Typical Answers

2.1 A program could be fortuitously correct, i.e., even though it is faulty, it produces correct results for some inputs. Give two example code segments and circumstances under which they become fortuitously correct.

Assume that Program 1.1 is the correct program, which is repeated below for reference.

Program 1.1

```
main ()

{
int i, j, k, match;

cin >> i >> j >> k;
cout << i << j << k;
if (i <= 0 || j <= 0 || k <= 0
|| i+j <= k || j+k <= i || k+i <= j)
match = 4;
else if !(i == j || j == k || k == i)
match = 3;
else if (i != j || j != k || k != i)
match = 2;
else match = 1;
cout << match << endl;
}
```

We can easily construct another program listed below that is fortuitously correct for only one input, viz., i = j = k = 5.

```
main ()

{
int i, j, k, match;

cin >> i >> j >> k;
cout << i << j << k;
if (i == 5 || j == 5 || k == 5)
     match = 3;
else match = 9;
cout << match << endl;
}
```

We construct another listed below which is fortuitously correct for many inputs.

```
main ()

{
int i, j, k, match;

cin >> i >> j >> k;
cout << i << j << k;
if (i <= 0 || j <= 0 || k <= 0
     || i+j <= k || j+k <= i || k+i <= j)
match = 4;
else if !(i == j || j == k || k == i)
match = 3;
else if (i != j || j != k || k = i)
match = 2;
else match = 1;
cout << match << endl;
}
```

[Explanation]

What is a fortuitously correct program?

Recall that, by definition, a program is said to be correct if, upon an execution, it will produce a correct output for any element in the intended input domain. A program is said to be incorrect, in error, or faulty, if it will produce an incorrect output for one or more element in the input domain. The point is that an incorrect program can produce a correct output for some input.

It is safe to say that all large real-world programs are not completely free of any error. In the subject of software testing, therefore, we will be dealing with programs that are incorrect by the definition given above but nevertheless would produce correct outputs for most test cases we can find. So instead of saying an incorrect program can produce correct output, many writers choose to say a program can be fortuitously correct.

The word *fortuitous* is synonymous to the word *accidental* or *unexpected*. If we are talking about a random program that happens to produce a correct output for some input, it is quite appropriate to say that the program is unexpectedly correct. But if we were talking about a program that is produced by a competent programmer, it would be rather strange to say that the program is unexpectedly correct!

Perhaps it is more appropriate to use the term *partially correct* to characterize such programs because it is correct only for some, but not all, elements in the input domain. Unfortunately, the term "partially correct" has already been taken for another purpose. In the subject of proving program correctness, a correctness proof should consist of two parts: it should show that the program would produce correct outputs, and it should show that the program would terminate. The correctness asserted by a proof is said to be partial if it does not include termination. In other words, a partially correct program would produce correct outputs if it ever terminates, but it may not terminate (see Section 6.5).

Thus, when we say a program is fortuitously correct, we do not mean that the program is accidentally (or unexpectedly) correct. Rather, we mean the correctness is incomplete in the sense that the program would produce correct outputs for a proper non-empty subset of the intended input domain.

How do we go about to construct fortuitously correct program?

By the definition given above, a program that will produce one and only one correct input-output pair is a fortuitously correct program. Use whatever information available to find a correct input-output pair, and write a program to implement that.

To construct one that would produce many correct input-output pairs, write a program to implement the specification as usual. Make sure that it will produce some correct input-output pairs. Then make a simple change to the program text to make sure that it will not be correct for the entire input domain.

2.2 A common problem in applying the test-case selection methods discussed in this chapter is to find elements of the input domain that will satisfy a path predicate, such as $b - a > 0.01 \wedge b + 2a \geq 6 \wedge 0.01 \geq 2(b - a) / 3$. In general, it is tedious and time consuming to do so. To get a feel for this problem, time yourself to see how long it takes for you to find two different sets of non-zero values for variables a and b that satisfy this predicate.

We need to make assignments (of values) to a and b. The first and the third predicates constrain the relative values of a and b, i.e.,

$b - a > 0.01$
$b - a \leq 0.015$

and the second component predicate constrains the absolute values of a and b, i.e.,

$b + 2a \geq 6$

So this predicate can be satisfied by the assignment a ← 2.0 and b ← 2.012, and by another assignment a ← 3.0 and b ← 3.014, for example.

2.3 Survey the field to see if there are any algorithms or software packages that can be used to find inputs that satisfy path predicates like the one given in the preceding problem.

[Hint] There is a branch of mathematics called linear programming that deals with equalities and inequalities. Check it out to see how much of that is applicable..

2.4 There are many software testing tools on the market. Do any of them offer direct help to the solution of Problem 2? If so, describe their technical capabilities and limitations.

[Hint] You can start by doing a Google search on software testing tools.

2.5 Is it possible for the same variable to have a p-use and c-use in the same statement in a C++ program?

It is possible because the assignment operator can be used in a predicate (logical expression). Example:

```
while ( isdigit ( ch = s[i++] && ch != '\0' ));
```

Here variable ch has a c-use because it is on the left-hand side of an assignment operator, and a p-use because it is a part of the loop predicate.

This language feature complicates the analysis tool needed in constructing test cases.

2.6 Consider a program with the path domain D_i in a two dimensional space defined by the following predicates:

x - 2 >= 0
x + y > 0
2x - y >= 0

Find three test cases (i.e., three on-off-on points) needed for testing the correctness of the boundary defined by the predicate x-2 >= 0, assuming that the smallest real number on the machine is 0.001.

x ← 2.000 y ← 2.500
x ← 1.999 y ← 0.000

x ← 2.000 y ← -1.500

[Note] The boundary defined by x-2 >= 0 is a closed boundary. The "off" point has to be outside, i.e., to the left of the borderline.

2.7 The following assumptions were made in developing the domain strategy testing method. Explain why these assumptions are necessary, and what would happen if these assumptions were not satisfied: Assumption (f): the path corresponding to each adjacent domain computes a different function. Assumption (g): functions defined in two adjacent subdomains yield different values for the same test point near the border.

In domain-strategy testing the tester needs be able to tell which function is used to compute the test result. Assumption (f) represents the necessary condition to make this possible. Since two different functions may produce the same values for some inputs, Assumption (g) represents a sufficient condition to ensure that different functions produce different values for each point near the domain boundary.

2.8 Given a program and a set of its mutants, how to select test cases so that the number of test cases required for differentiating all mutants from the original program is minimal?

Choose a test case to exercise each feasible execution path once and only once. If a test case would not differentiate the mutant from the original program, find another to traverse the same path until it does. At that point, all mutants should have been differentiated. If a mutant could not be differentiated, it is quite possible that that mutant is logically equivalent to the original program. Prove (or disprove) that it is.

2.9 Study old software RFQs (Requests for Quotation) issued by the federal government or industrial concerns in the United States to see what are the real-world software test requirements prevalent in the last decade.

3.1 Consider the program specification given below:

GW Ice-Cream Warehouse receives and distributes ice-cream in one-gallon cans. Information about each shipment received or distributed is entered by a clerk into a computerized database system.

A program is written to produce a weekly management report showing the net change in inventory of each item during the week. As depicted below, the report begins with a heading, which is followed by the report body. The report concludes with a line indicating the number of items that have a net change in inventory. Note that each item is identified by a string of no more than 16 characters.

GW Ice-Cream Co.
Weekly Management Report

Item	Net Change (gal.)
BingCherry	-33
ChocolateMint	+46
FrenchVanilla	-77
LemonLime	-46
NuttyCocktail	+33
Sundae	+125
# of items changed =	6

In practice, this program is to obtain all needed data from the database system. It is assumed that, with an appropriate retrieval command, this database system is capable of producing all pertinent data as a stream of records, each of which contains three fields: the item name (a string), the type of shipment (R for received, D for distributed), and the quantity of that shipment (a non-negative integer). The fields in a record are separated by a single blank character. The records in a stream are in alphabetical order by item names. A record is terminated by a carriage return, and a stream is terminated by a record with a single period as the item name.

Use the error-guessing method to find a set of test cases for the program.

Choose test cases that represent the following events:

- System startup (i.e., the program being made to accept input from the data base for very first time)
- Routine system startup (i.e. the program invoked for a production run)
- A production run disrupted by a power failure or accidental shutdown
- Illegal inputs, such as noise or inputs from a malfunctioning database
- Empty input file
- An excessively long input file (longer than that will ever be used in a production run)
- Excessively large entries (that cause the output to overflow)
- Abnormal input conditions, such as distributing more items then received
- Startup after a holiday

- Startup after the store has been closed for an extended period of time

Note that the required action of this program is not only dependent on the current input but also the preceding input as well. Thus the test set should include two consecutive inputs of different kinds, such as two blank entries, two negative entries, two plus entries, two identical entries, two terminating symbols, etc.

3.2 Use the predicate testing method to find a set of test cases for the program specified below:

Write a program that takes three positive integers as input and determine if they represent three sides of a triangle, and if they do, indicate what type of triangle it is. To be more specific, it should read three integers and set a flag as follows:
if they represent a scalene triangle then set it to 1,
if they represent an isosceles triangle then set it to 2,
if they represent an equilateral triangle then set it to 3,
and if they do not represent a triangle then set it to 4.

The inputs are triples of integers of the form <x, y, z>.

We can find four predicates in the program specification, viz.,

TRIG: <x, y, z> forms the three sides of a triangle.
EQUI: <x, y, z> forms the three sides of an equilateral triangle.
ISOS: <x, y, z> forms the three sides of an isosceles triangle.
SCAL: <x, y, z> forms the three sides of an scalene triangle.

For TRIG we may select <2, 3, 4> and <1, 2, 5>, for example.
For EQUI we may select <6, 6, 6> and <4, 5, 7>.
For ISOS we may select <5, 7, 5> and <5, 6, 7>.
For SCAL we may select <3, 4, 5> and <3, 3, 3>

3.3 Use the boundary-values analysis method to find a set of test cases for the program specified below:

Write a program to compute the cubic root of a real number by using the Newton's method. Set q to 1.0 initially, and compute the value of the expression (2.0 * q + r / (q * q)) / 3.0, where r is a given real number. If the value of this expression is almost the same as that of q, i.e., if the difference is less than or equal to d, where d is a given constant considerably smaller than q, return this value as the cube root of r. Otherwise, assign the value of this expression as the next value of q and repeat the computation.

There are two input variables of type *real*: r and d; and one output variable: d.

3.4 Apply the subfunction testing method to find a set of test cases for the program specified in Exercise 3.2.

There are four subfunctions in the specification given in Exercise 3.2.

The first subfuction set the flag to 1. It is defined in the subdomain defined by TRIG ∧ SCAL. We may choose <4, 5, 6>, for example, to test this subfunction.

The second subfuction set the flag to 2. It is defined in the subdomain defined by TRIG ∧ ISOS. We may choose <7, 5, 5>, for example, to test this subfunction. The third subfuction set the flag to 3. It is defined in the subdomain defined by TRIG ∧ EQUI. We may choose <8, 8, 8>, for example, to test this subfunction.

The fourth subfuction set the flag to 4. It is defined in the subdomain defined by ¬TRIG. We may choose <-4, 5, 0>, for example, to test this subfunction.

3.5 The method of predicate testing can be thought of as the counter part of branch testing among the code-based test-case selection methods. What are the counterparts of boundary-value analysis and subfunction testing?

The method of boundary-value analysis is the counter part of domain strategy testing, and the method of subfunction testing is the counter part of path testing.

3.6 Critics of the code-based test-case selection methods often argue that, if the programmer failed to implement a certain part of the specification, no test cases will be selected to exercise that part of the program if the test cases are selected based on the source code. Can you refute or support that argument?

This criticism is valid if omission of certain parts in a program causes certain elements in the input domain to be precluded from being selected as a test case.

All code-based test-case selection methods, with one exception that will talk about in a moment, use a certain attribute of the source code to partition the input domain into subdomains, and then arbitrarily select at least one element from each subdomain as a test case. As such, omission of a certain part in the program would not preclude any elements in the input domain from being selected as a test case.

To see why that is so, it is instructive to consider Specification 3.1, which can be implemented by the code listed below.

```
int  i, j, k, flag;
```

```
read(i, j, k);
if (i ≤ 0 ∨ j ≤ 0 ∨ k ≤ 0 ∨ i+j ≤ k ∨ j+k ≤ i ∨ k+i ≤ j) flag := 4;
      else if (i = j ∧ j = k ∧ k = i) flag := 3;
      else if (i = j ∨ j = k ∨ k = i) flag := 2;
      else flag := 1;
write(flag);
```

Now if the programmer somehow forgot to implement the part "if they represent an equilateral triangle then set the flag to 3" the program becomes

```
int  i, j, k, flag;

read(i, j, k);
if (i ≤ 0 ∨ j ≤ 0 ∨ k ≤ 0 ∨ i+j ≤ k ∨ j+k ≤ i ∨ k+i ≤ j) flag := 4;
      else if (i = j ∨ j = k ∨ k = i) flag := 2;
      else flag := 1;
write(flag);
```

This faulty program partitions the input domain into three subdomains instead of four. If any code-based method other than the domain strategy method is used, nothing will prevent inputs such as <5, 5, 5> from being selected, among others, as the test cases, and thus will cause the fault to be revealed.

The method of domain strategy testing is the only exception. Although the method in effect partitions the input domain into subdomains, it does not select test cases from each subdomain arbitrarily. Only those elements located on, or very close to a domain borderline are eligible to be selected. If a certain part of the program specification is not implemented on the program, some borderlines may not exist, and therefore certain test cases may not be selected to reveal the faults.

J.C. Huang 4/11/08 10:39 PM
Formatted: Left, Indent: Left: 0", Hanging: 0.38"

4.1 A program was written to determine if a given year in the Gregorian calendar is a leap year. The well-known part of the rule, stipulating that it is a leap year if it is divisible by 4, is correctly implemented in the program. The programmer, however, is unaware of the exceptions: A centenary year, although divisible by 4, is not a leap year unless it is also divisible by 400. Thus, while year 2000 is a leap year, year 1800 and 1900 are not. Determine if the following test-case selection criteria are reliable or valid.

(a) $C_1(T) \equiv (T = \{1, 101, 1001, 10001\})$ Reliable? _____ Valid? _____

(b) $C_2(T) \equiv (T = \{t \mid 1995 \le t \le 2005\})$ Reliable? _____ Valid? _____

(c) $C_3(T) \equiv (T = \{t \mid 1895 \le t \le 1905\})$ Reliable? _____ Valid? _____

(d) $C_4(T) \equiv (T = \{t\} \wedge t \in \{400, 800, 1200, 1600, 2000, 2400\})$ Reliable? _____ Valid? _____

(e) $C_5(T) \equiv (T = \{t, t+1, t+2, t+3, t+4\} \wedge t \in \{100, 200, 300, 400, 500\})$
Reliable? _____ Valid? _____
(f) $C_6(T) \equiv (T = \{t, t+1, t+2, \ldots, t+399\} \wedge t \in D$, D being the input domain.
Reliable? _____ Valid? _____
(g) $C_7(T) \equiv (T = \{t_1, t_2, t_3\} \wedge t_1, t_2, t_3 \in D)$ Reliable? _____ Valid? _____

(a) Reliable but not valid
(b) Reliable and valid
(c) Reliable but not valid
(d) Reliable and valid
(e) Not reliable but valid
(f) Reliable and valid
(g) Not reliable but valid

4.2 If the Second Principle of test-case selection (ref. Chapter 1) is used to develop a test-case selection method, then answer to the question of when to stop testing is obvious: stop testing when all components have been exercised at least once during the test. Identify all the methods to which this answer is not applicable.

J.C. Huang 4/11/08 10:39 PM
Formatted: Left, Indent: Left: 0", Hanging: 0.38"

Program mutation An input could cause a mutant to produce an output identical to that of the original program. In the mutant has to be test-executed with additional test cases until it produces a different output, or if the mutant is proved to be logically equivalent to the original program.

Domain strategy testing Each segment of the domain border has to be exercised with three test cases.

J.C. Huang 4/11/08 10:39 PM
Formatted: Left, Indent: Left: 0.38", First line: 0"

4.3 What are the similarities and differences between traditional programs and object-oriented programs as far as applications of the first and second principles of test-case selection are concerned?

Similarity: Both have well-defined execution paths in the program, and two inputs are computationally loosely coupled if they cause different paths to be traversed during test-execution.

Difference: The source code is organized in different way. The syntax and semantics of an object-oriented programming language is more complex than that of a traditional procedural language.

In other words, they are similar in that computations are essentially performed by similar sequences of steps (statements), but they are different in that the statements in the programs are sequenced in different ways.

J.C. Huang 4/11/08 10:39 PM
Formatted: Left, Indent: Left: 0.38", First line: 0"

4.4 Enumerate the factors you must consider in choosing between code inspection and testing, and between debug and operational testing.

Code inspection and testing:

- Technical capability of available personnel: Although the process of code inspection has been proved to be effective in fault detection, not everyone can do code inspection well. It requires an effective moderator, a number of capable and well-motivated participants to make an inspection session productive. Otherwise, choose testing.
- The cost would be high: What is the required level of reliability? Is the cost/benefit ratio justifiable?

Debug and operational testing:

- Availability of a valid operational profile: An operational profile may not be accurate, and may shift in time. Without a valid operational profile, the test may not reveal important faults, and may not provide an accurate estimation of the reliability.
- Availability of qualified debug testers: If the tester is not well versed in test-case selection methods, choose to do operational testing.

4.5 Identify tasks that are common in using different methods for test-case selection, and discuss the degree of difficulty in automating each.

J.C. Huang 4/11/08 10:39 PM
Formatted: Left, Indent: Left: 0", First line: 0"

Perhaps the most common task is to find one or more input that will cause a certain part of the program to be exercised during the test.

In essence we need to find an assignment to an input variable that will satisfy a path predicate. If the predicate is associated with an infeasible path, we will have to prove that the predicate is a contradiction. That is the same as proving its negated form is a theorem. So for a large, complex program, it would be as difficult as doing theorem proving.

5.1 Identify three feasible execution paths in Program 5.1 and determine for each (a) the condition under which it will be traversed, and (b) the computation it performs in the process. Simplify your answers to the extent possible.

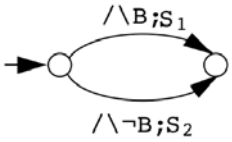
```
i = 0;
while (isspace(s[i]))
   i = i + 1;
if (s[i] == '-')
   sign = -1;
```

```
else
   sign = 1;
if (s[i] == '+' || s[i] == '-')
   i = i + 1;
n = 0;
while (isdigit(s[i])) {
   n = 10 * n + (s[i] - '0');
   i = i + 1;
}
return sign * n;
```

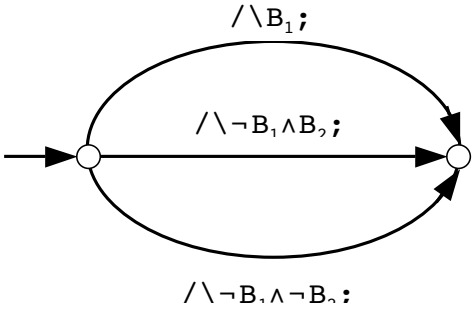
```
i = 0;
while (isspace(s[i]))
   i = i + 1;
if (s[i] == '-')
   sign = -1;
else
   sign = 1;
if (s[i] == '+' || s[i] == '-')
   i = i + 1;
n = 0;
while (isdigit(s[i])) {
   n = 10 * n + (s[i] - '0');
   i = i + 1;
}
return sign * n;
```

```
i = 0;
while (isspace(s[i]))
   i = i + 1;
if (s[i] == '-')
   sign = -1;
else
   sign = 1;
if (s[i] == '+' || s[i] == '-')
   i = i + 1;
n = 0;
while (isdigit(s[i])) {
   n = 10 * n + (s[i] - '0');
   i = i + 1;
}
return sign * n;
```

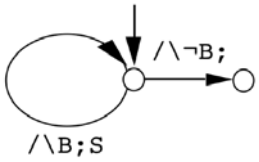
5.2 Draw the program graph for each of the following programming constructs:
(a) **if** B **then** S_1 **else** S_2;
(b) **if** B_1 **then** S_1 **else if** B_2 **then** S_2 **else** S_3;
(b) **while** B **do** S;
(c) **do** S **until** ¬B;



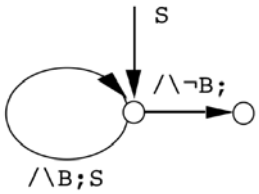
if B **then** S_1 **else** S_2



if B_1 **then** S_1 **else if** B_2 **then** S_2 **else** S_3



while B **do** S



do S **until** ¬B

5.3 How does a program graph differ from a traditional flowchart? What are the advantages and disadvantages of representing a program with a program graph?

The most fundamental difference is that in a program graph a program component is associated with an edge while in a traditional flowchart a program component is associated with a node (of a directed graph).

Association of the textual elements with the edges, instead of nodes, makes it possible to describe a program in terms of regular expressions. This makes it easy to represent, analyze, and manipulate the syntactic structure of a program.

5.4 Program 1.1 is logically equivalent to Program 5.16. What additional theorems are needed to show that rigorously?

Need theorems that allow a program to be represented as a set of symbolic traces. We can show the equivalence of Program 1.1 and 5.16 by showing that they can be decomposed into equivalent sets of symbolic traces that perform the same computation as explained in [HUAN08].

6.1 Find two examples (other than those given in the text) of programming constructs in C++ that are syntactically correct but semantically faulty.

First, if a function does not take any arguments, the parentheses should not be omitted. For example, one can assign a random number to a variable, say, x, by using the statement

```
x = ran();
```

If the parentheses are omitted, the expression

```
x = ran;
```

is still a syntactically correct statement but x will be assigned the starting address of function ran() instead of a random number.

Second, the order in which the components of an expression are evaluated may be compiler dependent. Thus, for example, the statement

```
x = pow( a[ n ], n++ );
```

may set x to $(a[n+1])^n$ instead of $(a[n])^n$ as intended because many compilers push function arguments onto the stack from right to left.

6.2 Consider the C++ program listed below:

```
1   #define YES 1
2   #define NO 0
3   main()
4   {
5   int c, nl, nw, nc, inword;

6   inword = NO;
7   nl = 0;
8   nw = 0;
9   nc = 0;
10  cin >> c;
11  while ( c != EOF ) {
12  nc = nc + 1;
13  if ( c == '\n')
14  nl = nl + 1;
15  if ( c == ' ' || c == '\n' || c == '\t')
16  inword = NO;
17  else if ( inword == NO ) {
18  inword = YES;
19  nw = nw + 1;
20  }
21  cin >> c;
22  }
23  cout << nl << endl;
24  cout << nw << endl;
25  cout << nc << endl;
26  }
```

Construct the minimum slice with the slicing criterion C = (26, c).

List below is the minimal slice.

```
1     #define YES 1
2     #define NO 0
3     main()
4     {
5     int c, nl, nw, nc, inword;

6     inword = NO;
7     nl = 0;
8     nw = 0;
9     nc = 0;
10    cin >> c;
11    while ( c != EOF ) {
12    nc = nc + 1;
13    if ( c == '\n')
14    nl = nl + 1;
15    if ( c == ' ' || c == '\n' || c == '\t')
16    inword = NO;
17    else if ( inword == NO ) {
18    inword = YES;
19    nw = nw + 1;
20    }
21    cin >> c;
22    }
23    cout << nl << endl;
24    cout << nw << endl;
25    cout << nc << endl;
26    }
```

6.3 In addition to technical competency, what are the personal traits that will make a software engineer (or a programmer) a more effective moderator in software inspection?

Being a good team player, not only proud of his or her own good work but also the quality of work done by the team as a whole.

Being a strong player, not only able to do his or her own work well but also have the additional capacity to do work that is often beyond the call of duty.

Being a confident and well-informed professional who knows the team members, including himself or herself, well.

6.4 Are there any additional materials that may be distributed to the participants of code inspection to make them more effective?

The participants often try to mimic the working of a computer to see how the computation is performed along an execution path. Automatically generated symbolic traces along major paths in the program should be of help to the inspection participants.

6.5 What are the main technical bottlenecks that prevent the methods of proving program correctness from becoming practical?

Difficulty in describing the intended function of a program in terms of pre- and post-condition.
Difficulty in guessing the right inductive assertions needed to construct a proof.
Difficulty in computing the weakest precondition of a loop construct.
Difficulty in finding the loop invariant of a loop construct.
Difficulty in automating the process of theorem proving.

6.6 If you are assigned to meet a sales representative who claims that his or her company has succeeded in developing a software tool that makes it practical to prove program correctness, what questions would you ask in order to ascertain that the claim is valid? The claim is most likely to remain invalid for some time to come, and you do not want to waste a lot of time with this sales representative. On the other hand, you may want to keep your door open for an unlikely event that there is a major technical break-through.

How do you automate the process of finding the weakest precondition of a loop construct? Or, how do you automate the process of finding the invariant of a loop?

6.7 Which concepts and techniques developed in proving program correctness are also used in program testing?

Precondition and postcondition
Weakest precondition
Backward substitution
Inductive assertion

7.1 Design and implement a software tool that will automatically instrument a program with software counters in such a way that a non-zero count of every counter implies traversal of all branches during the test execution.

See Sec. 7.1.

7.2 Discuss the mechanisms that may be used to detect the global violation of an assertion.

Local violation of an assertion can be detected by inserting an appropriate instrument at the specified location. Global violation of assertion can be detect in a similar manner, except that now an appropriate instrument has to be inserted at the entry point of the scope of that violation and wherever within the scope the values of the variables involved could be altered.

Because each of Problems 7.3 and 7.4 may take one person-semester or more to complete, assign them as software development projects instead of class-room exercises.

7.3 Design and implement a software tool that will automatically instrument a program for test-case effectiveness measurement.

7.4 Design and implement a software tool that will automatically instrument a program for data-flow anomaly detection.

7.5 Certain modern compilers are capable of detecting some types of data-flow anomaly. Investigate the capabilities of the compiler you use most often in this respect, and test it to see if it is capable of detecting all types of data-flow anomaly all the time.

7.6 The static method for data-flow anomaly detection may produce false alarms, i.e., may indicate that there is a data-flow anomaly while in fact there is none. Find an example significantly different from the ones given in the text that will trigger a false alarm if the static method is applied, but will not if the dynamic (instrumentation) method is applied.

7.7 Define TRACE( ) and INST( ) functions in Sec. 7.5 for a programming language of your choice.

7.8 Enumerate the situations under which the use of instrumentation method may become technically undesirable or unacceptable.