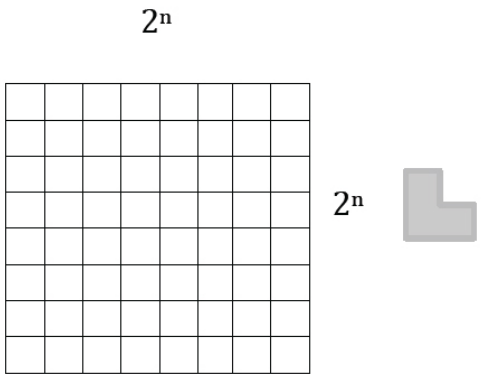
7. Tile That Courtyard, Please

We shape our buildings; thereafter they shape us. — Winston Churchill.

Programming constructs and algorithmic paradigms covered in this puzzle: List comprehension basics. Recursive Divide-and-Conquer search.

Consider the following tiling problem. We have a courtyard with $2^n \times 2^n$ squares and we need to tile the courtyard using L-shaped tiles or trominoes. Each trominoe consists of three square tiles attached to form an L shape as shown below.

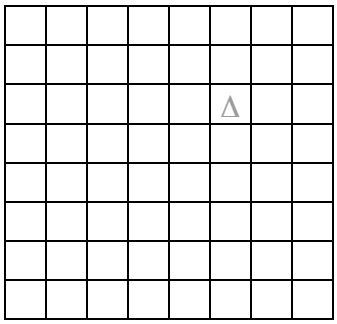


Can this be done without spilling over the boundaries, breaking a tromino or having overlapping trominoes? The answer is no, simply because $2^n \times 2^n = 2^{2n}$ is not divisible by 3, only by 2. However, if there is one square that can be left untiled, then $2^{2n} - 1$ is divisible by 3. Can you show this?[1]

We, therefore, have hope of properly tiling a $2^n \times 2^n$ courtyard with one square that we can leave untiled because, for example, there is a statue of your favorite President on it. We'll call this square that can be left untiled the missing square.

[1] Since this is not a book of mathematical puzzles we'll tell you! $2^{2n} - 1$ can be written as $(2^n - 1)(2^n + 1)$. 2^n is clearly not divisible by 3. And that means that either $2^n - 1$ is divisible by 3 or $2^n + 1$ is divisible by 3.

Is there an algorithm that tiles any $2^n \times 2^n$ courtyard with one missing square in an arbitrary location? As an example, below is a $2^3 \times 2^3$ where the missing square is marked ∆. Does the location of the missing square matter?

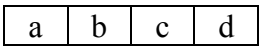


The answer is yes. We will describe and code a *recursive* Divide-and-Conquer algorithm that can tile a courtyard with $2^n \times 2^n$ squares with one missing square in an arbitrary location. To help you understand how recursive Divide-and-Conquer works, we will first describe how it is used in Merge Sort, a popular sorting algorithm.

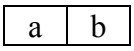
Merge Sort

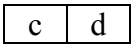
We can perform sorting using the elegant Divide-and-Conquer Merge Sort. Here's how Merge Sort works.

Suppose we have a list as shown here that needs to be sorted in ascending order:

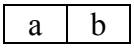


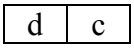
We divide it into two equal-sized sublists:





Then, we sort the sublists recursively. At the base case of the recursion, if we see a list of size 2, we simply compare the two elements in the list and swap if necessary. Let's say that $a < b$, and $c > d$. Since we want to sort in ascending order, after the two recursive calls return we end up getting:





Now we come back to the first (or topmost) call to sort and our task is to merge the two *sorted* sublists into one sorted list. We do this using a *merge* algorithm, which simply compares the first two elements of the two sublists, repeatedly. If $a < d$, then it first puts a into the merged (output) list and effectively removes it from the sublist. It leaves d where it was. It then compares b with d. Let's say that d is smaller. Merge puts d into the output list next to a. Next, b and c are compared. If c is smaller, c is placed next into the output list, and finally b is placed. The output will be:

Note that if the number of elements is odd, the subarrays will not be equal in size but differ in size by 1.

Below is code for Merge Sort.

```
 1.   def mergeSort(L):
 2.       if len(L) == 2:
 3.           if L[0] <= L[1]:
 4.               return [L[0], L[1]]
 5.           else:
 6.               return [L[1], L[0]]
 7.       else:
 8.           middle = len(L)//2
 9.           left = mergeSort(L[:middle])
10.           right = mergeSort(L[middle:])
11.           return merge(left, right)
```

Lines 2-6 correspond to the base case where we have a list of two elements and we place them in the right order. If the list is longer than two in length, we split the list in two (Line 8), and make two recursive calls, one on each sublist (Lines 9-10). We use list slicing; `L[:middle]` returns the part of the list `L` corresponding to `L[0]` through `L[middle-1]`, and `L[middle:]` returns the part of the list corresponding to `L[middle]` to `L[len(L)-1]`, so no elements are dropped. Finally, on Line 11, we call `merge` on the two sorted sublists and return the result.

All that remains is the code for `merge`.

```
 1.   def merge(left, right):
 2.       result = []
 3.       i, j = 0, 0
 4.       while i < len(left) and j < len(right):
 5.           if left[i] < right[j]:
 6.               result.append(left[i])
 7.               i += 1
 8.           else:
 9.               result.append(right[j])
10.               j += 1
11.       while i < len(left):
12.           result.append(left[i])
13.           i += 1
14.       while j < len(right):
15.           result.append(right[j])
16.           j += 1
17.       return result
```

Initially, `merge` creates a new empty list `result` (Line 2). There are three **`while`** loops in `merge`. The first one is the most interesting, which corresponds to the general case when the two sublists are both nonempty. In this case, we compare the current first elements of each of the sublists (represented by counters `i` and `j`), pick the smaller one, and increment the counter for the sublist whose element we selected to place in result. The first **`while`** loop terminates when either of the sublists becomes empty.

When one of the sublists is empty, we simply append the remaining elements of the nonempty sublist to the result. The second and third **`while`** loops correspond to the left sublist being nonempty and the right one being nonempty.

Merge Sort Execution and Analysis

Suppose we have this input list for Merge Sort:

```
inp = [23, 3, 45, 7, 6, 11, 14, 12]
```

How does the execution proceed? The list is split into two:

```
[23, 3, 45, 7]              [6, 11, 14, 12]
```

and the left list is sorted first, and the first step is to split it in two:

```
[23, 3]    [45, 7]          [6, 11, 14, 12]
```

Each of these 2-element lists is sorted in ascending order:

```
[3, 23]    [7, 45]          [6, 11, 14, 12]
```

The sorted lists of two elements each are merged into a sorted result:

```
[3, 7, 23, 45]              [6, 11, 14, 12]
```

Next, it looks at the right list and splits it in two:

```
[3, 7, 23, 45]              [6, 11]    [14, 12]
```

Each of the right sublists is sorted:

```
[3, 7, 23, 45]              [6, 11]    [12, 14]
```

The sorted sublists of two elements each are merged into a sorted result:

```
[3, 7, 23, 45]              [6, 11, 12, 14]
```

And finally, the two sorted sublists of 4 elements each are merged:

```
[3, 6, 7, 11, 12, 14, 23, 45]
```

Exercises

Exercise 1: You are given a $2^n \times 2^n$ courtyard with 4 tiles missing. There are (at least) two cases where this is possible:

1. When the four missing tiles are in four different quadrants.
2. When any three of the four tiles are such that you can tile them using a tromino.

Write a procedure that will determine if you can tile the courtyard or not using `recursiveTile`. The procedure can just return **`True`** or **`False`**, given `n`, and a four-element list of missing square coordinates.

Puzzle Exercise 2: Suppose you have a two-dimensional list or matrix `T` as shown below, where all rows and all columns are sorted. Devise and implement a binary search algorithm that works for lists such as T. You can assume that all elements are unique as in the example below.

```
T = [[ 1,  4,  7, 11, 15],
     [ 2,  5,  8, 12, 19],
     [ 3,  6,  9, 16, 22],
     [10, 13, 14, 17, 24],
     [18, 21, 23, 26, 30]]
```

Here's the strategy you should use: Guess that the value is at position `i, j` and think of what it means if the value is less than `T[i][j]`, or if the value is greater than `T[i][j]`. For example, if we are searching for `21` and we compare with `T[2][2] = 9`, we know that `21` cannot be in the `T[<= 2][<= 2]` upper left quadrant because all values in that quadrant are less than `9`. However, `21` could be in any of the three other quadrants, i.e., the bottom left `T[> 2][<= 2]` quadrant, which it is in this example, or the top right `T[<= 2][> 2]` or bottom right `T[> 2][> 2]` quadrants in different examples.

You can always eliminate one of the four quadrants in your two-dimensional binary search. You will have to make recursive calls on the other three quadrants.

Puzzle Exercise 3: It is a natural question to ask as to whether the two-dimensional binary search algorithm of Exercise 2 is the best possible algorithm. Let's look at `T` again:

```
T = [[ 1,  4,  7, 11, 15],
     [ 2,  5,  8, 12, 19],
     [ 3,  6,  9, 16, 22],
     [10, 13, 14, 17, 24],
     [18, 21, 23, 26, 30]]
```

Suppose we want to find out if element `13` exists in the two-dimensional list or matrix T. Here's the strategy you should use: Start with the top right element. If the element is

smaller than the element you are looking for, you can eliminate the entire first row and move one position down. If the element is larger, you can eliminate the entire last column and move one position left. Obviously, if the element is the top right element, you can stop.

The neat thing about the above strategy is that it either eliminates a row or a column in each step. So you will find the element you are searching for in at most 2n steps for a n × n matrix, or you will determine that it does not exist. Code the above algorithm by making recursive calls on the appropriate submatrix (with one less row or one less column). In our example above, we will move from `15` to `11` to `12` to `16` to `9` to `14` to `13`.

6.S095 Programming for the Puzzled
January IAP 2018