. If we add in the cost of the binary search, we have to multiply this by $O(\log \log_{1+\epsilon} R \log \log \log_{1+\epsilon} R)$, where the log-log-log comes from having to adjust δ so that the error doesn't accumulate too much over all $O(\log \log R)$ steps. The end result is that we can do approximate nearest-neighbor queries in

$$O\left(n^{1/(1+\epsilon)} \log(1/\delta) (\log n + d) \log \log_{1+\epsilon} R \log \log \log_{1+\epsilon} R\right)$$

time. For ϵ reasonably large, this is much better than naively testing against all points in our database, which takes $O(nd)$ time (although it does produce an exact result).

7.7.1.4 Hash functions for ℓ_1 distance

Essentially the same approach works for (bounded) ℓ_1 distance, using **discretization**, where we replace a continuous variable over some range with a discrete variable. Suppose we are working in $[0, 1]^d$ with the ℓ_1 metric. Represent each coordinate x_i as a sequence of d/ϵ values x_{ij} in **unary**, for $j = 1 \dots \epsilon d$, with $x_{ij} = 1$ if $\epsilon j/d < x_i$. Then the Hamming distance between the bit-vectors representing x and y is proportional to the ℓ_1 distance between the original vectors, plus an error term that is bounded by ϵ . We can then use the hash functions for Hamming distance to get a locality-sensitive hash family.

A nice bit about this construction is that we don't actually have to build the bit-vectors; instead, we can specify a coordinate x_i and a threshold c and get the same effect by recording whether $x_i > c$ or not.

⁹Indyk and Motwani pull this rabbit out of a hat a few steps earlier, but it's pretty much the same rabbit either way.

Note that this does increase the cost slightly: we are converting d -dimensional vectors into (d/ϵ) -long bit vectors, so the $\log(n+d)$ term becomes $\log(n+d/\epsilon)$. When n is small, this effectively multiplies the cost of a query by an extra $\log(1/\epsilon)$. More significant is that we have to cut ϵ in half to obtain the same error bounds, because we now pay ϵ error for the data structure itself and an additional ϵ error for the discretization. So our revised cost for the ℓ_1 case is

$$O\left(n^{1/(1+\epsilon/2)} \log(1/\delta)(\log n + d/\epsilon) \log \log_{1+\epsilon/2} R \log \log_{1+\epsilon/2} R\right).$$

Chapter 8

Martingales and stopping times

Last updated 2019. Some material may be out of date.

In §5.3.2, we used martingales to show that the outcome of some process was tightly concentrated. Here we will show how martingales interact with **stopping times**, which are random variables that control when we stop carrying out some task. This will require a few new definitions.

8.1 Definitions

The general form of a martingale $\{X_t, \mathcal{F}_t\}$ consists of:

- A sequence of random variables X_0, X_1, X_2, \dots ; and
- A **filtration** $\mathcal{F}_0 \subseteq \mathcal{F}_1 \subseteq \mathcal{F}_2 \dots$, where each σ -algebra \mathcal{F}_t represents our knowledge at time t ;

subject to the requirements that:

1. The sequence of random variables is **adapted** to the filtration, which just means that each X_t is \mathcal{F}_t -measurable or equivalently that \mathcal{F}_t (and thus all subsequent $\mathcal{F}_{t'}$ for $t' \geq t$) includes all knowledge of X_t ; and
2. The **martingale property**

$$E[X_{t+1} \mid \mathcal{F}_t] = X_t \tag{8.1.1}$$

holds for all t .

Together with this more general definition of a martingale, we will also use the following definition of a **stopping time**. Given a filtration $\{\mathcal{F}_t\}$, a random variable τ is a stopping time for $\{\mathcal{F}_t\}$ if $\tau \in \mathbb{N} \cup \{\infty\}$ and the event $[\tau \leq t]$ is \mathcal{F}_t -measurable for all $t \in \mathbb{N}$.¹ In simple terms, τ is a stopping time if you know at time t whether to stop there or not.

What we like about martingales is that iterating the martingale property shows that $E[X_t] = E[X_0]$ for all fixed t . We will show that, under reasonable conditions, the same holds for X_τ when τ is a stopping time. (The random variable X_τ is defined in the obvious way, as a random variable that takes on the value of X_t when $\tau = t$.)

8.2 Submartingales and supermartingales

In some cases we have a process where instead of getting equality in (8.1.1), we get an inequality instead. A **submartingale** replaces (8.1.1) with

$$X_t \leq E[X_{t+1} \mid \mathcal{F}_t] \quad (8.2.1)$$

while a **supermartingale** satisfies

$$X_t \geq E[X_{t+1} \mid \mathcal{F}_t]. \quad (8.2.2)$$

In each case, what is “sub” or “super” is the value at the current time compared to the expected value at the next time. Intuitively, a submartingale corresponds to a process where you win on average, while a supermartingale is a process where you lose on average. Casino games (in profitable casinos) are submartingales for the house and supermartingales for the player.

Sub- and supermartingales can be reduced to martingales by subtracting off the expected change at each step. For example, if $\{X_t\}$ is a submartingale with respect to $\{\mathcal{F}_t\}$, then the process $\{Y_t\}$ defined recursively by

$$\begin{aligned} Y_0 &= X_0 \\ Y_{t+1} &= Y_t + X_{t+1} - E[X_{t+1} \mid \mathcal{F}_t] \end{aligned}$$

¹Different authors impose different conditions on the range of τ ; for example, Mitzenmacher and Upfal [MU05] exclude the case $\tau = \infty$. We allow $\tau = \infty$ to represent the outcome where we never stop. This can be handy for modeling processes where this outcome is possible, although in practice we will typically insist that it occurs only with probability zero.

is a martingale, since

$$\begin{aligned} \mathbb{E}[Y_{t+1} | \mathcal{F}_t] &= \mathbb{E}[Y_t + X_{t+1} - \mathbb{E}[X_{t+1} | \mathcal{F}_t] | \mathcal{F}_t] \\ &= Y_t + \mathbb{E}[X_{t+1} | \mathcal{F}_t] - \mathbb{E}[X_{t+1} | \mathcal{F}_t] \\ &= Y_t. \end{aligned}$$

One way to think of this is that $Y_t = X_t + \Delta_t$, where Δ_t is a predictable, non-decreasing **drift process** that starts at 0. For supermartingales, the same result holds, but now Δ_t is non-increasing. This ability to decompose an adapted stochastic process into the sum of a martingale and a predictable drift process is known as the **Doob decomposition theorem**.

8.3 The optional stopping theorem

If (X_t, \mathcal{F}_t) is a martingale, then applying induction to the martingale property shows that $\mathbb{E}[X_t] = \mathbb{E}[X_0]$ for any fixed time t . The **optional stopping theorem** shows that this also happens for X_τ when τ is a stopping time, under various choices of additional conditions:

Theorem 8.3.1. *Let (X_t, \mathcal{F}_t) be a martingale and τ a stopping time for $\{\mathcal{F}_t\}$. Then $\mathbb{E}[X_\tau] = \mathbb{E}[X_0]$ if at least one of the following conditions holds:*

1. **Bounded time.** *There is a fixed n such that $\tau \leq n$ always.*
2. **Finite time and bounded range.** *$\Pr[\tau < \infty] = 1$, and there is a fixed M such that for all $t \leq \tau$, $|X_t| \leq M$.*
3. **Finite expected time and bounded increments.** *$\mathbb{E}[\tau] < \infty$, and there is a fixed c such that $|X_{t+1} - X_t| \leq c$ for all $t < \tau$.*
4. **General case.** *All three of the following conditions hold:*

- (a) $\Pr[\tau < \infty] = 1$,
- (b) $\mathbb{E}[|X_\tau|] < \infty$, and
- (c) $\lim_{t \rightarrow \infty} \mathbb{E}[X_t \cdot 1_{[\tau > t]}] = 0$.

It would be nice if we could show $\mathbb{E}[X_\tau] = \mathbb{E}[X_0]$ without the side conditions, but in general this isn't true. For example, the double-after-losing martingale strategy in the St. Petersburg paradox (see §3.4.1.1) eventually yields +1 with probability 1, so if τ is the time we stop playing, we have $\Pr[\tau < \infty] = 1$, $\mathbb{E}[|X_\tau|] < \infty$, but $\mathbb{E}[X_\tau] = 1 \neq \mathbb{E}[X_0] = 0$. But in order for

this to happen, we have to violate all of bounded time (τ is not bounded), bounded range ($|X_t|$ roughly doubles every step until we stop), bounded increments ($|X_{t+1} - X_t|$ doubles every step as well), or one of the three conditions of the general case (the last one: $\lim_{t \rightarrow \infty} \mathbb{E}[X_t] \cdot 1_{[\tau > t]} = -1 \neq 0$).

The next section gives a proof of the optional stopping theorem. The intuition is that for any fixed n , we can **truncate** X_τ to $X_{\min(\tau, n)}$ and show that $\mathbb{E}[X_{\min(\tau, n)}] = \mathbb{E}[X_0]$. This immediately gives the bounded time case. For the other cases, the argument is that $\lim_{n \rightarrow \infty} \mathbb{E}[X_{\min(\tau, n)}]$ converges to $\mathbb{E}[X_\tau]$ provided the missing part $\mathbb{E}[X_\tau - X_{\min(\tau, n)}]$ converges to zero. How we do this depends on which assumptions we are making.

A proof of the optional stopping theorem is given in the next section. You probably don't need to understand this proof to apply the theorem, and may wish to skip directly to applications in §8.5.

8.4 Proof of the optional stopping theorem (optional)

We start with the finite-time case, which we can enforce by truncating the actual process.

Lemma 8.4.1. *Let (X_t, \mathcal{F}_t) be a martingale and τ a stopping time for $\{\mathcal{F}_t\}$. Then for any $n \in \mathbb{N}$, $\mathbb{E}[X_{\min(\tau, n)}] = \mathbb{E}[X_0]$.*

Proof. Define $Y_t = X_0 + \sum_{i=1}^t (X_i - X_{i-1})1_{[\tau > i-1]}$. Then (Y_t, \mathcal{F}_t) is a martingale, because we can calculate $\mathbb{E}[Y_{t+1} | \mathcal{F}_t] = \mathbb{E}[Y_t + (X_{t+1} - X_t)1_{[\tau > t]} | \mathcal{F}_t] = Y_t + 1_{[\tau > t]} \cdot \mathbb{E}[X_{t+1} - X_t | \mathcal{F}_t] = Y_t$; effectively, we are treating $1_{[\tau \leq t-1]}$ as a sequence of bets, and we know that adjusting our bets doesn't change the martingale property. But then $\mathbb{E}[X_{\min(\tau, n)}] = \mathbb{E}[Y_n] = \mathbb{E}[Y_0] = \mathbb{E}[X_0]$. \square

This gives us the bounded-time variant for free: If $\tau \leq n$ always, then $X_\tau = X_{\min(\tau, n)}$, and $\mathbb{E}[X_\tau] = \mathbb{E}[X_{\min(\tau, n)}] = \mathbb{E}[X_0]$.

For the unbounded-time variants, we will apply some version of the following strategy:

1. Observe that since $\mathbb{E}[X_{\min(\tau, n)}] = \mathbb{E}[X_0]$ is a constant for any fixed n , $\lim_{n \rightarrow \infty} \mathbb{E}[X_{\min(\tau, n)}]$ converges to $\mathbb{E}[X_0]$.

2. Argue using whatever assumptions we are making that $\lim_{n \rightarrow \infty} \mathbb{E} [X_{\min(\tau, n)}]$ also converges to $\mathbb{E} [X_\tau]$.
3. Conclude that $\mathbb{E} [X_0] = \mathbb{E} [X_\tau]$, since they are both limits of the same sequence.

For the middle step, start with

$$X_\tau = X_{\min(\tau, n)} + 1_{[\tau > n]}(X_\tau - X_n).$$

This holds because either $\tau \leq n$, and we just get X_τ , or $\tau > n$, and we get $X_n + (X_\tau - X_n) = X_\tau$.

Taking the expectation of both sides gives

$$\begin{aligned} \mathbb{E} [X_\tau] &= \mathbb{E} [X_{\min(\tau, n)}] + \mathbb{E} [1_{[\tau > n]}(X_\tau - X_n)]. \\ &= \mathbb{E} [X_0] + \mathbb{E} [1_{[\tau > n]}(X_\tau - X_n)]. \end{aligned}$$

So if we can show that the right-hand term goes to zero in the limit, we are done.

For the bounded-range case, we have $|X_\tau - X_n| \leq 2M$, so $|\mathbb{E} [1_{[\tau > n]}(X_\tau - X_n)]| \leq 2M \cdot \Pr [\tau > n]$. Since in this case we assume $\Pr [\tau < \infty] = 1$, $\lim_{n \rightarrow \infty} \Pr [\tau > n] = 0$, and the theorem holds.

For bounded increments, we have

$$\begin{aligned} |\mathbb{E} [(X_\tau - X_n)1_{[\tau > n]}]| &= \left| \mathbb{E} \left[\sum_{t \geq n} (X_{t+1} - X_t) 1_{[\tau > t]} \right] \right| \\ &\leq \mathbb{E} \left[\sum_{t \geq n} |(X_{t+1} - X_t)| 1_{[\tau > t]} \right] \\ &\leq \mathbb{E} \left[\sum_{t \geq n} c 1_{[\tau > t]} \right] \\ &\leq c \mathbb{E} \left[\sum_{t \geq n} 1_{[\tau > t]} \right]. \end{aligned}$$

But $\mathbb{E} [\tau] = \sum_{t=0}^{\infty} 1_{[\tau > t]}$. Under the assumption that this sequence converges, its tail goes to zero, and again the theorem holds.

For the general case, we can expand

$$\mathbb{E} [X_\tau] = \mathbb{E} [X_{\min(\tau, n)}] + \mathbb{E} [1_{[\tau > n]}X_\tau] - \mathbb{E} [1_{[\tau > n]}X_n]$$

which implies

$$\lim_{n \rightarrow \infty} E[X_\tau] = \lim_{n \rightarrow \infty} E[X_{\min(\tau, n)}] + \lim_{n \rightarrow \infty} E[1_{[\tau > n]}X_\tau] - \lim_{n \rightarrow \infty} E[1_{[\tau > n]}X_n],$$

assuming all these limits exist and are finite. We've already established that the first limit is $E[X_0]$, which is exactly what we want. So we just need to show that the other two limits both converge to zero. For the last limit, we just use condition (4c), which gives $\lim_{n \rightarrow \infty} E[1_{[\tau > n]}X_n] = 0$; no further argument is needed. But we still need to show that the middle limit also vanishes.

Here we use condition (4b). Observe that $E[1_{[\tau > n]}X_\tau] = \sum_{t=n+1}^{\infty} E[1_{[\tau=t]}X_t]$. Compare this with $E[X_\tau] = \sum_{t=0}^{\infty} E[1_{[\tau=t]}X_t]$; this is an absolutely convergent series (this is why we need condition (4b)), so in the limit the sum of the terms for $i = 0 \dots n$ converges to $E[X_\tau]$. But this means that the sum of the remaining terms for $i = n + 1 \dots \infty$ converges to zero. So the middle term goes to zero as n goes to infinity. This completes the proof.

8.5 Applications

Here we give some example of the Optional Stopping Theorem in action. In each case, the trick is to find an appropriate martingale and stopping time, and let the theorem do all the work.

8.5.1 Random walks

Let X_t be an **unbiased ± 1 random walk** that starts at 0, adds ± 1 to its current position with equal probability at each step, and stops if it reaches $-a$ or $+b$.² We'd like to calculate the probability of reaching $+b$ before $-a$. Let τ be the time at which the process stops.

We can easily show that $\Pr[\tau < \infty] = 1$ and $E[\tau] < \infty$ by observing that from any state of the random walk, there is a probability of at least $2^{-(a+b)}$ that it stops within $a + b$ steps (by flipping heads $a + b$ times in a row), so that if we consider a sequence of intervals of length $a + b$, the expected number of such intervals we can have before we stop is at most 2^{a+b} , giving $E[\tau] \leq (a + b)2^{a+b}$ (we can do better than this).

We also have bounded increments by the definition of the process (bounded range also works, at least up until time τ). So $E[X_\tau] = E[X_0] = 0$

²This is called a **random walk with two absorbing barriers**.

and the probability p of landing on $+b$ instead of $-a$ must satisfy $pb - (1 - p)a = 0$, giving $p = \frac{a}{a+b}$.

Now suppose we want to find $E[\tau]$. Let $Y_t = X_t^2 - t$. Then $Y_{t+1} = (X_t \pm 1)^2 - (t+1) = X_t^2 \pm 2X_t + 1 - (t+1) = (X_t^2 - t) \pm 2X_t = Y_t \pm 2X_t$. Since the plus and minus cases are equally likely, they cancel out in expectation and $E[Y_{t+1} | \mathcal{F}_t] = Y_t$: we just showed Y_t is a martingale.³ We can also show it has bounded increments (at least up until time τ), because $|Y_{t+1} - Y_t| = 2|X_t| \leq \max(a, b)$.

From Theorem 8.3.1, $E[Y_\tau] = 0$, which gives $E[\tau] = E[X_\tau^2]$. But we can calculate $E[X_\tau^2]$: it is $a^2 \Pr[X_\tau = -a] + b^2 \Pr[X_\tau = b] = a^2(b/(a+b)) + b^2(a/(a+b)) = (a^2b + b^2a)/(a+b) = ab$.

If we have a random walk that only stops at $+b$,⁴ then if τ is the first time at which $X_\tau = b$, τ is a stopping time. However, in this case

³This construction generalizes in a nice way to arbitrary martingales. Suppose $\{X_t\}$ is a martingale with respect to $\{\mathcal{F}_t\}$. Let $\Delta_t = X_t - X_{t-1}$, and let $V_t = \text{Var}[\Delta_t | \mathcal{F}_{t-1}]$ be the conditional variance of the t -th increment (note that this is a random variable that may depend on previous outcomes). We can easily show that $Y_t = X_t^2 - \sum_{i=1}^t V_i$ is a martingale. The proof is that

$$\begin{aligned} E[Y_t | \mathcal{F}_{t-1}] &= E\left[X_t^2 - \sum_{i=1}^t V_i \mid \mathcal{F}_{t-1}\right] \\ &= E\left[(X_{t-1} + \Delta_t)^2 \mid \mathcal{F}_{t-1}\right] - \sum_{i=1}^t V_i \\ &= E\left[X_{t-1}^2 + 2X_{t-1}\Delta_t + \Delta_t^2 \mid \mathcal{F}_{t-1}\right] - \sum_{i=1}^t V_i \\ &= X_{t-1}^2 + 2X_{t-1}E[\Delta_t | \mathcal{F}_{t-1}] + E[\Delta_t^2 | \mathcal{F}_{t-1}] - \sum_{i=1}^t V_i \\ &= X_{t-1}^2 + 0 + V_t - \sum_{i=1}^t V_i \\ &= X_{t-1}^2 - \sum_{i=1}^{t-1} V_i \\ &= Y_{t-1}. \end{aligned}$$

For the ± 1 random walk case, we have $V_t = 1$ always, giving $\sum_{i=1}^t V_i = t$ and $E[X_\tau^2] = E[X_0^2] + E[\tau]$ when τ is a stopping time satisfying the conditions of the Optional Stopping Theorem. For the general case, the same argument gives $E[X_\tau^2] = E[X_0^2] + E[\sum_{t=1}^\tau V_t]$ instead: the expected square position of X_t is incremented by the conditional variance at each step.

⁴This would be a **random walk with one absorbing barrier**.

$E[X_\tau] = b \neq E[X_0] = 0$. So the optional stopping theorem doesn't apply in this case. But we have bounded increments, so Theorem 8.3.1 would apply if $E[\tau] < \infty$. It follows that the expected time until we reach b is unbounded, either because sometimes we never reach b , or because we always reach b but sometimes it takes a very long time. ⁵

We can also consider a **biased random walk** where $+1$ occurs with probability p and -1 with probability $q = 1 - p$. If X_t is the position of the random walk at time t , and \mathcal{F}_t is the associated σ -algebra, then X_t isn't a martingale with respect to \mathcal{F}_t . But there are at least two ways to turn it into one:

1. Define $Y_t = X_t - (p - q)t$. Then

$$\begin{aligned} E[Y_{t+1} | \mathcal{F}_t] &= E[X_{t+1} - (p - q)(t + 1) | \mathcal{F}_t] \\ &= p(X_t + 1)\mathcal{F}_t + q(X_t - 1)\mathcal{F}_t - (p - q)(t + 1) \\ &= (p + q)X_t + (p - q) - (p - q)(t + 1) \\ &= X_t - (p - q)t \\ &= Y_t, \end{aligned}$$

and Y_t is a martingale with respect to \mathcal{F}_t .

2. Define $Z_t = (q/p)^{X_t}$. Then

$$\begin{aligned} E[Z_{t+1} | \mathcal{F}_t] &= p(q/p)^{X_t+1} + q(q/p)^{X_t-1} \\ &= c^{X_t}(p(q/p) + q(p/q)) \\ &= c^{X_t}(q + p) \\ &= c^{X_t} \\ &= Z_t. \end{aligned}$$

Again we have a martingale with respect to \mathcal{F}_t .

Now let's see what we can do with the Optional Stopping Theorem.

⁵In fact, we always reach b . An easy way to see this is to imagine a sequence of intervals of length n_1, n_2, \dots , where $n_{i+1} = \left(b + \sum_{j=1}^i n_j\right)^2$. At the end of the i -th interval, we are no lower than $-\sum_{j=0}^i n_j$, so we only need to go up $\sqrt{n_{i+1}}$ positions to reach a by the end of the $(i + 1)$ -th interval. Since this is just one standard deviation, it occurs with constant probability, so after a finite expected number of intervals, we will reach $+a$. Since there are infinitely many intervals, we reach $+a$ with probability 1.

1. Suppose we start with $X_0 = 0$ and we want to know the probability p_b that we will reach $+b$ before we reach $-a$.

Let τ be the first time at which $X_\tau \in \{-a, b\}$. We can use the same argument as in the unbiased case to show that $\Pr[\tau < \infty] = 1$ and $E[\tau] < \infty$, because from any position X_t there is at least a $p^{a+b} > 0$ chance that the next $a + b$ steps will all be $+1$ and we will reach b . Since we can flip this p^{a+b} -probability coin independently every $a + b$ steps, eventually we reach b if we haven't already reached $-a$. The expected time is bounded by $(a + b)/p^{a+b}$.

We also have that $0 < Z_t \leq \max\left(\left(\frac{q}{p}\right)^a, \left(\frac{q}{p}\right)^b\right)$ for all $t \leq \tau$. This gives us bounded range, so the finite-time/bounded-range case of OST applies.

We thus have $E[Z_\tau] = p_b(q/p)^b + (1 - p_b)(q/p)^{-a} = E[Z_0] = (q/p)^0 = 1$. Solving for p_b gives

$$p_b = \frac{1 - (q/p)^{-a}}{\left(\frac{q}{p}\right)^b - (q/p)^{-a}}.$$

(Note that this only makes sense if $q \neq p$.)

As a test, if $-a = -1$ and $+b = +1$, then we get

$$\begin{aligned} p_b &= \frac{1 - p/q}{q/p - p/q} \\ &= \frac{pq - p^2}{q - p} \\ &= \frac{p(q - p)}{q - p} \\ &= p, \end{aligned}$$

which is what we would expect since we hit $-a$ or $+b$ on the first step.

A more interesting case is if we set $p < 1/2$. Then $q/p > 1$, and

$$\begin{aligned} p_b &= \frac{1 - (q/p)^{-a}}{\left(\frac{q}{p}\right)^b - (q/p)^{-a}} \\ &= (q/p)^{-b} \cdot \frac{1 - (q/p)^{-a}}{1 - (q/p)^{-a-b}} \\ &< (q/p)^{-b}. \end{aligned}$$

Any walk that is biased against us will be an exponentially improbable hill to climb.

2. Now suppose we want to know $E[\tau]$, the average time at which we first hit a or b . We already argued $E[\tau]$ is finite, and it's easy to see that $\{Y_t\}$ has bounded increments, so we can use the finite-expected-time/bounded-increment case of OST to get $E[Y_\tau] = E[Y_0] = 0$, or $E[X_\tau - (p - q)\tau] = 0$. It follows that $E[\tau] = E[X_\tau]/(p - q)$.

But we can compute $E[X_\tau]$, since it is just $p_b b - (1 - p_b)a$. So $E[\tau] = \frac{p_b b - (1 - p_b)a}{p - q}$. If $p > q$, and a and b are both large enough to make p_b very close to 1, this will be approximately $b/(p - q)$, the time to climb to b using our average return of $p - q$ per step.

8.5.2 Wald's equation

Suppose we run a Las Vegas algorithm until it succeeds, and the i -th attempt costs X_i , where all the X_i are independent, satisfy $0 \leq X_i \leq c$ for some c , and have a common mean $E[X_i] = \mu$.

Let N be the number of times we run the algorithm. Since we can tell when we are done, N is a stopping time with respect to some filtration $\{\mathcal{F}_i\}$ to which the X_i are adapted.⁶ Suppose also that $E[N]$ exists. What is $E\left[\sum_{i=1}^N X_i\right]$?

If N were not a stopping time, this might be a very messy problem indeed. But when N is a stopping time, we can apply it to the martingale $Y_t = \sum_{i=1}^t (X_i - \mu)$. This has bounded increments ($0 \leq X_i \leq c$, so $-c \leq X_i - E[X_i] \leq c$), and we've already said $E[N]$ is finite (which implies $\Pr[N < \infty] = 1$), so Theorem 8.3.1 applies. We thus have

$$\begin{aligned} 0 &= E[Y_N] \\ &= E\left[\sum_{i=1}^N (X_i - \mu)\right] \\ &= E\left[\sum_{i=1}^N X_i\right] - E\left[\sum_{i=1}^N \mu\right] \\ &= E\left[\sum_{i=1}^N X_i\right] - E[N]\mu. \end{aligned}$$

Rearranging this gives **Wald's equation**:

$$E\left[\sum_{i=1}^N X_i\right] = E[N]\mu. \quad (8.5.1)$$

⁶A stochastic process $\{X_t\}$ is **adapted** to a filtration $\{\mathcal{F}_t\}$ if each X_t is \mathcal{F}_t -measurable.

This is the same formula as in §3.4.3.1, but we've eliminated the bound on N and allowed for much more dependence between N and the X_i .⁷

8.5.3 Maximal inequalities

Suppose we have a martingale $\{X_i\}$ with $X_i \geq 0$ always, and we want to bound $\max_{i \leq n} X_i$. We can do this using the Optional Stopping Theorem:

Lemma 8.5.1. *Let $\{X_i\}$ be a martingale with $X_i \geq 0$. Then for any fixed n ,*

$$\Pr \left[\max_{i \leq n} X_i \geq \alpha \right] \leq \frac{\mathbb{E}[X_0]}{\alpha}. \quad (8.5.2)$$

Proof. The idea is to pick a stopping time τ such that $\max_{i \leq n} X_i \geq \alpha$ if and only if $X_\tau \geq \alpha$.

Let τ be the first time such that $X_\tau \geq \alpha$ or $\tau \geq n$. Then τ is a stopping time for $\{X_i\}$, since we can determine from X_0, \dots, X_t whether $\tau \leq t$ or not. We also have that $\tau \leq n$ always, which is equivalent to $\tau = \min(\tau, n)$. Finally, $X_\tau \geq \alpha$ means that $\max_{i \leq n} X_i \geq X_\tau \geq \alpha$, and conversely if there is some $t \leq n$ with $X_t = \max_{i \leq n} X_i \geq \alpha$, then τ is the first such t , giving $X_\tau \geq \alpha$.

Lemma 8.4.1 says $\mathbb{E}[X_\tau] = \mathbb{E}[X_0]$. So Markov's inequality gives $\Pr[\max_{i \leq n} X_i \geq \alpha] = \Pr[X_\tau \geq \alpha] \leq \frac{\mathbb{E}[X_\tau]}{\alpha} = \frac{\mathbb{E}[X_0]}{\alpha}$, as claimed. \square

Lemma 8.5.1 is a special case of **Doob's martingale inequality**, which says that for a non-negative *submartingale* $\{X_i\}$,

$$\Pr \left[\max_{i \leq n} X_i \geq \alpha \right] \leq \frac{\mathbb{E}[X_n]}{\alpha}. \quad (8.5.3)$$

The proof is similar, but requires showing first that $\mathbb{E}[X_\tau] \leq \mathbb{E}[X_n]$ when $\tau \leq n$ is a stopping time and $\{X_i\}$ is a submartingale.

8.5.4 Waiting times for patterns

Let's suppose we flip coins until we see some pattern appear: for example, we might flip coins until we see HTHH. What is the expected number of coin-flips until this happens?

⁷In fact, looking closely at the proof reveals that we don't even need the X_i to be independent of each other. We just need that $\mathbb{E}[X_{i+1} | \mathcal{F}_i] = \mu$ for all i to make (Y_t, \mathcal{F}_t) a martingale. But if we don't carry any information from one iteration of our Las Vegas algorithm to the next, we'll get independence anyway. So the big payoff is not having to worry about whether N has some devious dependence on the X_i .

A very clever trick due to Li [Li80] solves this problem exactly using the Optional Stopping Theorem. Suppose our pattern is $x_1x_2 \dots x_k$. We imagine an army of gamblers, one of which shows up before each coin-flip. Each gambler starts by betting \$1 that next coin-flip will be x_1 . If she wins, she bets \$2 that the next coin-flip will be x_2 , continuing to play double-or-nothing until either she loses (and is down \$1) or wins her last bet on x_k (and is up $2^k - 1$). Because each gambler's winnings form a martingale, so does their sum, and so the expected total return of all gamblers up to the stopping time τ at which our pattern first occurs is 0.

We can now use this fact to compute $E[\tau]$. When we stop at time τ , we have one gambler who has won $2^k - 1$. We may also have other gamblers who are still in play. For each i with $x_1 \dots x_i = x_{k-i+1} \dots x_k$, there will be a gambler with net winnings $\sum_{j=1}^i 2^{j-1} = 2^i - 1$. The remaining gamblers will all be at -1 .

Let $\chi_i = 1$ if $x_1 \dots x_i = x_{k-i+1} \dots x_k$, and 0 otherwise. Then the number of losers is given by $\tau - \sum_{i=1}^k \chi_i$ and the total expected payoff is

$$\begin{aligned} E[X_\tau] &= E \left[-\left(\tau - \sum_{i=1}^k \chi_i\right) + \sum_{i=1}^k \chi_i (2^i - 1) \right] \\ &= E \left[-\tau + \sum_{i=1}^k \chi_i (2^i) \right] \\ &= 0. \end{aligned}$$

It follows that $E[\tau] = \sum_{i=1}^k \chi_i 2^i$.

As a quick test, the pattern H has $E[\tau] = 2^1 = 2$. This is consistent with what we know about geometric distributions.

For a longer example, the pattern HTHH only overlaps with its prefix H, so in this case we have $E[\tau] = \sum \chi_i 2^i = 16 + 2 = 18$. But HHHH overlaps with all of its prefixes, giving $E[\tau] = 16 + 8 + 4 + 2 = 30$. At the other extreme, THHH has no overlap at all and gives $E[\tau] = 16$.

In general, for a pattern of length k , we expect a waiting time somewhere between 2^k and $2^{k+1} - 2$ —almost a factor of 2 difference depending on how much overlap we get.

This analysis generalizes in the obvious way to biased coins and larger alphabets. See the paper [Li80] for details.

Chapter 9

Markov chains

Last updated 2019. Some material may be out of date.

A (discrete time) **Markov chain** is a sequence of random variables X_0, X_1, X_2, \dots , which we think of as the position of some particle at increasing times in \mathbb{N} , where the distribution of X_{t+1} depends only on the value of X_t . A typical example of a Markov chain is a random walk on a graph: each X_t is a node in the graph, and a step moves to one of the neighbors of the current node chosen at random, each with equal probability.

Markov chains come up in randomized algorithms both because the execution of any randomized algorithm is, in effect, a Markov chain (the random variables are the states of the algorithm); and because we can often sample from distributions that are difficult to sample from directly by designing a Markov chain that converges to the distribution we want. Algorithms that use this latter technique are known as **Markov chain Monte Carlo** algorithms, and rely on the fundamental fact that a Markov chain that satisfies a few straightforward conditions will always converge in the limit to a **stationary distribution**, no matter what state it starts in.

An example of this technique that predates randomized algorithms is card shuffling: each permutation of the deck is a state, and the shuffling operation sends the deck to a new state each time it is applied. Assuming the shuffling operation is not too deterministic, it is possible to show that enough shuffling will eventually produce a state that is close to being a uniform random permutation. The big algorithmic question for this and similar Markov chains is how quickly this happens: what is the **mixing time** of the Markov chain, a measure of how long we have to run it to get close to its limit distribution (this notion is defined formally in §9.2.3). Many of the techniques in this chapter will be aimed at finding bounds on the mixing

time for particular Markov processes.

If you want to learn more about Markov chains than presented here, they are usually covered in general probability textbooks (for example, in [Fel68] or [GS01]), mentioned in many linear algebra textbooks [Str03], covered in some detail in stochastic processes textbooks [KT75], and covered in exquisite detail in many books dedicated specifically to the subject [KS76, KSK76]. Good sources for mixing times for Markov chains are the textbook of Levin, Peres, and Wilmer [LPW09] and the survey paper by Montenegro and Tetali [MT05]. An early reference on the mixing times for random walks on graphs that helped inspire much subsequent work is the Aldous-Fill manuscript [AF01], which can be found on-line at <http://www.stat.berkeley.edu/~aldous/RWG/book.html>.

9.1 Basic definitions and properties

A **Markov chain** or **Markov process** is a stochastic process¹ where the distribution of X_{t+1} depends only on the value of X_t and not any previous history. Formally, this means that

$$\Pr[X_{t+1} = j \mid X_t = i_t, X_{t-1} = i_{t-1}, \dots, X_0 = i_0] = \Pr[X_{t+1} = j \mid X_t = i_t]. \quad (9.1.1)$$

A stochastic process with this property is called **memoryless**: at any time, you know where you are, and you can figure out where you are going, but you don't know where you were before.

The **state space** of the chain is just the set of all values that each X_t can have. A Markov chain is **finite** or **countable** if it has a finite or countable state space, respectively. We'll mostly be interested in finite Markov chains (since we have to be able to fit them inside our computer), but countable Markov chains will come up in some contexts.²

We'll also assume that our Markov chains are **homogeneous**, which means that $\Pr[X_{t+1} = j \mid X_t = i]$ doesn't depend on t .

¹A **stochastic process** is just a sequence of random variables $\{S_t\}$, where we usually think of t as representing time and the sequence as representing the evolution of some system over time. Here we are considering discrete-time processes, where t will typically be a non-negative integer.

²If the state space is not countable, we run into the same measure-theoretic issues as with continuous random variables, and have to replace (9.1.1) with the more general condition that

$$\mathbb{E}[1_{[X_t \in A]} \mid X_t, X_{t-1}, \dots, X_0] = \mathbb{E}[1_{[X_t \in A]} \mid X_t],$$

provided A is measurable with respect to some appropriate σ -algebra. We don't really want to deal with this, and for the most part we don't have to, so we won't.

For a homogeneous countable Markov chain, we can describe its behavior completely by giving the state space and the one-step **transition probabilities** $p_{ij} = \Pr[X_{t+1} = j \mid X_t = i]$. Given p_{ij} , we can calculate two-step transition probabilities

$$\begin{aligned} p_{ij}^{(2)} &= \Pr[X_{t+2} = j \mid X_t = i] \\ &= \sum_k \Pr[X_{t+2} = j \mid X_{t+1} = k] \Pr[X_{t+1} = k \mid X_t = i] \\ &= \sum_k p_{ik} p_{kj}. \end{aligned}$$

This is identical to the formula for matrix multiplication. For a Markov chain with n states, we can specify the transition probabilities p_{ij} using an $n \times n$ **transition matrix** P with $P_{ij} = p_{ij}$, and the two-step transition probabilities are given by $p_{ij}^{(2)} = P_{ij}^2$. More generally, the t -step transition probabilities are given by $p_{ij}^{(t)} = (P^t)_{ij}$.

Conversely, given any matrix with non-negative entries where the rows sum to 1 ($\sum_j P_{ij} = 1$, or $P\mathbf{1} = \mathbf{1}$, where $\mathbf{1}$ in the second equation stands for the all-ones vector), there is a corresponding Markov chain given by $p_{ij} = P_{ij}$. Such a matrix is called a **stochastic matrix**; and for every stochastic matrix there is a corresponding finite Markov chain and vice versa.

The general formula for $(s+t)$ -step transition probabilities is that $p_{ij}^{(s+t)} = \sum_k p_{ik}^{(s)} p_{kj}^{(t)}$. This is known as the **Chapman-Kolmogorov equation** and is equivalent to the matrix identity $P^{s+t} = P^s P^t$.

A distribution over states of a finite Markov chain at some time t can be given by a row vector x , where $x_i = \Pr[X_t = i]$. To compute the distribution at time $t+1$, we use the law of total probability: $\Pr[X_{t+1} = j] = \sum_i \Pr[X_t = i] \Pr[X_{t+1} = j \mid X_t = i] = \sum_i x_i p_{ij}$. Again we have the formula for matrix multiplication (where we treat x as a $1 \times n$ matrix); so the distribution vector at time $t+1$ is just xP , and at time $t+n$ is xP^n .

We like Markov chains for two reasons:

1. They describe what happens in a randomized algorithm; the state space is just the set of all states of the algorithm, and the Markov property holds because the algorithm can't remember anything that isn't part of its state. So if we want to analyze randomized algorithms, we will need to get good at analyzing Markov chains.
2. They can be used to do sampling over interesting distributions. Under appropriate conditions (see below), the state of a Markov chain converges to a **stationary distribution**. If we build the right Markov

chain, we can control what this stationary distribution looks like, run the chain for a while, and get a sample close to the stationary distribution.

In both cases we want to have a bound on how long it takes the Markov chain to converge, either because it tells us when our algorithm terminates, or because it tells us how long to mix it up before looking at the current state.

9.1.1 Examples

- A fair ± 1 random walk. The state space is \mathbb{Z} , the transition probabilities are $p_{ij} = 1/2$ if $|i - j| = 1$, 0 otherwise. This is an example of a Markov chain that is also a martingale.
- A fair ± 1 random walk on a cycle. As above, but now the state space is \mathbb{Z}/m , the integers mod m . This is a finite Markov chain. It is also in some sense a martingale, although we usually don't define martingales over finite groups.
- Random walks with absorbing and/or reflecting barriers.
- Random walk on a graph $G = (V, E)$. The state space is V , the transition probabilities are $p_{uv} = 1/d(u)$ if $uv \in E$.

One can also have more general transition probabilities, where the probability of traversing a particular edge is a property of the edge and not the degree of its source. In principle we can represent any Markov chain as a random walk on graph in this way: the states become vertices, and the transitions become edges, each labeled with its transition probability. It's conventional in this representation to exclude edges with probability 0 and include self-loops for any transitions $i \rightarrow i$.

If the resulting graph is small enough or has a nice structure, this can be a convenient way to draw a Markov chain.

- The Markov chain given by $X_{t+1} = X_t + 1$ with probability $1/2$, and 0 with probability $1/2$. The state space is \mathbb{N} .
- A finite-state machine running on a random input. The sequence of states acts as a Markov chain, assuming each input symbol is independent of the rest.

- A classic randomized algorithm for 2-SAT, due to Papadimitriou [Pap91]. Each state is a truth-assignment. The transitional probabilities are messy but arise from the following process: pick an unsatisfied clause, pick one of its two variables uniformly at random, and invert it. Then there is an absorbing state at any satisfying assignment. With a bit of work, it can be shown that the Hamming distance between the current assignment and some satisfying assignment follows a random walk biased toward 0, giving a satisfying assignment after $O(n^2)$ steps on average.
- A similar process works for 2-colorability, 3-SAT, 3-colorability, etc., although for NP-hard problems, it may take a while to reach an absorbing state. The constructive Lovász Local Lemma proof from §11.3.5 also follows this pattern.

9.2 Convergence of Markov chains

We want to use Markov chains for sampling. Typically this means that we have some subset S of the state space, and we want to know what proportion of the states are in S . If we can't sample states from the state space a priori, we may be able to get a good approximation by running the Markov chain for a while and hoping that it converges to something predictable.

To show this, we will proceed through several steps:

1. We will define a class of distributions, known as **stationary distributions**, that we hope to converge to (§9.2.1)
2. We will define a distance between distributions, the **total variation distance** (§9.2.2).
3. We will define the **mixing time** of a Markov chain as the minimum time for the distribution of the position of a particle to get within a given total variation distance of the stationary distribution (§9.2.3).
4. We will describe a technique called **coupling** that can be used to bound total variation distance (§9.2.4), in terms of the probability that two dependent variables X and Y with the distributions we are looking at are or are not equal to each other.
5. We will define **reducible** and **periodic** Markov chains, which have structural properties that prevent convergence to a unique stationary distribution (§9.2.5).

6. We will use a coupling between two copies of a Markov chain to show that *any* Markov chain that does not have these properties does converge in total variation distance to a unique stationary distribution (§9.2.6).
7. Finally, we will show that if we can construct a coupling between two copies of a particular chain that causes both copies to reach the same state quickly on average, then we can use this expected **coupling time** to bound the mixing time of the chain that we defined previously. This will give us a practical tool for showing that many Markov chains not only converge eventually but converge at a predictable rate, so we can tell when it is safe to stop running the chain and take a sample (§9.4).

Much of this section follows the approach of Chapter 4 of Levin *et al.* [LPW09].

9.2.1 Stationary distributions

For a finite Markov chain, there is a transition matrix P for which each row sums to 1. We can write this fact compactly as $P\mathbf{1} = \mathbf{1}$, where $\mathbf{1}$ is the all-ones column vector. This means that $\mathbf{1}$ is a right eigenvector of P with eigenvalue 1.³ Because the left eigenvalues of a matrix are equal to the right eigenvalues, this means that there will be at least one left eigenvector π such that $\pi P = \pi$, and in fact it is possible to show that there is at least one such π that represents a probability distribution in that each $\pi_i \geq 0$ and $\sum \pi_i = 1$. Such a distribution is called a **stationary distribution**⁴ of the Markov chain, and if π is unique, the probability π_i is called the **stationary probability** for i .

Every finite Markov chain has at least one stationary distribution, but it may not be unique. For example, if the transition matrix is the identity matrix (meaning that the particle never moves), then all distributions are stationary.

If a Markov chain does have a unique stationary distribution, we can calculate it from the transition matrix, by observing that the equations

$$\pi P = \pi$$

³Given a square matrix A , a vector x is a right eigenvector of A with eigenvalue λ if $Ax = \lambda x$. Similarly, a vector y is a left eigenvector of A with eigenvalue λ if $yA = \lambda y$.

⁴Spelling is important here. A *stationery distribution* would mean handing out office supplies.

and

$$\pi \mathbf{1} = \mathbf{1}$$

together give $n + 1$ equations in n unknowns, which we can solve for π . (We need an extra equation because the stochastic property of P means that it has rank at most $n - 1$.)

Often this will be impractical, especially if our state space is large enough or messy enough that we can't write down the entire matrix. In these cases we may be able to take advantage of a special property of some Markov chains, called **reversibility**. We'll discuss this in §9.3. For the moment we will content ourselves with showing that we do in fact converge to some unique stationary distribution if our Markov chain has the right properties.

9.2.2 Total variation distance

Definition 9.2.1. *Let X and Y be random variables defined on the same probability space. Then the **total variation distance** between X and Y , written $d_{TV}(X, Y)$ is given by*

$$d_{TV}(X, Y) = \sup_A (\Pr[X \in A] - \Pr[Y \in A]), \quad (9.2.1)$$

where the supremum is taken over all sets A for which $\Pr[X \in A]$ and $\Pr[Y \in A]$ are both defined.⁵

An equivalent definition is

$$d_{TV}(X, Y) = \sup_A |\Pr[X \in A] - \Pr[Y \in A]|.$$

The reason this is equivalent is that if $\Pr[X \in A] - \Pr[Y \in A]$ is negative, we can replace A by its complement.

Less formally, given any test set A , X and Y show up in A with probabilities that differ by at most $d_{TV}(X, Y)$. This is usually what we want for sampling, since this says that if we are testing some property (represented by A) of the states we are sampling, the answer ($\Pr[X \in A]$) that we get for how likely this property is to occur is close to the correct answer ($\Pr[Y \in A]$).

⁵For discrete random variables, this just means all A , since we can write $\Pr[X \in A]$ as $\sum_{x \in A} \Pr[X = x]$; we can also replace \sup with \max for this case. For continuous random variables, we want that $X^{-1}(A)$ and $Y^{-1}(A)$ are both measurable. If our X and Y range over the states of a countable Markov chain, we will be working with discrete random variables, so we can just consider all A .

Total variation distance is a property of distributions, and is not affected by dependence between X and Y . For finite Markov chains, we can define the total variation distance between two distributions x and y as

$$d_{TV}(x, y) = \max_A \sum_{i \in A} (x_i - y_i) = \max_A \left| \sum_{i \in A} (x_i - y_i) \right|.$$

A useful fact is that $d_{TV}(x, y)$ is directly connected to the ℓ_1 distance between x and y . If we let $B = \{i \mid x_i \geq y_i\}$, then

$$\begin{aligned} d_{TV}(x, y) &= \max_A \sum_{i \in A} (x_i - y_i) \\ &\leq \sum_{i \in B} (x_i - y_i), \end{aligned}$$

because if A leaves out an element of B , or includes an element of \bar{B} , this can only reduce the sum. But if we consider \bar{B} instead, we get

$$\begin{aligned} d_{TV}(y, x) &= \max_A \sum_{i \in A} (y_i - x_i) \\ &\leq \sum_{i \in \bar{B}} (y_i - x_i). \end{aligned}$$

Now observe that

$$\begin{aligned} \|x - y\|_1 &= \sum_i |x_i - y_i| \\ &= \sum_{i \in B} (x_i - y_i) + \sum_{i \in \bar{B}} (y_i - x_i) \\ &= d_{TV}(x, y) + d_{TV}(y, x) \\ &= 2d_{TV}(x, y). \end{aligned}$$

So $d_{TV}(x, y) = \frac{1}{2} \|x - y\|_1$.

9.2.2.1 Total variation distance and expectation

Sometimes it's useful to translate a bound on total variation to a bound on the error when getting the expectation of a random variable. The following lemma may be handy:

Lemma 9.2.2. *Let x and y be two distributions of some discrete random variable Z . Let $E_x(Z)$ and $E_y(Z)$ be the expectations of Z with respect to each of these distributions. Suppose that $|Z| \leq M$ always. Then*

$$|E_x(Z) - E_y(Z)| \leq 2Md_{TV}(x, y). \quad (9.2.2)$$

Proof. Compute

$$\begin{aligned}
 |E_x(Z) - E_y(Z)| &= \left| \sum_z z \left(\Pr_x(Z = z) - \Pr_y(Z = z) \right) \right| \\
 &\leq \sum_z |z| \cdot \left| \Pr_x(Z = z) - \Pr_y(Z = z) \right| \\
 &\leq M \sum_z \left| \Pr_x(Z = z) - \Pr_y(Z = z) \right| \\
 &\leq M \|x - y\|_1 \\
 &= 2M d_{TV}(x, y).
 \end{aligned}$$

□

9.2.3 Mixing time

We are going to show that well-behaved finite Markov chains eventually converge to some stationary distribution π in total variation distance. This means that for any $\epsilon > 0$, there is a **mixing time** $t_{\text{mix}}(\epsilon)$ such that for any initial distribution x and any $t \geq t_{\text{mix}}(\epsilon)$,

$$d_{TV}(xP^t, \pi) \leq \epsilon.$$

It is common to standardize ϵ as $1/4$: if we write just t_{mix} , this means $t_{\text{mix}}(1/4)$. The choice of $1/4$ is somewhat arbitrary, but has some nice technical properties that we will see below.

9.2.4 Coupling of Markov chains

A **coupling** of two random variables X and Y is a joint distribution on $\langle X, Y \rangle$ that gives the correct marginal distribution for each of X and Y while creating a dependence between them with some desirable property (for example, minimizing total variation distance or maximize $\Pr[X = Y]$).

We will use couplings between Markov chains to prove convergence. Here we take two copies of the chain, one of which starts in an arbitrary distribution, and one of which starts in the stationary distribution, and show that we can force them to converge to each other by carefully correlating their transitions. Since the second chain is always in the stationary distribution, this will show that the first chain converges to the stationary distribution as well.

The tool that makes this work is the **Coupling Lemma**:⁶

Lemma 9.2.3. *For any discrete random variables X and Y ,*

$$d_{TV}(X, Y) \leq \Pr[X \neq Y].$$

Proof. Let A be any set for which $\Pr[X \in A]$ and $\Pr[Y \in A]$ are defined. Then

$$\begin{aligned} \Pr[X \in A] &= \Pr[X \in A \wedge Y \in A] + \Pr[X \in A \wedge Y \notin A], \\ \Pr[Y \in A] &= \Pr[X \in A \wedge Y \in A] + \Pr[X \notin A \wedge Y \in A], \end{aligned}$$

and thus

$$\begin{aligned} \Pr[X \in A] - \Pr[Y \in A] &= \Pr[X \in A \wedge Y \notin A] - \Pr[X \notin A \wedge Y \in A] \\ &\leq \Pr[X \in A \wedge Y \notin A] \\ &\leq \Pr[X \neq Y]. \end{aligned}$$

Since this holds for any particular set A , it also holds when we take the maximum over all A to get $d_{TV}(X, Y)$. \square

For Markov chains, our goal will be to find a useful coupling between a sequence of random variables X_0, X_1, X_2, \dots corresponding to the Markov chain starting in an arbitrary distribution with a second sequence Y_0, Y_1, Y_2, \dots corresponding to the same chain starting in a stationary distribution. What will make a coupling useful is if $\Pr[X_t \neq Y_t]$ is small for reasonably large t : since Y_t has the stationary distribution, this will show that $d_{TV}(xP^t, \pi)$ is also small.

Our first use of this technique will be to show, using a rather generic coupling, that Markov chains with certain nice properties converge to their

⁶It turns out that the bound in the Coupling Lemma is tight in the following sense: for any given distributions on X and Y , there exists a joint distribution giving these distributions such that $d_{TV}(X, Y)$ is exactly equal to $\Pr[X \neq Y]$ when X and Y are sampled from the joint distribution. For discrete distributions, the easiest way to construct the joint distribution is first to let $Y = X = i$ for each i with probability $\min(\Pr[X = i], \Pr[Y = i])$, and then distribute the remaining probability for X over all the cases where $\Pr[X = i] > \Pr[Y = i]$ and similarly for Y over all the cases where $\Pr[Y = i] > \Pr[X = i]$. Looking at the unmatched values for X gives $\Pr[X \neq Y] \leq \sum_{\{x \mid \Pr[X=x] > \Pr[Y=x]\}} (\Pr[X = x] - \Pr[Y = x]) \leq d_{TV}(X, Y)$. So in this case $\Pr[X \neq Y] = d_{TV}(X, Y)$.

Unfortunately, the fact that there always exists a perfect coupling in this sense does not mean that we can express it in any convenient way, or that even if we could, it would arise from the kind of causal, step-by-step construction that we will use for couplings between Markov processes.

stationary distribution in the limit. Later we will construct specialized couplings for particular Markov chains to show that they converge quickly. But first we will consider what properties a Markov chain must have to converge at all.

9.2.5 Irreducible and aperiodic chains

Not all chains are guaranteed to converge to their stationary distribution. If some states are not reachable from other states, it may be that starting in one part of the chain will keep us from ever reaching another part of the chain. Even if the chain is not disconnected in this way, we might still not converge if the distribution oscillates back and forth due to some periodicity in the structure of the chain. But if these conditions do not occur, then we will be able to show convergence.

Let p_{ij}^t be $\Pr[X_t = j \mid X_0 = i]$.

A Markov chain is **irreducible** if, for all states i and j , there exists some t such that $p_{ij}^t \neq 0$. This says that we can reach any state from any other state if we wait long enough. If we think of a directed graph of the Markov chain where the states are vertices and each edge represents a transition that occurs with nonzero probability, the Markov chain is irreducible if its graph is strongly connected.

The **period** of a state i of a Markov chain is $\gcd(\{t > 0 \mid p_{ii}^t \neq 0\})$. If the period of i is m , then starting from i we can only return to i at times that are multiples of m . If $m = 1$, state i is said to be **aperiodic**. A Markov chain as a whole is aperiodic if all of its states are aperiodic. In graph-theoretic terms, this means that the graph of the chain is not k -partite for any $k > 1$. Reversible chains are also an interesting special case: if a chain is reversible, it can't have a period greater than 2, since we can always step off a node and step back.

If our Markov chain is not aperiodic, we can make it aperiodic by flipping a coin at each step to decide whether to move or not. This gives a **lazy Markov chain** whose transition probabilities are given by $\frac{1}{2}p_{ij}$ when $i \neq j$ and $\frac{1}{2} + \frac{1}{2}p_{ij}$ when $i = j$. This doesn't affect the stationary distribution: if we replace our transition matrix P with a new transition matrix $\frac{P+I}{2}$, and $\pi P = \pi$, then $\pi\left(\frac{P+I}{2}\right) = \frac{1}{2}\pi P + \frac{1}{2}\pi I = \frac{1}{2}\pi + \frac{1}{2}\pi = \pi$.

Unfortunately there is no quick fix for reducible Markov chains. But since we will often be designing the Markov chains we will be working with, we can just take care to make sure they are not reducible.

We will later need the following lemma about aperiodic Markov chains, which is related to the **Frobenius problem** of finding the minimum value

that cannot be constructed using coins of given denominations:

Lemma 9.2.4. *Let i be an aperiodic state of some Markov chain. Then there is a time t_0 such that $p_{ii}^t \neq 0$ for all $t \geq t_0$.*

Proof. Let $S = \{t \mid p_{ii}(t) \neq 0\}$. Since $\gcd(S) = 1$, there is a finite subset S' of S such that $\gcd S' = 1$. Write the elements of S' as m_1, m_2, \dots, m_k and let $M = \prod_{j=1}^k m_j$. From the extended Euclidean algorithm, there exist integer coefficients a_j with $|a_j| \leq M/m_j$ such that $\sum_{j=1}^k a_j m_j = 1$. We would like to use each a_j as the number of times to go around the length m_j loop from i to i . Unfortunately many of these a_j will be negative.

To solve this problem, we replace a_j with $b_j = a_j + M/m_j$. This makes all the coefficients non-negative, and gives $\sum_{j=1}^k b_j m_j = kM + 1$. This implies that there is a sequence of loops that gets us from i back to i in $kM + 1$ steps, or in other words that $p_{ii}^{kM+1} \neq 0$. By repeating this sequence ℓ times, we can similarly show that $p_{ii}^{\ell kM + \ell} \neq 0$ for any ℓ .

We can also pad any of these sequences out by as many copies of M as we like. In particular, given $\ell kM + \ell$, where $\ell \in \{0, \dots, M-1\}$, we can add $(kM - \ell)M$ to it to get $(kM)^2 + \ell$. This means that we can express any $t \in \{(kM)^2, \dots, (kM)^2 + M - 1\}$ as a sum of elements of S , or equivalently that $p_{ii}^t \neq 0$ for any such t . But for larger t , we can just add in more copies of M . So in fact $p_{ii}^t \neq 0$ for all $t \geq t_0 = (kM)^2$. \square

9.2.6 Convergence of finite irreducible aperiodic Markov chains

We can now show:

Theorem 9.2.5. *Any finite irreducible aperiodic Markov chain converges to a unique stationary distribution in the limit.*

Proof. Consider two copies of the chain $\{X_t\}$ and $\{Y_t\}$, where X_0 starts in some arbitrary distribution x and Y_0 starts in a stationary distribution π . Define a coupling between $\{X_t\}$ and $\{Y_t\}$ by the rule: (a) if $X_t \neq Y_t$, then $\Pr[X_{t+1} = j \wedge Y_{t+1} = j' \mid X_t = i \wedge Y_t = i'] = p_{ij} p_{i'j'}$; and (b) if $X_t = Y_t$, then $\Pr[X_{t+1} = Y_{t+1} = j \mid X_t = Y_t = i] = p_{ij}$. Intuitively, we let both chains run independently until they collide, after which we run them together. Since each chain individually moves from state i to state j with probability p_{ij} in either case, we have that X_t evolves normally and Y_t remains in the stationary distribution.

Now let us show that $d_{TV}(xP^t, \pi) \leq \Pr[X_t \neq Y_t]$ goes to zero in the limit. Pick some state i . Let r be the maximum over all states j of the **first**

passage time f_{ji} where f_{ji} is the minimum time t such that $p_{ji}^t \neq 0$. Let s be a time such that $p_{ii}^t \neq 0$ for all $t \geq s$ (the existence of such an s is given by Lemma 9.2.4).

Suppose that at time $\ell(r+s)$, where $\ell \in \mathbb{N}$, $X_{\ell(r+s)} = j \neq j' = Y_{\ell(r+s)}$. Then there are times $\ell(r+s) + u$ and $\ell(r+s) + u'$, where $u, u' \leq r$, such that X reaches i at time $\ell(r+s) + u$ and Y reaches i at time $\ell(r+s) + u'$ with nonzero probability. Since $(r+s-u) \leq s$, then having reached i at these times, X and Y both return to i at time $\ell(r+s) + (r+s) = (\ell+1)(r+s)$ with nonzero probability. Let $\epsilon > 0$ be the product of these nonzero probabilities; then $\Pr[X_{(\ell+1)(r+s)} \neq Y_{(\ell+1)(r+s)}] \leq (1-\epsilon) \Pr[X_{\ell(r+s)} = Y_{\ell(r+s)}]$, and in general we have $\Pr[X_t \neq Y_t] \leq (1-\epsilon)^{\lfloor t/(r+s) \rfloor}$, which goes to zero in the limit. This implies that $d_{TV}(xP^t, \pi)$ also goes to zero in the limit (using the Coupling Lemma), and since any initial distribution (including a stationary distribution) converges to π , π is the unique stationary distribution as claimed. \square

This argument requires that the chain be finite, because otherwise we cannot take the maximum over all the first passage times. For infinite Markov chains, it is not always enough to be irreducible and aperiodic to converge to a stationary distribution (or even to have a stationary distribution at all). However, with some additional conditions a similar result can be shown: see for example [GS01, §6.4].

9.3 Reversible chains

A Markov chain with transition probabilities p_{ij} is **reversible** if there is a distribution π such that, for all i and j ,

$$\pi_i p_{ij} = \pi_j p_{ji}. \quad (9.3.1)$$

These are called the **detailed balance equations**—they say that in the stationary distribution, the probability of seeing a transition from i to j is equal to the probability of seeing a transition from j to i). If this is the case, then $\sum_i \pi_i p_{ij} = \sum_i \pi_j p_{ji} = \pi_j$, which means that π is stationary.

It's worth noting that this works for countable chains even if they are not finite, because the sums always converge since each term is non-negative and $\sum_i \pi_i p_{ij}$ is dominated by $\sum_i \pi_i = 1$. However, it may not be the case for any particular p that there exists a corresponding stationary distribution π . If this happens, the chain is not reversible.

9.3.1 Stationary distributions

The detailed balance equations often give a very quick way to compute the stationary distribution, since if we know π_i , and $p_{ij} \neq 0$, then $\pi_j = \pi_i p_{ij} / p_{ji}$. If the transition probabilities are reasonably well-behaved (for example, if $p_{ij} = p_{ji}$ for all i, j), we may even be able to characterize the stationary distribution up to a constant multiple even if we have no way to efficiently enumerate all the states of the process.

What (9.3.1) says is that if we start in the stationary distribution and observe either a forward transition $\langle X_0, X_1 \rangle$ or a backward transition $\langle X_1, X_0 \rangle$, we can't tell which is which; $\Pr[X_0 = i \wedge X_1 = j] = \Pr[X_1 = i \wedge X_0 = j]$. This extends to longer sequences. The probability that $X_0 = i$, $X_1 = j$, and $X_2 = k$ is given by $\pi_i p_{ij} p_{jk} = p_{ji} \pi_j p_{jk} = p_{ji} p_{kj} \pi_k$, which is the probability that $X_0 = k$, $X_1 = j$, and $X_2 = i$. (A similar argument works for finite sequences of any length.) So a reversible Markov chain is one with no arrow of time in the stationary distribution.

A typical reversible Markov chain is a random walk on a graph, where a step starting from a vertex u goes to one of its neighbors v , which each neighbor chosen with probability $\frac{1}{d(u)}$. This has a stationary distribution

$$\begin{aligned} \pi_u &= \frac{d(u)}{\sum_u d(u)} \\ &= \frac{d(u)}{2|E|}, \end{aligned}$$

which satisfies

$$\pi_u p_{uv} = \frac{d(u)}{2|E|} \cdot \frac{1}{d(u)} = \frac{d(v)}{2|E|} \cdot \frac{1}{d(v)} = \pi_v p_{vu}.$$

If we don't know π in advance, we can often guess it by observing that

$$\pi_i p_{ij} = \pi_j p_{ji}$$

implies

$$\pi_j = \pi_i \frac{p_{ij}}{p_{ji}}, \tag{9.3.2}$$

provided $p_{ij} \neq 0$. This gives us the ability to calculate π_k starting from any initial state i as long as there is some chain of transitions $i = i_0 \rightarrow i_1 \rightarrow i_2 \rightarrow \dots \rightarrow i_\ell = k$ where each step $i_m \rightarrow i_{m+1}$ has $p_{i_m, i_{m+1}} \neq 0$. For a random walk on a graph, this implies that π is unique as long as the graph is connected.

This of course only works for reversible chains; if we try to do this with a non-reversible chain, we are likely to get a contradiction.

For example, if we consider a biased random walk on the n -cycle, which moves from i to $(i + 1) \bmod n$ with probability p and in the other direction with probability $q = 1 - p$, then applying (9.3.2) repeatedly would give $\pi_i = \pi_0 \left(\frac{p}{q}\right)^i$. This is not a problem when $p = q = 1/2$, since we get $\pi_i = \pi_0$ for all i and can deduce that $\pi_i = 1/n$ is the unique stationary distribution. But if we try it for $p = 2/3$, then we get $\pi_i = \pi_0 2^i$, which is fine up until we hit $\pi_0 = \pi_n = \pi_0 2^n$. So for $p \neq q$, this process is not reversible, which is not surprising if we realize that the $n = 60, p = 1$ case describes precisely the movement of the second hand on a clock.⁷

Reversible chains have a special role in Markov chain theory, because some techniques for proving convergence only apply to reversible chains (see §9.5.4). They are also handy for sampling from large, irregular state spaces, because by tuning the transition probabilities locally we can often adjust the relative likelihoods of states in the stationary distribution to be closer to what we want (see §9.3.4).

9.3.2 Examples

Random walk on a weighted graph Here each edge has a weight w_{uv} where $0 < w_{uv} = w_{vu} < \infty$, with self-loops permitted. A step of the random walk goes from u to v with probability $w_{uv} / \sum_{v'} w_{uv'}$. It is easy to show that this random walk has stationary distribution $\pi_u = \sum_v w_{uv} / \sum_u \sum_v w_{uv}$, generalizing the previous case, and that the resulting Markov chain satisfies the detailed balance equations.

Random walk with uniform stationary distribution Now let Δ be the maximum degree of the graph, and traverse each edge with probability $1/\Delta$, staying put on each vertex u with probability $1 - d(u)/\Delta$. The stationary distribution is uniform, since for each pair of vertices u and v we have $p_{uv} = p_{vu} = 1/\Delta$ if u and v are adjacent and 0 otherwise. This is a special case of the Metropolis-Hastings algorithm (see §9.3.4).

9.3.3 Time-reversed chains

Another way to get a reversible chain is to take an arbitrary chain with a stationary distribution and rearrange it so that it can run both forwards

⁷It happens to be the case that $\pi_i = 1/n$ is a stationary distribution for any value of p , we just can't prove this using (9.3.1).

and backwards in time. This is not necessarily useful, but it illustrates the connection between reversible and irreversible chains.

Given a finite Markov chain with transition matrix P and stationary distribution π , define the corresponding **time-reversed chain** with matrix P^* where $\pi_i p_{ij} = \pi_j p_{ji}^*$.

To make sure that this actually works, we need to verify that:

1. The matrix P^* is stochastic:

$$\begin{aligned} \sum_j p_{ij}^* &= \sum_j p_{ji} \pi_j / \pi_i \\ &= \pi_i / \pi_i \\ &= 1. \end{aligned}$$

2. The reversed chain has the same stationary distribution as the original chain:

$$\begin{aligned} \sum_j \pi_j p_{ji}^* &= \sum_j \pi_i p_{ij} \\ &= \pi_i. \end{aligned}$$

3. And that in general P^* 's paths starting from the stationary distribution are a reverse of P 's paths starting from the same distribution. For length-1 paths, this is just $\pi_j p_{ji}^* = \pi_i p_{ij}$. For longer paths, this follows from an argument similar to that for reversible chains.

This gives an alternate definition of a reversible chain as a chain for which $P = P^*$.

We can also use time-reversal to generate reversible chains from arbitrary chains. The chain with transition matrix $(P + P^*)/2$ (corresponding to moving 1 step forward or back with equal probability at each step) is always a reversible chain.

Examples:

- Given a biased random walk on a cycle that moves right with probability p and left with probability q , its time-reversal is the walk that moves left with probability p and right with probability q . (Here the fact that the stationary distribution is uniform makes things simple.) The average of this chain with its time-reversal is an unbiased random walk.

- Given the random walk defined by $X_{t+1} = X_t + 1$ with probability $1/2$ and 0 with probability $1/2$, we have $\pi_i = 2^{-i-1}$. This is not reversible (there is a transition from 1 to 2 but none from 2 to 1), but we can reverse it by setting $p_{ij}^* = 1$ for $i = j + 1$ and $p_{0i}^* = 2^{-i-1}$. (Check: $\pi_i p_{ii+1} = 2^{-i-1}(1/2) = \pi_{i+1} p_{i+1i}^* = 2^{-i-2}(1)$; $\pi_i p_{i0} = 2^{-i-1}(1/2) = \pi_0 p_{0i}^* = (1/2)2^{-i-1}$.)

These examples work because the original chains are simple and have clean stationary distributions. Reversed versions of chains with messier stationary distributions are usually messier. In practice, building reversible chains using time-reversal is often painful precisely because we don't have a good characterization of the stationary distribution of the original non-reversible chain. So we will often design our chains to be reversible from the start rather than relying on after-the-fact flipping.

9.3.4 Adjusting stationary distributions with the Metropolis-Hastings algorithm

Sometimes we have a reversible Markov chain, but it doesn't have the stationary distribution we want. The **Metropolis-Hastings algorithm** [MRR⁺53, Has70] (sometimes just called **Metropolis**) lets us start with a reversible Markov chain P with a known stationary distribution π and convert it to a chain Q on the same states with a different stationary distribution μ , where $\mu_i = f(i) / \sum_j f(j)$ is proportional to some function $f \geq 0$ on states that we can compute easily.

A typical application is that we want to sample according to $\Pr[i | A]$, but A is highly improbable (so we can't just use **rejection sampling**, where we sample random points from the original distribution until we find one for which A holds), and $\Pr[i | A]$ is easy to compute for any fixed i but tricky to compute for arbitrary events (so we can't use divide-and-conquer). If we let $f(i) \propto \Pr[i | A]$, then Metropolis-Hastings will do exactly what we want, assuming it converges in a reasonable amount of time.

Let q be the transition probability for Q . Define, for $i \neq j$,

$$\begin{aligned} q_{ij} &= p_{ij} \min\left(1, \frac{\pi_i f(j)}{\pi_j f(i)}\right) \\ &= p_{ij} \min\left(1, \frac{\pi_i \mu_j}{\pi_j \mu_i}\right) \end{aligned}$$

and let q_{ii} be whatever probability is left over. Now consider two states i

and j , and suppose that $\pi_i f(j) \geq \pi_j f(i)$. Then

$$q_{ij} = p_{ij}$$

which gives

$$\mu_i q_{ij} = \mu_i p_{ij},$$

while

$$\begin{aligned} \mu_j q_{ji} &= \mu_j p_{ji} (\pi_j \mu_i / \pi_i \mu_j) \\ &= p_{ji} (\pi_j \mu_i / \pi_i) \\ &= \mu_i (p_{ji} \pi_j) / \pi_i \\ &= \mu_i (p_{ij} \pi_i) / \pi_i \\ &= \mu_i p_{ij} \end{aligned}$$

(note the use of reversibility of P in the second-to-last step). So we have $\mu_j q_{ji} = \mu_i p_{ij} = \mu_i q_{ij}$ and Q is a reversible Markov chain with stationary distribution μ .

We can simplify this when our underlying chain P has a uniform stationary distribution (for example, when it's the random walk on a graph with maximum degree Δ , where we traverse each edge with probability $1/\Delta$). Then we have $\pi_i = \pi_j$ for all i, j , so the new transition probabilities q_{ij} are just $\frac{1}{\Delta} \min(1, f(j)/f(i))$. Most of our examples of reversible chains will be instances of this case (see also [MU05, §10.4.1]).

9.4 The coupling method

In order to use the stationary distribution of a Markov chain to do sampling, we need to have a bound on the rate of convergence to tell us when it is safe to take a sample. There are two standard techniques for doing this: coupling, where we show that a copy of the process starting in an arbitrary state can be made to converge to a copy starting in the stationary distribution; and spectral methods, where we bound the rate of convergence by looking at the second-largest eigenvalue of the transition matrix. We'll start with coupling because it requires less development.

(See also [Gur00] for a survey of the relationship between the various methods.)

Note: these notes will be somewhat sketchy. If you want to read more about coupling, a good place to start might be Chapter 11 of [MU05]; Chapter

4-3 (<http://www.stat.berkeley.edu/~aldous/RWG/Chap4-3.pdf>) of the unpublished but nonetheless famous **Aldous-Fill manuscript** (<http://www.stat.berkeley.edu/~aldous/RWG/book.html>, [AF01]), which is a good place to learn about Markov chains and Markov chain Monte Carlo methods in general; or even an entire book [Lin92]. We'll mostly be using examples from the Aldous-Fill text.

9.4.1 Random walk on a cycle

Let's suppose we do a random walk on \mathbb{Z}_m , where to avoid periodicity at each step we stay put with probability $1/2$, move counterclockwise with probability $1/4$, and move clockwise with probability $1/4$: in other words, we are doing a lazy unbiased random walk. What's a good choice for a coupling to show this process converges quickly?

Specifically, we need to create a joint process (X_t, Y_t) , where each of the marginal processes X and Y looks like a lazy random walk on \mathbb{Z}_m , X_0 has whatever distribution our real process starts with, and Y_0 has the stationary distribution. Our goal is to structure the combined process so that $X_t = Y_t$ as soon as possible.

Let $Z_t = X_t - Y_t \pmod{m}$. If $Z_t = 0$, then X_t and Y_t have collided and we will move both together. If $Z_t \neq 0$, then flip a coin to decide whether to move X_t or Y_t ; whichever one moves then moves up or down with equal probability. It's not hard to see that this gives a probability of exactly $1/2$ that $X_{t+1} = X_t$, $1/4$ that $X_{t+1} = X_t + 1$, and $1/4$ that $X_{t+1} = X_t - 1$, and similarly for Y_t . So the transition functions for X and Y individually are the same as for the original process.

Whichever way the first flip goes, we get $Z_{t+1} = Z_t \pm 1$ with equal probability. So Z acts as an unbiased random walk on \mathbb{Z}_m with an absorbing barriers at 0 ; this is equivalent to a random walk on $0 \dots m$ with absorbing barriers at both endpoints. The expected time for this random walk to reach a barrier starting from an arbitrary initial state is at most $m^2/4$, so if τ is the first time at which $X_\tau = Y_\tau$, we have $E[\tau] \leq m^2/4$.⁸

Using Markov's inequality, after $t = 2(m^2/4) = m^2/2$ steps we have $\Pr[X_t \neq Y_t] = \Pr[\tau > m^2/2] \leq \frac{E[\tau]}{m^2/2} \leq 1/2$. We can also iterate the whole argument, starting over in whatever state we are in at time t if we don't converge. This gives at most a $1/2$ chance of not converging for each interval of $m^2/2$ steps. So after $\alpha m^2/2$ steps we will have $\Pr[X_t \neq Y_t] \leq 2^{-\alpha}$. This

⁸If we know that Y_0 is uniform, then Z_0 is also uniform, and we can use this fact to get a slightly smaller bound on $E[\tau]$, around $m^2/6$. But this will cause problems if we want to re-run the coupling starting from a state where X_t and Y_t have not yet converged.

gives $t_{\text{mix}}(\epsilon) \leq \frac{1}{2}m^2 \lceil \lg(1/\epsilon) \rceil$, where as before $t_{\text{mix}}(\epsilon)$ is the time needed to make $d_{TV}(X_t, \pi) \leq \epsilon$ (see §9.2.3).

The choice of 2 for the constant in Markov's inequality could be improved. The following lemma gives an optimized version of this argument:

Lemma 9.4.1. *Let the expected **coupling time**, at which two coupled processes $\{X_t\}$ and $\{Y_t\}$ starting from an arbitrary state are first equal, be T . Then $d_{TV}(X_{T_\epsilon}, Y_{T_\epsilon}) \leq \epsilon$ for $T_\epsilon \geq Te \lceil \ln(1/\epsilon) \rceil$.*

Proof. Essentially the same argument as above, but replacing 2 with a constant c to be determined. Suppose we restart the process every cT steps. Then at time t we have a total variation bounded by $c^{-\lfloor t/cT \rfloor}$. The expression $c^{-t/cT}$ is minimized by minimizing $c^{-1/c}$ or equivalently $-\ln c/c$, which occurs at $c = e$. This gives $t_{\text{mix}}(\epsilon) \leq Te \lceil \ln(1/\epsilon) \rceil$. \square

It's worth noting that the random walk example was very carefully rigged to make the coupling argument clean. A similar argument still works (perhaps with a change in the bound) for other irreducible aperiodic walks on the ring, but the details are messier.

9.4.2 Random walk on a hypercube

Start with a bit-vector of length n . At each step, choose an index uniformly at random, and set the value of the bit-vector at that index to 0 or 1 with equal probability. How long until we get a nearly-uniform distribution over all 2^n possible bit-vectors?

Here we apply the same transformation to both the X and Y vectors. It's easy to see that the two vectors will be equal once every index has been selected once. The waiting time for this to occur is just the waiting time nH_n for the coupon collector problem. We can either use this expected time directly to show that the process mixes in time $O(n \log n \log(1/\epsilon))$ as above, or apply a sharp concentration bound to the coupon collector process. It is known that (see [MU05, §5.4.1] or [MR95, §3.6.3]), $\lim_{n \rightarrow \infty} \Pr [T \geq n(\ln n + c)] = 1 - \exp(-\exp(-c))$, so in the limit $n \ln n + n \ln \ln(1/(1 - \epsilon)) = n \ln n + O(n \log(1/\epsilon))$ would seem to be enough. But this is a little tricky: we don't know from this bound alone how fast the probability converges as a function of n , so to do this right we need to look into the bound in more detail.

Fortunately, we are looking at large enough values of c that we can get a

bound that is just as good using a simple argument. We have

$$\begin{aligned} \Pr[\text{there is an empty bin after } t \text{ insertions}] &\leq n \left(1 - \frac{1}{n}\right)^t \\ &\leq ne^{-t/n}, \end{aligned}$$

and setting $t = n \ln n + cn$ gives a bound of e^{-c} . We can then set $c = \ln(1/\epsilon)$ to get a $\Pr[X_t \neq Y_t] \leq \epsilon$ at time $n \ln n + n \ln(1/\epsilon)$.

We can improve the bound slightly by observing that, on average, half the bits in X_0 and Y_0 are already equal; doing this right involves summing over a lot of cases, so we won't do it.

This is an example of a Markov chain with the **rapid mixing** property: the mixing time is polylogarithmic in the number of states (2^n in this case) and $1/\epsilon$. For comparison, the random walk on the ring is not rapid mixing, because the coupling time is polynomial in $n = m$ rather than $\log n$.

9.4.3 Various shuffling algorithms

Here we have a deck of n cards, and we repeatedly apply some random transformation to the deck to converge to a stationary distribution that is uniform over all permutations of the cards (usually this is obvious by symmetry, so we won't bother proving it). Our goal is to show that the expected **coupling time** at which our deck ends up in the same permutation as an initially-stationary deck is small. Typically we do this by linking identical cards on each side of the coupling, so that each linked pair moves through the same positions with each step. When all n cards are linked, we will have achieved $X = Y$.

Lest somebody try implementing one of these shuffling algorithms, it's probably worth mentioning that they are all terrible. If you actually want to shuffle an array of values, the usual approach is to swap an element chosen uniformly at random into the first position, then shuffle the remaining $n - 1$ positions recursively. This gives a uniform shuffle in $O(n)$ steps.

9.4.3.1 Move-to-top

This is a variant of card shuffling that is interesting mostly because it gives about the easiest possible coupling argument. At each step, we choose one of the cards uniformly at random (including the top card) and move it to the top of the deck. How long until the deck is fully shuffled, i.e., until the total variation distance between the actual distribution and the stationary distribution is bounded by ϵ ?

Here the trick is that when we choose a card to move to the top in the X process, we choose the same card in the Y process. It's not hard to see that this links the two cards together so that they are always in the same position in the deck in all future states. So to keep track of how well the coupling is working, we just keep track of how many cards are linked in this way, and observe that as soon as $n - 1$ are, the two decks are identical.

Note: Unlike some of the examples below, we don't consider two cards to be linked just because they are in the same position. We are only considering cards that have gone through the top position in the deck (which corresponds to some initial segment of the deck, viewed from above). The reason is that these cards never become unlinked: if we pick two cards from the initial segment, the cards above them move down together. But deeper cards that happen to match might become separated if we pull a card from one deck that is above the matched pair while its counterpart in the other deck is below the matched pair.

Given k cards linked in this way, the probability that the next step links another pair of cards is exactly $(n - k)/n$. So the expected time until we get $k + 1$ cards is $n/(n - k)$, and if we sum these waiting times for $k = 0 \dots n - 1$, we get nH_n , the waiting time for the coupon collector problem. So the bound on the mixing time is the same as for the random walk on a hypercube.

9.4.3.2 Random exchange of arbitrary cards

Here we pick two cards uniformly and independently at random and swap them. (Note there is a $1/n$ chance they are the same card; if we exclude this case, the Markov chain has period 2.) To get a coupling, we reformulate this process as picking a random card and a random location, and swapping the chosen card with whatever is in the chosen location in both the X and Y processes.

First let's observe that the number of linked cards never decreases. Let x_i, y_i be the position of card i in each process, and suppose $x_i = y_i$. If neither card i nor position x_i is picked, i doesn't move, and so it stays linked. If card i is picked, then both copies are moved to the same location; it stays linked. If position x_i is picked, then it may be that i becomes unlinked; but this only happens if the card j that is picked has $x_j \neq y_j$. In this case j becomes linked, and the number of linked cards doesn't drop.

Now we need to know how likely it is that we go from k to $k + 1$ linked cards. We've already seen a case where the number of linked cards increases; we pick two cards that aren't linked and a location that contains cards that aren't linked. The probability of doing this is $((n - k)/n)^2$, so our total

expected waiting time is $n^2 \sum (n - k)^{-2} = n^2 \sum k^{-2} \leq n\pi^2/6$. The final bound is $O(n^2 \log(1/\epsilon))$.

This bound is much worse than the bound for move-to-top, which is surprising. In fact, the real bound is $O(n \log n)$ with high probability, although the proof uses very different methods (see <http://www.stat.berkeley.edu/~aldous/RWG/Chap7.pdf>). This shows that the coupling method doesn't always give tight bounds (or perhaps we need a better coupling?).

9.4.3.3 Random exchange of adjacent cards

Suppose now that we only swap adjacent cards. Specifically, we choose one of the n positions i in the deck uniformly at random, and then swap the cards at positions i and $i + 1 \pmod n$ with probability $1/2$. (The $1/2$ is there for the usual reason of avoiding periodicity.)

So now we want a coupling between the X and Y processes where each possible swap occurs with probability $\frac{1}{2n}$ on both sides, but somehow we correlate things so that like cards are pushed together but never pulled apart. The trick is that we will use the same position i on both sides, but be sneaky about when we swap. In particular, we will aim to arrange things so that once some card is in the same position in both decks, both copies move together, but otherwise one copy changes its position by ± 1 relative to the other with a fixed probability $\frac{1}{2n}$.

The coupled process works like this. Let D be the set of indices i where the same card appears in both decks at position i or at position $i + 1$. Then we do:

1. For $i \in D$, swap $(i, i + 1)$ in both decks with probability $\frac{1}{2n}$.
2. For $i \notin D$, swap $(i, i + 1)$ in the X deck only with probability $\frac{1}{2n}$.
3. For $i \notin D$, swap $(i, i + 1)$ in the Y deck only with probability $\frac{1}{2n}$.
4. Do nothing with probability $\frac{|D|}{2n}$.

It's worth checking that the total probability of all these events is $|D|/2n + 2(n - |D|)/2n + |D|/2n = 1$. More important is that if we consider only one of the decks, the probability of doing a swap at $(i, i + 1)$ is exactly $\frac{1}{2n}$ (since we catch either case 1 or 2 for the X deck or 1 or 3 for the Y deck).

Now suppose that some card c is at position x in X and y in Y . If $x = y$, then both x and $x - 1$ are in D , so the only way the card can move is if it moves in both decks: linked cards stay linked. If $x \neq y$, then c

moves in deck X or deck Y , but not both. (The only way it can move in both is in case 1, where $i = x$ and $i + 1 = y$ or vice versa; but in this case i can't be in D since the copy of c at position x doesn't match whatever is in deck Y , and the copy at position y doesn't match what's in deck X .) In this case the distance $x - y$ goes up or down by 1 with equal probability $\frac{1}{2n}$. Considering $x - y \pmod n$, we have a "lazy" random walk that moves with probability $1/n$, with absorbing barriers at 0 and n . The worst-case expected time to converge is $n(n/2)^2 = n^3/4$, giving $\Pr[\text{time for } c \text{ to become linked} \geq \alpha n^3/8] \leq 2^{-\alpha}$ using the usual argument. Now apply the union bound to get $\Pr[\text{time for every } c \text{ to become linked} \geq \alpha n^3/8] \leq n2^{-\alpha}$ to get an expected coupling time of $O(n^3 \log n)$.

To simplify the argument, we assumed that the deck wraps around, so that we can swap the first and last card in addition to swapping physically adjacent cards. If we restrict to swapping i and $i + 1$ for $i \in \{0, \dots, n - 2\}$, we get a slightly different process that converges in essentially the same time $O(n^3 \log n)$. A result of David Bruce Wilson [Wil04] shows both this upper bound holds and that the bound is optimal up to a constant factor.

9.4.3.4 Real-world shuffling

In real life, the handful of people who still use physical playing cards tend to use a **dovetail shuffle**, which is closely approximated by the reverse of a process where each card in a deck is independently assigned to a left or right pile and the left pile is placed on top of the right pile. Coupling doesn't really help much here. Instead, the process can be analyzed using more sophisticated techniques due to Bayer and Diaconis [BD92]. The short version of the result is that $\Theta(\log n)$ shuffles are needed to randomize a deck of size n .

9.4.4 Path coupling

If the states of our Markov process are the vertices of a graph, we may be able to construct a coupling by considering a path between two vertices and showing how to shorten this path on average at each step. This technique is known as path coupling. Typically, the graph we use will be the graph of possible transitions of the underlying Markov chain (possibly after making all edges undirected).

There are two ideas at work here. The first is that the expected distance $\mathbb{E}[d(X_t, Y_t)]$ between X_t and Y_t in the graph gives an upper bound on $\Pr[X_t \neq Y_t]$ (by Markov's inequality, since if $X_t \neq Y_t$ then $d(X_t, Y_t) \geq 1$).

The second is that to show that $\mathbb{E}[d(X_{t+1}, Y_{t+1}) \mid \mathcal{F}_t] \leq \alpha \cdot d(X_t, Y_t)$ for some $\alpha < 1$, it is enough to show how to contract a single edge, that is, to show that $\mathbb{E}[d(X_{t+1}, Y_{t+1}) \mid d(X_t, Y_t) = 1] \leq \alpha$. The reason is that if we have a coupling that contracts one edge, we can apply this inductively along each edge in the path to get a coupling between all the vertices in the path that still leaves each pair with expected distance at most α . The result for the whole path then follows from linearity of expectation.

Formally, instead of just looking at X_t and Y_t , consider a path of intermediate states $X_t = Z_{0,t}Z_{1,t}Z_{2,t} \dots Z_{m,t} = Y_t$, where $d(Z_{i,t}, Z_{i+1,t}) = 1$ for each i (the vertices are adjacent in the graph). We now construct a coupling only for adjacent nodes that reduces their distance on average. The idea is that $d(X_t, Y_t) \leq \sum d(Z_{i,t}, Z_{i+1,t})$, so if the distance between each adjacent pair shrinks on average, so does the total length of the path.

The coupling on each edge gives a joint conditional probability

$$\Pr [Z_{i,t+1} = z'_i, Z_{i+1,t+1} = z'_{i+1} \mid Z_{i,t} = z_i, Z_{i+1,t} = z_{i+1}].$$

We can extract from this a conditional distribution on $Z_{i+1,t+1}$ given the other three variables:

$$\Pr [Z_{i+1,t+1} = z'_{i+1} \mid Z_{i,t+1} = z'_i, Z_{i,t} = z_i, Z_{i+1,t} = z_{i+1}].$$

Multiplying these conditional probabilities together lets us compute a joint distribution on X_{t+1}, Y_{t+1} conditioned on X_t, Y_t , which is the ordinary coupling we really want.

It's worth noting that the path is entirely notional, and we don't actually use it to construct an explicit coupling or even keep it around between steps of the coupled Markov chains. The only purpose of Z_0, Z_1, \dots, Z_m is to show that X and Y move closer together. Even though we could imagine that we are coalescing these nodes together to create a new path at each step (or throwing in a few extra nodes if some Z_i, Z_{i+1} move away from each other), we could also imagine that we start with a fresh path at each step and throw it away as soon as it has done its job.

9.4.5 Random walk on a hypercube

As a warm-up, let's redo the argument about lazy random walks on a hypercube from §9.4.2 using path coupling. Each state X^t or Y^t is an n -bit vector, and with probability $1/2$ a step flips one of the bits chosen uniformly at random. The obvious metric d is Hamming distance: $d(x, y)$ is the number of indices i for which $x_i \neq y_i$.

For path coupling, we only need to push adjacent Z_i^t and Z_{i+1}^t . Adjacency means that there is exactly one index j at which these two bit-vectors differ. We apply the following coupling (which looks suspiciously like the more generic coupling in §9.4.2):

1. Pick a random index r .
2. If $r \neq j$, which occurs with probability $1 - 1/n$, flip position r in both Z_i^t and Z_{i+1}^t with probability $1/2$. Whether we flip or not, we get $d(Z_i^{t+1}, Z_{i+1}^{t+1}) = 1$.
3. If $r = j$, pick a new bit value b uniformly at random and set position j in both Z_i^{t+1} and Z_{i+1}^{t+1} to b . From either side alone, this looks like flipping bit j with probability $1/2$, exactly as in the original process. But now $d(Z_i^{t+1}, Z_{i+1}^{t+1}) = 0$.

Averaging over both cases gives $\mathbb{E} [d(Z_{i+1}^t, Z_{i+1}^{t+1}) \mid \mathcal{F}_t] = 1 - 1/n$. It follows that $\mathbb{E} [d(X^t, Y^t) \mid \mathcal{F}_0] \leq (1 - 1/n)^t d(X_0, Y_0) \leq ne^{-n/t}$. Since $d(X^t, Y^t)$ is always at least 1 if it's not zero, this gives $\Pr [X^t \neq Y^t] \leq ne^{-n/t}$ by Markov's inequality, so $d_{TV}(X^t, Y^t) \leq \epsilon$ after $O(n \log \frac{n}{\epsilon})$ steps.

9.4.5.1 Sampling graph colorings

For this example, we'll look at sampling k -colorings of a graph with maximum degree Δ . We will assume that $k \geq 2\Delta + 1$.⁹ For smaller values of k , it might still be the case that the chain converges reasonably quickly for some graphs, but our analysis will not show this.

Unlike the hypercube case, the path coupling we will construct might allow the distance between X^t and Y^t to rise with some probability. But the expected distance will always decrease, which is enough.

Consider the following chain on proper k -colorings of a graph with maximum degree Δ . At each step, we choose one of the n nodes v , compute the set S of colors not found on any of v 's neighbors, and recolor v with a color chosen uniformly from S (which may be its original color).

Suppose $p_{ij} \neq 0$. Then i and j differ in at most one place v , and so the set S of permitted colors for each process—those not found on v 's neighbors—are the same. This gives $p_{ij} = p_{ji} = \frac{1}{n \cdot |S|}$, and the detailed balance equations (9.3.1) hold when π_i is constant for all i . So we have a reversible Markov

⁹An analysis based on a standard coupling for a different version of the Markov chain that works for the same bound on k is given in [MU05, §11.5]. More sophisticated results and a history of the problem can be found in [DGM02].

chain with a uniform stationary distribution. Now we will apply a path coupling to show that we converge to this stationary distribution reasonably quickly when k is large enough.

We'll think of colorings as vectors. Given two colorings x and y , let $d(x, y)$ be the Hamming distance between them, which is the number of nodes u for which $x_u \neq y_u$. To show convergence, we will construct a coupling that shows that $d(X^t, Y^t)$ converges to 0 over time starting from arbitrary initial points X^0 and Y^0 .

A complication is that it's not immediately evident that the length of the shortest path from X^t to Y^t in the transition graph of our Markov chain is $d(X^t, Y^t)$. The problem is that it may not be possible to transform X^t into Y^t one node at a time without producing improper colorings. With enough colors, we can explicitly construct a short path between X^t and Y^t that uses only proper colorings; but for this particular process it is easier to simply extend the Markov chain to allow improper colorings, and show that our coupling works anyway. This also allows us to start with an improper coloring for X^0 if we are particularly lazy. The stationary distribution is not affected, because if i is a proper coloring and j is an improper coloring that differs from i in exactly one place, we have $p_{ij} = 0$ and $p_{ji} \neq 0$, so the detailed balance equations hold with $\pi_j = 0$.

The natural coupling to consider given adjacent X^t and Y^t is to pick the same node and the same new color for both, provided we can do so. If we pick the one node v on which they differ, and choose a color that is not used by any neighbor (which will be the same for both copies of the process, since all the neighbors have the same colors), then we get $X^{t+1} = Y^{t+1}$; this event occurs with probability at least $1/n$. If we pick a node that is neither v nor adjacent to it, then the distance between X and Y doesn't change; either both get a new identical color or both don't.

Things get a little messier when we pick some node u adjacent to v , an event that occurs with probability at most Δ/n . Let c be the color of v in X^t , c' the color of u in Y^t , and T the set of colors that do not appear among the other neighbors of u . Let $\ell = |T| \geq k - (\Delta - 1)$.

Conditioning on choosing u to recolor, X^{t+1} picks a color uniformly from $T \setminus \{c\}$ and Y^{t+1} picks a color uniformly from $T \setminus \{c'\}$. We'd like these colors to be the same if possible, but these are not the same sets, and they aren't even necessarily the same size.

There are three cases:

1. Neither c nor c' are in T . Then X^{t+1} and Y^{t+1} are choosing a new color from the same set, and we can make both choose the same color:

the distance between X and Y is unchanged.

2. Exactly one of c and c' is in T . Suppose that it's c . Then $|T \setminus \{c\}| = \ell - 1$ and $|T \setminus \{c'\}| = |T| = \ell$. Let X^{t+1} choose a new color c'' first. Then let $Y_u^{t+1} = c''$ with probability $\frac{\ell-1}{\ell}$ (this gives a probability of $\frac{1}{\ell}$ of picking each color in $T \setminus \{c\}$, which is what we want), and let $Y_u^{t+1} = c$ with probability $\frac{1}{\ell}$. Now the distance between X and Y increases with probability $\frac{1}{\ell}$.
3. Both c and c' are in T . For each c'' in $T \setminus \{c, c'\}$, let $X_u^{t+1} = Y_u^{t+1} = c''$ with probability $\frac{1}{\ell-1}$; since there are $\ell - 2$ such c'' , this accounts for $\frac{\ell-2}{\ell-1}$ of the probability. Assign the remaining $\frac{1}{\ell-1}$ to $X_u^{t+1} = c', Y_u^{t+1} = c$. In this case the distance between X and Y increases with probability $\frac{1}{\ell-1}$, making this the worst case.

Putting everything together, we have a $1/n$ chance of picking a node that guarantees to reduce $d(X, Y)$ by 1, and at most a Δ/n chance of picking a node that may increase $d(X, Y)$ by at most $\frac{1}{\ell-1}$ on average, where $\ell \geq k - \Delta + 1$, giving a maximum expected increase of $\frac{\Delta}{n} \cdot \frac{1}{k-\Delta}$. So

$$\begin{aligned} \mathbb{E} \left[d(X^{t+1}, Y^{t+1}) - d(X^t, Y^t) \mid d(X^t, Y^t) = 1 \right] &\leq \frac{-1}{n} + \frac{\Delta}{n} \cdot \frac{1}{k-\Delta} \\ &= \frac{1}{n} \left(-1 + \frac{\Delta}{k-\Delta} \right) \\ &= \frac{1}{n} \left(\frac{-(k-\Delta) + \Delta}{k-\Delta} \right) \\ &= -\frac{1}{n} \left(\frac{k-2\Delta}{k-\Delta} \right). \end{aligned}$$

So we get

$$\begin{aligned} d_{TV}(X^t, Y^t) &\leq \Pr \left[X^t \neq Y^t \right] \\ &\leq \left(1 - \frac{1}{n} \cdot \frac{k-2\Delta}{k-\Delta} \right)^t \cdot \mathbb{E} \left[d(X^0, Y^0) \right] \\ &\leq \exp \left(-\frac{t}{n} \cdot \frac{k-2\Delta}{k-\Delta} \right) \cdot n. \end{aligned}$$

For fixed k and Δ with $k > 2\Delta$, this is $e^{-\Theta(t/n)}n$, which will be less than ϵ for $t = \Omega(n(\log n + \log(1/\epsilon)))$.

9.4.5.2 Simulated annealing

Recall that the Metropolis-Hastings algorithm constructs a reversible Markov chain with a desired stationary distribution from any reversible Markov chain on the same states (see §9.3.4 for details.)

A variant, which generally involves tinkering with the chain while it's running, is the global optimization heuristic known as **simulated annealing**. Here we have some function g that we are trying to minimize. So we set $f(i) = \exp(-\alpha g(i))$ for some $\alpha > 0$. Running Metropolis-Hastings gives a stationary distribution that is exponentially weighted to small values of g ; if i is the global minimum and j is some state with high $g(j)$, then $\pi(i) = \pi(j) \exp(\alpha(g(j) - g(i)))$, which for large enough α goes a long way towards compensating for the fact that in most problems there are likely to be exponentially more bad j 's than good i 's. The problem is that the same analysis applies if i is a local minimum and j is on the boundary of some depression around i ; large α means that it is exponentially unlikely that we escape this depression and find the global minimum.

The simulated annealing hack is to vary α over time; initially, we set α small, so that we get conductance close to that of the original Markov chain. This gives us a sample that is roughly uniform, with a small bias towards states with smaller $g(i)$. After some time we increase α to force the process into better states. The hope is that by increasing α slowly, by the time we are stuck in some depression, it's a deep one—optimal or close to it. If it doesn't work, we can randomly restart and/or decrease α repeatedly to jog the chain out of whatever depression it is stuck in. How to do this effectively is deep voodoo that depends on the structure of the underlying chain and the shape of $g(i)$, so most of the time people who use simulated annealing just try it out with some generic **annealing schedule** and hope it gives some useful result. (This is what makes it a heuristic rather than an algorithm. Its continued survival is a sign that it does work at least sometimes.)

Here are toy examples of simulated annealing with provable convergence times.

Single peak Let's suppose x is a random walk on an n -dimensional hypercube (i.e., n -bit vectors where we set 1 bit at a time), $g(x) = |x|$, and we want to maximize g . Now a transition that increase $|x|$ is accepted always and a transition that decreases $|x|$ is accepted only with probability $e^{-\alpha}$. For large enough α , this puts a constant fraction of π on the single peak at $x = \mathbf{1}$; the observation is that that there are only $\binom{n}{k} \leq n^k$ points with k zeros, so the total weight of all points is at most $\pi(\mathbf{1}) \sum_{k \geq 0} n^k \exp(-\alpha k) =$

$\pi(\mathbf{1}) \sum \exp(\ln n - \alpha)^k = \pi(\mathbf{1}) / (1 - \exp(\ln n - \alpha)) = \pi(\mathbf{1}) \cdot O(1)$ when $\alpha > \ln n$, giving $\pi(\mathbf{1}) = \Omega(1)$ in this case.

So what happens with convergence? Let $p = \exp(-\alpha)$. Let's try doing a path coupling between two adjacent copies x and y of the Metropolis-Hastings process, where we first pick a bit to change, then pick a value to assign to it, accepting the change in both processes if we can. The expected change in $|x - y|$ is then $(1/2n)(-1 - p)$, since if we pick the bit where x and y differ, we have probability $1/2n$ of setting both to 1 and probability $p/2n$ of setting both to 0, and if we pick any other bit, we get the same distribution of outcomes in both processes. This gives a general bound of $E[|X_{t+1} - Y_{t+1}| \mid |X_t - Y_t|] \leq (1 - (1+p)/2n)|X_t - Y_t|$, from which we have $E[|X_t - Y_t|] \leq \exp(-t(1+p)/2n) E[|X_0 - Y_0|] \leq n \exp(-t(1+p)/2n)$. So after $t = 2n/(1+p) \ln(n/\epsilon)$ steps, we expect to converge to within ϵ of the stationary distribution in total variation distance. This gives an $O(n \log n)$ algorithm for finding the peak.

This is kind of a silly example, but if we suppose that g is better disguised (for example, $g(x)$ could be $|x \oplus r|$ where r is a random bit vector), then we wouldn't really expect to do much better than $O(n)$. So $O(n \log n)$ is not bad for an algorithm with no brains at all.

Somewhat smooth functions Now we'll let $g : 2^n \rightarrow \mathbb{N}$ be some arbitrary Lipschitz function, that is, a function with the property that $|g(x) - g(y)| \leq |x - y|$, and ask for what values of $p = e^{-\alpha}$ the Metropolis-Hastings walk with $f(i) = e^{-\alpha g(i)}$ can be shown to converge quickly. Given adjacent states x and y , with $x_i \neq y_i$ but $x_j = y_j$ for all $j \neq i$, we still have a probability of at least $(1+p)/2n$ of coalescing the states by setting $x_i = y_i$. But now there is a possibility that if we try to move to $(x[j/b], y[j/b])$ for some j and b , that x rejects while y does not or vice versa (note if $x_j = y_j = b$, we don't move in either copy of the process). Conditioning on j and b , this occurs with probability $1 - p$ precisely when $x[j/b] < x$ and $y[j/b] \geq y$ or vice versa, giving an expected increase in $|x - y|$ of $(1-p)/2n$. We still get an expected net change of $-2p/2n = -p/n$ provided there is only one choice of j and b for which this occurs. So we converge in time $\tau(\epsilon) \leq (n/p) \log(n/\epsilon)$ in this case.¹⁰

One way to think of this is that the shape of the neighborhoods of nearby points is similar. If I go up in a particular direction from point x , it's very likely that I go up in the same direction from some neighbor y of x .

¹⁰You might reasonably ask if such functions g exist. One example is $g(x) = (x_1 \oplus x_2) + \sum_{i>2} x_i$.

If there are more bad choices for j and b , then we need a much larger value of p : the expected net change is now $(k(1-p)-1-p)/2n = (k-1-(k+1)p)/2n$, which is only negative if $p > (k-1)/(k+1)$. This gives much weaker pressure towards large values of g , which still tends to put us in high neighborhoods but creates the temptation to fiddle with α to try to push us even higher once we think we are close to a peak.

9.5 Spectral methods for reversible chains

(See also <http://www.stat.berkeley.edu/~aldous/RWG/Chap3.pdf>, from which many of the details in the notes below are taken.)

The problem with coupling is that (a) it requires cleverness to come up with a good coupling; and (b) in many cases, even that doesn't work—there are problems for which no coupling that only depends on current and past transitions coalesces in a reasonable amount of time.¹¹ When we run into these problems, we may be able to show convergence instead using a linear-algebraic approach, where we look at the eigenvalues of the transition matrix of our Markov chain. This approach works best for reversible Markov chains, where $\pi_i p_{ij} = \pi_j p_{ji}$ for all states i and j and some distribution π .

9.5.1 Spectral properties of a reversible chain

Suppose that P is the transition matrix of an irreducible, reversible Markov chain. Then it has a unique stationary distribution π that is a left **eigenvector** corresponding to the **eigenvalue** 1, which just means that $\pi P = 1\pi$. For irreducible aperiodic chains, P will have a total of n real eigenvalues $\lambda_1 > \lambda_2 \geq \lambda_3 \geq \dots \geq \lambda_n$, where $\lambda_1 = 1$ and $\lambda_n > -1$. (This follows for symmetric chains from the **Perron-Frobenius theorem**, which we will not attempt to prove here; for general irreducible aperiodic reversible chains, we will show below that we can reduce to the symmetric case.) Each eigenvalue λ_i has a corresponding eigenvector u^i , a nonzero vector that satisfies $u^i P = \lambda_i u^i$. In Markov chain terms, these eigenvectors correspond to deviations from the stationary distribution that shrink over time.

For example, the transition matrix

$$S = \begin{bmatrix} p & q \\ q & p \end{bmatrix}$$

¹¹Such a coupling is called a **causal coupling**; an example of a Markov chain for which causal couplings are known not to work is the one used for sampling perfect matchings in bipartite graphs as described in §9.5.8.4 [KR99].

corresponding to a Markov chain on two states that stays in the same state with probability p and switches to the other state with probability $q = 1 - p$ has eigenvectors $u_1 = \begin{bmatrix} 1 & 1 \end{bmatrix}$ and $u_2 = \begin{bmatrix} 1 & -1 \end{bmatrix}$ with corresponding eigenvalues $\lambda_1 = 1$ and $\lambda_2 = p - q$, as shown by computing

$$\begin{bmatrix} 1 & 1 \end{bmatrix} \begin{bmatrix} p & q \\ q & p \end{bmatrix} = \begin{bmatrix} p+q & q+p \end{bmatrix} = 1 \cdot \begin{bmatrix} 1 & 1 \end{bmatrix}$$

and

$$\begin{bmatrix} 1 & -1 \end{bmatrix} \begin{bmatrix} p & q \\ q & p \end{bmatrix} = \begin{bmatrix} p-q & q-p \end{bmatrix} = (p-q) \cdot \begin{bmatrix} 1 & -1 \end{bmatrix}.$$

9.5.2 Analysis of symmetric chains

To make our life easier, we will assume that in addition to being reversible, our Markov chain has a uniform stationary distribution. Then $\pi_i = \pi_j$ for all i and j , and so reversibility implies $p_{ij} = p_{ji}$ for all i and j as well, meaning that the transition matrix P is symmetric. Symmetric matrices have the very nice property that their eigenvectors are all orthogonal (this is the **spectral theorem**), which allows for a very straightforward decomposition of the distribution on the initial state (represented as a vector x_0) as a linear combination of the eigenvectors. For example, if initially we put all our weight on state 1, we get $x_0 = \begin{bmatrix} 1 & 0 \end{bmatrix} = \frac{1}{2}u^1 + \frac{1}{2}u^2$.

If we now take a step in the chain, we multiply x by P . We can express this as

$$\begin{aligned} xP &= \left(\frac{1}{2}u^1 + \frac{1}{2}u^2 \right) P \\ &= \frac{1}{2}u^1 P + \frac{1}{2}u^2 P \\ &= \frac{1}{2}\lambda_1 u^1 + \frac{1}{2}\lambda_2 u^2. \end{aligned}$$

This uses the defining property of eigenvectors, that $u^i P = \lambda_i u^i$.

In general, if $x = \sum_i a_i u^i$, then $xP = \sum_i a_i \lambda_i u^i$ and $xP^t = \sum_i a_i \lambda_i^t u^i$. For any eigenvalue λ_i with $|\lambda_i| < 1$, λ_i^t goes to zero in the limit. So only those eigenvectors with $\lambda_i = 1$ survive. For irreducible aperiodic chains, these consist only of the stationary distribution $\pi = u^i / \|u^i\|_1$. For periodic chains, there may be some additional eigenvalues with $|\lambda_i| = 1$, but the only

possibility that arises for a reversible chain is $\lambda_n = -1$, corresponding to a chain with period 2.¹²

Assuming that $|\lambda_2| \geq |\lambda_n|$, as t grows large λ_2^t will dominate the other smaller eigenvalues, and so the size of λ_2 will control the rate of convergence of the underlying Markov process.

This assumption is always true for lazy walks that stay put with probability $1/2$, because all eigenvalues of a lazy walk are non-negative. The reason is that any such walk has transition matrix $\frac{1}{2}(P + I)$, where I is the identity matrix and P is the transition matrix of the unlazy version of the walk. If $xP = \lambda x$ for some x and λ , then $x\left(\frac{1}{2}(P + I)\right) = \left(\frac{1}{2}(\lambda + 1)\right)x$. This means that x is still an eigenvector of the lazy walk, and its corresponding eigenvalue is $\frac{1}{2}(\lambda + 1) \geq \frac{1}{2}((-1) + 1) \geq 0$.

But what does having small λ_2 mean for total variation distance? If x^t is a vector representing the distribution of our position at time t , then $d_{TV} = \frac{1}{2} \sum_{i=1}^n |x_i^t - \pi_i| = \frac{1}{2} \|x^t - \pi\|_1$. But we know that $x^0 = \pi + \sum_{i=2}^n c_i u^i$ for some coefficients c_i , and $x^t = \pi + \sum_{i=2}^n \lambda_i^t c_i u^i$. So we are looking for a bound on $\|\sum_{i=2}^n \lambda_i^t c_i u^i\|_1$.

It turns out that it is easier to get a bound on the ℓ_2 norm $\|\sum_{i=2}^n \lambda_i^t c_i u^i\|_2$. Here we use the fact that the eigenvectors are orthogonal. This means that the Pythagorean theorem holds and $\|\sum_{i=2}^n \lambda_i^t c_i u^i\|_2^2 = \sum_{i=2}^n \lambda_i^{2t} c_i^2 \|u^i\|_2^2 = \sum_{i=2}^n \lambda_i^{2t} c_i^2$ if we normalize each u^i so that $\|u^i\|_2^2 = 1$. But then

$$\begin{aligned} \|x^t - \pi\|_2^2 &= \sum_{i=2}^n \lambda_i^{2t} c_i^2 \\ &\leq \sum_{i=2}^n \lambda_2^{2t} c_i^2 \\ &= \lambda_2^{2t} \sum_{i=2}^n c_i^2 \\ &= \lambda_2^{2t} \|x^0 - \pi\|_2^2. \end{aligned}$$

Now take the square root of both sides to get

$$\|x^t - \pi\|_2 \leq \lambda_2^t \|x^0 - \pi\|_2.$$

¹²Chains with periods greater than 2 (which are never reversible) have pairs of complex-valued eigenvalues that are roots of unity, which happen to cancel out to only produce real probabilities in vP^t . Chains that aren't irreducible will have one eigenvector with eigenvalue 1 for each final component; the stationary distributions of these chains are linear combinations of these eigenvectors (which are just the stationary distributions on each component).

To translate this back into d_{TV} , we use the inequality

$$\|x\|_2 \leq \|x\|_1 \leq \sqrt{n} \cdot \|x\|_2,$$

which holds for any n -dimensional vector x . Because $\|x^0\|_1 = \|\pi\|_1 = 1$, $\|x^0 - \pi\|_1 \leq 2$ by the triangle inequality, which also gives $\|x^0 - \pi\|_2 \leq 2$. So

$$\begin{aligned} d_{TV}(x^t, \pi) &= \frac{1}{2} \|x^t - \pi\|_1 \\ &\leq \frac{\sqrt{n}}{2} \lambda_2^t \|x^0 - \pi\|_2 \\ &\leq \lambda_2^t \sqrt{n}. \end{aligned}$$

If we want to get $d_{TV}(x^t, \pi) \leq \epsilon$, we will need $t \ln(\lambda_2) + \ln \sqrt{n} \leq \ln \epsilon$ or $t \geq \frac{1}{\ln(1/\lambda_2)} \left(\frac{1}{2} \ln n + \ln(1/\epsilon) \right)$.

The factor $\frac{1}{\ln(1/\lambda_2)} = \frac{1}{-\ln \lambda_2}$ can be approximated by $\tau_2 = \frac{1}{1-\lambda_2}$, which is called the **mixing rate** or **relaxation time** of the Markov chain. Indeed, our old friend $1 + x \leq e^x$ implies $\ln(1 + x) \leq x$, which gives $\ln \lambda_2 \leq \lambda_2 - 1$ and thus $\frac{1}{-\ln \lambda_2} \leq \frac{1}{1-\lambda_2} = \tau_2$. So $\tau_2 \left(\frac{1}{2} \ln n + \ln(1/\epsilon) \right)$ gives a conservative estimate on the time needed to achieve $d_{TV}(x^t, \pi) \leq \epsilon$ starting from an arbitrary distribution.

It's worth comparing this to the mixing time $t_{\text{mix}} = \arg \min_t d_{TV}(x^t, \pi) \leq 1/4$ that we used with coupling. With τ_2 , we have to throw in an extra factor of $\frac{1}{2} \ln n$ to get the bound down to 1, but after that the bound continues to drop by a factor of e every τ_2 steps. With t_{mix} , we have to go through another t_{mix} steps for each factor of 2 improvement in total variation distance, even if the initial drop avoids the log factor. Since we can't always tell whether coupling arguments or spectral arguments will do better for a particular chain, and since convergence bounds are often hard to obtain no matter how many techniques we throw at them, we will generally be happy with any reasonable bound on either τ_2 or t_{mix} .

9.5.3 Analysis of asymmetric chains

If the stationary distribution is not uniform, then in general the transition matrix will not be symmetric. We can make it symmetric by scaling the probability of being in the i -th state by $\pi_i^{-1/2}$. The idea is to decompose our transition matrix P as $\Pi A \Pi^{-1}$, where Π is a diagonal matrix with

$\Pi_{ii} = \pi_i^{-1/2}$, and $A_{ij} = \sqrt{\frac{\pi_i}{\pi_j}} P_{ij}$. Then A is symmetric, because

$$\begin{aligned} A_{ij} &= \sqrt{\frac{\pi_i}{\pi_j}} P_{ij} \\ &= \sqrt{\frac{\pi_i}{\pi_j} \frac{\pi_j}{\pi_i}} P_{ji} \\ &= \sqrt{\frac{\pi_j}{\pi_i}} P_{ji} \\ &= A_{ji}. \end{aligned}$$

This means in particular that A has orthogonal eigenvectors u^1, u^2, \dots, u^n with corresponding eigenvalues $\lambda_1 \geq \lambda_2 \geq \dots \geq \lambda_n$. These eigenvalues carry over to $P = \Pi A \Pi^{-1}$, since $(u^i \Pi^{-1}) P = u^i \Pi^{-1} \Pi A \Pi^{-1} = u^i A \Pi^{-1} = \lambda_i u^i$.

So now given an initial distribution x_0 , we can first pass it through Π^{-1} and then apply the same reasoning as before to show that $\|\pi \Pi^{-1} - x \Pi^{i-1}\|_2$ shrinks by λ_2 with every step of the Markov chain. The difference now is that the initial distance is affected by the scaling we did in Π^{-1} ; so instead of getting $d_{TV}(x^t, \pi) \leq \lambda_2^t \sqrt{n}$, we get $d_{TV}(x^t, \pi) \leq \lambda_2^t (\pi_{\min})^{-1/2}$, where π_{\min} is the smallest probability of any single node in the stationary distribution π . The bad initial x_0 in this case is the one that puts all its weight on this node, since this maximizes its distance from π after scaling.

(A uniform π is a special case of this, since when $\pi_i = 1/n$ for all i , $\pi_{\min}^{-1/2} = (1/n)^{-1/2} = \sqrt{n}$.)

For more details on this, see [AF01, §3.4].

So now we just need a tool for bounding λ_2 . For a small chain with a known transition matrix, we can just feed it to our favorite linear algebra library, but most of the time we will not be able to construct the matrix explicitly. So we need a way to bound λ_2 indirectly, in terms of other structural properties of our Markov chain.

9.5.4 Conductance

The **conductance** or **Cheeger constant** to $\Phi(S)$ of a set S of states in a Markov chain is

$$\Phi(S) = \frac{\sum_{i \in S, j \notin S} \pi_i P_{ij}}{\pi(S)}. \quad (9.5.1)$$

This is the probability of leaving S on the next step starting from the stationary distribution conditioned on being in S . The conductance is a

measure of how easy it is to escape from a set. It can also be thought of as a weighted version of edge expansion.

The conductance of a Markov chain as a whole is obtained by taking the minimum of $\Phi(S)$ over all S that occur with probability at most $1/2$:

$$\Phi = \min_{0 < \pi(S) \leq 1/2} \Phi(S). \quad (9.5.2)$$

The usefulness of conductance is that it bounds λ_2 :

Theorem 9.5.1. *In a reversible Markov chain,*

$$1 - 2\Phi \leq \lambda_2 \leq 1 - \Phi^2/2. \quad (9.5.3)$$

The bound (9.5.3) is known as the **Cheeger inequality**. We won't attempt to prove it here, but the intuition is that in order for a reversible chain not to mix, it has to get stuck in some subset of the states. Having high conductance prevents this disaster, while having low conductance causes it.

For lazy walks we always have $\lambda_2 = \lambda_{\max}$, and so we can convert (9.5.3) to a bound on the relaxation time:

Corollary 9.5.2.

$$\frac{1}{2\Phi} \leq \tau_2 \leq \frac{2}{\Phi^2}. \quad (9.5.4)$$

In other words, high conductance implies low relaxation time and vice versa, up to squaring.

9.5.5 Easy cases for conductance

For very simple Markov chains we can compute the conductance directly. Consider a lazy random walk on a cycle. Any proper subset S has at least two outgoing edges, each of which carries a flow of $1/4n$, giving $\Phi_S \geq (1/2n)/\pi(S)$. If we now take the minimum of Φ_S over all S with $\pi(S) \leq 1/2$, we get $\phi \geq 1/n$, which gives $\tau_2 \leq 2n^2$. This is essentially the same bound as we got from coupling.

Here's a slightly more complicated chain. Take two copies of K_n , where n is odd, and join them by a path with n edges. Now consider Φ_S for a lazy random walk on this graph where S consists of half the graph, split at the edge in the middle of the path. There is a single outgoing edge uv , with $\pi(u) = d(u)/2|E| = 2/(2n(n-1)n/2 + n) = 2n^{-2}$ and $p_{uv} = 1/4$, for

$\pi(u)p_{uv} = n^{-2}/2$. By symmetry, we have $\pi(S) \rightarrow 1/2$ as $n \rightarrow \infty$, giving $\Phi_S \rightarrow n^{-2}(1/2)/(1/2) = n^{-2}$. So we have $n^2/2 \leq \tau_2 \leq 2n^4$.

How does this compare to the actual mixing time? In the stationary distribution, we have a constant probability of being in each of the copies of K_n . Suppose we start in the left copy. At each step there is a $1/n$ chance that we are sitting on the path endpoint. We then step onto the path with probability $1/n$, and reach the other end before coming back with probability $1/n$. So (assuming we can make this extremely sloppy handwaving argument rigorous) it takes at least n^3 steps on average before we reach the other copy of K_n , which gives us a rough estimate of the mixing time of $\Theta(n^3)$. In this case the exponent is exactly in the middle of the bounds derived from conductance.

9.5.6 Edge expansion using canonical paths

(Here and below we are mostly following the presentation of Guruswami [Gur00], but with slightly different examples.)

For more complicated Markov chains, it is helpful to have a tool for bounding conductance that doesn't depend on intuiting what sets have the smallest boundary. The **canonical paths** [JS89] technique does this by assigning a unique path γ_{xy} from each state x to each state y in a way that doesn't send too many paths across any one edge. So if we have a partition of the state space into sets S and T , then there are $|S| \cdot |T|$ paths from states in S to states in T , and since (a) every one of these paths crosses an S - T edge, and (b) each S - T edge carries at most ρ paths, there must be at least $|S| \cdot |T|/\rho$ edges from S to T . Note that because there is no guarantee we chose good canonical paths, this is only useful for getting lower bounds on conductance—and thus upper bounds on mixing time—but this is usually what we want.

Let's start with a small example. Let $G = C_n \square C_m$, the $n \times m$ torus. A lazy random walk on this graph moves north, east, south, or west with probability $1/8$ each, wrapping around when the coordinates reach n or m . Since this is a random walk on a regular graph, the stationary distribution is uniform. What is the relaxation time?

Intuitively, we expect it to be $O(\max(n, m)^2)$, because we can think of this two-dimensional random walk as consisting of two one-dimensional random walks, one in the horizontal direction and one in the vertical direction, and we know that a random walk on a cycle mixes in $O(n^2)$ time. Unfortunately, the two random walks are not independent: every time I take a horizontal step is a time I am not taking a vertical step. We *can* show that the expected

coupling time is $O(n^2 + m^2)$ by running two sequential instances of the coupling argument for the cycle, where we first link the two copies in the horizontal direction and then in the vertical direction. So this gives us one bound on the mixing time. But what happens if we try to use conductance?

Here it is probably easiest to start with just a cycle. Given points x and y on C_n , let the canonical path γ_{xy} be a shortest path between them, breaking ties so that half the paths go one way and half the other. Then each edge is crossed by exactly k paths of length k for each $k = 1 \dots (n/2 - 1)$, and at most $n/4$ paths of length $n/2$ (0 if n is odd), giving a total of $\rho \leq \binom{n/2}{2} + \frac{n}{4} = \frac{(n/2-1)(n/2)}{2} + \frac{n}{4} = n^2/8$ paths across the edge.

If we now take an S - T partition where $|S| = m$, we get at least $m(n - m)/\rho = 8m(n - m)/n^2$ S - T edges. This peaks at $m = n/2$, where we get 2 edges—exactly the right number—and in general when $m \leq n/2$ we get at least $8m(n/2)/n^2 = 4m/n$ outgoing edges, giving a conductance $\Phi_S \geq (1/4n)(4m/n)/(m/n) = 1/n$.

This is essentially what we got before, except we have to divide by 2 because we are doing a lazy walk. Note that for small m , the bound is a gross underestimate, since we know that every nonempty proper subset has at least 2 outgoing edges.

Now let's go back to the torus $C_n \square C_m$. Given x and y , define γ_{xy} to be the L-shaped path that first changes x_1 to y_1 by moving the shortest distance vertically, then changes x_2 to y_2 by moving the shortest distance horizontally. For a vertical edge, the number of such paths that cross it is bounded by $n^2m/8$, since we get at most $n^2/8$ possible vertical path segments and for each such vertical path segment there are m possible horizontal destinations to attach to it. For a horizontal edge, n and m are reversed, giving $m^2n/8$ paths. To make our life easier, let's assume $n \geq m$, giving a maximum of $\rho = n^2m/8$ paths over any edge.

For $|S| \leq nm/2$, this gives at least

$$\frac{|S| \cdot |G \setminus S|}{\rho} \geq \frac{|S|(nm/2)}{n^2m/8} = \frac{4|S|}{n}$$

outgoing edges. We thus have

$$\Phi(S) \geq \frac{1}{8nm} \cdot \frac{4|S|/n}{|S|/nm} = \frac{1}{2n}.$$

This gives $\tau_2 \leq 2/\Phi^2 \leq 8n^2$. Given that we assumed $n \geq m$, this is essentially the same $O(n^2 + m^2)$ bound that we would get from coupling.

9.5.7 Congestion

For less symmetric chains, we weight paths by the probabilities of their endpoints when counting how many cross each edge, and treat the flow across the edge as a capacity. This gives the **congestion** of a collection of canonical paths $\Gamma = \{\gamma_{xy}\}$, which is computed as

$$\rho(\Gamma) = \max_{uv \in E} \frac{1}{\pi_u p_{uv}} \sum_{\gamma_{xy} \ni uv} \pi_x \pi_y,$$

and we define $\rho = \min_{\Gamma} \rho(\Gamma)$.

The intuition here is that the congestion bounds the ratio between the canonical path flow across an edge and the Markov chain flow across the edge. If the congestion is not too big, this means that any cut that has a lot of canonical path flow across it also has a lot of Markov chain flow. When $\pi(T) \geq \frac{1}{2}$, the total canonical path flow $\pi(S)\pi(T)$ is at least $\frac{1}{2}\pi(S)$. This means that when $\pi(S)$ is large but less than $\frac{1}{2}$, the Markov chain flow leaving S is also large. This gives a lower bound on conductance.

Formally, we have the following lemma:

Lemma 9.5.3. *For any reversible aperiodic irreducible Markov chain,*

$$\Phi \geq \frac{1}{2\rho} \tag{9.5.5}$$

from which it follows that

$$\tau_2 \leq 8\rho^2. \tag{9.5.6}$$

Proof. Pick some set of canonical paths Γ with $\rho(\Gamma) = \rho$. Now pick some S - T partition with $\phi(S) = \phi$. Consider a flow where we route $\pi(x)\pi(y)$ units of flow along each path γ_{xy} with $x \in S$ and $y \in T$. This gives a total flow of $\pi(S)\pi(T) \geq \pi(S)/2$. We are going to show that we need a lot of capacity across the S - T cut to carry this flow, which will give the lower bound on conductance.

For any S - T edge uv , we have

$$\frac{1}{\pi_u p_{uv}} \sum_{\gamma_{xy} \ni uv} \pi_x \pi_y \leq \rho$$

or

$$\sum_{\gamma_{xy} \ni uv} \pi_x \pi_y \leq \rho \pi_u p_{uv}.$$

Since each S - T path crosses at least one S - T edge, we have

$$\begin{aligned} \pi(S)\pi(T) &= \sum_{x \in S, y \in T} \pi_x \pi_y \\ &\leq \sum_{u \in S, v \in T, uv \in E} \sum_{\gamma_{xy} \ni uv} \pi_x \pi_y \\ &\leq \sum_{u \in S, v \in T, uv \in E} \rho \pi_u p_{uv} \\ &= \rho \sum_{u \in S, v \in T, uv \in E} \pi_u p_{uv}. \end{aligned}$$

But then

$$\begin{aligned} \Phi(S) &= \frac{\sum_{u \in S, t \in T, uv \in E} \pi_u p_{uv}}{\pi_S} \\ &\geq \frac{\pi(S)\pi(T)/\rho}{\pi(S)} \\ &= \frac{\pi(T)}{\rho} \\ &\geq \frac{1}{2\rho}. \end{aligned}$$

To get the bound on τ_2 , use (9.5.4) to compute $\tau_2 \leq 2/\Phi^2 \leq 8\rho^2$. \square

9.5.8 Examples

Here are some more examples of applying canonical paths.

9.5.8.1 Lazy random walk on a line

Consider a lazy random walk on a line with reflecting barriers at 0 and $n - 1$. Here $\pi_x = \frac{1}{n}$ for all x . There aren't a whole lot of choices for canonical paths; the obvious choice for γ_{xy} with $x < y$ is $x, x + 1, x + 2, \dots, y$. This puts $(n/2)^2$ paths across the middle edge (which is the most heavily loaded), each of which has weight $\pi_x \pi_y = n^{-2}$. So the congestion is $\frac{1}{(1/n)(1/4)}(n/2)^2 n^{-2} = n$, giving a mixing time of at most $8n^2$. This is a pretty good estimate.

9.5.8.2 Random walk on a hypercube

Let's try a more complicated example: the random walk on a hypercube from §9.4.2. Here at each step we pick some coordinate uniformly at random

and set it to 0 or 1 with equal probability; this gives a transition probability $p_{uv} = \frac{1}{2^n}$ whenever i and j differ by exactly one bit. A plausible choice for canonical paths is to let γ_{xy} use **bit-fixing routing**, where we change bits in x to the corresponding bits in y from left to right. To compute congestion, pick some edge uv , and let k be the bit position in which u and v differ. A path γ_{xy} will cross uv if $u_k \dots u_n = x_k \dots x_n$ (because when we are at u we haven't fixed those bits yet) and $v_1 \dots v_k = y_1 \dots y_k$ (because at v we have fixed all of those bits). There are 2^{k-1} choices of $x_1 \dots x_{k-1}$ consistent with the first condition and 2^{n-k} choices of $y_{k+1} \dots y_n$ consistent with the second, giving exactly 2^{n-1} total paths γ_{xy} crossing uv . Since each path occurs with weight $\pi_x \pi_y = 2^{-2n}$, and the flow across uv is $\pi_u p_{uv} = 2^{-n} \frac{1}{2^n}$, we can calculate the congestion

$$\begin{aligned} \rho(\Gamma) &= \max_{uv \in E} \frac{1}{\pi_u p_{uv}} \sum_{\gamma_{xy} \ni uv} \pi_x \pi_y \\ &= \frac{1}{2^{-n}/2^n} \cdot 2^{n-1} \cdot 2^{-2n} \\ &= n. \end{aligned}$$

This gives a relaxation time $\tau_2 \leq 8\rho^2 = 8n^2$. In this case the bound is substantially worse than what we previously proved using coupling.

The fact that the number of canonical paths that cross a particular edge is exactly one half the number of nodes in the hypercube is not an accident: if we look at what information we need to fill in to compute x and y from u and v , we need (a) the part of x we've already gotten rid of, plus (b) the part of y we haven't filled in yet. If we stitch these two pieces together, we get all but one of the n bits we need to specify a node in the hypercube, the missing bit being the bit we flip to get from u to v . This sort of thing shows up a lot in conductance arguments where we build our canonical paths by fixing a structure one piece at a time.

9.5.8.3 Matchings in a graph

A **matching** in a graph $G = (V, E)$ is a subset of the edges with no two elements adjacent to each other; equivalently, it's a subgraph that includes all the vertices in which each vertex has degree at most 1. We can use a random walk to sample matchings from an arbitrary graph uniformly.

Here is the random walk. Let S_t be the matching at time t . At each step, we choose an edge $e \in E$ uniformly at random, and flip a coin to decide whether to include it or not. If the coin comes up tails, set $S_{t+1} = S_t \setminus \{e\}$ (this

may leave $S_{t+1} = S_t$ if S_t already omitted e); otherwise, set $S_{t+1} = S_t \cup \{e\}$ unless one of the endpoints of e is already incident to an edge in S_t , in which case set $S_{t+1} = S_t$.

Because this chain contains many self-loops, it's aperiodic. It's also straightforward to show that any transition between two adjacent matchings occurs with probability exactly $\frac{1}{2m}$, where $m = |E|$, and thus that the chain is reversible with a uniform stationary distribution. We'd like to bound the congestion of the chain to show that it converges in a reasonable amount of time.

Let N be the number of matchings in G , and let $n = |V|$ and $m = |E|$ as usual. Then $\pi_S = 1/N$ for all S and $\pi_S p_{ST} = \frac{1}{2Nm}$. Our congestion for any transition ST will then be $2NmN^{-2} = 2m/N$ times the number of paths that cross ST ; ideally this number of paths will be at most N times some small polynomial in n and/or m .

Suppose we are trying to get from some matching X to another matching Y . The graph $X \cup Y$ has maximum degree 2, so each of its connected components is either a path or a cycle; in addition, we know that the edges in each of these paths or cycles alternate whether they come from X or Y , which among other things implies that the cycles are all even cycles.

We can transform X to Y by processing these components in a methodical way: first order the components (say, by the increasing identity of the smallest vertex in each), then for each component replace the X edges with Y edges. If we do this cleverly enough, we can guarantee that for any transition ST , the set of edges $(X \cup Y) \setminus (S \cup T)$ always consists of a matching plus at most two extra edges, and that S , T , $(X \cup Y) \setminus (S \cup T)$ are enough to reconstruct X and Y . Since there are at most Nm^2 choices of $(X \cup Y) \setminus (S \cup T)$, this will give at most Nm^2 canonical paths across each transition.¹³

Here is how we do the replacement within a component. If C is a cycle with k vertices, order the vertices v_0, \dots, v_{k-1} such that v_0 has the smallest index in C and $v_0v_1 \in X$. Then $X \cap C$ consists of the even-numbered edges $e_{2i} = v_{2i}v_{2i+1}$ and $Y \cap C$ consists of the odd-numbered edges $e_{2i+1} = v_{2i+1}v_{2i+2}$, where we take $v_k = v_0$. We now delete e_0 , then alternate between deleting e_{2i} and adding e_{2i-1} for $i = 1 \dots k/2 - 1$. Finally we add e_{k-1} .

The number of edges in C at each step in this process is always one of $k/2$, $k/2 - 1$, or $k/2 - 2$, and since we always add or delete an edge, the number of edges in $C \cap (S \cup T)$ for any transition will be at least $k/2 - 1$, which means that the total degree of the vertices in $C \cap (S \cup T)$ is at least

¹³We could improve the constant a bit by using $N \binom{m}{2}$, but we won't.

$k - 2$. We also know that SUT is a matching, since it's either equal to S or equal to T . So there are at least $k - 2$ vertices in $C \cap (S \cup T)$ with degree 1, and none with degree 2, leaving at most 2 vertices with degree 0. In the complement $C \setminus (S \cup T)$, this becomes at most two vertices with degree 2, with the rest having degree 1. If these vertices are adjacent, removing the edge between them leaves a matching; if not, removing one edge adjacent to each does so. So two extra edges are enough.

A similar argument works for paths, which we will leave as an exercise for the reader.

Now suppose we know S , T , and $(XUY) \setminus (S \cup T)$. We can compute $X \cup Y = ((X \cup Y) \setminus (S \cup T)) \cup (S \cup T)$; this means we can reconstruct the graph $X \cup Y$, identify its components, and so on. We know which component C we are working on because we know which edge changes between S and T . For any earlier component C' , we have $C' \cap S = C' \cap Y$ (since we finished already), and similarly $C' \setminus S = C' \cap X$. For components C' we haven't reached yet, the reverse holds, $C' \cap S = C' \cap X$ and $C' \setminus S = C' \cap Y$. This determines the edges in both X and Y for all edges not in C .

For C , we know from $C \cap (X \cup Y)$ which vertex is v_0 . In a cycle, whenever we remove an edge, its lower-numbered endpoint is always an even distance from v_0 , and similarly when we add an edge, its lower-numbered endpoint is always an odd distance from v_0 . So we can orient the cycle and tell which edges in $S \cup T$ have already been added (and are thus part of Y) and which have been removed (and are thus part of X). Since we know that $C \cap Y = C \setminus X$, this is enough to reconstruct both $C \cap X$ and $C \cap Y$. (The same general idea works for paths, although for paths we do not need to be as careful to figure out orientation since we can only leave v_0 in one direction.)

The result is that given S and T , there are at most Nm^2 choices of X and Y such that the canonical path γ_{XY} crosses ST . This gives $\rho \leq \frac{Nm^2 N^{-2}}{N^{-1} \frac{1}{2m}} = 2m^3$, which gives $\tau_2 \leq 8m^6$. This is not great but it is at least polynomial in the size of the graph.¹⁴

¹⁴However, it might not give rapid mixing if N is small enough, which can occur in dense graphs. For example, on a clique we have $N = m + 1$, since no matching can contain more than one edge, making the bound $8m^6 = \Theta(N^6)$ which is polynomial in N but *not* polynomial in $\log N$. Polynomial in $\log N$ is what we want. The actual mixing time in this case is $\Theta(m)$ (for the upper bound, wait to delete the only edge, and do so on both sides of the coupling; for the lower bound, observe that until we delete the only edge we are still in our initial state). This is much better than the upper bound from canonical paths, but it still doesn't give rapid mixing.

9.5.8.4 Perfect matchings in dense bipartite graphs

(Basically doing [MR95, §11.3].)

A similar chain can be used to sample perfect matchings in dense bipartite graphs, as shown by Jerrum and Sinclair [JS89] based on an algorithm by Broder [Bro86] that turned out to have a bug in the analysis [Bro88].

A **perfect matching** of a graph G is a subgraph that includes all the vertices and gives each of them exactly one incident edge. We'll be looking for perfect matchings in a **bipartite graph** consisting of n left vertices u_1, \dots, u_n and n right vertices v_1, \dots, v_n , where every edge goes between a left vertex and a right vertex. We'll also assume that the graph is **dense**, which in this case means that every vertex has at least $n/2$ neighbors. This density assumption was used in the original Broder and Jerrum-Sinclair papers but removed in a later paper by Jerrum, Sinclair, and Vigoda [JSV04].

The random walk is similar to the random walk from the previous section restricted to the set of matchings with either $n - 1$ or n edges. At each step, we first flip a coin (the usual lazy walk trick); if it comes up heads, we choose an edge uv uniformly at random and apply one of the following transformations depending on how uv fits in the current matching m_t :

1. If $uv \in m_t$, and $|m_t| = n$, set $m_{t+1} = m_t \setminus \{uv\}$.
2. If u and v are both unmatched in m_t , and $|m_t| = n - 1$, set $m_{t+1} = m_t \cup \{uv\}$.
3. If exactly one of u and v is matched to some other node w , and $|m_t| = n - 1$, perform a rotation that deletes the w edge and adds uv .
4. If none of these conditions hold, set $m_{t+1} = m_t$.

The walk can be started from any perfect matching, which can be found in $O(n^{5/2})$ time using a classic algorithm of Hopcroft and Karp [HK73] that repeatedly searches for an **augmenting path**, which is a path in G between two unmatched vertices that alternates between edges not in the matching and edges in the matching. (We'll see augmenting paths again below when we show that any near-perfect matching can be turned into a perfect matching using at most two transitions.)

We can show that this walk converges in polynomial time using a canonical path argument. This is done in two stages: first, we define canonical paths between all perfect matchings. Next, we define a short path from any matching of size $n - 1$ to some nearby perfect matching, and build paths between arbitrary matchings by pasting one of these short paths on one or

both ends of a long path between perfect matchings. This gives a collection of canonical paths that we can show to have low congestion.

To go between perfect matchings X and Y , consider $X \cup Y$ as a collection of paths and even cycles as in §9.5.8.3. In some standard order, fix each path cycle by first deleting one edge to make room, then using rotate operations to move the rest of the edges, then putting the last edge back in. For any transition S – T along the way, we can use the same argument that we can compute $X \cap Y$ from $S \cap T$ by supplying the missing edges, which will consist of a matching of size n plus at most one extra edge. So if N is the number of matchings of size n or $n - 1$, the same argument used previously shows that at most Nm of these long paths cross any S – T transition.

For the short paths, we must use the density property. The idea is that for any matching that is not perfect, we can find an augmenting path of length at most 3 between two unmatched nodes on either side. Pick some unmatched nodes u and v . Each of these nodes is adjacent to at least $n/2$ neighbors; if any of these neighbors are unmatched, we just found an augmenting path of length 1. Otherwise the $n/2$ neighbors of u and the $n/2$ nodes matched to neighbors of v overlap (because v is unmatched, leaving at most $n - 1$ matched nodes and thus at most $n/2 - 1$ nodes that are matched to something that's not a neighbor of v). So for each matching of size $n - 1$, in at most two steps (rotate and then add an edge) we can reach some specific perfect matching. There are at most m^2 ways that we can undo this, so each perfect matching is associated with at most m^2 smaller matchings. This blows up the number of canonical paths crossing any transition by roughly m^4 ; by counting carefully we can thus show congestion that is $O(m^6)$ ($O(m^4)$ from the blow-up, m from the m in Nm , and m from $1/p_{ST}$).

It follows that for this process, $\tau_2 = O(m^{12})$. (I think a better analysis is possible.)

As noted earlier, this is an example of a process for which causal coupling doesn't work in less than exponential time [KR99], a common problem with Markov chains that don't have much symmetry. So it's not surprising that stronger techniques were developed specifically to attack this problem.

Chapter 10

Approximate counting

Last updated 2019. Some material may be out of date.

(See also [MR95, Chapter 11].)

The basic idea: we have some class of objects, and we want to know how many of them there are. Ideally we can build an algorithm that just prints out the exact number, but for many problems this is hard.

A **fully polynomial-time randomized approximation scheme** or **FPRAS** for a numerical problem outputs a number that is between $(1 - \epsilon)$ and $(1 + \epsilon)$ times the correct answer, with probability at least $3/4$ (or some constant bounded away from $1/2$ —we can amplify to improve it), in time polynomial in the input size n and $(1/\epsilon)$. In this chapter, we'll be hunting for FPRASs. But first we will discuss briefly why we can't just count directly in many cases.

10.1 Exact counting

A typical application is to a problem in the complexity class $\#\mathbf{P}$, problems that involve counting the number of accepting computation paths in a nondeterministic polynomial-time Turing machine. Equivalently, these are problems that involve counting for some input x the number of values r such that $M(x, r) = 1$, where M is some machine in \mathbf{P} . An example would be the problem $\#\mathbf{SAT}$ of counting the number of satisfying assignments of some CNF formula.

The class $\#\mathbf{P}$ (which is usually pronounced **sharp P** or **number P**) was defined by Leslie Valiant in a classic paper [Val79]. The central result in this paper is **Valiant's theorem**. This shows that any problem in $\#\mathbf{P}$

can be **reduced** (by **Cook reductions**, meaning that we are allowed to use the target problem as a subroutine instead of just calling it once) to the problem of computing the **permanent** of a square 0–1 matrix A , where the permanent is given by the formula $\sum_{\pi} \prod_i A_{i,\pi(i)}$, where the sum ranges over all $n!$ permutations π of the indices of the matrix. An equivalent problem is counting the number of **perfect matchings** (subgraphs including all vertices in which every vertex has degree exactly 1) of a bipartite graph. Other examples of **#P**-complete problems are **#SAT** (defined above) and **#DNF** (like **#SAT**, but the input is in DNF form; **#SAT** reduces to **#DNF** by negating the formula and then subtracting the result from 2^n).

Exact counting of **#P**-hard problems is likely to be very difficult: **Toda’s theorem** [Tod91] says that being able to make even a single query to a **#P**-oracle is enough to solve any problem in the **polynomial-time hierarchy**, which contains most of the complexity classes you have probably heard of. Nonetheless, it is often possible to obtain good approximations to such problems.

10.2 Counting by sampling

If many of the things we are looking for are in the target set, we can count by sampling; this is what poll-takers do for a living. Let U be the universe we are sampling from and G be the “good” set of points we want to count. Let $\rho = |G|/|U|$. If we can sample uniformly from U , then we can estimate ρ by taking N independent samples and dividing the number of samples in G by N . This will give us an answer $\hat{\rho}$ whose expectation is ρ , but the accuracy may be off. Since the variance of each sample is $\rho(1 - \rho) \approx \rho$ (when ρ is small), we get $\text{Var}[\sum X_i] \approx m\rho$, giving a standard deviation of $\sqrt{m\rho}$. For this to be less than our allowable error $\epsilon m\rho$, we need $1 \leq \epsilon\sqrt{m\rho}$ or $N \geq \frac{1}{\epsilon^2\rho}$. This gets bad if ρ is exponentially small. So if we are to use sampling, we need to make sure that we only use it when ρ is large.

On the positive side, if ρ is large enough we can easily compute how many samples we need using Chernoff bounds. The following lemma gives a convenient estimate; it is based on [MR95, Theorem 11.1] with a slight improvement on the constant:

Lemma 10.2.1. *Sampling N times gives relative error ϵ with probability at least $1 - \delta$ provided $\epsilon \leq 1.81$ and*

$$N \geq \frac{3}{\epsilon^2\rho} \ln \frac{2}{\delta}. \quad (10.2.1)$$

Proof. Suppose we take N samples, and let X be the total count for these samples. Then $E[X] = \rho N$, and (5.2.7) gives (for $\epsilon \leq 1.81$):

$$\Pr[|X - \rho N| \geq \epsilon \rho N] \leq 2e^{-\rho N \epsilon^2/3}.$$

Now set $2e^{-\rho N \epsilon^2/3} \leq \delta$ and solve for N to get (10.2.1). \square

10.2.1 Generating samples

An issue that sometimes comes up in this process is that it is not always obvious how to generate a sample from a given universe U . What we want is a function that takes random bits as input and produces an element of U as output.

In the simplest case, we know $|U|$ and have a function f from $\{0, \dots, |U| - 1\}$ to U (the application of such a function is called **unranking**). Here we can do **rejection sampling**: we choose a bit-vector of length $\lceil \lg|U| \rceil$, interpret it as an integer x expressed in binary, and try again until $x < |U|$. This requires less than $2\lceil \lg|U| \rceil$ bits on average in the worst case, since we get a good value at least half the time.

Unranking (and the converse operation of ranking) can be a non-trivial problem in combinatorics, and there are entire textbooks on the subject [SW13]. But some general principles apply. If U is a union of disjoint sets $U_1 \cup U_2 \cup \dots \cup U_n$, we can compute the sizes of each $|U_i|$, and we can unrank each U_i , then we can map some $x \in \{0 \dots |U| - 1\}$ to a specific element of U by choosing the maximum i such that $\sum_{j=1}^{i-1} |U_j| \leq x$ and choosing the k -th element of U_i , where $k = x - \sum_{j=1}^{i-1} |U_j|$. This is essentially what happens if we want to compute the x -th day of the year. The first step of this process requires computing the sizes of $O(n)$ sets if we generate the sum one element at a time, although if we have a fast way of computing $\sum_{i=1}^j |U_i|$ for each i , we can cut this time to $O(\log n)$ such computations down using binary search. The second step depends on whatever mechanism we use to unrank within U_i .

An example would be generating one of the $\binom{n}{k}$ subsets of a set of size k uniformly at random. If we let S be the set of all such k -subsets, we can express S as $\bigcup_{i=1}^n S_i$ where i is the set of all k -subsets that have the i -th element as their smallest element in some fixed ordering. But then we can easily compute $|S_i| = \binom{n-i}{k-1}$ for each i , apply the above technique to pick a particular S_i to start with, and then recurse within S_i to get the rest of the elements.

A more general case is when we can't easily sample from U but we can sample from some $T \supseteq U$. Here rejection sampling comes to our rescue

as long as $|U|/|T|$ is large enough. For example, if we want to generate a k -subset of n elements when $k \ll \sqrt{n}$, we can choose a list of k elements with replacement and discard any lists that contain duplicates. But this doesn't work so well for larger k . In this particular case, there is an easy out: we can sample k elements without replacement and forget their order. This corresponds to sampling from a universe T of ordered k -subsets, then mapping down to U . As long as the inverse image of each element of U has the same size in T , this will give us a uniform sample from U .

When rejection sampling fails, we may need to come up with a more clever approach to concentrate on the particular values we want. One approach that can work well if we can express U as a union of sets that are not necessarily disjoint is the Karp-Luby technique [KL85], discussed in §10.3 below.

All of this assumes that we are interested in getting uniform samples. For non-uniform samples, we replace the assumption that we are trying to generate a uniform $X \in \{0 \dots m-1\}$ with some $X \in \{0 \dots m-1\}$ for which we can efficiently calculate $\Pr[X \leq i]$ for each i . In this case, we can (in principle) generate a continuous random variable $Y \in [0, 1]$ and choose the maximum i such that $\Pr[X \leq i] \leq Y$, which we can find using binary search.

The complication is that we can't actually generate Y in finite time. Instead, we generate Y one bit at a time; this gives a sequence of rational values Y_t where each Y_t is of the form $k/2^t$. We can stop when every value in interval $[Y_t, Y_t + 2^{-t})$ lies with the range corresponding to some particular value of X . This technique is closely related to **arithmetic coding** [WNC87], and requires generating $H(X) + O(1)$ bits on average, where $H(X) = -\sum_{i=0}^{m-1} \Pr[X = i] \lg \Pr[X = i]$ is the (base 2) **entropy** of X .

10.3 Approximating #DNF

A classical algorithm of Karp and Luby [KL85] gives a FPRAS for #DNF. We'll describe how this works, mostly following the presentation in [MR95, §11.2]. The key idea of the Karp-Luby technique is to express the set S whose size we want to know, as a union of a polynomial number of simpler sets S_1, \dots, S_k . If for each i we can sample uniformly from S_i , compute $|S_i|$, and determine membership in S_i , then some clever sampling will let us approximate $|S|$ by sampling from a *disjoint* union of the S_i to determine how much $\sum |S_i|$ overestimates $|S|$.

To make this concrete, let's look at the specific problem studied by Karp and Luby of approximating the number of satisfying assignment to a DNF

formula. A **DNF formula** is a formula that is in **disjunctive normal form**: it is an OR of zero or more **clauses**, each of which is an AND of variables or their negations. An example would be $(x_1 \wedge x_2 \wedge x_3) \vee (\neg x_1 \wedge x_4) \vee x_2$. The **#DNF** problem is to count the number of satisfying assignments of a formula presented in disjunctive normal form.

Solving #DNF exactly is #P-complete, so we don't expect to be able to do it. Instead, we'll get a FPRAS by cleverly sampling solutions. The need for cleverness arises because just sampling solutions directly by generating one of the 2^n possible assignments to the n variables may find no satisfying assignments at all, since the size of any individual clause might be big enough that getting a satisfying assignment for that clause at random is exponentially unlikely.

So instead we will sample pairs (x, i) , where x is an assignment that satisfies clause C_i ; these are easier to find, because if we know which clause C_i we are trying to satisfy, we can read off the satisfying assignment from its variables. Let S be the set of such pairs. For each pair (x, i) , define $f(x, i) = 1$ if and only if $C_j(x) = 0$ for all $j < i$. Then $\sum_{(x,i) \in S} f(x, i)$ counts every satisfying assignment x , because (a) there exists some i such that x satisfies C_i , and (b) only the smallest such i will have $f(x, i) = 1$. In effect, f is picking out a single canonical satisfied clause from each satisfying assignment. Note that we can compute f efficiently by testing x against all clauses C_j with $j < i$.

Our goal is to estimate the proportion ρ of "good" pairs with $f(x, i) = 1$ out of all pairs in S , and then use this to estimate $\sum_{(x,i) \in S} f(x, i) = \rho|S|$. If we can sample from S uniformly, the proportion ρ of "good" pairs with $f(x, i) = 1$ is at least $1/m$, because every satisfying assignment x contributes at most m pairs total, and one of them is good.

The only tricky part is figuring out how to sample pairs (x, i) with $C_i(x) = 1$ so that all pairs occur with the same probability. Let $U_i = \{(x, i) \mid C_i(x) = 1\}$. Then we can compute $|U_i| = 2^{n-k_i}$ where k_i is the number of literals in C_i . Using this information, we can sample i first with probability $|U_i| / \sum_j |U_j|$, then sample x from U_i just by picking independent uniform random values for the $n - k$ variables not fixed by C_i .

With $N \geq \frac{4}{\epsilon^2(1/m)} \ln \frac{2}{\delta} = \frac{4m}{\epsilon^2} \ln \frac{2}{\delta}$, we obtain an estimate $\hat{\rho}$ for the proportion of pairs (x, i) with $f(x, i) = 1$ that is within ϵ relative error of ρ with probability at least $1 - \delta$. Multiplying this by $\sum |U_i|$ then gives the desired count of satisfying assignments.

It's worth noting that there's nothing special about DNF formulas in this method. Essentially the same trick will work for estimating the size of

the union of any collection of sets U_i where we can (a) compute the size of each U_i ; (b) sample from each U_i individually; and (c) test membership of our sample x in U_j for $j < i$.

10.4 Approximating #KNAPSACK

Here is an algorithm for approximating the number of solutions to a **KNAPSACK** problem, due to Dyer [Dye03]. We'll concentrate on the simplest version, 0–1 KNAPSACK, following the analysis in Section 2.1 of [Dye03].

For the 0–1 KNAPSACK problem, we are given a set of n objects of weight $0 \leq a_1 \leq a_2 \leq \dots \leq a_n \leq b$, and we want to find a 0–1 assignment x_1, x_2, \dots, x_n such that $\sum_{i=1}^n a_i x_i \leq b$ (usually while optimizing some property that prevents us from setting all the x_i to zero). We'll assume that the a_i and b are all integers.

For #KNAPSACK, we want to compute $|S|$, where S is the set of all assignments to the x_i that make $\sum_{i=1}^n a_i x_i \leq b$.

There is a well-known **fully polynomial-time approximation scheme** for optimizing KNAPSACK, based on dynamic programming. The idea is that a maximum-weight solution can be found exactly in time polynomial in b , and if b is too large, we can reduce it by rescaling all the a_i and b at the cost of a small amount of error. A similar idea is used in Dyer's algorithm: the KNAPSACK problem is rescaled so that size of the solution set S' of the rescaled version can be computed in polynomial time. Sampling is then used to determine what proportion of the solutions in S' correspond to solutions of the original problem.

Scaling step: Let $a'_i = \lfloor n^2 a_i / b \rfloor$. Then $0 \leq a'_i \leq n^2$ for all i . Taking the floor creates some error: if we try to reconstruct a_i from a'_i , the best we can do is argue that $a'_i \leq n^2 a_i / b < a'_i + 1$ implies $(b/n^2)a'_i \leq a_i < (b/n^2)a'_i + (b/n^2)$. The reason for using n^2 as our rescaled bound is that the total error in the upper bound on a_i , summed over all i , is bounded by $n(b/n^2) = b/n$, a fact that will become important soon.

Let $S' = \{\vec{x} \mid \sum_{i=1}^n a'_i x_i \leq n^2\}$ be the set of solutions to the rescaled knapsack problem, where we substitute a'_i for a_i and $n^2 = (n^2/b)b$ for b .

Claim: $S \subseteq S'$. Proof: $\vec{x} \in S$ if and only if $\sum_{i=1}^n a_i x_i \leq b$. But then

$$\begin{aligned} \sum_{i=1}^n a'_i x_i &= \sum_{i=1}^n \lfloor n^2 a_i / b \rfloor x_i \\ &\leq \sum_{i=1}^n (n^2 / b) a_i x_i \\ &= (n^2 / b) \sum_{i=1}^n a_i x_i \\ &\leq n^2, \end{aligned}$$

which shows $\vec{x} \in S'$.

The converse does not hold. However, we can argue that any $\vec{x} \in S'$ can be shoehorned into S by setting at most one of the x_i to 0. Consider the set of all positions i such that $x_i = 1$ and $a_i > b/n$. If this set is empty, then $\sum_{i=1}^n a_i x_i \leq \sum_{i=1}^n b/n = b$, and \vec{x} is already in S . Otherwise, pick any position i with $x_i = 1$ and $a_i > b/n$, and let $y_j = 0$ when $j = i$ and $y_j = x_j$ otherwise. If we recall that the total error from the floors in our approximation was at most $(b/n^2)n = b/n$, the intuition is that deleting y_j removes it. Formally, we can write

$$\begin{aligned} \sum_{j=1}^n a_j y_j &= \sum_{j=1}^n a_j x_j - a_i \\ &< \sum_{j=1}^n ((b/n^2) a'_j + b/n^2) x_j - b/n \\ &\leq (b/n^2) \sum_{j=1}^n a'_j x_j + b/n - b/n \\ &\leq (b/n^2) n^2 \\ &= b. \end{aligned}$$

Applying this mapping to all elements \vec{x} of S maps at most $n + 1$ of them to each \vec{y} in S' ; it follows that $|S'| \leq (n + 1)|S|$, which means that if we can sample elements of S' uniformly, each sample will hit S with probability at least $1/(n + 1)$.

To compute $|S'|$, let $C(k, m) = \left| \left\{ \vec{x} \mid \sum_{i=1}^k a'_i x_i \leq m \right\} \right|$ be the number of subsets of $\{a'_1, \dots, a'_k\}$ that sum to m or less. Then $C(k, m)$ satisfies the

recurrence

$$\begin{aligned} C(k, m) &= C(k-1, m - a'_k) + C(k-1, m) \\ C(0, m) &= 1 \end{aligned}$$

where k ranges from 0 to n and m ranges from 0 to n^2 , and we treat $C(k-1, m - a'_k) = 0$ if $m - a'_k < 0$. The idea is that $C(k-1, m - a'_k)$ counts all the ways to make m if we include a'_k , and $C(k-1, m)$ counts all the ways to make m if we exclude it. The base case corresponds to the empty set (which sums to $\leq m$ no matter what m is).

We can compute a table of all values of $C(k, m)$ by iterating through m in increasing order; this takes $O(n^3)$ time. At the end of this process, we can read off $|S'| = C(n, n^2)$.

But we can do more than this: we can also use the table of counts to sample uniformly from S' . The probability that $x'_n = 1$ for a uniform random element of S' is exactly $C(n-1, n^2 - a'_n) / C(n, n^2)$; having chosen $x'_n = 1$ (say), the probability that $x'_{n-1} = 1$ is then $C(n-2, n^2 - a'_n - a'_{n-1}) / C(n-2, n^2 - a'_n)$, and so on. So after making $O(n)$ random choices (with $O(1)$ arithmetic operations for each choice to compute the probabilities) we get a uniform element of S' , which we can test for membership in S in an additional $O(n)$ operations.

We've already established that $|S|/|S'| \geq 1/(n+1)$, so we can apply Lemma 10.2.1 to get ϵ relative error with probability at least $1 - \delta$ using $\frac{3(n+1)}{\epsilon^2} \ln \frac{2}{\delta}$ samples. This gives a cost of $O(n^2 \log(1/\delta)/\epsilon^2)$ for the sampling step, or a total cost of $O(n^3 + n^2 \log(1/\delta)/\epsilon^2)$ after including the cost of building the table (in practice, the second term will dominate unless we are willing to accept $\epsilon = \omega(1/\sqrt{n})$).

It is possible to improve this bound. Dyer [Dye03] shows that using **randomized rounding** on the a'_i instead of just truncating them gives a FPRAS that runs in $O(n^{5/2} \sqrt{\log(1/\epsilon)} + n^2/\epsilon^2)$ time.

10.5 Approximating exponentially improbable events

For #DNF and #KNAPSACK, we saw how restricting our sampling to a cleverly chosen sample space could boost the hit ratio ρ to something that gave a reasonable number of samples using Lemma 10.2.1. For other problems, it is often not clear how to do this directly, and the best sample spaces we can come up with make our target points an exponentially small fraction of the whole.

In these cases, it is sometimes possible to approximate this exponentially small fraction as a product of many more reasonable ratios. The idea is to express our target set as the last of a sequence of sets S_0, S_1, \dots, S_k , where we can compute the size of S_0 and can estimate $|S_{i+1}|/|S_i|$ accurately for each i . This gives $|S_k| = |S_0| \cdot \prod_{i=1}^k \frac{|S_{i+1}|}{|S_i|}$, with a relative error that grows roughly linearly with k . Specifically, if we can approximate each $|S_{i+1}|/|S_i|$ ratio to between $1 - \epsilon$ and $1 + \epsilon$ of the correct value, then the product of these ratios will be between $(1 - \epsilon)^k$ and $(1 + \epsilon)^k$ of the correct value; these bounds approach $1 - \epsilon k$ and $1 + \epsilon k$ in the limit as ϵk goes to zero, using the binomial theorem, although to get a real bound we will need to do more careful error analysis.

10.5.1 Matchings

We saw in §9.5.8.3 that a random walk on matchings on a graph with m edges has mixing time $\tau_2 \leq 8m^6$, where the walk is defined by selecting an edge uniformly at random and flipping whether it is in the matching or not, while rejecting any steps that produce a non-matching. This allows us to sample matchings of a graph with δ total variation distance from the uniform distribution in $O\left(m^6\left(\log m + \log \frac{1}{\delta}\right)\right)$ time.

Suppose now that we want to count matchings instead of sampling them. It's easy to show that for any particular edge $uv \in G$, at least half of all matchings in G don't include uv : the reason is that if M is a matching in G , then $M' = M \setminus \{uv\}$ is also a matching, and at most two matchings M' and $M' \cup \{uv\}$ are mapped to any one M' by this mapping.

Order the edges of G arbitrarily as e_1, e_2, \dots, e_m . Let S_i be the set of matchings in $G \setminus \{e_1 \dots e_i\}$. Then S_0 is the set of all matchings, and we've just argued that $\rho_{i+1} = |S_{i+1}|/|S_i| \geq 1/2$. We also know that $|S_m|$ counts the number of matchings in a graph with no edges, so it's exactly one. So we can use the product-of-ratios trick to compute $S_0 = \prod_{i=1}^m \frac{|S_i|}{|S_{i+1}|}$.

A random walk of length $O\left(m^6\left(\log m + \log \frac{1}{\eta}\right)\right)$ can sample matchings from S_i with a probability ρ' of getting a matching in S_{i+1} that is between $(1 - \eta)\rho_{i+1}$ and $(1 + \eta)\rho_{i+1}$. From Lemma 10.2.1, we can estimate ρ' within relative error γ with probability at least $1 - \zeta$ using $O\left(\frac{1}{\gamma^2 \rho'} \log \frac{1}{\zeta}\right) = O\left(\frac{1}{\gamma} \log \frac{1}{\zeta}\right)$ samples. Combined with the error on ρ' , this gives relative error at most $\gamma + \eta + \gamma\eta$ in $O\left(m^6\left(\log m + \log \frac{1}{\eta}\right) \log \frac{1}{\gamma} \log \frac{1}{\zeta}\right)$ operations.¹ If we then

¹This is the point where sensible people start hauling out the \tilde{O} notation, where a function is $\tilde{O}(f(n))$ if it is $O(f(n)g)$ where g is polylogarithmic in n and any other parameters

multiply out all the estimates for $|S_i|/|S_{i+1}|$, we get an estimate of S_0 that is at most $(1 + \gamma + \eta + \gamma\eta)^m$ times the correct value with probability at least $1 - m\zeta$ (with a similar bound on the other side), in total time $O\left(m^7 \left(\log m \log \frac{1}{\eta}\right) \log \frac{1}{\gamma} \log \frac{1}{\zeta}\right)$.

To turn this into a fully polynomial-time approximation scheme, given ϵ , δ , and m , we need to select η , γ , and ζ to get relative error ϵ with probability at least $1 - \delta$. Letting $\zeta = \delta/m$ gets the δ part. For ϵ , we need $(1 + \gamma + \eta + \gamma\eta)^m \leq 1 + \epsilon$. Suppose that $\epsilon < 1$ and let $\gamma = \eta = \epsilon/6m$. Then

$$\begin{aligned} (1 + \gamma + \eta + \gamma\eta)^m &\leq \left(1 + \frac{\epsilon}{2m}\right)^m \\ &\leq e^{\epsilon/2} \\ &\leq 1 + \epsilon. \end{aligned}$$

Plugging these values into our cost formula gives $O(m^7)$ times a bunch of factors that are polynomial in $\log m$ and $\frac{1}{\epsilon}$, which we can abbreviate as $\tilde{O}(m^7)$.

10.5.2 Other problems

Similar methods work for other problems that self-reduce by restricting particular features of the solution. Examples include colorings (fix the color of some vertex), independent sets (remove a vertex), and approximating the volume of a convex body (take the intersection with a sphere of appropriate radius; see [MR95, §11.4] for a slightly less sketchy description). We will not go into details on any of these applications here.

that may be running around $(\frac{1}{\epsilon}, \frac{1}{\eta}, \text{etc.})$.

Chapter 11

The probabilistic method

Last updated 2019. Some material may be out of date.

The **probabilistic method** is a tool for proving the existence of objects with particular combinatorial properties, by showing that some process generates these objects with nonzero probability.

The relevance of this to randomized algorithms is that in some cases we can make the probability large enough that we can actually produce such objects.

We'll mostly be following Chapter 5 of [\[MR95\]](#) with some updates for more recent results. If you'd like to read more about these techniques, a classic reference on the probabilistic method in combinatorics is the text of Alon and Spencer [\[AS92\]](#).

11.1 Randomized constructions and existence proofs

Suppose we want to show that some object exists, but we don't know how to construct it explicitly. One way to do this is to devise some random process for generating objects, and show that the probability that it generates the object we want is greater than zero. This implies that something we want exists, because otherwise it would be impossible to generate; and it works even if the nonzero probability is very, very small. The systematic development of the method is generally attributed to the notoriously productive mathematician Paul Erdős and his frequent collaborator Alfréd Rényi.

From an algorithmic perspective, the probabilistic method is useful mainly when we can make the nonzero probability substantially larger than zero—and especially if we can recognize when we've won. But sometimes just demonstrating the existence of an object is a start.

We give a couple of examples of the probabilistic method in action below. In each case, the probability that we get a good outcome is actually pretty high, so we could in principle generate a good outcome by retrying our random process until it works. There are some more complicated examples of the method for which this doesn't work, either because the probability of success is vanishingly small, or because we can't efficiently test whether what we did succeeded (the last example below may fall into this category). This means that we often end up with objects whose existence we can demonstrate even though we can't actually point to any examples of them. For example, it is known that there exist **sorting networks** (a special class of circuits for sorting numbers in parallel) that sort in time $O(\log n)$, where n is the number of values being sorted [AKS83]; these can be generated randomly with nonzero probability. But the best explicit constructions of such networks take time $\Theta(\log^2 n)$, and the question of how to find an *explicit* network that achieves $O(\log n)$ time has been open for decades despite many efforts to solve it.

11.1.1 Unique hats

A collection of n robots each wishes to own a unique hat. Unfortunately, the State Ministry for Hat Production only supplies one kind of hat. A robot will only be happy if (a) it has a hat, and (b) no robot it sees has a hat. Fortunately each robot can only see k other robots. How many robots can be happy?

We could try to be clever about answering this question, or we could apply the probabilistic method. Suppose we give each robot a hat with independent probability p . Then the probability that any particular robot r is happy is pq^k , where $q = 1 - p$ is the probability that a robot doesn't have a hat and q^k gives the probability that none of the robots that r sees has a hat. If we let X_r be the indicator for the event that r is happy, we have $E[X_r] = pq^k$ and the expected number of happy robots is $E[\sum X_r] = \sum E[X_r] = npq^k$. Since we can achieve this value on average, there must be some specific assignment that achieves it as well.

To choose a good p , we apply the usual calculus trick of looking for a maximum by looking for the place where the derivative is zero. We have $\frac{d}{dp} npq^k = nq^k - nkpq^{k-1}$ (since $\frac{dq}{dp} = -1$), so we get $nq^k - nkpq^{k-1} = 0$. Factoring out n and q^{k-1} gives $q - pk = 0$ or $1 - p - pk = 0$ or $p = 1/(k+1)$. For this value of p , the expected number of happy robots is exactly $n \left(\frac{1}{k+1}\right) \left(1 - \frac{1}{k+1}\right)^k$. For large k the last factor approaches $1/e$, giving us

approximately $(1/e)^{\frac{n}{k+1}}$ happy robots.

Up to constant factors this is about as good an outcome as we can hope for: we can set things up so that our robots are arranged in groups of $k+1$, where each robot sees all the other robots in its group, and here we can clearly only have 1 happy robot per group, for a maximum of $n/(k+1)$ happy robots.

Note that we can improve the constant $1/e$ slightly by being smarter than just handing out hats at random. Starting with our original n robots, look for a robot that is observed by the smallest number of other robots. Since there are only nk pairs of robots (r_1, r_2) where r_1 sees r_2 , one of the n robots is only seen by at most k other robots. Now give this robot a hat, and give no hat to any robot that sees it or that it sees. This produces 1 happy robot while knocking out at most $2k+1$ robots total. Now remove these robots from consideration and repeat the analysis on the remaining robots, to get another happy robot while again knocking out at most $2k+1$ robots. Iterating this procedure until no robots are left gives at least $n/(2k+1)$ happy robots; this is close to $(1/2)^{\frac{n}{k+1}}$ for large k , which is a bit better than $(1/e)^{\frac{n}{k+1}}$. (It may be possible to improve the constant further with more analysis.) This shows that the probabilistic method doesn't necessarily produce better results than we can get by being smart. But the hope is that with the probabilistic method we don't have to be smart, and for some problems it seems that solving them without using the probabilistic method requires being exceptionally smart.

11.1.2 Ramsey numbers

Consider a collection of n schoolchildren, and imagine that each pair of schoolchildren either like each other or dislike each other. We assume that these preferences are symmetric: if x likes y , then y likes x , and similarly if x dislikes y , y dislikes x . Let $R(k, h)$ be the smallest value for n that ensures that among any group of n schoolchildren, there is either a subset of k children that all like each other or a subset of h children that all dislike each other.¹

It is not hard to show that $R(k, h)$ is finite for all k and h .² The exact

¹In terms of graphs, any graph G with at least $R(k, h)$ nodes contains either a **clique** of size k or an **independent set** of size h .

²A simple proof, due to Erdős and Szekeres [ES35], proceeds by showing that $R(k, h) \leq R(k-1, h) + R(k, h-1)$. Given a graph G of size at least $R(k-1, h) + R(k, h-1)$, choose a vertex v and partition the graph into two induced subgraphs G_1 and G_2 , where G_1 contains all the vertices adjacent to v and G_2 contains all the vertices not adjacent to v .

value of $R(k, h)$ is known only for small values of k and h .³ But we can use the probabilistic method to show that for $k = h$, it is reasonably large. The following theorem is due to Erdős, and was the first known lower bound on $R(k, k)$.

Theorem 11.1.1 ([Erd47]). *If $k \geq 3$, then $R(k, k) > 2^{k/2}$.*

Proof. Suppose each pair of schoolchildren flip a fair coin to decide whether they like each other or not. Then the probability that any particular set of k schoolchildren all like each other is $2^{-\binom{k}{2}}$ and the probability that they all dislike each other is the same. Summing over both possibilities and all subsets gives a bound of $\binom{n}{k} 2^{1-\binom{k}{2}}$ on the probability that there is at least one subset in which everybody likes everybody or everybody dislikes everybody. For $n = 2^{k/2}$, we have

$$\begin{aligned} \binom{n}{k} 2^{1-\binom{k}{2}} &\leq \frac{n^k}{k!} 2^{1-\binom{k}{2}} \\ &= \frac{2^{k^2/2+1-k(k-1)/2}}{k!} \\ &= \frac{2^{k^2/2+1-k^2/2+k/2}}{k!} \\ &= \frac{2^{1+k/2}}{k!} \\ &< 1. \end{aligned}$$

Because the probability that there is an all-liking or all-hating subset is less than 1, there must be some chance that we get a collection that doesn't have one. So such a collection exists. It follows that $R(k, k) > 2^{k/2}$,

Either $|G_1| \geq R(k-1, h)$ or $|G_2| \geq R(k, h-1)$. If $|G_1| \geq R(k-1, h)$, then G_1 contains either a clique of size $k-1$ (which makes a clique of size k in G when we add v to it) or an independent set of size h (which is also in G). Alternatively, if $|G_2| \geq R(k, h-1)$, then G_2 either gives us a clique of size k by itself or an independent set of size h after adding v . Together with the fact that $R(1, h) = R(k, 1) = 1$, this recurrence gives $R(k, h) \leq \binom{(k-1)+(h-1)}{k-1}$.

³There is a fairly current table at http://en.wikipedia.org/wiki/Ramsey's_Theorem. Some noteworthy values are $R(3, 3) = 6$, $R(4, 4) = 18$, and $43 \leq R(5, 5) \leq 49$. One problem with computing exact values is that as $R(k, h)$ grows, the number of graphs one needs to consider gets very big. There are $2^{\binom{n}{2}}$ graphs with n vertices, and even detecting the presence or absence of a moderately-large clique or independent set in such a graph can be expensive. This pretty much rules out any sort of brute-force approach based on simply enumerating candidate graphs.

because we have shown that not all collections at $n = 2^{k/2}$ have the Ramsey property. \square

The last step in the proof uses the fact that $2^{1+k/2} < k!$ for $k \geq 3$, which can be tested explicitly for $k = 3$ and proved by induction for larger k . The resulting bound is a little bit weaker than just saying that n must be large enough that $\binom{n}{k} 2^{1-\binom{k}{2}} \geq 1$, but it's easier to use.

The proof can be generalized to the case where $k \neq h$ by tweaking the bounds and probabilities appropriately. Note that even though this process generates a graph with no large cliques or independent sets with reasonably high probability, we don't have any good way of testing the result, since testing for the existence of a clique is **NP**-hard.

11.1.3 Directed cycles in tournaments

In the previous example we used the probabilistic method to show that there existed a structure that didn't contain some particular substructure. For this example we will do the reverse: show that there exists a structure that contains many instances of a particular substructure.

Imagine that n wrestlers wrestle in a round-robin tournament, so that ever wrestler wrestles every other wrestler exactly once, and so that for each pair of wrestlers one wrestler beats the other. Consider a cycle of wrestlers x_1, x_2, \dots, x_n such that each wrestler x_i beats x_{i+1} and x_n beats x_1 . (Note that we consider two cycles equivalent if one is a cyclic shift of the other.)

Theorem 11.1.2. *For $n \geq 3$, there exists a tournament with at least $2^{-n}(n-1)!$ directed cycles.*

Proof. Technical detail: We need $n \geq 3$ because the edges in a 2-cycle always go in opposite directions.

Flip a coin to decide the outcome of each pairing. For any particular sequence x_1, \dots, x_n , the probability that it is a directed cycle is then 2^{-n} (since there are n edges in the sequence and each has independent probability 2^{-1} of going the right way). Now let σ range over all $(n-1)!$ cycles and define X_σ as the indicator variable for the event that each edge is directed the right way. We've just shown that $E[X_\sigma] = 2^{-n}$; so the expected total number of cycles is $E[\sum_\sigma X_\sigma] = \sum_\sigma E[X_\sigma] = 2^{-n}(n-1)!$. But if the average value is at least $2^{-n}(n-1)!$, there must be some specific outcome that gives a value at least this high. \square

As in the Ramsey graph example, Theorem 11.1.2 says that generating a random tournament is likely to give us many directed cycles, but whether a

particular tournament has this property may be hard to test. This is not unusual for probabilistic-method arguments.

11.2 Approximation algorithms

One use of a randomized construction is to approximate the solution to an otherwise difficult problem. In this section, we start with a trivial approximation algorithm for the largest cut in a graph, and then show a more powerful randomized approximation algorithm, due to Goemans and Williamson [GW94], that gets a better approximation ratio for a much larger class of problems.

11.2.1 MAX CUT

We've previously seen (§2.3.3.2) a randomized algorithm for finding small cuts in a graph. What if we want to find a large cut?

Here is a particularly brainless algorithm that finds a large cut. For each vertex, flip a coin: if the coin comes up heads, put the vertex in S , otherwise, put in it T . For each edge, there is a probability of exactly $1/2$ that it is included in the S - T cut. It follows that the expected size of the cut is exactly $m/2$.

One consequence of this is that every graph has a global cut that includes at least half the edges. Another is that this algorithm finds such a cut, with probability at least $\frac{1}{m+1}$. To prove this, let X be the random variable representing the number of edges in the cut, and let p be the probability that $X \geq m/2$. Then

$$\begin{aligned} m/2 &= \mathbf{E}[X] \\ &= (1-p)\mathbf{E}[X \mid X < m/2] + p\mathbf{E}[X \mid X \geq m/2] \\ &\leq (1-p)\frac{m-1}{2} + pm. \end{aligned}$$

Solving this for p gives the claimed bound.⁴

By running this enough times to get a good cut, we get a polynomial-time randomized algorithm for approximating the **maximum cut** within a factor of 2, which is pretty good considering that MAX CUT is **NP**-hard.

⁴This is tight for $m = 1$, but I suspect it's an underestimate for larger m . The main source of slop in the analysis seems to be the step $\mathbf{E}[X \mid X \geq m/2] \leq m$; using a concentration bound, we should be able to show a much stronger inequality here and thus a much larger lower bound on p .

There exist better approximation algorithms. Goemans and Williamson [GW95] give a 0.87856-approximation algorithm for MAX CUT based on randomized rounding of a semidefinite program.⁵ The analysis of this algorithm is a little involved, so we won't attempt to show this here, but we will describe (in §11.2.2) an earlier result, also due to Goemans and Williamson [GW94], that gives a $\frac{3}{4}$ -approximation to MAX SAT using a similar technique.

11.2.2 MAX SAT

Like MAX CUT, MAX SAT is an **NP**-hard optimization problem that sounds like a very short story about Max. We are given a **satisfiability problem** in **conjunctive normal form**: as a set of m **clauses**, each of which is the OR of a bunch of variables or their negations. We want to choose values for the n variables that **satisfy** as many of the clauses as possible, where a clause is satisfied if it contains a true variable or the negation of a false variable.⁶

We can instantly satisfy at least $m/2$ clauses on average by assigning values to the variables independently and uniformly at random; the analysis is the same as in §11.2.1 for large cuts, since a random assignment makes the first variable in a clause true with probability $1/2$. Since this approach doesn't require thinking and doesn't use the fact that many of our clauses may have more than one variable, we can probably do better. Except we can't do better in the worst case, because it might be that our clauses consist entirely of x and $\neg x$ for some variable x ; clearly, we can only satisfy half of these. We could do better if we knew all of our clauses consisted of at least k distinct literals (satisfied with probability $1 - 2^{-k}$), but we can't count on this. We also can't count on clauses not being duplicated, so it may turn out that skipping a few hard-to-satisfy small clauses hurts us if they occur many times.

Our goal will be to get a good **approximation ratio**, defined as the ratio between the number of clauses we manage to satisfy and the actual maximum that can be satisfied. The tricky part in designing a **approxima-**

⁵Semidefinite programs are like linear programs except that the variables are vectors instead of scalars, and the objective function and constraints apply to linear combinations of dot-products of these variables. The Goemans-Williamson MAX CUT algorithm is based on a relaxation of the integer optimization problem of maximizing $\sum x_i x_j$ where each $x_i \in \{-1, +1\}$ encodes membership in S or T . They instead allow x_i to be any unit vector in an n -dimensional space, and then take the sign of the dot-product with a random unit vector r to map each optimized x_i to one side of the cut or the other.

⁶The presentation here follows [MR95, §5.2], which in turn is mostly based on a classic paper of Goemans and Williamson [GW94].

tion algorithms is showing that the denominator here won't be too big. We can do this using a standard trick, of expressing our original problem as an **integer program** and then **relaxing** it to a **linear program**⁷ whose solution doesn't have to consist of integers. We then convert the fractional solution back to an integer solution by rounding some of the variables to integer values randomly in a way that preserves their expectations, a technique known as **randomized rounding**.⁸

Here is the integer program (taken from [MR95, §5.2]). We let $z_j \in \{0, 1\}$ represent whether clause C_j is satisfied, and let $y_i \in \{0, 1\}$ be the value of variable x_i . We also let C_j^+ and C_j^- be the set of variables that appear in C_j with and without negation. The problem is to maximize

$$\sum_{j=1}^m z_j$$

subject to

$$\sum_{i \in C_j^+} y_i + \sum_{i \in C_j^-} (1 - y_i) \geq z_j,$$

for all j .

⁷A **linear program** is an optimization problem where we want to maximize (or minimize) some linear **objective function** of the variables subject to linear-inequality constraints. A simple example would be to maximize $x + y$ subject to $2x + y \leq 1$ and $x + 3y \leq 1$; here the **optimal solution** is the assignment $x = \frac{2}{5}, y = \frac{1}{5}$, which sets the objective function to its maximum possible value $\frac{3}{5}$. An **integer program** is a linear program where some of the variables are restricted to be integers. Determining if an integer program even has a solution is **NP**-complete; in contrast, linear programs can be solved in polynomial time. We can **relax** an integer program to a linear program by dropping the requirements that variables be integers; this will let us find a **fractional solution** that is at least as good as the best integer solution, but might be undesirable because it tells us to do something that is ludicrous in the context of our original problem, like only putting half a passenger on the next plane.

Linear programming has an interesting history. The basic ideas were developed independently by Leonid Kantorovich in the Soviet Union and George Dantzig in the United States around the start of the Second World War. Kantorovich's work had direct relevance to Soviet planning problems, but wasn't pursued seriously because it threatened the political status of the planners, required computational resources that weren't available at the time, and looked suspiciously like trying to sneak a capitalist-style price system into the planning process; for a fictionalized account of this tragedy, see [Spu12]. Dantzig's work, which included the development of the **simplex method** for solving linear programs, had a higher impact, although its publication was delayed until 1947 by wartime secrecy.

⁸ Randomized rounding was invented by Raghavan and Thompson [RT87]; the particular application here is due to Goemans and Williamson [GW94].

The main trick here is to encode OR in the constraints; there is no requirement that z_j is the OR of the y_i and $(1 - y_i)$ values, but we maximize the objective function by setting it that way.

Sadly, solving integer programs like the above is **NP**-hard (which is not surprising, since if we could solve this particular one, we could solve SAT). But if we drop the requirements that $y_i, z_j \in \{0, 1\}$ and replace them with $0 \leq y_i \leq 1$ and $0 \leq z_j \leq 1$, we get a linear program—solvable in polynomial time—with an optimal value at least as good as the value for the integer program, for the simple reason that any solution to the integer program is also a solution to the linear program.

The problem now is that the solution to the linear program is likely to be **fractional**: instead of getting useful 0–1 values, we might find out we are supposed to make x_i only $2/3$ true. So we need one more trick to turn the fractional values back into integers. This is the randomized rounding step: given a fractional assignment \hat{y}_i , we set x_i to true with probability \hat{y}_i .

So what does randomized rounding do to clauses? In our fractional solution, a clause might have value \hat{z}_j , obtained by summing up bits and pieces of partially-true variables. We'd like to argue that the rounded version gives a similar probability that C_j is satisfied.

Suppose C_j has k variables; to make things simpler, we'll pretend that C_j is exactly $x_1 \vee x_2 \vee \dots \vee x_k$. Then the probability that C_j is satisfied is exactly $1 - \prod_{i=1}^k (1 - \hat{y}_i)$. This quantity is minimized subject to $\sum_{i=1}^k \hat{y}_i \geq \hat{z}_j$ by setting all \hat{y}_i equal to \hat{z}_j/k (easy application of Lagrange multipliers, or can be shown using a convexity argument). Writing z for \hat{z}_j , this gives

$$\begin{aligned} \Pr [C_j \text{ is satisfied}] &= 1 - \prod_{i=1}^k (1 - \hat{y}_i) \\ &\geq 1 - \prod_{i=1}^k (1 - z/k) \\ &= 1 - (1 - z/k)^k \\ &\geq z(1 - (1 - 1/k)^k). \\ &\geq z(1 - 1/e). \end{aligned}$$

The second-to-last step looks like a typo, but it actually works. The idea is to observe that the function $f(z) = 1 - (1 - z/k)^k$ is concave (Proof: $\frac{d^2}{dz^2} f(z) = -\frac{k-1}{k}(1 - z/k)^{k-2} < 0$), while $g(z) = z(1 - (1 - 1/k)^k)$ is linear, so since $f(0) = 0 = g(0)$ and $f(1) = 1 - (1 - 1/k)^k = g(1)$, any point in between must have $f(z) \geq g(z)$. An example of this argument for $k = 3$ is depicted in Figure 11.1.

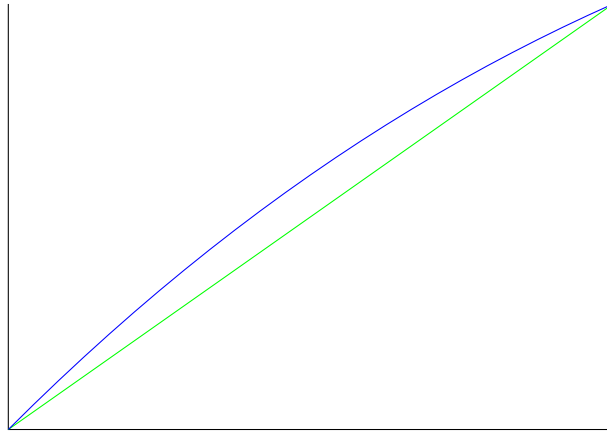


Figure 11.1: Tricky step in MAX SAT argument, showing $(1 - (1 - z/k)^k) \geq z(1 - (1 - 1/k)^k)$ when $0 \leq z \leq 1$. The case $k = 3$ is depicted.

Since each clause is satisfied with probability at least $\hat{z}_j(1 - 1/e)$, the expected number of satisfied clauses is at least $(1 - 1/e) \sum_j \hat{z}_j$, which is at least $(1 - 1/e)$ times the optimum. This gives an approximation ratio of slightly more than 0.632, which is better than $1/2$, but still kind of weak.

So now we apply the second trick from [GW94]: we'll observe that, on a per-clause basis, we have a randomized rounding algorithm that is good at satisfying small clauses (the coefficient $(1 - (1 - 1/k)^k)$ goes all the way up to 1 when $k = 1$), and our earlier dumb algorithm that is good at satisfying big clauses. We can't combine these directly (the two algorithms demand different assignments), but we can run both in parallel and take whichever gives a better result.

To show that this works, let X_j be indicator for the event that clause j is satisfied by the randomized-rounding algorithm and Y_j the indicator for the event that it is satisfied by the simple algorithm. Then if C_j has k literals,

$$\begin{aligned} \mathbb{E}[X_j] + \mathbb{E}[Y_j] &\geq (1 - 2^{-k}) + (1 - (1 - 1/k)^k) \hat{z}_j \\ &\geq ((1 - 2^{-k}) + (1 - (1 - 1/k)^k)) \hat{z}_j \\ &= (2 - 2^{-k} - (1 - 1/k)^k) \hat{z}_j. \end{aligned}$$

The coefficient here is exactly $3/2$ when $k = 1$ or $k = 2$, and rises thereafter, so for integer k we have $\mathbb{E}[X_j] + \mathbb{E}[Y_j] \geq (3/2) \hat{z}_j$. Summing over all j then gives $\mathbb{E}[\sum_j X_j] + \mathbb{E}[\sum_j Y_j] \geq (3/2) \sum_j \hat{z}_j$. But then one of the two expected sums must beat $(3/4) \sum_j \hat{z}_j$, giving us a $(3/4)$ -approximation algorithm.

11.3 The Lovász Local Lemma

Suppose we have a finite set of bad events \mathcal{A} , and we want to show that with nonzero probability, none of these events occur. Formally, we want to show $\Pr \left[\bigcap_{A \in \mathcal{A}} \bar{A} \right] > 0$.

Our usual trick so far has been to use the union bound (4.1.1) to show that $\sum_{A \in \mathcal{A}} \Pr[A] < 1$. But this only works if the events are actually improbable. If the union bound doesn't work, we might be lucky enough to have the events be independent; in this case, $\Pr \left[\bigcap_{A \in \mathcal{A}} \bar{A} \right] = \prod_{A \in \mathcal{A}} \Pr[\bar{A}] > 0$, as long as each event \bar{A} occurs with positive probability. But most of the time, we'll find that the events we care about aren't independent, so this won't work either.

The **Lovász Local Lemma** [EL75] handles a situation intermediate between these two extremes, where events are generally not independent of each other, but each collection of events that are not independent of some particular event A has low total probability. In the original version, it's non-constructive: the lemma shows a nonzero probability that none of the events occur, but this probability may be very small if we try to sample the events at random, and there is no guidance for how to find a particular outcome that makes all the events false.

Subsequent work [Bec91, Alo91, MR98, CS00, Sri08, Mos09, MT10] showed how, when the events A are determined by some underlying set of independent variables and independence between two events is detected by having non-overlapping sets of underlying variables, an actual solution could be found in polynomial expected time. The final result in this series, due to Moser and Tardos [MT10], gives the same bounds as in the original non-constructive lemma, using the simplest algorithm imaginable: whenever some bad event A occurs, try to get rid of it by resampling all of its variables, and continue until no bad events are left.

11.3.1 General version

A formal statement of the general lemma is:⁹

Lemma 11.3.1. *Let $\mathcal{A} = A_1, \dots, A_m$ be a finite collection of events on some probability space, and for each $A \in \mathcal{A}$, let $\Gamma(A)$ be a set of events such that A is independent of all events not in $\Gamma^+ A = \{A\} \cup \Gamma(A)$. If there exist real*

⁹This version is adapted from [MT10].

numbers $0 < x_A < 1$ such that, for all events $A \in \mathcal{A}$

$$\Pr[A] \leq x_A \prod_{B \in \Gamma(A)} (1 - x_B), \quad (11.3.1)$$

then

$$\Pr \left[\bigcap_{A \in \mathcal{A}} \bar{A} \right] \geq \prod_{A \in \mathcal{A}} (1 - x_A). \quad (11.3.2)$$

In particular, this means that the probability that none of the A_i occur is not zero, since we have assumed $x_{A_i} < 1$ holds for all i .

The role of x_A in the original proof is to act as an upper bound on the probability that A occurs given that some collection of other events doesn't occur. For the constructive proof, the x_A are used to show a bound on the number of resampling steps needed until none of the A occur.

11.3.2 Symmetric version

For many applications, having to come up with the x_A values can be awkward. The following symmetric version is often used instead:

Corollary 11.3.2. *Let \mathcal{A} and Γ be as in Lemma 11.3.1. Suppose that there are constants p and d , such that for all $A \in \mathcal{A}$, we have $\Pr[A] \leq p$ and $|\Gamma(A)| \leq d$. Then if $ep(d+1) < 1$, $\Pr \left[\bigcap_{A \in \mathcal{A}} \bar{A} \right] \neq 0$.*

Proof. Basically, we are going to pick a single value x such that $x_A = x$ for all A in \mathcal{A} , and (11.3.1) is satisfied. This works as long as $p \leq x(1-x)^d$, as in this case we have, for all A , $\Pr[A] \leq p \leq x(1-x)^d \leq x(1-x)^{|\Gamma(A)|} = x_A \left(\prod_{B \in \Gamma(A)} (1 - x_B) \right)$.

For fixed d , $x(1-x)^d$ is maximized using the usual trick: $\frac{d}{dx} x(1-x)^d = (1-x)^d - xd(1-x)^{d-1} = 0$ gives $(1-x) - xd = 0$ or $x = \frac{1}{d+1}$. So now we need $p \leq \frac{1}{d+1} \left(1 - \frac{1}{d+1}\right)^d$. It is possible to show that $1/e < \left(1 - \frac{1}{d+1}\right)^d$ for all $d \geq 0$.¹⁰ So $ep(d+1) \leq 1$ implies $p \leq \frac{1}{e(d+1)} \leq \left(1 - \frac{1}{d+1}\right)^d \frac{1}{d+1} \leq x(1-x)^{|\Gamma(A)|}$ as required by Lemma 11.3.1. \square

¹⁰Observe that $\lim_{d \rightarrow \infty} \left(1 - \frac{1}{d+1}\right)^d = e^{-1}$ and that $f(d) = \left(1 - \frac{1}{d+1}\right)^d = e^{-1}$ is a decreasing function. The last part can be shown by taking the derivative of $\log f(d)$. See the solution to Problem F.1.4 for the gory details.

11.3.3 Applications

Here we give some simple applications of the lemma.

11.3.3.1 Graph coloring

Let's start with a problem where we know what the right answer should be. Suppose we want to color the vertices of a cycle with c colors, so that no edge has two endpoints with the same color. How many colors do we need?

Using brains, we can quickly figure out that $c = 3$ is enough. Without brains, we could try coloring the vertices randomly: but in a cycle with n vertices and n edges, on average n/c of the edges will be monochromatic, since each edge is monochromatic with probability $1/c$. If these bad events were independent, we could argue that there was a $(1 - 1/c)^n > 0$ probability that none of them occurred, but they aren't, so we can't. Instead, we'll use the local lemma.

The set of bad events \mathcal{A} is just the set of events $A_i = [\text{edge } i \text{ is monochromatic}]$. We've already computed $p = 1/c$. To get d , notice that each edge only shares a vertex with two other edges, so $|\Gamma(A_i)| \leq 2$. Corollary 11.3.2 then says that there is a good coloring as long as $ep(d + 1) = 3e/c \leq 1$, which holds as long as $c \geq 9$. We've just shown we can 9-color a cycle. If use the asymmetric version, we can set all x_A to $1/3$ and show that $p \leq \frac{1}{3} \left(1 - \frac{1}{3}\right)^2 = \frac{4}{27}$ would also work; with this we can 7-color a cycle. This is still not as good as what we can do if we are paying attention, but not bad for a procedure that doesn't use the structure of the problem much.

11.3.3.2 Satisfiability of k -CNF formulas

A more sophisticated application is demonstrating satisfiability for k -CNF formulas where each variable appears in a bounded number of clauses. Recall that a **k -CNF formula** consists of m **clauses**, each of which consists of exactly k variables or their negations (collectively called **literals**). It is **satisfied** if at least one literal in every clause is assigned the value true.

Suppose that each variable appears in at most ℓ clauses. Let \mathcal{A} consist of all the events $A_i = [\text{clause } i \text{ is not satisfied}]$. Then, for all i , $\Pr[A_i] = 2^{-k}$ exactly, and $|\Gamma(A_i)| \leq d = k(\ell - 1)$ since each variable in A_i is shared with at most $\ell - 1$ other clauses, and A_i will be independent of all events A_j with which it doesn't share a variable. So if $ep(d + 1) = e2^{-k}(k(\ell - 1) + 1) \leq 1$, which holds if $\ell \leq \frac{2^k}{ek} + 1$, Corollary 11.3.2 tells us that a satisfying assignment

exists.¹¹

Corollary 11.3.2 doesn't let us actually find a satisfying assignment, but it turns out we can do that too. We'll return to this when we talk about the constructive version of the local lemma in §11.3.5.

11.3.3.3 Hypergraph 2-colorability

This was the original motivation for the lemma. A **k -uniform hypergraph** $G = \langle V, E \rangle$ has a set of **hyperedges** E , each of which contains exactly k vertices from V . A **2-coloring** of the hypergraph assigns one of two colors to each vertex in v , so that no edge is monochromatic.

Suppose we color vertices at random. Pick some edge S , and let A_S be the probability that it is monochromatic. Then $\Pr[A_S] = 2^{1-k}$ exactly. We also have that A_S depends only on events A_T where $S \cap T \neq \emptyset$. Suppose that there is a bound d on the number of hyperedges T that overlap with any hyperedge S . Then Corollary 11.3.2 says that there exists a good coloring as long as $e2^{1-k}(d+1) < 1$. This holds for any hypergraph with $d < 2^{k-1}/e - 1$.

Unlike graph coloring, it's not so obvious how to solve hypergraph coloring with a simple greedy algorithm. So here the local lemma seems to buy us something we didn't already have.

11.3.4 Non-constructive proof

This is essentially the same argument presented in [MR95, §5.5], but we adapt the notation to match the statement in terms of neighborhoods $\Gamma(A)$ instead of edges in a **dependency graph**, where an event is independent of all events that are not its successors in the graph. The two formulations are identical, since we can always represent the neighborhoods $\Gamma(A)$ by creating an edge from A to B when B is in $\Gamma(A)$; and conversely we can convert a dependency graph into a neighborhood structure by making $\Gamma(A)$ the set of successors of A in the dependency graph.

The proof is a bit easier than the non-constructive version, while also being a bit more general because the neighborhood structure doesn't depend on identifying a collection of underlying independent variables.

The idea is to order the events in \mathcal{A} as A_1, A_2, \dots, A_n and expand $\Pr\left[\bigcap_{i=1}^n \bar{A}_i\right]$ as $\prod_{i=1}^n \Pr\left[\bar{A}_i \mid \bigcap_{j=1}^{i-1} \bar{A}_j\right]$. If we can show that every term in the product is nonzero, we are done.

¹¹To avoid over-selling this claim, it's worth noting that the bound on ℓ only reaches 2 at $k = 4$, although it starts growing pretty fast after that.

To do this, we will show by induction on $|S|$ the more general statement that for any A and any $S \subseteq \mathcal{A}$ with $A \notin S$,

$$\Pr \left[A \mid \bigcap_{B \in S} \bar{B} \right] \leq x_A. \quad (11.3.3)$$

When $|S| = 0$, this just says $\Pr[A] \leq x_A$, which follows immediately from (11.3.1).

For larger S , split S into $S_1 = S \cap \Gamma(A)$, the events in S that might not be independent of A ; and $S_2 = S \setminus \Gamma(A)$, the events in S that we know to be independent of A . If $S_2 = S$, then A is independent of all events in S , and (11.3.3) follows immediately from $\Pr \left[A \mid \bigcap_{B \in S} \bar{B} \right] = \Pr[A] \leq x_A \prod_{B \in \Gamma(A)} (1 - x_B) \leq x_A$. Otherwise $|S_2| < |S|$, which means that we can assume the induction hypothesis holds for S_2 .

Write C_1 for the event $\bigcap_{B \in S_1} \bar{B}$ and C_2 for the event $\bigcap_{B \in S_2} \bar{B}$. Then $\bigcap_{B \in S} \bar{B} = C_1 \cap C_2$ and we can expand

$$\begin{aligned} \Pr \left[A \mid \bigcap_{B \in S} \bar{B} \right] &= \frac{\Pr[A \cap C_1 \cap C_2]}{\Pr[C_1 \cap C_2]} \\ &= \frac{\Pr[A \cap C_1 \mid C_2] \Pr[C_2]}{\Pr[C_1 \mid C_2] \Pr[C_2]} \\ &= \frac{\Pr[A \cap C_1 \mid C_2]}{\Pr[C_1 \mid C_2]}. \end{aligned} \quad (11.3.4)$$

We don't need to do anything particularly clever with the numerator:

$$\begin{aligned} \Pr[A \cap C_1 \mid C_2] &\leq \Pr[A \mid C_2] \\ &= \Pr[A] \\ &\leq x_A \left(\prod_{B \in \Gamma(A)} (1 - x_B) \right), \end{aligned} \quad (11.3.5)$$

from (11.3.1) and the fact that A is independent of all B in S_2 and thus also independent of C_2 .

For the denominator, we expand C_1 back out to $\bigcap_{B \in S_1} \bar{B}$ and break out the induction hypothesis. To bound $\Pr \left[\bigcap_{B \in S_1} \bar{B} \mid C_2 \right]$, we order S_1 arbitrarily as $\{B_1, \dots, B_r\}$, for some r , and show by induction on ℓ as ℓ goes from 1 to r that

$$\Pr \left[\bigcap_{i=1}^{\ell} \bar{B}_i \mid C_2 \right] \geq \prod_{i=1}^{\ell} (1 - x_{B_i}). \quad (11.3.6)$$

The proof is that, for $\ell = 1$,

$$\begin{aligned}\Pr[\bar{B}_1 \mid C_2] &= 1 - \Pr[B_1 \mid C_2] \\ &\geq 1 - x_{B_1}\end{aligned}$$

using the outer induction hypothesis (11.3.3), and for larger ℓ , we can compute

$$\begin{aligned}\Pr\left[\bigcap_{i=1}^{\ell} \bar{B}_i \mid C_2\right] &= \Pr\left[\bar{B}_\ell \mid \left(\bigcap_{i=1}^{\ell-1} \bar{B}_i\right) \cap C_2\right] \cdot \Pr\left[\bigcap_{i=1}^{\ell-1} \bar{B}_i \mid C_2\right] \\ &\geq (1 - x_{B_\ell}) \prod_{i=1}^{\ell-1} (1 - x_{B_i}) \\ &= \prod_{i=1}^{\ell} (1 - x_{B_i}),\end{aligned}$$

where the second-to-last step uses the outer induction hypothesis (11.3.3) for the first term and the inner induction hypothesis (11.3.6) for the rest. This completes the proof of the inner induction.

When $\ell = r$, we get

$$\begin{aligned}\Pr[C_1 \mid C_2] &= \Pr\left[\bigcap_{i=1}^r \bar{B}_i \mid C_2\right] \\ &\geq \prod_{B \in S_1} (1 - x_B).\end{aligned}\tag{11.3.7}$$

Substituting (11.3.5) and (11.3.7) into (11.3.4) gives

$$\begin{aligned}\Pr\left[A \mid \bigcap_{B \in S} \bar{B}\right] &\leq \frac{x_A \left(\prod_{B \in \Gamma(A)} (1 - x_B)\right)}{\prod_{B \in S_1} (1 - x_B)} \\ &= x_A \left(\prod_{B \in \Gamma(A) \setminus S_1} (1 - x_B)\right) \\ &\leq x_A.\end{aligned}$$

This completes the proof of the outer induction.

To get the bound (11.3.2), we reach back inside the proof and repeat the argument for (11.3.7) with $\bigcap_{A \in \mathcal{A}} \bar{A}$ in place of C_1 and without the conditioning on C_2 . We order \mathcal{A} arbitrarily as $\{A_1, A_2, \dots, A_m\}$ and show by induction on k that

$$\Pr\left[\bigcap_{i=1}^k \bar{A}_i\right] \geq \prod_{i=1}^k (1 - x_{A_i}).\tag{11.3.8}$$

For the base case we have $k = 0$ and $\Pr[\Omega] \geq 1$, using the usual conventions on empty products. For larger k , we have

$$\begin{aligned} \Pr \left[\bigcap_{i=1}^k \bar{A}_i \right] &= \Pr \left[\bar{A}_k \mid \bigcap_{i=1}^{k-1} \bar{A}_i \right] \Pr \left[\bigcap_{i=1}^{k-1} \bar{A}_i \right] \\ &\geq (1 - x_{A_k}) \prod_{i=1}^{k-1} (1 - x_{A_i}) \\ &\geq \prod_{i=1}^k (1 - x_{A_i}), \end{aligned}$$

where in the second-to-last step we use (11.3.3) for the first term and the induction hypothesis (11.3.8) for the big product.

Setting $k = n$ finishes the proof.

11.3.5 Constructive proof

We now describe the constructive proof of the Lovász local lemma due to Moser and Tardos [MT10], which is based on a slightly more specialized construction of Moser alone [Mos09]. This version applies when our set of bad events \mathcal{A} is defined in terms of a set of *independent* variables \mathcal{P} , where each $A \in \mathcal{A}$ is determined by some set of variables $\text{vbl}(A) \subseteq \mathcal{P}$, and $\Gamma(A)$ is defined to be the set of all events $B \neq A$ that share at least one variable with A ; i.e., $\Gamma(A) = \{B \in \mathcal{A} \setminus \{A\} \mid \text{vbl}(B) \cap \text{vbl}(A) \neq \emptyset\}$.

In this case, we can attempt to find an assignment to the variables that makes none of the A occur using the obvious algorithm of sampling an initial state randomly, then resampling all variables in $\text{vbl}(A)$ whenever we see some bad A occur. Astonishingly, this actually works in a reasonable amount of time, without even any cleverness in deciding which A to resample at each step, if the conditions for Lemma 11.3.1 hold for x that are not too large. In particular, we will show that a good assignment is found after each A is resampled at most $\frac{x_A}{1-x_A}$ times on average.

Lemma 11.3.3. *Under the conditions of Lemma 11.3.1, the Moser-Tardos procedure does at most*

$$\sum_A \frac{x_A}{1-x_A}$$

resampling steps on average.

We defer the proof of Lemma 11.3.3 for the moment. For most applications, the following symmetric version is easier to work with:

Corollary 11.3.4. *Under the conditions of Corollary 11.3.2, the Moser-Tardos procedure does at most m/d resampling steps on average.*

Proof. Follows from Lemma 11.3.3 and the choice of $x_A = \frac{1}{d+1}$ in the proof of Corollary 11.3.2. \square

How this expected m/d bound translates into actual time depends on the cost of each resampling step. The expensive part at each step is likely to be the cost of finding an A that occurs and thus needs to be resampled.

Intuitively, we might expect the resampling to work because if each $A \in \mathcal{A}$ has a small enough neighborhood $\Gamma(A)$ and a low enough probability, then whenever we resample A 's variables, it's likely that we fix A and unlikely that we break too many B events in A 's neighborhood. It turns out to be tricky to quantify how this process propagates outward, so the actual proof uses a different idea that essentially looks at this process in reverse, looking for each resampled event A at a set of previous events whose resampling we can blame for making A occur, and then showing that this tree (which will include every resampling operation as one of its vertices) can't be too big.

The first step is to fix some strategy for choosing which event A to resample at each step. We don't care what this strategy is; we just want it to be the case that the sequence of events depends only on the random choices made by the algorithm in its initial sampling and each resampling. We can then define an **execution log** C that lists the sequence of events C_1, C_2, C_3, \dots that are resampled at each step of the execution.

From C we construct a **witness tree** T_t for each resampling step t whose nodes are labeled with events, with the property that the children of a node v labeled with event A_v are labeled with events in $\Gamma^+(A_v) = \{A_v\} \cup \Gamma(A_v)$.

The root of T_t is labeled with C_t . To construct the rest of the tree, we work backwards through $C_{t-1}, C_{t-2}, \dots, C_1$, and for each event C_i we encounter, we attach C_i as a child of the deepest v we can find with $C_i \in \Gamma^+(A_v)$, choosing arbitrarily if there is more than one such v , and discarding C_i if there is no such v .

Now we can ask what the probability is that we see some particular witness tree T in the execution log. Each vertex of T corresponds to some event A_v that we resample because it occurs; in order for it to occur, the previous assignments of each variable in $\text{vbl}(A_v)$ must have made A_v true, which occurs with probability $\Pr[A_v]$. But since we resample all the variables in A_v , any subsequent assignments to these variables are independent of the

ones that contributed to v ; with sufficient handwaving (or a rather detailed coupling argument as found in [MT10]) this gives that each event A_v occurs with independent probability $\Pr[A_v]$, giving $\Pr[T] = \prod_{v \in T} \Pr[A_v]$.

Why do we care about this? Because every event we resample is the root of some witness tree, and we can argue that every event we resample is the root of a *distinct* witness tree. The proof is that since we only discard events B that have $\text{vbl}(B)$ disjoint from all nodes already in the tree, once we put A at the root, any other instance of A gets included. So the witness tree rooted at the i -th occurrence of A in C will include exactly i copies of A , unlike the witness tree rooted at the j -th copy for $j \neq i$.

Now comes the sneaky trick: we'll bound how many distinct witness trees T we expect to see rooted at A , given that each occurs with probability $\prod_{v \in T} \Pr[A_v]$. This is done by constructing a **branching process** using the x_B values from Lemma 11.3.1 as probabilities of a node with label A having a child labeled B for each B in $\Gamma^+(A)$, and doing algebraic manipulations on the resulting probabilities until $\prod_{v \in T} \Pr[A_v]$ shows up. We can then sum over the expected number of copies of trees to get a bound on the expected number of events in the execution log (since each such even is the root of some tree), which is equal to the expected number of resamplings.

Consider the process where we start with a root labeled A , and for each vertex v with label A_v , give it a child labeled B for each $B \in \Gamma^+(A_v)$ with independent probability x_B . We'll now calculate the probability p_T that this process generates a particular tree T in the set \mathcal{T}_A of trees with root A .

Let $x'_B = x_B \prod_{C \in \Gamma(B)} (1 - x_C)$. Note that (11.3.1) says precisely that $\Pr[B] \leq x'_B$.

For each vertex v in T , let $W_v \subseteq \Gamma^+(A_v)$ be the set of events $B \in \Gamma^+(A_v)$ that *don't* occur as labels of children of v . The probability of getting T is equal to the product of the probabilities at each v of getting all of its children and none of its non-children. The non-children of v collectively contribute $\prod_{B \in W_v} (1 - x_B)$ to the product, and v itself contributes x_{A_v} (via the product for its parent), unless v is the root node. So we can express the giant product as

$$p_T = \frac{1}{x_A} \prod_{v \in T} \left(x_{A_v} \prod_{B \in W_v} (1 - x_B) \right).$$

We don't like the W_v very much, so we get rid of them by taking the product

of B in $\Gamma^+(A)$, then dividing out the ones that aren't in W_v . This gives

$$\begin{aligned} p_T &= \frac{1}{x_A} \prod_{v \in T} \left(x_{A_v} \prod_{B \in W_v} (1 - x_B) \right). \\ &= \frac{1}{x_A} \prod_{v \in T} \left(x_{A_v} \prod_{B \in \Gamma^+(A_v)} (1 - x_B) \prod_{B \in \Gamma^+(A_v) \setminus W_v} \frac{1}{1 - x_B} \right). \end{aligned}$$

This seems like we exchanged one annoying index set for another, but each element of $\Gamma^+(A_v) \setminus W_v$ is $A_{v'}$ for some child of v in T . So we can push these factors down to the children, and since we are multiplying over all vertices in T , they will each show up exactly once except at the root. To keep the products clean, we'll throw in $\frac{1}{1-x_A}$ for the root as well, but compensate for this by multiplying by $1 - x_A$ on the outside of the product. This gives

$$\begin{aligned} p_T &= \frac{1 - x_A}{x_A} \prod_{v \in T} \left(\frac{x_{A_v}}{1 - x_{A_v}} \prod_{B \in \Gamma^+(A_v)} (1 - x_B) \right). \\ &= \frac{1 - x_A}{x_A} \prod_{v \in T} \left(x_{A_v} \prod_{B \in \Gamma(A_v)} (1 - x_B) \right). \\ &= \frac{1 - x_A}{x_A} \prod_{v \in T} x'_{A_v}. \end{aligned}$$

Now we can bound the expected number of trees rooted at A that appear in C , assuming (11.3.1) holds. Letting \mathcal{T}_A as before be the set of all such trees and N_A the number that appear in C , we have

$$\begin{aligned} \mathbb{E}[N_A] &= \sum_{T \in \mathcal{T}_A} \Pr[T \text{ appears in } C] \\ &\leq \sum_{T \in \mathcal{T}_A} \prod_{v \in T} \Pr[A(v)] \\ &\leq \sum_{T \in \mathcal{T}_A} \prod_{v \in T} x'_{A_v} \\ &= \sum_{T \in \mathcal{T}_A} \frac{x_A}{1 - x_A} p_T \\ &= \frac{x_A}{1 - x_A} \sum_{T \in \mathcal{T}_A} p_T \\ &\leq \frac{x_A}{1 - x_A}. \end{aligned}$$

The last sum is bounded by one because occurrences of particular trees T in \mathcal{T}_A are all disjoint events.

Now sum over all A , and we're done.

Chapter 12

Derandomization

Last updated 2019. Some material may be out of date.

Derandomization is the process of taking a randomized algorithm and turning it into a deterministic algorithm. This is useful both for practical reasons (deterministic algorithms are more predictable, which makes them easier to debug and gives hard guarantees on running time) and theoretical reasons (if we can derandomize any randomized algorithm we could show results like $\mathbf{P} = \mathbf{RP}$,¹ which would reduce the number of complexity classes that complexity theorists otherwise have to deal with). It may also be the case that derandomizing a randomized algorithm can be used for **probability amplification**, where we replace a low probability of success with a higher probability, in this case 1.

There are basically two approaches to derandomization:

1. Reduce the number of random bits used down to $O(\log n)$, and then search through all choices of random bits exhaustively. For example, if we only need pairwise independence, we could use the XOR technique from §5.1.2.1 to replace a large collection of variables with a small collection of random bits.

Except for the exhaustive search part, this is how randomized algorithms are implemented in practice: rather than burning random bits continuously, a **pseudorandom generator** is initialized from a **seed** consisting of a small number of random bits. For pretty much all of

¹The class \mathbf{RP} consists of all languages L for which there is a polynomial-time randomized algorithm that correctly outputs “yes” given an input x in L with probability at least $1/2$, and never answers “yes” given an input x not in L . See §1.5.2 for a more extended description of \mathbf{RP} and other randomized complexity classes.

the randomized algorithms we know about, we don't even need to use a particularly strong pseudorandom generator. This is largely because current popular generators are the products of a process of evolution: pseudorandom generators that cause wonky behavior or fail to pass tests that approximate the assumptions made about them by typical randomized algorithms are abandoned in favor of better generators.²

From a theoretical perspective, pseudorandom generators offer the possibility of eliminating randomization from all randomized algorithms, except there is a complication. While (under reasonable cryptographic assumptions) there exist **cryptographically secure pseudorandom generators** whose output is indistinguishable from a genuinely random source by polynomial-time algorithms (including algorithms originally intended for other purposes), such generators are inherently incapable of reducing the number of random bits down to the $O(\log n)$ needed for exhaustive search. The reason is that any pseudorandom generator with only polynomially-many seeds can't be cryptographically secure, because we can distinguish it from a random source by just checking its output against the output for all possible seeds. Whether there is some other method for transforming an arbitrary algorithm in **RP** or **BPP** into a deterministic algorithm remains an open problem in complexity theory (and beyond the scope of this course).

2. Start with a specific randomized protocol and analyze its behavior enough that we can replace the random bits it uses with specific, deterministically-chosen bits we can compute. This is the main approach we will describe below. A non-constructive variant of this shows that we can always replace the random bits used by all inputs of a given size with a few carefully-selected fixed sequences (Adleman's Theorem, described in §12.2). More practical is the **method of conditional probabilities**, which chooses random bits sequentially based on which value is more likely to give a good outcome (see §12.4).

12.1 Deterministic vs. randomized algorithms

In thinking about derandomization, it can be helpful to have more than one way to look at a randomized algorithm. So far, we've describe random-

²Having cheaper computers helps here as well. Nobody would have been willing to spend 2496 bytes on the state vector for Mersenne Twister [MN98] back in 1975, but by 1998 this amount of memory was trivial for pretty much any computing device except the tiniest microcontrollers.

ized algorithms as random choices of deterministic algorithms ($M_r(x)$) or, equivalently, as deterministic algorithms that happen to have random inputs ($M(r, x)$). This gives a very static view of how randomness is used in the algorithm. A more dynamic view is obtained by thinking of the computation of a randomized algorithm as a **computation tree**, where each path through the tree corresponds to a computation with a fixed set of random bits and a branch in the tree corresponds to a random decision. In either case we want an execution to give us the right answer with reasonably high probability, whether that probability measures our chance of getting a good deterministic machine for our particular input or landing on a good computation path.

12.2 Adleman's theorem

The idea of picking a good deterministic machine is the basis for **Adleman's theorem** [Adl78], a classic result in complexity theory. Adleman's theorem says that we can always replace randomness by an oracle that presents us with a fixed string of **advice** p_n that depends only on the size of the input n and has size polynomial in n . The formal statement of the theorem relates the class **RP**, which is the class of problems for which there exists a polynomial-time Turing machine $M(x, r)$ that outputs 1 at least half the time when $x \in L$ and never when $x \notin L$; and the class **P/poly**, which is the class of problems for which there is a polynomial-sized string p_n for each input size n and a polynomial-time Turing machine M' such that $M'(x, p_{|x|})$ outputs 1 if and only if $x \in L$.

Theorem 12.2.1. $\mathbf{RP} \subseteq \mathbf{P/poly}$.

Proof. The intuition is that if any one random string has a constant probability of making M happy, then by choosing enough random strings we can make the probability that M fails using on every random string for any given input so small that even after we sum over all inputs of a particular size, the probability of failure is still small using the union bound (4.1.1). This is an example of **probability amplification**, where we repeat a randomized algorithm many times to reduce its failure probability.

Formally, consider any fixed input x of size n , and imagine running M repeatedly on this input with $n + 1$ independent sequences of random bits r_1, r_2, \dots, r_{n+1} . If $x \notin L$, then $M(x, r_i)$ never outputs 1. If $x \in L$, then for each r_i , there is an independent probability of at least $1/2$ that $M(x, r_i) = 1$. So $\Pr[M(x, r_i) = 0] \leq 1/2$, and $\Pr[\forall i M(x, r_i) = 0] \leq 2^{-(n+1)}$. If we sum this probability of failure for each individual $x \in L$ of length n over the at

most 2^n such elements, we get a probability that any of them fail of at most $2^n 2^{-(n+1)} = 1/2$. Turning this upside down, any sequence of $n + 1$ random inputs includes a **witness** that $x \in L$ for *all* inputs x with probability at least $1/2$. It follows that a good sequence r_1, \dots, r_{n+1} , exists.

Our advice p_n is now some good sequence $p_n = \langle r_1 \dots r_{n+1} \rangle$, and the deterministic advice-taking algorithm that uses it runs $M(x, r_i)$ for each r_i and returns true if and only if at least one of these executions returns true. \square

The classic version of this theorem shows that anything you can do with a polynomial-size randomized circuit (a circuit made up of AND, OR, and NOT gates where some of the inputs are random bits, corresponding to the r input to M) can be done with a polynomial-size deterministic circuit (where now the p_n input is baked into the circuit, since we need a different circuit for each size n anyway).

A limitation of this result is that ordinary algorithms seem to be better described by **uniform** families of circuits, where there exists a polynomial-time algorithm that, given input n , outputs the circuit C_n for processing size- n inputs. In contrast, the class of circuits generated by Adleman's theorem is most likely **non-uniform**: the process of finding the good witnesses r_i is not something we know how to do in polynomial time (with the usual caveat that we can't prove much about what we can't do in polynomial time).

12.3 Limited independence

For some algorithms, it may be that full independence is not needed for all of the random bits. If the amount of independence needed is small enough, this may allow us to reduce the actual number of random bits consumed down to $O(\log n)$, at which point we can try all possible sequences of random bits in polynomial time.

Variants of this technique have been used heavily in the cryptography and complexity; see [LW05] for a survey of some early work in this area. We'll do a quick example of the method before moving onto more direct approaches.

12.3.1 MAX CUT

Let's look at the randomized MAX CUT algorithm from §11.2.1. In the original algorithm, we use n independent random bits to assign the n vertices to S or T . The idea is that by assigning each vertex independently at

random to one side of the cut or the other, each edge appears in the cut with probability $1/2$, giving a total of $m/2$ edges in the cut in expectation.

Suppose that we replace these n independent random bits with n pairwise-independent bits generated by taking XORs of subsets of $\lceil \lg(n+1) \rceil$ independent random bits as described in §5.1.2.1. Because the bits are pairwise-independent, the probability that the two endpoints of an edge are assigned to different sides of the cut is still exactly $1/2$. So on average we get $m/2$ edges in the cut as before, and there is at least one sequence of random bits that guarantees a cut at least this big.

But with only $\lceil \lg(n+1) \rceil$ random bits, there are only $2^{\lceil \lg(n+1) \rceil} < 2(n+1)$ possible sequences of random bits. If we try all of them, then we find a cut of size $m/2$ always. The total cost is $O(n(n+m))$ if we include the $O(n+m)$ cost of testing each cut. Note that this algorithm does not generate all 2^n possible cuts, but among those it *does* generate, there must be a large one.

In this particular case, we'll see below how to get the same result at a much lower cost, using more knowledge of the problem. So we have the typical trade-off between algorithm performance and algorithm designer effort.

12.4 The method of conditional probabilities

The **method of conditional probabilities** [Rag88] follows an execution of the randomized algorithm, but at each point where we would otherwise make a random decision, it makes a decision that minimizes the conditional probability of losing.

Structurally, this is similar to the method of bounded differences (see §5.3.3). Suppose our randomized algorithm generates m random values X_1, X_2, \dots, X_m . Let $f(X_1, \dots, X_m)$ be the indicator variable for our randomized algorithm failing (more generally, we can make it an expected cost or some other performance measure). Extend f to shorter sequences of values by defining $f(x_1, \dots, x_k) = \mathbb{E}[f(x_1, \dots, x_k, X_{k+1}, \dots, X_m)]$. Then $Y_k = f(X_1, \dots, X_k)$ is a Doob martingale, just as in the method of bounded differences. This implies that, for any partial sequence of values x_1, \dots, x_k , there exists some next value x_{k+1} such that $f(x_1, \dots, x_k) \geq f(x_1, \dots, x_k, x_{k+1})$. If we can find this value, we can follow a path on which f always decreases, and obtain an outcome of the algorithm $f(x_1, \dots, x_m)$ less than or equal to the initial value $f(\langle \rangle)$. If our outcomes are 0–1 (as in failure probabilities), and our initial value for f is less than 1, this means that we reach an outcome with $f = 0$.

The tricky part here is that it may be hard to compute $f(x_1, \dots, x_k)$. (It's always possible to do so in principle by enumerating all assignments of the remaining variables, but if we have time to do this, we can just search for a winning assignment directly.) What makes the method of conditional probabilities practical in many cases is that it's not necessary for f to compute the actual probability of failure, as long as (a) f gives an upper bound on the real probability of failure, at least in terminal configurations, and (b) f has the property used in the argument that for any partial sequence x_1, \dots, x_k there exists an extension x_1, \dots, x_k, x_{k+1} with $f(x_1, \dots, x_k) \geq f(x_1, \dots, x_k, x_{k+1})$. Such an f is called a **pessimistic estimator**. If we can find a pessimistic estimator that is easy to compute and starts out less than 1, then we can just follow it down the tree to a leaf that doesn't fail.

12.4.1 A trivial example

Here is a very bad randomized algorithm for generating a string of n zeros: flip n coins, and output the results. This has a $1 - 2^{-n}$ probability of failure, which is not very good.

We can derandomize this algorithm and get rid of all probability of failure at the same time. Let $f(x_1, \dots, x_k)$ be the probability of failure after the first k bits. Then we can easily compute f (it's 1 if any of the bits are 1 and $1 - 2^{-(n-k)}$ otherwise). We can use this to find a bit x_{k+1} at each stage that reduces f . After noticing that all of these bits are zero, we improve the deterministic algorithm even further by leaving out the computation of f .

12.4.2 Deterministic construction of Ramsey graphs

Here is an example that is slightly less trivial. For this example we let f be a count of bad events rather than a failure probability, but the same method applies.

Recall from §11.1.2 that if $k \geq 3$, for $n \leq 2^{k/2}$ there exists a graph with n nodes and no cliques or independent sets of size k . The proof of this fact is to observe that each subset of k vertices is bad with probability 2^{-k+1} , and when $\binom{n}{k} 2^{-k+1} < 1$, the expected number of bad subsets in $G_{n,1/2}$ is less than 1, showing that some good graph exists.

We can turn this into a deterministic $n^{O(\log n)}$ algorithm for finding a Ramsey graph in the worst case when $n = 2^{k/2}$. The trick is to set the edges to be present or absent one at a time, and for each edge, take the value that minimizes the expected number of bad subsets conditioned on the choices so far. We can easily calculate the conditional probability that

a subset is bad in $O(k)$ time: if it already has both a present and missing edge, it's not bad. Otherwise, if we've already set ℓ of its edges to the same value, it's bad with probability exactly $2^{-k+\ell}$. Summing this over all $O(n^k)$ subsets takes $O(n^k k)$ time per edge, and we have $O(n^2)$ edges, giving $O(n^{k+2}k) = O\left(n^{(2\lg n + 2 + \lg \lg n)}\right) = n^{O(\log n)}$ time total.

It's worth mentioning that there are better deterministic constructions in the literature. The best construction that I am aware of is given by an algorithm of Barak *et al.* [BRSW06], which constructs a graph with $k^{\log^c k}$ vertices with no clique or independent set of size k for any fixed c .

12.4.3 MAX CUT using conditional probabilities

Again we consider the algorithm from §11.2.1 that assigns vertices to S and T at random. To derandomize this algorithm, at each step we pick a vertex and assign it to the side of the cut that maximizes the conditional expectation of the number of edges that cross the cut. We can compute this conditional expectation as follows:

1. For any edge that already has both endpoints assigned to S or T , it's either in the cut or it isn't: add 0 or 1 as appropriate to the conditional expectation.
2. For any edge with only one endpoint assigned, there's a $1/2$ probability that the other endpoint gets assigned to the other side (in the original randomized algorithm). Add $1/2$ to the conditional expectation for these edges.
3. For any edge with neither endpoint assigned, we again have a $1/2$ probability that it crosses the cut. Add $1/2$ for these as well.

So now let us ask how assigning a particular previously unassigned vertex v to S or T affects the conditional probability. For any neighbor w of v that is not already assigned, adding v to S or T doesn't affect the $1/2$ contribution of vw . So we can ignore these. The only effects we see are that if some neighbor w is in S , assigning v to S decreases the conditional expectation by $1/2$ and assigning v to T increases the expectation by $1/2$. So to maximize the conditional expectation, we should assign v to whichever side currently holds fewer of v 's neighbors—the obvious greedy algorithm, which runs in $O(n + m)$ time if we are reasonably clever about keeping track of how many neighbors each unassigned node has in S and T . The advantage of the method of conditional probabilities here is that we immediately get that

the greedy algorithm achieves a cut of size $m/2$, which might require actual intelligence to prove otherwise.

12.4.4 Set balancing

Here we have a collection of vectors v_1, v_2, \dots, v_n in $\{0, 1\}^m$. We'd like to find ± 1 coefficients $\epsilon_1, \epsilon_2, \dots, \epsilon_n$ that minimize $\max_j |X_j|$ where $X_j = \sum_{i=1}^n \epsilon_i v_{ij}$.

If we choose the ϵ_i randomly, Hoeffding's inequality (5.3.1) says for each fixed j that $\Pr[|X_j| > t] < 2 \exp(-t^2/2n)$ (since there are at most n non-zero values v_{ij}). Setting $2 \exp(-t^2/2n) \leq 1/m$ gives $t \geq \sqrt{2n \ln 2m}$. So by the union bound, we have $\Pr[\max_j |X_j| > \sqrt{2n \ln 2m}] < 1$: a solution exists.

Because there may be very complicated dependence between the X_j , it is difficult to calculate the probability of the event $\bigcup_j [|X_j| \geq t]$, whether conditioned on some of the ϵ_i or not. However, we can calculate the probability of the individual events $[|X_j| \geq t]$ exactly. Conditioning on $\epsilon_1, \dots, \epsilon_k$, the expected value of X_j is just $\ell = \sum_{i=1}^k \epsilon_i v_{ij}$, and the distribution of $Y = X_j - \ell$ is the sum $r \leq n - k$ independent ± 1 random variables. So

$$\begin{aligned} \Pr[|X_j| > t \mid \epsilon_1, \dots, \epsilon_k] &= 1 - \Pr[-t - \ell \leq Y \leq t - \ell] \\ &= \sum_{i=-t-\ell}^{t-\ell} \binom{r}{i} 2^{-r}. \end{aligned}$$

This last expression involves a linear number of terms, each of which we can calculate using a linear number of operations on rational numbers that fit in a linear number of bits, so we can calculate the probability exactly in polynomial time by just adding them up.

For our pessimistic estimator, we take

$$U(\epsilon_1, \dots, \epsilon_k) = \sum_{i=j}^n \Pr[|X_j| > \sqrt{2n \ln 2m} \mid \epsilon_1, \dots, \epsilon_k].$$

Note that we can calculate each term in the sum by adding up a big pile of binomial coefficients. Furthermore, since each term in the sum is a Doob martingale, the sum is a martingale as well, so $\mathbb{E}[U(\epsilon_1, \dots, \epsilon_{k+1}) \mid \epsilon_1, \dots, \epsilon_k] = U(\epsilon_1, \dots, \epsilon_k)$. It follows that for any choice of $\epsilon_1, \dots, \epsilon_k$ there exists some ϵ_{k+1} such that $U(\epsilon_1, \dots, \epsilon_k) \geq U(\epsilon_1, \dots, \epsilon_{k+1})$, and we can determine this winning ϵ_{k+1} explicitly. Our previous argument shows that $U(\langle \rangle) < 1$, which implies that our final value $U(\epsilon_1, \dots, \epsilon_n)$ will also be less than 1. Since U is an integer, this means it must be 0, and we find an assignment in which $|X_j| < \sqrt{2n \ln 2m}$ for all j .

Chapter 13

Quantum computing

Last updated 2019. Some material may be out of date.

Quantum computing is a currently almost-entirely-theoretical branch of randomized algorithms that attempts to exploit the fact that probabilities at a microscopic scale arise in a mysterious way from more fundamental **probability amplitudes**, which are complex-valued and can cancel each other out where probabilities can only add. In a quantum computation, we replace random bits with **quantum bits**—**qubits** for short—and replace random updates to the bits with quantum operations.

To explain how this works, we'll start by re-casting our usual model of a randomized computation to make it look more like the standard **quantum circuits** of Deutsch [Deu89]. We'll then get quantum computation by replacing all the real-valued probabilities with complex-valued amplitudes.

13.1 Random circuits

Let's consider a very simple randomized computer whose memory consists of two bits. We can describe our knowledge of the state of this machine using a vector of length 4, with the coordinates in the vector giving the probability of states 00, 01, 10, and 11. For example, if we know that the initial state is always 00, we would have the (column) vector

$$\begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}.$$

Any such **state vector** x for the system must consist of non-negative real values that sum to 1; this is just the usual requirement for a discrete probability space. Operations on the state consist of taking the old values of one or both bits and replacing them with new values, possibly involving both randomness and dependence on the old values. The law of total probability applies here, so we can calculate the new state vector x' from the old state vector x by the rule

$$x'_{b'_1 b'_2} = \sum_{b_1 b_2} x_{b_1 b_2} \Pr [X_{t+1} = b'_1 b'_2 \mid X_t = b_1 b_2].$$

These are linear functions of the previous state vector, so we can summarize the effect of our operation using a transition matrix A , where $x' = Ax$.¹

We imagine that these operations are carried out by feeding the initial state into some circuit that generates the new state. This justifies the calling this model of computation a **random circuit**. But the actual implementation might be an ordinary computer than is just flipping coins. If we can interpret each step of the computation as applying a transition matrix, the actual implementation doesn't matter.

For example, if we negate the second bit 2/3 of the time while leaving the first bit alone, we get the matrix

$$A = \begin{bmatrix} 1/3 & 2/3 & 0 & 0 \\ 2/3 & 1/3 & 0 & 0 \\ 0 & 0 & 1/3 & 2/3 \\ 0 & 0 & 2/3 & 1/3 \end{bmatrix}.$$

One way to derive this matrix other than computing each entry directly is that it is the **tensor product** of the matrices that represent the operations on each individual bit. The idea here is that the tensor product of A and B , written $A \otimes B$, is the matrix C with $C_{ij,kl} = A_{ik} B_{jl}$. We're cheating a little bit by allowing the C matrix to have indices consisting of pairs of indices, one for each of A and B ; there are more formal definitions that justify this at the cost of being harder to understand.

¹Note that this is the reverse of the convention we adopted for Markov chains in Chapter 9. There it was convenient to have $P_{ij} = p_{ij} = \Pr [X_{t+1} = j \mid X_t = i]$. Here we defer to the physicists and make the update operator come in front of its argument, like any other function.

In this particular case, we have

$$\begin{bmatrix} 1/3 & 2/3 & 0 & 0 \\ 2/3 & 1/3 & 0 & 0 \\ 0 & 0 & 1/3 & 2/3 \\ 0 & 0 & 2/3 & 1/3 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \otimes \begin{bmatrix} 1/3 & 2/3 \\ 2/3 & 1/3 \end{bmatrix}.$$

The first matrix in the tensor product gives the update rule for the first bit (the identity matrix—do nothing), while the second gives the update rule for the second.

Some operations are not decomposable in this way. If we swap the values of the two bits, we get the matrix

$$S = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix},$$

which maps 00 and 11 to themselves but maps 01 to 10 and vice versa.

The requirement for all of these matrices is that they be **stochastic**. This means that each column has to sum to 1, or equivalently that $1A = 1$, where 1 is the all-ones vector. This just says that our operations map probability distributions to probability distributions; we don't apply an operation and find that the sum of our probabilities is now $3/2$ or something. (Proof: If $1A = 1$, then $|Ax|_1 = 1(Ax) = (1A)x = 1x = |x|_1$.)

A **randomized computation** in this model now consists of a sequence of these stochastic updates to our random bits, and at the end performing a **measurement** by looking at what the values of the bits actually are. If we want to be mystical about it, we could claim that this measurement collapses a probability distribution over states into a single unique state, but really we are just opening the box to see what we got.

For example, we could generate two bias-2/3 coin-flips by starting with 00 and using the algorithm flip second, swap bits, flip second, or in matrix

terms:

$$\begin{aligned}
 x_{\text{out}} &= ASAx_{\text{in}} \\
 &= \begin{bmatrix} 1/3 & 2/3 & 0 & 0 \\ 2/3 & 1/3 & 0 & 0 \\ 0 & 0 & 1/3 & 2/3 \\ 0 & 0 & 2/3 & 1/3 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1/3 & 2/3 & 0 & 0 \\ 2/3 & 1/3 & 0 & 0 \\ 0 & 0 & 1/3 & 2/3 \\ 0 & 0 & 2/3 & 1/3 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} \\
 &= \begin{bmatrix} 1/9 \\ 2/9 \\ 2/9 \\ 4/9 \end{bmatrix}.
 \end{aligned}$$

When we look at the output, we find 11 with probability $4/9$, as we'd expect, and similarly for the other values.

13.2 Bra-ket notation

A notational oddity that scares many people away from quantum mechanics in general and quantum computing in particular is the habit of practitioners in these fields of writing basis vectors and their duals using **bra-ket notation**, a kind of typographical pun invented by the physicist Paul Dirac [Dir39].

This is based on a traditional way of writing an **inner product** of two vectors x and y in “bracket form” as $\langle x|y\rangle$. The interpretation of this is $\langle x|y\rangle = x^*y$, where x^* is the **conjugate transpose** of x .

For real-valued x the conjugate transpose is the same as the transpose. For complex-valued x , each coordinate $x_i = a + bi$ is replaced by its **complex conjugate** $\bar{x}_i = a - bi$.) Using the conjugate transpose makes $\langle x|x\rangle$ equal $|x|_2^2$ when x is complex-valued.

For example, for our vector x_{in} above that puts all of its probability on 00, we have

$$\langle x_{\text{in}}|x_{\text{in}}\rangle = \begin{bmatrix} 1 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} = 1. \quad (13.2.1)$$

The typographic trick is to split in half both $\langle x|y\rangle$ and its expansion. For

example, we could split (13.2.1) as

$$\langle x_{\text{in}}| = [1 \quad 0 \quad 0 \quad 0] \qquad |x_{\text{in}}\rangle = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}.$$

In general, wherever we used to have a bracket $\langle x|y\rangle$, we now have a **bra** $\langle x|$ and a **ket** $|y\rangle$. These are just row vectors and column vectors, and $\langle x|$ is always the conjugate transpose of $|x\rangle$.

The second trick in bra-ket notation is to make the contents of the bra or ket an arbitrary name. For kets, this will usually describe some state. As an example, we might write x_{in} as $|00\rangle$ to indicate that it's the basis vector that puts all of its weight on the state 00. For bras, this is the linear operator that returns 1 when applied to the given state and 0 when applied to any orthogonal state. So $\langle 00|00\rangle = \langle 00|00\rangle = 1$ but $\langle 00|01\rangle = \langle 00|01\rangle = 0$.

13.2.1 States as kets

This notation is useful for the same reason that variable names are useful. It's a lot easier to remember that $|01\rangle$ refers to the distribution assigning probability 1 to state 01 than that $[0 \quad 1 \quad 0 \quad 0]^\top$ is.

Other vectors can be expressed using a linear combination of kets. For example, we can write

$$x_{\text{out}} = \frac{1}{9}|00\rangle + \frac{2}{9}|01\rangle + \frac{2}{9}|10\rangle + \frac{4}{9}|11\rangle.$$

This is not as compact as just writing out the vector as a matrix, but it has the advantage of clearly labeling what states the probabilities apply to.

13.2.2 Operators as sums of kets times bras

A similar trick can be used to express operators, like the swap operator S . We can represent S as a combination of maps from specific states to specific other states. For example, the operator

$$|01\rangle\langle 10| = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} [0 \quad 0 \quad 1 \quad 0] = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

maps $|10\rangle$ to $|01\rangle$ (Proof: $|01\rangle\langle 10| |10\rangle = |01\rangle\langle 10|10\rangle = |01\rangle$) and sends all other states to 0. Add up four of these mappings to get

$$S = |00\rangle\langle 00| + |10\rangle\langle 01| + |01\rangle\langle 10| + |11\rangle\langle 11| = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

Here the bra-ket notation both labels what we are doing and saves writing a lot of zeros.

The intuition is that just like a ket represents a state, a bra represents a test for being in that state. So something like $|01\rangle\langle 10|$ tests if we are in the 10 state, and if so, sends us to the 01 state.

13.3 Quantum circuits

So how do we turn our random circuits into **quantum circuits**?

The first step is to replace our random bits with quantum bits (qubits).

For a single random bit, the state vector represents a probability distribution

$$p_0 |0\rangle + p_1 |1\rangle,$$

where p_0 and p_1 are non-negative real numbers with $p_0 + p_1 = 1$.

For a single qubit, the state vector represents amplitudes

$$a_0 |0\rangle + a_1 |1\rangle,$$

where a_0 and a_1 are complex numbers with $|a_0|^2 + |a_1|^2 = 1$.² The reason for this restriction on amplitudes is that if we measure a qubit, we will see state 0 with probability $|a_0|^2$ and state 1 with probability $|a_1|^2$. Unlike with random bits, these probabilities are not mere expressions of our ignorance but arise

²The **absolute value, norm, or magnitude** $|a + bi|$ of a complex number is given by $\sqrt{a^2 + b^2}$. When $b = 0$, this is the same as the absolute value for the corresponding real number. For any complex number x , the norm can also be written as $\sqrt{\bar{x}x}$, where \bar{x} is the complex conjugate of x . This is because $\sqrt{(a + bi)(a - bi)} = \sqrt{a^2 - (bi)^2} = \sqrt{a^2 + b^2}$. The appearance of the complex conjugate here explains why we define $\langle x|y\rangle = x^*y$; the conjugate transpose means that for $\langle x|x\rangle$, when we multiply x_i^* by x_i we are computing a squared norm.

through a still somewhat mysterious process from the more fundamental amplitudes.³

With multiple bits, we get amplitudes for all combinations of the bits, e.g.

$$\frac{1}{2} (|00\rangle + |01\rangle + |10\rangle + |11\rangle)$$

gives a state vector in which each possible measurement will be observed with equal probability $\left(\frac{1}{2}\right)^2 = \frac{1}{4}$. We could also write this state vector as

$$\begin{bmatrix} 1/2 \\ 1/2 \\ 1/2 \\ 1/2 \end{bmatrix}.$$

13.3.1 Quantum operations

In the random circuit model, at each step we pick a small number of random bits, apply a stochastic transformation to them, and replace their values with the results. In the quantum circuit model, we do the same thing, but now our transformations must have the property of being **unitary**. Just as a stochastic matrix preserves the property that the probabilities in a state vector sum to 1, a **unitary matrix** preserves the property that the squared norms of the amplitudes in a state vector sum to 1.

Formally, a square, complex matrix A is unitary if it preserves inner products: $\langle Ax|Ay\rangle = \langle x|y\rangle$ for all x and y . Alternatively, A is unitary if $A^*A = AA^* = I$, where A^* is the conjugate transpose of A , because if this holds, $\langle Ax|Ay\rangle = (Ax)^*(Ay) = x^*A^*Ay = x^*Iy = x^*y = \langle x|y\rangle$. Yet another

³In the old days of “shut up and calculate,” this process was thought to involve the unexplained power of a conscious observer to collapse a superposition into a classical state. Nowadays the most favored explanation involves **decoherence**, the difficulty of maintaining superpositions in systems that are interacting with large, warm objects with lots of thermodynamic degrees of freedom (measuring instruments, brains). The decoherence explanation is particularly useful for explaining why real-world quantum computers have a hard time keeping their qubits mixed even when nobody is looking at them. Decoherence by itself does not explain *which* basis states a system collapses to. Since bases in linear algebra are pretty much arbitrary, it would seem that we could end up running into a physics version of Goodman’s grue-bleen paradox [Goo83], but there are apparently ways of dealing with this too using a mechanism called **einselection** [Zur03], that favors classical states over weird ones. Since all of this is (a) well beyond my own limited comprehension of quantum mechanics and (b) irrelevant to the theoretical model we are using, these issues will not be discussed further.

way to state this is that the columns of A form an orthonormal basis: this means that $\langle A_i | A_j \rangle = 0$ if $i \neq j$ and 1 if $i = j$, which is just a more verbose way to say $A^* A = I$. The same thing also works if we consider rows instead of columns.

The rule then is: at each step, we can operate on some constant number of qubits by applying a unitary transformation to them. In principle, this could be *any* unitary transformation, but some particular transformations show up often in actual quantum algorithms.⁴

The simplest unitary transformations are permutations on states (like the operator that swaps two qubits), and rotations of a single state. One particularly important rotation is the **Hadamard operator**

$$H = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}.$$

This maps $|0\rangle$ to the superposition $\frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle$; since this superposition collapses to either $|0\rangle$ or $|1\rangle$ with probability $1/2$, the state resulting from $H|0\rangle$ is the quantum-computing equivalent of a fair coin-flip. Note that $H|1\rangle = \frac{1}{\sqrt{2}}|0\rangle - \frac{1}{\sqrt{2}}|1\rangle \neq H|0\rangle$. Even though both yield the same probabilities; these two superpositions have different **phases** and may behave differently when operated on further. That $H|0\rangle$ and $H|1\rangle$ are different is necessary, and indeed a similar outcome occurs for any quantum operation: all quantum operations are **reversible**, because any unitary matrix U has an inverse U^* .

If we apply H in parallel to all the qubits in our system, we get the n -fold tensor product $H^{\otimes n}$, which (if we take our bit-vector indices as integers $0 \dots N - 1 = 2^n - 1$ represented in binary) maps $|0\rangle$ to $\frac{1}{\sqrt{N}} \sum_{i=0}^{N-1} |i\rangle$. So n applications of H effectively scramble a deterministic initial state into a uniform distribution across all states. We'll see this scrambling operation again when we look at Grover's algorithm in §13.5.

13.3.2 Quantum implementations of classical operations

One issue that comes up with trying to implement classical algorithms in the quantum-circuit model is that classical operations are generally not reversible: if I execute $x \leftarrow x \wedge y$, it may not be possible to reconstruct the old state of x . So I can't implement AND directly as a quantum operation.

⁴Deutsch's original paper [Deu89] shows that repeated applications of single-qubit rotations and the CNOT operation (described in §13.3.2) are enough to approximate any unitary transformation.

The solution is to use more sophisticated reversible operations from which standard classical operations can be extracted as a special case. A simple example is the **controlled NOT** or **CNOT** operator, which computes the mapping $(x, y) \mapsto (x, x \oplus y)$. This corresponds to the matrix (over the basis $|00\rangle, |01\rangle, |10\rangle, |11\rangle$)

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix},$$

which is clearly unitary (the rows are just the standard basis vectors). We could also write this more compactly as $|00\rangle\langle 00| + |01\rangle\langle 01| + |11\rangle\langle 10| + |10\rangle\langle 11|$.

The CNOT operator gives us XOR, but for more destructive operations we need to use more qubits, possibly including junk qubits that we won't look at again but that are necessary to preserve reversibility. The **Toffoli gate** or **controlled controlled NOT** gate (**CCNOT**) is a 3-qubit gate that was originally designed to show that classical computation could be performed reversibly [Tof80]. It implements the mapping $(x, y, z) \mapsto (x, y, (x \wedge y) \oplus z)$, which corresponds to the 8×8 matrix

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}.$$

By throwing in some extra qubits we don't care about, Toffoli gates can implement basic operations like NAND $((x, y, 1) \mapsto (x, y, \neg(x \wedge y)))$, NOT $((x, 1, 1) \mapsto (x, 1, \neg x))$, and fan-out $((x, 1, 0) \mapsto (x, 1, x))$.⁵ This gives

⁵In the case of fan-out, this only works with perfect accuracy for classical bits and not superpositions, which run into something called the **no-cloning theorem**. For example, applying CCNOT to $\frac{1}{\sqrt{2}}|010\rangle + \frac{1}{\sqrt{2}}|110\rangle$ yields $\frac{1}{\sqrt{2}}|010\rangle + \frac{1}{\sqrt{2}}|111\rangle$. This works, sort of, but the problem is that the first and last bits are still entangled, meaning we can't operate on them independently. This is actually not all that different from what happens in the probabilistic case (if I make a copy of a random variable X , it's correlated with the original X), but it has good or bad consequences depending on whether you want to prevent people from stealing your information undetected or run two copies of a quantum circuit on independent replicas of a single superposition.

a sufficient basis for implementing all classical circuits.

13.3.3 Representing Boolean functions

Suppose we have a quantum circuit that computes a Boolean function f . There are two conventions for representing the output of f :

1. We can represent $f(x)$ by XORing it with an extra qubit y : $|x, y\rangle \mapsto |x, y \oplus f(x)\rangle$. This is the approach taken by the CNOT ($f(x) = x$) and CCNOT ($f(x_1x_2) = x_1 \wedge x_2$) gates.
2. We can represent $f(x)$ by changing the phase of $|x\rangle$: $|x\rangle \mapsto (-1)^{f(x)} |x\rangle$. The actual unitary operator corresponding to this is $\sum_x (-1)^{f(x)} |x\rangle \langle x|$, which in matrix form looks like a truth table for f expressed as ± 1 values along the diagonal, e.g.:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

for the XOR function.

This has the advantage of requiring one fewer qubit, but we can't observe the value of $f(x)$ directly, because the amplitude of $|x\rangle$ is unchanged. Where this comes in handy is when we can use the change in phase in the context of a larger quantum algorithm to get cancellations for some values of x .

The first representation makes more sense for modeling classical circuits. The second turns out to be more useful when f is a subroutine in a larger quantum algorithm. Fortunately, the operator with matrix

$$\begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$$

converts the $|0\rangle-|1\rangle$ representation into the ± 1 representation, so we can build any desired classical f using $|0\rangle-|1\rangle$ logic and convert it to ± 1 as needed.

13.3.4 Practical issues (which we will ignore)

The big practical question is whether any of these operations—or even non-trivial numbers of independently manipulable qubits—can be implemented

in a real, physical system. As theorists, we can ignore these issues, but in real life they are what would make quantum computing science instead of science fiction.⁶

13.3.5 Quantum computations

Putting everything together, a quantum computation consists of three stages:

1. We start with a collection of qubits in some known state x_0 (e.g., $|000\dots 0\rangle$).
2. We apply a sequence of unitary operators A_1, A_2, \dots, A_m to our qubits.
3. We take a measurement of the final superposition $A_m A_{m-1} \dots A_1 x_0$ that collapses it into a single state, with probability equal to the square of the amplitude of that state.

Our goal is for this final state to tell us what we want to know, with reasonably high probability.

13.4 Deutsch's algorithm

We now have enough machinery to describe a real quantum algorithm. Known as Deutsch's algorithm, this computes $f(0) \oplus f(1)$ while evaluating f once [Deu89]. The trick, of course, is that f is applied to a superposition.

Assumption: f is implemented reversibly, as a quantum computation that maps $|x\rangle$ to $(-1)^{f(x)} |x\rangle$. To compute $f(0) \oplus f(1)$, evaluate

$$\begin{aligned} HfH|0\rangle &= \sqrt{\frac{1}{2}} Hf(|0\rangle + |1\rangle) \\ &= \sqrt{\frac{1}{2}} H\left((-1)^{f(0)}|0\rangle + (-1)^{f(1)}|1\rangle\right) \\ &= \frac{1}{2} \left(\left((-1)^{f(0)} + (-1)^{f(1)}\right)|0\rangle + \left((-1)^{f(0)} - (-1)^{f(1)}\right)|1\rangle \right). \end{aligned}$$

Suppose now that $f(0) = f(1) = b$. Then the $|1\rangle$ terms cancel out and we are left with

$$\frac{1}{2} \left(2 \cdot (-1)^b |0\rangle \right) = (-1)^b |0\rangle.$$

⁶Current technology, sadly, still puts quantum computing mostly in the science fiction category.

This puts all the weight on $|0\rangle$, so when we take our measurement at the end, we'll see 0.

Alternatively, if $f(0) = b \neq f(1)$, it's the $|0\rangle$ terms that cancel out, leaving $(-1)^b |1\rangle$. Again the phase depends on b , but we don't care about the phase: the important thing is that if we measure the qubit, we always see 1.

The result in either case is that with probability 1, we determine the value of $f(0) \oplus f(1)$, after evaluating f once (albeit on a superposition of quantum states).

This is kind of a silly example, because the huge costs involved in building our quantum computer probably swamp the factor-of-2 improvement we got in the number of calls to f . But a generalization of this trick, known as the Deutsch-Josza algorithm [DJ92], solves the much harder (although still a bit contrived-looking) problem of distinguishing a constant Boolean function on n bits from a function that outputs one for exactly half of its inputs. No deterministic algorithm can solve this problem without computing at least $2^n/2 + 1$ values of f , giving an exponential speed-up.⁷

13.5 Grover's algorithm

Grover's algorithm [Gro96] is one of two main exemplars for the astonishing power of quantum computers.⁸ The idea of Grover's algorithm is that if we have a function f on $N = 2^n$ possible inputs whose value is 1 for exactly one possible input w , we can find this w with high probability using $O(\sqrt{N})$ quantum evaluations of f . As with Deutsch's algorithm, we assume that f is encoded as an operator (conventionally written U_w) that maps each $|x\rangle$ to $(-1)^{f(x)} |x\rangle$.

The basic outline of the algorithm:

1. Start in the superposition $|s\rangle = \frac{1}{\sqrt{N}} \sum_x |x\rangle = H^{\otimes n} |0\rangle$.
2. Alternate between applying the **Grover diffusion operator** $D = 2|s\rangle\langle s| - I$ and the f operator $U_w = 2|w\rangle\langle w| - I$. Do this $O(\sqrt{n})$ times (the exact number of iterations is important and will be calculated below).

⁷The speed-up compared to a randomized algorithm that works with probability $1 - \epsilon$ is less impressive. With randomization, we only need to look at $O(\log 1/\epsilon)$ values of f to see both a 0 and a 1 in the non-constant case. But even here, the Deutsch-Josza algorithm does have the advantage of giving the correct answer always.

⁸The other is Shor's algorithm [Sho97], which allows a quantum computer to factor n -bit integers in time polynomial in n . Sadly, Shor's algorithm is a bit too complicated to talk about here.

3. Take a measurement of the state. It will be w with high probability.

Making this work requires showing that (a) we can generate the original superposition $|s\rangle$, (b) we can implement D efficiently using unitary operations on a constant number of qubits each, and (c) we actually get w at the end of this process.

13.5.1 Initial superposition

To get the initial superposition, start with $|0^n\rangle$ and apply the Hadamard transform to each bit individually; this gives $\frac{1}{\sqrt{N}} \sum_x |x\rangle$ as claimed.

13.5.2 The Grover diffusion operator

We have the definition $D = 2|s\rangle\langle s| - I$.

Before we try to implement this, let's start by checking that it is in fact unitary. Compute

$$\begin{aligned} DD^* &= (2|s\rangle\langle s| - I)^2 \\ &= 4|s\rangle\langle s||s\rangle\langle s| - 4|s\rangle\langle s| + I^2 \\ &= 4|s\rangle\langle s| - 4|s\rangle\langle s| + I \\ &= I. \end{aligned}$$

Here we use the fact that $|s\rangle\langle s||s\rangle\langle s| = |s\rangle\langle s|s\rangle\langle s| = |s\rangle(1)\langle s| = |s\rangle\langle s|$.

Recall that $|s\rangle = H^{\otimes n}|0^n\rangle$, where $H^{\otimes n}$ is the result of applying H to each of the n bits individually. We also have that $H^* = H$ and $HH^* = I$, from which $H^{\otimes n}H^{\otimes n} = I$ as well.

So we can expand

$$\begin{aligned} D &= 2|s\rangle\langle s| - I \\ &= 2H^{\otimes n}|0^n\rangle\langle 0^n|H^{\otimes n} - I \\ &= 2H^{\otimes n}|0^n\rangle\langle 0^n|H^{\otimes n} - I \\ &= H^{\otimes n}(2|0^n\rangle\langle 0^n| - I)H^{\otimes n}. \end{aligned}$$

The two copies of $H^{\otimes n}$ involve applying H to each of the n bits, which we can do. The operator in the middle, $2|0^n\rangle\langle 0^n| - I$, maps $|0^n\rangle$ to $|0^n\rangle$ and maps all other basis vectors $|x\rangle$ to $-|x\rangle$. This can be implemented as a NOR of all the qubits, which we can do using our tools for classical computations. So the entire operator D can be implemented using $O(n)$ qubit operations, most of which can be done in parallel.

13.5.3 Effect of the iteration

To see what happens when we apply $U_w D$, it helps to represent the state in terms of a particular two-dimensional basis. The idea here is that the initial state $|s\rangle$ and the operation $U_w D$ are symmetric with respect to any basis vectors $|x\rangle$ that aren't $|w\rangle$, so instead of tracking all of these non- $|w\rangle$ vectors separately, we will represent all of them by a single composite vector

$$|u\rangle = \sqrt{\frac{1}{N-1}} \sum_{x \neq w} |x\rangle.$$

The coefficient $\sqrt{\frac{1}{N-1}}$ is chosen to make $\langle u|u\rangle = 1$. As always, we like our vectors to have length 1.

Using $|u\rangle$, we can represent

$$|s\rangle = \sqrt{\frac{1}{N}} |w\rangle + \sqrt{\frac{N-1}{N}} |u\rangle. \quad (13.5.1)$$

A straightforward calculation shows that this indeed puts $\sqrt{\frac{1}{N}}$ amplitude on each $|x\rangle$.

Now we're going to bring in some trigonometry. Let $\theta = \sin^{-1} \sqrt{\frac{1}{N}}$, so that $\sin \theta = \sqrt{\frac{1}{N}}$ and $\cos \theta = \sqrt{1 - \sin^2 \theta} = \sqrt{\frac{N-1}{N}}$. We can then rewrite (13.5.1) as

$$|s\rangle = (\sin \theta) |w\rangle + (\cos \theta) |u\rangle. \quad (13.5.2)$$

Let's look at what happens if we expand D using (13.5.2):

$$\begin{aligned} D &= 2|s\rangle\langle s| - I \\ &= 2((\sin \theta) |w\rangle + (\cos \theta) |u\rangle)((\sin \theta) \langle w| + (\cos \theta) \langle u|) - I \\ &= (2 \sin^2 \theta - 1) |w\rangle\langle w| + (2 \sin \theta \cos \theta) |w\rangle\langle u| + (2 \sin \theta \cos \theta) |u\rangle\langle w| + (2 \cos^2 \theta - 1) |u\rangle\langle u| \\ &= (-\cos 2\theta) |w\rangle\langle w| + (\sin 2\theta) |w\rangle\langle u| + (\sin 2\theta) |u\rangle\langle w| + (\cos 2\theta) |u\rangle\langle u| \\ &= \begin{bmatrix} -\cos 2\theta & \sin 2\theta \\ \sin 2\theta & \cos 2\theta \end{bmatrix}, \end{aligned}$$

where the matrix is over the basis $(|w\rangle, |u\rangle)$.

Multiplying by U_w negates all the $|w\rangle$ coefficients. So we get

$$\begin{aligned} U_w D &= \begin{bmatrix} -1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} -\cos 2\theta & \sin 2\theta \\ \sin 2\theta & \cos 2\theta \end{bmatrix} \\ &= \begin{bmatrix} \cos 2\theta & -\sin 2\theta \\ \sin 2\theta & \cos 2\theta \end{bmatrix}. \end{aligned} \tag{13.5.3}$$

Aficionados of computer graphics, robotics, or just matrix algebra in general may recognize (13.5.3) as the matrix that rotates two-dimensional vectors by 2θ . Since we started with $|s\rangle$ at an angle of θ , after t applications of this matrix we will be at an angle of $(2t + 1)\theta$, or in state

$$(\sin(2t + 1)\theta) |w\rangle + (\cos(2t + 1)\theta) |u\rangle.$$

Ideally, we pick t so that $(2t + 1)\theta = \pi/2$, which would put all of the amplitude on $|w\rangle$. Because t is an integer, we can't do this exactly, but setting $t = \lfloor \frac{\pi/2\theta - 1}{2} \rfloor$ will get us somewhere between $\pi/2 - 2\theta$ and $\pi/2$. Since $\theta \approx \sqrt{\frac{1}{N}}$, this gives us a probability of seeing $|w\rangle$ in our final measurement of $1 - O(\sqrt{1/N})$ after $O(\sqrt{N})$ iterations of $U_w D$.

Sadly, this is as good as it gets. A lower bound of Bennet *et al.* [BBBV97] shows that *any* quantum algorithm using U_w as the representation for f must apply U_w at least $\Omega(\sqrt{N})$ times to find w . So we get a quadratic speedup but not the exponential speedup we'd need to solve **NP**-complete problems directly.

Chapter 14

Randomized distributed algorithms

Last updated 2019. Some material may be out of date.

A **distributed algorithm** is one that runs on multiple machines that communicate with each other in some way, typically via **message-passing** (an abstraction of packet-based computer networks) or **shared memory** (an abstraction of multi-core CPUs and systems with a common memory bus). What generally distinguishes **distributed computing** from the closely-related idea of **parallel computing** is that we expect a lot of nondeterminism in a distributed algorithm: events take place at unpredictable times, processes may crash, and in particular bad cases we may have **Byzantine processes** that work deliberately against the algorithm. We are not going to assume Byzantine processes; instead, we'll look at a particular model called **wait-free shared memory**.

This hostile nondeterminism is modeled by an **adversary** that controls scheduling and failures. For a shared-memory model, we have a collection of processes that each have a **pending operation** that is either a read or write to some **register**. The adversary chooses which of these pending operations happens next (so that concurrency between processes is modeled by interleaving their operations). Unlike the adversary that supplies the worst-case input to a traditional algorithm before it executes, an adversary scheduler might be able to observe the execution of a distributed algorithm in progress and adjust its choices in response. If it has full knowledge of the system (including internal states of processes), we call it an **adaptive adversary**; at the other extreme is an **oblivious adversary** that chooses the schedule of which processes execute operations at which times in advance.

This is similar to the distinction between an adaptive and oblivious adversary required for the analysis of randomized data structures (see §6.3.1).

For either adversary, what makes the system **wait-free** is that any process that gets to take an unbounded number of steps has to finish whatever it is doing no matter how the other processes are scheduled. In particular, this means that no process can wait for any of the others, because they might never be scheduled. The processes that do run all run the algorithm correctly, so the model is equivalent to assuming up to $n - 1$ of the processes may **crash** but that none are Byzantine.

As with traditional algorithms, distributed algorithms can use randomization to make themselves less predictable. The assumption is that even if the adversary is adaptive, it can't predict the outcome of future coin-flips. For some problems, avoiding such predictability is provably necessary.

Distributed computing is a huge field, and we aren't going to be able to cover much of it in the limited space we have here. So we are going to concentrate on a few simple problems that give some of the flavor or randomized distributed algorithms (and that lead to problems that are still open).

14.1 Consensus

The **consensus** problem is to get a collection of n processes to agree on a value. The requirements are that all the processes that don't crash finish the protocol (with probability 1 for randomized protocols) (**termination**), that they all output the same value (**agreement**), and that this value was an input to one of the processes (**validity**)—this last condition excludes protocols that always output the same value no matter what happens during the execution, and makes consensus useful for choosing among values generated by some other process.

There are many versions of consensus. The original problem as proposed by Pease, Shostak, and Lamport [PSL80] assumes Byzantine processes in a synchronous message-passing system. Here scheduling is entirely predictable, and the obstacle to agreement is dissension sown by the lying Byzantine processes. We will instead consider wait-free shared-memory consensus, where scheduling is unpredictable but the processes and memory are trustworthy. Even in this case, the unpredictable scheduling makes solving the problem deterministically impossible.

14.1.1 Impossibility of deterministic algorithms

The FLP impossibility result [FLP85] and its extension to shared memory [LAA87] show that consensus is impossible to solve in an asynchronous model with even one crash failure. The general proof is a bit involved, but there is a simple intuitive proof of this result for the wait-free where up to $n - 1$ of the processes may crash.

Crash all but two processes, one with input 0 and one with input 1. Define the **preference** of a process as the value it will decide in a solo execution (this is well-defined because the processes are deterministic). In the initial state, the process with input b has preference b because of validity—it doesn't know the other process exists. But before it returns b , it has to cause the other process to change its preference (once it leaves the other process is running alone). It can do so only by sending a message or writing a register. When it is about to do this, stop it and run the other process until it does the same thing.

Either the other process somehow neutralizes the effect of the delayed message/write during this time, or it doesn't. In the first case, restart the first process (which still has its original preference and now must do something else to make the other process change). In the second case, deliver both operations and let the processes exchange preferences. The resulting execution looks very much like when two people are trying to pass each other in a hallway and oscillate back-and-forth—but since our process's have their timing controlled by an adversary, the natural damping or randomness that occurs in humans doesn't ever resolve the situation.

We can avoid this impossibility result with randomness. If either of the process decides to choose a new preference at random, then there is a 50% chance it will end up agreeing with the other one, no matter what the other one is doing. Assuming we can detect this agreement, this solves consensus in constant expected time for two processes.

Unfortunately, this flip-if-confused approach does not generalize very well to n processes. The first known randomized shared-memory algorithm for consensus, due to Abrahamson [Abr88], used essentially this idea. But because it required waiting for a long run of identical coin-flips across all processes, it took $O(2^{n^2})$ steps on average to finish. In a little while, we will see how to do better.

14.2 Leader election

In a shared-memory system, **leader election** is a very similar problem to consensus. Here we want one of the processes to decide that it is the leader, and the rest to decide that they are followers. The difference from consensus is that we don't require that the followers learn the identity of the leader. This means that followers can drop out before the algorithm determines the leader, which may allow a faster algorithm.

Note however that if there are only two processes, the losing process knows who the winner is, because there is only one possibility. So the same argument that shows that we can't do consensus deterministically with two processes also works for leader election.

14.3 How randomness helps

We've already hinted at the idea of using randomness to shake processes into agreement. In the current literature on shared-memory consensus and leader election, this idea shows up in two main forms:

1. We can replace the n separate coin-flips of stalled processes with a single **weak shared coin** protocol. This is a protocol that has for each outcome $b \in \{0, 1\}$ a minimum probability $\delta > 0$ that every process receives b as the value of the coin. Because δ will typically be less than $1/2$, it is possible that we get disagreements, or that the adversary can bias the coin toward a particular value that it likes. But with a bit of extra machinery that we will not describe here, we can get agreement after $O(1/\delta)$ calls to the shared coin on average [AH90]. An example of a weak shared coin protocol that assumes an adaptive adversary is given in §14.4.
2. We can eliminate processes quickly using a **sifter**, which solves a weak version of leader election that allows for multiple winners but guarantees that the expected number of winners is small relative to n . Repeated application of a sifter can quickly knock the number of repeated winners down to 1, which solves leader election (§14.5). With some tinkering, it is possible to use some sifters to also solve consensus (§14.6). In both cases we assume an oblivious adversary.

14.4 Building a weak shared coin

The goal of a weak shared coin is to minimize the influence of the adversary scheduler. An adaptive adversary can bias the outcome of the coin by looking at the random values generated by the individual processes, and withholding values that it doesn't like by delaying any write operations by those processes. So we would like to find a way to combine the local coins at the processes together into a single global coin that minimizes the influence of any particular local coin. The easiest way to do this is by applying the majority function: each process repeatedly generates ± 1 values with equal probability and adds them to a common pile. When the pile includes enough local coins, we take the sign of the total to get the return value of the global coin.

The nice thing about this approach is that the behavior of the processes is symmetric. It doesn't matter what order the adversary runs them in, the next coin-flip is always another fair coin-flip. So we can analyze the sequence of partial sums of *generated* coin-flips using the tools we have developed for sums of independent variables, random walks, and martingales. Unfortunately things get a little more complicated when we look at the totals actually observed by any particular processes.

The problem is that we need some mechanism for gathering up the local coin values. Using read-write registers, we can have each process write the count and sum of its own local coins to a register writable by it alone (this avoids processes overwriting each other). Because the adversary can only delay one coin from each process, both total count and the total sum are always within n of the correct value. If we could read all the registers instantaneously and stop as soon as the count reached some threshold T , then for $T = \Theta(n^2)$ the distribution on the generated coins would be spread out enough that the total would be at least $n + 1$ away from 0 with constant probability. But the actual situation is more complicated.

The problem is that each process must do n sequential reads to read all the registers. This is not only expensive—it's not something we want to do after every local flip—but it also means that more coins may be generated while I'm reading the registers, meaning any sum I compute at the end of the algorithm might have only a tenuous connection with the actual sum of the generated coins at any point in the execution.

The solution to the first part of this problem was proposed by Bracha and Rachman [BR91]: only check the total after writing out $\Theta(n/\log n)$ local coin-flips, giving an amortized cost per coin-flip of $\Theta(\log n)$ register operations. Unfortunately this makes the second problem worse: a process p

might continue generating many local coins after the total count crosses the threshold, and each other process might see a different prefix of these coins depending on when it reads p 's register.

Bracha and Rachman showed that this wasn't too much of a problem using Azuma's inequality (this is where the $O(\log n)$ factor comes in). But later work by Attiya and Censor [AC08] allowed for a simpler analysis of a slightly different algorithm, which we will describe here.

Pseudocode for the Attiya-Censor coin is given in Algorithm 14.1. The algorithm only checks the total count once for every n coin-flips, giving an amortized cost of 1 read per coin-flip. But to keep the processes for running too long, each process checks a multi-writer stop bit after every coin-flip.

```

1 while  $\sum_{j=1}^n r_j.\text{count} < T$  do
2   for  $i \leftarrow 1 \dots n$  do
3      $\langle r_i.\text{count}, r_i.\text{sum} \rangle \leftarrow \langle r_i.\text{count} + 1, r_i.\text{sum} \pm 1 \rangle$ 
4     if stop then
5       break;
6   stop  $\leftarrow$  true
7   return  $\text{sgn}(\sum_{j=1}^n r_j.\text{sum})$ 

```

Algorithm 14.1: Attiya-Censor weak shared coin [AC08]

Each process may see a different total sum at the end, but our hope is that if T is large enough, there is at least a δ probability that all these total sums are positive (or, by symmetry, negative). We can represent the total sum seen by any particular process i as a sum $S + D + X_i - H_i$, where:

1. S is the sum of the first T coin-flips. This has a binomial distribution, equal to the sum of T independent ± 1 random variables. Letting $T = cn^2$ for some reasonably large c gives us a constant probability that $|S| > 4n$.
2. D is the sum of all coin-flips after the first T that are generated before some process sets the stop bit. There are at most $n^2 + n$ such coin-flips, and they form a martingale with bounded increments. So Azuma's inequality gives us that $|D| \leq 2n$ with at least constant probability, independent of S .
3. $X_i = \sum_{j=1}^n Y_{ij}$, where Y_{ij} is the sum of all coin-flips generated by process j between some process setting the stop bit and process i

reading r_j . Since each process can generate at most one extra coin-flip before checking **stop**, $|X_i| \leq n$ always.

4. $H_i = \sum_{j=1}^n Z_{ij}$, where Z_{ij} is the sum of all votes that are generated by process j before i reads r_j , but that are not included in r_j .sum because they haven't been written yet. Again, each process can contribute only one coin-flip to H_i . So $|H_i| \leq n$ always.

Adding up the error terms $D + X_i - H_i$ gives a total that is bounded by $4n$ with at least constant probability. This probability is independent of the constant probability that $|S| > 4n$. So if both of these events occur, we win.

The total cost of the coin is $O(T + n^2) = O(n^2)$. This also gives a cost of $O(n^2)$ for consensus.

This turns out to be optimal, because of a lower bound on the expected total steps for consensus appearing in the same paper [AC08]. This is a bit disappointing, because an $\Omega(n^2)$ lower bound translates to $\Omega(n)$ steps for each process even if we can distribute the load evenly across all processes (which Algorithm 14.1 might not). We'd like to get something that scales better, but to get around the lower bound we will need to switch to an oblivious adversary.

14.5 Leader election with sifters

The idea of a **sifter** [AA11] is to use randomization to quickly knock out processes while leaving at least one process always. The current best known sifter for standard read-write registers, due to Giakkoupis and Woelfel [GW12], is shown in Algorithm 14.2.

```

1 Choose  $r \in \mathbb{Z}^+$  such that  $\Pr[r = i] = 2^{-i}$ 
2  $A[r] \leftarrow 1$ 
3 if  $A[r + 1] = 0$  then
4   | stay
5 else
6   | leave

```

Algorithm 14.2: Giakkoupis-Woelfel sifter [GW12]

The cost of executing the sifter is two operations. Each process chooses an index r according to a geometric distribution, writes $A[r]$, and then checks if any other process has written $A[r + 1]$. The process stays if it sees nothing.

Because any process with the maximum value of r always stays, at least one process stays. To show that not too many processes stay, let X_i be the number of survivors with $r = i$. This will be bounded by the number of processes that write to $A[i]$ before any process writes to $A[i + 1]$. We can immediately see that $E[X_i] \leq n \cdot 2^{-i}$, since each process has probability 2^{-i} of writing to $A[i]$. But we can also argue that $E[X_i] \leq 2$ for any value of n .

The reason is that because the adversary is oblivious, the choice of which process writes next is independent of the location it writes to. If we condition on some particular write being to either $A[i]$ or $A[i + 1]$, there is a $1/3$ chance that it writes to $A[i + 1]$. So we can think of the subsequence of writes that either land in $A[i + 1]$ or $A[i]$ as a sequence of biased coin-flips, and we are counting how many probability- $2/3$ tails we get before the first probability- $1/3$ heads. This will be 2 on average, or at most 2 if we take into account that we will stop after n writes.

We thus have $E[X_i] \leq \min(2, n \cdot 2^{-i})$. So the expected total number of winners is bounded by $\sum_{i=1}^{\infty} E[X_i] \min(2, n \cdot 2^{-i}) = 2 \lg n + O(1)$.

Now comes the fun part. Take all the survivors of our sifter, and run them through more sifters. Let S_i be the number of survivors of the first i sifters. We've shown that $E[S_{i+1} | S_i] = O(\log S_i)$. Since \log is concave, Jensen's inequality then gives $E[S_{i+1}] = O(\log E[S_i])$. Iterating this gives $E[S_i] = O(\log^{(i)} S_0) = O(\log^{(i)} n)$. So there is some $i = O(\log^* n)$ at which $E[S_i]$ is a constant.

This doesn't quite give us leader election because the constant might not be 1. But there are known leader election algorithms that run in $O(1)$ time with $O(1)$ expected participants [AAG⁺10], so we can use these to clean up any excess processes that make it through all the sifters. The total cost is $O(\log^* n)$ operations for each process.

14.6 Consensus with sifters

We've seen that we can speed up leader election by discarding potential leaders quickly. For consensus, we want to discard potential output values quickly. This is a little more complicated, because processes carrying losing output values can't just drop out if we haven't determined the winning values yet.

We are going to look at an algorithm by Aspnes and Er [AE19] that solves consensus in $O(\log^* n)$ operations per process, using sifters built on top of a stronger primitive than normal registers. (This algorithm uses some ideas from an earlier paper of Aspnes [Asp15], which gives a more

complicated $O(\log \log n)$ -time algorithm for standard registers.) As with the Giakkoupis-Woelfel leader election algorithm, we assume an oblivious adversary scheduler.

The stronger primitive we need is a **max register**[AACH12]. Unlike a standard register, reading a max register returns the largest value ever written to it; equivalently, trying to write a smaller value to a max register than it already contains has no effect.

What this means for randomized algorithms is that we can knock out values using a max register by a simple priority scheme. If each process writes a tuple $\langle r, v \rangle$ to the max register, where r is a unique random priority and v is the process's preferred value, then the i -th value written appears in the max register only if it is a left-to-right maximum of the sequence of priorities, which occurs with probability roughly $1/i$. (This follows from symmetry and the fact that the oblivious adversary can't selectively schedule smaller priorities first.) So on average only $H_n = O(\log n)$ of these values will ever appear in the max register. By having each process read and return a value from the max register, we've gone from n possible values to $O(\log n)$ possible values on average.

We now run into two technical obstacles. The first is that we can't necessarily guarantee that all the random priorities are unique, which we would need to get an exact $1/i$ probability for survival. This can be dealt with by choosing priorities over a sufficiently large range that the rare collisions don't add much to the expected survivors (say, $O(n^3)$). The second issue is that if we try to iterate the sifter, in the second round we have many copies of each input value. So even if one copy drops out because of a low priority, some other copy might get lucky and survive. To go from $O(\log n)$ expected values to $O(\log \log n)$ expected values, we need to make sure that no value gets more than one chance at surviving.

Here is where we use an idea that first appeared in [Asp15]. Instead of generating a new priority for each value in each round, we generate a sequence r_1, r_2, \dots, r_ℓ of priorities for all ℓ rounds all at once at the beginning. We can then store $\langle r_i, \dots, r_\ell, v \rangle$ in the max register for round i , and the leading priority r_i controls survival. Now we don't care about having multiple copies of a value, because they will all have the same initial r_i , and if the first copy to be written doesn't survive, none of them will.

The result of applying this idea is Algorithm 14.3. Because each iteration of the loop reduces S_i survivors to $S_{i+1} = O(\log S_i)$ survivors on average, the same application of Jensen's inequality that we used for Algorithm 14.2 works here as well, so for an appropriate $\ell = O(\log^* n)$ we can get down to a single surviving value with reasonably high probability.

```

1 procedure sifter( $v$ )
2   Choose random ranks  $r_1 \dots r_\ell$ 
3   for  $i \leftarrow 1 \dots \ell$  do
4     writeMax( $M_i, \langle r_i, \dots, r_\ell, v \rangle$ )
5      $\langle r_i, \dots, r_\ell, v \rangle \leftarrow$  readMax( $M_i$ )
6   return  $v$ 

```

Algorithm 14.3: Sifter using max registers [AE19]

This is not quite enough to get consensus, which requires agreement always, but with some additional machinery it is possible to detect when the protocol fails and re-run it if necessary. The final result is consensus in expected $O(\log^* n)$ max register operations.

Appendix A

Assignments

Assignments are typically due Wednesdays at 23:00, or Fridays at 23:00 when Wednesday would be too close to a break.

Assignments should be uploaded to Canvas in PDF format.

Do not include any identifying information in your submissions.

This will allow grading to be done anonymously. Points will be deducted if you violate this rule.

Make sure that your submissions are readable. You are strongly advised to use L^AT_EX, Microsoft Word, Google Docs, or similar software to generate typeset solutions. Scanned or photographed handwritten submissions often come out badly, and submissions that are difficult for the graders to read will be penalized.

Sample solutions will appear in this appendix after the assignment is due. To maintain anonymity of both submitters and graders, questions about grading should be submitted through Canvas.

A.1 Assignment 1: due Wednesday, 2021-02-24, at 23:00

[[[To be announced.]]]

A.2 Assignment 2: due Friday, 2021-03-12, at 23:00

[[[To be announced.]]]

A.3 Assignment 3: due Friday, 2021-03-26, at 23:00

[[[To be announced.]]]

**A.4 Assignment 4: due Wednesday, 2021-04-07, at
23:00**

[[[To be announced.]]]

**A.5 Assignment 5: due Wednesday, 2021-04-21, at
23:00**

[[[To be announced.]]]

**A.6 Assignment 6: due Wednesday, 2021-05-05, at
23:00**

[[[To be announced.]]]

Appendix B

Sample assignments from Fall 2019

B.1 Assignment 1: due Thursday, 2019-09-12, at 23:00

Bureaucratic part

Send me email! My address is james.aspnes@gmail.com.

In your message, include:

1. Your name.
2. Your status: whether you are an undergraduate, grad student, auditor, etc.
3. Whether you are taking the course as CPSC 469 or CPSC 569.
4. Anything else you'd like to say.

(You will not be graded on the bureaucratic part, but you should do it anyway.)

B.1.1 The golden ticket

An infamous candy company announces it is placing a golden ticket uniformly at random in one of n candy bars that are distributed to n children, with unimaginable wealth to be bestowed on the child who receives the ticket.

Unfortunately, the candy company owner is notorious for trickery and lies, so the probability that the golden ticket actually exists is only $1/2$.

Consider the last child to open their candy bar, and what estimate they should make of their probability of winning after seeing the first k children open their candy bars to find no ticket.

An optimist reasons: If the ticket does exist, then the last child's chances went from $1/n$ to $1/(n - k)$. So the fact that none of the first k children got the ticket is good news!

A pessimist reasons: The more candy bars are opened without seeing the ticket, the more likely it is that the ticket doesn't exist. So the fact that none of the first k children got the ticket is bad news!

Which of them is right? Compute the probability that the last child receives the ticket given the first k candy bars come up empty, and compare this to the probability that the last child receives the ticket given no information other than the initial setup of the problem.

Solution

Let W be the event that the last child wins, C the event that the candy bar exists, and L_k the event that the first k children lose.

Without conditioning, we have

$$\begin{aligned}\Pr[W] &= \Pr[W | C] \Pr[C] + \Pr[W | \neg C] \Pr[\neg C] \\ &= \frac{1}{n} \cdot \frac{1}{2} + 0 \cdot \frac{1}{2} \\ &= \frac{1}{2n}.\end{aligned}$$

With conditioning, we have

$$\begin{aligned}
 \Pr[W \mid L_k] &= \frac{\Pr[W \wedge L_k]}{\Pr[L_k]} \\
 &= \frac{1/2n}{\Pr[L_k]C\Pr[C] + \Pr[L_k]\neg C\Pr[\neg C]} \\
 &= \frac{1/2n}{\frac{n-k}{n} \cdot \frac{1}{2} + 1 \cdot \frac{1}{2}} \\
 &= \frac{1/2n}{\frac{n-k}{2n} + \frac{1}{2}} \\
 &= \frac{1/2n}{\frac{2n-k}{2n}} \\
 &= \frac{1}{2n-k}.
 \end{aligned}$$

Since $\frac{1}{2n-k} > \frac{1}{2n}$ when $k > 0$, the optimist is correct.

B.1.2 Exploding computers

Suppose we have a system of $3n$ machines $m_0, m_2, \dots, m_{3n-1}$ in a ring, and we turn each machine on at random with independent probability p . Over time, the machines may overheat: if we turn on three consecutive machines m_i, m_{i+1} , and m_{i+2} (wrapping indices around mod $3n$), the middle machine m_{i+1} overheats and explodes. Explosions happen simultaneously and are purely a function of which machines are initially turned on; so if m_1, m_2, m_3, m_4 , and m_5 are all turned on, m_2, m_3 , and m_4 will all explode.

Call a machine a **active** if it is turned on and does not explode.

We want to choose p to maximize the number of active machines X . But there are two ways to interpret this goal.

1. Suppose our goal is to maximize the *expected* number of active machines $E[X]$. What value of p should we choose, and what does this give us for $E[X]$?
2. Suppose instead our goal is to maximize the probability of having the maximum possible number of active machines. More formally, letting M be the number of survivors we could get if we chose which machines to turn on optimally, we want to choose p to maximize $\Pr[X = M]$. What value of p should we choose, and what does this give us for $\Pr[X = M]$?

Solution

1. Let A_i be the event that machine m_i is active, and let X_i be the indicator variable for m_i being active and not exploding. We are trying to maximize $E[\sum X_i] = \sum E[X_i] = n E[X_1]$, where the last equation holds by symmetry.

Observe that $X_1 = 1$ if A_1 occurs and it is not the case that both A_0 and A_2 occur. This gives $E[X_1] = \Pr[X_1 = 1] = p(1 - p^2) = p - p^3$. By setting the derivative $1 - 3p^2$ to zero we find a unique extremum at $p = \sqrt{1/3}$; this is a maximum since $p(1 - p^2) = 0$ at the two corner solutions $p = 0$ and $p = 1$. Setting $p = \sqrt{1/3}$ gives $3n E[X_i] = 3n \cdot \sqrt{1/3}(2/3) = n \frac{2}{\sqrt{3}} \approx n \cdot 1.154701 \dots$

2. First we need to figure out M .

We can assume that we set up the optimal configuration so that none explode, because any initial configuration that produces explosions can be replaced by one where the dead machines are just turned off to start with, with no reduction in the number of active machines.

Conveniently, we can divide the machines into consecutive groups of three, and observe that (a) turning on the first two machines in each group causes no explosions, (b) turning on three machines in any group causes the middle one to explode. This shows $M = 2n$ is the maximum achievable number of machines, because (a) lets us get M and (b) says we can't do more, since this would give us a group with three by the Pigeonhole Principle.

We still need to characterize how many possible patterns of $M = 2n$ machines produce no explosions. Since we can only have two consecutive active machines, each pair of active machines must be followed by an inactive machine. Putting two inactive machines in a row would allow a partitioning into groups of three with some group getting only one active machine, leaving some other group with three, which is trouble. So the pattern must consist of pairs of active machines alternating with single inactive machines. There are exactly three ways to do this.

Each of these three patterns occurs with probability $p^{2n}q^n$ where $q = 1 - p$. Maximizing this probability can again be done by taking the derivative $2np^{2n-1}q^n - np^{2n}q^{n-1}$ and setting to 0. Removing common factors gives $2q - p = 0$, or $2 - 3p = 0$, giving $p = 2/3$, which is about what one would expect.

We then get $\Pr[X = 2n] = 3 \cdot (2/3)^{2n}(1/3)^n$. This is pretty small for reasonably large n , which suggests that a randomized algorithm may not be the best in this case.

B.2 Assignment 2: due Thursday, 2019-09-26, at 23:00

B.2.1 A logging problem

An operating system generates a new log entry at time t with independent probability p_t , and the size of each log entry for any t is an integer X chosen uniformly in the range $a \leq X \leq b$, where the choice of X at a particular time t is independent of all the other log entry sizes and all choices of whether to write a log entry.

Suppose that over some fixed period of time, the operating system generates n log entries on average. Compute the expectation of the total size S of all the log entries, and show that the probability that S is more than a small constant fraction larger than $E[S]$ is exponentially small as a function of n .

Solution

For each t , let X_t be the size of the log entry at time t . Then $E[S] = \sum_t E[X_t] = \sum_t p_t \frac{a+b}{2} = \frac{a+b}{2} E[\sum_t p_t] = \frac{a+b}{2} \cdot n$.

We can't use Chernoff bounds directly on the X_t because they aren't Bernoulli random variables, and we can't use Hoeffding's inequality because we don't actually know how many of them there are. So we will use Chernoff first to bound the number of nonzero X_t and then use Hoeffding to bound their sum.

The easiest way to think about this is to imagine we generate the nonzero X_t by generating a sequence of independent values Y_1, Y_2, \dots , with each Y_i uniform in $[a, b]$, then select the i -th nonzero X_t to have value Y_i .

Let Z_t be the indicator variable for the event that a log entry is written at time t . Then $N = \sum_t Z_t$ gives the total number of log entries, and $E[N] = E[\sum_t Z_t] = \sum_t p_t = n$.

Pick some $\delta < 1$; then Chernoff's inequality (5.2.2) gives

$$\Pr[N \geq (1 + \delta)n] \leq e^{-n\delta^2/3}.$$

Suppose $N < (1 + \delta)n$. Then $\sum_t X_t = \sum_{i=1}^N Y_i \leq \sum_{i=1}^{(1+\delta)n} Y_i$. This last quantity is a sum of independent bounded random variables, so we can apply

the asymmetric version of Hoeffding's inequality (5.3.16) to get

$$\Pr \left[\sum_{i=1}^{(1+\delta)n} Y_i - (1+\delta)n \cdot \frac{a+b}{2} \geq s \right] \leq \exp \left(-\frac{s^2}{2(1+\delta)n \left(\frac{b-a}{2}\right)^2} \right)$$

For $s = \delta n \cdot \frac{a+b}{2}$ this gives a bound of

$$\begin{aligned} \exp \left(-\frac{-\delta^2 n^2 \left(\frac{a+b}{2}\right)^2}{2(1+\delta)n \left(\frac{b-a}{2}\right)^2} \right) &\leq \exp \left(-\frac{-\delta^2 n \left(\frac{a+b}{2}\right)^2}{2(1+\delta) \left(\frac{b-a}{2}\right)^2} \right) \\ &= \exp \left(-\frac{\delta^2 n}{2(1+\delta)} \right). \end{aligned}$$

If we put this all together, we get

$$\Pr \left[S \geq (1+2\delta)n \frac{a+b}{2} \right] \leq e^{-n\delta^2/3} + e^{-n\delta^2/2(1+\delta)} = e^{-\Theta(n)}.$$

B.2.2 Return of the exploding computers

Suppose we are working in the model described in §B.1.2, where $3n$ machines in a ring are each turned on independently at random with probability p , and a machine remains active only if it is turned on and at least one of its neighbors is not turned on. We have seen that an optimal choice of p gives cn active machines on average for a constant c independent of n .

Unfortunately, averages are not good enough for our customers. Show that there is a choice of p and constants $c' > 0$ and $\delta > 0$ such that at least $c'n$ machines are active with probability at least $1 - e^{-\delta n}$ for any n .

Solution

We can use the method of bounded differences. Let X_i be the indicator for the event that machine i is turned on, and let $S = f(X_1, \dots, X_{3n})$ compute the number of active machines. Then changing one input to f from 0 to 1 increases f by at most 1, and decreases it by at most 2 (since it may be that the new machine explodes, producing no increase, and it also explodes both neighbors, producing a net -2 change). So McDiarmid's inequality (5.3.12) applies with $c_i = 2$ for all i , giving

$$\Pr [f(X_1, \dots, X_{3n}) - \mathbb{E}[f(X_1, \dots, X_{3n})] \leq t] \leq \exp \left(-\frac{2t^2}{3n \cdot 2^2} \right) = e^{-t^2/6n}.$$

Choose p to maximize the expected number of active machines, and let $E[S] = c'n$. Choose c so that $0 < c < c'$. Then $\Pr[S \leq cn] = \Pr[S - E[S] \leq c'n - cn] \leq e^{-(c'-c)^2 n^2 / 6n} = e^{-(c'-c)^2 n / 6} = e^{-\delta n}$, where $\delta = (c' - c)^2 / 6 > 0$.

The above is my original sample solution, below are some quick sketches of alternative approaches that showed up in submissions:

- Analyze the dependence and use Lemma 5.2.2. This gives similar results to McDiarmid's inequality with a bit more work. The basic idea is that $E[X_i | X_1 \dots X_{i-1}] \geq p(1-p)$ for $i < 3n$ whether or not X_{i-1} is 1 (we don't care about the other variables). So we can let $p_i = p(1-p)$ for these values of i . To avoid wrapping around at the end, either omit X_{3n} entirely or let $p_{3n} = 0$.
- Even simpler: Let Y_i be the indicator for the event that machine i is active. Then Y_3, Y_6, \dots, Y_{3n} are all independent, and we can bound $\sum_{i=1}^n Y_{3i} \leq \sum_{i=1}^{3n} Y_i$ using either Chernoff or Hoeffding.

B.3 Assignment 3: due Thursday, 2019-10-10, at 23:00

B.3.1 Two data plans

A cellphone company offers a choice of two data plans, both based on penalizing the user whenever they use more data than in the past. For both plans the assumption is that your data usage in each month $i \in \{0 \dots \infty\}$ is an independent continuous random variable that is uniform in the range $[0, c]$ for some fixed constant c .

With **Plan A**, you pay a penalty of \$1 for every month i in which your usage X_i exceeds $\max_{j < i} X_j$. This excludes month 0, for which the max of the previous empty set is taken to be $+\infty$.

With **Plan B**, you pay a penalty of \$100 for every pair of months $2i$ and $2i + 1$ in which X_{2i} and X_{2i+1} both exceed $\max_{j < 2i} X_j$. As in Plan A, you never pay a penalty for the pair of months 0 and 1.

1. Suppose that you expect to live forever (and keep whatever plan you pick forever as well). If your goal is to minimize your expected total cost over eternity, which plan should you pick?
2. Which plan is cheaper on average if you don't plan to live for more than, say, the next 1000 years?

(Assume no discounting of future payments in either case.)

Solution

Let A_i be the event that we pay a penalty in month i with plan A, and let B_i for even i be the event that we pay a penalty for months i and $i + 1$ in plan B. Then for $i > 0$,

$$\begin{aligned}\Pr[A_i] &= \Pr\left[X_i > \max_{j < i} X_j\right] \\ &= \Pr\left[X_i = \max_{j \leq i} X_j\right] \\ &= \frac{i!}{(i+1)!} \\ &= \frac{1}{i+1}.\end{aligned}$$

Similarly,

$$\begin{aligned}\Pr[B_i] &= \Pr\left[X_i > \max_{j < i} X_j \text{ and } X_{i+1} > \max_{j < i} X_j\right] \\ &= \frac{i! \cdot 2}{(i+2)!} \\ &= \frac{2}{(i+2)(i+1)}.\end{aligned}$$

(In the first case, we count all orderings of the $i + 1$ elements $0 \dots i$ that make X_i largest, and divide by the total number of orderings. In the second case, we count all the orderings that make X_i and X_{i+1} both larger than all the other values.)

Now let us do some sums.

1. For the infinite-lifetime version, Plan A costs

$$\sum_{i=1}^{\infty} \frac{1}{i+1},$$

which diverges. Plan B costs

$$\begin{aligned} 100 \cdot \sum_{i=1}^{\infty} \frac{2}{(2i)(2i-1)} &\leq 100 \cdot \sum_{i=1}^{\infty} \frac{2}{(2i)^2} \\ &= 100 \cdot \sum_{i=1}^{\infty} \frac{1}{2i^2} \\ &= \frac{100}{2} \cdot \frac{\pi^2}{6}. \end{aligned}$$

which is finite. So Plan B is better for immortals.

- For the finite-lifetime version, over n months, Plan A costs

$$\sum_{i=1}^n \frac{1}{i} \approx \ln n + \gamma$$

Plan B has a $\frac{2}{3 \cdot 4} = \frac{1}{6}$ chance of costing \$100 after the first four months, giving an expected cost of at least \$16. As long as you keep both for at least four months, for $n \ll e^{16} \approx 8886110$ months ≈ 740509 years, plan A is better.

B.3.2 A randomly-indexed list

Consider the following data structure: We store elements in a sorted linked list, and as each element is inserted, with probability p we also insert it in a balanced binary search tree data structure with guaranteed $O(\log n)$ -time search and insert operations (say, a red-black tree). We can now search for an element x by searching for the largest element y in the tree less than or equal to x , then walking along the list starting from y until we find x (or some value bigger than x).

Recall that for a skip list with (small) parameter p , the expected number of pointers stored per element is $1 + O(p)$ and the expected cost of a search operation is $O\left(\frac{1}{p} \log n\right)$.

- What are the corresponding asymptotic values for the expected pointers per element and the expected cost of a search for this new data structure, as a function of p and n ?
- Suppose you are given n in advance. Is it possible to choose p based on n for the tree-based data structure to get lower expected space overhead (expressed in terms of pointers per element) than with a skip list, while still getting $O(\log n)$ expected search time?

Solution

1. This is actually easier than the analysis for a skip list, because there is no recursion to deal with.

A balanced binary search tree stores exactly 2 pointers per element, so with an expected pn elements in the tree, we get $n + 2pn$ pointers total, or $1 + 2p$ pointers per element.

For the search cost, work backwards from the target x to argue that we traverse an expected $O(1/p)$ edges in the linked list before hitting a tree node. Searching the tree takes $O(\log n)$ time independent of p . This gives a cost of $O\left(\frac{1}{p} + \log n\right)$.

2. Yes. Let $p = \Theta(1/\log n)$. Then we use an expected $1 + O(1/\log n)$ pointers per element, while the search cost is $O\left(\frac{1}{1/\log n} + \log n\right) = O(\log n)$. In contrast, getting $1 + O(1/\log n)$ expected space with a skip list would also require setting $p = \Theta(1/\log n)$, but this would give an expected search cost $\Theta(\log^2 n)$.

B.4 Assignment 4: due Thursday, 2019-10-31, at 23:00

B.4.1 A hash tree

Suppose we attempt to balance a binary search tree by inserting n elements one at a time, using hashes of the values instead of the values themselves.

Specifically, let $M = \{1 \dots n^2\}$ be the set of possible hash values; let $U = \{1 \dots u\}$ be the universe of possible values, where $u \gg n^2$; and let H be a 2-universal family of hash functions $h : U \rightarrow M$. After the adversary chooses n distinct values x_1, \dots, x_n , the algorithm chooses a hash function $h \in H$ uniformly at random, and inserts x_1 through x_n in order into a binary search tree with no rebalancing, where the key for each value x_i is $h(x_i)$.

In the case of duplicate keys (which shouldn't happen very often), assume that if we try to insert x into a subtree with root r , and $h(x) = h(r)$, we just adopt some fixed rule like always inserting x into the right subtree of r .

For any n and $u \gg n^2$, prove or disprove: for any 2-universal family H of hash functions, and any sequence of values x_1, \dots, x_n , the expected depth of a tree constructed by the above process is $O(\log n)$.

Solution

Disproof: We will construct a 2-universal family H such that $h(x) = x$ for all $h \in H$ whenever $1 \leq x \leq m$. We can then insert the values $x_i = i$ with corresponding hash values $h(x_i) = i$ and get a tree of depth exactly n .

Start with a *strongly* 2-universal family H' . For each h' in H , construct a corresponding h according to the rule $h(x) = x$ when $1 \leq x \leq m$ and $h(x) = h'(x)$ otherwise. We claim that this gives a 2-universal family of hash functions.

Recall that H is 2-universal if, for any $x \neq y$, $\Pr[h(x) = h(y)] \leq 1/m$, when h is chosen uniformly from H . We consider three cases:

1. If $1 \leq x \leq m$ and $1 \leq y \leq m$, then $h(x) = x \neq y = h(y)$ always.
2. If $1 \leq x \leq m$ and $m < y$, then $\Pr[h(y) = h(x)] = \Pr[h'(y) = x] = 1/m$, since $h(y)$ is equally likely to be any value by strong 2-universality. A symmetric argument works if $m < x$ and $1 \leq y \leq m$.
3. If $m < x$ and $m < y$, then $\Pr[h(x) = h(y)] = \Pr[h'(x) = h'(y)] = 1/m$.

In each case we have $\Pr[h(x) = h(y)] \leq 1/m$. So H gives a 2-universal hash family that produces trees of depth n , much greater than $O(\log n)$.

B.4.2 Randomized robot rendezvous on a ring

Suppose we have a ring of size nm , consisting of positions $0, \dots, nm - 1$, with n robots starting at positions $X_{i0} = im$.

At each time step t , an adversary chooses one of the n robots i , and this robot moves from position X_{it} to $X_{i,t+1} = (X_{it} \pm 1) \bmod mn$, moving clockwise or counterclockwise with equal probability. If any other robot or robots i' are at the same position $X_{i't} = X_{it}$, they also move to the same new position with robot i . Eventually, the robots slowly coalesce until all robots are in the same position.

The process finishes at the first time τ where $X_{i\tau} = X_{j\tau}$ for all i and j . What is $E[\tau]$?

(Assume that the adversary's choice of robot to move at each time t doesn't involve predicting the future. Formally, this choice must be measurable \mathcal{F}_t , where \mathcal{F}_t is the σ -algebra generated by all the positions of the robots at times up to and including t .)

Solution

We'll use a variant of the $X_t^2 - t$ martingale for random walks.

For each i and t , let $Y_{it} = X_{i+1,t} - X_{it}$ be the size of the gap between X_{i+1} and X_i (wrapping around in the obvious way).

Suppose at some step t the adversary chooses robot j to move. Let i and k be the smallest and largest robot ids such that $X_{it} = X_{jt} = X_{kt}$ (again wrapping around in the obvious way).

Then $Y_{i-1,t}$ and Y_{kt} are the only gaps that change, and each rises or drops by 1 with equal probability. So $E[Y_{i-1,t+1}^2 | \mathcal{F}_t] = Y_{i-1,t}^2 + 1$ and $E[Y_{j,t+1}^2 | \mathcal{F}_t] = Y_{j,t}^2 + 1$. If we define $Z_t = \sum_{i=1}^n Y_{it}^2 - 2t$, then $\{Z_t\}$ is a martingale with respect to $\{\mathcal{F}_t\}$.

For any adversary strategy, starting from any configurations, there is a sequence of at most n^2m coin-flips that causes all robots to coalesce. So $E[\tau] \leq n^2m2^{n^2m} < \infty$. We also have that $|Z_{t+1} - Z_t| \leq 2$, so we can apply the finite-expectation/bounded-increments case of Theorem 8.3.1 to show that $E[Z_\tau] = E[Z_0] = nm^2$. But at time τ , all but one interval Y_{it} is zero, and the remaining interval has length mn . This gives $E[Z_\tau] = E[(mn)^2 - 2\tau]$, giving $E[\tau] = \frac{1}{2}(n^2m^2 - nm^2) = \binom{n}{2}m^2$.

The nice thing about this argument is that it instantly shows that the adversary's strategy doesn't affect the expected time until all robots coalesce, but the price is that the argument is somewhat indirect. For specific adversary strategies, it may be possible to show the coalescence time more directly. For example, consider an adversary that always picks robot 0. Then we can break down the resulting process into a sequence of phases delimited by collisions between 0 and robots that have not yet been added to its entourage. The first such collision happens after robot 0 hits one of the two absorbing barriers at $-m$ and $+m$, which occurs in m^2 steps on average. At this point, we now have a new random walk with absorbing barriers at $-m$ and $+2m$ relative to the position of robot 0 (possibly after flipping the directions to get the signs right). This second random walk takes $2m^2$ steps on average to hit one of the barriers. Continuing in this way gives us a sequence of random walks with absorbing barriers at $-m$ and $+km$ for each $k \in \{1 \dots n-1\}$. The total time is thus $\sum_{k=1}^{n-1} km^2 = \frac{(n-1)n}{2}m^2 = \binom{n}{2}m^2$, as shown above for the general case.

B.5 Assignment 5: due Thursday, 2019-11-14, at 23:00

B.5.1 Non-exploding computers

Engineers from the Exploding Computer Company have devised a new scheme for managing the ring of overheating computers from Problem B.1.2. In the new scheme, we start with an n -bit vector $S^0 = 0^n$. Each 1 represents a computer that is turned on, and each 0 represents a computer that is turned off.¹

Call a state S **permissible** if there is no j such that $S_j^{t+1} = S_{j+1}^{t+1} = S_{j+2}^{t+1} = 1$, where the indices wrap around mod n .

At each step, some position i is selected uniformly at random. If $S_i^t = 1$, S_i^{t+1} is set to 0 with probability p . If $S_i^t = 0$, S_i^{t+1} is set to 1 if this creates a permissible configuration, and stays 0 otherwise. For any positions $j \neq i$, $S_j^{t+1} = S_j^t$.

1. Show that when $0 < p < 1$, the sequence of values S^t forms a Markov chain with states consisting of all permissible sets, and that this chain has a unique stationary distribution π in which $\pi(S) = \pi(T)$ whenever $|S| = |T|$, where $|S|$ is the number of 1 bits in S .
2. Let τ be the first time at which $|S^\tau| \geq n/2$. Show that for any n , there is a choice of $p \geq 0$ such that $E[\tau] = O(n \log n)$.

Solution

1. First observe that the update rule for generating S^{t+1} depends only on the state S^t ; this gives that $\{S^t\}$ is a Markov chain. It is irreducible, because there is always a nonzero-probability path from any permissible configuration S to any permissible configuration T that goes through the empty configuration 0^n . It is aperiodic, because for any nonempty configuration S^t , there is a nonzero probability that $S^{t+1} = S^t$, since we can pick a position i with $S_i^t = 1$ and then choose not set S_i^{t+1} to 0. So a unique stationary distribution π exists.

We can show that $\pi(S)$ depends only on $|S|$ by using the fact that the Markov chain is reversible. Let S and T be reachable states that differ

¹The marketing department convinced the company to change the number of computers in the ring from $3n$ to n in response to consumer complaints. If the engineers' scheme works, the next meeting of the marketing department will consider choosing a new name for the company.

only in position i . Suppose that $S_i^t = 0$ and $T_i^t = 1$. Then $p_{ST} = 1/n$ and $p_{TS} = p/n$. Let $c = \sum_S p^{-|S|}$, where S ranges over all admissible states, and let $\pi_S = p^{-|S|}/c$. Pick some S and T as above and let $k = |S|$.

$$\begin{aligned} \pi_S p_{ST} &= \frac{p^{-k}}{c} \frac{1}{n} \\ &= \frac{p^{-k-1}}{c} \frac{p}{n} \\ &= \pi_T p_{TS}. \end{aligned}$$

So this particular choice of π satisfies the detailed balance equations and is the stationary distribution of the chain. Since $\pi(S)$ only depends on $|S|$, it is immediate that $\pi(S) = \pi(T)$ when $|S| = |T|$.

2. For this part, we'll set $p = 0$, then walk directly up to a configuration with $|S^t| \geq n/2$ in $O(n \log n)$ steps on average. Setting $p = 0$ means that the chain is no longer irreducible, but we won't care about this since we are not worried about convergence.

Let S be a permissible configuration with $|S| = k$. We want to get a lower bound on the number of positions i such that $S[i/1]$ is also permissible. There $n - k$ zeroes in S ; each such 0 can be switched to 1 unless it would create three ones in a row. We will show that at most k of the $n - k$ zeroes cannot be switched, by constructing an injection f from the set of unswitchable 0 positions to the set of 1 positions.

Let i be a position such that $S_i = 0$ but $S[i/1]$ is not permissible. Then one of the following cases holds:

- (a) $S_{i-2} = S_{i-1} = 1$. Let $f(i) = i - 1$.
- (b) $S_{i-1} = S_{i+1} = 1$. Let $f(i) = i - 1$.
- (c) $S_{i+1} = S_{i+2} = 1$. Let $f(i) = i + 1$.

To show that this is an injection, observe that the only way for $f(i) = f(j)$ to occur is if $f(i) = i + 1$ and $f(j) = j - 1$ (or vice versa). But if $f(i) = i + 1$, then $S_j = S_{i+2} = 1$, and S_j is not an unswitchable zero.

So out of $n - k$ zeroes we have at most k unswitchable zeroes, giving at least $n - 2k$ switchable zeroes. The expected waiting time to hit one of these at least $n - 2k$ switchable zeroes is at most $\frac{n}{n-2k}$, giving a total expected waiting time of at most $\sum_{k=0}^{\lfloor (n-1)/2 \rfloor} \frac{n}{n-2k} \leq nH_n = O(n \log n)$.

I suspect the correct expected waiting time is in fact $O(n)$, because (a) the pattern 11011 includes at least one 1 that is not the image of any unswitchable zero; (b) after, say, $n/8$ steps it is likely (via linearity of expectation and McDiarmid's inequality) that there are a linear number ℓ of such patterns in S ; so that (c) even as k gets close to $n/2$, the number of unswitchable zeros is at least $n - 2k + \ell = \Theta(n)$, giving constant expected waiting times to find the next switchable zero. But actually proving (b) given all the dependencies flying around looks at least mildly painful, so we may want to just hope that an $O(n \log n)$ bound is good enough.

B.5.2 A wordy walk

Consider the following random walk on strings over some alphabet Σ . At each step:

1. With probability p , if $|X^t| > 1$, delete a character from a position chosen uniformly at random from all positions in X^t . If $|X^t| = 1$, do nothing.
2. With probability q , insert a new character chosen uniformly at random from Σ into X^t . The new character will be inserted after the first i characters in X^t , where i is chosen uniformly at random from $\{0 \dots |X^t|\}$.
3. With probability r , choose one of the character positions in X^t uniformly at random and replace the character in that position with a character chosen uniformly at random from Σ .

For example, here is the result of running the above process for a few steps with Σ the lowercase Latin alphabet, $p = 1/2$, $q = 1/4$, and $r = 1/4$, starting from the string `markov: markov markzv marzv carzv caurzv cauzv cauav cauav cavav avav`.

Assume $p + q + r = 1$, $p > q > 0$, and $r > 0$.

Suppose that we start with $|X^0| = n$. Show that for any constants p, q, r satisfying the above constraints, and any constant $\epsilon > 0$, there is a time $t = O(n)$ such that $d_{TV}(X^t, \pi) \leq \epsilon$, where π is a stationary distribution of this process.

Solution

We can simplify things a bit by tracking $X^t = |S^t|$. Since changes in the length of S don't depend on the characters in S , $\{X^t\}$ is also a Markov

chain, with transition probabilities

$$p_{ij} = \begin{cases} p & j = i - 1 \\ q & j = i + 1 \\ r & \text{otherwise} \end{cases}$$

Applying the detailed balance equations gives a stationary distribution ρ for X^t of $\rho_i = (q/p)^{i-1}/(1 - q/p)$. By symmetry, this gives a stationary distribution $\pi_S = \frac{(q/p)^{|S|-1}|\Sigma|^{-|S|}}{1-q/p}$. For X with distribution ρ , we have $E[X]$ finite when $p > q$ since $E[X] = \frac{1}{1-q/p} \sum_{n=1}^{\infty} n(q/p)^{n-1}$ converges.

We'll show convergence for S^t starting from a particular S^0 using a coupling with a stationary copy of the chain $\{T^t\}$.

Our strategy will go as follows:

1. Let $m = |T^0|$. We will argue that a coupling that applies the same operation (delete, insert, change) to both copies of the chain reaches a configuration where $|S^t| = |T^t| = 1$ in $O \max(n, m)$ expected steps.
2. From a state with $|S^t| = |T^t| = 1$, with probability $r > 0$, both chains switch to a new character. Make this the same character and get $S^t = Y^t$.
3. If this doesn't work, repeat the argument from the start. Note that we now have $|S^t| = |T^t| \leq 2$, so after $O(1)$ expected steps we are back at $|S^t| = |T^t| = 1$ again. The waiting time for convergence is now geometric and has expectation $O(1)$.

Let us now justify the claim that the waiting time to reach $|S^t| = |T^t| = 1$ is $O(\max(n, m))$. This is just the usual argument for a biased random walk with one absorbing barrier. Let $Z^t = \max(|S^t|, |Y^t|)$. Let τ be the stopping time at which Z^t is first equal to 1. Observe that for $t < \tau$, $E[Z^{t+1} | Z^t \dots Z^0] = Z^t + q - p$, so $Q^t = Z^t + (p - q)t$ is a martingale with bounded increments, and we can apply OST to get that $E[Q_\tau] = 1 - (p - q)\tau = E[Q_0] = \max n, m$. This gives $E[\tau] = \frac{\max n, m - 1}{p - q}$.

We now have a coupling that gives an expected d coalescence time of $O(\max(n, m))$. If $\max(n, m) = n$, we are done. But it could be that $\max(n, m) = m$. Conditioning on $|T^t| > n$, we are truncating the distribution of $|T^t|$ to be a geometric distribution starting at $n + 1$; this has expected value $n + O(1)$, giving $E[\max(n, m)] = n + O(1)$ in either case. So the coalescence time is $O(n + O(1)) = O(n)$.

Since ϵ is a constant, after applying the Coupling Lemma and Markov's inequality, the O eats it.

B.6 Assignment 6: due Monday, 2019-12-09, at 23:00

B.6.1 Randomized colorings

Suppose we want to use a constant number k of colors to color the vertices of a graph G with n vertices and m edges so that we minimize the number of monochromatic edges. A particularly simple approach is to color the vertices independently and uniformly at random. This gives m/k monochromatic edges on average. But we'd like to derandomize the algorithm to get at most m/k monochromatic edges always.

There are two standard ways to do this:

1. Replace the independent random colors with pairwise independent random colors and then consider all possible random choices. Show that we can do this by generating n pairwise-independent colors from a small number of random values, and determine the asymptotic time complexity, as a function of n , m , and k , of generating and testing all the resulting colorings to find the best one.
2. Apply the method of conditional probabilities and assign colors one at a time to minimize the expected number of monochromatic edges conditioned on the assignment so far. Show that we can do so in this case, and compute the asymptotic time complexity as a function of n , m , and k of the resulting deterministic algorithm.

Solution

1. The obvious way to generate n pairwise-independent random values in $\{0, \dots, k-1\}$ is to apply the subset-sum construction described in §5.1.2.1. Let $\ell = \lceil \lg(n+1) \rceil$, assign a distinct nonempty subset S_v of $\{1, \dots, \ell\}$ to each vertex v , and let $X_v = \sum_{i \in S_v} r_i$ where r_1, \dots, r_ℓ are independent random values chosen uniformly from \mathbb{Z}_k . Then the X_v are pairwise independent, giving a probability of $1/k$ that any particular edge is monochromatic. Since this gives m/k monochromatic edges on average, enumerating all choices of r_1, \dots, r_ℓ will find a particular coloring that gets at least m/k monochromatic edges.

There are $k^\ell = k^{\lceil \lg(n+1) \rceil}$ possible choices. This is bounded by $k^{2+\lg n} = O(k^2 n^{\lg k}) = O(n^{\lg k})$, which is polynomial in n .

Generating each coloring takes $O(n \log n)$ time, and testing the edges takes $O(m)$ time, for a total time of $O(n^{\lg k}(m + n \log n))$.

With some data structure tinkering we can reduce the complexity a bit. We can enumerate all of r_1, \dots, r_k with an amortized $O(1)$ values changing at each iteration. This still requires changing about half the colors, but we can update each dynamically by subtracting off the old value of r_i and adding the new one, reducing the amortized cost per iteration from $O(n \log n)$ to $O(n)$. We still end up checking most of the edges, so the cost will now be $O(n^{\lg k}(m+n))$. I don't see an obvious way to do better.

2. For this part we let X_1, \dots, X_n be the colors assigned to the vertices (ordering them arbitrarily) and let $f(X_1, \dots, X_n)$ be the number of monochromatic edges given a particular coloring. Having fixed X_1, \dots, X_i , we want to choose X_{i+1} to minimize $E[f | X_1, \dots, X_{i+1}]$.

Observe that $f(X_1, \dots, X_n) = \sum_{uv \in E} [X_u = X_v]$, so $E[f | X_1, \dots, X_i] = \sum_{uv \in E} \Pr[X_u = X_v]$. For each uv , if $i+1 \notin \{u, v\}$, then $[X_u = X_v]$ is independent of X_{i+1} , giving $\Pr[X_u = X_v | X_1, \dots, X_{i+1}] = \Pr[X_u = X_v | X_1, \dots, X_i]$. So we only care about the cases where $i+1 \in \{u, v\}$. Assume without loss of generality that $i+1 = u$. If $v \leq i$, then $[X_u = X_v]$ is either 1 or 0 depending on whether we set $X_u = X_v$ or not. If $v > i+1$, then X_v is independent of X_{i+1} , so $\Pr[X_u = X_v | X_1, \dots, X_{i+1}] = \Pr[X_u = X_v | X_1, \dots, X_i]$.

This means that to minimize $E[f | X_1, \dots, X_{i+1}]$, the only edges we need to consider are those edges X_{uv} where one endpoint is $i+1$ and the other endpoint is in $\{1, \dots, i\}$. Minimizing the conditional expectation of f turns into the obvious greedy algorithm of picking a color for v_{i+1} that is least represented among its neighbors, which we can do in $O(1 + d(v_{i+1}))$ time. Adding up over all vertices v gives $O(\sum_{v \in V} (1 + d(v))) = O(n + m)$ time.

B.6.2 No long paths

Given a directed graph G with n vertices and maximum out-degree d , we would like to find an induced subgraph G' with at least $(1 - \delta)n/(d + 1)$ vertices, such that G' has no simple paths² of length $\lceil 2d \ln n \rceil$ or greater.

Find an algorithm that does this in polynomial expected time for any constant $d > 0$ and $\delta > 0$, and sufficiently large n .

²For the purposes of this problem, a **simple path** of length ℓ is a directed path with ℓ edges and no repeated vertices.

Solution

We'll put each vertex v to G' with independent probability $p = 1/(d+1)$. We now have two possible bad outcomes:

1. We may get a path of length $\lceil 2d \ln n \rceil$ in G' .
2. We may get fewer than $(1 - \delta)n/(d+1)$ vertices in G' .

We'll start by showing that the sum of the probabilities of these bad outcomes is not too big.

Let $\ell = \lceil 2d \ln n \rceil$. Observe that we can describe a path of length ℓ in G by specifying its starting vertex (n choices) and a sequence of ℓ edges each leaving the last vertex added so far (d choices each). This gives at most nd^ℓ paths of length ℓ in G . Each such path appears in G' with probability exactly $p^{\ell+1}$. Using the usual probabilistic-method argument, we get

$$\begin{aligned}
 \Pr [G' \text{ includes at least one path of length } \ell] &\leq \mathbb{E} [\text{number of paths of length } \ell \text{ in } G'] \\
 &\leq nd^\ell p^{\ell+1} \\
 &= np(pd)^\ell \\
 &= \frac{1}{d+1} n \left(\frac{d}{d+1} \right)^\ell \\
 &= \frac{1}{d+1} n \left(1 - \frac{1}{d+1} \right)^\ell \\
 &= \frac{1}{d+1} n \left(1 - \frac{1}{d+1} \right)^{\lceil 2d \ln n \rceil} \\
 &\leq \frac{1}{d+1} n \left(1 - \frac{1}{d+1} \right)^{2d \ln n} \\
 &\leq \frac{1}{d+1} n e^{-\ln n} \\
 &= \frac{1}{d+1} \\
 &\leq \frac{1}{2}.
 \end{aligned}$$

For the second-to-last inequality, we use the inequality $\left(1 - \frac{1}{d+1}\right)^{2d} \leq e^{-1}$ for $d \geq 1$. This is easiest to demonstrate numerically, but if we had to prove it we could argue that it holds when $d = 1$ (since $1/4 < 1/e$) and that $\left(1 - \frac{1}{d+1}\right)^{2d}$ is a decreasing function of d .

For getting too few vertices, use Chernoff bounds. Let X be the number of vertices in G' . Let $\mu = \mathbb{E}[X] = \frac{n}{d+1}$.

$$\begin{aligned} \Pr[X < (1 - \delta)n/(d + 1)] &= \Pr[X < (1 - \delta)\mu] \\ &\leq e^{-\mu\delta^2/2} \\ &= e^{-n\delta^2/(2(d+1))} \\ &< 1/4, \end{aligned}$$

for any fixed $d > 0$, $\delta > 0$, and sufficiently large n .

Adding these error probabilities together gives a total probability of failure of at most $3/4$. We can select the vertices in G' in time linear in n . So if we can detect failure in polynomial time and try again, we only have to run at most 4 poly-time attempts on average until we win, giving polynomial expected cost.

So now let us detect failure. There are at most nd^ℓ paths of length ℓ in G . We can enumerate all of them and check if each is in G' in time

$$\begin{aligned} O(nd^\ell \ell) &= O(nd^2 d^{\ln n + 1}) \\ &= O(nlde^{2d \ln n \ln d}) \\ &= O(n^{2d \ln d + 1} d^2 \log n), \end{aligned}$$

which is polynomial in n for fixed d .

B.7 Final exam

Write your answers in the blue book(s). Justify your answers. Work alone. Do not use any notes or books.

There are three problems on this exam, each worth 20 points, for a total of 60 points. You have approximately three hours to complete this exam.

B.7.1 A uniform ring

Suppose we have a ring of n nodes, each of which holds a bit. At each step, we pick a node uniformly at random and have it copy the bit of its left neighbor. For example, if we start in state 010, then picking the rightmost node will send us to state 011, and if we then pick the leftmost node we will reach state 111. From this last state no further changes are possible.

Suppose n is even and we start with a state where positions 1 through $n/2$ are all 0 and positions $n/2 + 1$ through n are all 1. What is the expected number of steps, as an asymptotic function of n , to reach a state where all bits are the same?

Solution

First observe that any transition preserves the property that our state looks like a string of the form $0^k 1^{n-k}$, where both the zeroes and ones all appear in consecutive positions around the ring.

Let X_t be the number of zeroes after t steps. We have $X_0 = n/2$, and our possible transitions when $0 < X_t < n$ are:

1. With probability $1/n$, we choose the leftmost zero and replace it with a one. This makes $X_{t+1} = X_t - 1$.
2. With probability $1/n$, we choose the leftmost one and replace it with a zero. This makes $X_{t+1} = X_t + 1$.
3. With the remaining probability $1 - 2/n$, we choose some node whose bit is equal to that of its left neighbor. This makes $X_{t+1} = X_t$.

When $X_t = 0$ or $X_t = n$, we have converged: no further changes are possible.

Looking at the transitions for X_t , we see that it is doing a very lazy random walk with absorbing barriers at 0 and n . If we remember that a non-lazy random walk converges in $O(n^2)$ steps on average, we can argue that the lazy version converges in $O(n^3)$ steps on average, because the expected waiting time between changes in X_t is $n/2 = O(n)$, and we can multiply this by the number of changes because of Wald's equation.

If we want to be very formal about this, we can also calculate the exact expected convergence time using the Optional Stopping Theorem. Let's guess that $Y_t = X_t^2 - ct$ is a martingale for some coefficient c . To find c , observe that we need

$$\begin{aligned} \mathbb{E} \left[\left(X_{t+1}^2 - c(t+1) \right) - \left(X_t^2 - ct \right) \mid X_t \right] &= \mathbb{E} \left[X_{t+1}^2 - X_t^2 - c \mid X_t \right] \\ &= \frac{1}{n} (2X_t + 1) + \frac{1}{n} (-2X_t + 1) - c \\ &= \frac{2}{n} - c \\ &= 0. \end{aligned}$$

So $\{Y_t\}$ is a martingale when $c = \frac{2}{n}$.

Let τ be the stopping time at which X_τ first equals 0 or n . The usual argument shows that $E[\tau]$ is finite, and $\{Y_t\}$ has bounded increments because Y_t never changes by more than $2X_t + 1 \leq 2n + 1$. So OST applies and $E[Y_\tau] = E[Y_0] = (n/2)^2$. But we can expand

$$\begin{aligned} E[Y_\tau] &= \frac{1}{2} \cdot 0 + \frac{1}{2} \cdot n^2 - \frac{2}{n} \cdot E[\tau] \\ &= \frac{n^2}{2} - \frac{2}{n} \cdot E[\tau]. \end{aligned}$$

This gives $\frac{n^2}{4} = \frac{n^2}{2} - \frac{2}{n} \cdot \tau$. Solving for $E[\tau]$ gives $E[\tau] = \frac{n^3}{8} = O(n^3)$.

B.7.2 Forbidden runs

Given a sequence of coin-flips, a **run** is a maximal subsequence of either all heads or all tails. The **length** of a run is the number of flips in the run. For example, the sequence **HHHTTHTHH** has five runs: a run **HHH** of length 3, a run **TT** of length 2, a run **H** of length 1, a run **T** of length 1, and a run **HH** of length 2.

Suppose we are attempting to transmit a sequence of n fair, independent coin-flips across a communication channel that uses a run of length k to signal an emergency condition. To avoid false alarms, we must encode our sequence to include no runs of length k . We do this by encoding any run of length $\ell \geq k$ as a run of length $\ell + 1$ of the same value. For example, when $k = 2$, we would encode the previous example **HHHTTHTHH** as **HHHHTTTHTHHH**.

As a function of n and k , what is the expected length of the encoded sequence?

Solution

Let i range from 1 to n , and let X_i be the i -th coin-flip in the sequence. Let R_i be the indicator variable for the event that there is a run of length k or more starting at position i . Then the expected number of runs of length k or more is given by $\sum_{i=1}^n E[R_i]$ by linearity of expectation, and because each such run adds one to the length of the encoded sequence, the expected encoded length is $n + \sum_{i=1}^n E[R_i]$.

To compute $E[R_i]$, we consider three cases:

1. If $i = 1$, then $R_i = 1$ when X_1, \dots, X_k are all H or all T. This gives $E[R_i] = 2 \cdot 2^{-k} = 2^{-k+1}$.

2. If $1 < i \leq n - k + 1$, then $R_i = 1$ when X_1, \dots, X_k are all H or all T, and X_{i-1} is not this common value (because otherwise the run starts earlier). This gives $E[R_i] = 2 \cdot 2^{-k-1} = 2^{-k}$.
3. If $i > n - k + 1$, then $R_i = 0$, because there aren't enough coins left to get a run of length k .

Summing over all the cases gives

$$\begin{aligned} \sum_{i=1}^n E[R_i] &= 2^{-k+1} + \sum_{i=2}^{n-k+1} 2^{-k} \\ &= 2 \cdot 2^{-k} + (n - k) \cdot 2^{-k} \\ &= (n - k + 2) \cdot 2^{-k}. \end{aligned}$$

For the expected encoded sequence length, we must add back n to get $n + (n - k + 2) \cdot 2^{-k}$.

B.7.3 A derandomized balancing scheme

Suppose we construct an $n \times n$ matrix A where the elements A_{ij} are n^2 independent fair ± 1 random variables. Then Hoeffding's inequality tells us that

$$\Pr \left[\left| \sum_{i=1}^n \sum_{j=1}^n A_{ij} \right| \geq t \right] \leq 2 \exp \left(-\frac{t^2}{2n^2} \right).$$

Unfortunately this requires using n^2 random bits.

As an alternative, suppose we use $2n$ random bits to generate two sequences X and Y of n independent fair ± 1 random variables, and define $B_{ij} = X_i Y_j$.

Show that

$$\Pr \left[\left| \sum_{i=1}^n \sum_{j=1}^n B_{ij} \right| \geq t \right] \leq 4 \exp \left(-\frac{t}{2n} \right).$$

Solution

Start by factoring

$$\begin{aligned} \sum_{i=1}^n \sum_{j=1}^n B_{ij} &= \sum_{i=1}^n \sum_{j=1}^n X_i Y_j \\ &= \left(\sum_{i=1}^n X_i \right) \left(\sum_{j=1}^n Y_j \right). \end{aligned}$$

Each factor is a sum of n independent ± 1 random variables with expectation 0, so Hoeffding's inequality applies, giving

$$\Pr \left[\left| \sum_{i=1}^n X_i \right| \geq s \right] \leq 2 \exp \left(-\frac{s^2}{2n} \right)$$

and

$$\Pr \left[\left| \sum_{j=1}^n Y_j \right| \geq s \right] \leq 2 \exp \left(-\frac{s^2}{2n} \right).$$

If neither of these events holds, then $\left| \sum_{i=1}^n \sum_{j=1}^n B_{ij} \right| = \left| \sum_{i=1}^n X_i \right| \cdot \left| \sum_{j=1}^n Y_j \right| < s^2$.

So by the union bound,

$$\begin{aligned} \Pr \left[\left| \sum_{i=1}^n \sum_{j=1}^n B_{ij} \right| \geq s^2 \right] &\leq \Pr \left[\left| \sum_{i=1}^n X_i \right| \geq s \right] + \Pr \left[\left| \sum_{j=1}^n Y_j \right| \geq s \right] \\ &\leq 4 \exp \left(-\frac{s^2}{2n} \right). \end{aligned}$$

Now substitute t for s^2 .

Appendix C

Sample assignments from Fall 2016

C.1 Assignment 1: due Sunday, 2016-09-18, at 17:00

Bureaucratic part

Send me email! My address is james.aspnes@gmail.com.

In your message, include:

1. Your name.
2. Your status: whether you are an undergraduate, grad student, auditor, etc.
3. Whether you are taking the course as CPSC 469 or CPSC 569.
4. Anything else you'd like to say.

(You will not be graded on the bureaucratic part, but you should do it anyway.)

C.1.1 Bubble sort

Algorithm [C.1](#) implements one pass of the (generally pretty slow) **bubble sort** algorithm. This involves comparing each array location with its successor, and swapping the elements if they are out of order. The full bubble sort

```

1 procedure BubbleSortOnePass( $A, n$ )
2   for  $i \leftarrow 1$  to  $n - 1$  do
3     if  $A[i] > A[i + 1]$  then
4       Swap  $A[i]$  and  $A[i + 1]$ 

```

Algorithm C.1: One pass of bubble sort

algorithm repeats this loop until the array is sorted, but here we just do it once.

Suppose that A starts out as a uniform random permutation of distinct elements. As a function of n , what is the exact expected number of swaps performed by Algorithm C.1?

Solution

The answer is $n - H_n$, where $H_n = \sum_{i=1}^n \frac{1}{i}$ is the n -th harmonic number. There are a couple of ways to prove this. Below, we let A_i represent the original contents of $A[i]$, before doing any swaps. In each case, we will use the fact that after i iterations of the loop, $A[i]$ contains the largest of A_1, \dots, A_i ; this is easily proved by induction on i .

- Let X_{ij} be the indicator variable for the event that A_i is eventually swapped with A_j . For this to occur, A_i must be bigger than A_j , and must be present in $A[j-1]$ after $j-1$ passes through the loop. This happens if and only if A_i is the largest value in A_1, \dots, A_j . Because these values are drawn from a uniform random permutation, by symmetry A_i is largest with probability exactly $1/j$. So $E[X_{ij}] = 1/j$.

Now sum X_{ij} over all pairs $i < j$. It is easiest to do this by summing

over j first:

$$\begin{aligned}
 \mathbb{E} \left[\sum_{i < j} X_{ij} \right] &= \sum_{i < j} \mathbb{E} [X_{ij}] \\
 &= \sum_{j=2}^n \sum_{i=1}^{j-1} \mathbb{E} [X_{ij}] \\
 &= \sum_{j=2}^n \sum_{i=1}^{j-1} \frac{1}{j} \\
 &= \sum_{j=2}^n \frac{j-1}{j} \\
 &= \sum_{j=1}^n \frac{j-1}{j} \\
 &= \sum_{j=1}^n \left(1 - \frac{1}{j}\right) \\
 &= n - \sum_{j=1}^n \frac{1}{j} \\
 &= n - H_n.
 \end{aligned}$$

- Alternatively, let's count how many values are *not* swapped from $A[i]$ to $A[i-1]$. We can then subtract from n to get the number that are.

Let Y_i be the indicator variable for the event that A_i is not swapped into $A[i-1]$. This occurs if, when testing $A[i-1]$ against $A[i]$, $A[i]$ is larger. Since we know that at this point $A[i-1]$ is the largest value among A_1, \dots, A_{i-1} , $Y_i = 1$ if and only if A_i is greater than all of A_1, \dots, A_{i-1} , or equivalently if A_i is the largest value in A_1, \dots, A_i . Again by symmetry we have $\mathbb{E}[Y_i] = \frac{1}{i}$, and summing over all i gives an expected H_n values that are not swapped down. So there are $n - H_n$ values on average that are swapped down, which also gives the expected number of swaps.

C.1.2 Finding seats

A huge lecture class with n students meets in a room with $m \geq n$ seats. Because the room is dark, loud, and confusing, the students adopt the following seat-finding algorithm:

1. The students enter the room one at a time. No student enters until the previous student has found a seat.
2. To find a seat, a student chooses one of the m seats uniformly at random. If the seat is occupied, the student again picks a seat uniformly at random, continuing until they either find a seat or have made k attempts that led them to an occupied seat.
3. If a student fails to find a seat in k attempts, they contact an Undergraduate Seating Assistant, who leads them to an unoccupied seat.

Give an asymptotic (big- Θ) expression for the number of students who are helped by Undergraduate Seating Assistants, as a function of n , m , and k , where k is a constant independent of n and m . As with any asymptotic formula, this expression should work for large n and m ; and should be as simple as possible.¹

Solution

Define the i -th student to be the student who takes their seat after i students have already sat (note that we are counting from zero here, which makes things a little easier). Let X_i be the indicator variable for the event that the i -th student does not find a seat on their own and needs help from a USA. Each attempt to find a seat fails with probability i/m . Since each attempt is independent of the others, the probability that all k attempts fail is $(i/m)^k$.

The number of students who need help is $\sum_{i=0}^{n-1} X_i$, so the expected number is

$$\begin{aligned} \mathbf{E} \left[\sum_{i=0}^{n-1} X_i \right] &= \sum_{i=0}^{n-1} \mathbf{E}[X_i] \\ &= \sum_{i=0}^{n-1} (i/m)^k \\ &= m^{-k} \sum_{i=0}^{n-1} i^k. \end{aligned}$$

Unfortunately there is no simple expression for $\sum i^k$ that works for all k . However, we can approximate it from above and below using integrals.

¹An earlier version of this problem allowed k to grow with n , which causes some trouble with the asymptotics.

From below, we have

$$\begin{aligned}
 \sum_{i=0}^{n-1} i^k &= \sum_{i=1}^{n-1} i^k \\
 &\geq \int_0^{n-1} x^k dx \\
 &= \left[\frac{1}{k+1} x^{k+1} \right]_0^{n-1} \\
 &= \frac{1}{k+1} (n-1)^{k+1} \\
 &= \Omega\left(\frac{1}{k+1} n^{k+1}\right).
 \end{aligned}$$

From above, an almost-identical computation gives

$$\begin{aligned}
 \sum_{i=0}^{n-1} i^k &\leq \int_0^n x^k dx \\
 &= \left[\frac{1}{k+1} x^{k+1} \right]_0^n \\
 &= \frac{1}{k+1} n^{k+1} \\
 &= O\left(\frac{1}{k+1} n^{k+1}\right).
 \end{aligned}$$

Combining the two cases gives that the sum is $\Theta\left(\frac{1}{k+1} n^{k+1}\right)$. We must also include m^{-k} , giving the full answer

$$\Theta\left(\frac{1}{k+1} n^{k+1} m^{-k}\right).$$

I like writing this as $\Theta\left(n \frac{(n/m)^k}{k+1}\right)$ to emphasize that the cost per student is $\Theta\left(\frac{(n/m)^k}{k+1}\right)$, but there are a lot of different ways to write the same thing. Note that since we are dealing with asymptotics, we could also replace the $\frac{1}{k+1}$ with just $\frac{1}{k}$ and still be correct, but that seems less informative somehow.

C.2 Assignment 2: due Thursday, 2016-09-29, at 23:00

C.2.1 Technical analysis

It is well known that many human endeavors—including **sports reporting**, **financial analysis**, and **Dungeons and Dragons**—involve building narra-

tives on top of the output of a weighted random number generator [Mun11], a process that is now often automated [Hol16]. For this problem, we are going to consider a simplified model of automated financial analysis.

Let us model the changes in a stock price over n days using a sequence of independent ± 1 random variables X_1, X_2, \dots, X_n , where each X_i is ± 1 with probability $1/2$. Let $S_i = \sum_{j=1}^i X_j$, where $i \in \{0, \dots, n\}$, be the price after i days.

Our automated financial reporter will declare a **dead cat bounce** if a stock falls in price for two days in a row, followed by rising in price, followed by falling in price again. Formally, a dead cat bounce occurs on day i if $i \geq 4$ and $S_{i-4} > S_{i-3} > S_{i-2} < S_{i-1} > S_i$. Let D be the number of dead cat bounces over the n days.

1. What is $E[D]$?
2. What is $\text{Var}[D]$?
3. Our investors will shut down our system if it doesn't declare at least one dead cat bounce during the first n days. What upper bound you can get on $\Pr[D = 0]$ using Chebyshev's inequality?
4. Show $\Pr[D = 0]$ is in fact exponentially small in n .

Note added 2016-09-28: It's OK if your solutions only work for sufficiently large n . This should save some time dealing with weird corner cases when n is small.

Solution

1. Let D_i be the indicator variable for the event that a dead cat bounce occurs on day i . Let $p = E[D_i] = \Pr[D_i = 1]$. Then

$$p = \Pr[X_{i-3} = -1 \wedge X_{i-2} = -1 \wedge X_{i-1} = +1 \wedge X_i = -1] = \frac{1}{16},$$

since the X_i are independent.

Then

$$\begin{aligned} E[D] &= E\left[\sum_{i=4}^n D_i\right] \\ &= \sum_{i=4}^n E[D_i] \\ &= \sum_{i=4}^n \frac{1}{16} \\ &= \frac{n-3}{16}, \end{aligned}$$

assuming n is at least 3.

2. For variance, we can't just sum up $\text{Var}[D_i] = p(1-p) = \frac{15}{256}$, because the D_i are not independent. Instead, we have to look at covariance.

Each D_i depends on getting a particular sequence of four values for X_{i-3} , X_{i-2} , X_{i-1} , and X_i . If we consider how D_i overlaps with D_j for $j > i$, we get these cases:

Variable	Pattern	Correlation
D_i	---+	
D_{i+1}	----	inconsistent
D_{i+2}	---+	inconsistent
D_{i+3}	---+	overlap in one place

For larger j , there is no overlap, so D_i and D_j are independent for $j \geq 4$, giving $\text{Cov}[D_i, D_j] = 0$ for these cases.

Because D_i can't occur with D_{i+1} or D_{i+2} , when $j \in \{i+1, i+2\}$ we have

$$\text{Cov}[D_i, D_j] = E[D_i D_j] - E[D_i] E[D_j] = 0 - (1/16)^2 = -1/256.$$

When $j = i+3$, the probability that both D_i and D_j occur is $1/2^7$, since we only need to get 7 coin-flips right. This gives

$$\text{Cov}[D_i, D_{i+3}] = E[D_i D_{i+3}] - E[D_i] E[D_{i+3}] = 1/128 - (1/16)^2 = 1/256.$$

Now apply (5.1.5):

$$\begin{aligned}
 \text{Var}[D] &= \text{Var}\left[\sum_{i=4}^n D_i\right] \\
 &= \sum_{i=4}^n \text{Var}[D_i] + 2 \sum_{i=4}^{n-1} \text{Cov}[D_i, D_{i+1}] + 2 \sum_{i=4}^{n-2} \text{Cov}[D_i, D_{i+2}] + 2 \sum_{i=4}^{n-3} \text{Cov}[D_i, D_{i+3}] \\
 &= (n-3) \frac{15}{256} + 2(n-4) \frac{-1}{256} + 2(n-5) \frac{-1}{256} + 2(n-6) \frac{1}{256} \\
 &= \frac{13n-39}{256},
 \end{aligned}$$

where to avoid special cases, we assume n is at least 6.

3. At this point we are just doing algebra. For $n \geq 6$, we have:

$$\begin{aligned}
 \Pr[D = 0] &\leq \Pr[|D - \mathbb{E}[D]| \geq \mathbb{E}[D]] \\
 &= \Pr\left[|D - \mathbb{E}[D]| \geq \frac{n-3}{16}\right] \\
 &\leq \frac{(13n-39)/256}{((n-3)/16)^2} \\
 &= \frac{13n-39}{(n-3)^2} \\
 &= \frac{13}{n-3}.
 \end{aligned}$$

This is $\Theta(1/n)$, which is not an especially strong bound.

4. Here is an upper bound that avoids the mess of computing the exact probability, but is still exponentially small in n . Consider the events $[D_{4i} = 1]$ for $i \in \{1, 2, \dots, \lfloor n/4 \rfloor\}$. These events are independent since they are functions of non-overlapping sequences of increments. So we can compute $\Pr[D = 0] \leq \Pr[D_{4i} = 0, \forall i \in \{1, \dots, \lfloor n/4 \rfloor\}] = (15/16)^{\lfloor n/4 \rfloor}$.

This expression is a little awkward, so if we want to get an asymptotic estimate we can simplify it using $1+x \leq e^x$ to get $(15/16)^{\lfloor n/4 \rfloor} \leq \exp(-(1/16)\lfloor n/4 \rfloor) = O(\exp(-n/64))$.

With a better analysis, we should be able to improve the constant in the exponent; or, if we are real fanatics, calculate $\Pr[D = 0]$ exactly. But this answer is good enough given what the problems asks for.

C.2.2 Faulty comparisons

Suppose we want to do a comparison-based sort of an array of n distinct elements, but an adversary has sabotaged our comparison subroutine. Specifically, the adversary chooses a sequence of times t_1, t_2, \dots, t_n , such that for each i , the t_i -th comparison we do returns the wrong answer. Fortunately, this can only happen n times. In addition, the adversary is oblivious: thought it can examine the algorithm before choosing the input and the times t_i , it cannot predict any random choices made by the algorithm.

A simple deterministic work-around is to replace each comparison in our favorite $\Theta(n \log n)$ -comparison sorting algorithm with $2n + 1$ comparisons, and take the majority value. But this gives us a $\Theta(n^2 \log n)$ -comparison algorithm. We would like to do better using randomization.

Show that in this model, for every fixed $c > 0$ and sufficiently large n , it is possible to sort correctly with probability at least $1 - n^{-c}$ using $O(n \log^2 n)$ comparisons.

Solution

We'll adapt QuickSort. If the bad comparisons occurred randomly, we could just take the majority of $\Theta(\log n)$ faulty comparisons to simulate a non-faulty comparison with high probability. But the adversary is watching, so if we do these comparisons at predictable times, it could hit all $\Theta(\log n)$ of them and stay well within its fault budget. So we are going to have to be more sneaky.

Here's this idea: Suppose we have a list of n comparisons $\langle x_1, y_1 \rangle, \langle x_2, y_2 \rangle, \dots, \langle x_n, y_n \rangle$ that we want to perform. We will use a subroutine that carries out these comparisons with high probability by doing $kn \ln n + n$ possibly-faulty comparisons, with k a constant to be chosen below, where each of the possibly-faulty comparisons looks at x_r and y_r where each r is chosen independently and uniformly at random. The subroutine collects the results of these comparisons for each pair $\langle x_i, y_i \rangle$ and takes the majority value.

At most n of these comparisons are faulty, and we get an error only if some pair $\langle x_i, y_i \rangle$ gets more faulty comparisons than non-faulty ones. Let B_i be the number of bad comparisons of the pair $\langle x_i, y_i \rangle$ and G_i the number of good comparisons. We want to bound the probability of the event $B_i > G_i$.

The probability that any particular comparison lands on a particular pair is exactly $1/n$; so $E[B_i] = 1$ and $E[G_i] = k \ln n$. Now apply Chernoff bounds. From (5.2.4), $\Pr[B_i \geq (k/2) \ln n] \leq 2^{-(k/2) \ln n} = n^{-(k \ln 2)/2}$ provided $(k/2) \ln n$ is at least 6. In the other direction, (5.2.6) says that $\Pr[G_i \leq (k/2) \log n] \leq e^{-(1/2)^2 k \log n/2} = n^{-k/8}$. So we have a probability of

at most $n^{-k/2} + n^{-k/8}$ that we get the wrong result for this particular pair, and from the union bound we have a probability of at most $n^{1-k/2} + n^{1-k/8}$ that we get the wrong result for any pair. We can simplify this a bit by observe that n must be at least 2 (or we have nothing to sort), so we can put a bound of $n^{2-k/8}$ on the probability of any error among our n simulated comparisons, provided $k \geq 12/\ln 2$. We'll choose $k = \max(12/\ln 2, 8(c+3))$ to get a probability of error of at most n^{-c-1} .

So now let's implement QuickSort. The first round of QuickSort picks a pivot and performs $n - 1$ comparisons. We perform these comparisons using our subroutine (note we can always throw in extra comparisons to bring the total up to n). Now we have two piles on which to run the algorithm recursively. Comparing all nodes in each pile to the pivot for that pile requires $n - 3$ comparisons, which can again be done by our subroutine. At the next state, we have four piles, and we can again perform the $n - 7$ comparisons we need using the subroutine. Continue until all piles have size 1 or less; this takes $O(\log n)$ rounds with high probability. Since each round does $O(n \log n)$ comparisons and fails with probability at most n^{-c-1} , the entire process takes $O(n \log^2 n)$ comparisons and fails with probability at most $n^{-c-1}O(\log n) \leq n^{-c}$ when n is sufficiently large.

C.3 Assignment 3: due Thursday, 2016-10-13, at 23:00

C.3.1 Painting with sprites

In the early days of computing, memory was scarce, and primitive home game consoles like the **Atari 2600** did not possess enough RAM to store the image they displayed on your television. Instead, these machines used a graphics chip that would composite together fixed **sprites**, bitmaps stored in ROM that could be displayed at locations specified by the CPU.² One common programming trick of this era was to repurpose existing data in memory as sprites, for example by searching through a program's machine code instructions to find sequences of bytes that looked like explosions when displayed.

²In the actual Atari 2600 graphics hardware, the CPU did this not by writing down the locations in RAM somewhere (which would take memory), but by counting instruction cycles since the graphics chip started refreshing the screen and screaming "display this sprite NOW!" when the CRT beam got to the right position. For this problem we will assume a more modern approach, where we can just give a list of locations to the graphics hardware.

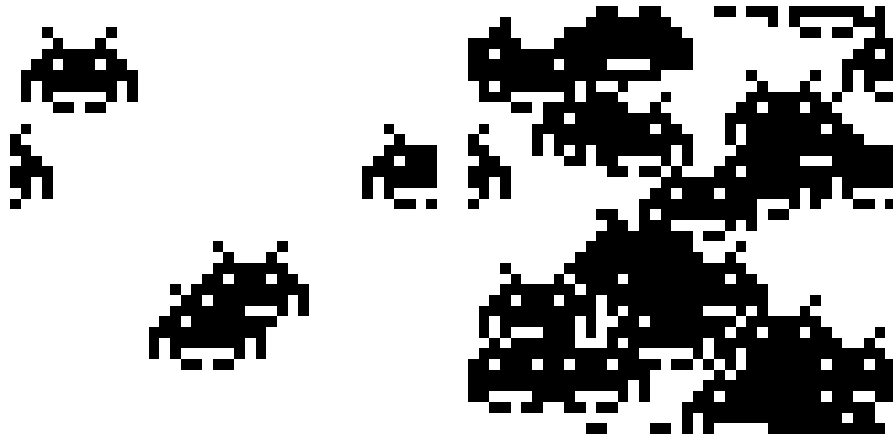


Figure C.1: Filling a screen with Space Invaders. The left-hand image places four copies of a 46-pixel sprite in random positions on a 40×40 screen. The right-hand image does the same thing with 24 copies.

For this problem, we will imagine that we have a graphics device that can only display one sprite. This is a bitmap consisting of m ones, at distinct relative positions $\langle y_1, x_1 \rangle, \langle y_2, x_2 \rangle, \dots, \langle y_m, x_m \rangle$. Displaying a sprite at position y, x on our $n \times n$ screen sets the pixels at positions $\langle (y + y_i) \bmod n, (x + x_i) \bmod n \rangle$ for all $i \in \{1, \dots, m\}$. The screen is initially blank (all pixels 0) and setting a pixel at some position $\langle y, x \rangle$ changes it to 1. Setting the same pixel more than once has no effect.

We would like to use these sprites to simulate white noise on the screen, by placing them at independent uniform random locations with the goal of setting roughly half of the pixels. Unfortunately, because the contents of the screen are not actually stored anywhere, we can't detect when this event occurs. Instead, we want to fix the number of sprites ℓ so that we get $1/2 \pm o(1)$ of the total number of pixels set to 1 with high probability, by which we mean that $(1/2 \pm o(1))n^2$ total pixels are set with probability $1 - n^{-c}$ for any fixed $c > 0$ and sufficiently large n , assuming m is fixed.

An example of this process, using Taito Corporation's classic **Space Invader** bitmap, is shown in Figure C.1.

Compute the value of ℓ that we need as a function of n and m , and show that this choice of ℓ does in fact get $1/2 \pm o(1)$ of the pixels set with high probability.

Solution

We will apply the following strategy. First, we'll choose ℓ so that each individual pixel is set with probability close to $1/2$. Linearity of expectation then gives roughly $n^2/2$ total pixels set on average. To show that we get close to this number with high probability, we'll use the method of bounded differences.

The probability that pasting in a single copy of the sprite sets a particular pixel is exactly m/n^2 . So the probability that the pixel is not set after ℓ sprites is $(1 - m/n^2)^\ell$. This will be exactly $1/2$ if $\ell = \log_{1-m/n^2} 1/2 = \frac{\ln(1/2)}{\ln(1-m/n^2)}$.

Since ℓ must be an integer, we can just round this quantity to the nearest integer to pick our actual ℓ . For example, when $n = 40$ and $m = 46$, $\frac{\ln(1/2)}{\ln(1-46/40^2)} = 23.7612\dots$ which rounds to 24.

For large n , we can use the approximation $\ln(1+x) \approx x$ (which holds for small x) to approximate ℓ as $(n^2 \ln 2/m) + o(1)$. We will use this to get a concentration bound using (5.3.12).

We have ℓ independent random variables corresponding to the positions of the ℓ sprites. Moving a single sprite changes the number of set pixels by at most m . So the probability that the total number of set pixels deviates from its expectation by more than $an\sqrt{\ln n}$ is bounded by

$$\begin{aligned} 2 \exp\left(-\frac{(an\sqrt{\ln n})^2}{\ell m^2}\right) &= 2 \exp\left(-\frac{a^2 n^2 \ln n}{(n^2 \ln 2/m + o(1))m^2}\right) \\ &= 2 \exp\left(-\frac{a^2 \ln n}{m \ln 2 + o(m^2/n^2)}\right) \\ &\leq \exp\left(-(a^2/m) \ln n\right) \\ &= n^{-a^2/m}, \end{aligned}$$

where the inequality holds when n is sufficiently large. Now choose $a = \sqrt{cm}$ to get a probability of deviating by more than $n\sqrt{cm \ln n}$ of at most n^{-c} . Since $n\sqrt{cm \ln n} = o(1)n^2$, this gives us our desired bound.

C.3.2 Dynamic load balancing

A low-cost competitor to better-managed data center companies offers a scalable dynamic load balancing system that supplies a new machine whenever a new job shows up, so that there are always n machines available to run n jobs. Unfortunately, one of the ways the company cuts cost is by not

hiring programmers to replace the code from their previous randomized load balancing mechanism, so when the i -th job arrives, it is still assigned uniformly at random to one of the i available machines. This means that job 1 is always assigned to machine 1; job 2 is assigned with equal probability to machine 1 or 2; job 3 is assigned with equal probability to machine 1, 2, or 3; and so on. These choices are all independent. The company claims that the maximum load is still not too bad, and that the reduced programming costs that they pass on to their customers make up for their poor quality control and terrible service.

Justify the first part of this claim by showing that, with high probability, the most loaded machine after n jobs has arrived has load $O(\log n)$.

Solution

Let X_{ij} be the indicator variable for the event that machine i gets job j . Then $E[X_{ij}] = 1/j$ for all $i \leq j$, and $E[X_{ij}] = 0$ when $i > j$.

Let $Y_i = \sum_{j=1}^n X_{ij}$ be the load on machine i .

Then $E[Y_i] = E\left[\sum_{j=1}^n X_{ij}\right] = \sum_{j=i}^n 1/j \leq \sum_{j=1}^n 1/j = H_n \leq \ln n + 1$.

From (5.2.4), we have $\Pr[Y_i \geq R] \leq 2^{-R}$ as long as $R > 2eE[Y_i]$. So if we let $R = (c+1)\lg n$, we get a probability of at most n^{-c-1} that we get more than R jobs on machine i . Taking the union bound over all i gives a probability of at most n^{-c} that any machine gets a load greater than $(c+1)\lg n$. This works as long as $(c+1)\lg n \geq 2e(\ln n + 1)$, which holds for sufficiently large c . For smaller c , we can just choose a larger value c' that does work, and get that $\Pr[\max Y_i \geq (c'+1)\lg n] \leq n^{-c'} \leq n^{-c}$.

So for any fixed c , we get that with probability at least $1 - n^{-c}$ the maximum load is $O(\log n)$.

C.4 Assignment 4: due Thursday, 2016-11-03, at 23:00

C.4.1 Re-rolling a random treap

Suppose that we add to a random treap (see §6.3) an operation that re-rolls the priority of the node with a given key x . This replaces the old priority for x with a new priority generated uniformly at random, independently of the old priority any other priorities in the treap. We assume that the range of priorities is large enough that the probability that this produces a duplicate is negligible, and that the choice of which node to re-roll is done obliviously,

without regard to the current priorities of nodes or the resulting shape of the tree.

Changing the priority of x may break the heap property. To fix the heap property, we either rotate x up (if its priority now exceeds its parent's priority) or down (if its priority is now less than that of one or both of its children). In the latter case, we rotate up the child with the higher priority. We repeat this process until the heap property is restored.

Compute the best constant upper bound you can on the expected number of rotations resulting from executing a re-roll operation.

Solution

The best possible bound is at most 2 rotations on average in the worst case.

There is an easy but incorrect argument that 2 is an upper bound, which says that if we rotate up, we do at most the same number of rotations as when we insert a new element, and if we rotate down, we do at most the same number of rotations as when we delete an existing element. This gives the right answer, but for the wrong reasons: the cost of deleting x , conditioned on the event that re-rolling its priority gives a lower priority, is likely to be greater than 2, since the conditioning means that x is likely to be higher up in the tree than average; the same thing happens in the other direction when x moves up. Fortunately, it turns out that the fact that we don't necessarily rotate x all the way to or from the bottom compensates for this issue.

This can be formalized using the following argument, for which I am indebted to Adit Singha and Stanislaw Swidinski. Fix some element x , and suppose its old and new priorities are p and p' . If $p < p'$, we rotate up, and the sequence of rotations is exactly the same as we get if we remove all elements of the original treap with priority p or less and then insert a new element x with priority p' . But now if we condition on the number k of elements with priority greater than p , their priorities together with p' are all independent and identically distributed, since they are all obtained by taking their original distribution and conditioning on being greater than p . So all $(k + 1)!$ orderings of these priorities are equally likely, and this means that we have the same expected cost as an insertion into a treap with k elements, which is at most 2. Averaging over all k shows that the expected cost of rotating up is at most 2, and, since rotating down is just the reverse of this process with a reversed distribution on priorities (since we get it by choosing p' as our old priority and p as the new one), the expected cost of rotating down is also at most 2. Finally, averaging the up and down cases gives that the expected number of rotations without conditioning on anything is at

most 2.

We now give an exact analysis of the expected number of rotations, which will show that 2 is in fact the best bound we can hope for.

The idea is to notice that whenever we do a rotation involving x , we change the number of ancestors of x by exactly one. This will always be a decrease in the number of ancestors if the priority of x went up, or an increase if the priority of x went down, so the total number of rotations will be equal to the change in the number of ancestors.

Letting A_i be the indicator for the event that i is an ancestor of x before the re-roll, and A'_i the indicator for the event that i is an ancestor of x after the re-roll, then the number of rotations is just $|\sum_i A_i - \sum_i A'_i|$, which is equal to $\sum_i |A_i - A'_i|$ since we know that all changes $A_i - A'_i$ have the same sign. So the expected number of rotations is just $E[\sum_i |A_i - A'_i|] = \sum_i E[|A_i - A'_i|]$, by linearity of expectation.

So we have to compute $E[|A_i - A'_i|]$. Using the same argument as in §6.3.2, we have that $A_i = 1$ if and only if i has the highest initial priority of all elements in the range $[\min(i, x), \max(i, x)]$, and the same holds for A'_i if we consider updated priorities. So we want to know the probability that changing only the priority of x to a new random value changes whether i has the highest priority.

Let $k = \max(i, x) - \min(i, x) + 1$ be the number of elements in the range under consideration. To avoid writing a lot of mins and maxes, let's renumber these elements as 1 through k , with $i = 1$ and $x = k$ (this may involve flipping the sequence if $i > x$). Let X_1, \dots, X_k be the priorities of these elements, and let X'_k be the new priority of x . These $k + 1$ random variables are independent and identically distributed, so conditioning on the event that no two are equal, all $(k + 1)!$ orderings of their values are equally likely.

So now let us consider how many of these orderings result in $|A_i - A'_i| = 1$. For A_i to be 1, X_1 must exceed all of X_2, \dots, X_k . For A'_i to be 0, X_1 must not exceed all of $X_2, \dots, X_{k-1}, X'_k$. The intersection of these events is when $X'_k > X_1 > \max(X_2, \dots, X_k)$. Since X_2, \dots, X_k can be ordered in any of $(k - 1)!$ ways, this gives

$$\Pr [A_i = 1 \wedge A'_i = 0] = \frac{(k - 1)!}{(k + 1)!} = \frac{1}{k(k + 1)}.$$

In the other direction, for A_i to be 0 and A'_i to be 1, we must have $X_k >$

$X_1 > \max(X_1, \dots, X_{k-1}, X_k)$. This again gives

$$\Pr [A_i = 0 \wedge A'_i = 1] = \frac{1}{k(k+1)}$$

and combining the two cases gives

$$\mathbb{E} [|A_i - A'_i|] = \frac{2}{k(k+1)}.$$

Now sum over all i :

$$\begin{aligned} \mathbb{E} [\text{number of rotations}] &= \sum_{i=1}^n \mathbb{E} [|A'_i - A_i|] \\ &= \sum_{i=1}^{x-1} \frac{2}{(x-i+1)(x-i+2)} + \sum_{i=x+1}^n \frac{2}{(i-x+1)(i-x+2)} \\ &= \sum_{k=2}^x \frac{2}{k(k+1)} + \sum_{k=2}^{n-x+1} \frac{2}{k(k+1)} \\ &= \sum_{k=2}^x \left(\frac{2}{k} - \frac{2}{k+1} \right) + \sum_{k=2}^{n-x+1} \left(\frac{2}{k} - \frac{2}{k+1} \right) \\ &= 2 - \frac{2}{x+1} - \frac{2}{n-x+2}. \end{aligned}$$

In the limit as n goes to infinity, choosing $x = \lfloor n/2 \rfloor$ makes both fractional terms converge to 0, making the expected number of rotations arbitrarily close to 2. So 2 is the best possible bound that doesn't depend on n .

C.4.2 A defective hash table

A foolish programmer implements a hash table with m buckets, numbered $0, \dots, m-1$, with the property that any value hashed to an even-numbered bucket is stored in a linked list as usual, but any value hashed to an odd-numbered bucket is lost.

1. Suppose we insert a set S of $n = |S|$ items into this hash table, using a hash function h chosen at random from a strongly 2-universal hash family H . Show that there is a constant c such that, for any $t > 0$, the probability that at least $n/2 + t$ items are lost is at most cn/t^2 .
2. Suppose instead that we insert a set S of $n = |S|$ items into this hash table, using a hash function h chosen at random from a hash family

H' that is 2-universal, but that may not be strongly 2-universal. Show that if $n \leq m/2$, it is possible for an adversary that knows S to design a 2-universal hash family H' that ensures that all n items are lost with probability 1.

Solution

- Let X_i be the indicator variable for the event that the i -th element of S is hashed to an odd-numbered bucket. Since H is strongly 2-universal, $\mathbb{E}[X_i] \leq 1/2$ (with equality when m is even), from which it follows that $\text{Var}[X_i] = \mathbb{E}[X_i](1 - \mathbb{E}[X_i]) \leq 1/4$; and the X_i are pairwise independent. Letting $Y = \sum_{i=1}^n X_i$ be the total number of items lost, we get $\mathbb{E}[Y] \leq n/2$ and $\text{Var}[Y] \leq n/4$. But then we can apply Chebyshev's inequality to show

$$\begin{aligned} \Pr[Y \geq n/2 + t] &\leq \Pr[Y \geq \mathbb{E}[Y] + t] \\ &\leq \frac{\text{Var}[Y]}{t^2} \\ &\leq \frac{n/4}{t^2} \\ &= \frac{1}{4}(n/t^2). \end{aligned}$$

So the desired bound holds with $c = 1/4$.

- Write $[m]$ for the set $\{0, \dots, m-1\}$. Let $S = \{x_1, x_2, \dots, x_n\}$.

Consider the set H' of all functions $h : U \rightarrow [m]$ with the property that $h(x_i) = 2i + 1$ for each i . Choosing a function h uniformly from this set corresponds to choosing a random function conditioned on this constraint; and this conditioning guarantees that every element of S is sent to an odd-numbered bucket and lost. This gives us half of what we want.

The other half is that H' is 2-universal. Observe that for any $y \notin S$, the conditioning does not constrain $h(y)$, and so $h(y)$ is equally likely to be any element of $[m]$; in addition, $h(y)$ is independent of $h(z)$ for any $z \neq y$. So for any $y \neq z$, $\Pr[h(y) = h(z)] = 1/m$ if one or both of y and z is not an element of S , and $\Pr[h(y) = h(z)] = 0$ if both are elements of S . In either case, $\Pr[h(y) = h(z)] \leq 1/m$, and so H' is 2-universal.

C.5 Assignment 5: due Thursday, 2016-11-17, at 23:00

C.5.1 A spectre is haunting Halloween

An adult in possession of n pieces of leftover Halloween candy is distributing them to n children. The candies have known integer values v_1, v_2, \dots, v_n , and the adult starts by permuting the candies according to a uniform random permutation π . They then give candy $\pi(1)$ to child 1, candy $\pi(2)$ to child 2, and so on. Or at least, that is their plan.

Unbeknownst to the adult, child n is a very young Karl Marx, and after observing the first k candies $\pi(1), \dots, \pi(k)$, he can launch a communist revolution and seize the means of distribution. This causes the remaining $n - k$ candies $\pi(k + 1)$ through $\pi(n)$ to be distributed evenly among the remaining $n - k$ children, so that each receives a value equal to exactly $\frac{1}{n-k} \sum_{i=k+1}^n v_{\pi(i)}$. Karl may declare the revolution at any time before the last candy is distributed (even at time 0, when no candies have been distributed). However, his youthful understanding of the mechanisms of historical determinism are not detailed enough to predict the future, so his decision to declare a revolution after seeing k candies can only depend on those k candies and his knowledge of the values of all of the candies but not their order.

Help young Karl optimize his expected take by devising an algorithm that takes as input v_1, \dots, v_n and $v_{\pi(1)}, \dots, v_{\pi(k)}$, where $0 \leq k < n$, and outputs whether or not to launch a revolution at this time. Compute Karl's exact expected return as a function of v_1, \dots, v_n when running your algorithm, and show that no algorithm can do better.

Solution

To paraphrase an often-misquoted line of Trotsky's, young Karl Marx may not recognize the Optional Stopping Theorem, but the Optional Stopping does not permit him to escape its net. No strategy, no matter how clever, can produce an expected return better or worse than simply waiting for the last candy.

Let X_t be the expected return if Karl declares the revolution after seeing t cards. We will show that $\{X_t, \mathcal{F}_t\}$ is a martingale where each \mathcal{F}_t is the σ -algebra generated by the random variables $v_{\pi(1)}$ through $v_{\pi(t)}$.

The value of X_t is exactly

$$X_t = \frac{1}{n-t} \sum_{i=t+1}^n v_{\pi(i)} = \frac{1}{n-t} \left(\sum_{i=t+1}^n v_n - \sum_{i=1}^t v_{\pi(i)} \right).$$

This is also $\mathbb{E} [v_{\pi(t+1)} \mid \mathcal{F}_t]$, since the next undistributed candy is equally likely to be any of the remaining candies, and \mathcal{F}_t determines the value of $v_{\pi(i)}$ for all $i \leq t$. So

$$\begin{aligned} \mathbb{E} [X_{t+1} \mid \mathcal{F}_t] &= \mathbb{E} \left[\frac{1}{n-t-1} \left(\sum_{i=1}^n v_i - \sum_{i=1}^{t+1} v_{\pi(i)} \right) \mid \mathcal{F}_t \right] \\ &= \frac{1}{n-t-1} \left(\sum_{i=1}^n v_i - \sum_{i=1}^t v_{\pi(i)} \right) - \frac{1}{n-t-1} \mathbb{E} [v_{\pi(t+1)} \mid \mathcal{F}_T] \\ &= \frac{n-t}{n-t-1} X_t - \frac{1}{n-t-1} X_t \\ &= X_t. \end{aligned}$$

Now fix some strategy for Karl, and let τ be the time at which he launches the revolution. Then $\tau < n$ is a stopping time with respect to the \mathcal{F}_t , and the Optional Stopping Theorem (bounded time version) says that $\mathbb{E} [X_\tau] = \mathbb{E} [X_0]$. So any strategy is equivalent (in expectation) to launching the revolution immediately.

C.5.2 Colliding robots on a line

Suppose we have k robots on a line of n positions, numbered 1 through n . No two robots can pass each other or occupy the same position, so we can specify the positions of all the robots at time t as an increasing vector $X^t = \langle X_1^t, X_2^t, \dots, X_k^t \rangle$. At each time t , we pick one of the robots i uniformly at random, and also choose a direction $d = \pm 1$ uniformly and independently at random. Robot i moves to position $X^{t+1} = X_i^t + d$ if (a) $0 \leq X_i^t + d < n$, and (b) no other robot already occupies position $X_i^t + d$. If these conditions do not hold, robot i stays put, and $X_i^{t+1} = X_i^t$.

1. Given k and n , determine the stationary distribution of this process.
2. Show that the mixing time t_{mix} for this process is polynomial in n .

Solution

1. The stationary distribution is a uniform distribution on all $\binom{n}{k}$ placements of the robots. To prove this, observe that two vectors increasing vectors x and y are adjacent if and only if there is some i such that $x_j = y_j$ for all $j \neq i$ and $x_i + d = y_i$ for some $d \in \{-1, +1\}$. In this case, the transition probability p_{xy} is $\frac{1}{2n}$, since there is a $\frac{1}{n}$ chance that we choose i and a $\frac{1}{2}$ chance that we choose d . But this is the same as the probability that starting from y we choose i and $-d$. So we have $p_{xy} = p_{yx}$ for all adjacent x and y , which means that a uniform distribution π satisfies $\pi_x p_{xy} = \pi_y p_{yx}$ for all x and y .

To show that this stationary distribution is unique, we must show that there is at least one path between any two states x and y . One way to do this is to show that there is a path from any state x to the state $\langle 1, \dots, k \rangle$, where at each step we move the lowest-index robot i that is not already at position i . Since we can reverse this process to get to y , this gives a path between any x and y that occurs with nonzero probability.

2. This one could be done in a lot of ways. Below I'll give sketches of three possible approaches, ordered by increasing difficulty. The first reduces to card shuffling by adjacent swaps, the second uses an explicit coupling, and the third uses conductance.

Of these approaches, I am personally only confident of the coupling argument, since it's the one I did before handing out the problem, and indeed this is the only one I have written up in enough detail below to be even remotely convincing. But the reduction to card shuffling is also pretty straightforward and was used in several student solutions, so I am convinced that it can be made to work as well. The conductance idea I am not sure works at all, but it seems like it could be made to work with enough effort.

- (a) Let's start with the easy method. Suppose that instead of colliding robots, we have a deck of n cards, of which k are specially marked. Now run a shuffling algorithm that swaps adjacent cards at each step. If we place a robot at the position of each marked card, the trajectories of the robots follow the pretty much the same distribution as in the colliding-robots process. This is trivially the case when we swap a marked card with an unmarked card (a robot moves), but it also works when we swap two marked cards

(no robot moves, since the positions of the set of marked cards stays the same; this corresponds to a robot being stuck).

Unfortunately we can't use exactly the same process we used in §9.4.3.3, because this (a) allows swapping the cards in the first and last positions of the deck, and (b) doesn't include any moves corresponding to a robot at position 1 or n trying to move off the end of the line.

The first objection is easily dealt with, and indeed the cited result of Wilson [Wil04] doesn't allow such swaps either. The second can be dealt with by adding extra no-op moves to the card-shuffling process that occur with probability $1/n$, scaling the probabilities of the other operations to keep the sum to 1. This doesn't affect the card-shuffling convergence argument much, but it is probably a good idea to check that everything still works.

Finally, even after fixing the card-shuffling argument, we still have to argue that convergence in the card-shuffling process implies convergence in the corresponding colliding-robot process. Here is where the definition of total variation distance helps. Let C^t be the permutation of the cards after t steps, and let $f : C^t \mapsto X^t$ map permutations of cards to positions of robots. Let π and π' be the stationary distributions of the card and robot processes, respectively. Then $d_{TV}(f(C^t), f(\pi)) = \max_A |\Pr[X^t \in A] - \pi'(A)| = \max_A |\Pr[C^t \in f^{-1}(A)] - \pi(f^{-1}(A))| \leq \max_B |\Pr[C^t \in B] - \pi(B)| = d_{TV}(C^t, \pi)$. So convergence of C implies convergence of $X = f(C)$.

- (b) Alternatively, we can construct a coupling between two copies of the process X^t and Y^t , where as usual we start X^0 in our initial distribution (whatever it is) and Y^0 in the uniform stationary distribution. At each step we generate a pair (i, d) uniformly at random from $\{1, \dots, n\} \times \{-1, +1\}$. Having generated this pair, we move robot i in direction d in both processes if possible. It is clear that once $X^t = Y^t$, it will continue to hold that $X^{t'} = Y^{t'}$ for all $t' > t$.

To show convergence we must show that X^t will in fact eventually hit Y^t . To do so, let us track $Z^t = \|X^t - Y^t\|_1 = \sum_{i=1}^k |X_i^t - Y_i^t|$. We will show that Z_t is a supermartingale with respect to the history \mathcal{F}_t generated by the random variables $\langle X^s, Y^s \rangle$ for $s \leq t$. Specifically, we will show that at each step, the expected change in Z^t conditioning on X^t and Y^t is non-positive.

Fix X^t , and consider all $2n$ possible moves (i, d) . Since (i, d)

doesn't change X_j^t for any $j \neq i$, the only change in Z^t will occur if one of X_i^t and Y_i^t can move and the other can't. There are several cases:

- i. If $i = k$, $d = +1$, and exactly one of X_k^t and Y_k^t is n , then the copy of the robot not at n moves toward the copy at n , giving $Z_{t+1} - Z_t = -1$. The same thing occurs if $i = 1$, $d = -1$, and exactly one of X_1^t and Y_1^t is 1.

It is interesting to note that these two cases will account for the entire nonzero part of $E[Z_{t+1} - Z_t \mid \mathcal{F}_t]$, although we will not use this fact.

- ii. If $i < k$, $d = +1$, $X_i^t + 1 = X_{i+1}^t$, but $Y_i^t + 1 < X_{i+1}^t$, and $X_i^t \leq Y_i^t$, then robot i can move right in Y^t but not in X^t . This gives $Z^{t+1} - Z^t = +1$.

However, in this case the move $(i+1, -1)$ moves robot $i+1$ left in Y^t but not in X^t , and since $Y_{i+1}^t > Y_i^t + 1 > X_i^t = X_{i+1}^{t+1}$, this move gives $Z^{t+1} - Z^t = -1$. Since the moves $(i, +1)$ and $(i+1, -1)$ occur with equal probability, these two changes in Z^t cancel out on average.

Essentially the same analysis applies if we swap X^t with Y^t , and if we consider moves $(i, -1)$ where $i > 1$ in which one copy of the robot is blocked but not the other. If we enumerate all such cases, we find that for every move of these types that increase Z^t , there is a corresponding move that decreases Z^t ; it follows that these changes sum to zero in expectation.

Because each X_i^t and Y_i^t lies in the range $[1, n]$, it holds trivially that $Z^t \leq k(n-1) < n^2$. Consider an unbiased ± 1 random walk W^t with a reflecting barrier at n^2 (meaning that at n^2 we always move down) and an absorbing barrier at 0. Then the expected time to reach 0 starting from $W^t = x$ is given by $x(2n^2 - x) \leq n^4$. Since this unbiased random walk dominates the biased random walk corresponding to changes in Z^t , this implies that Z^t can change at most n^4 times on average before reaching 0.

Now let us ask how often Z^t changes. The analysis above implies that there is a chance of at least $1/n$ that Z^t changes in any state where some robot i with $X_i^t \neq Y_i^t$ is blocked from moving freely in one or both directions in either the X^t or Y^t process. To show that this occurs after polynomially many steps, choose some i with $X_i^t \neq Y_i^t$ that is not blocked in either process. Then i is selected every $\frac{1}{n}$ steps on average, and it moves according to a ± 1 random

walk as long as it is not blocked. But a ± 1 random walk will reach position 1 or n in $O(n^2)$ steps on average (corresponding to $O(n^3)$ steps of the original Markov chain), and i will be blocked by the left or right end of the line if it is not blocked before then. It follows that we reach a state in which Z^t changes with probability at least $\frac{1}{n}$ after $O(n^3)$ expected steps, which means that Z^t changes every $O(n^4)$ on average. Since we have previously established that Z^t reaches 0 after $O(n^4)$ expected changes, this gives a total of $O(n^8)$ expected steps of the Markov process before $Z^t = 0$, giving $X^t = Y^t$. Markov's inequality and the Coupling Lemma then give $t_{\text{mix}} = O(n^8)$.

- (c) I believe it may also be possible to show convergence using a conductance argument. The idea is to represent a state $\langle x_1, \dots, x_k \rangle$ as the differences $\langle \Delta_1, \dots, \Delta_n \rangle$, where $\Delta_1 = x_1$ and $\Delta_{i+1} = x_{i+1} - x_i$. This representation makes the set of possible states $\left\{ \Delta \mid \Delta_i \geq 1, \sum_{i=1}^k \Delta_i \leq n \right\}$ look like a simplex, a pretty well-behaved geometric object. In principle it should be possible to show an isoperimetric inequality that the lowest-conductance sets in this simplex are the N points closest to any given corner, or possibly get a lower bound on conductance using canonical paths. But the Δ version of the process is messy (two coordinates change every time a robot moves), and there are some complication with this not being a lazy random walk, so I didn't pursue this myself.

C.6 Assignment 6: due Thursday, 2016-12-08, at 23:00

C.6.1 Another colliding robot

A warehouse consists of an $n \times n$ grid of locations, which we can think of as indexed by pairs (i, j) where $1 \leq i, j \leq n$. At night, the warehouse is patrolled by a robot executing a lazy random walk. Unfortunately for the robot, there are also m crates scattered about the warehouse, and if the robot attempts to walk onto one of the m grid locations occupied by a crate, it fails to do so, and instead emits a loud screeching noise. We would like to use the noises coming from inside the warehouse to estimate m .

Formally, if the robot is at position (i, j) at time t , then at time $t + 1$ it either (a) stays at (i, j) with probability $1/2$; or (b) chooses a position (i', j') uniformly at random from $\{(i - 1, j), (i + 1, j), (i, j - 1), (i, j + 1)\}$

and attempts to move to it. If the robot chooses not to move, it makes no noise. If it chooses to move, but (i', j') is off the grid or occupied by one of the m crates, then the robot stays at (i, j) and emits the noise. Otherwise it moves to (i', j') and remains silent. The robot's position at time 0 can be any unoccupied location on the grid.

To keep the robot from getting walled in somewhere, whatever adversary placed the crates was kind enough to ensure that if a crate was placed at position (i, j) , then all of the eight positions

$$\begin{array}{ccccc} (i-1, j+1) & (i, j+1) & (i+1, j+1) & & \\ & (i-1, j) & & & (i+1, j) \\ (i-1, j-1) & (i, j-1) & (i+1, j-1) & & \end{array}$$

reachable by a king's move from (i, j) are unoccupied grid locations. This means that they are not off the grid and not occupied by another crate, so that the robot can move to any of these eight positions.

Your job is to devise an algorithm for estimating m to within ϵ relative error with probability at least $1 - \delta$, based on the noises emitted by the robot. The input to your algorithm is the sequence of bits x_1, x_2, x_3, \dots , where x_i is 1 if the robot makes a noise on its i -th step. Your algorithm should run in time polynomial in n , $1/\epsilon$, and $\log(1/\delta)$.

Solution

It turns out that ϵ is a bit of a red herring: we can in fact compute the *exact* number of crates with probability $1 - \delta$ in time polynomial in n and $\log(1/\delta)$.

The placement restrictions and laziness make this an irreducible aperiodic chain, so it has a unique stationary distribution π . It is easy to argue from reversibility that this is uniform, so each of the $N = n^2 - m$ unoccupied positions occurs with probability exactly $1/N$.

It will be useful to observe that we can assign three unique unoccupied positions to each crate, to the east, south, and southeast, and this implies $m \leq n^2/4$.

The idea now is to run the robot until the distribution on its position is close to π , and then see if it hits an obstacle on the next step. We can easily count the number of possible transitions that hit an obstacle, since there are $4m$ incoming edges to the crates, plus $4n$ incoming edges to the walls. Since each edge uv has probability $\pi_u p_{uv} = \frac{1}{8N}$ of being selected in the stationary distribution, the probability q that we hit an obstacle starting from π is exactly $\frac{n+m}{2N} = \frac{n+m}{n^2-m}$. This function is not trivial to invert, but we don't have to invert it: if we can compute its value (to some reasonable precision),

we can compare it to all $n^2/4 + 1$ possible values of m , and see which it matches. But we still have to deal with both getting enough samples and the difference between our actual distribution when we take a sample and the stationary distribution. For the latter, we need a convergence bound.

We will get this bound by constructing a family of canonical paths between each distinct pair of locations (i, j) and (i', j') . In the absence of crates, we can use the L-shaped paths that first change i to i' and then j to j' , one step at a time in the obvious way. If we encounter a crate at some position along this path, we replace that position with a detour that uses up to three locations off the original path to get around the obstacle. The general rule is that if we are traveling left-to-right or right-to-left, we shift up one row to move past the obstacle, and if we are moving up-to-down or down-to-up, we shift left one column. An annoying special case is when the obstacle appears exactly at the corner (i', j) ; here we replace the blocked position with the unique diagonally adjacent position that still gives us a path.

Consider the number of paths $(i, j) \rightarrow (i', j')$ crossing a particular left-right edge $(x, y) \rightarrow (x + 1, y)$. There are two cases we have to consider:

1. We cross the edge as part of the left-to-right portion of the path (this includes left-to-right moves that are part of detours). In this case we have $|j - y| \leq 1$. This gives at most 3 choices for j , giving at most $3n^3$ possible paths. (The constants can be improved here.)
2. We cross the edge as part of a detour on the vertical portion of the path. Then $|i - x| \leq 1$, and so we again have at most $3n^3$ possible paths.

This gives at most $6n^3$ possible paths across each edge, giving a congestion

$$\begin{aligned} \rho &\leq \frac{1}{\pi_{ij} p_{ij, i'j'}} (6n^3) \pi_{ij} \pi_{i'j'} \\ &= \frac{1}{N^{-1}(1/8)} (6n^3) N^{-2} \\ &= 24n^3 N^{-1} \\ &\leq 24n^3 \left(\frac{3}{4}n^2\right)^{-1} \\ &= 32n, \end{aligned}$$

since we can argue from the placement restrictions that $N \leq n^2/4$. This immediately gives a bound $\tau_2 \leq 8\rho^2 \leq O(n^2)$. using Lemma 9.5.3.

So let's run for $\lceil 51\tau_2 \ln n \rceil = O(n^2 \ln n)$ steps for each sample. Starting from any initial location, we will reach some distribution σ with $d_{TV}\sigma, \pi = O(n^{-50})$. Let X be the number of obstacles (walls or crates) adjacent to the current position, then we can apply Lemma 9.2.2 to get $|\mathbb{E}_\sigma(X) - \mathbb{E}_\pi(X)| \leq 4d_{TV}(\sigma, \pi) = O(n^{-50})$. The same bound (up to constants) also applies to the probability $\rho = X/8$ of hitting an obstacle, giving $\rho = 4(n+m)/(n^2-m) \pm O(n^{-50})$. Note that ρ is $\Theta(n^{-1})$ for all values of m .

Now take $n^{10} \sqrt{\ln(1/\delta)}$ samples, with a gap of $\lceil 51\tau_2 \ln n \rceil$ steps between each. The expected number of positive samples is $\mu = (\rho + O(n^{-50}))n^{10} \sqrt{\ln(1/\delta)} = \Theta(n^9 \sqrt{\ln(1/\delta)})$, and from Lemma 5.2.2, the probability that the number of positive samples exceeds $\mu(1+n^{-4})$ is at most $\exp(-\mu n^{-8}/3) = \exp(-\Theta(n \ln(1/\delta))) = \delta^{\Theta(n)} \leq \delta/2$ for sufficiently large n . A similar bound holds on the other side. So with probability at least $1-\delta$ we get an estimate $\hat{\rho}$ of $\rho = 4(n+m)/(n^2-m)$ that is accurate to within a relative error of n^{-4} . Since $\frac{d\rho}{dm} = O(n^{-2})$ throughout the interval spanned by m , and since ρ itself is $\Theta(n^{-1})$, any relative error that is $o(n^{-1})$ will give us an exact answer for m when we round to the nearest value of ρ . We are done.

We've chosen big exponents because all we care about is getting a polynomial bound. With a cleaner analysis we could probably get better exponents. It is also worth observing that the problem is ambiguous about whether we have random access to the data stream, and whether the time taken by the robot to move around counts toward our time bound. If we are processing the stream after the fact, and are allowed to skip to specific places in the stream at no cost, we can get each sample in $O(1)$ time. This may further reduce the exponents, and demonstrates why it is important to nail down all the details of a model.

C.6.2 Return of the sprites

The Space Invaders from Problem C.3.1 are back, and now they are trying to hide in random $n \times n$ bitmaps, where each pixel is set with probability $1/2$. Figure C.2 shows an example of this, with two sprites embedded in an otherwise random bitmap.

The sprites would like your assistance in making their hiding places convincing. Obviously it's impossible to disguise their presence completely, but to make the camouflage as realistic as possible, they would like you to provide them with an algorithm that will generate an $n \times n$ bitmap uniformly at random, conditioned on containing *at least two* copies of a given m -pixel sprite at distinct offsets. (It is OK for the sprites to overlap, but they cannot be directly on top of each other, and sprites that extend beyond the edges of



Figure C.2: Two hidden Space Invaders. On the left, the Space Invaders hide behind random pixels. On the right, their positions are revealed by turning the other pixels gray.

the bitmap wrap around as in Problem C.3.1.) Your algorithm should make the probability of each bitmap that contains at least two copies of the sprite be exactly equally likely, and should run in expected time polynomial in n and m .

Solution

Rejection sampling doesn't work here because if the sprites are large, the chances of getting two sprites out of a random bitmap are exponentially small. Generating a random bitmap and slapping two sprites on top of it also doesn't work, because it gives a non-uniform distribution: if the sprites overlap in k places, there are 2^{n^2-2m+k} choices for the remaining bits, which means that we would have to adjust the probabilities to account for the effect of the overlap. But even if we do this, we still have issues with bitmaps that contain more than two sprites: a bitmap with three sprites can be generated in three different ways, and it gets worse quickly as we generate more. It may be possible to work around these issues, but a simpler approach is to use the sampling mechanism from Karp-Luby [KL85] (see also §10.3).

Order all $\binom{n^2}{2}$ positions $u < v$ for the two planted sprites in lexicographic order. For each pair of positions $u < v$, let A_{uv} be the set of all bitmaps with sprites at u and v . Then we can easily calculate $|A_{uv}| = 2^{n^2-2m+k}$ where k is the number of positions where sprites at u and v overlap, and so we can sample a particular A_{uv} with probability $|A_{uv}| / \sum_{st} |A_{st}|$ and then choose an

element of A_{uv} uniformly at random by filling in the positions not covered by the sprites with independent random bits. To avoid overcounting, we discard any bitmap that contains a sprite at a position less than $\max(u, v)$; this corresponds to discarding any bitmap that appears in A_{st} for some $st < uv$. By the same argument as in Karp-Luby, we expect to discard at most $\binom{n^2}{2}$ bitmaps before we get one that works. This gives an expected $O(n^4)$ attempts, each of which requires about $O(n^2)$ work to generate a bitmap and $O(n^2m)$ work to test it, for a nicely polynomial $O(n^6m)$ expected time in total.

If we don't want to go through all this analysis, we could also reduce to Karp-Luby directly by encoding the existence of two sprites as a large DNF formula consisting of $\binom{n^2}{2}$ clauses, and then argue that the sampling mechanism inside Karp-Luby gives us what we want.

C.7 Final exam

Write your answers in the blue book(s). Justify your answers. Work alone. Do not use any notes or books.

There are three problems on this exam, each worth 20 points, for a total of 60 points. You have approximately three hours to complete this exam.

C.7.1 Virus eradication (20 points)

Viruses have infected a network structured as an undirected graph $G = (V, E)$. At time 0, all $n = |V|$ nodes are infected. At each step, as long as there is at least one infected node, we can choose a node v to disinfect; this causes v to become uninfected, but there is an independent $\frac{1}{2d(v)}$ chance that each of v 's neighbors becomes infected if it is not infected already, where $d(v)$ is the degree of v .

Let X_t be the number of nodes that are infected after t disinfection steps. Let τ be the first time at which $X_\tau = 0$. Show that, no matter what strategy is used to select nodes to disinfect, $E[\tau] = O(n)$.

Clarification added during exam: When choosing a node to disinfect, the algorithm can only choose nodes that are currently infected.

Solution

Let $Y_t = X_t + \frac{1}{2} \min(t, \tau)$. We will show that $\{Y_t\}$ is a supermartingale. Suppose we disinfect node v at time t , and that v has $d' \leq d(v)$ uninfected neighbors. Then $E[X_{t+1} | X_t] = X_t - 1 + \frac{d'}{2d(v)} \leq X_t - \frac{1}{2}$, because we always

disinfect v and each of its d' uninfected neighbors contributes $\frac{1}{2d(v)}$ new infections on average. We also have $\min(\tau, t)$ increasing by 1 in this case. This gives $E[Y_{t+1} | Y_t] \leq Y_t - \frac{1}{2} + \frac{1}{2} = Y_t$ when $t < \tau$; it also holds trivially that $Y_{t+1} = Y_t$ when $t \geq \tau$. So $E[Y_{t+1} | Y_t] \leq Y_t$ always.

Now apply the optional stopping theorem. We have bounded increments, because $|Y_{t+1} - Y_t| \leq n + \frac{1}{2}$. We also have bounded expected time, because there is a nonzero probability ϵ that each sequence of n consecutive disinfections produces no new infections, giving $E[\tau] \leq 1/\epsilon$. So $n = Y_0 \geq E[Y_\tau] = \frac{1}{2} E[\tau]$, giving $E[\tau] \leq 2n = O(n)$.

C.7.2 Parallel bubblesort (20 points)

Suppose we are given a random bit-vector and choose to sort it using parallel bubblesort. The initial state consists of n independent fair random bits $X_1^0, X_2^0, \dots, X_n^0$, and in each round, for each pair of positions X_i^t and X_{i+1}^t , we set new values $X_i^{t+1} = 0$ and $X_{i+1}^{t+1} = 1$ if $X_i^t = 1$ and $X_{i+1}^t = 0$. When it holds that $X_i^t = 1$ and $X_{i+1}^t = 0$, we say that there is a swap at position i in round t . Denote the total number of swaps in round t by Y_t .

1. What is $E[Y_0]$?
2. What is $E[Y_1]$?
3. Show that both Y_0 and Y_1 are concentrated around their expectations, by showing, for any fixed $c > 0$ and each $t \in \{0, 1\}$, that there exists a function $f_t(n) = o(n)$ such that $\Pr[|Y_t - E[Y_t]| \geq f_t(n)] \leq n^{-c}$.

Solution

1. This is a straightforward application of linearity of expectation. Let Z_i^t , for each $i \in \{1, \dots, n-1\}$, be the indicator for the event that $X_i^t = 1$ and $X_{i+1}^t = 0$. For $t = 0$, X_i^0 and X_{i+1}^0 are independent, so this event occurs with probability $\frac{1}{4}$. So $E[Y_0] = E\left[\sum_{i=1}^{n-1} Z_i^0\right] = \sum_{i=1}^{n-1} E[Z_i^0] = (n-1)\frac{1}{4} = \frac{n-1}{4}$.
2. Again we want to use linearity of expectation, but we have to do a little more work to calculate $E[Z_i^1]$. For Z_i^1 to be 1, we need $X_i^1 = 1$ and $X_{i+1}^1 = 0$. If we ask where that 1 in X_i^1 came from, either it started in X_{i-1}^0 and moved to position i because $X_i^0 = 0$, or it started in X_i^0 and didn't move because $X_{i+1}^0 = 1$. Similarly the 0 in X_{i+1}^1 either started in X_{i+2}^0 and moved down to replace $X_{i+1}^0 = 1$ or started in X_{i+1}^0 and

didn't move because $X_i^0 = 0$. Enumerating all the possible assignments of X_{i-1}^0 , X_i^0 , X_{i+1}^0 , and X_{i+2}^0 consistent with these conditions gives 0110, 1000, 1001, 1010, and 1110, making $E[Z_i^1] = 5 \cdot 2^{-4} = \frac{5}{16}$ when $2 \leq i \leq n-2$. This accounts for $n-3$ of the positions.

There are two annoying special cases: when $i = 1$ and when $i = n-1$. We can handle these with the above analysis by pretending that $X_0^t = 0$ and $X_{n+1}^t = 1$ for all t ; this leaves the initial patterns 0110 for $i = 1$ and 1001 for $i = n-1$, each of which occurs with probability $\frac{1}{8}$.

Summing over all cases then gives $E[Y_1] = \frac{1}{8} + \sum_{i=2}^{n-3} \frac{5}{16} + \frac{1}{8} = \frac{5n-11}{16}$.

3. The last part is just McDiarmid's inequality. Because the algorithm is deterministic aside from the initial random choice of input, We can express each Y_t as a function $g_t(X_1^0, \dots, X_n^0)$ of the independent random variables X_1^0, \dots, X_n^0 . Changing some X_i^0 can change Z_{i-1}^0 and Z_i^0 , for a total change to Y_0 of at most 2; similarly, changing X_i^0 can change at most $Z_{i-2}^1, Z_{i-1}^1, Z_i^1$, and Z_{i+1}^1 , for a total change of at most 4. So McDiarmid says that, for each $t \in \{0, 1\}$, $\Pr[|Y_t - E[Y_t]| \leq s] \leq 2 \exp -\Theta(s^2/n)$. For any fixed $c > 0$, there is some $s = \Theta(\sqrt{n \log n}) = o(n)$ such that the right-hand side is less than n^{-c} .

C.7.3 Rolling a die (20 points)

Suppose we have a device that generates a sequence of independent fair random coin-flips, but what we want is a six-sided die that generates the values 1, 2, 3, 4, 5, 6 with equal probability.

1. Give an algorithm that does so using $O(1)$ coin-flips on average.
2. Show that any correct algorithm for this problem will use more than n coin-flips with probability at least $2^{-O(n)}$.

Solution

1. Use rejection sampling: generate 3 bits, and if the resulting binary number is not in the range $1, \dots, 6$, try again. Each attempt consumes 3 bits and succeeds with probability $3/4$, so we need to generate $\frac{4}{3} \cdot 3 = 4$ bits on average.

It is possible to improve on this by reusing the last bit of a discarded triple of bits as the first bit in the next triple. This requires a more complicated argument to show uniformity, but requires only two bits

for each attempt after the first, for a total of $3 + \frac{1}{4} \cdot \frac{4}{3} \cdot 2 = \frac{11}{3}$ bits on average. This is still $O(1)$, so unless that $\frac{1}{3}$ bit improvement is really important, it's probably easiest just to do rejection sampling.

2. Suppose that we have generated n bits so far. Since 6 does not evenly divide 2^n for any n , we cannot assign an output from $1, \dots, 6$ to all possible 2^n sequences of bits without giving two outputs different probabilities. So we must keep going in at least one case, giving a probability of at least $2^{-n} = 2^{-O(n)}$ that we continue.

Appendix D

Sample assignments from Spring 2014

Assignments are typically due Wednesdays at 17:00. Assignments should be submitted in PDF format via the classesv2 Drop Box.

D.1 Assignment 1: due Wednesday, 2014-09-10, at 17:00

D.1.1 Bureaucratic part

Send me email! My address is james.aspnes@gmail.com.

In your message, include:

1. Your name.
2. Your status: whether you are an undergraduate, grad student, auditor, etc.
3. Anything else you'd like to say.

(You will not be graded on the bureaucratic part, but you should do it anyway.)

D.1.2 Two terrible data structures

Consider the following data structures for maintaining a sorted list:

1. A sorted array A . Inserting a new element at position $A[i]$ into an array that currently contains k elements requires moving the previous values in $A[i] \dots A[k]$ to $A[i+1] \dots A[k+1]$. Each element moved costs one unit.

For example, if the array currently contains 1 3 5, and we insert 2, the resulting array will contain 1 2 3 5 and the cost of the operation will be 2, because we had to move 3 and 5.

2. A sorted doubly-linked list. Inserting a new element at a new position requires moving the head pointer from its previous position to a neighbor of the new node, and then to the new node; each pointer move costs one unit.

For example, if the linked list currently contains 1 3 5, with the head pointer pointing to 5, and we insert 2, the resulting linked list will contain 1 2 3 5, with the head pointing to 2, and the cost will be 2, because we had to move the pointer from 5 to 3 and then from 3 to 2. Note that we do not charge for updating the pointers between the new element and its neighbors. We will also assume that inserting the first element is free.

Suppose that we insert the elements 1 through n , in random order, into both data structures. Give an exact closed-form expression for the expected cost of each sequence of insertions.

Solution

1. Suppose we have already inserted k elements. Then the next element is equally likely to land in any of the positions $A[1]$ through $A[k+1]$. The number of displaced elements is then uniformly distributed in 0 through k , giving an expected cost for this insertion of $\frac{k}{2}$.

Summing over all insertions gives

$$\begin{aligned} \sum_{k=0}^{n-1} \frac{k}{2} &= \frac{1}{2} \sum_{k=0}^{n-1} k \\ &= \frac{n(n-1)}{4}. \end{aligned}$$

An alternative proof, which also uses linearity of expectation, is to define X_{ij} as the indicator variable for the event that element j moves when element $i < j$ is inserted. This is 1 if and only if j is inserted

before i , which by symmetry occurs with probability exactly $1/2$. So the expected total number of moves is

$$\begin{aligned} \sum_{1 \leq i < j \leq n} \mathbb{E}[X_{ij}] &= \sum_{1 \leq i < j \leq n} \frac{1}{2} \\ &= \frac{1}{2} \binom{n}{2} \\ &= \frac{n(n-1)}{4}. \end{aligned}$$

It's always nice when we get the same answer in situations like this.

2. Now we need to count how far the pointer moves between any two consecutive elements. Suppose that we have already inserted $k-1 > 0$ elements, and let X_k be the cost of inserting the k -th element. Let i and j be the indices in the sorted list of the new and old pointer positions after the k -th insertion. By symmetry, all pairs of distinct

positions $i \neq j$ are equally likely. So we have

$$\begin{aligned}
\mathbb{E}[X_k] &= \frac{1}{k(k-1)} \sum_{i \neq j} |i - j| \\
&= \frac{1}{k(k-1)} \left(\sum_{1 \leq i < j \leq k} (j - i) + \sum_{1 \leq j < i \leq k} (i - j) \right) \\
&= \frac{2}{k(k-1)} \sum_{1 \leq i < j \leq k} (j - i) \\
&= \frac{2}{k(k-1)} \sum_{j=1}^k \sum_{\ell=1}^{j-1} \ell \\
&= \frac{2}{k(k-1)} \sum_{j=1}^k \frac{j(j-1)}{2} \\
&= \frac{1}{k(k-1)} \sum_{j=1}^k j(j-1). \\
&= \frac{1}{k(k-1)} \sum_{j=1}^k (j^2 - j) \\
&= \frac{1}{k(k-1)} \cdot \left(\frac{(2k+1)k(k+1)}{6} - \frac{k}{k+1} 2 \right) \\
&= \frac{1}{k(k-1)} \cdot \frac{(2k-2)k(k+1)}{6} \\
&= \frac{k+1}{3}.
\end{aligned}$$

This is such a simple result that we might reasonably expect that there is a faster way to get it, and we'd be right. A standard trick is to observe that we can simulate choosing k points uniformly at random from a line of n points by instead choosing $k+1$ points uniformly at random from a cycle of $n+1$ points, and deleting the first point chosen to turn the cycle back into a line. In the cycle, symmetry implies that the expected distance between each point and its successor is the same as for any other point; there are $k+1$ such distances, and they add up to $n+1$, so each expected distance is exactly $\frac{n+1}{k+1}$.

In our particular case, n (in the formula) is k and k (in the formula) is 2, so we get $\frac{k+1}{3}$. Note we are sweeping the whole absolute value thing under the carpet here, so maybe the more explicit derivation is safer.

However we arrive at $E[X_k] = \frac{k+1}{3}$ (for $k > 1$), we can sum these expectations to get our total expected cost:

$$\begin{aligned} E\left[\sum_{k=2}^n X_k\right] &= \sum_{k=2}^n E[X_k] \\ &= \sum_{k=2}^n \frac{k+1}{3} \\ &= \frac{1}{3} \sum_{\ell=3}^{n+1} \ell \\ &= \frac{1}{3} \left(\frac{(n+1)(n+2)}{2} - 3 \right) \\ &= \frac{(n+1)(n+2)}{6} - 1. \end{aligned}$$

It's probably worth checking a few small cases to see that this answer actually makes sense.

For large n , this shows that the doubly-linked list wins, but not by much: we get roughly $n^2/6$ instead of $n^2/4$. This is a small enough difference that in practice it is probably dominated by other constant-factor differences that we have neglected.

D.1.3 Parallel graph coloring

Consider the following algorithm for assigning one of k colors to each node in a graph with m edges:

1. Assign each vertex u a color c_u , chosen uniformly at random from all k possible colors.
2. For each vertex u , if u has a neighbor v with $c_u = c_v$, assign u a new color c'_u , again chosen uniformly at random from all k possible colors. Otherwise, let $c'_u = c_u$.

Note that any new colors c' do not affect the test $c_u = c_v$. A node changes its color only if it has the same original color as the original color of one or more of its neighbors.

Suppose that we run this algorithm on an r -regular¹ triangle-free² graph. As a function of k , r , and m , give an exact closed-form expression for the

¹Every vertex has exactly r neighbors.

²There are no vertices u , v , and w such that all are neighbors of each other.

expected number of monochromatic³ edges after running both steps of the algorithm.

Solution

We can use linearity of expectation to compute the probability that any particular edge is monochromatic, and then multiply by m to get the total.

Fix some edge uv . If either of u or v is recolored in step 2, then the probability that $c'_u = c'_v$ is exactly $1/k$. If neither is recolored, the probability that $c'_u = c'_v$ is zero (otherwise $c_u = c_v$, forcing both to be recolored). So we can calculate the probability that $c'_u = c'_v$ by conditioning on the event A that neither vertex is recolored.

This event occurs if both u and v have no neighbors with the same color. The probability that $c_u = c_v$ is $1/k$. The probability that any particular neighbor w of u has $c_w = c_u$ is also $1/k$; similarly for any neighbor w of v . These events are all independent on the assumption that the graph is triangle-free (which implies that no neighbor of u is also a neighbor of v). So the probability that none of these $2r - 1$ events occur is $(1 - 1/k)^{2r-1}$.

We then have

$$\begin{aligned} \Pr [c_u = c_v] &= \Pr [c_u = c_v \mid \bar{A}] \Pr [\bar{A}] + \Pr [c_u = c_v \mid A] \Pr [A] \\ &= \frac{1}{k} \cdot \left(1 - \left(1 - \frac{1}{k} \right)^{2r-1} \right). \end{aligned}$$

Multiply by m to get

$$\frac{m}{k} \cdot \left(1 - \left(1 - \frac{1}{k} \right)^{2r-1} \right).$$

For large k , this is approximately $\frac{m}{k} \cdot \left(1 - e^{-(2r-1)/k} \right)$, which is a little bit better than than the $\frac{m}{k}$ expected monochrome edges from just running step 1.

Repeated application of step 2 may give better results, particular if k is large relative to r . We will see this technique applied to a more general class of problems in §11.3.5.

³Both endpoints have the same color.

D.2 Assignment 2: due Wednesday, 2014-09-24, at 17:00

D.2.1 Load balancing

Suppose we distribute n processes independently and uniformly at random among m machines, and pay a communication cost of 1 for each pair of processes assigned to different machines. Let C be the total communication cost, that is, the number of pairs of processes assigned to different machines. What is the expectation and variance of C ?

Solution

Let C_{ij} be the communication cost between machines i and j . This is just an indicator variable for the event that i and j are assigned to different machines, which occurs with probability $1 - \frac{1}{m}$. We have $C = \sum_{1 \leq i < j \leq n} C_{ij}$.

1. Expectation is a straightforward application of linearity of expectation. There are $\binom{n}{2}$ pairs of processes, and $E[C_{ij}] = 1 - \frac{1}{m}$ for each pair, so

$$E[C] = \binom{n}{2} \left(1 - \frac{1}{m}\right).$$

2. Variance is a little trickier because the C_{ij} are not independent. But they are pairwise independent: even if we fix the location of i and j , the expectation of C_{jk} is still $1 - \frac{1}{m}$, so $\text{Cov}[C_{ij}, C_{jk}] = E[C_{ij}C_{jk}] - E[C_{ij}] \cdot E[C_{jk}] = 0$. So we can compute

$$\text{Var}[C] = \sum_{1 \leq i < j < n} \text{Var}[C_{ij}] = \binom{n}{2} \frac{1}{m} \left(1 - \frac{1}{m}\right).$$

D.2.2 A missing hash function

A clever programmer inserts X_i elements in each of m buckets in a hash table, where each bucket i is implemented as a balanced binary search tree with search cost at most $\lceil \lg(X_i + 1) \rceil$. We are interested in finding a particular target element x , which is equally likely to be in any of the buckets, but we don't know what the hash function is.

Suppose that we know that the X_i are independent and identically distributed with $E[X_i] = k$, and that the location of x is independent of the values of the X_i . What is best upper bound we can put on the expected cost of finding x ?

Solution

The expected cost of searching bucket i is $E[\lceil \lg(X_i + 1) \rceil]$. This is the expectation of a function of X_i , so we would like to bound it using Jensen's inequality (§4.3).

Unfortunately the function $f(n) = \lceil \lg(n + 1) \rceil$ is not concave (because of the ceiling), but $1 + \lg(n + 1) > \lceil \lg(n + 1) \rceil$ is. So the expected cost of searching bucket i is bounded by $1 + \lg(E[X_i] + 1) = 1 + \lg(k + 1)$.

Assuming we search the buckets in some fixed order until we find x , we will search Y buckets where $E[Y] = \frac{n+1}{2}$. Because Y is determined by the position of x , which is independent of the X_i , Y is also independent of the X_i . So Wald's equation (3.4.3) applies, and the total cost is bounded by

$$\frac{n+1}{2} (1 + \lg(k + 1)).$$

D.3 Assignment 3: due Wednesday, 2014-10-08, at 17:00

D.3.1 Tree contraction

Suppose that you have a tree data structure with n nodes, in which each node i has a pointer $\text{parent}(u)$ to its parent (or itself, in the case of the root node root).

Consider the following randomized algorithm for shrinking paths in the tree: in the first phase, each node u first determines its parent $v = \text{parent}(u)$ and its grandparent $w = \text{parent}(\text{parent}(u))$. In the second phase, it sets $\text{parent}'(u)$ to v or w according to a fair coin-flip independent of the coin-flips of all the other nodes.

Let T be the tree generated by the parent pointers and T' the tree generated by the parent' pointers. An example of two such trees is given in Figure D.1.

Recall that the **depth** of a node is defined by $\text{depth}(\text{root}) = 0$ and $\text{depth}(u) = 1 + \text{depth}(\text{parent}(u))$ when $u \neq \text{root}$. The depth of a tree is equal to the maximum depth of any node.

Let D be depth of T , and D' be the depth of T' . Show that there is a constant a , such that for any fixed $c > 0$, with probability at least $1 - n^{-c}$ it holds that

$$a \cdot D - O(\sqrt{D \log n}) \leq D' \leq a \cdot D + O(\sqrt{D \log n}). \quad (\text{D.3.1})$$

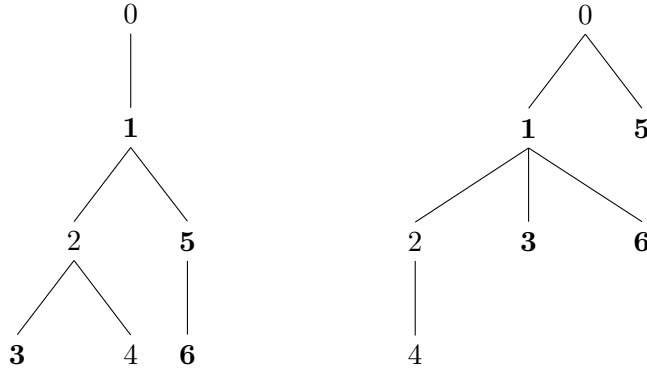


Figure D.1: Example of tree contraction for Problem D.3.1. Tree T is on the left, T' on the right. Nodes 1, 3, 5, and 6 (in boldface) switch to their grandparents. The other nodes retain their original parents.

Solution

For any node u , let $\text{depth}(u)$ be the depth of u in T and $\text{depth}'(u)$ be the depth of u in T' . Note that $\text{depth}'(u)$ is a random variable. We will start by computing $E[\text{depth}'(u)]$ as a function of $\text{depth}(u)$, by solving an appropriate recurrence.

Let $S(k) = \text{depth}'(u)$ when $\text{depth}(u) = k$. The base cases are $S(0) = 0$ (the depth of the root never changes) and $S(1) = 1$ (same for the root's children). For larger k , we have

$$E[\text{depth}'(u)] = \frac{1}{2} E[1 + \text{depth}'(\text{parent}(u))] + \frac{1}{2} E[1 + \text{depth}'(\text{parent}(\text{parent}(u)))]$$

or

$$S(k) = 1 + \frac{1}{2}S(k - 1) + \frac{1}{2}S(k - 2).$$

There are various ways to solve this recurrence. The most direct may be to define a generating function $F(z) = \sum_{k=0}^{\infty} S(k)z^k$. Then the recurrence becomes

$$F = \frac{z}{1 - z} + \frac{1}{2}zF + \frac{1}{2}z^2F.$$

Solving for F gives

$$\begin{aligned}
 F &= \frac{\frac{z}{1-z}}{1 - \frac{1}{2}z - \frac{1}{2}z^2} \\
 &= \frac{2z}{(1-z)(2-z-z^2)} \\
 &= \frac{2z}{(1-z)^2(2+z)} \\
 &= 2z \left(\frac{1/3}{(1-z)^2} + \frac{1/9}{1-z} + \frac{1/18}{1+\frac{1}{2}z} \right) \\
 &= \frac{2}{3} \cdot \frac{z}{(1-z)^2} + \frac{2}{9} \cdot \frac{z}{1-z} + \frac{1}{9} \cdot \frac{z}{1+\frac{1}{2}z},
 \end{aligned}$$

from which we can read off the exact solution

$$S(k) = \frac{2}{3} \cdot k + \frac{2}{9} + \frac{1}{9} \left(-\frac{1}{2} \right)^{k-1}$$

when $k \geq 1$.⁴

We can easily show that $\text{depth}'(u)$ is tightly concentrated around $\text{depth}(u)$ using McDiarmid's inequality (5.3.12). Let X_i , for $i = 2 \dots \text{depth } u$, be the choice made by u 's depth- i ancestor. Then changing one X_i changes $\text{depth}'(u)$ by at most 1. So we get

$$\Pr [\text{depth}'(u) \geq \mathbb{E} [\text{depth}'(u) + t]] \leq e^{-2t^2/(\text{depth}(u)-1)} \quad (\text{D.3.2})$$

and similarly

$$\Pr [\text{depth}'(u) \leq \mathbb{E} [\text{depth}'(u) - t]] \leq e^{-2t^2/(\text{depth}(u)-1)}. \quad (\text{D.3.3})$$

Let $t = \sqrt{\frac{1}{2}D \ln(1/\epsilon)}$. Then the right-hand side of (D.3.2) and (D.3.3) becomes $e^{-D \ln(1/\epsilon)/(\text{depth}(u)-1)} < e^{-\ln(1/\epsilon)} = \epsilon$. For $\epsilon = \frac{1}{2}n^{-c-1}$, we get $t = \sqrt{\frac{1}{2}D \ln(2n^{c+1})} = \sqrt{\frac{c+1}{2}D(\ln n + \ln 2)} = O(\sqrt{D \log n})$ when c is constant.

For the lower bound on D' , when can apply (D.3.3) to some single node u with $\text{depth}(u) = D$; this node by itself will give $D' \geq \frac{2}{3}D - O(\sqrt{D \log n})$ with

⁴A less direct but still effective approach is to guess that $S(k)$ grows linearly, and find a and b such that $S(k) \leq ak + b$. For this we need $ak + b \leq 1 + \frac{1}{2}(a(k-1) + b) + \frac{1}{2}(a(k-2) + b)$. The b 's cancel, leaving $ak \leq 1 + ak - \frac{3}{2}a$. Now the ak 's cancel, leaving us with $0 \leq 1 - \frac{3}{2}a$ or $a \geq 2/3$. We then go back and make $b = 1/3$ to get the right bound on $S(1)$, giving the bound $S(k) \leq \frac{2}{3} \cdot k + \frac{1}{3}$. We can then repeat the argument for $S(k) \geq a'k + b'$ to get a full bound $\frac{2}{3}k \leq S(k) \leq \frac{2}{3}k + \frac{1}{3}$.

probability at least $1 - \frac{1}{2}n^{-c-1}$. For the upper bound, we need to take the maximum over all nodes. In general, an upper bound on the maximum of a bunch of random variables is likely to be larger than an upper bound on any one of the random variables individually, because there is a lot of room for one of the variables to get unlucky, but we can apply the union bound to get around this. For each individual u , we have $\Pr \left[\text{depth}'(u) \geq \frac{2}{3}D + O(\sqrt{D \log n}) \right] \leq \frac{1}{2}n^{-c-1}$, so $\Pr \left[D' \geq \frac{2}{3}D + O(\sqrt{D \log n}) \right] \leq \sum_u \frac{1}{2}n^{-c-1} = \frac{1}{2}n^{-c}$. This completes the proof.

D.3.2 Part testing

You are running a factory that produces parts for some important piece of machinery. Many of these parts are defective, and must be discarded. There are two levels of tests that can be performed:

- A normal test, which passes a part with probability $2/3$.
- A rigorous test, which passes a part with probability $1/3$.

At each step, the part inspectors apply the following rules to decide which test to apply:

- For the first part, a fair coin-flip decides between the tests.
- For subsequent parts, if the previous part passed, the inspectors become suspicious and apply the rigorous test; if it failed, they relax and apply the normal test.

For example, writing N+ for a part that passes the normal test, N- for one that fails the normal test, R+ for a part that passes the rigorous test, and R- for one that fails the rigorous test, a typical execution of the testing procedure might look like N- N+ R- N- N+ R- N+ R- N- N- N- N+ R+ R- N+ R+. This execution tests 16 parts and passes 7 of them.

Suppose that we test n parts. Let S be the number that pass.

1. Compute $E[S]$.
2. Show that there a constant $c > 0$ such that, for any $t > 0$,

$$\Pr [|S - E[S]| \geq t] \leq 2e^{-ct^2/n}. \quad (\text{D.3.4})$$

Solution

Using McDiarmid's inequality and some cleverness Let X_i be the indicator variable for the event that part i passes, so that $S = \sum_{i=1}^n X_i$.

1. We can show by induction that $E[X_i] = 1/2$ for all i . The base case is X_1 , where $\Pr[\text{part 1 passes}] = \frac{1}{2} \Pr[\text{part 1 passes rigorous test}] + \frac{1}{2} \Pr[\text{part 1 passes normal test}] = \frac{1}{2} \left(\frac{1}{3} + \frac{2}{3} \right) = \frac{1}{2}$. For $i > 1$, $E[X_{i-1}] = 1/2$ implies that part i is tested with the normal and rigorous tests with equal probability, so the analysis for X_1 carries through and gives $E[X_i] = 1/2$ as well. Summing over all X_i gives $E[S] = n/2$.
2. We can't use Chernoff, Hoeffding, or Azuma here, because the X_i are not independent, and do not form a martingale difference sequence even after centralizing them by subtracting off their expectations. So we are left with McDiarmid's inequality unless we want to do something clever and new (we don't). Applying McDiarmid to the X_i directly doesn't work so well, but we can make it work with a different set of variables that generate the same outcomes.

Let $Y_i \in \{A, B, C\}$ be the grade of part i , where A means that it passes both the rigorous and the normal test, B means that fails the rigorous test but passes the normal test, and C means that it fails both tests. In terms of the X_i , $Y_i = A$ means $X_i = 1$, $Y_i = C$ means $X_i = 0$, and $Y_i = B$ means $X_i = 1 - X_{i-1}$ (when $i > 1$). We get the right probabilities for passing each test by assigning equal probabilities.

We can either handle the coin-flip at the beginning by including an extra variable Y_0 , or we can combine the coin-flip with Y_1 by assuming that Y_1 is either A or C with equal probability. The latter approach improves our bound a little bit since then we only have n variables and not $n + 1$.

Now suppose that we fix all Y_j for $j \neq i$ and ask what happens if Y_i changes.

- (a) If $j < i$, then X_j is not affected by Y_i .
- (b) Let $k > i$ be such that $Y_k \in \{A, C\}$. Then X_k is not affected by Y_i , and neither is X_j for any $j > k$.

It follows that changing Y_i can only change $X_i, \dots, X_{i+\ell}$, where ℓ is the number of B grades that follow position i .

There are two cases for the sequence $X_0 \dots X_\ell$:

- (a) If $X_i = 0$, then $X_1 = 1, X_2 = 0$, etc.
- (b) If $X_i = 1$, then $X_1 = 0, X_2 = 1$, etc.

If ℓ is odd, changing Y_i thus has no effect on $\sum_{j=i}^{i+\ell} X_j$, while if ℓ is even, changing Y_i changes the sum by 1. In either case, the effect of changing Y_i is bounded by 1, and McDiarmid's inequality applies with $c_i = 1$, giving

$$\Pr[|S - \mathbb{E}[S]| \geq t] \leq 2e^{-2t^2/n}.$$

D.4 Assignment 4: due Wednesday, 2014-10-29, at 17:00

D.4.1 A doubling strategy

A common strategy for keeping the load factor of a hash table down is to double its size whenever it gets too full. Suppose that we start with a hash table of size 1 and double its size whenever two items in a row hash to the same location.

Effectively, this means that we attempt to insert all elements into a table of size 1; if two consecutive items hash to the same location, we start over and try to do the same to a table of size 2, and in general move to a table of size 2^{k+1} whenever any two consecutive elements hash to the same location in our table of size 2^k .

Assume that we are using an independently chosen 2-universal hash function for each table size. Show that the expected final table size is $O(n)$.

Solution

Let X be the random variable representing the final table size. Our goal is to bound $\mathbb{E}[X]$.

First let's look at the probability that we get at least one collision between consecutive elements when inserting n elements into a table with m locations. Because the pairs of consecutive elements overlap, computing the exact probability that we get a collision is complicated, but we only need an upper bound.

We have $n - 1$ consecutive pairs, and each produces a collision with probability at most $1/m$. This gives a total probability of a collision of at most $(n - 1)/m$.

Let $k = \lceil \lg n \rceil$, so that $2^k \geq n$. Then the probability of a consecutive collision in a table with $2^{k+\ell}$ locations is at most $(n-1)/2^{k+\ell} < 2^k/2^{k+\ell} = 2^{-\ell}$.

Since the events that collisions occur at each table size are independent, we can compute, for $\ell > 0$,

$$\begin{aligned} \Pr [X = 2^{k+\ell}] &\leq \Pr [X \geq 2^{k+\ell}] \\ &\leq \prod_{i=0}^{\ell-1} 2^{-i} \\ &= 2^{-\ell(\ell-1)/2}. \end{aligned}$$

From this it follows that

$$\begin{aligned} \mathbb{E}[X] &= \sum_{i=0}^{\infty} 2^i \Pr [X = 2^i] \\ &< 2^k \Pr [X \leq 2^k] + \sum_{l=1}^{\infty} 2^{k+l} \Pr [X = 2^{k+l}] \\ &\leq 2^k + \sum_{l=1}^{\infty} 2^{k+l} \cdot 2^{-\ell(\ell-1)/2} \\ &\leq 2^k + 2^k \sum_{l=1}^{\infty} 2^{\ell-\ell(\ell-1)/2} \\ &\leq 2^k + 2^k \sum_{l=1}^{\infty} 2^{-(\ell^2-2\ell)/2} \\ &= O(2^k), \end{aligned}$$

since the series converges to some constant that does not depend on k . But we chose k so that $2^k = O(n)$, so this gives us our desired bound.

D.4.2 Hash treaps

Consider the following variant on a treap: instead of choosing the heap keys randomly, we choose a hash function $h : U \rightarrow [1 \dots m]$ from some strongly 2-universal hash family, and use $h(x)$ as the heap key for tree key x . Otherwise the treap operates as usual.⁵

Suppose that $|U| \geq n$. Show that there is a sequence of n distinct tree keys such that the total expected time to insert them into an initially empty hash treap is $\Omega(n^2/m)$.

⁵If m is small, we may get collisions in the heap key values. Assume in this case that a node will stop rising if its parent has the same key.

Solution

Insert the sequence $1 \dots n$.

Let us first argue by induction on i that all elements x with $h(x) = m$ appear as the uppermost elements of the right spine of the treap. Suppose that this holds for $i - 1$. If $h(i) = m$, then after insertion i is rotated up until it has a parent that also has heap key m ; this extends the sequence of elements with heap key m in the spine by 1. Alternatively, if $h(i) < m$, then i is never rotated above an element x with $h(x) = m$, so the sequence of elements with heap key m is unaffected.

Because each new element has a larger tree key than all previous elements, inserting a new element i requires moving past any elements in the right spine, and in particular requires moving past any elements $j < i$ with $h(j) = m$. So the expected cost of inserting i is at least the expected number of such elements j . Because h is chosen from a strongly 2-universal hash family, $\Pr[h(j) = m] = 1/m$ for any j , and by linearity of expectation, $E[|\{j < i \mid h(j) = m\}|] = (i - 1)/m$. Summing this quantity over all i gives a total expected insertion cost of at least $n(n - 1)/2m = \Omega(n^2/m)$.

D.5 Assignment 5: due Wednesday, 2014-11-12, at 17:00

D.5.1 Agreement on a ring

Consider a ring of n processes at positions $0, 1, \dots, n - 1$. Each process i has a value $A[i]$ that is initially 0 or 1. At each step, we choose a process r uniformly at random, and copy $A[r]$ to $A[(r + 1) \bmod n]$. Eventually, all $A[i]$ will have the same value.

Let $A_t[i]$ be the value of $A[i]$ at time t , and let $X_t = \sum_{i=0}^{n-1} A_t[i]$ be the total number of ones in A_t . Let τ be the first time at which $X_\tau \in \{0, n\}$.

1. Suppose that we start with $X_0 = k$. What is the probability that we eventually reach a state that is all ones?
2. What is $E[\tau]$, assuming we start from the worst possible initial state?

Solution

This is a job for the optional stopping theorem. Essentially we are going to follow the same analysis from §8.5.1 for a random walk with two absorbing barriers, applied to X_t .

Let \mathcal{F}_t be the σ -algebra generated by A_0, \dots, A_t . Then $\{\mathcal{F}_t\}$ forms a filtration, and each X_t is \mathcal{F}_t -measurable.

1. For the first part, we show that (X_t, \mathcal{F}_t) is a martingale. The intuition is that however the bits in A_t are arranged, there are always exactly the same number of positions where a zero can be replaced by a one as there are where a one can be replaced by a zero.

Let R_t be the random location chosen in state A_t . Observe that

$$\mathbb{E}[X_{t+1} \mid \mathcal{F}_t, R_t = i] = X_t + A[i] - A[(i + 1) \bmod n].$$

But then

$$\begin{aligned} \mathbb{E}[X_{t+1} \mid \mathcal{F}_t] &= \sum_{i=0}^{n-1} \frac{1}{n} (X_t + A[i] - A[(i + 1) \bmod n]) \\ &= X_t + \frac{1}{n} X_t - \frac{1}{n} X_t \\ &= X_t, \end{aligned}$$

which establishes the martingale property.

We also have that (a) τ is a stopping time with respect to the \mathcal{F}_i ; (b) $\Pr[\tau < \infty] = 1$, because from any state there is a nonzero chance that all $A[i]$ are equal n steps later; and (c) X_t is bounded between 0 and n . So the optional stopping theorem applies, giving

$$\mathbb{E}[X_\tau] = \mathbb{E}[X_0] = k.$$

But then

$$\mathbb{E}[X_\tau] = n \cdot \Pr[X_\tau = n] + 0 \cdot \Pr[X_\tau = 0] = k,$$

so $\mathbb{E}[X_\tau] = k/n$.

2. For the second part, we use a variant on the $X_t^2 - t$ martingale.

Let Y_t count the number of positions i for which $A_t[i] = 1$ and $A_t[(i + 1) \bmod n] = 0$. Then, conditioning on \mathcal{F}_t , we have

$$X_{t+1}^2 = \begin{cases} X_t^2 + 2X_t + 1 & \text{with probability } Y_t/n, \\ X_t^2 - 2X_t + 1 & \text{with probability } Y_t/n, \text{ and} \\ X_t^2 & \text{otherwise.} \end{cases}$$

The conditional expectation sums to

$$\mathbb{E} \left[X_{t+1}^2 \mid \mathcal{F}_t \right] = X_t^2 + 2Y_t/n.$$

Let $Z_t = X_t^2 - 2t/n$ when $t \leq \tau$ and $X_\tau^2 - 2\tau/n$ otherwise. Then, for $t < \tau$, we have

$$\begin{aligned} \mathbb{E} [Z_{t+1} \mid \mathcal{F}_t] &= \mathbb{E} \left[X_{t+1}^2 - 2(t+1)/n \mid \mathcal{F}_t \right] \\ &= X_t^2 + 2Y_t/n - 2(t+1)/n \\ &= X_t^2 - 2t/n + 2Y_t/n - 2/n \\ &= Z_t + 2Y_t/n - 2/n \\ &\leq Z_t. \end{aligned}$$

For $t \geq \tau$, $Z_{t+1} = Z_t$, so in either case Z_t has the submartingale property.

We have previously established that τ has bounded expectation, and it's easy to see that Z_t has bounded step size. So the optional stopping theorem applies to Z_t , and $\mathbb{E} [Z_0] \leq \mathbb{E} [Z_\tau]$.

Let $X_0 = k$. Then $\mathbb{E} [Z_0] = k^2$, and $\mathbb{E} [Z_\tau] = (k/n) \cdot n^2 - 2\mathbb{E} [\tau] / n = kn - 2\mathbb{E} [\tau] / n$. But then

$$k^2 \leq kn - 2\mathbb{E} [\tau] / n$$

which gives

$$\begin{aligned} \mathbb{E} [\tau] &\leq \frac{kn - k^2}{2/n} \\ &= \frac{kn^2 - k^2n}{2}. \end{aligned}$$

This is maximized at $k = \lfloor n/2 \rfloor$, giving

$$\mathbb{E} [\tau] \leq \frac{\lfloor n/2 \rfloor \cdot n^2 - (\lfloor n/2 \rfloor)^2 \cdot n}{2}.$$

For even n , this is just $n^3/4 - n^3/8 = n^3/8$.

For odd n , this is $(n-1)n^2/4 - (n-1)^2n/8 = n^3/8 - n/8$. So there is a slight win for odd n from not being able to start with an exact half-and-half split.

To show that this bound applies in the worst case, observe that if we start with have contiguous regions of k ones and $n - k$ zeros in A_t , then (a) $Y_t = 1$, and (b) the two-region property is preserved in A_{t+1} . In this case, for $t < \tau$ it holds that $E[Z_{t+1} | \mathcal{F}_t] = Z_t + 2Y_t/n - 2/n = Z_t$, so Z_t is a martingale, and thus $E[\tau] = \frac{kn^2 - k^2n}{2}$. This shows that the initial state with $\lfloor n/2 \rfloor$ consecutive zeros and $\lceil n/2 \rceil$ consecutive ones (or vice versa) gives the claimed worst-case time.

D.5.2 Shuffling a two-dimensional array

A programmer working under tight time constraints needs to write a procedure to shuffle the elements of an $n \times n$ array, so that all $(n^2)!$ permutations are equally likely. Some quick Googling suggests that this can be reduced to shuffling a one-dimensional array, for which the programmer's favorite language provides a convenient library routine that runs in time linear in the size of the array. Unfortunately, the programmer doesn't read the next paragraph about converting the 2-d array to a 1-d array first, and instead decides to pick one of the $2n$ rows or columns uniformly at random at each step, and call the 1-d shuffler on this row or column.

Let A^t be the state of the array after t steps, where each step shuffles one row or column, and let B be a random variable over permutations of the original array state that has a uniform distribution. Show that the 2-d shuffling procedure above is asymptotically worse than the direct approach, by showing that there is some $f(n) = \omega(n)$ such that after $f(n)$ steps, $d_{TV}(A^t, B) = 1 - o(1)$.⁶

Solution

Consider the n diagonal elements in positions A_{ii} . For each such element, there is a $1/n$ chance at each step that its row or column is chosen. The

⁶I've been getting some questions about what this means, so here is an attempt to translate it into English.

Recall that $f(n)$ is $\omega(g(n))$ if $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)}$ goes to infinity, and $f(n)$ is $o(g(n))$ if $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)}$ goes to zero.

The problem is asking you to show that there is some $f(n)$ that is more than a constant times n , such that the total variation distance between $A^{f(n)}$ becomes arbitrarily close to 1 for sufficiently large n .

So for example, if you showed that at $t = n^4$, $d_{TV}(A^t, B) \geq 1 - \frac{1}{\log^2 n}$, that would demonstrate the claim, because $\lim_{n \rightarrow \infty} \frac{n^2}{n}$ goes to infinity and $\lim_{n \rightarrow \infty} \frac{1/\log n}{1} = 0$. These functions are, of course, for illustration only. The actual process might or might not converge by time n^4 .)

time until every diagonal node is picked at least once maps to the coupon collector problem, which means that it is $\Omega(n \log n)$ with high probability using standard concentration bounds.

Let C be the event that there is at least one diagonal element that is in its original position. If there is some diagonal node that has not moved, C holds; so with probability $1 - o(1)$, A^t holds at some time t that is $\Theta(n \log n) = \omega(n)$. But by the union bound, C holds in B with probability at most $n \cdot n^{-2} = 1/n$. So the difference between the probability of C in A^t and B is at least $1 - o(1) - 1/n = 1 - o(1)$.

D.6 Assignment 6: due Wednesday, 2014-12-03, at 17:00

D.6.1 Sampling colorings on a cycle

Devise a Las Vegas algorithm for sampling 3-colorings of a cycle.

Given $n > 1$ nodes in a cycle, your algorithm should return a random coloring X of the nodes with the usual constraint that no edge should have the same color on both endpoints. Your algorithm should generate all possible colorings with equal probability, and run in time polynomial in n on average.

Solution

Since it's a Las Vegas algorithm, Markov chain Monte Carlo is not going to help us here. So we can set aside couplings and conductance and just go straight for generating a solution.

First, let's show that we can generate uniform colorings of an n -node line in linear time. Let X_i be the color of node i , where $0 \leq i < n$. Choose X_0 uniformly at random from all three colors; then for each $i > 0$, choose X_i uniformly at random from the two colors not chosen for X_{i-1} . Given a coloring, we can show by induction on i that it can be generated by this process, and because each choice is uniform and each coloring is generated only once, we get all $3 \cdot 2^{n-1}$ colorings of the line with equal probability.

Now we try hooking X_{n-1} to X_0 . If $X_{n-1} = X_0$, then we don't have a cycle coloring, and have to start over. The probability that this event occurs is at most $1/2$, because for every path coloring with $X_{n-1} = X_0$, there is another coloring where we replace X_{n-1} with a color not equal to X_{n-2} or X_0 . So after at most 2 attempts on average we get a good cycle coloring. This gives a total expected cost of $O(n)$.

D.6.2 A hedging problem

Suppose that you own a portfolio of n investment vehicles numbered $1, \dots, n$, where investment i pays out $a_{ij} \in \{-1, +1\}$ at time j , where $0 < j \leq m$. You have carefully chosen these investments so that your total payout $\sum_{i=1}^n a_{ij}$ for any time j is zero, eliminating all risk.

Unfortunately, in doing so you have run afoul of securities regulators, who demand that you sell off half of your holdings—a demand that, fortunately, you anticipated by making n even.

This will leave you with a subset S consisting of $n/2$ of the a_i , and your net worth at time t will now be $w_t = \sum_{j=1}^t \sum_{i \in S} a_{ij}$. If your net worth drops too low at any time t , all your worldly goods will be repossessed, and you will have nothing but your talent for randomized algorithms to fall back on.

Note: an earlier version of this problem demanded a tighter bound.

Show that when n and m are sufficiently large, it is always possible to choose a subset S of size $n/2$ so that $w_t \geq -m\sqrt{n \ln nm}$ for all $0 < t \leq m$, and give an algorithm that finds such a subset in time polynomial in n and m on average.

Solution

Suppose that we flip a coin independently to choose whether to include each investment i . There are two bad things that can happen:

1. We lose too much at some time t from the investments the coin chooses.
2. We don't get exactly $n/2$ heads.

If we can show that the sum of the probabilities of these bad events is less than 1, we get the existence proof we need. If we can show that it is enough less than 1, we also get an algorithm, because we can test in time $O(nm)$ if a particular choice works.

Let X_i be the indicator variable for the event that we include investment i . Then

$$\begin{aligned}
w_t &= \sum_{i=1}^n \left(X_i \sum_{j=1}^t a_{ij} \right) \\
&= \sum_{i=1}^n \left(\left(\frac{1}{2} + \left(X_i - \frac{1}{2} \right) \right) \sum_{j=1}^t a_{ij} \right) \\
&= \frac{1}{2} \sum_{j=1}^t \sum_{i=1}^n a_{ij} + \sum_{i=1}^n \left(X_i - \frac{1}{2} \right) \left(\sum_{j=1}^t a_{ij} \right) \\
&= \sum_{i=1}^n \left(X_i - \frac{1}{2} \right) \left(\sum_{j=1}^t a_{ij} \right).
\end{aligned}$$

Because $\left| X_i - \frac{1}{2} \right|$ is always $\frac{1}{2}$, $\mathbb{E} \left[X_i - \frac{1}{2} \right] = 0$, and each a_{ij} is ± 1 , each term in the outermost sum is a zero-mean random variable that satisfies $\left| \left(X_i - \frac{1}{2} \right) \sum_{j=1}^t a_{ij} \right| \leq \frac{t}{2} \leq \frac{m}{2}$. So Hoeffding's inequality says

$$\begin{aligned}
\Pr \left[w_t - \mathbb{E} [w_t] < m\sqrt{n \ln nm} \right] &\leq e^{-m^2 n \ln nm / (2n(m/2)^2)} \\
&= e^{-2 \ln nm} \\
&= (nm)^{-2}.
\end{aligned}$$

Summing over all t , the probability that this bound is violated for any t is at most $\frac{1}{n^2 m}$.

For the second source of error, we have $\Pr [\sum_{i=1}^n X_i \neq n/2] = 1 - \binom{n}{n/2} / 2^n = 1 - \Theta(1/\sqrt{n})$. So the total probability that the random assignment fails is bounded by $1 - \Theta(1/\sqrt{n}) + 1/n$, giving a probability that it succeeds of at least $\Theta(1/\sqrt{n}) - 1/(n^2 m) = \Theta(\sqrt{n})$. It follows that generating and testing random assignments gives an assignment with the desired characteristics after $\Theta(\sqrt{n})$ trials on average, giving a total expected cost of $\Theta(n^{3/2} m)$.

D.7 Final exam

Write your answers in the blue book(s). Justify your answers. Work alone. Do not use any notes or books.

There are four problems on this exam, each worth 20 points, for a total of 80 points. You have approximately three hours to complete this exam.

D.7.1 Double records (20 points)

Let $A[1 \dots n]$ be an array holding the integers $1 \dots n$ in random order, with all $n!$ permutations of the elements of A equally likely.

Call $A[k]$, where $1 \leq k \leq n$, a **record** if $A[j] < A[k]$ for all $j < k$.

Call $A[k]$, where $2 \leq k \leq n$, a **double record** if both $A[k-1]$ and $A[k]$ are records.

Give an asymptotic (big- Θ) bound on the expected number of double records in A .

Solution

Suppose we condition on a particular set of k elements appearing in positions $A[1]$ through $A[k]$. By symmetry, all $k!$ permutations of these elements are equally likely. Putting the largest two elements in $A[k-1]$ and $A[k]$ leaves $(k-2)!$ choices for the remaining elements, giving a probability of a double record at k of exactly $\frac{(k-2)!}{k!} = \frac{1}{k(k-1)}$.

Applying linearity of expectation gives a total expected number of double records of

$$\begin{aligned} \sum_{k=2}^n \frac{1}{k(k-1)} &\leq \sum_{k=2}^n \frac{1}{(k-1)^2} \\ &\leq \sum_{k=2}^{\infty} \frac{1}{(k-1)^2} \\ &= \sum_{k=1}^{\infty} \frac{1}{k^2} \\ &= \frac{\pi^2}{6} \\ &= O(1). \end{aligned}$$

Since the expected number of double records is at least $1/2 = \Omega(1)$ for $n \geq 2$, this gives a tight asymptotic bound of $\Theta(1)$.

I liked seeing our old friend $\pi^2/6$ so much that I didn't notice an easier

exact bound, which several people supplied in their solutions:

$$\begin{aligned} \sum_{k=2}^n \frac{1}{k(k-1)} &= \sum_{k=2}^n \left(\frac{1}{k-1} - \frac{1}{k} \right) \\ &= \sum_{k=1}^{n-1} \frac{1}{k} - \sum_{k=2}^n \frac{1}{k} \\ &= 1 - \frac{1}{n}. \\ &= O(1). \end{aligned}$$

D.7.2 Hipster graphs (20 points)

Consider the problem of labeling the vertices of a 3-regular graph with labels from $\{0, 1\}$ to maximize the number of *happy* nodes, where a node is *happy* if its label is the opposite of the majority of its three neighbors.

Give a *deterministic* algorithm that takes a 3-regular graph as input and computes, in time polynomial in the size of the graph, a labeling that makes at least half of the nodes happy.

Solution

This problem produced the widest range of solutions, including several very clever deterministic algorithms. Here are some examples.

Using the method of conditional probabilities If we are allowed a randomized algorithm, it's easy to get at exactly half of the nodes on average: simply label each node independently and uniformly at random, observe that each node individually has a $1/2$ chance at happiness, and apply linearity of expectation.

To turn this into a deterministic algorithm, we'll apply the method of conditional expectations. Start with an unlabeled graph. At each step, pick a node and assign it a label that maximizes the expected number of happy nodes conditioned on the labeling so far, and assuming that all later nodes are labeled independently and uniformly at random. We can compute this conditional expectation in linear time by computing the value for each node (there are only 3^4 possible partial labelings of the node and its immediate neighbors, so computing the expected happiness of any particular node can be done in constant time by table lookup; to compute the table, we just enumerate all possible assignments to the unlabeled nodes). So in $O(n^2)$

time we get a labeling in which the number of happy nodes is at least the $n/2$ expected happy nodes we started with.

With a bit of care, the cost can be reduced to linear: because each new labeled node only affects its own probability of happiness and those of its three neighbors, we can update the conditional expectation by just updating the values for those four nodes. This gives $O(1)$ cost per step or $O(n)$ total.

Using hill climbing The following algorithm is an adaptation of the solution of Rose Sloan, and demonstrates that it is in fact possible to make *all* of the nodes happy in linear time.

Start with an arbitrary labeling (say, all 0). At each step, choose an unhappy node and flip its label. This reduces the number of monochromatic edges by at least 1. Because we have only $3n/2$ edges, we can repeat this process at most $3n/2$ times before it terminates. But it only terminates when there are no unhappy nodes.

To implement this in linear time, maintain a queue of all unhappy nodes.

D.7.3 Storage allocation (20 points)

Suppose that you are implementing a stack in an array, and you need to figure out how much space to allocate for the array. At time 0, the size of the stack is $X_0 = 0$. At each subsequent time t , the user flips an independent fair coin to choose whether to push onto the stack ($X_t = X_{t-1} + 1$) or pop from the stack ($X_t = X_{t-1} - 1$). The exception is that when the stack is empty, the user always pushes.

You are told in advance that the stack will only be used for n time units. Your job is to choose a size s for the array so that it will overflow at most half the time: $\Pr[\max_t X_t > s] < 1/2$. As an asymptotic (big- O) function of n , what is the smallest value of s you can choose?

Solution

We need two ideas here. First, we'll show that $X_t - t^2$ is a martingale, despite the fact that X_t by itself isn't. Second, we'll use the idea of stopping X_t when it hits $s + 1$, creating a new martingale $Y_t = X_{t \wedge \tau}^2 - (t \wedge \tau)$ where τ is the first time where $X_t = s$ (and $t \wedge \tau$ is shorthand for $\min(t, \tau)$). We can then apply Markov's inequality to $X_{n \wedge \tau}$.

To save time, we'll skip directly to showing that Y_t is a martingale. There are two cases:

1. If $t < \tau$, then

$$\begin{aligned} \mathbb{E}[Y_{t+1} \mid Y_t, t < \tau] &= \frac{1}{2}((X_t + 1)^2 - (t + 1)) + \frac{1}{2}((X_t - 1)^2 - (t + 1)) \\ &= X_t^2 + 1 - (t + 1) \\ &= X_t^2 - t \\ &= Y_t. \end{aligned}$$

2. If $t \geq \tau$, then $Y_{t+1} = Y_t$, so $\mathbb{E}[Y_{t+1} \mid Y_t, t \geq \tau] = Y_t$.

In either case the martingale property holds.

It follows that $\mathbb{E}[Y_n] = \mathbb{E}[Y_0] = 0$, or $\mathbb{E}[X_{n \wedge \tau}^2 - n] = 0$, giving $\mathbb{E}[X_{n \wedge \tau}^2] = n$. Now apply Markov's inequality:

$$\begin{aligned} \Pr[\max X_t > s] &= \Pr[X_{n \wedge \tau} \geq s + 1] \\ &= \Pr[X_{n \wedge \tau}^2 \geq (s + 1)^2] \\ &\leq \frac{\mathbb{E}[X_{n \wedge \tau}^2]}{(s + 1)^2} \\ &= \frac{n}{(\sqrt{2n} + 1)^2} \\ &< \frac{n}{2n} \\ &= 1/2. \end{aligned}$$

So $s = \sqrt{2n} = O(\sqrt{n})$ is enough.

D.7.4 Fault detectors in a grid (20 points)

A processing plant for rendering discarded final exam problems harmless consists of n^2 processing units arranged in an $n \times n$ grid with coordinates (i, j) each in the range 1 through n . We would like to monitor these processing units to make sure that they are working correctly, and have access to a supply of monitors that will detect failures. Each monitor is placed at some position (i, j) , and will detect failures at any of the five positions (i, j) , $(i - 1, j)$, $(i + 1, j)$, $(i, j - 1)$, and $(i, j + 1)$ that are within the bounds of the grid. This plus-shaped detection range is awkward enough that the engineer designing the system has given up on figuring out how to pack the detectors properly, and instead places a detector at each grid location with independent probability $1/4$.

The engineer reasons that since a typical monitor covers 5 grid locations, using $n^2/4$ monitors on average should cover $(5/4)n^2$ locations, with the extra monitors adding a little bit of safety to deal with bad random choices. So few if any processing units should escape.

1. Compute the exact expected number of processing units that are not within range of any monitor, as a function of n . You may assume $n > 0$.
2. Show that for any fixed c , the actual number of unmonitored processing units is within $O(n\sqrt{\log n})$ of the expected number with probability at least $1 - n^{-c}$.

Solution

1. This part is actually more annoying, because we have to deal with nodes on the edges. There are three classes of nodes:
 - (a) The four corner nodes. To be unmonitored, each corner node needs to have no monitor in its own location or either of the two adjacent locations. This event occurs with probability $(3/4)^3$.
 - (b) The $4(n - 2)$ edge nodes. These have three neighbors in the grid, so they are unmonitored with probability $(3/4)^4$.
 - (c) The $(n - 2)^2$ interior nodes. These are each unmonitored with probability $(3/4)^5$.

Adding these cases up gives a total expected number of unmonitored nodes of

$$4 \cdot (3/4)^3 + 4(n - 2) \cdot (3/4)^4 + (n - 2)^2 \cdot (3/4)^5 = \frac{243}{1024}n^2 - \frac{81}{32}n + \frac{189}{64}. \quad (\text{D.7.1})$$

For $n = 1$, this analysis breaks down; instead, we can calculate directly that the sole node in the grid has a $3/4$ chance of being unmonitored.

Leaving the left-hand side of (D.7.1) in the original form is probably a good idea for understanding how the result works, but the right-hand side demonstrates that this strategy leaves slightly less than a quarter of the processing units uncovered on average.

2. Let X_{ij} be the indicator for a monitor at position (i, j) . Recall that we have assumed that these variables are independent.

Let $S = f(X_{11}, \dots, X_{nn})$ compute the number of uncovered processing units. Then changing a single X_{ij} changes the value of f by at most 5. So we can apply McDiarmid's inequality to get

$$\Pr[|S - \mathbb{E}[S]| \geq t] \leq 2e^{-2t^2/(\sum c_{ij}^2)} = 2e^{-2t^2/(n^2 \cdot 5^2)} = 2e^{-2t^2/(25n^2)}.$$

(The actual bound is slightly better, because we are overestimating c_{ij} for boundary nodes.)

Set this to n^{-c} and solve for $t = n\sqrt{(25/2)(c \ln n + \ln 2)} = O(n\sqrt{\log n})$. Then the bound becomes $2e^{-c \ln n - \ln 2} = n^{-c}$, as desired.

Appendix E

Sample assignments from Spring 2013

E.1 Assignment 1: due Wednesday, 2013-01-30, at 17:00

E.1.1 Bureaucratic part

Send me email! My address is james.aspnes@gmail.com.

In your message, include:

1. Your name.
2. Your status: whether you are an undergraduate, grad student, auditor, etc.
3. Anything else you'd like to say.

(You will not be graded on the bureaucratic part, but you should do it anyway.)

E.1.2 Balls in bins

Throw m balls independently and uniformly at random into n bins labeled $1 \dots n$. What is the expected number of positions $i < n$ such that bin i and bin $i + 1$ are both empty?

Solution

If we can figure out the probability that bins i and $i + 1$ are both empty for some particular i , then by symmetry and linearity of expectation we can just multiply by $n - 1$ to get the full answer.

For bins i and $i + 1$ to be empty, every ball must choose another bin. This occurs with probability $(1 - 2/n)^m$. The full answer is thus $n(1 - 2/n)^m$, or approximately $ne^{-2m/n}$ when n is large.

E.1.3 A labeled graph

Suppose you are given a graph $G = (V, E)$, where $|V| = n$, and you want to assign labels to each vertex such that the sum of the labels of each vertex and its neighbors modulo $n + 1$ is nonzero. Consider the naïve randomized algorithm that assigns a label in the range $0 \dots n$ to each vertex independently and uniformly at random and then tries again if the resulting labeling doesn't work. Show that this algorithm will find a correct labeling in time polynomial in n on average.

Solution

We'll use the law of total probability. First observe that the probability that a random labeling yields a zero sum for any single vertex and its neighbors is exactly $1/(n + 1)$; the easiest way to see this is that after conditioning on the values of the neighbors, there is only one value in $n + 1$ that can be assigned to the vertex itself to cause a failure. Now sum this probability over all n vertices to get a probability of failure of at most $n/(n + 1)$. It follows that after $n + 1$ attempts on average (each of which takes $O(n^2)$ time to check all the neighborhood sums), the algorithm will find a good labeling, giving a total expected time of $O(n^3)$.

E.1.4 Negative progress

An algorithm has the property that if it has already run for n steps, it runs for an additional $n + 1$ steps on average. Formally, let $T \geq 0$ be the random variable representing the running time of the algorithm, then

$$E[T \mid T \geq n] = 2n + 1. \quad (\text{E.1.1})$$

For each $n \geq 0$, what is the conditional probability $\Pr[T = n \mid T \geq n]$ that the algorithm stops just after its n -th step?

Solution

Expand (E.1.1) using the definition of conditional expectation to get

$$\begin{aligned}
 2n + 1 &= \sum_{x=0}^{\infty} x \Pr [T = x \mid T \geq n] \\
 &= \sum_{x=0}^{\infty} x \frac{\Pr [T = x \wedge T \geq n]}{\Pr [T \geq n]} \\
 &= \frac{1}{\Pr [T \geq n]} \sum_{x=n}^{\infty} x \Pr [T = x],
 \end{aligned}$$

which we can rearrange to get

$$\sum_{x=n}^{\infty} x \Pr [T = x] = (2n + 1) \Pr [T \geq n], \quad (\text{E.1.2})$$

provided $\Pr [T \geq n]$ is nonzero. We can justify this assumption by observing that (a) it holds for $n = 0$, because $T \geq 0$ always; and (b) if there is some $n > 0$ such that $\Pr [T \geq n] = 0$, then $E [T \mid T \geq n - 1] = n - 1$, contradicting (E.1.1).

Substituting $n + 1$ into (E.1.2) and subtracting from the original gives the equation

$$\begin{aligned}
 n \Pr [T = n] &= \sum_{x=n}^{\infty} x \Pr [T = x] - \sum_{x=n+1}^{\infty} x \Pr [T = x] \\
 &= (2n + 1) \Pr [T \geq n] - (2n + 3) \Pr [T \geq n + 1] \\
 &= (2n + 1) \Pr [T = n] + (2n + 1) \Pr [T \geq n + 1] - (2n + 3) \Pr [T \geq n + 1] \\
 &= (2n + 1) \Pr [T = n] - 2 \Pr [T \geq n + 1].
 \end{aligned}$$

Since we are looking for $\Pr [T = n \mid T \geq n] = \Pr [T = n] / \Pr [T \geq n]$, having an equation involving $\Pr [T \geq n + 1]$ is a bit annoying. But we can borrow a bit of $\Pr [T = n]$ from the other terms to make it work:

$$\begin{aligned}
 n \Pr [T = n] &= (2n + 1) \Pr [T = n] - 2 \Pr [T \geq n + 1] \\
 &= (2n + 3) \Pr [T = n] - 2 \Pr [T = n] - 2 \Pr [T \geq n + 1] \\
 &= (2n + 3) \Pr [T = n] - 2 \Pr [T \geq n].
 \end{aligned}$$

A little bit of algebra turns this into

$$\Pr [T = n \mid T \geq n] = \frac{\Pr [T = n]}{\Pr [T \geq n]} = \frac{2}{n + 3}.$$

E.2 Assignment 2: due Thursday, 2013-02-14, at 17:00

E.2.1 A local load-balancing algorithm

Suppose that we are trying to balance n jobs evenly between two machines. Job 1 chooses the left or right machine with equal probability. For $i > 1$, job i chooses the same machine as job $i - 1$ with probability p , and chooses the other machine with probability $1 - p$. This process continues until every job chooses a machine. We want to estimate how imbalanced the jobs are at the end of this process.

Let X_i be $+1$ if the i -th job chooses the left machine and -1 if it chooses the right. Then $S = \sum_{i=1}^n X_i$ is the difference between the number of jobs that choose the left machine and the number that choose the right. By symmetry, the expectation of S is zero. What is the variance of S as a function of p and n ?

Solution

To compute the variance, we'll use (5.1.5), which says that $\text{Var}[\sum_i X_i] = \sum_i \text{Var}[X_i] + 2 \sum_{i < j} \text{Cov}[X_i, X_j]$.

Recall that $\text{Cov}[X_i, X_j] = \text{E}[X_i X_j] - \text{E}[X_i] \text{E}[X_j]$. Since the last term is 0 (symmetry again), we just need to figure out $\text{E}[X_i X_j]$ for all $i \leq j$ (the $i = j$ case gets us $\text{Var}[X_i]$).

First, let's compute $\text{E}[X_j = 1 \mid X_i = 1]$. It's easiest to do this starting with the $j = i$ case: $\text{E}[X_i \mid X_i = 1] = 1$. For larger j , compute

$$\begin{aligned} \text{E}[X_j \mid X_{j-1}] &= pX_{j-1} + (1-p)(-X_{j-1}) \\ &= (2p-1)X_{j-1}. \end{aligned}$$

It follows that

$$\begin{aligned} \text{E}[X_j \mid X_i = 1] &= \text{E}[(2p-1)X_{j-1} \mid X_i = 1] \\ &= (2p-1)\text{E}[X_{j-1} \mid X_i = 1]. \end{aligned} \tag{E.2.1}$$

The solution to this recurrence is $\text{E}[X_j \mid X_i = 1] = (2p-1)^{j-i}$.

We next have

$$\begin{aligned}
 \text{Cov}[X_i, X_j] &= \text{E}[X_i X_j] \\
 &= \text{E}[X_i X_j \mid X_i = 1] \Pr[X_i = 1] + \text{E}[X_i X_j \mid X_i = -1] \Pr[X_i = -1] \\
 &= \frac{1}{2} \text{E}[X_j \mid X_i = 1] + \frac{1}{2} \text{E}[-X_j \mid X_i = -1] \\
 &= \frac{1}{2} \text{E}[X_j \mid X_i = 1] + \frac{1}{2} \text{E}[X_j \mid X_i = 1] \\
 &= \text{E}[X_j \mid X_i = 1] \\
 &= (2p - 1)^{j-i}
 \end{aligned}$$

as calculated in (E.2.1).

So now we just need to evaluate the horrible sum.

$$\begin{aligned}
 \sum_i \text{Var}[X_i] + 2 \sum_{i < j} \text{Cov}[X_i, X_j] &= n + 2 \sum_{i=1}^n \sum_{j=i+1}^n (2p - 1)^{j-i} \\
 &= n + 2 \sum_{i=1}^n \sum_{k=1}^{n-i} (2p - 1)^k \\
 &= n + 2 \sum_{i=1}^n \frac{(2p - 1) - (2p - 1)^{n-i+1}}{1 - (2p - 1)} \\
 &= n + \frac{n(2p - 1)}{1 - p} - \frac{1}{1 - p} \sum_{m=1}^n (2p - 1)^m \\
 &= n + \frac{n(2p - 1)}{1 - p} - \frac{(2p - 1) - (2p - 1)^{n+1}}{2(1 - p)^2}.
 \end{aligned} \tag{E.2.2}$$

This covers all but the $p = 1$ case, for which the geometric series formula fails. Here we can compute directly that $\text{Var}[S] = n^2$, since S will be $\pm n$ with equal probability.

For smaller values of p , plotting (E.2.2) shows the variance increasing smoothly starting at 0 (for even n) or 1 (for odd n) at $p = 0$ to n^2 in the limit as p goes to 1, with an interesting intermediate case of n at $p = 1/2$, where all terms but the first vanish. This makes a certain intuitive sense: when $p = 0$, the processes alternative which machine they take, which gives an even split for even n and a discrepancy of ± 1 for odd n ; when $p = 1/2$, the processes choose machines independently, giving variance n ; and for $p = 1$, the processes all choose the same machine, giving n^2 .

E.2.2 An assignment problem

Here is an algorithm for placing n balls into n bins. For each ball, we first select two bins uniformly at random without replacement. If at least one of the chosen bins is unoccupied, the ball is placed in the empty bin at no cost. If both chosen bins are occupied, we execute an expensive parallel scan operation to find an empty bin and place the ball there.

1. Compute the exact value of the expected number of scan operations.
2. Let $c > 0$. Show that the absolute value of the difference between the actual number of scan operations and the expected number is at most $O(\sqrt{cn \log n})$ with probability at least $1 - n^{-c}$.

Solution

1. Number the balls 1 to n . For ball i , there are $i-1$ bins already occupied, giving a probability of $\binom{i-1}{n} \binom{i-2}{n-1}$ that we choose an occupied bin on both attempts and incur a scan. Summing over all i gives us that the expected number of scans is:

$$\begin{aligned} \sum_{i=1}^n \binom{i-1}{n} \binom{i-2}{n-1} &= \frac{1}{n(n-1)} \sum_{i=1}^{n-1} (i^2 - i) \\ &= \frac{1}{n(n-1)} \left(\frac{(n-1)n(2n-1)}{6} - \frac{(n-1)n}{2} \right) \\ &= \frac{(2n-1)}{6} - \frac{1}{2} \\ &= \frac{n-2}{3}, \end{aligned}$$

provided $n \geq 2$. For $n < 2$, we incur no scans.

2. It's tempting to go after this using Chernoff's inequality, but in this case Hoeffding gives a better bound. Let S be the number of scans. Then S is the sum of n independent Bernoulli random variables, so 5.3.3 says that $\Pr[|S - \mathbb{E}[S]| \geq t] \leq 2e^{-2t^2/n}$. Now let $t = \sqrt{cn \ln n} = O(\sqrt{cn \log n})$ to make the right-hand side $2n^{-2c} \leq n^{-c}$ for sufficiently large n .

E.2.3 Detecting excessive collusion

Suppose you have n students in some unnamed Ivy League university's *Introduction to Congress* class, and each generates a random ± 1 vote, with

both outcomes having equal probability. It is expected that members of the same final club will vote together, so it may be that many groups of up to k students each will all vote the same way (flipping a single coin to determine the vote of all students in the group, with the coins for different groups being independent). However, there may also be a much larger conspiracy of exactly m students who all vote the same way (again based on a single independent fair coin), in violation of academic honesty regulations.

Let $c > 0$. How large must m be in asymptotic terms as a function of n , k , and c so that the existence of a conspiracy can be detected solely by looking at the total vote, where the probability of error (either incorrectly claiming a conspiracy when none exists or incorrectly claiming no conspiracy when one exists) is at most n^{-c} ?

Solution

Let S be the total vote. The intuition here is that if there is no conspiracy, S is concentrated around 0, and if there is a conspiracy, S is concentrated around $\pm m$. So if m is sufficiently large and $|S| \geq m/2$, we can reasonably guess that there is a conspiracy.

We need to prove two bounds: first, that the probability that we see $|S| \geq m/2$ when there is no conspiracy is small, and second, that the probability that we see $|S| < m/2$ when there is a conspiracy is large.

For the first case, let X_i be the total vote cast by the i -th group. This will be $\pm n_i$ with equal probability, where $n_i \leq k$ is the size of the group. This gives $E[X_i] = 0$. We also have that $\sum n_i = n$.

Because the X_i are all bounded, we can use Hoeffding's inequality (5.3.1), so long as we can compute an upper bound on $\sum n_i^2$. Here we use the fact that $\sum n_i^2$ is maximized subject to $0 \leq n_i \leq k$ and $\sum n_i = n$ by setting as many n_i as possible to k ; this follows from convexity of $x \mapsto x^2$.¹ We thus have

$$\begin{aligned} \sum n_i^2 &\leq \lfloor n/k \rfloor k^2 + (n \bmod k)^2 \\ &\leq \lfloor n/k \rfloor k^2 + (n \bmod k)k \\ &\leq (n/k)k^2 \\ &= nk. \end{aligned}$$

¹The easy way to see this is that if f is strictly convex, then f' is increasing. So if $0 < n_i \leq n_j < k$, increasing n_j by ϵ while decreasing n_i by ϵ leaves $\sum n_i$ unchanged while increasing $\sum f(n_i)$ by $\epsilon(f'(n_j) - f'(n_i)) + O(\epsilon^2)$, which will be positive when ϵ is small enough. So at any global maximum, we must have that at least one of n_i or n_j equals 0 or k for any $i \neq j$.

Now apply Hoeffding's inequality to get

$$\Pr [|S| \geq m/2] \leq 2e^{-(m/2)^2/2nk}.$$

We want to set m so that the right-hand side is less than n^{-c} . Taking logs as usual gives

$$\ln 2 - m^2/8nk \leq -c \ln n,$$

so the desired bound holds when

$$\begin{aligned} m &\geq \sqrt{8nk(c \ln n + \ln 2)} \\ &= \Omega(\sqrt{ckn \log n}). \end{aligned}$$

For the second case, repeat the above analysis on the $n - m$ votes except the $\pm m$ from the conspiracy. Again we get that if $m = \Omega(\sqrt{ckn \log n})$, the probability that these votes exceed $m/2$ is bounded by n^{-c} . So in both cases $m = \Omega(\sqrt{ckn \log n})$ is enough.

E.3 Assignment 3: due Wednesday, 2013-02-27, at 17:00

E.3.1 Going bowling

For your crimes, you are sentenced to play n frames of **bowling**, a game that involves knocking down pins with a heavy ball, which we are mostly interested in because of its complicated scoring system.

In each frame, your result may be any of the integers 0 through 9, a **spare** (marked as /), or a **strike** (marked as X). We can think of your result on the i -th frame as a random variable X_i . Each result gives a base score B_i , which is equal to X_i when $X_i \in \{0 \dots 9\}$ and 10 when $X_i \in \{/, X\}$. The actual score Y_i for frame i is the sum of the base scores for the frame and up to two subsequent frames, according to the rule:

$$Y_i = \begin{cases} B_i & \text{when } X_i \in \{0 \dots 9\}, \\ B_i + B_{i+1} & \text{when } X_i = /, \text{ and} \\ B_i + B_{i+1} + B_{i+2} & \text{when } X_i = X. \end{cases}$$

To ensure that B_{i+2} makes sense even for $i = n$, assume that there exist random variables X_{n+1} and X_{n+2} and the corresponding B_{n+1} and B_{n+2} .

Suppose that the X_i are independent (but not necessarily identically distributed). Show that your final score $S = \sum_{i=1}^n Y_i$ is exponentially concentrated² around its expected value.

Solution

This is a job for McDiarmid’s inequality (5.3.12). Observe that S is a function of $X_1 \dots X_{n+2}$. We need to show that changing any one of the X_i won’t change this function by too much.

From the description of the Y_i , we have that X_i can affect any of Y_{i-2} (if $X_{i-2} = \mathbf{x}$), Y_{i-1} (if $X_{i-1} = /$) and Y_i . We can get a crude bound by observing that each Y_i ranges from 0 to 30, so changing X_i can change $\sum Y_i$ by at most ± 90 , giving $c_i \leq 90$. A better bound can be obtained by observing that X_i contributes only B_i to each of Y_{i-2} and Y_{i-1} , so changing X_i can only change these values by up to 10; this gives $c_i \leq 50$. An even more pedantic bound can be obtained by observing that X_1 , X_2 , X_{n+1} , and X_{n+2} are all special cases, with $c_1 = 30$, $c_2 = 40$, $c_{n+1} = 20$, and $c_{n+2} = 10$, respectively; these values can be obtained by detailed meditation on the rules above.

We thus have $\sum_{i=1}^{n+2} c_i^2 = (n-2)50^2 + 30^2 + 40^2 + 20^2 + 10^2 = 2500(n-2) + 3000 = 2500n - 2000$, assuming $n \geq 2$. This gives $\Pr[|S - \mathbb{E}[S]| \geq t] \leq \exp(-2t^2/(2500n - 2000))$, with the symmetric bound holding on the other side as well.

For the standard game of bowling, with $n = 10$, this bound starts to bite at $t = \sqrt{11500} \approx 107$, which is more than a third of the range between the minimum and maximum possible scores. There’s a lot of variance in bowling, but this looks like a pretty loose bound for players who don’t throw a lot of strikes. For large n , we get the usual bound of $O(\sqrt{n \log n})$ with high probability: the averaging power of endless repetition eventually overcomes any slop in the constants.

E.3.2 Unbalanced treaps

Recall that a **treap** (§6.3) is only likely to be balanced if the sequence of insert and delete operations applied to it is independent of the priorities chosen by the algorithm.

Suppose that we insert the keys 1 through n into a treap with random priorities as usual, but then allow the adversary to selectively delete whichever key it wants to after observing the priorities assigned to each key.

²This means “more concentrated than you can show using Chebyshev’s inequality.”

Show that there is an adversary strategy that produces a path in the treap after deletions that has expected length $\Omega(\sqrt{n})$.

Solution

An easy way to do this is to produce a tree that consists of a single path, which we can do by arranging that the remaining keys have priorities that are ordered the same as their key values.

Here's a simple strategy that works. Divide the keys into \sqrt{n} ranges of \sqrt{n} keys each ($1 \dots \sqrt{n}$, $\sqrt{n} + 1 \dots 2\sqrt{n}$, etc.).³ Rank the priorities from 1 to n . From each range $(i-1)\sqrt{n} \dots i\sqrt{n}$, choose a key to keep whose priority is also ranked in the range $(i-1)\sqrt{n} \dots i\sqrt{n}$ (if there is one), or choose no key (if there isn't). Delete all the other keys.

For a particular range, we are drawing \sqrt{n} samples without replacement from the n priorities, and there are \sqrt{n} possible choices that cause us to keep a key in that range. The probability that every draw misses is $\prod_{i=1}^{\sqrt{n}} (1 - \sqrt{n}/(n-i+1)) \leq (1 - 1/\sqrt{n})^{\sqrt{n}} \leq e^{-1}$. So each range contributes at least $1 - e^{-1}$ keys on average. Summing over all \sqrt{n} ranges gives a sequence of keys with increasing priorities with expected length at least $(1 - e^{-1})\sqrt{n} = \Omega(\sqrt{n})$.

An alternative solution is to apply the **Erdős-Szekeres theorem** [ES35], which says that every sequence of length $k^2 + 1$ has either an increasing subsequence of length $k + 1$ or a decreasing sequence of $k + 1$. Consider the sequence of priorities corresponding to the keys $1 \dots n$; letting $k = \lfloor \sqrt{n-1} \rfloor$ gives a subsequence of length at least $\sqrt{n-1}$ that is either increasing or decreasing. If we delete all other elements of the treap, the elements corresponding to this subsequence will form a path, giving the desired bound. Note that this does not require any probabilistic reasoning at all.

Though not required for the problem, it's possible to show that $\Theta(\sqrt{n})$ is the best possible bound here. The idea is that the number of possible sequences of keys that correspond to a path of length k in a binary search tree is exactly $\binom{n}{k} 2^{k-1}$; the $\binom{n}{k}$ corresponds to choosing the keys in the path, and the 2^{k-1} is because for each node except the last, it must contain either the smallest or the largest of the remaining keys because of the binary search tree property.

Since each such sequence will be a treap path only if the priorities are decreasing (with probability $1/k!$), the union bound says that the probability

³To make our life easier, we'll assume that n is a square. This doesn't affect the asymptotic result.

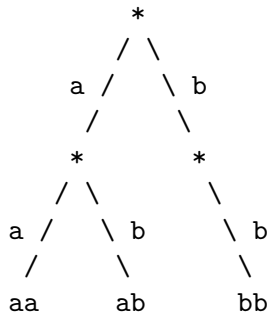


Figure E.1: A radix tree, storing the strings aa, ab, and ba.

of having any length- k paths is at most $\binom{n}{k}2^{k-1}/k!$. But

$$\begin{aligned} \binom{n}{k}2^{k-1}/k! &\leq \frac{(2n)^k}{2(k!)^2} \\ &\geq \frac{(2n)^k}{2(k/e)^{2k}} \\ &= \frac{1}{2}(2e^2n/k^2)^k. \end{aligned}$$

This is exponentially small for $k \gg \sqrt{2e^2n}$, showing that with high probability all possible paths have length $O(\sqrt{n})$.

E.3.3 Random radix trees

A **radix tree** over an alphabet of size m is a tree data structure where each node has up to m children, each corresponding to one of the letters in the alphabet. A string is represented by a node at the end of a path whose edges are labeled with the letters in the string in order. For example, in Figure E.1, the string **ab** is stored at the node reached by following the **a** edge out of the root, then the **b** edge out of this child.

The only nodes created in the radix tree are those corresponding to stored keys or ancestors of stored keys.

Suppose you have a radix tree into which you have already inserted n strings of length k from an alphabet of size m , generated uniformly at random with replacement. What is the expected number of new nodes you need to create to insert a new string of length k ?

Solution

We need to create a new node for each prefix of the new string that is not already represented in the tree.

For a prefix of length ℓ , the chance that none of the n strings have this prefix is exactly $(1 - m^{-\ell})^n$. Summing over all ℓ gives that the expected number of new nodes is $\sum_{\ell=0}^k (1 - m^{-\ell})^n$.

There is no particularly clean expression for this, but we can observe that $(1 - m^{-\ell})^n \leq \exp(-nm^{-\ell})$ is close to zero for $\ell < \log_m n$ and close to 1 for $\ell > \log_m n$. This suggests that the expected value is $k - \log_m n + O(1)$.

E.4 Assignment 4: due Wednesday, 2013-03-27, at 17:00

E.4.1 Flajolet-Martin sketches with deletion

A **Flajolet-Martin sketch** [FNM85] is a streaming data structure for approximately counting the number of distinct items n in a large data stream using only $m = O(\log n)$ bits of storage.⁴ The idea is to use a hash function h that generates each value $i \in \{1, \dots, m\}$ with probability 2^{-i} , and for each element x that arrives in the data stream, we write a 1 to $A[h(x)]$. (With probability 2^{-m} we get a value outside this range and write nowhere.) After inserting n distinct elements, we estimate n as $\hat{n} = 2^{\hat{k}}$, where $\hat{k} = \max\{k \mid A[k] = 1\}$, and argue that this is likely to be reasonably close to n .

Suppose that we modify the Flajolet-Martin sketch to allow an element x to be deleted by writing 0 to $A[h(x)]$. After n insertions and d deletions (of distinct elements in both cases), we estimate the number of remaining elements $n - d$ as before by $\widehat{n - d} = 2^{\hat{k}}$, where $\hat{k} = \max\{k \mid A[k] = 1\}$.

Assume that we never delete an element that has not previously been inserted, and that the values of h are for different inputs are independent of each other and of the sequence of insertions and deletions.

Show that there exist constants $c > 1$ and $\epsilon > 0$, such that for n sufficiently large, after inserting n distinct elements then deleting $d \leq \epsilon n$ of them, $\Pr\left[(n - d)/c \leq \widehat{n - d} \leq (n - d)c\right] \geq 2/3$.

⁴If you actually need to do this, there exist better data structures for this problem. See [KNW10].

Solution

We'll apply the usual error budget approach and show that the probability that $\widehat{n-d}$ is too big and the probability that $\widehat{n-d}$ is too small are both small. For the moment, we will leave c and ϵ as variables, and find values that work at the end.

Let's start with the too-big side. To get $A[k] = 1$, we need $h(x_i) = k$ for some x_i that is inserted but not subsequently deleted. There are $n-d$ such x_i , and each gives $h(x_i) = k$ with probability 2^{-k} . So $\Pr[A[k] = 1] \leq (n-d)2^{-k}$. This gives

$$\begin{aligned} \Pr[\widehat{n-d} \geq (n-d)c] &= \Pr[\widehat{k} \geq \lceil \lg((n-d)c) \rceil] \\ &\leq \sum_{k=\lceil \lg((n-d)c) \rceil}^{\infty} (n-d)2^{-k} \\ &= 2(n-d)2^{-\lceil \lg((n-d)c) \rceil} \\ &\leq \frac{2}{c}. \end{aligned}$$

On the too-small side, fix $k = \lceil \lg((n-d)/c) \rceil$. Since $A[k] = 1$ gives $\widehat{k} \geq k \geq \lceil \lg((n-d)/c) \rceil$, we have $\Pr[\widehat{n-d} < (n-d)/c] = \Pr[\widehat{k} < \lg(n-d)/c] \leq \Pr[A[k] = 0]$. (We might be able to get a better bound by looking at larger indices, but to solve the problem this one k will turn out to be enough.)

Let $x_1 \dots x_{n-d}$ be the values that are inserted and not later deleted, and $x_{n-d+1} \dots x_n$ the values that are inserted and then deleted. For $A[k]$ to be zero, either (a) no x_i for i in $1 \dots x_{n-d}$ has $h(x_i) = k$; or (b) some x_i for i in $n-d+1 \dots x_n$ has $h(x_i) = k$. The probability of the first event is

$(1 - 2^{-k})^{n-d}$; the probability of the second is $1 - (1 - 2^{-k})^d$. So we have

$$\begin{aligned}
 \Pr[A[k] = 0] &\leq (1 - 2^{-k})^{n-d} + \left(1 - (1 - 2^{-k})^d\right) \\
 &\leq \exp\left(-2^{-k}(n-d)\right) + \left(1 - \exp\left(-\left(2^{-k} + 2^{-2k}\right)d\right)\right) \\
 &\leq \exp\left(-2^{-\lceil \lg((n-d)/c) \rceil}(n-d)\right) + \left(1 - \exp\left(-2 \cdot 2^{-\lceil \lg((n-d)/c) \rceil}d\right)\right) \\
 &\leq \exp\left(-2^{-\lg((n-d)/c)}(n-d)\right) + \left(1 - \exp\left(-2 \cdot 2^{-\lg((n-d)/c)+1}d\right)\right) \\
 &= e^{-c} + \left(1 - \exp\left(-\frac{4cd}{n-d}\right)\right) \\
 &\leq e^{-c} + \frac{4cd}{n-d} \\
 &\leq e^{-c} + \frac{4c\epsilon}{1-\epsilon}.
 \end{aligned}$$

So our total probability of error is bounded by $\frac{2}{c} + e^{-c} + \frac{4c\epsilon}{1-\epsilon}$. Let $c = 8$ and $\epsilon = 1/128$ to make this less than $\frac{1}{4} + e^{-8} + \frac{128}{127} \cdot \frac{1}{16} \approx 0.313328 < 1/3$, giving the desired bound.

E.4.2 An adaptive hash table

Suppose we want to build a hash table, but we don't know how many elements we are going to put in it, and because we allow undisciplined C programmers to obtain pointers directly to our hash table entries, we can't move an element once we assign it a position to it. Here we will consider a data structure that attempts to solve this problem.

Construct a sequence of tables T_0, T_1, \dots , where each T_i has $m_i = 2^{2^i}$ slots. For each table T_i , choose k independent strongly 2-universal hash functions $h_{i1}, h_{i2}, \dots, h_{ik}$.

The insertion procedure is given in Algorithm E.1. The essentially idea is that we make k attempts (each with a different hash function) to fit x into T_0 , then k attempts to fit it in T_1 , and so on.

If the tables T_i are allocated only when needed, the space complexity of this data structure is given by the sum of m_i for all tables that have at least one element.

Show that for any fixed $\epsilon > 0$, there is a constant k such that after inserting n elements:

1. The expected cost of an additional insertion is $O(\log \log n)$, and
2. The expected space complexity is $O(n^{2+\epsilon})$.

```

1 procedure insert( $x$ )
2   for  $i \leftarrow 0$  to  $\infty$  do
3     for  $j \leftarrow 1$  to  $k$  do
4       if  $T_i[h_{ij}(x)] = \perp$  then
5          $T_i[h_{ij}(x)] \leftarrow x$ 
6       return

```

Algorithm E.1: Adaptive hash table insertion

Solution

The idea is that we use T_{i+1} only if we get a collision in T_i . Let X_i be the indicator for the event that there is a collision in T_i . Then

$$E[\text{steps}] \leq 1 + \sum_{i=0}^{\infty} E[X_i] \tag{E.4.1}$$

and

$$E[\text{space}] \leq m_0 + \sum_{i=0}^{\infty} E[X_i] m_{i+1}. \tag{E.4.2}$$

To bound $E[X_i]$, let's calculate an upper bound on the probability that a newly-inserted element x_{n+1} collides with any of the previous n elements $x_1 \dots x_n$ in table T_i . This occurs if, for every location $h_{ij}(x_{n+1})$, there is some x_r and some j' such that $h_{ij'}(x_r) = h_{ij}(x_{n+1})$. The chance that this occurs for any particular j, j' , and r is at most $1/m_i$ (if $j = j'$, use 2-universality of h_{ij} , and if $j \neq j'$, use independence and uniformity), giving a chance that it occurs for fixed j that is at most n/m_i . The chance that it occurs for all j is at most $(n/m_i)^k$, and the expected number of such collisions summed over any $n + 1$ elements that we insert is bounded by $n(n/m_i)^k$ (the first element can't collide with any previous elements). So we have $E[X_i] \leq \min(1, n(n/m_i)^k)$.

Let ℓ be the largest value such that $m_\ell \leq n^{2+\epsilon}$. We will show that, for an appropriate choice of k , we are sufficiently unlikely to get a collision in round ℓ that the right-hand sides of (E.4.1) and (E.4.2) end up being not much more than the corresponding sums up to $\ell - 1$.

From our choice of ℓ , it follows that (a) $\ell \leq \lg \lg n^{2+\epsilon} = \lg \lg n + \lg(2+\epsilon) = O(\log \log n)$; and (b) $m_{\ell+1} > n^{2+\epsilon}$, giving $m_\ell = \sqrt{m_{\ell+1}} > n^{1+\epsilon/2}$. From this we get $E[X_\ell] \leq n(n/m_\ell)^k < n^{1-k\epsilon/2}$.

By choosing k large enough, we can make this an arbitrarily small polynomial in n . Our goal is to wipe out the $E[X_\ell] m_{\ell+1}$ and subsequent terms in (E.4.2).

Observe that $E[X_i] m_{i+1} \leq n(n/m_i)^k m_i^2 = n^{k+1} m_i^{2-k}$. Let's choose k so that this is at most $1/m_i$, when $i \geq \ell$, so we get a nice convergent series.⁵ This requires $n^{k+1} m_i^{3-k} \leq 1$ or $k + 1 + (3 - k) \log_n m_i \leq 0$. If $i \geq \ell$, we have $\log_n m_i > 1 + \epsilon/2$, so we win if $k + 1 + (3 - k)(1 + \epsilon/2) \leq 0$. Setting $k \geq 8/\epsilon + 3$ works. (Since we want k to be an integer, we probably want $k = \lceil 8/\epsilon \rceil + 3$ instead.)

So now we have

$$\begin{aligned} E[\text{space}] &\leq m_0 + \sum_{i=0}^{\infty} E[X_i] m_{i+1} \\ &\leq \sum_{i=0}^{\ell} m_i + \sum_{i=\ell}^{\infty} \frac{1}{m_i} \\ &\leq 2m_\ell + \frac{2}{m_\ell} \\ &= O(n^{2+\epsilon}). \end{aligned}$$

For $E[\text{steps}]$, compute the same sum without all the m_{i+1} factors. This makes the tail terms even smaller, so they are still bounded by a constant, and the head becomes just $\sum_{i=0}^{\ell} 1 = O(\log \log n)$.

E.4.3 An odd locality-sensitive hash function

A deranged computer scientist decides that if taking one bit from a random index in a bit vector is a good way to do locality-sensitive hashing (see §7.7.1.3), then taking the exclusive OR of k independently chosen indices must be even better.

Formally, given a bit-vector $x_1 x_2 \dots x_n$, and a sequence of indices $i_1 i_2 \dots i_k$, define $h_i(x) = \bigoplus_{j=1}^k x_{i_j}$. For example, if $x = 00101$ and $i = 3, 5, 2$, $h_i(x) = 1 \oplus 1 \oplus 0 = 0$.

Suppose x and y are bit-vectors of length n that differ in m places.

1. Give a *closed-form expression* for the probability that $h_i(x) \neq h_i(y)$, assuming i consists of k indices chosen uniformly and independently at random from $1 \dots n$

⁵We could pick a smaller k , but why not make things easier for ourselves?

2. Use this to compute the exact probability that $h_i(x) \neq h_i(y)$ when $m = 0$, $m = n/2$, and $m = n$.

Hint: You may find it helpful to use the identity $(a \bmod 2) = \frac{1}{2}(1 - (-1)^a)$.

Solution

1. Observe that $h_i(x) \neq h_i(y)$ if and only if i chooses an odd number of indices where x and y differ. Let $p = m/n$ be the probability that each index in i hits a position where x and y differ, and let $q = 1 - p$. Then the event that we get an odd number of differences is

$$\begin{aligned} \sum_{j=0}^k (j \bmod 2) \binom{k}{j} p^j q^{k-j} &= \sum_{j=0}^k \frac{1}{2} (1 - (-1)^j) \binom{k}{j} p^j q^{k-j} \\ &= \frac{1}{2} \sum_{j=0}^k \binom{k}{j} p^j q^{k-j} - \frac{1}{2} \sum_{j=0}^k \binom{k}{j} (-p)^j q^{k-j} \\ &= \frac{1}{2} (p + q)^k - \frac{1}{2} (-p + q)^k \\ &= \frac{1 - (1 - 2(m/n))^k}{2}. \end{aligned}$$

2.
 - For $m = 0$, this is $\frac{1-1^k}{2} = 0$.
 - For $m = n/2$, it's $\frac{1-0^k}{2} = \frac{1}{2}$ (assuming $k > 0$).
 - For $m = n$, it's $\frac{1-(-1)^k}{2} = (k \bmod 2)$.

In fact, the chances of not colliding as a function of m are symmetric around $m = n/2$ if k is even and increasing if k is odd. So we can only hope to use this as locality-sensitive hash function in the odd case.

E.5 Assignment 5: due Friday, 2013-04-12, at 17:00

E.5.1 Choosing a random direction

Consider the following algorithm for choosing a random direction in three dimensions. Start at the point $(X_0, Y_0, Z_0) = (0, 0, 0)$. At each step, pick one of the three coordinates uniformly at random and add ± 1 to it with equal probability. Continue until the resulting vector has length at least k , i.e., until $X_t^2 + Y_t^2 + Z_t^2 \geq k^2$. Return this vector.

What is the expected running time of this algorithm, as an asymptotic function of k ?

Solution

The trick is to observe that $X_t^2 + Y_t^2 + Z_t^2 - t$ is a martingale, essentially following the same analysis as for $X_t^2 - t$ for a one-dimensional random walk. Suppose we pick X_t to change. Then

$$\begin{aligned} \mathbb{E} [X_{t+1}^2 \mid X_t] &= \frac{1}{2} \left((X_t + 1)^2 + (X_t - 1)^2 \right) \\ &= X_t^2 + 1. \end{aligned}$$

So

$$\mathbb{E} [X_{t+1}^2 + Y_{t+1}^2 + Z_{t+1}^2 - (t + 1) \mid X_t, Y_t, Z_t, X \text{ changes}] = X_t^2 + Y_t^2 + Z_t^2 - t.$$

But by symmetry, the same equation holds if we condition on Y or Z changing. It follows that $\mathbb{E} [X_{t+1}^2 + Y_{t+1}^2 + Z_{t+1}^2 - (t + 1) \mid X_t, Y_t, Z_t] = X_t^2 + Y_t^2 + Z_t^2 - t$, and that we have a martingale as claimed.

Let τ be the first time at which $X_t^2 + Y_t^2 + Z_t^2 \leq k^2$. From the optional stopping theorem (specifically, the bounded-increments case of Theorem 8.3.1), $\mathbb{E} [X_\tau^2 + Y_\tau^2 + Z_\tau^2 - \tau] = 0$, or equivalently $\mathbb{E} [\tau] = \mathbb{E} [X_\tau^2 + Y_\tau^2 + Z_\tau^2]$. This immediately gives $\mathbb{E} [\tau] \geq k^2$.

To get an upper bound, observe that $X_{\tau-1}^2 + Y_{\tau-1}^2 + Z_{\tau-1}^2 < k^2$, and that exactly one of these three term increases between $\tau - 1$ and τ . Suppose it's X (the other cases are symmetric). Increasing X by 1 sets $X_\tau^2 = X_{\tau-1}^2 + 2X_{\tau-1} + 1$. So we get

$$\begin{aligned} X_\tau^2 + Y_\tau^2 + Z_\tau^2 &= \left(X_{\tau-1}^2 + Y_{\tau-1}^2 + Z_{\tau-1}^2 \right) + 2X_{\tau-1} + 1 \\ &< k^2 + 2k + 1. \end{aligned}$$

So we have $k^2 \leq \mathbb{E} [\tau] < k^2 + 2k + 1$, giving $\mathbb{E} [\tau] = \Theta(k^2)$. (Or $k^2 + O(k)$ if we are feeling really precise.)

E.5.2 Random walk on a tree

Consider the following random walk on a (possibly unbalanced) binary search tree: At each step, with probability $1/3$ each, move to the current node's parent, left child, or right child. If the target node does not exist, stay put.

Suppose we adapt this random walk using Metropolis-Hastings (see §9.3.4) so that the probability of each node at depth d in the stationary distribution is proportional to α^{-d} .

Use a coupling argument to show that, for any constant $\alpha > 2$, this adapted random walk converges in $O(D)$ steps, where D is the depth of the tree.

Solution

As usual, let X_t be a copy of the chain starting in an arbitrary initial state and Y_t be a copy starting in the stationary distribution.

From the Metropolis-Hastings algorithm, the probability that the walk moves to a particular child is $1/3\alpha$, so the probability that the depth increases after one step is at most $2/3\alpha$. The probability that the walk moves to the parent (if we are not already at the root) is $1/3$.

We'll use the same choice (left, right, or parent) in both the X and Y processes, but it may be that only one of the particles moves (because the target node doesn't exist). To show convergence, we'll track $Z_t = \max(\text{depth}(X_t), \text{depth}(Y_t))$. When $Z_t = 0$, both X_t and Y_t are the root node.

There are two ways that Z_t can change:

1. Both processes choose "parent"; if Z_t is not already 0, $Z_{t+1} = Z_t - 1$. This case occurs with probability $1/3$.
2. Both processes choose one of the child directions. If the appropriate child exists for the deeper process (or for either process if they are at the same depth), we get $Z_{t+1} = Z_t + 1$. This even occurs with probability at most $2/3\alpha < 1/3$.

So the expected change in Z_{t+1} conditioned on $Z_t > 0$ is at most $-1/3 + 2/3\alpha = -(1/3)(2/\alpha - 1)$. Let τ be the first time at which $Z_t = 0$. Then the process $Z'_t = Z_t - (1/3)(2/\alpha - 1)t$ for $t \leq \tau$ and 0 for $t > \tau$ is a supermartingale, so $E[Z_\tau] = E[Z_0] = E[\max(\text{depth}(X_0), \text{depth}(Y_0))] \leq D$. This gives $E[\tau] \leq \frac{3D}{2/\alpha - 1}$.

E.5.3 Sampling from a tree

Suppose we want to sample from the stationary distribution of the Metropolis-Hastings walk in the previous problem, but we don't want to do an actual random walk. Assuming $\alpha > 2$ is a constant, give an algorithm for sampling *exactly* from the stationary distribution that runs in constant expected time.

Your algorithm should not require knowledge of the structure of the tree. Its only input should be a pointer to the root node.

Clarification added 2013-04-09: Your algorithm can determine the children of any node that it has already found. The idea of not knowing the structure of the tree is that it can't, for example, assume a known bound on the depth, counts of the number of nodes in subtrees, etc., without searching through the tree to find this information directly.

Solution

We'll use rejection sampling. The idea is to choose a node in the infinite binary tree with probability proportional to α^{-d} , and then repeat the process if we picked a node that doesn't actually exist. Conditioned on finding a node i that exists, its probability will be $\frac{\alpha^{\text{depth}(x)}}{\sum_j \alpha^{-\text{depth}(j)}}$.

If we think of a node in the infinite tree as indexed by a binary string of length equal to its depth, we can generate it by first choosing the length X and then choosing the bits in the string. We want $\Pr[X = n]$ to be proportional to $2^n \alpha^{-n} = (2/\alpha)^n$. Summing the geometric series gives

$$\Pr[X = n] = \frac{(2/\alpha)^n}{1 - (2/\alpha)}.$$

This is a geometric distribution, so we can sample it by repeatedly flipping a biased coin. The number of such coin-flips is $O(X)$, as is the number of random bits we need to generate and the number of tree nodes we need to check. The *expected* value of X is given by the infinite series

$$\sum_{n=0}^{\infty} \frac{(2/\alpha)^n n}{1 - (2/\alpha)},$$

this series converges to some constant by the ratio test.

So each probe costs $O(1)$ time on average, and has at least a constant probability of success, since we choose the root with $\Pr[X = 0] = \frac{1}{1-2/\alpha}$. Using Wald's equation (8.5.1), the total expected time to run the algorithm is $O(1)$.

This is a little surprising, since the output of the algorithm may have more than constant length. But we are interested in expectation, and when $\alpha > 2$ most of the weight lands near the top of the tree.

E.6 Assignment 6: due Friday, 2013-04-26, at 17:00**E.6.1 Increasing subsequences**

Let S_1, \dots, S_m be sets of indices in the range $1 \dots n$. Say that a permutation π of $1 \dots n$ is increasing on S_j if $\pi(i_1) < \pi(i_2) < \dots < \pi(i_{k_j})$ where $i_1 < i_2 < \dots < i_{k_j}$ are the elements of S_j .

Given a fully polynomial-time randomized approximation scheme that takes as input n and a sequence of sets S_1, \dots, S_m , and approximates the number of permutations π that are increasing on at least one of the S_j .

Solution

This can be solved using a fairly straightforward application of Karp-Luby [KL85] (see §10.3). Recall that for Karp-Luby we need to be able to express our target set U as the union of a polynomial number of covering sets U_j , where we can both compute the size of each U_j and sample uniformly from it. We can then estimate $|U| = \sum_{j,x \in U_j} f(j,x) = \left(\sum_j |U_j|\right) \Pr[f(j,x) = 1]$ where $f(j,x)$ is the indicator for the event that $x \notin U_{j'}$ for any $j' < j$ and in the probability, the pair (j,x) is chosen uniformly at random from $\{(j,x) \mid x \in U_j\}$.

In this case, let U_j be the set of all permutations that are increasing on S_j . We can specify each such permutation by specifying the choice of which $k_j = |S_j|$ elements are in positions $i_1 \dots i_{k_j}$ (the order of these elements is determined by the requirement that the permutation be increasing on S_j) and specifying the order of the remaining $n - k_j$ elements. This gives $\binom{n}{k_j}(n - k_j)! = (n)_{n-k_j}$ such permutations. Begin by computing these counts for all S_j , as well as their sum.

We now wish to sample uniformly from pairs (j, π) where each π is an element of S_j . First sample each j with probability $|S_j| / \sum_\ell |S_\ell|$, using the counts we've already computed. Sampling a permutation uniformly from S_j mirrors the counting argument: choose a k_j -subset for the positions in S_j , then order the remaining elements randomly. The entire sampling step can easily be done in $O(n + m)$ time.

Computing $f(j, \pi)$ requires testing π to see if it is increasing for any $S_{j'}$ for $j < j'$; without doing anything particularly intelligent, this takes $O(nm)$ time. So we can construct and test one sample in $O(nm)$ time. Since each sample has at least a $\rho = 1/m$ chance of having $f(j, \pi) = 1$, from Lemma 10.2.1 we need $O\left(\frac{1}{\epsilon^2 \rho} \log \frac{1}{\delta}\right) = O\left(m\epsilon^{-2} \log \frac{1}{\delta}\right)$ samples to get relative error with probability at least $1 - \delta$, for a total cost of $O\left(m^2 n \epsilon^{-2} \log \frac{1}{\delta}\right)$.

E.6.2 Futile word searches

A **word search puzzle** consists of an $n \times n$ grid of letters from some alphabet Σ , where the goal is to find contiguous sequences of letters in one of the eight orthogonal or diagonal directions that form words from some lexicon. For example, in Figure E.2, the left grid contains an instance of **aabc** (running up and left from the rightmost a character on the last line), while the right grid contains no instances of this word.

For this problem, you are asked to build an algorithm for constructing

```

bacab  bacab
ccaac  ccaac
bbbac  babac
bbaaa  bbaaa
acbab  acbab

```

Figure E.2: Non-futile (left) and futile (right) word search grids for the lexicon $\{\text{aabc}, \text{ccca}\}$

word search puzzles with no solution for a given lexicon. That is, given a set of words S over some alphabet and a grid size n , the output should be an $n \times n$ grid of letters such that no word in S appears as a contiguous sequence of letters in one of the eight directions anywhere in the grid. We will refer to such puzzles as **futile word search puzzles**.

1. Suppose the maximum length of any word in S is k . Let p_S be the probability that some word in S is a prefix of an infinite string generated by picking letters uniformly and independently from Σ . Show that there is a constant $c > 0$ such that for any k , Σ , and S , $p_S < ck^{-2}$ implies that there exists, for all n , an $n \times n$ futile word search puzzle for S using only letters from Σ .
2. Give an algorithm that constructs a futile word search puzzle given S and n in expected time polynomial in $|S|$, k , and n , provided $p_S < ck^{-2}$ as above.

Solution

1. We'll apply the symmetric version of the Lovász local lemma. Suppose the grid is filled in independently and uniformly at random with characters from Σ . Given a position ij in the grid, let A_{ij} be the event that there exists a word in S whose first character is at position ij ; observe that $\Pr[A_{ij}] \leq 8p_S$ by the union bound (this may be an overestimate, both because we might run off the grid in some directions and because the choice of initial character is not independent). Observe also that A_{ij} is independent of any event $A_{i'j'}$ where $|i - i'| \geq 2k - 1$ or $|j - j'| \geq 2k - 1$, because no two words starting at these positions can overlap. So we can build a dependency graph with $p \leq 8p_S$ and $d \leq (4k - 3)^2$. The Lovász local lemma shows that there exists an assignment

where no A_{ij} occurs provided $ep(d+1) < 1$ or $8eps((4k-3)^2+1) < 1$. This easily holds if $p_S < \frac{1}{8e(4k^2)} = \frac{1}{128e}k^{-2}$.

- For this part, we can just use Moser-Tardos [MT10], particularly the symmetric version described in Corollary 11.3.4. We have a collection of $m = O(n^2)$ bad events, with $d = \Theta(k^2)$, so the expected number of resamplings is bounded by $m/d = O(n^2/k^2)$. Each resampling requires checking every position in the new grid for an occurrence of some string in S ; this takes $O(n^2k \cdot |S|)$ time per resampling even if we are not very clever about it. So the total expected cost is $O(n^4 \cdot |S|/k)$.

With some more intelligence, this can be improved. We don't need to recheck any position at distance greater than k from any of the at most k letters we resample, and if we are sensible, we can store S using a radix tree or some similar data structure that allows us to look up all words that appear as a prefix of a given length- k string in time $O(k)$. This reduces the cost of each resampling to $O(k^3)$, with an additive cost of $O(k \cdot |S|)$ to initialize the data structure. So the total expected cost is now $O(n^2k + |S|)$.

E.6.3 Balance of power

Suppose you are given a set of n MMORPG players, and a sequence of subsets S_1, S_2, \dots, S_m of this set, where each subset S_i gives the players who will participate in some raid. Before any of the raids take place, you are to assign each player permanently to one of three factions. If for any i , $|S_i|/2$ or more of the players are in the same faction, then instead of carrying out the raid they will overwhelm and rob the other participants.

Give a randomized algorithm for computing a faction assignment that prevents this tragedy from occurring (for all i) and thus allows all m raids to be completed without incident, assuming that $m > 1$ and $\min_i |S_i| \geq c \ln m$ for some constant $c > 0$ that does not depend on n or m . Your algorithm should run in expected time polynomial in n and m .

Solution

Assign each player randomly to a faction. Let X_{ij} be the number of players in S_i that are assigned to faction j . Then $E[X_{ij}] = |S_i|/3$. Applying the

Chernoff bound (5.2.2), we have

$$\begin{aligned}\Pr[X_{ij} \geq |S_i|/2] &= \Pr\left[X_{ij} \geq \left(1 + \frac{1}{2}\right) \mathbb{E}[X_{ij}]\right] \\ &\leq \exp\left(-(|S_i|/3) \left(\frac{1}{2}\right)^2 / 3\right) \\ &= e^{-|S_i|/36}.\end{aligned}$$

Let $c = 3 \cdot 36 = 108$. Then if $\min_i |S_i| \geq c \ln m$, for each i, j , it holds that $\Pr[X_{ij} \geq |S_i|/2] \leq e^{-3 \ln m} = m^{-3}$. So the probability that this bound is exceeded for any i and j is at most $(3m)m^{-2} = 3/m^2$. So a random assignment works with at least $1/4$ probability for $m > 1$.

We can generate and test each assignment in $O(nm)$ time. So our expected time is $O(nm)$.

E.7 Final exam

Write your answers in the blue book(s). Justify your answers. Work alone. Do not use any notes or books.

There are four problems on this exam, each worth 20 points, for a total of 80 points. You have approximately three hours to complete this exam.

E.7.1 Dominating sets

A **dominating set** in a graph is a subset S of the vertices for which every vertex v is either in S or adjacent to a vertex in S .

Show that for any graph G , there is an aperiodic, irreducible Markov chain on the dominating sets of G , such that (a) the transition rule for the chain can be implemented in polynomial time; and (b) the stationary distribution of the chain is uniform. (You do not need to say anything about the convergence rate of this chain.)

Solution

Suppose we have state S_t at time t . We will use a random walk where we choose a vertex uniformly at random to add to or remove from S_t , and carry out the action only if the resulting set is still a dominating set.

In more detail: For each vertex v , with probability $1/n$, $S_{t+1} = S_t \cup \{v\}$ if $v \notin S_t$, $S_{t+1} = S_t \setminus \{v\}$ if $v \in S_t$ and $S_t \setminus \{v\}$ is a dominating set, and $S_{t+1} = S_t$ otherwise. To implement this transition rule, we need to be able

to choose a vertex v uniformly at random (easy) and test in the case where $v \in S_t$ if $S_t \setminus \{v\}$ is a dominating set (also polynomial: for each vertex, check if it or one of its neighbors is in $S_t \setminus \{v\}$, which takes time $O(|V| + |E|)$). Note that we do not need to check if $S_t \cup \{v\}$ is a dominating set.

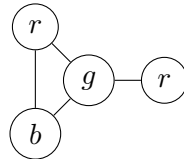
For any pair of adjacent states S and $S' = S \setminus \{v\}$ the probability of moving from S to S' and the probability of moving from S' to S are both $1/n$. So the Markov chain is reversible with a uniform stationary distribution.

This is an aperiodic chain, because there exist minimal dominating sets for which there is a nonzero chance that $S_{t+1} = S_t$.

It is irreducible, because for any dominating set S , there is a path to the complete set of vertices V by adding each vertex in $V \setminus S$ one at a time. Conversely, removing these vertices from V gives a path to S . This gives a path $S \rightsquigarrow V \rightsquigarrow T$ between any two dominating sets S and T .

E.7.2 Tricolor triangles

Suppose you wish to count the number of assignments of colors $\{r, g, b\}$ to nodes of a graph G that have the property that some triangle in G contains all three colors. For example, the four-vertex graph shown below is labeled with one of 18 such colorings (6 permutations of the colors of the triangle nodes times 3 unconstrained choices for the degree-1 node).



Give a fully polynomial-time randomized approximation scheme for this problem.

Solution

Though it is possible, and tempting, to go after this using Karp-Luby (see §10.3), naive sampling is enough.

If a graph has at least one triangle (which can be checked in $O(n^3)$ time just by enumerating all possible triangles), then the probability that that particular triangle is tricolored when colors are chosen uniformly and independently at random is $6/27 = 2/9$. This gives a constant hit rate ρ , so by Lemma 10.2.1, we can get ϵ relative error with $1 - \delta$ probability using

$O\left(\frac{1}{\epsilon^2} \log \frac{1}{\delta}\right)$ samples. Each sample costs $O(n^3)$ time to evaluate (again, brute-force checking of all possible triangles), for a total cost of $O\left(n^3 \epsilon^{-2} \log \frac{1}{\delta}\right)$.

E.7.3 The n rooks problem

The n rooks problem requires marking as large a subset as possible of the squares in an $n \times n$ grid, so that no two squares in the same row or column are marked.⁶

Consider the following randomized algorithm that attempts to solve this problem:

1. Give each of the n^2 squares a distinct label using a uniformly chosen random permutation of the integers $1 \dots n^2$.
2. Mark any square whose label is larger than any other label in its row and column.

What is the expected number of marked squares?

Solution

Each square is marked if it is the largest of the $2n - 1$ total squares in its row and column. By symmetry, each of these $2n - 1$ squares is equally likely to be the largest, so the probability that a particular square is marked is exactly $\frac{1}{2n-1}$. By linearity of expectation, the total expected number of marked squares is then $\frac{n^2}{2n-1}$.

E.7.4 Pursuing an invisible target on a ring

Suppose that you start at position 0 on a ring of size $2n$, while a target particle starts at position n . At each step, starting at position i , you can choose whether to move to any of positions $i - 1$, i , or $i + 1$. At the same time, the target moves from its position j to either $j - 1$ or $j + 1$ with equal probability, independent of its previous moves or your moves. Aside from knowing that the target starts at n at time 0, you cannot tell where the target moves.

Your goal is to end up on the same node as the target after some step.⁷ Give an algorithm for choosing your moves such that, for any $c > 0$, you

⁶This is not actually a hard problem.

⁷Note that you must be in the same place at the end of the step: if you move from 1 to 2 while the target moves from 2 to 1, that doesn't count.

encounter the target in at most $2n$ steps with probability at least $1 - n^{-c}$ for sufficiently large n .

Solution

The basic idea is to just go through positions $0, 1, 2, \dots$ until we encounter the target, but we have to be a little careful about parity to make sure we don't pass it by accident.⁸

Let $X_i = \pm 1$ be the increment of the target's i -th move, and let $S_i = \sum_{j=1}^i X_j$, so that its position after i steps is $n + S_i \bmod 2n$.

Let Y_i be the position of the pursuer after i steps.

First move: stay at 0 if n is odd, move to 1 if n is even. The purpose of this is to establish the invariant that $n + S_i - Y_i$ is even starting at $i = 1$. For subsequent moves, let $Y_{i+1} = Y_i + 1$. Observe that this maintains the invariant.

We assume that n is at least 2. This is necessary to ensure that at time 1, $Y_1 \leq n + S_1$.

Claim: if at time $2n$, $Y_{2n} \geq n + S_{2n}$, then at some time $i \leq 2n$, $Y_i = n + S_i$.

Proof: Let i be the first time at which $Y_i \geq n + S_i$; under the assumption that $n \geq 2$, $i \geq 1$. So from the invariant, we can't have $Y_i = n + S_i + 1$, and if $Y_i \geq n + S_i + 2$, we have $Y_{i-1} \geq Y_i - 1 \geq n + S_i + 1 \geq n + S_{i-1}$, contradicting our assumption that i is minimal. The remaining alternative is that $Y_i = n + S_i$, giving a collision at time i .

We now argue that $Y_{2n} \geq n - 1$ is very likely to be at least $n + S_{2n}$. Since S_{2n} is a sum of $2n$ independent ± 1 variables, from Hoeffding's inequality we have $\Pr[Y_n < n + S_{2n}] \leq \Pr[S_{2n} \geq n] \leq e^{-n^2/4n} = e^{-n/4}$. For sufficiently large n , this is much smaller than n^{-c} for any fixed c .

⁸This is not the only possible algorithm, but there are a lot of plausible-looking algorithms that turn out not to work. One particularly tempting approach is to run to position n using the first n steps and then spend the next n steps trying to hit the target in the immediate neighborhood of n , either by staying put (a sensible strategy when lost in the woods in real life, assuming somebody is looking for you), or moving in a random walk of some sort starting at n . This doesn't work if we want a high-probability bound. To see this, observe that the target has a small but nonzero constant probability in the limit of begin at some position greater than or equal to $n + 4\sqrt{n}$ after exactly $n/2$ steps. Conditioned on starting at $n + 4\sqrt{n}$ or above, its chances of moving below $n + 4\sqrt{n} - 2\sqrt{n} = n + 2\sqrt{n}$ at any time in the next $3n/2$ steps is bounded by $e^{-4n/2(3n/2)} = e^{-4/3}$ (Azuma), and a similar bound holds independently for our chances of getting up to $n + 2\sqrt{n}$ or above. Multiplying out all these constants gives a constant probability of failure. A similar but bigger disaster occurs if we don't rush to n first.

Appendix F

Sample assignments from Spring 2011

F.1 Assignment 1: due Wednesday, 2011-01-26, at 17:00

F.1.1 Bureaucratic part

Send me email! My address is james.aspnes@gmail.com.

In your message, include:

1. Your name.
2. Your status: whether you are an undergraduate, grad student, auditor, etc.
3. Anything else you'd like to say.

(You will not be graded on the bureaucratic part, but you should do it anyway.)

F.1.2 Rolling a die

The usual model of a randomized algorithm assumes a source of fair, independent random bits. This makes it easy to generate uniform numbers in the range $0 \dots 2^n - 1$, but not so easy for other ranges. Here are two algorithms for generating a uniform random integer $0 \leq s < n$:

- **Rejection sampling** generates a uniform random integer $0 \leq s < 2^{\lceil \lg n \rceil}$. If $s < n$, return s ; otherwise keep trying until you get some $s < n$.
 - **Arithmetic coding or range coding** generates a sequence of bits r_1, r_2, \dots, r_k until the half-open interval $[\sum_{i=1}^k 2^{-i} r_i, \sum_{i=1}^k 2^{-i} r_i + 2^{-k-1})$ is a subset of $[s/n, (s+1)/n)$ for some s ; it then returns s .
1. Show that both rejection sampling and range coding produce a uniform value $0 \leq s < n$ using an expected $O(\log n)$ random bits.
 2. Which algorithm has a better constant?
 3. Does there exist a function f and an algorithm that produces a uniform value $0 \leq s < n$ for any n using $f(n)$ random bits with probability 1?

Solution

1. For rejection sampling, each sample requires $\lceil \lg n \rceil$ bits and is accepted with probability $n/2^{\lceil \lg n \rceil} \geq 1/2$. So rejection sampling returns a value after at most 2 samples on average, using no more than an expected $2\lceil \lg n \rceil < 2(\lg n + 1)$ expected bits for the worst n .

For range coding, we keep going as long as one of the $n-1$ nonzero endpoints s/n lies inside the current interval. After k bits, the probability that one of the 2^k intervals contains an endpoint is at most $(n-1)2^{-k}$; in particular, it drops below 1 as soon as $k = 2^{\lceil \lg n \rceil}$ and continues to drop by $1/2$ for each additional bit, requiring 2 more bits on average. So the expected cost of range coding is at most $\lceil \lg n \rceil + 2 < \lg n + 3$ bits.

2. We've just shown that range coding beats rejection sampling by a factor of 2 in the limit, for worst-case n . It's worth noting that other factors might be more important if random bits are cheap: rejection sampling is much easier to code and avoids the need for division.
3. There is no algorithm that produces a uniform value $0 \leq s < n$ for all n using any fixed number of bits. Suppose such an algorithm existed. Fix some n . For all n values s to be equally likely, the sets of random bits $M^{-1}(s) = \{r \mid M(r) = s\}$ must have the same size. But this can only happen if n divides $2^{f(n)}$, which works only for n a power of 2.

F.1.3 Rolling many dice

Suppose you repeatedly roll an n -sided die. Give an asymptotic (big- Θ) bound on the expected number of rolls until you roll some number you have already rolled before.

Solution

In principle, it is possible to compute this value exactly, but we are lazy.

For a lower bound, observe that after m rolls, each of the $\binom{m}{2}$ pairs of rolls has probability $1/n$ of being equal, for an expected total of $\binom{m}{2}/n$ duplicates. For $m = \sqrt{n}/2$, this is less than $1/8$, which shows that the expected number of rolls is $\Omega(\sqrt{n})$.

For the upper bound, suppose we have already rolled the die \sqrt{n} times. If we haven't gotten a duplicate already, each new roll has probability at least $\sqrt{n}/n = 1/\sqrt{n}$ of matching a previous roll. So after an additional \sqrt{n} rolls on average, we get a repeat. This shows that the expected number of rolls is $O(\sqrt{n})$.

Combining these bounds shows that we need $\Theta(\sqrt{n})$ rolls on average.

F.1.4 All must have candy

A set of n_0 children each reach for one of n_0 candies, with each child choosing a candy independently and uniformly at random. If a candy is chosen by exactly one child, the candy and child drop out. The remaining n_1 children and candies then repeat the process for another round, leaving n_2 remaining children and candies, etc. The process continues until every child has a candy.

Give the best bound you can on the expected number of rounds until every child has a candy.

Solution

Let $T(n)$ be the expected number of rounds remaining given we are starting with n candies. We can set up a probabilistic recurrence relation $T(n) = 1 + T(n - X_n)$ where X_n is the number of candies chosen by exactly one child. It is easy to compute $E[X_n]$, since the probability that any candy gets chosen exactly once is $n(1/n)(1 - 1/n)^{n-1} = (1 - 1/n)^{n-1}$. Summing over all candies gives $E[X_n] = n(1 - 1/n)^{n-1}$.

The term $(1 - 1/n)^{n-1}$ approaches e^{-1} in the limit, so for any fixed $\epsilon > 0$, we have $n(1 - 1/n)^{n-1} \geq n(e^{-1} - \epsilon)$ for sufficiently large n . We can get a quick bound by choosing ϵ so that $e^{-1} - \epsilon \geq 1/4$ (for example) and then

applying the Karp-Upfal-Wigderson inequality (H.3.1) with $\mu(n) = n/4$ to get

$$\begin{aligned} \mathbb{E}[T(n)] &\leq \int_1^n \frac{1}{t/4} dt \\ &= 4 \ln n. \end{aligned}$$

There is a sneaky trick here, which is that we stop if we get down to 1 candy instead of 0. This avoids the usual problem with KUW and $\ln 0$, by observing that we can't ever get down to exactly one candy: if there were exactly one candy that gets grabbed twice or not at all, then there must be some other candy that also gets grabbed twice or not at all.

This analysis is sufficient for an asymptotic estimate: the last candy gets grabbed in $O(\log n)$ rounds on average. For most computer-science purposes, we'd be done here.

We can improve the constant slightly by observing that $(1 - 1/n)^{n-1}$ is in fact always greater than or equal to e^{-1} . The easiest way to see this is to plot the function, but if we want to prove it formally we can show that $(1 - 1/n)^{n-1}$ is a decreasing function by taking the derivative of its logarithm:

$$\begin{aligned} \frac{d}{dn} \ln(1 - 1/n)^{n-1} &= \frac{d}{dn} (n-1) \ln(1 - 1/n) \\ &= \ln(1 - 1/n) + \frac{n-1}{1 - 1/n} \cdot \frac{-1}{n^2}. \end{aligned}$$

and observing that it is negative for $n > 1$ (we could also take the derivative of the original function, but it's less obvious that it's negative). So if it approaches e^{-1} in the limit, it must do so from above, implying $(1 - 1/n)^{n-1} \geq e^{-1}$.

This lets us apply (H.3.1) with $\mu(n) = n/e$, giving $\mathbb{E}[T(n)] \leq e \ln n$.

If we skip the KUW bound and use the analysis in §H.4.2 instead, we get that $\Pr[T(n) \geq \ln n + \ln(1/\epsilon)] \leq \epsilon$. This suggests that the actual expected value should be $(1 + o(1)) \ln n$.

F.2 Assignment 2: due Wednesday, 2011-02-09, at 17:00

F.2.1 Randomized dominating set

A **dominating set** in a graph $G = (V, E)$ is a set of vertices D such that each of the n vertices in V is either in D or adjacent to a vertex in D .

Suppose we have a d -regular graph, in which every vertex has exactly d neighbors. Let D_1 be a random subset of V in which each vertex appears with independent probability p . Let D be the union of D_1 and the set of all vertices that are not adjacent to any vertex in D_1 . (This construction is related to a classic maximal independent set algorithm of Luby [Lub85], and has the desirable property in a distributed system of finding a dominating set in only one round of communication.)

1. What would be a good value of p if our goal is to minimize $E[|D|]$, and what bound on $E[|D|]$ does this value give?
2. For your choice of p above, what bound can you get on $\Pr[|D| - E[|D|] \geq t]$?

Solution

1. First let's compute $E[|D|]$. Let X_v be the indicator for the event that $v \in D$. Then $X_v = 1$ if either (a) v is in D_1 , which occurs with probability p ; or (b) v and all d of its neighbors are not in D_1 , which occurs with probability $(1 - p)^{d+1}$. Adding these two cases gives $E[X_v] = p + (1 - p)^{d+1}$ and thus

$$E[|D|] = \sum_v E[X_v] = n \left(p + (1 - p)^{d+1} \right). \quad (\text{F.2.1})$$

We optimize $E[|D|]$ in the usual way, by seeking a minimum for $E[X_v]$. Differentiating with respect to p and setting to 0 gives $1 - (d + 1)(1 - p)^d = 0$, which we can solve to get $p = 1 - (d + 1)^{-1/d}$. (We can easily observe that this must be a minimum because setting p to either 0 or 1 gives $E[X_v] = 1$.)

The value of $E[|D|]$ for this value of p is the rather nasty expression $n(1 - (d + 1)^{-1/d} + (d + 1)^{-1-1/d})$.

Plotting the d factor up suggests that it goes to $\ln d/d$ in the limit, and both Maxima and www.wolframalpha.com agree with this. Knowing

the answer, we can prove it by showing

$$\begin{aligned}
 \lim_{d \rightarrow \infty} \frac{1 - (d+1)^{-1/d} + (d+1)^{-1-1/d}}{\ln d/d} &= \lim_{d \rightarrow \infty} \frac{1 - d^{-1/d} + d^{-1-1/d}}{\ln d/d} \\
 &= \lim_{d \rightarrow \infty} \frac{1 - d^{-1/d}}{\ln d/d} \\
 &= \lim_{d \rightarrow \infty} \frac{1 - e^{-\ln d/d}}{\ln d/d} \\
 &= \lim_{d \rightarrow \infty} \frac{1 - \left(1 - \ln d/d + O(\ln^2 d/d^2)\right)}{\ln d/d} \\
 &= \lim_{d \rightarrow \infty} \frac{\ln d/d + O(\ln^2 d/d^2)}{\ln d/d} \\
 &= \lim_{d \rightarrow \infty} (1 + O(\ln d/d)) \\
 &= 1.
 \end{aligned}$$

This lets us write $E[|D|] = (1 + o(1))n \ln d/d$, where we bear in mind that the $o(1)$ term depends on d but not n .

- Suppose we fix X_v for all but one vertex u . Changing X_u from 0 to 1 can increase $|D|$ by at most one (if u wasn't already in D) and can decrease it by at most $d - 1$ (if u wasn't already in D and adding u to D_1 lets all d of u 's neighbors drop out). So we can apply the method of bounded differences with $c_i = d - 1$ to get

$$\Pr[|D| - E[|D|] \geq t] \leq \exp\left(-\frac{t^2}{2n(d-1)^2}\right).$$

A curious feature of this bound is that it doesn't depend on p at all. It may be possible to get a tighter bound using a better analysis, which might pay off for very large d (say, $d \gg \sqrt{n}$).

F.2.2 Chernoff bounds with variable probabilities

Let $X_1 \dots X_n$ be a sequence of Bernoulli random variables, where for all i , $E[X_i | X_1 \dots X_{i-1}] \leq p_i$. Let $S = \sum_{i=1}^n X_i$ and $\mu = \sum_{i=1}^n p_i$. Show that, for all $\delta \geq 0$,

$$\Pr[S \geq (1 + \delta)\mu] \leq \left(\frac{e^\delta}{(1 + \delta)^{1+\delta}}\right)^\mu.$$

Solution

Let $S_t = \sum_{i=1}^t X_i$, so that $S = S_n$, and let $\mu_t = \sum_{i=1}^t p_i$. We'll show by induction on t that $\mathbb{E} [e^{\alpha S_t}] \leq \exp(e^{\alpha-1} \mu_t)$, when $\alpha > 0$.

Compute

$$\begin{aligned}
 \mathbb{E} [e^{\alpha S}] &= \mathbb{E} [e^{\alpha S_{n-1}} e^{\alpha X_n}] \\
 &= \mathbb{E} [e^{\alpha S_{n-1}} \mathbb{E} [e^{\alpha X_n} \mid X_1, \dots, X_{n-1}]] \\
 &= \mathbb{E} [e^{\alpha S_{n-1}} (\Pr [X_n = 0 \mid X_1, \dots, X_{n-1}] + e^\alpha \Pr [X_n = 1 \mid X_1, \dots, X_{n-1}])] \\
 &= \mathbb{E} [e^{\alpha S_{n-1}} (1 + (e^\alpha - 1) \Pr [X_n = 1 \mid X_1, \dots, X_{n-1}])] \\
 &\leq \mathbb{E} [e^{\alpha S_{n-1}} (1 + (e^\alpha - 1)p_n)] \\
 &\leq \mathbb{E} [e^{\alpha S_{n-1}} \exp((e^\alpha - 1)p_n)] \\
 &\leq \mathbb{E} [e^{\alpha S_{n-1}}] \exp((e^\alpha - 1)p_n) \\
 &\leq \exp(e^\alpha - 1)\mu_{n-1} \exp((e^\alpha - 1)p_n) \\
 &= \exp(e^\alpha - 1)\mu_n.
 \end{aligned}$$

Now apply the rest of the proof of (5.2.1) to get the full result.

F.2.3 Long runs

Let W be a binary string of length n , generated uniformly at random. Define a **run** of ones as a maximal sequence of contiguous ones; for example, the string 1110011001111101011 contains 5 runs of ones, of length 3, 2, 6, 1, and 2.

Let X_k be the number of runs in W of length k or more.

1. Compute the exact value of $\mathbb{E} [X_k]$ as a function of n and k .
2. Give the best concentration bound you can for $|X_k - \mathbb{E} [X_k]|$.

Solution

1. We'll compute the probability that any particular position $i = 1 \dots n$ is the start of a run of length k or more, then sum over all i . For a run of length k to start at position i , either (a) $i = 1$ and $W_i \dots W_{i+k-1}$ are all 1, or (b) $i > 1$, $W_{i-1} = 0$, and $W_i \dots W_{i+k-1}$ are all 1. Assuming $n \geq k$, case (a) adds 2^{-k} to $\mathbb{E} [X_k]$ and case (b) adds $(n-k)2^{-k-1}$, for a total of $2^{-k} + (n-k)2^{-k-1} = (n-k+2)2^{-k-1}$.

2. We can get an easy bound without too much cleverness using McDiarmid’s inequality (5.3.12). Observe that X_k is a function of the independent random variables $W_1 \dots W_n$ and that changing one of these bits changes X_k by at most 1 (this can happen in several ways: a previous run of length $k-1$ can become a run of length k or vice versa, or two runs of length k or more separated by a single zero may become a single run, or vice versa). So (5.3.12) gives $\Pr [|X - E[X]| \geq t] \leq 2 \exp\left(-\frac{t^2}{2n}\right)$.

We can improve on this a bit by grouping the W_i together into blocks of length ℓ . If we are given control over a block of ℓ consecutive bits and want to minimize the number of runs, we can either (a) make all the bits zero, causing no runs to appear within the block and preventing adjacent runs from extending to length k using bits from the block, or (b) make all the bits one, possibly creating a new run but possibly also causing two existing runs on either side of the block to merge into one. In the first case, changing all the bits to one except for a zero after every k consecutive ones creates at most $\lfloor \frac{\ell+2k-1}{k+1} \rfloor$ new runs. Treating each of the $\lceil n/\ell \rceil \ell$ blocks as a single variable then gives $\Pr [|X - E[X]| \geq t] \leq 2 \exp\left(-\frac{t^2}{2\lceil n/\ell \rceil (\lfloor (\ell+2k-1)/(k+1) \rfloor)^2}\right)$. Staring at plots of the denominator for a while suggests that it is minimized at $\ell = k + 3$, the largest value with $\lfloor (\ell + 2k - 1)/(k + 1) \rfloor \leq 2$. This gives $\Pr [|X - E[X]| \geq t] \leq 2 \exp\left(-\frac{t^2}{8\lceil n/(k+3) \rceil}\right)$, improving the bound on t from $\Theta(\sqrt{n \log(1/\epsilon)})$ to $\Theta(\sqrt{(n/k) \log(1/\epsilon)})$.

For large k , the expectation of any individual X_k becomes small, so we’d expect that Chernoff bounds would work better on the upper bound side than the method of bounded differences. Unfortunately, we don’t have independence. But from Problem F.2.2, we know that the usual Chernoff bound works as long as we can show $E[X_i | X_1, \dots, X_{i-1}] \leq p_i$ for some sequence of fixed bounds p_i .

For X_1 , there are no previous X_i , and we have $E[X_1] = 2^{-k}$ exactly.

For X_i with $i > 1$, fix X_1, \dots, X_{i-1} ; that is, condition on the event $X_j = x_j$ for all $j < i$ with some fixed sequence x_1, \dots, x_{i-1} . Let’s call this event A . Depending on the particular values of the x_j , it’s not clear how conditioning on A will affect X_i ; but we can split on the value of W_{i-1} to show that either it has no effect or $X_i = 0$:

$$\begin{aligned} E[X_i | A] &= E[X_i | A, W_{i-1} = 0] \Pr[W_{i-1} = 0 | A] + E[X_i | A, W_{i-1} = 1] \Pr[W_{i-1} = 1 | A] \\ &\leq 2^{-k} \Pr[W_{i-1} = 0 | A] + 0 \\ &\leq 2^{-k}. \end{aligned}$$

So we have $p_i \leq 2^{-k}$ for all $1 \leq i \leq n - k + 1$. This gives $\mu = 2^{-k}(n - k + 1)$, and $\Pr[X \geq (1 + \delta)\mu] \leq \left(\frac{e^\delta}{(1+\delta)^{1+\delta}}\right)^\mu$.

If we want a two-sided bound, we can set $\delta = 1$ (since X can't drop below 0 anyway, and get $\Pr[|X - \mathbb{E}[X]| > 2^{-k}(n - k + 1)] \leq \left(\frac{e}{4}\right)^{2^{-k}(n-k+1)}$. This is exponentially small for $k = o(\lg n)$. If k is much bigger than $\lg n$, then we have $\mathbb{E}[X] \ll 1$, so Markov's inequality alone gives us a strong concentration bound.

However, in both cases, the bounds are competitive with the previous bounds from McDiarmid's inequality only if $\mathbb{E}[X] = O(\sqrt{n \log(1/\epsilon)})$. So McDiarmid's inequality wins for $k = o(\log n)$, Markov's inequality wins for $k = \omega(\log n)$, and Chernoff bounds may be useful for a small interval in the middle.

F.3 Assignment 3: due Wednesday, 2011-02-23, at 17:00

F.3.1 Longest common subsequence

A **common subsequence** of two sequences v and w is a sequence u of length k such that there exist indices $i_1 < i_2 < \dots < i_k$ and $j_1 < j_2 < \dots < j_k$ with $u_\ell = v_{i_\ell} = w_{j_\ell}$ for all ℓ . For example, `ardab` is a common subsequence of `abracadabra` and `cardtable`.

Let v and w be words of length n over an alphabet of size n drawn independently and uniformly at random. Give the best upper bound you can on the expected length of the longest common subsequence of v and w .

Solution

Let's count the expectation of the number X_k of common subsequences of length k . We have $\binom{n}{k}$ choices of positions in v , and $\binom{n}{k}$ choices of positions in w ; for each such choices, there is a probability of exactly n^{-k} that the

corresponding positions match. This gives

$$\begin{aligned} \mathbb{E}[X_k] &= \binom{n}{k} n^{-k} \\ &< \frac{n^k}{(k!)^2} \\ &< \frac{n^k}{(k/e)^{2k}} \\ &= \left(\frac{ne^2}{k^2}\right)^k. \end{aligned}$$

We'd like this bound to be substantially less than 1. We can't reasonably expect this to happen unless the base of the exponent is less than 1, so we need $k > e\sqrt{n}$.

If $k = (1 + \epsilon)e\sqrt{n}$ for any $\epsilon > 0$, then $\mathbb{E}[X_k] < (1 + \epsilon)^{-2e\sqrt{n}} < \frac{1}{n}$ for sufficiently large n . It follows that the expected length of the longest common subsequence is at most $(1 + \epsilon)e\sqrt{n}$ for sufficiently large n (because if there are no length- k subsequences, the longest subsequence has length at most $k - 1$, and if there is at least one, the longest has length at most n ; this gives a bound of at most $(1 - 1/n)(k - 1) + (1/n)n < k$). So in general we have the length of the longest common subsequence is at most $(1 + o(1))e\sqrt{n}$.

Though it is not required by the problem, here is a quick argument that the expected length of the longest common subsequence is $\Omega(\sqrt{n})$, based on the **Erdős-Szekeres theorem** [ES35].¹ The Erdős-Szekeres theorem says that any permutation of $n^2 + 1$ elements contains either an increasing sequence of $n + 1$ elements or a decreasing sequence of $n + 1$ elements. Given two random sequences of length n , let S be the set of all elements that appear in both, and consider two permutations ρ and σ of S corresponding to the order in which the elements appear in v and w , respectively (if an element appears multiple times, pick one of the occurrences at random). Then the Erdős-Szekeres theorem says that ρ contains a sequence of length at least $\lfloor \sqrt{|\rho| - 1} \rfloor$ that is either increasing or decreasing with respect to the order given by σ ; by symmetry, the probability that it is increasing is at least $1/2$. This gives an expected value for the longest common subsequence that is at least $\mathbb{E} \left[\sqrt{|\rho| - 1} \right] / 2$.

Let $X = |\rho|$. We can compute a lower bound $\mathbb{E}[X]$ easily; each possible element fails to occur in v with probability $(1 - 1/n)^n \leq e^{-1}$, and similarly for w .

¹As suggested by Benjamin Kunsberg.

So the chance that an element appears in both sequences is at least $(1 - e^{-1})^2$, and thus $E[X] \geq n(1 - e^{-1})^2$. What we want is $E[\sqrt{X - 1}] / 2$; but here the fact that \sqrt{x} is concave means that $E[\sqrt{X - 1}] \geq \sqrt{E[X - 1]}$ by Jensen's inequality (4.3.1). So we have $E[\sqrt{|\rho| - 1}] / 2 \geq \sqrt{n(1 - e^{-1})^2 - 1} / 2 \approx \frac{1 - e^{-1}}{2} \sqrt{n}$.

This is not a very good bound (empirically, the real bound seems to be in the neighborhood of $1.9\sqrt{n}$ when $n = 10000$), but it shows that the upper bound of $(1 + o(1))e\sqrt{n}$ is tight up to constant factors.

F.3.2 A strange error-correcting code

Let Σ be an alphabet of size $m + 1$ that includes m non-blank symbols and a special blank symbol. Let S be a set of $\binom{n}{k}$ strings of length n with non-blank symbols in exactly k positions each, such that no two strings in S have non-blank symbols in the same k positions.

For what value of m can you show S exists such that no two strings in S have the same non-blank symbols in $k - 1$ positions?

Solution

This is a job for the Lovász Local Lemma. And it's even symmetric, so we can use the symmetric version (Corollary 11.3.2).

Suppose we assign the non-blank symbols to each string uniformly and independently at random. For each $A \subseteq S$ with $|A| = k$, let X_A be the string that has non-blank symbols in all positions in A . For each pair of subsets A, B with $|A| = |B| = k$ and $|A \cap B| = k - 1$, let $C_{A,B}$ be the event that X_A and X_B are identical on all positions in $A \cap B$. Then $\Pr[C_{A,B}] = m^{-k+1}$.

We now need to figure out how many events are in each neighborhood $\Gamma(C_{A,B})$. Since $C_{A,B}$ depends only on the choices of values for A and B , it is independent of any events $C_{A',B'}$ where neither of A' or B' is equal to A or B . So we can make $\Gamma(C_{A,B})$ consist of all events $C_{A,B'}$ and $C_{A',B}$ where $B' \neq B$ and $A' \neq A$.

For each fixed A , there are exactly $(n - k)k$ events B that overlap it in $k - 1$ places, because we can specify B by choosing the elements in $B \setminus A$ and $A \setminus B$. This gives $(n - k)k - 1$ events $C_{A,B'}$ where $B' \neq B$. Applying the same argument for A' gives a total of $d = 2(n - k)k - 2$ events in $\Gamma(C_{A,B})$. Corollary 11.3.2 applies if $ep(d + 1) \leq 1$, which in this case means

$em^{-(k-1)}(2(n-k)k-1) \leq 1$. Solving for m gives

$$m \geq (2e(n-k)k-1)^{1/(k-1)}. \quad (\text{F.3.1})$$

For $k \ll n$, the $(n-k)^{1/(k-1)} \approx n^{1/(k-1)}$ term dominates the shape of the right-hand side asymptotically as k gets large, since everything else goes to 1. This suggests we need $k = \Omega(\log n)$ to get m down to a constant.

Note that (F.3.1) doesn't work very well when $k = 1$.² For the $k = 1$ case, there is no overlap in non-blank positions between different strings, so $m = 1$ is enough.

F.3.3 A multiway cut

Given a graph $G = (V, E)$, a **3-way cut** is a set of edges whose endpoints lie in different parts of a partition of the vertices V into three disjoint parts $S \cup T \cup U = V$.

1. Show that any graph with m edges has a 3-way cut with at least $2m/3$ edges.
2. Give an efficient deterministic algorithm for finding such a cut.

Solution

1. Assign each vertex independently to S , T , or U with probability $1/3$ each. Then the probability that any edge uv is contained in the cut is exactly $2/3$. Summing over all edges gives an expected $2m/3$ edges.
2. We'll derandomize the random vertex assignment using the method of conditional probabilities. Given a partial assignment of the vertices, we can compute the conditional expectation of the size of the cut assuming all other vertices are assigned randomly: each edge with matching assigned endpoints contributes 0 to the total, each edge with non-matching assigned endpoints contributes 1, and each edge with zero or one assigned endpoints contributes $2/3$. We'll pick values for the vertices in some arbitrary order to maximize this conditional expectation (since our goal is to get a large cut). At each step, we need only consider the effect on edges incident to the vertex we are assigning whose other endpoints are already assigned, because the contribution of any other edge is not changed by the assignment. Then maximizing

²Thanks to Brad Hayes for pointing this out.

the conditional probability is done by choosing an assignment that matches the assignment of the fewest previously-assigned neighbors: in other words, the natural greedy algorithm works. The cost of this algorithm is $O(n + m)$, since we loop over all vertices and have to check each edge at most once for each of its endpoints.

F.4 Assignment 4: due Wednesday, 2011-03-23, at 17:00

F.4.1 Sometimes successful betting strategies are possible

You enter a casino with $X_0 = a$ dollars, and leave if you reach 0 dollars or b or more dollars, where $a, b \in \mathbb{N}$. The casino is unusual in that it offers arbitrary fair games subject to the requirements that:

- Any payoff resulting from a bet must be a nonzero integer in the range $-X_t$ to X_t , inclusive, where X_t is your current wealth.
- The expected payoff must be exactly 0. (In other words, your assets X_t should form a martingale sequence.)

For example, if you have 2 dollars, you may make a bet that pays off -2 with probability $2/5$, $+1$ with probability $2/5$ and $+2$ with probability $1/5$; but you may not make a bet that pays off -3 , $+3/2$, or $+4$ under any circumstances, or a bet that pays off -1 with probability $2/3$ and $+1$ with probability $1/3$.

1. What strategy should you use to maximize your chances of leaving with at least b dollars?
2. What strategy should you use to maximize your chances of leaving with nothing?
3. What strategy should you use to maximize the number of bets you make before leaving?

Solution

1. Let X_t be your wealth at time t , and let τ be the stopping time when you leave. Because $\{X_t\}$ is a martingale, $E[X_0] = a = E[X_\tau] = \Pr[X_\tau \geq b]E[X_\tau | X_\tau \geq b]$. So $\Pr[X_\tau \geq b]$ is maximized by making $E[X_\tau | X_\tau \geq b]$ as small as possible. It can't be any smaller than b ,

which can be obtained exactly by making only ± 1 bets. This gives a probability of leaving with b of exactly a/b .

2. Here our goal is to minimize $\Pr[X_\tau \geq b]$, so we want to make $E[X_\tau | X_\tau \geq b]$ as large as possible. The largest value of X_τ we can possibly reach is $2(b - 1)$; we can obtain this value by betting ± 1 until we reach $b - 1$, then making any fair bet with positive payoff $b - 1$ (for example, $\pm(b - 1)$ with equal probability works, as does a bet that pays off $b - 1$ with probability $1/b$ and -1 with probability $(b - 1)/b$). In this case we get a probability of leaving with 0 of $1 - \frac{a}{2(b-1)}$.
3. For each t , let $\delta_t = X_t - X_{t-1}$ and $V_t = \text{Var}[\delta_t | \mathcal{F}_t]$. We have previously shown (see the footnote to §8.5.1) that $E[X_\tau^2] = E[X_0^2] + E[\sum_{i=1}^\tau V_i]$ where τ is an appropriate stopping time. When we stop, we know that $X_\tau^2 \leq (2(b - 1))^2$, which puts an upper bound on $E[\sum_{i=1}^\tau V_i]$. We can spend this bound most parsimoniously by minimizing V_i as much as possible. If we make each $\delta_t = \pm 1$, we get the smallest possible value for V_t (since any change contributes at least 1 to the variance). However, in this case we don't get all the way out to $2(b - 1)$ at the high end; instead, we stop at b , giving an expected number of steps equal to $a(b - a)$.

We can do a bit better than this by changing our strategy at $b - 1$. Instead of betting ± 1 , let's pick some x and place a bet that pays off $b - 1$ with probability $\frac{1}{b}$ and -1 with probability $\frac{b-1}{b} = 1 - \frac{1}{b}$. (The idea here is to minimize the conditional variance while still allowing ourselves to reach $2(b - 1)$.) Each ordinary random walk step has $V_t = 1$; a "big" bet starting at $b - 1$ has $V_t = 1 - \frac{1}{b} + \frac{(b-1)^2}{b} = \frac{b-1+b^2-2b+1}{b} = b - \frac{1}{b}$.

To analyze this process, observe that starting from a , we first spending $a(b - a - 1)$ steps on average to reach either 0 (with probability $1 - \frac{a}{b-1}$ or $b - 1$ (with probability $\frac{a}{b-1}$). In the first case, we are done. Otherwise, we take one more step, then with probability $\frac{1}{b}$ we lose and with probability $\frac{b-1}{b}$ we continue starting from $b - 2$. We can write a recurrence for our expected number of steps $T(a)$ starting from a , as:

$$T(a) = a(b - a - 1) + \frac{a}{b - 1} \left(1 + \frac{b - 1}{b} T(b - 2) \right). \quad (\text{F.4.1})$$

When $a = b - 2$, we get

$$\begin{aligned} T(b-2) &= (b-2) + \frac{b-2}{b-1} \left(1 + \frac{b-1}{b} T(b-2) \right) \\ &= (b-2) \left(1 + \frac{1}{b-1} \right) + \frac{b-2}{b} T(b-2), \end{aligned}$$

which gives

$$\begin{aligned} T(b-2) &= \frac{(b-2)^{\frac{2b-1}{b-1}}}{2/b} \\ &= \frac{b(b-2)(2b-1)}{2(b-1)}. \end{aligned}$$

Plugging this back into (F.4.1) gives

$$\begin{aligned} T(a) &= a(b-a-1) + \frac{a}{b-1} \left(1 + \frac{b-1}{b} \frac{b(b-2)(2b-1)}{2(b-1)} \right) \\ &= ab - a^2 + a + \frac{a}{b-1} + \frac{a(b-2)(2b-1)}{2(b-1)} \\ &= \frac{3}{2}ab + O(b). \end{aligned} \tag{F.4.2}$$

This is much better than the $a(b-a)$ value for the straight ± 1 strategy, especially when a is also large.

I don't know if this particular strategy is in fact optimal, but that's what I'd be tempted to bet.

F.4.2 Random walk with reset

Consider a random walk on \mathbb{N} that goes up with probability $1/2$, down with probability $3/8$, and resets to 0 with probability $1/8$. When $X_t > 0$, this gives:

$$X_{t+1} = \begin{cases} X_t + 1 & \text{with probability } 1/2, \\ X_t - 1 & \text{with probability } 3/8, \text{ and} \\ 0 & \text{with probability } 1/8. \end{cases}$$

When $X_t = 0$, we let $X_{t+1} = 1$ with probability $1/2$ and 0 with probability $1/2$.

1. What is the stationary distribution of this process?

2. What is the mean recurrence time μ_n for some state n ?
3. Use μ_n to get a tight asymptotic (i.e., big- Θ) bound on $\mu_{0,n}$, the expected time to reach n starting from 0.

Solution

1. For $n > 0$, we have $\pi_n = \frac{1}{2}\pi_{n-1} + \frac{3}{8}\pi_{n+1}$, with a base case $\pi_0 = \frac{1}{8} + \frac{3}{8}\pi_0 + \frac{3}{8}\pi_1$.

The π_n expression is a linear homogeneous recurrence, so its solution consists of linear combinations of terms b^n , where b satisfies $1 = \frac{1}{2}b^{-1} + \frac{3}{8}b$. The solutions to this equation are $b = 2/3$ and $b = 2$; we can exclude the $b = 2$ case because it would make our probabilities blow up for large n . So we can reasonably guess $\pi_n = a(2/3)^n$ when $n > 0$.

For $n = 0$, substitute $\pi_0 = \frac{1}{8} + \frac{3}{8}\pi_0 + \frac{3}{8}a(2/3)$ to get $\pi_0 = \frac{1}{5} + \frac{2}{5}a$. Now substitute

$$\begin{aligned} \pi_1 &= (2/3)a \\ &= \frac{1}{2}\pi_0 + \frac{3}{8}a(2/3)^2 \\ &= \frac{1}{2}\left(\frac{1}{5} + \frac{2}{5}a\right) + \frac{3}{8}a(2/3)^2 \\ &= \frac{1}{10} + \frac{11}{30}a, \end{aligned}$$

which we can solve to get $a = 1/3$.

So our candidate π is $\pi_0 = 1/3$, $\pi_n = (1/3)(2/3)^n$, and in fact we can drop the special case for π_0 .

As a check, $\sum_{i=0}^n \pi_n = (1/3) \sum_{i=0}^n (2/3)^n = \frac{1/3}{1-2/3} = 1$.

2. Since $\mu_n = 1/\pi_n$, we have $\mu_n = 3(3/2)^n$.
3. In general, let $\mu_{k,n}$ be the expected time to reach n starting at k . Then $\mu_n = \mu_{n,n} = 1 + \frac{1}{8}\mu_{0,n} + \frac{1}{2}\mu_{n+1,n} + \frac{3}{8}\mu_{n-1,n} \geq 1 + \mu_{0,n}/8$. It follows that $\mu_{0,n} \leq 8\mu_n + 1 = 24(3/2)^n + 1 = O((3/2)^n)$.

For the lower bound, observe that $\mu_n \leq \mu_{n,0} + \mu_{0,n}$. Since there is a $1/8$ chance of reaching 0 from any state, we have $\mu_{n,0} \leq 8$. It follows that $\mu_n \leq 8 + \mu_{0,n}$ or $\mu_{0,n} \geq \mu_n - 8 = \Omega((3/2)^n)$.

F.4.3 Yet another shuffling algorithm

Suppose we attempt to shuffle a deck of n cards by picking a card uniformly at random, and swapping it with the top card. Give the best bound you can on the mixing time for this process to reach a total variation distance of ϵ from the uniform distribution.

Solution

It's tempting to use the same coupling as for move-to-top (see §9.4.3). This would be that at each step we choose the same card to swap to the top position, which increases by at least one the number of cards that are in the same position in both decks. The problem is that at the next step, these two cards are most likely separated again, by being swapped with other cards in two different positions.

Instead, we will do something slightly more clever. Let Z_t be the number of cards in the same position at time t . If the top cards of both decks are equal, we swap both to the same position chosen uniformly at random. This has no effect on Z_t . If the top cards of both decks are not equal, we pick a card uniformly at random and swap it to the top in both decks. This increases Z_t by at least 1, unless we happen to pick cards that are already in the same position; so Z_t increases by at least 1 with probability $1 - Z_t/n$.

Let's summarize a state by an ordered pair (k, b) where $k = Z_t$ and b is 0 if the top cards are equal and 1 if they are not equal. Then we have a Markov chain where $(k, 0)$ goes to $(k, 1)$ with probability $\frac{n-k}{n}$ (and otherwise stays put); and $(k, 1)$ goes to $(k+1, 0)$ (or higher) with probability $\frac{n-k}{n}$ and to $(k, 0)$ with probability $\frac{k}{n}$.

Starting from $(k, 0)$, we expect to wait $\frac{n}{n-k}$ steps on average to reach $(k, 1)$, at which point we move to $(k+1, 0)$ or back to $(k, 0)$ in one more step; we iterate through this process $\frac{n}{n-k}$ times on average before we are successful. This gives an expected number of steps to get from $(k, 0)$ to $(k+1, 0)$ (or possibly a higher value) of $\frac{n}{n-k} \left(\frac{n}{n-k} + 1 \right)$. Summing over k up to $n-2$ (since once $k > n-2$, we will in fact have $k = n$, since k can't be $n-1$), we

get

$$\begin{aligned} \mathbb{E}[\tau] &\leq \sum_{k=0}^{n-2} \frac{n}{n-k} \left(\frac{n}{n-k} + 1 \right) \\ &= \sum_{m=2}^n \left(\frac{n^2}{m^2} + \frac{n}{m} \right) \\ &\leq n^2 \left(\frac{\pi^2}{6} - 1 \right) + n \ln n. \\ &= O(n^2). \end{aligned}$$

So we expect the deck to mix in $O(n^2 \log(1/\epsilon))$ steps. (I don't know if this is the real bound; my guess is that it should be closer to $O(n \log n)$ as in all the other shuffling procedures.)

F.5 Assignment 5: due Thursday, 2011-04-07, at 23:59

F.5.1 A reversible chain

Consider a random walk on \mathbb{Z}_m , where $p_{i,i+1} = 2/3$ for all i and $p_{i,i-1} = 1/3$ for all i except $i = 0$. Is it possible to assign values to $p_{0,m-1}$ and $p_{0,0}$ to make this chain reversible, and if so, what stationary distribution do you get?

Solution

Suppose we can make this chain reversible, and let π be the resulting stationary distribution. From the detailed balance equations, we have $(2/3)\pi_i = (1/3)\pi_{i+1}$ or $\pi_{i+1} = 2\pi_i$ for $i = 0 \dots m-2$. The solution to this recurrence is $\pi_i = 2^i \pi_0$, which gives $\pi_i = \frac{2^i}{2^m - 1}$ when we set π_0 to get $\sum_i \pi_i = 1$.

Now solve $\pi_0 p_{0,m-1} = \pi_{m-1} p_{m-1,0}$ to get

$$\begin{aligned} p_{0,m-1} &= \frac{\pi_{m-1} p_{m-1,0}}{\pi_0} \\ &= 2^{m-1} (2/3) \\ &= 2^m / 3. \end{aligned}$$

This is greater than 1 for $m > 1$, so except for the degenerate cases of $m = 1$ and $m = 2$, it's not possible to make the chain reversible.

F.5.2 Toggling bits

Consider the following Markov chain on an array of n bits $a[1], a[2], \dots, a[n]$. At each step, we choose a position i uniformly at random. We then change $A[i]$ to $\neg A[i]$ with probability $1/2$, provided $i = 1$ or $A[i - 1] = 1$ (if neither condition holds, do nothing).³

1. What is the stationary distribution?
2. How quickly does it converge?

Solution

1. First let's show irreducibility. Starting from an arbitrary configuration, repeatedly switch the leftmost 0 to a 1 (this is always permitted by the transition rules); after at most n steps, we reach the all-1 configuration. Since we can repeat this process in reverse to get to any other configuration, we get that every configuration is reachable from every other configuration in at most $2n$ steps ($2n - 1$ if we are careful about handling the all-0 configuration separately).

We also have that for any two adjacent configurations x and y , $p_{xy} = p_{yx} = \frac{1}{2n}$. So we have a reversible, irreducible, aperiodic (because there exists at least one self-loop) chain with a uniform stationary distribution $\pi_x = 2^{-n}$.

2. Here is a bound using the obvious coupling, where we choose the same position in X and Y and attempt to set it to the same value. To show this coalesces, given X_t and Y_t define Z_t to be the position of the rightmost 1 in the common prefix of X_t and Y_t , or 0 if there is no 1 in the common prefix of X_t and Y_t . Then Z_t increases by at least 1 if we attempt to set position $Z_t + 1$ to 1, which occurs with probability $\frac{1}{2n}$, and decreases by at most 1 if we attempt to set Z_t to 0, again with probability $\frac{1}{2n}$.

It follows that Z_t reaches n no later than a ± 1 random walk on $0 \dots n$ with reflecting barriers that takes a step every $1/n$ time units on average. The expected number of steps to reach n from the worst-case

³Motivation: Imagine each bit represents whether a node in some distributed system is inactive (0) or active (1), and you can only change your state if you have an active left neighbor to notify. Also imagine that there is an always-active *base station* at -1 (alternatively, imagine that this assumption makes the problem easier than the other natural arrangement where we put all the nodes in a ring).

starting position of 0 is exactly n^2 . (Proof: model the random walk with a reflecting barrier at 0 by folding a random walk with absorbing barriers at $\pm n$ in half, then use the bound from §8.5.1.) We must then multiply this by n to get an expected n^3 steps in the original process. So the two copies coalesce in at most n^3 expected steps. My suspicion is one could improve this bound with a better analysis by using the bias toward increasing Z_t to get the expected time to coalesce down to $O(n^2)$, but I don't know any clean way to do this.

The path coupling version of this is that we look at two adjacent configurations X_t and Y_t , use the obvious coupling again, and see what happens to $E[d(X_{t+1}, Y_{t+1}) | X_t, Y_t]$, where the distance is the number of transitions needed to convert X_t to Y_t or vice versa. If we pick the position i where X_t and Y_t differ, then we coalesce; this occurs with probability $1/n$. If we change the 1 to the left of i to a 0, then $d(X_{t+1}, Y_{t+1})$ rises to 3 (because to get from X_{t+1} to Y_{t+1} , we have to change position $i - 1$ to 1, change position i , and then change position $i - 1$ back to 0); this occurs with probability $1/2n$ if $i > 1$. But we can also get into trouble if we try to change position $i + 1$; we can only make the change in one of X_t and Y_t , so we get $d(X_{t+1}, Y_{t+1}) = 2$ in this case, which occurs with probability $1/2n$ when $i < n$. Adding up all three cases gives a worst-case expected change of $-1/n + 2/2n + 1/2n = 1/2n > 0$. So unless we can do something more clever, path coupling won't help us here.

However, it is possible to get a bound using canonical paths, but the best bound I could get was not as good as the coupling bound. The basic idea is that we will change x into y one bit at a time (from left to right, say), so that we will go through a sequence of intermediate states of the form $y[1]y[2] \dots y[i]x[i+1]x[i+2] \dots x[n]$. But to change $x[i+1]$ to $y[i+1]$, we may also need to reach out with a tentacle of 1 bits from from the last 1 in the current prefix of y (and then retract it afterwards). Given a particular transition where we change a 0 to a 1, we can reconstruct the original x and y by specifying (a) which bit i at or after our current position we are trying to change; (b) which 1 bit before our current position is the last "real" 1 bit in y as opposed to something we are creating to reach out to position i ; and (c) the values of $x[1] \dots x[i-1]$ and $y[i+1] \dots y[i]$. A similar argument applies to $1 \rightarrow 0$ transitions. So we are routing at most $n^2 2^{n-1}$ paths across each

transition, giving a bound on the congestion

$$\begin{aligned}\rho &\leq \left(\frac{1}{2^{-n}/2n}\right) n^2 2^{n-1} 2^{-2n} \\ &= n^3.\end{aligned}$$

The bound on τ_2 that follows from this is $8n^6$, which is pretty bad (although the constant could be improved by counting the (a) and (b) bits more carefully). As with the coupling argument, it may be that there is a less congested set of canonical paths that gives a better bound.

F.5.3 Spanning trees

Suppose you have a connected graph $G = (V, E)$ with n nodes and m edges. Consider the following Markov process. Each state H_t is a subgraph of G that is either a spanning tree or a spanning tree plus an additional edge. At each step, flip a fair coin. If it comes up heads, choose an edge e uniformly at random from E and let $H_{t+1} = H_t \cup \{e\}$ if H_t is a spanning tree and let $H_{t+1} = H_t \setminus \{e\}$ if H_t is not a spanning tree and $H_t \setminus \{e\}$ is connected. If it comes up tails and H_t is a spanning tree, let H_{t+1} be some other spanning tree, sampled uniformly at random. In all other cases, let $H_{t+1} = H_t$.

Let N be the number of states in this Markov chain.

1. What is the stationary distribution?
2. How quickly does it converge?

Solution

1. Since every transition has a matching reverse transition with the same transition probability, the chain is reversible with a uniform stationary distribution $\pi_H = 1/N$.
2. Here's a coupling that coalesces in at most $4m/3 + 2$ expected steps:
 - (a) If X_t and Y_t are both trees, then send them to the same tree with probability $1/2$; else let them both add edges independently (or we could have them add the same edge—it doesn't make any difference to the final result).
 - (b) If only one of X_t and Y_t is a tree, with probability $1/2$ scramble the tree while attempting to remove an edge from the non-tree,

and the rest of the time scramble the non-tree (which has no effect) while attempting to add an edge to the tree. Since the non-tree has at least three edges that can be removed, this puts $(X_{t+1}, Y_{t+1}$ in the two-tree case with probability at least $3/2m$.

- (c) If neither X_t nor Y_t is a tree, attempt to remove an edge from both. Let S and T be the sets of edges that we can remove from X_t and Y_t , respectively, and let $k = \min(|S|, |T|) \geq 3$. Choose k edges from each of S and T and match them, so that if we remove one edge from each pair, we also remove the other edge. As in the previous case, this puts $(X_{t+1}, Y_{t+1}$ in the two-tree case with probability at least $3/2m$.

To show this coalesces, starting from an arbitrary state, we reach a two-tree state in at most $2m/3$ expected steps. After one more step, we either coalesce (with probability $1/2$) or restart from a new arbitrary state. This gives an expected coupling time of at most $2(2m/3 + 1) = 4m/3 + 2$ as claimed.

F.6 Assignment 6: due Monday, 2011-04-25, at 17:00

F.6.1 Sparse satisfying assignments to DNFs

Given a formula in disjunctive normal form, we'd like to estimate the number of satisfying assignments in which exactly w of the variables are true. Give a fully polynomial-time randomized approximation scheme for this problem.

Solution

Essentially, we're going to do the Karp-Luby covering trick [KL85] described in §10.3, but will tweak the probability distribution when we generate our samples so that we only get samples with weight w .

Let U be the set of assignment with weight w (there are exactly $\binom{n}{w}$ such assignments, where n is the number of variables). For each clause C_i , let $U_i = \{x \in U \mid C_i(x) = 1\}$. Now observe that:

1. We can compute $|U_i|$. Let $k_i = |C_i|$ be the number of variables in C_i and $k_i^+ = |C_i^+|$ the number of variables that appear in positive form in C_i . Then $|U_i| = \binom{n-k_i}{w-k_i^+}$ is the number of ways to make a total of w variables true using the remaining $n - k_i$ variables.

2. We can sample uniformly from U_i , by sampling a set of $w - k_i^+$ true variables not in C_i uniformly from all variables not in C_i .
3. We can use the values computed for $|U_i|$ to sample i proportionally to the size of $|U_i|$.

So now we sample pairs (i, x) with $x \in U_i$ uniformly at random by sampling i first, then sampling $x \in U_i$. As in the original algorithm, we then count (i, x) if and only if C_i is the leftmost clause for which $C_i(x) = 1$. The same argument that at least $1/m$ of the (i, x) pairs count applies, and so we get the same bounds as in the original algorithm.

F.6.2 Detecting duplicates

Algorithm F.1 attempts to detect duplicate values in an input array S of length n .

```

1 Initialize  $A[1 \dots n]$  to  $\perp$ 
2 Choose a hash function  $h$ 
3 for  $i \leftarrow 1 \dots n$  do
4    $x \leftarrow S[i]$ 
5   if  $A[h(x)] = x$  then
6     return true
7   else
8      $A[h(x)] \leftarrow x$ 
9 return false

```

Algorithm F.1: Dubious duplicate detector

It's easy to see that Algorithm F.1 never returns **true** unless some value appears twice in S . But maybe it misses some duplicates it should find.

1. Suppose h is a random function. What is the worst-case probability that Algorithm F.1 returns **false** if S contains two copies of some value?
2. Is this worst-case probability affected if h is drawn instead from a 2-universal family of hash functions?

Solution

1. Suppose that $S[i] = S[j] = x$ for $i < j$. Then the algorithm will see x in $A[h(x)]$ on iteration j and return **true**, unless it is overwritten by some

value $S[k]$ with $i < k < j$. This occurs if $h(S[k]) = h(x)$, which occurs with probability exactly $1 - (1 - 1/n)^{j-i-1}$ if we consider all possible k . This quantity is maximized at $1 - (1 - 1/n)^{n-2} \approx 1 - (1 - 1/n)^2/e \approx 1 - (1 - 1/2n)/e$ when $i = 1$ and $j = n$.

- As it happens, the algorithm can fail pretty badly if all we know is that h is 2-universal. What we can show is that the probability that some $S[k]$ with $i < j < k$ gets hashed to the same place as $x = S[i] = S[j]$ in the analysis above is at most $(j - i - 1)/n \leq (n - 2)/n = (1 - 2/n)$, since each $S[k]$ has at most a $1/n$ chance of colliding with x and the union bound applies. But it is possible to construct a 2-universal family for which we get exactly this probability in the worst case.

Let $U = \{0 \dots n\}$, and define for each a in $\{0 \dots n - 1\}$ $h_a : U \rightarrow n$ by $h_a(n) = 0$ and $h_a(x) = (x + a) \bmod n$ for $x \neq n$. Then $H = \{h_a\}$ is 2-universal, since if $x \neq y$ and neither x nor y is n , $\Pr[h_a(x) = h_a(y)] = 0$, and if one of x or y is n , $\Pr[h_a(x) = h_a(y)] = 1/n$. But if we use this family in Algorithm F.1 with $S[1] = S[n] = n$ and $S[k] = k$ for $1 < k < n$, then there are $n - 2$ choices of a that put one of the middle values in $A[0]$.

F.6.3 Balanced Bloom filters

A clever algorithmist decides to solve the problem of Bloom filters filling up with ones by capping the number of ones at $m/2$. As in a standard Bloom filter, an element is inserted by writing ones to $A[h_1(x)], A[h_2(x)], \dots, A[h_k(x)]$; but after writing each one, if the number of one bits in the bit-vector is more than $m/2$, one of the ones in the vector (chosen uniformly at random) is changed back to a zero.

Because some of the ones associated with a particular element might be deleted, the membership test answers yes if at least $3/4$ of the bits $A[h_1(x)] \dots A[h_k(x)]$ are ones.

To simplify the analysis, you may assume that the h_i are independent random functions. You may also assume that $(3/4)k$ is an integer.

- Give an upper bound on the probability of a false positive when testing for a value x that has never been inserted.
- Suppose that we insert x at some point, and then follow this insertion with a sequence of insertions of new, distinct values y_1, y_2, \dots . Assuming a worst-case state before inserting x , give asymptotic upper and

lower bounds on the expected number of insertions until a test for x fails.

Solution

1. The probability of a false positive is maximized when exactly half the bits in A are one. If x has never been inserted, each $A[h_i(x)]$ is equally likely to be zero or one. So $\Pr[\text{false positive for } x] = \Pr[S_k \geq (3/4)k]$ when S_k is a binomial random variable with parameters $1/2$ and k . Chernoff bounds give

$$\begin{aligned} \Pr[S_k \geq (3/4)k] &= \Pr[S_k \geq (3/2) \mathbb{E}[S_k]] \\ &\leq \left(\frac{e^{1/2}}{(3/2)^{3/2}} \right)^{k/2} \\ &\leq (0.94734)^k. \end{aligned}$$

We can make this less than any fixed ϵ by setting $k \geq 20 \ln(1/\epsilon)$ or thereabouts.

2. For false negatives, we need to look at how quickly the bits for x are eroded away. A minor complication is that the erosion may start even as we are setting $A[h_1(x)] \dots A[h_k(x)]$.

Let's consider a single bit $A[i]$ and look at how it changes after (a) setting $A[i] = 1$, and (b) setting some random $A[r] = 1$.

In the first case, $A[i]$ will be 1 after the assignment unless it is set back to zero, which occurs with probability $\frac{1}{m/2+1}$. This distribution does not depend on the prior value of $A[i]$.

In the second case, if $A[i]$ was previously 0, it becomes 1 with probability

$$\begin{aligned} \frac{1}{m} \left(1 - \frac{1}{m/2+1} \right) &= \frac{1}{m} \cdot \frac{m/2}{m/2+1} \\ &= \frac{1}{m+2}. \end{aligned}$$

If it was previously 1, it becomes 0 with probability

$$\frac{1}{2} \cdot \frac{1}{m/2+1} = \frac{1}{m+2}.$$

So after the initial assignment, $A[i]$ just flips its value with probability $\frac{1}{m+2}$.

It is convenient to represent $A[i]$ as ± 1 ; let $X_i^t = -1$ if $A[i] = 0$ at time t , and 1 otherwise. Then X_i^t satisfies the recurrence

$$\begin{aligned} \mathbb{E} \left[X_i^{t+1} \mid X_i^t \right] &= \frac{m+1}{m+2} X_i^t - \frac{1}{m+2} X_i^t \\ &= \frac{m}{m+2} X_i^t \\ &= \left(1 - \frac{2}{m+2} \right) X_i^t. \end{aligned}$$

We can extend this to $\mathbb{E} \left[X_i^t \mid X_i^0 \right] = \left(1 - \frac{2}{m+2} \right)^t X_i^0 \approx e^{-2t/(m+2)} X_i^0$.

Similarly, after setting $A[i] = 1$, we get $\mathbb{E} \left[X_i \right] = 1 - 2 \frac{1}{m/2+1} = 1 - \frac{4}{2m+1} = 1 - o(1)$.

Let $S^t = \sum_{i=1}^k X_{h_i(x)}^t$. Let 0 be the time at which we finish inserting x . Then each for each i we have

$$1 - o(1)e^{-2k/(m+2)} \leq \mathbb{E} \left[X_{h_i(x)}^0 \right] \leq 1 - o(1),$$

from which it follows that

$$k(1 - o(1))e^{-2k/(m+2)} \leq \mathbb{E} \left[S^0 \right] \leq 1 - o(1)$$

and in general that

$$k(1 - o(1))e^{-2(k+t)/(m+2)} \leq \mathbb{E} \left[S^t \right] \leq 1 - o(1)e^{-2t/(m+2)}.$$

So for any fixed $0 < \epsilon < 1$ and sufficiently large m , we will have $\mathbb{E} \left[S^t \right] = \epsilon k$ for some t' where $t \leq t' \leq k + t$ and $t = \Theta(m \ln(1/\epsilon))$.

We are now looking for the time at which S^t drops below $k/2$ (the $k/2$ is because we are working with ± 1 variables). We will bound when this time occurs using Markov's inequality.

Let's look for the largest time t with $\mathbb{E} \left[S^t \right] \geq (3/4)k$. Then $\mathbb{E} \left[k - S^t \right] \leq k/4$ and $\Pr \left[k - S^t \geq k/2 \right] \leq 1/2$. It follows that after $\Theta(m) - k$ operations, x is still visible with probability $1/2$, which implies that the expected time at which it stops being visible is at least $(\Omega(m) - k)/2$. To get the expected number of insert operations, we divide by k , to get $\Omega(m/k)$.

For the upper bound, apply the same reasoning to the first time at which $E[S^t] \leq k/4$. This occurs at time $O(m)$ at the latest (with a different constant), so after $O(m)$ steps there is at most a $1/2$ probability that $S^t \geq k/2$. If S^t is still greater than $k/2$ at this point, try again using the same analysis; this gives us the usual geometric series argument that $E[t] = O(m)$. Again, we have to divide by k to get the number of insert operations, so we get $O(m/k)$ in this case.

Combining these bounds, we have that x disappears after $\Theta(m/k)$ insertions on average. This seems like about what we would expect.

F.7 Final exam

Write your answers in the blue book(s). Justify your answers. Work alone. Do not use any notes or books.

There are four problems on this exam, each worth 20 points, for a total of 80 points. You have approximately three hours to complete this exam.

F.7.1 Leader election

Suppose we have n processes and we want to elect a leader. At each round, each process flips a coin, and drops out if the coin comes up tails. We win if in some round there is exactly one process left.

Let $T(n)$ be the probability that this event eventually occurs starting with n processes. For small n , we have $T(0) = 0$ and $T(1) = 1$. Show that there is a constant $c > 0$ such that $T(n) \geq c$ for all $n > 1$.

Solution

Let's suppose that there is some such c . We will necessarily have $c \leq 1 = T(1)$, so the induction hypothesis will hold in the base case $n = 1$.

For $n \geq 2$, compute

$$\begin{aligned} T(n) &= \sum_{k=0}^n 2^{-n} \binom{n}{k} T(k) \\ &= 2^{-n} T(n) + 2^{-n} n T(1) + \sum_{k=2}^{n-1} 2^{-n} \binom{n}{k} T(k) \\ &\geq 2^{-n} T(n) + 2^{-n} n + 2^{-n} (2^n - n - 2)c. \end{aligned}$$

Solve for $T(n)$ to get

$$\begin{aligned} T(n) &\geq \frac{n + (2^n - n - 2)c}{2^n - 1} \\ &= c \left(\frac{2^n - n - 2 + n/c}{2^n - 1} \right). \end{aligned}$$

This will be greater than or equal to c if $2^n - n - 2 + n/c \geq 2^n - 1$ or $n/c \geq n + 1$, which holds if $c \leq \frac{n}{n+1}$. The worst case is $n = 2$ giving $c = 2/3$.

Valiant and Vazirani [VV86] used this approach to reduce solving general instances of SAT to solving instances of SAT with unique solutions; they prove essentially the result given above (which shows that fixing variables in a SAT formula is likely to produce a SAT formula with a unique solution at some point) with a slightly worse constant.

F.7.2 Two-coloring an even cycle

Here is a not-very-efficient algorithm for 2-coloring an even cycle. Every node starts out red or blue. At each step, we pick one of the n nodes uniformly at random, and change its color if it has the same color as at least one of its neighbors. We continue until no node has the same color as either of its neighbors.

Suppose that in the initial state there are exactly two monochromatic edges. What is the worst-case expected number of steps until there are no monochromatic edges?

Solution

Suppose we have a monochromatic edge surrounded by non-monochromatic edges, e.g. $RBRBR$. If we pick one of the endpoints of the edge (say the left endpoint in this case), then the monochromatic edge shifts in the direction of that endpoint: $RBRBR$. Picking any node not incident to a monochromatic edge has no effect, so in this case there is no way to increase the number of monochromatic edges.

It may also be that we have two adjacent monochromatic edges: $BRRRB$. Now if we happen to pick the node in the middle, we end up with no monochromatic edges ($BRBR$) and the process terminates. If on the other hand we pick one of the nodes on the outside, then the monochromatic edges move away from each other.

We can thus model the process with 2 monochromatic edges as a random walk, where the difference between the leftmost nodes of the edges (mod

n) increases or decreases with equal probability $2/n$ except if the distance is 1 or -1 ; in this last case, the distance increases (going to 2 or -2) with probability $2/n$, but decreases only with probability $1/n$. We want to know when this process hits 0 (or n).

Imagine a random walk starting from position k with absorbing barriers at 1 and $n - 1$. This reaches 1 or $n - 1$ after $(k - 1)(n - 1 - k)$ steps on average, which translates into $(n/4)(k - 1)(n - 1 - k)$ steps of our original process if we take into account that we only move with probability $4/n$ per time unit. This time is maximized by setting $k = n/2$, which gives $(n/4)(n/2 - 1)^2 = n^3/16 - n^2/4 + n/4$ expected time units to reach 1 or $n - 1$ for the first time.

At 1 or $n - 1$, we wait an addition $n/3$ steps on average; then with probability $1/3$ the process finishes and with probability $2/3$ we start over from position 2 or $n - 2$; in the latter case, we run $(n/4)(n - 3) + n/3$ time units on average before we may finish again. On average, it takes 3 attempts to finish. Each attempt incurs the expected $n/3$ cost before taking a step, and all but the last attempt incur the expected $(n/4)(n - 3)$ additional steps for the random walk. So the last phase of the process the process adds $(1/2)n(n - 3) + n = (1/2)n^2 - (5/4)n$ steps.

Adding up all of the costs gives $n^3/16 - n^2/4 + n/4 + n/3 + (1/2)n^2 - (5/4)n = \frac{1}{16}n^3 + \frac{1}{4}n^2 - \frac{2}{3}n$ steps.

F.7.3 Finding the maximum

```

1 Randomly permute  $A$ 
2  $m \leftarrow -\infty$ 
3 for  $i \leftarrow 1 \dots n$  do
4   if  $A[i] > m$  then
5      $m \leftarrow A[i]$ 
6 return  $m$ 

```

Algorithm F.2: Randomized max-finding algorithm

Suppose that we run Algorithm F.2 on an array with n elements, all of which are distinct. What is the expected number of times Line 5 is executed as a function of n ?

Solution

Let X_i be the indicator variable for the event that Line 5 is executed on the i -th pass through the loop. This will occur if $A[i]$ is greater than $A[j]$ for all $j < i$, which occurs with probability exactly $1/i$ (given that A has been permuted randomly). So the expected number of calls to Line 5 is $\sum_i 1^n \mathbb{E}[X_i] = \sum_{i=1}^n \frac{1}{i} = H_n$.

F.7.4 Random graph coloring

Let G be a random d -regular graph on n vertices, that is, a graph drawn uniformly from the family of all n -vertex graph in which each vertex has exactly d neighbors. Color the vertices of G red or blue independently at random.

1. What is the expected number of monochromatic edges in G ?
2. Show that the actual number of monochromatic edges is tightly concentrated around its expectation.

Solution

The fact that G is itself a random graph is a red herring; all we really need to know is that it's d -regular.

1. Because G has exactly $dn/2$ edges, and each edge has probability $1/2$ of being monochromatic, the expected number of monochromatic edges is $dn/4$.
2. This is a job for Azuma's inequality. Consider the vertex exposure martingale. Changing the color of any one vertex changes the number of monochromatic edges by at most d . So we have $\Pr[|X - \mathbb{E}[X]| \geq t] \leq 2 \exp(-t^2/2 \sum_{i=1}^n d^2) = 2e^{-t^2/2nd^2}$, which tells us that the deviation is likely to be not much more than $O(d\sqrt{n})$.

Appendix G

Sample assignments from Spring 2009

G.1 Final exam, Spring 2009

Write your answers in the blue book(s). Justify your answers. Work alone. Do not use any notes or books.

There are four problems on this exam, each worth 20 points, for a total of 80 points. You have approximately three hours to complete this exam.

G.1.1 Randomized mergesort (20 points)

Consider the following randomized version of the mergesort algorithm. We take an unsorted list of n elements and split it into two lists by flipping an independent fair coin for each element to decide which list to put it in. We then recursively sort the two lists, and merge the resulting sorted lists. The merge procedure involves repeatedly comparing the smallest element in each of the two lists and removing the smaller element found, until one of the lists is empty.

Compute the expected number of comparisons needed to perform this final merge. (You do not need to consider the cost of performing the recursive sorts.)

Solution

Color the elements in the final merged list red or blue based on which sublist they came from. The only elements that do not require a comparison to insert into the main list are those that are followed only by elements of the

same color; the expected number of such elements is equal to the expected length of the longest monochromatic suffix. By symmetry, this is the same as the expected longest monochromatic prefix, which is equal to the expected length of the longest sequence of identical coin-flips.

The probability of getting k identical coin-flips in a row followed by a different coin-flip is exactly 2^{-k} ; the first coin-flip sets the color, the next $k-1$ must follow it (giving a factor of 2^{-k+1} , and the last must be the opposite color (giving an additional factor of 2^{-1}). For n identical coin-flips, there is a probability of 2^{-n+1} , since we don't need an extra coin-flip of the opposite color. So the expected length is $\sum_{k=1}^{n-1} k2^{-k} + n2^{-n+1} = \sum_{k=0}^n k2^{-k} + n2^{-n}$.

We can simplify the sum using generating functions. The sum $\sum_{k=0}^n 2^{-k} z^k$ is given by $\frac{1-(z/2)^{n+1}}{1-z/2}$. Taking the derivative with respect to z gives $\sum_{k=0}^n 2^{-k} k z^{k-1} = (1/2) \frac{1-(z/2)^{n+1} 2}{1-z/2} + (1/2) \frac{(n+1)(z/2)^n}{1-z/2}$. At $z = 1$ this is $2(1 - 2^{-n-1}) - 2(n+1)2^{-n} = 2 - (n+2)2^{-n}$. Adding the second term gives $E[X] = 2 - 2 \cdot 2^{-n} = 2 - 2^{-n+1}$.

Note that this counts the expected number of elements for which we do not have to do a comparison; with n elements total, this leaves $n - 2 + 2^{-n+1}$ comparisons on average.

G.1.2 A search problem (20 points)

Suppose you are searching a space by generating new instances of some problem from old ones. Each instance is either good or bad; if you generate a new instance from a good instance, the new instance is also good, and if you generate a new instance from a bad instance, the new instance is also bad.

Suppose that you start with X_0 good instances and Y_0 bad instances, and that at each step you choose one of the instances you already have uniformly at random to generate a new instance. What is the expected number of good instances you have after n steps?

Hint: Consider the sequence of values $\{X_t/(X_t + Y_t)\}$.

Solution

We can show that the suggested sequence is a martingale, by computing

$$\begin{aligned} \mathbb{E} \left[\frac{X_{t+1}}{X_{t+1} + Y_{t+1}} \mid X_t, Y_t \right] &= \frac{X_t}{X_t + Y_t} \cdot \frac{X_t + 1}{X_t + Y_t + 1} + \frac{Y_t}{X_t + Y_t} \cdot \frac{X_t}{X_t + Y_t + 1} \\ &= \frac{X_t(X_t + 1) + Y_t X_t}{(X_t + Y_t)(X_t + Y_t + 1)} \\ &= \frac{X_t(X_t + Y_t + 1)}{(X_t + Y_t)(X_t + Y_t + 1)} \\ &= \frac{X_t}{X_t + Y_t}. \end{aligned}$$

From the martingale property we have $\mathbb{E} \left[\frac{X_n}{X_n + Y_n} \right] = \frac{X_0}{X_0 + Y_0}$. But $X_n + Y_n = X_0 + Y_0 + n$, a constant, so we can multiply both sides by this value to get $\mathbb{E}[X_n] = X_0 \left(\frac{X_0 + Y_0 + n}{X_0 + Y_0} \right)$.

G.1.3 Support your local police (20 points)

At one point I lived in a city whose local police department supported themselves in part by collecting fines for speeding tickets. A speeding ticket would cost 1 unit (approximately \$100), and it was unpredictable how often one would get a speeding ticket. For a price of 2 units, it was possible to purchase a metal placard to go on your vehicle identifying yourself as a supporter of the police union, which (at least according to local legend) would eliminate any fines for subsequent speeding tickets, but which would not eliminate the cost of any previous speeding tickets.

Let us consider the question of when to purchase a placard as a problem in on-line algorithms. It is possible to achieve a strict¹ competitive ratio of 2 by purchasing a placard after the second ticket. If one receives fewer than 2 tickets, both the on-line and off-line algorithms pay the same amount, and at 2 or more tickets the on-line algorithm pays 4 while the off-line pays 2 (the off-line algorithm purchased the placard before receiving any tickets at all).

1. Show that no deterministic algorithm can achieve a lower (strict) competitive ratio.
2. Show that a randomized algorithm can do so, against an oblivious adversary.

¹I.e., with no additive constant.

Solution

1. Any deterministic algorithm essentially just chooses some fixed number m of tickets to collect before buying the placard. Let n be the actual number of tickets issued. For $m = 0$, the competitive ratio is infinite when $n = 0$. For $m = 1$, the competitive ratio is 3 when $n = 1$. For $m > 2$, the competitive ratio is $(m + 2)/2 > 2$ when $n = m$. So $m = 2$ is the optimal choice.
2. Consider the following algorithm: with probability p , we purchase a placard after 1 ticket, and with probability $q = 1 - p$, we purchase a placard after 2 tickets. This gives a competitive ratio of 1 for $n = 0$, $1 + 2p$ for $n = 1$, and $(3p + 4q)/2 = (4 - p)/2 = 2 - p/2$ for $n \geq 2$. There is a clearly a trade-off between the two ratios $1 + 2p$ and $2 - p/2$. The break-even point is when they are equal, at $p = 2/5$. This gives a competitive ratio of $1 + 2p = 9/5$, which is less than 2.

G.1.4 Overloaded machines (20 points)

Suppose n^2 jobs are assigned to n machines with each job choosing a machine independently and uniformly at random. Let the load on a machine be the number of jobs assigned to it. Show that for any fixed $\delta > 0$ and sufficiently large n , there is a constant $c < 1$ such that the maximum load exceeds $(1 + \delta)n$ with probability at most nc^n .

Solution

This is a job for Chernoff bounds. For any particular machine, the load S is a sum of independent indicator variables and the mean load is $\mu = n$. So we have

$$\Pr[S \geq (1 + \delta)\mu] \leq \left(\frac{e^\delta}{(1 + \delta)^{1+\delta}} \right)^n.$$

Observe that $e^\delta/(1 + \delta)^{1+\delta} < 1$ for $\delta > 0$. One proof of this fact is to take the log to get $\delta - (1 + \delta) \log(1 + \delta)$, which equals 0 at $\delta = 0$, and then show that the logarithm is decreasing by showing that $\frac{d}{d\delta} \dots = 1 - \frac{1+\delta}{1+\delta} - \log(1 + \delta) = -\log(1 + \delta) < 0$ for all $\delta > 0$.

So we can let $c = e^\delta/(1 + \delta)^{1+\delta}$ to get a bound of c^n on the probability that any particular machine is overloaded and a bound of nc^n (from the union bound) on the probability that any of the machines is overloaded.

Appendix H

Probabilistic recurrences

Randomized algorithms often produce recurrences with ugly sums embedded inside them (see, for example, (1.3.1)). We'd like to have tools for pulling at least asymptotic bounds out of these recurrences without having to deal with the sums. This isn't always possible, but for certain simple recurrences we can make it work.

H.1 Recurrences with constant cost functions

Let us consider probabilistic recurrences of the form $T(n) = 1 + T(n - X_n)$, where X_n is a random variable with $0 < X_n \leq n$ and $T(0) = 0$. We assume we can compute a lower bound on $E[X_n]$ for each n , and we want to translate this lower bound into an upper bound on $E[T(n)]$.

H.2 Examples

- How long does it take to get our first heads if we repeatedly flip a coin that comes up heads with probability p ? Even though we probably already know the answer to this, we can solve it by solving the recurrence $T(1) = 1 + T(1 - X_1)$, $T(0) = 0$, where $E[X_1] = p$.
- **Hoare's FIND** [Hoa61b], often called **QuickSelect**, is an algorithm for finding the k -th smallest element of an unsorted array. It works like QuickSort, only after partitioning the array around a random pivot we throw away the part that doesn't contain our target and recurse only on the surviving piece. How many rounds of this must we do? Here $E[X_n]$ is more complicated, since after splitting our array of size

n into piles of size n' and $n - n' - 1$, we have to pick one or the other (or possibly just the pivot alone) based on the value of k .

- Suppose we start with n biased coins that each come up heads with probability p . In each round, we flip all the coins and throw away the ones that come up tails. How many rounds does it take to get rid of all of the coins? (This essentially tells us how tall a skip list [Pug90] can get.) Here we have $E[X_n] = (1 - p)n$.
- In the **coupon collector problem**, we sample from $1 \dots n$ with replacement until we see every value at least once. We can model this by a recurrence in which $T(k)$ is the time to get all the coupons given there are k left that we haven't seen. Here X_n is 1 with probability k/n and 0 with probability $(n - k)/n$, giving $E[X_n] = k/n$.
- Let's play **Chutes and Ladders** without the chutes and ladders. We start at location n , and whenever it's our turn, we roll a fair six-sided die X and move to $n - X$ unless this value is negative, in which case we stay put until the next turn. How many turns does it take to get to 0?

H.3 The Karp-Upfal-Wigderson bound

This is a bound on the expected number of rounds to finish a process where we start with a problem instance of size n , and after one round of work we get a new problem instance of size $n - X_n$, where X_n is a random variable whose distribution depends on n . It was originally described in a paper by Karp, Upfal, and Wigderson on analyzing parallel search algorithms [KUW88]. The bound applies when $E[X_n]$ is bounded below by a non-decreasing function $\mu(n)$.

Lemma H.3.1. *Let a be a constant, let $T(n) = 1 + T(n - X_n)$, where for each n , X_n is an integer-valued random variable satisfying $0 \leq X_n \leq n - a$ and let $T(a) = 0$. Let $E[X_n] \geq \mu(n)$ for all $n > a$, where μ is a positive non-decreasing function of n . Then*

$$E[T(n)] \leq \int_a^n \frac{1}{\mu(t)} dt. \quad (\text{H.3.1})$$

To get an intuition for why this works, imagine that X_n is the speed at which we drop from n , expressed in units per round. Traveling at this speed, it takes $1/X_n$ rounds to cross from $k + 1$ to k for any such interval

we pass. From the point of view of the interval $[k, k + 1]$, we don't know which n we are going to start from before we cross it, but we do know that for any $n \geq k + 1$ we start from, our speed will be at least $\mu(n) \geq \mu(k + 1)$ on average. So the time it takes will be at most $\int_k^{k+1} \frac{1}{\mu(t)} dt$ on average, and the total time is obtained by summing all of these intervals.

Of course, this intuition is not even close to a real proof (among other things, there may be a very dangerous confusion in there between $1/\mathbb{E}[X_n]$ and $\mathbb{E}[1/X_n]$), so we will give a real proof as well.

Proof of Lemma H.3.1. This is essentially the same proof as in Motwani and Raghavan [MR95], but we add some extra detail to allow for the possibility that $X_n = 0$.

Let $p = \Pr[X_n = 0]$, $q = 1 - p = \Pr[X_n \neq 0]$. Note we have $q > 0$ because otherwise $\mathbb{E}[X_n] = 0 < \mu(n)$. Then we have

$$\begin{aligned} \mathbb{E}[T(n)] &= 1 + \mathbb{E}[T(n - X_n)] \\ &= 1 + p \mathbb{E}[T(n - X_n) \mid X_n = 0] + q \mathbb{E}[T(n - X_n) \mid X_n \neq 0] \\ &= 1 + p \mathbb{E}[T(n)] + q \mathbb{E}[T(n - X_n) \mid X_n \neq 0]. \end{aligned}$$

Now we have $\mathbb{E}[T(n)]$ on both sides, which we don't like very much. So we collect it on the left-hand side:

$$(1 - p) \mathbb{E}[T(n)] = 1 + q \mathbb{E}[T(n - X_n) \mid X_n \neq 0],$$

divide both sides by $q = 1 - p$, and apply the induction hypothesis:

$$\begin{aligned} \mathbb{E}[T(n)] &= 1/q + \mathbb{E}[T(n - X_n) \mid X_n \neq 0] \\ &= 1/q + \mathbb{E}[\mathbb{E}[T(n - X_n) \mid X_n] \mid X_n \neq 0] \\ &\leq 1/q + \mathbb{E}\left[\int_a^{n-X_n} \frac{1}{\mu(t)} dt \mid X_n \neq 0\right] \\ &= 1/q + \mathbb{E}\left[\int_a^n \frac{1}{\mu(t)} dt - \int_{n-X_n}^n \frac{1}{\mu(t)} dt \mid X_n \neq 0\right] \\ &\leq 1/q + \int_a^n \frac{1}{\mu(t)} dt - \mathbb{E}\left[\frac{X_n}{\mu(n)} \mid X_n \neq 0\right] \\ &\leq 1/q + \int_a^n \frac{1}{\mu(t)} dt - \frac{\mathbb{E}[X_n \mid X_n \neq 0]}{\mu(n)}. \end{aligned}$$

The second-to-last step uses the fact that $\mu(t) \leq \mu(n)$ for $t \leq n$.

It may seem like we don't know what $\mathbb{E}[X_n \mid X_n \neq 0]$ is. But we know that $X_n \geq 0$, so we have $\mathbb{E}[X_n] = p \mathbb{E}[X_n \mid X_n = 0] + q \mathbb{E}[X_n \mid X_n \neq 0] =$

$qE[X_n | X_n \neq 0]$. So we can solve for $E[X_n | X_n \neq 0] = E[X_n]/q$. So let's plug this in:

$$\begin{aligned} E[T(n)] &\leq 1/q + \int_a^n \frac{1}{\mu(t)} dt - \frac{E[X_n]/q}{\mu(n)} \\ &\leq 1/q + \int_a^n \frac{1}{\mu(t)} dt - 1/q \\ &= \int_a^n \frac{1}{\mu(t)} dt. \end{aligned}$$

This concludes the proof. \square

Now we just need to find some applications.

H.3.1 Waiting for heads

For the recurrence $T(1) = 1 + T(1 - X_1)$ with $E[X_1] = p$, we set $\mu(n) = p$ and get $E[T(1)] \leq \int_0^1 \frac{1}{p} dt = \frac{1}{p}$, which happens to be exactly the right answer.

H.3.2 Quickselect

In Quickselect, we pick a random pivot and split the original array of size n into three piles of size m (less than the pivot), 1 (the pivot itself), and $n - m - 1$ (greater than the pivot). We then figure out which of the three piles contains the k -th smallest element (depend on how k compares to $m - 1$) and recurse, stopping when we hit a pile with 1 element. It's easiest to analyze this by assuming that we recurse in the largest of the three piles, i.e., that our recurrence is $T(n) = 1 + \max(T(m), T(n - m - 1))$, where m is uniform in $0 \dots n - 1$. The exact value of $E[\max(m, n - m - 1)]$ is a little messy to compute (among other things, it depends on whether n is odd or even), but it's not hard to see that it's always less than $(3/4)n$. So letting $\mu(n) = n/4$, we get

$$E[T(n)] \leq \int_1^n \frac{1}{t/4} dt = 4 \ln n.$$

H.3.3 Tossing coins

Here we have $E[X_n] = (1 - p)n$. If we let $\mu(n) = (1 - p)n$ and plug into the formula without thinking about it too much, we get

$$E[T(n)] \leq \int_0^n \frac{1}{(1-p)t} dt = \frac{1}{1-p} (\ln n - \ln 0).$$

That $\ln 0$ is trouble. We can fix it by making $\mu(n) = (1 - p)\lceil n \rceil$, to get

$$\begin{aligned} \mathbb{E}[T(n)] &\leq \int_{0+}^n \frac{1}{(1-p)\lceil t \rceil} dt \\ &= \frac{1}{1-p} \sum_{k=1}^n \frac{1}{k} \\ &= \frac{H_n}{1-p}. \end{aligned}$$

H.3.4 Coupon collector

Now that we know how to avoid dividing by zero, this is easy and fun. Let $\mu(x) = \lceil x \rceil/n$, then we have

$$\begin{aligned} \mathbb{E}[T(n)] &\leq \int_{0+}^n \frac{n}{\lceil t \rceil} dt \\ &= \sum_{k=1}^n \frac{n}{k} \\ &= nH_n. \end{aligned}$$

As it happens, this is the exact answer for this case. This will happen whenever X is always a 0–1 variable¹ and we define $\mu(x) = \mathbb{E}[X \mid n = \lceil x \rceil]$, which can be seen by spending far too much time thinking about the precise sources of error in the inequalities in the proof.

H.3.5 Chutes and ladders

Let $\mu(n)$ be the expected drop from position n . We have to be a little bit careful about small n , but we can compute that in general $\mu(n) = \frac{1}{6} \sum_{i=0}^{\min(n,6)} i$. For fractional values x we will set $\mu(x) = \mu(\lceil x \rceil)$ as before.

¹A **0–1 random variable** is also called a **Bernoulli random variable**, but 0–1 is shorter to type and more informative. Even more confusing, the underlying experiment that gives rise to a Bernoulli random variable goes by the entirely different name of a **Poisson trial**. Jacob Bernoulli and Siméon-Denis Poisson were great mathematicians, but there are probably better ways to remember them.

Then we have

$$\begin{aligned} \mathbb{E}[T(n)] &\leq \int_{0+}^n \frac{1}{\mu(t)} dt \\ &= \sum_{k=1}^n \frac{1}{\mu(k)} \end{aligned}$$

We can summarize the values in the following table:

n	$\mu(n)$	$1/\mu(n)$	$\sum 1/\mu(k)$
1	1/6	6	6
2	1/2	2	8
3	1	1	9
4	5/3	3/5	48/5
5	5/2	2/5	10
≥ 6	7/2	2/7	$10 + (2/7)(n - 5) = (2/7)n + 65/7$

This is a slight overestimate; for example, we can calculate by hand that the expected waiting time for $n = 2$ is 6 and for $n = 3$ that it is $20/3 = 6 + 2/3$.

We can also consider the generalized version of the game where we start at n and drop by $1 \dots n$ each turn as long as the drop wouldn't take us below 0. Now the expected drop from position k is $k(k+1)/2n$, and so we can apply the formula to get

$$\mathbb{E}[T(n)] \leq \sum_{k=1}^n \frac{2n}{k(k+1)}.$$

The sum of $\frac{1}{k(k+1)}$ when k goes from 1 to n happens to have a very nice value; it's exactly $\frac{n}{n+1} = 1 + \frac{1}{n+1}$.² So in this case we can rewrite the bound as $2n \cdot \frac{n}{n+1} = \frac{2n^2}{n+1}$.

H.4 High-probability bounds

So far we have only considered bounds on the expected value of $T(n)$. Suppose we want to show that $T(n)$ is in fact small with high probability,

²Proof: Trivially true for $n = 0$; for larger n , compute $\sum_{k=1}^n \frac{1}{k(k+1)} = \sum_{k=1}^{n-1} \frac{1}{k(k+1)} + \frac{1}{n(n+1)} = \frac{n-1}{n} + \frac{1}{n(n+1)} = \frac{(n-1)(n+1)-1}{n(n+1)} = \frac{n^2}{n(n+1)} = n/(n+1)$.

i.e., a statement of the form $\Pr [T(n) \geq t] \leq \epsilon$. There are two natural ways to do this: we can repeatedly apply Markov's inequality to the expectation bound, or we can attempt to analyze the recurrence in more detail. The first method tends to give weaker bounds but it's easier.

H.4.1 High-probability bounds from expectation bounds

Given $E[T(n)] \leq m$, we have $\Pr [T(n) \geq \alpha m] \leq \alpha^{-1}$. This does not give a very good bound on probability; if we want to show $\Pr [T(n) \geq t] \leq n^{-c}$ for some constant c (a typical high-probability bound), we need $t \geq n^c m$. But we can get a better bound if m bounds the expected time starting from any reachable state, as is the case for the class of problems we have been considering.

The idea is that if $T(n)$ exceeds αm , we restart the analysis and argue that $\Pr [T(n) \geq 2\alpha m \mid T(n) \geq \alpha m] \leq \alpha^{-1}$, from which it follows that $\Pr [T(n) \geq 2\alpha m] \leq \alpha^{-2}$. In general, for any non-negative integer k , we have $\Pr [T(n) \geq k\alpha m] \leq \alpha^{-k}$. Now we just need to figure out how to pick α to minimize this quantity for fixed t .

Let $t = k\alpha m$. Then $k = t/\alpha m$ and we seek to minimize $\alpha^{-t/\alpha m}$. Taking the logarithm gives $-(t/m)(\ln \alpha)/\alpha$. The t/m factor is irrelevant to the minimization problem, so we are left with minimizing $-(\ln \alpha)/\alpha$. Taking the derivative gives $-\alpha^{-2} + \alpha^{-2} \ln \alpha$; this is zero when $\ln \alpha = 1$ or $\alpha = e$. (This popular constant shows up often in problems like this.) So we get $\Pr [T(n) \geq kem] \leq e^{-k}$, or, letting $k = \ln(1/\epsilon)$, $\Pr [T(n) \geq em \ln(1/\epsilon)] \leq \epsilon$.

So, for example, we can get an n^{-c} bound on the probability of running too long by setting our time bound to $em \ln(n^c) = cem \ln n = O(m \log n)$. We can't hope to do better than $O(m)$, so this bound is tight up to a log factor.

H.4.2 Detailed analysis of the recurrence

As Lance Fortnow has explained,³ getting rid of log factors is what theoretical computer science is all about. So we'd like to do better than an $O(m \log n)$ bound if we can. In some cases this is not too hard.

Suppose for each n , $T(n) = 1 + T(X)$, where $E[X] \leq \alpha n$ for a fixed constant α . Let $X_0 = n$, and let X_1, X_2 , etc., be the sequence of sizes of the remaining problem at time 1, 2, etc. Then we have $E[X_1] \leq \alpha n$ from our assumption. But we also have $E[X_2] = E[E[X_2 \mid X_1]] \leq E[\alpha X_1] = \alpha E[X_1] \leq \alpha^2 n$, and by induction we can show that $E[X_k] \leq \alpha^k n$ for all k .

³<http://weblog.fortnow.com/2009/01/soda-and-me.html>

Since X_k is integer-valued, $E[X_k]$ is an upper bound on $\Pr[X_k > 0]$; we thus get $\Pr[T(n) \geq k] = \Pr[X_k > 0] \leq \alpha^k n$. We can solve for the value of k that makes this less than ϵ : $k = -\log(n/\epsilon)/\log \alpha = \log_{1/\alpha} n + \log_{1/\alpha}(1/\epsilon)$.

For comparison, the bound on the expectation of $T(n)$ from Lemma [H.3.1](#) is $H(n)/(1 - \alpha)$. This is actually pretty close to $\log_{1/\alpha} n$ when α is close to 1, and is not too bad even for smaller α . But the difference is that the dependence on $\log(1/\epsilon)$ is additive with the tighter analysis, so for fixed c we get $\Pr[T(n) \geq t] \leq n^{-c}$ at $t = O(\log n + \log n^c) = O(\log n)$ instead of $O(\log n \log n^c) = O(\log^2 n)$.

H.5 More general recurrences

We didn't do these, but if we had to, it might be worth looking at Roura's Improved Master Theorems [[Rou01](#)].

Bibliography

- [AA11] Dan Alistarh and James Aspnes. Sub-logarithmic test-and-set against a weak adversary. In *Distributed Computing: 25th International Symposium, DISC 2011*, volume 6950 of *Lecture Notes in Computer Science*, pages 97–109. Springer-Verlag, September 2011.
- [AACH12] James Aspnes, Hagit Attiya, and Keren Censor-Hillel. Poly-logarithmic concurrent data structures from monotone circuits. *Journal of the ACM*, 59(1):2:1–2:24, February 2012.
- [AAG⁺10] Dan Alistarh, Hagit Attiya, Seth Gilbert, Andrei Giurgiu, and Rachid Guerraoui. Fast randomized test-and-set and renaming. In Nancy A. Lynch and Alexander A. Shvartsman, editors, *Distributed Computing, 24th International Symposium, DISC 2010, Cambridge, MA, USA, September 13-15, 2010. Proceedings*, volume 6343 of *Lecture Notes in Computer Science*, pages 94–108. Springer, 2010.
- [ABMRT96] Arne Andersson, Peter Bro Miltersen, Søren Riis, and Mikkel Thorup. Static dictionaries on AC^0 RAMs: Query time $\theta(\sqrt{\log n / \log \log n})$ is necessary and sufficient. In *FOCS*, pages 441–450, 1996.
- [Abr88] Karl Abrahamson. On achieving consensus using a shared memory. In *Proceedings of the 2nd annual ACM-SIAM symposium on Discrete algorithms*, pages 291–302, 1988.
- [AC08] Hagit Attiya and Keren Censor. Tight bounds for asynchronous randomized consensus. *Journal of the ACM*, 55(5):20, October 2008.

- [ACBF02] Peter Auer, Nicolò Cesa-Bianchi, and Paul Fischer. Finite-time analysis of the multiarmed bandit problem. *Machine Learning*, 47:235–256, 2002.
- [Ach03] Dimitris Achlioptas. Database-friendly random projections: Johnson-Lindenstrauss with binary coins. *J. Comput. Syst. Sci.*, 66(4):671–687, June 2003.
- [Adl78] Leonard Adleman. Two theorems on random polynomial time. In *Proceedings of the 19th Annual Symposium on Foundations of Computer Science*, pages 75–83, Washington, DC, USA, 1978. IEEE Computer Society.
- [AE19] James Aspnes and He Yang Er. Consensus with max registers. In Jukka Suomela, editor, *33rd International Symposium on Distributed Computing (DISC 2019)*, volume 146 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 1:1–1:9, Dagstuhl, Germany, 2019. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.
- [AF01] David Aldous and James Allen Fill. Reversible Markov chains and random walks on graphs. Unpublished manuscript, available at <http://www.stat.berkeley.edu/~aldous/RWG/book.html>, 2001.
- [AH90] James Aspnes and Maurice Herlihy. Fast randomized consensus using shared memory. *Journal of Algorithms*, 11(3):441–461, September 1990.
- [AKS83] M. Ajtai, J. Komlós, and E. Szemerédi. An $O(n \log n)$ sorting network. In *STOC '83: Proceedings of the fifteenth annual ACM symposium on Theory of computing*, pages 1–9, New York, NY, USA, 1983. ACM.
- [AKS04] Manindra Agrawal, Neeraj Kayal, and Nitin Saxena. PRIMES is in P. *Annals of Mathematics*, 160:781–793, 2004.
- [Alo91] Noga Alon. A parallel algorithmic version of the local lemma. *Random Structures & Algorithms*, 2(4):367–378, 1991.
- [AS92] Noga Alon and Joel H. Spencer. *The Probabilistic Method*. John Wiley & Sons, 1992.

- [AS07] James Aspnes and Gauri Shah. Skip graphs. *ACM Transactions on Algorithms*, 3(4):37, November 2007.
- [Asp15] James Aspnes. Faster randomized consensus with an oblivious adversary. *Distributed Computing*, 28(1):21–29, February 2015.
- [AVL62] G. M. Adelson-Velskii and E. M. Landis. An information organization algorithm. *Doklady Akademia Nauk SSSR*, 146:263–266, 1962.
- [AW96] James Aspnes and Orli Waarts. Randomized consensus in $O(n \log^2 n)$ operations per processor. *SIAM Journal on Computing*, 25(5):1024–1044, October 1996.
- [Azu67] Kazuoki Azuma. Weighted sums of certain dependent random variables. *Tôhoku Mathematical Journal*, 19(3):357–367, 1967.
- [Bab79] László Babai. Monte-carlo algorithms in graph isomorphism testing. Technical Report D.M.S. 79-10, Université de Montréal, 1979.
- [BBBV97] Charles H. Bennett, Ethan Bernstein, Gilles Brassard, and Umesh Vazirani. Strengths and weaknesses of quantum computing. *SIAM J. Comput.*, 26(5):1510–1523, October 1997.
- [BCB12] Sébastien Bubeck and Nicolò Cesa-Bianchi. Regret analysis of stochastic and nonstochastic multi-armed bandit problems. *Foundations and Trends in Machine Learning*, 5(1):1–122, 2012.
- [BD92] Dave Bayer and Persi Diaconis. Trailing the dovetail shuffle to its lair. *Annals of Applied Probability*, 2(2):294–313, 1992.
- [Bec91] József Beck. An algorithmic approach to the lovász local lemma. i. *Random Structures & Algorithms*, 2(4):343–365, 1991.
- [Bel57] Richard Earnest Bellman. *Dynamic Programming*. Princeton University Press, 1957.
- [Blo70] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, 1970.
- [Bol01] Béla Bollobás. *Random Graphs*. Cambridge University Press, second edition, 2001.

- [BR91] Gabriel Bracha and Ophir Rachman. Randomized consensus in expected $O(n^2 \log n)$ operations. In Sam Toueg, Paul G. Spirakis, and Lefteris M. Kirousis, editors, *Distributed Algorithms, 5th International Workshop*, volume 579 of *Lecture Notes in Computer Science*, pages 143–150, Delphi, Greece, 7–9 October 1991. Springer, 1992.
- [Bro86] Andrei Z. Broder. How hard is to marry at random? (on the approximation of the permanent). In *Proceedings of the Eighteenth Annual ACM Symposium on Theory of Computing, 28-30 May 1986, Berkeley, California, USA*, pages 50–58, 1986.
- [Bro88] Andrei Z. Broder. Errata to “how hard is to marry at random? (on the approximation of the permanent)”. In *Proceedings of the Twentieth Annual ACM Symposium on Theory of Computing, 2-4 May 1988, Chicago, Illinois, USA*, page 551, 1988.
- [BRSW06] Boaz Barak, Anup Rao, Ronen Shaltiel, and Avi Wigderson. 2-source dispersers for sub-polynomial entropy and Ramsey graphs beating the Frankl-Wilson construction. In *Proceedings of the thirty-eighth annual ACM symposium on Theory of computing, STOC '06*, pages 671–680, New York, NY, USA, 2006. ACM.
- [CCFC04] Moses Charikar, Kevin Chen, and Martin Farach-Colton. Finding frequent items in data streams. *Theor. Comput. Sci.*, 312(1):3–15, 2004.
- [CM03] Saar Cohen and Yossi Matias. Spectral bloom filters. In Alon Y. Halevy, Zachary G. Ives, and AnHai Doan, editors, *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data, San Diego, California, USA, June 9-12, 2003*, pages 241–252, 2003.
- [CM05] Graham Cormode and S. Muthukrishnan. An improved data stream summary: the count-min sketch and its applications. *J. Algorithms*, 55(1):58–75, 2005.
- [CMV13] Kai-Min Chung, Michael Mitzenmacher, and Salil Vadhan. Why simple hash functions work: Exploiting the entropy in a data stream. *Theory of Computing*, 9(1):897–945, 2013.

- [CS00] Artur Czumaj and Christian Scheideler. Coloring non-uniform hypergraphs: a new algorithmic approach to the general Lovász local lemma. In *Proceedings of the eleventh annual ACM-SIAM symposium on Discrete algorithms*, SODA '00, pages 30–39, Philadelphia, PA, USA, 2000. Society for Industrial and Applied Mathematics.
- [CW77] J. Lawrence Carter and Mark N. Wegman. Universal classes of hash functions (extended abstract). In *Proceedings of the ninth annual ACM symposium on Theory of computing*, STOC '77, pages 106–112, New York, NY, USA, 1977. ACM.
- [Deu89] David Deutsch. Quantum computational networks. *Proceedings of the Royal Society of London. A. Mathematical and Physical Sciences*, 425(1868):73–90, 1989.
- [Dev88] Luc Devroye. Applications of the theory of records in the study of random trees. *Acta Informatica*, 26(1-2):123–130, October 1988.
- [DGM02] Martin Dyer, Catherine Greenhill, and Mike Molloy. Very rapid mixing of the Glauber dynamics for proper colorings on bounded-degree graphs. *Random Struct. Algorithms*, 20:98–114, January 2002.
- [DHKP97] Martin Dietzfelbinger, Torben Hagerup, Jyrki Katajainen, and Martti Penttonen. A reliable randomized algorithm for the closest-pair problem. *J. Algorithms*, 25(1):19–51, 1997.
- [Dir39] P. A. M. Dirac. A new notation for quantum mechanics. *Mathematical Proceedings of the Cambridge Philosophical Society*, 35:416–418, 6 1939.
- [DJ92] David Deutsch and Richard Jozsa. Rapid solution of problems by quantum computation. *Proceedings of the Royal Society of London. Series A: Mathematical and Physical Sciences*, 439(1907):553–558, 1992.
- [Dum56] A. I. Dumey. Indexing for rapid random-access memory. *Computers and Automation*, 5(12):6–9, 1956.
- [Dye03] Martin Dyer. Approximate counting by dynamic programming. In *Proceedings of the thirty-fifth annual ACM symposium on*

- Theory of computing*, STOC '03, pages 693–699, New York, NY, USA, 2003. ACM.
- [EL75] Paul Erdős and Laszlo Lovász. Problems and results on 3-chromatic hypergraphs and some related questions. In A. Hajnal, R. Rado, and V. T. Sós, editors, *Infinite and Finite Sets (to Paul Erdős on his 60th birthday)*, pages 609–627. North-Holland, 1975.
- [Epp16] David Eppstein. Cuckoo filter: Simplification and analysis. In Rasmus Pagh, editor, *15th Scandinavian Symposium and Workshops on Algorithm Theory, SWAT 2016, June 22–24, 2016, Reykjavik, Iceland*, volume 53 of *LIPICs*, pages 8:1–8:12. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2016.
- [Erd45] P. Erdős. On a lemma of Littlewood and Offord. *Bulletin of the American Mathematical Society*, 51(12):898–902, 1945.
- [Erd47] P. Erdős. Some remarks on the theory of graphs. *Bulletin of the American Mathematical Society*, 53:292–294, 1947.
- [ES35] P. Erdős and G. Szekeres. A combinatorial problem in geometry. *Compositio Mathematica*, 2:463–470, 1935.
- [Eul68] M. L. Euler. Remarques sur un beau rapport entre les séries des pussances tant directes que réciproques. *Mémoires de l'Académie des Sciences de Berlin*, 17:83–106, 1768.
- [FAKM14] Bin Fan, David G. Andersen, Michael Kaminsky, and Michael Mitzenmacher. Cuckoo filter: Practically better than bloom. In Aruna Seneviratne, Christophe Diot, Jim Kurose, Augustin Chaintreau, and Luigi Rizzo, editors, *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies, CoNEXT 2014, Sydney, Australia, December 2–5, 2014*, pages 75–88. ACM, 2014.
- [FCAB00] Li Fan, Pei Cao, Jussara M. Almeida, and Andrei Z. Broder. Summary cache: a scalable wide-area web cache sharing protocol. *IEEE/ACM Trans. Netw.*, 8(3):281–293, 2000.
- [Fel68] William Feller. *An Introduction to Probability Theory and Its Applications*, volume 1. Wiley, third edition, 1968.

- [Fel71] William Feller. *An Introduction to Probability Theory and Its Applications*, volume 2. Wiley, second edition, 1971.
- [FGQ12] Xiequan Fan, Ion Grama, and Liu Quansheng. Hoeffding's inequality for supermartingales. Available as <http://arxiv.org/abs/1109.4359>, July 2012.
- [FKS84] M.L. Fredman, J. Komlós, and E. Szemerédi. Storing a Sparse Table with $O(1)$ Worst Case Access Time. *Journal of the ACM (JACM)*, 31(3):538–544, 1984.
- [FLP85] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, April 1985.
- [FNM85] Philippe Flajolet and G Nigel Martin. Probabilistic counting algorithms for data base applications. *Journal of computer and system sciences*, 31(2):182–209, 1985.
- [FPSS03] Dimitris Fotakis, Rasmus Pagh, Peter Sanders, and Paul Spirakis. Space efficient hash tables with worst case constant access time. In Helmut Alt and Michel Habib, editors, *STACS 2003*, volume 2607 of *Lecture Notes in Computer Science*, pages 271–282. Springer Berlin Heidelberg, 2003.
- [FR75] Robert W. Floyd and Ronald L. Rivest. Expected time bounds for selection. *Commun. ACM*, 18(3):165–172, 1975.
- [GKP88] Ronald L. Graham, Donald E. Knuth, and Oren Patashnik. *Concrete Mathematics: A Foundation for Computer Science*. Addison-Wesley, 1988.
- [Goo83] Nelson Goodman. *Fact, Fiction, and Forecast*. Harvard University Press, 1983.
- [GR93] Igal Galperin and Ronald L. Rivest. Scapegoat trees. In *Proceedings of the Fourth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 165–174. Society for Industrial and Applied Mathematics, 1993.
- [Gro96] Lov K Grover. A fast quantum mechanical algorithm for database search. In *Proceedings of the twenty-eighth annual ACM symposium on Theory of computing*, pages 212–219.

- ACM, 1996. Available as <http://arxiv.org/abs/quant-ph/9605043>.
- [GS78] Leo J. Guibas and Robert Sedgewick. A dichromatic framework for balanced trees. In *Foundations of Computer Science, 1978., 19th Annual Symposium on*, pages 8–21. IEEE, 1978.
- [GS92] G. R. Grimmett and D. R. Stirzaker. *Probability and Random Processes*. Oxford Science Publications, second edition, 1992.
- [GS01] Geoffrey R. Grimmett and David R. Stirzaker. *Probability and Random Processes*. Oxford University Press, third edition, 2001.
- [Gur00] Venkatesen Guruswami. Rapidly mixing Markov chains: A comparison of techniques. Available at <ftp://theory.lcs.mit.edu/pub/people/venkat/markov-survey.ps>, 2000.
- [GW94] Michel X. Goemans and David P. Williamson. New $3/4$ -approximation algorithms for the maximum satisfiability problem. *SIAM J. Discret. Math.*, 7:656–666, November 1994.
- [GW95] Michel X. Goemans and David P. Williamson. Improved approximation algorithms for maximum cut and satisfiability problems using semidefinite programming. *Journal of the ACM*, 42(6):1115–1145, 1995.
- [GW12] George Giakkoupis and Philipp Woelfel. On the time and space complexity of randomized test-and-set. In Darek Kowalski and Alessandro Panconesi, editors, *ACM Symposium on Principles of Distributed Computing, PODC '12, Funchal, Madeira, Portugal, July 16-18, 2012*, pages 19–28. ACM, 2012.
- [Has70] W. K. Hastings. Monte carlo sampling methods using Markov chains and their applications. *Biometrika*, 57(1):97–109, 1970.
- [HH80] P. Hall and C.C. Heyde. *Martingale Limit Theory and Its Application*. Academic Press, 1980.
- [HK73] John E. Hopcroft and Richard M. Karp. An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs. *SIAM J. Comput.*, 2(4):225–231, 1973.

- [Hoa61a] C. A. R. Hoare. Algorithm 64: Quicksort. *Commun. ACM*, 4:321, July 1961.
- [Hoa61b] C. A. R. Hoare. Algorithm 65: find. *Commun. ACM*, 4:321–322, July 1961.
- [Hoe63] Wassily Hoeffding. Probability inequalities for sums of bounded random variables. *Journal of the American Statistical Association*, 58(301):13–30, March 1963.
- [Hol16] Jonathan Holmes. AI is already making inroads into journalism but could it win a Pulitzer? <https://www.theguardian.com/media/2016/apr/03/artificial-intelligence-robot-reporter-pulitzer-prize>, April 2016. Accessed September 18th, 2016.
- [HST08] Maurice Herlihy, Nir Shavit, and Moran Tzafrir. Hopscotch hashing. In *Proceedings of the 22nd international symposium on Distributed Computing, DISC '08*, pages 350–364, Berlin, Heidelberg, 2008. Springer-Verlag.
- [IM98] Piotr Indyk and Rajeev Motwani. Approximate nearest neighbors: Towards removing the curse of dimensionality. In *STOC*, pages 604–613, 1998.
- [JL84] William B. Johnson and Joram Lindenstrauss. Extensions of Lipschitz mappings into a Hilbert space. In *Conference in Modern Analysis and Probability (New Haven, Connecticut, 1982)*, number 26 in Contemporary Mathematics, pages 189–206. American Mathematical Society, 1984.
- [JLR00] Svante Janson, Tomasz Łuczak, and Andrzej Ruciński. *Random Graphs*. John Wiley & Sons, 2000.
- [JS89] Mark Jerrum and Alistair Sinclair. Approximating the permanent. *SIAM J. Comput.*, 18(6):1149–1178, 1989.
- [JSV04] Mark Jerrum, Alistair Sinclair, and Eric Vigoda. A polynomial-time approximation algorithm for the permanent of a matrix with nonnegative entries. *J. ACM*, 51(4):671–697, 2004.
- [Kar93] David R. Karger. Global min-cuts in rnc, and other ramifications of a simple min-out algorithm. In *Proceedings of the*

- fourth annual ACM-SIAM Symposium on Discrete algorithms, SODA '93*, pages 21–30, Philadelphia, PA, USA, 1993. Society for Industrial and Applied Mathematics.
- [KL85] Richard M. Karp and Michael Luby. Monte-carlo algorithms for the planar multiterminal network reliability problem. *J. Complexity*, 1(1):45–64, 1985.
- [KM08] Adam Kirsch and Michael Mitzenmacher. Less hashing, same performance: Building a better bloom filter. *Random Struct. Algorithms*, 33(2):187–218, 2008.
- [KNW10] Daniel M. Kane, Jelani Nelson, and David P. Woodruff. An optimal algorithm for the distinct elements problem. In *Proceedings of the Twenty-Ninth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS '10*, pages 41–52, New York, NY, USA, 2010. ACM.
- [Kol33] A.N. Kolmogorov. *Grundbegriffe der Wahrscheinlichkeitsrechnung*. Springer, 1933.
- [KR99] V. S. Anil Kumar and H. Ramesh. Markovian coupling vs. conductance for the Jerrum-Sinclair chain. In *FOCS*, pages 241–252, 1999.
- [KS76] J.G. Kemeny and J.L. Snell. *Finite Markov Chains: With a New Appendix "Generalization of a Fundamental Matrix"*. Undergraduate Texts in Mathematics. Springer, 1976.
- [KSK76] John G. Kemeny, J. Laurie. Snell, and Anthony W. Knapp. *Denumerable Markov Chains*, volume 40 of *Graduate Texts in Mathematics*. Springer, 1976.
- [KT75] Samuel Karlin and Howard M. Taylor. *A First Course in Stochastic Processes*. Academic Press, second edition, 1975.
- [KUW88] Richard M. Karp, Eli Upfal, and Avi Wigderson. The complexity of parallel search. *Journal of Computer and System Sciences*, 36(2):225–253, 1988.
- [LAA87] Michael C. Loui and Hosame H. Abu-Amara. Memory requirements for agreement among unreliable asynchronous processes. *Advances in Computing Research*, pages 163–183, 1987.

- [Li80] Shuo-Yen Robert Li. A martingale approach to the study of occurrence of sequence patterns in repeated experiments. *Annals of Probability*, 8(6):1171–1176, 1980.
- [Lin92] Torgny Lindvall. *Lectures on the Coupling Method*. Wiley, 1992.
- [LPW09] David A. Levin, Yuval Peres, and Elizabeth L. Wimmer. *Markov Chains and Mixing Times*. American Mathematical Society, 2009.
- [LR85] T. L. Lai and Herbert Robbins. Asymptotically efficient adaptive allocation rules. *Advances in Applied Mathematics*, 6(1):4–22, March 1985.
- [Lub85] Michael Luby. A simple parallel algorithm for the maximal independent set problem. In *Proceedings of the seventeenth annual ACM symposium on Theory of computing*, pages 1–10, New York, NY, USA, 1985. ACM.
- [LW05] Michael Luby and Avi Wigderson. Pairwise independence and derandomization. *Foundations and Trends in Theoretical Computer Science*, 1(4):237–301, 2005.
- [McC85] Edward M. McCreight. Priority search trees. *SIAM J. Comput.*, 14(2):257–276, 1985.
- [McD89] Colin McDiarmid. On the method of bounded differences. In *Surveys in Combinatorics, 1989: Invited Papers at the Twelfth British Combinatorial Conference*, pages 148–188, 1989.
- [MH92] Colin McDiarmid and Ryan Hayward. Strong concentration for Quicksort. In *Proceedings of the Third Annual ACM-SIAM Symposium on Discrete algorithms*, SODA '92, pages 414–421, Philadelphia, PA, USA, 1992. Society for Industrial and Applied Mathematics.
- [Mil76] Gary L. Miller. Riemann's hypothesis and tests for primality. *Journal of Computer and System Sciences*, 13(3):300–317, 1976.
- [ML86] Lothar F. Mackert and Guy M. Lohman. R* optimizer validation and performance evaluation for distributed queries. In Wesley W. Chu, Georges Gardarin, Setsuo Ohsuga, and Yahiko

- Kambayashi, editors, *VLDB'86 Twelfth International Conference on Very Large Data Bases, August 25-28, 1986, Kyoto, Japan, Proceedings*, pages 149–159. Morgan Kaufmann, 1986.
- [MN98] Makoto Matsumoto and Takuji Nishimura. Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Trans. Model. Comput. Simul.*, 8:3–30, January 1998.
- [Mos09] Robin A. Moser. A constructive proof of the Lovász local lemma. In *Proceedings of the 41st annual ACM Symposium on Theory of Computing, STOC '09*, pages 343–350, New York, NY, USA, 2009. ACM.
- [MR95] Rajeev Motwani and Prabhakar Raghavan. *Randomized Algorithms*. Cambridge University Press, 1995.
- [MR98] Michael Molloy and Bruce Reed. Further algorithmic aspects of the local lemma. In *Proceedings of the thirtieth annual ACM symposium on Theory of computing, STOC '98*, pages 524–529, New York, NY, USA, 1998. ACM.
- [MRR⁺53] Nicholas Metropolis, Arianna W. Rosenbluth, Marshall N. Rosenbluth, Augusta H. Teller, and Edward Teller. Equation of state calculations by fast computing machines. *J. Chem. Phys.*, 21(6):1087–1092, 1953.
- [MT05] Ravi Montenegro and Prasad Tetali. Mathematical aspects of mixing times in markov chains. *Foundations and Trends in Theoretical Computer Science*, 1(3), 2005.
- [MT10] Robin A. Moser and Gábor Tardos. A constructive proof of the general Lovász local lemma. *J. ACM*, 57:11:1–11:15, February 2010.
- [MU05] Michael Mitzenmacher and Eli Upfal. *Probability and Computing: Randomized Algorithms and Probabilistic Analysis*. Cambridge University Press, 2005.
- [Mun11] Randall Munroe. Sports. <http://xkcd.com/904/>, 2011. Accessed September 18th, 2016.
- [Pag01] Rasmus Pagh. On the cell probe complexity of membership and perfect hashing. In *STOC*, pages 425–432, 2001.

- [Pag06] Rasmus Pagh. Cuckoo hashing for undergraduates. Available at <http://www.it-c.dk/people/pagh/papers/cuckoo-undergrad.pdf>, 2006.
- [Pap91] Christos H. Papadimitriou. On selecting a satisfying truth assignment (extended abstract). In *32nd Annual Symposium on Foundations of Computer Science, San Juan, Puerto Rico, 1-4 October 1991*, pages 163–169. IEEE Computer Society, 1991.
- [PPR05] Anna Pagh, Rasmus Pagh, and S. Srinivasa Rao. An optimal bloom filter replacement. In *Proceedings of the Sixteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2005, Vancouver, British Columbia, Canada, January 23-25, 2005*, pages 823–829. SIAM, 2005.
- [PR04] Rasmus Pagh and Flemming Friche Rodler. Cuckoo hashing. *J. Algorithms*, 51(2):122–144, 2004.
- [PSL80] M. Pease, R. Shostak, and L. Lamport. Reaching agreement in the presence of faults. *Journal of the ACM*, 27(2):228–234, April 1980.
- [PT12] Mihai Patrascu and Mikkel Thorup. The power of simple tabulation hashing. *J. ACM*, 59(3):14, 2012.
- [Pug90] William Pugh. Skip Lists: A Probabilistic Alternative to Balanced Trees. *Communications of the ACM*, 33(6):668–676, June 1990.
- [Rab80] Michael O Rabin. Probabilistic algorithm for testing primality. *Journal of Number Theory*, 12(1):128–138, 1980.
- [Rag88] Prabhakar Raghavan. Probabilistic construction of deterministic algorithms: approximating packing integer programs. *J. Comput. Syst. Sci.*, 37:130–143, October 1988.
- [Rou01] Salvador Roura. Improved master theorems for divide-and-conquer recurrences. *J. ACM*, 48:170–205, March 2001.
- [RT87] Prabhakar Raghavan and Clark D. Tompson. Randomized rounding: a technique for provably good algorithms and algorithmic proofs. *Combinatorica*, 7:365–374, December 1987.

- [SA96] Raimund Seidel and Cecilia R. Aragon. Randomized search trees. *Algorithmica*, 16(4/5):464–497, 1996. Available at <http://people.ischool.berkeley.edu/~aragon/pubs/rst96.pdf>.
- [She11] Irina Shevtsova. On the asymptotically exact constants in the Berry-Esseen-Katz inequality. *Theory of Probability & Its Applications*, 55(2):225–252, 2011. A preprint is available at <http://arxiv.org/pdf/1111.6554.pdf>.
- [Sho97] Peter W Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM journal on computing*, 26(5):1484–1509, 1997.
- [Spu12] Francis Spufford. *Red Plenty*. Graywolf Press, 2012.
- [Sri08] Aravind Srinivasan. Improved algorithmic versions of the Lovász local lemma. In *Proceedings of the nineteenth annual ACM-SIAM symposium on Discrete algorithms*, SODA '08, pages 611–620, Philadelphia, PA, USA, 2008. Society for Industrial and Applied Mathematics.
- [SS71] A. Schönhage and V. Strassen. Schnelle multiplikation großer zahlen. *Computing*, 7(3–4):281–292, 1971.
- [SS87] E. Shamir and J. Spencer. Sharp concentration of the chromatic number on random graphs $G_{n,p}$. *Combinatorica*, 7:121–129, January 1987.
- [Str03] Gilbert Strang. *Introduction to linear algebra*. Wellesley-Cambridge Press, 2003.
- [SW13] Dennis Stanton and Dennis White. *Constructive Combinatorics*. Undergraduate Texts in Mathematics. Springer, 2013.
- [Tod91] Seinosuke Toda. Pp is as hard as the polynomial-time hierarchy. *SIAM J. Comput.*, 20(5):865–877, 1991.
- [Tof80] Tommaso Toffoli. Reversible computing. Technical Report MIT/LCS/TM-151, MIT Laboratory for Computer Science, 1980.
- [Val79] Leslie G. Valiant. The complexity of computing the permanent. *Theor. Comput. Sci.*, 8:189–201, 1979.

- [Val82] Leslie G. Valiant. A scheme for fast parallel communication. *SIAM J. Comput.*, 11(2):350–361, 1982.
- [VB81] Leslie G. Valiant and Gordon J. Brebner. Universal schemes for parallel communication. In *STOC*, pages 263–277. ACM, 1981.
- [Vui80] Jean Vuillemin. A unifying look at data structures. *Communications of the ACM*, 23(4):229–239, 1980.
- [VV86] Leslie G. Valiant and Vijay V. Vazirani. Np is as easy as detecting unique solutions. *Theor. Comput. Sci.*, 47(3):85–93, 1986.
- [Wil04] David Bruce Wilson. Mixing times of lozenge tiling and card shuffling Markov chains. *Annals of Applied Probability*, 14(1):274–325, 2004.
- [Wil06] Herbert S. Wilf. *generatingfunctionology*. A. K. Peters, third edition, 2006. Second edition is available on-line at <http://www.math.penn.edu/~wilf/DownldGF.html>.
- [WNC87] Ian H. Witten, Radford M. Neal, and John G. Cleary. Arithmetic coding for data compression. *Communications of the ACM*, 30(6):520–540, 1987.
- [Yao77] Andrew Chi-Chin Yao. Probabilistic computations: Toward a unified measure of complexity. In *18th Annual Symposium on Foundations of Computer Science*, pages 222–227. IEEE, 1977.
- [Zur03] Wojciech Hubert Zurek. Decoherence, einselection, and the quantum origins of the classical. *Rev. Mod. Phys.*, 75:715–775, May 2003.

Index

- (r_1, r_2, p_1, p_2) -sensitive, 134
- 2-coloring, 220
- 3-way cut, 383
- $G(n, p)$, 88
- Φ , 186
- ϵ -NNS, 133
- ϵ -PLEB, 133
- ϵ -nearest neighbor search, 133
- ϵ -point location in equal balls, 133
- \mathcal{F} -measurable, 43
- σ -algebra, 15, 42
 - generated by a random variable, 43
- τ_2 , 185
- d -ary cuckoo hashing, 122
- k -CNF formula, 219
- k -uniform hypergraph, 220
- k -universal, 112
- k -wise independence, 20
- t_{mix} , 160
- #DNF, 198, 201
- #P, 197
- #SAT, 197
- 0–1 random variable, 410
- 2-universal, 112

- absolute value, 241
- adapted, 82, 149
- adapted sequence of random variables, 140
- adaptive adversary, 104, 251
- Adleman’s theorem, 230

- adversary, 251
 - adaptive, 104, 251
 - oblivious, 104, 251
- advice, 230
- agreement, 252
- Aldous-Fill manuscript, 170
- algorithm
 - Deutsch’s, 246
 - distributed, 251
 - UCB1, 92
- algorithmic information theory, 110
- amplitude
 - probability, 236
- ancestor, 104
 - common, 105
- annealing schedule, 180
- anti-concentration bound, 96
- aperiodic, 162
- approximation algorithm, 214
- approximation ratio, 213
- approximation scheme
 - fully polynomial-time, 202
 - fully polynomial-time randomized, 197
- arithmetic coding, 200, 373
- arm, 91
- Atari 2600, 296
- augmenting path, 195
- average-case analysis, 2
- AVL tree, 100
- avoiding worst-case inputs, 12
- axioms of probability, 15

- Azuma's inequality, 78, 83
- Azuma-Hoeffding inequality, 78
- balanced, 99
- bandit
 - multi-armed, 91
- Bernoulli random variable, 32, 410
- Berry-Esseen theorem, 97
- biased random walk, 147
- binary search tree, 99
 - balanced, 99
- binary tree, 99
- binomial random variable, 32
- bipartite graph, 195
- bit-fixing routing, 76, 192
- Bloom filter, 123, 395
 - counting, 128
 - spectral, 128
- Bloomjoin, 127
- bounded difference property, 86
- bowling, 352
- bra, 240
- bra-ket notation, 239
- branching process, 225
- bubble sort, 287
- Byzantine processes, 251
- canonical paths, 188
- Catalan number, 100
- causal coupling, 182
- CCNOT, 244
- central limit theorem, 96
- chaining, 111
- Chapman-Kolmogorov equation, 154
- Chebyshev's inequality, 58
- Cheeger constant, 186
- Cheeger inequality, 187
- chromatic number, 88
- Chutes and Ladders, 407
- circuit
 - quantum, 236, 241
 - random, 237
- clause, 201, 213, 219
- clique, 209
- closed loop, 120
- CLT, 96
- CNOT, 244
- coin
 - weak shared, 254
- collision, 12, 110, 111
- coloring
 - hypergraph, 220
- common ancestor, 105
- common subsequence, 380
- complement, 15
- complex conjugate, 239
- computation
 - randomized, 238
- computation path, 230
- computation tree, 230
- concave, 54
- concentration bound, 58
 - coupon collector problem, 171
- conditional expectation, 6, 36
- conditional probability, 20
- conductance, 186
- congestion, 190
- conjugate
 - complex, 239
- conjugate transpose, 239
- conjunctive normal form, 213
- consensus, 252
- continuous random variable, 30
- contraction, 24
- controlled controlled NOT, 244
- controlled NOT, 244
- convex, 54
- convex function, 54
- Cook reduction, 198
- count-min sketch, 129

- countable additivity, 15
- countable Markov chain, 153
- counting Bloom filter, 128
- coupling, 160
 - causal, 182
 - path, 175
- Coupling Lemma, 161
- coupling time, 171, 172
- coupon collector problem, 48, 407
 - concentration bounds, 171
- course staff, xviii
- covariance, 34, 60
- crash, 252
- cryptographic hash function, 13
- cryptographically secure pseudorandom generator, 229
- cryptographically secure pseudorandom number generator, 9
- Cryptographically-secure pseudorandomness., 9
- cuckoo, 118
- cuckoo filter, 126
- cuckoo hash table, 118
- cuckoo hashing, 118
 - d -ary, 122
- cumulative distribution function, 30
- curse of dimensionality, 133
- cut
 - minimum, 24
- cylinder set, 16
- data stream computation, 129
- dead cat bounce, 292
- decoherence, 242
- decomposition
 - Doob, 84
- dense, 195
- dependency graph, 220
- depth
 - tree, 325
- derandomization, 228
- detailed balance equations, 164
- dictionary, 110
- discrete probability space, 15
- discrete random variable, 27, 30
- discretization, 138
- disjunctive normal form, 201
- distance
 - total variation, 158
- distributed algorithm, 251
- distributed algorithms, 13
- distributed computing, 251
- distribution, 30
 - geometric, 46
 - stationary, 157
- DNF formula, 201
- dominating set, 368, 375
- Doob decomposition, 84
- Doob decomposition theorem, 142
- Doob martingale, 86, 232
- Doob's martingale inequality, 150
- double record, 339
- dovetail shuffle, 175
- drift process, 142
- Dungeons and Dragons, 291
- eigenvalue, 182
- eigenvector, 182
- einselection, 242
- entropy, 200
- equation
 - Wald's, 35
- Erdős-Szekeres theorem, 354, 381
- error budget, 51
- Euler-Mascheroni constant, 8
- event, 14, 15
- events
 - independent, 18
- execution log, 224
- expectation, 31

- conditional, 36
- conditioned on a σ -algebra, 42
- conditioned on a random variable, 37
- conditioned on an event, 36
- iterated
 - law of, 40
 - linearity of, 31
- expected time, 2
- expected value, 2, 31
- expected worst-case, 1
- exponential generating function, 67
- false positive, 123
- filter
 - cuckoo, 126
- filtration, 43, 140
- financial analysis, 291
- Find the Lady, 2
- fingerprint, 13, 126
- fingerprinting, 13
- finite Markov chain, 153
- first passage time, 164
- Flajolet-Martin sketch, 356
- FPRAS, 197
- fractional solution, 214, 215
- Frobenius problem, 162
- fully polynomial-time approximation scheme, 202
- fully polynomial-time randomized approximation scheme, 197
- function
 - measurable, 27
- futile word search puzzle, 366
- gate
 - Toffoli, 244
- generated by, 43
- generating function
 - exponential, 67
 - probability, 32, 55
- geometric distribution, 46
- geometric random variable, 46
- graph
 - bipartite, 195
 - random, 27, 88
- Grover diffusion operator, 247
- Hadamard operator, 243
- handwaving, 8, 225
- harmonic number, 7
- hash function, 12, 111
 - perfect, 116
 - tabulation, 115
 - universal, 112
- hash table, 13, 110
 - cuckoo, 118
- hashing, 12
 - cuckoo, 118
 - hopscotch, 123
 - locality-sensitive, 133
- heap, 102
- heavy hitter, 129
- high probability, 50
- high-probability bound, 1
- Hoare's FIND, 48, 406
- Hoeffding's inequality, 78
- Hoeffding's lemma, 78
- homogeneous, 153
- hopscotch hashing, 123
- hypercube network, 75, 88
- hyperedge, 220
- hypergraph, 220
 - k -uniform, 220
- inclusion-exclusion formula, 18
- independence, 18
 - of events, 18
 - of sets of events, 20
 - pairwise, 20

- independent, 18, 20
- independent events, 18
- independent random variables, 30
- independent set, 209
- index, 111
- indicator random variable, 28
- indicator variable, 7
- inequality
 - Azuma's, 78, 83
 - Azuma-Hoeffding, 78
 - Chebyshev's, 58
 - Cheeger, 187
 - Doob's martingale, 150
 - Hoeffding's, 78
 - Markov's, 52
 - McDiarmid's, 86, 87
- inner product, 239
- instructor, xviii
- integer program, 214
- irreducible, 162
- iterated expectation
 - law of, 40
- Iverson bracket, 28
- Iverson notation, 28

- Johnson-Lindenstrauss lemma, 134
- joint distribution, 30
- joint probability mass function, 30

- Karger's min-cut algorithm, 24
- Karp-Upfal-Wigderson bound, 407
- ket, 240
- key, 99, 111
- KNAPSACK, 202
- Kolmogorov extension theorem, 16

- Las Vegas algorithm, 10
- law of iterated expectation, 40
- law of total probability, 5, 21
- lazy Markov chain, 162
- leader election, 13, 254

- left spine, 106
- lemma
 - coupling, 161
 - Johnson-Lindenstrauss, 134
 - Lovász local, 217
 - Yao's, 46
- length, 284
- linear program, 214
- linearity of expectation, 6, 31
- Lipschitz function, 181
- literal, 219
- Littlewood-Offord problem, 97
- load balancing, 13
- load factor, 111
- locality-sensitive hashing, 133
- loop
 - closed, 120
- Lovász Local Lemma, 217

- magnitude, 241
- Markov chain, 152, 153
 - lazy, 162
- Markov chain Monte Carlo, 152
- Markov process, 153
- Markov's inequality, 52
- martingale, 82, 140
 - Doob, 232
 - vertex exposure, 88
- martingale difference sequence, 82
- martingale property, 140
- matching, 192
- matrix
 - stochastic, 238
 - transition, 154
 - unitary, 242
- max register, 259
- maximum cut, 212
- McDiarmid's inequality, 86, 87
- measurable, 30, 43
- measurable function, 27

- measurable sets, 15
- measure
 - probability, 14
- measurement, 238
- memoryless, 153
- message-passing, 251
- method of bounded differences, 86
- method of conditional probabilities, 232
- Metropolis, 168
- Metropolis-Hastings
 - convergence, 180
- Metropolis-Hastings algorithm, 168
- Miller-Rabin primality test, 4
- min-cut, 24
- minimum cut, 24
- mixing
 - rapid, 172
- mixing rate, 185
- mixing time, 152, 160
- moment, 67
- moment generating function, 67
- Monte Carlo algorithm, 4, 10
- Monte Carlo simulation, 10
- multi-armed bandit, 91
- multigraph, 24

- nearest neighbor search, 133
- network
 - sorting, 208
- NNS, 133
- no-cloning theorem, 244
- non-uniform, 231
- norm, 241
- normal form
 - conjunctive, 213
 - disjunctive, 201
- notation
 - bra-ket, 239
- number
 - Catalan, 100
 - number P, 197

- objective function, 214
- oblivious adversary, 104, 251
- open addressing, 111
- operator
 - Grover diffusion, 247
 - Hadamard, 243
- optimal solution, 214
- optional stopping theorem, 142
- outcomes, 15

- pairwise independence, 61
- pairwise independent, 20
- paradox
 - St. Petersburg, 33
- parallel computing, 251
- partition, 15
- path
 - simple, 280
- pending operation, 251
- perfect hash function, 116
- perfect matching, 195
- perfect matchings, 198
- period, 162
- periodic, 156
- permanent, 198
- permutation routing, 75
- Perron-Frobenius theorem, 182
- pessimistic estimator, 233
- phase, 243
- physical randomness, 8
- Physical randomness., 8
- pivot, 5
- PLEB, 133
- point location in equal balls, 133
- point query, 129
- points, 15
- Poisson trial, 410

- polynomial-time hierarchy, 198
- preference, 253
- primality test
 - Miller-Rabin, 4
- probabilistic method, 13, 207
- probabilistic recurrence, 89
- probability, 15
 - conditional, 20
 - high, 50
 - stationary, 157
- probability amplification, 228, 230
- probability amplitude, 236
- probability generating function, 32, 55
- probability mass function, 28
- probability measure, 14
- probability of A conditioned on B , 20
- probability of A given B , 20
- probability space, 14
 - discrete, 15
- probability theory, 14
- probing, 111
- product
 - inner, 239
 - tensor, 237
- pseudorandom generator, 228
 - cryptographically secure, 229
- pseudorandom number generator, 9
- puzzle
 - word search, 365
 - futile, 366
- quantum bit, 236
- quantum circuit, 236, 241
- quantum computing, 236
- qubit, 236
- query
 - point, 129
- QuickSelect, 48, 406
- QuickSort, 5, 89
- radix tree, 355
- RAM, 1
- random circuit, 237
- random graph, 27, 88
- random structure, 13
- random variable, 2, 14, 27
 - Bernoulli, 32
 - binomial, 32
 - discrete, 27, 30
 - geometric, 46
 - indicator, 28
- random walk, 145
 - biased, 147
 - unbiased, 145
 - with one absorbing barrier, 146
 - with two absorbing barriers, 145
- random-access machine, 1
- randomized computation, 238
- randomized rounding, 204, 214
- randomness
 - physical, 8
- range coding, 373
- rapid mixing, 172
- record, 339
 - double, 339
- red-black tree, 100
- reducible, 156
- reduction, 198
- register, 251
 - max, 259
- regret, 91
- rejection sampling, 168, 199, 373
- relax, 214
- relaxation, 214
- relaxation time, 185
- reversibility, 158
- reversible, 164, 243
- right spine, 106
- ring-cover tree, 134
- root, 4, 99

- rotation, 100
 - tree, 100
- routing
 - bit-fixing, 76, 192
 - permutation, 75
- run, 284, 378
- sampling, 10, 12
 - rejection, 168, 199, 373
- SAT, 399
- satisfiability, 219
- satisfiability problem, 213
- satisfy, 213
- scapegoat tree, 100
- search tree
 - balanced, 99
 - binary, 99
- second-moment method, 61
- seed, 9, 228
- separate chaining, 111
- SHA, 111
- shared coin
 - weak, 254
- shared memory, 251
- sharp P, 197
- sifter, 56, 254, 257
- simple path, 280
- simplex method, 214
- simulated annealing, 180
- simulation
 - Monte Carlo, 10
- sketch, 129
 - count-min, 129
 - Flajolet-Martin, 356
- sort
 - bubble, 287
- sorting network, 208
- Space Invaders, 297
- spanning tree, 392
- spare, 352
- spectral Bloom filter, 128
- spectral theorem, 183
- spine
 - left, 106
 - right, 106
- sports reporting, 291
- sprite, 296
- staff, xviii
- standard deviation, 96
- state space, 153
- state vector, 237
- stationary distribution, 152, 154, 157
- stationary probability, 157
- Statistical pseudorandomness., 9
- stochastic matrix, 154, 238
- stochastic process, 153
- stopping time, 140, 141
- strike, 352
- strong law of large numbers, 96
- strongly k -universal, 112
- strongly 2-universal, 112
- submartingale, 82, 84, 141
- subtree, 99
- supermartingale, 82, 84, 89, 141
- symmetry breaking, 13
- tabulation hashing, 115
- tensor product, 237
- termination, 252
- theorem
 - Adleman's, 230
 - Berry-Esseen, 97
 - Doob decomposition, 142
 - Erdős-Szekeres, 354
 - Kolmogorov extension, 16
 - no-cloning, 244
 - optional stopping, 142
 - Perron-Frobenius, 182
 - spectral, 183
 - Toda's, 198

- Valiant's, 197
- third moment, 97
- Three-card Monte, 2
- time
 - coupling, 171, 172
 - expected, 2
 - mixing, 160
 - relaxation, 185
 - stopping, 140, 141
- time-reversed chain, 167
- Toda's theorem, 198
- Toffoli gate, 244
- total path length, 101
- total variation distance, 158
- transformation
 - unitary, 242
- transition matrix, 154
- transition probabilities, 154
- transpose
 - conjugate, 239
- treap, 102, 353
- tree
 - AVL, 100
 - binary, 99
 - binary search, 99
 - cartesian, 102
 - radix, 355
 - red-black, 100
 - ring-cover, 134
 - rotation, 100
 - scapegoat, 100
 - witness, 224
- tree rotation, 100
- truncation, 143

- UCB1 algorithm, 92
- unary, 138
- unbiased random walk, 145
- uniform, 231
- unitary matrix, 242
- unitary transformation, 242
- universal hash family, 112
- unranking, 199
- upper confidence bound, 92

- Valiant's theorem, 197
- validity, 252
- variance, 58
- vector
 - state, 237
- vector space, 41
- vertex exposure martingale, 88

- wait-free, 252
- wait-free shared memory, 251
- Wald's equation, 35, 149
- weak shared coin, 254
- with high probability, 50
- witness, 4, 231
- witness tree, 224
- word search puzzle, 365
 - futile, 366
- worst-case analysis, 2

- xxHash, 111

- Yao's lemma, 46