Testing Software

Diane Horton
Department of Computer Science
University of Toronto

July 1999

1 Introduction

Suppose I have written a method called `maximum()` that reads an unknown number of integers and prints the largest one. Let's say I've shown you many test runs (summarized in the table below) and the method works correctly on all of them. Would you then believe that the method works?

Input	Output	Correct?
3 16 4 32 9	32	yes
9 32 4 16 3	32	yes
22 32 59 17 88 1	88	yes
1 88 17 59 32 22	88	yes
1 3 5 7 9 1 3 5 7	9	yes
7 5 3 1 9 7 5 3 1	9	yes
9 6 7 11 5	11	yes
5 11 7 6 9	11	yes
561 13 1024 79 86 222 97	1024	yes
97 222 86 79 1024 13 561	1024	yes

One would think that ten test cases is plenty for such a trivial method. The problem is, they are not well chosen cases. It is easy to write a method that handles all ten, but fails in situations such as these:

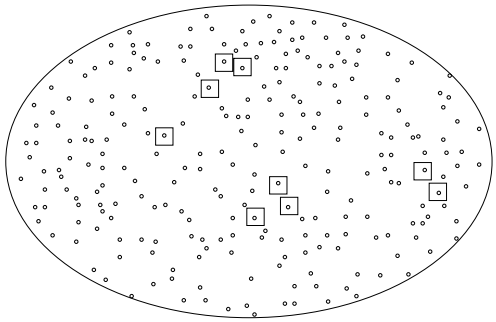
- A very short list (*i.e.,* of length 1, 2 or 3).
- An empty list (*i.e.,* of length 0).
  In fact, it's also easy to forget, as we did, to specify the method's behaviour in this sort of "boundary" case.
- A list where the maximum element is the first or last element.
- A list where the maximum element is negative.

All ten tests cover essentially the same situation: a list of moderate length, containing all positive integers, where the maximum is somewhere in the middle.

2 Choosing Test Cases

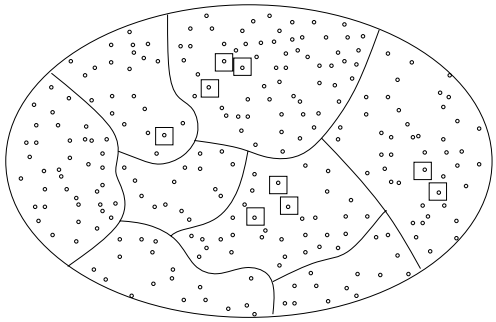
When you test a method, your goal is to confirm that it works not only on the particular cases that you tested, but on *all* possible cases. Even for our trivial little maximum-finder, the number of possible inputs is far too great for you to test each one. (Think about it!)

Consider a Venn diagram, where each element of the set is a possible input sequence to the method. Suppose we are testing a method to which there are millions of possible input sequences. We'll only draw a few of them here. Ones on which the method has been tested are boxed:

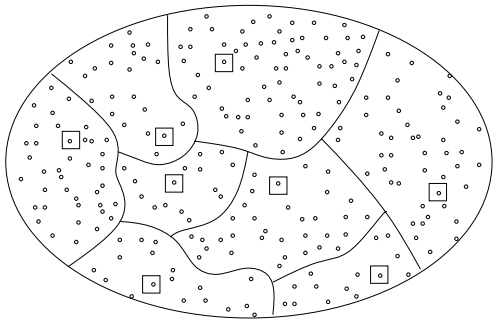


If the boxed test cases are chosen seemingly at random, there is absolutely no reason to believe that the method would work correctly on the millions of cases that were *not* tested.

If there are too many cases to test individually, and testing a random subset of them is unconvincing, what can we do? The solution is to be systematic: to (1) carve up all the possible inputs into appropriate categories:



and then (2) pick a representative test case from each one.



If a category is well chosen, it will be reasonable to conclude that because the method works correctly on one representative test case from that category, it works on all cases within that category. For

example, if the `maximum()` method handles any list of length 1, it is quite likely that it handles all lists of length 1. And if the categories completely cover the set of possible inputs, then you have constructed a convincing argument that the code works on all possible inputs. (Still, it is only an argument, and can only increase our confidence in the code. Unless we actually test every possible input, which is impractical for non-trivial code, testing cannot prove that there are no errors.)

This leaves an important question: How does one choose "appropriate" categories? Here are some principles to guide you:

Identify relevant properties of the input and systematically vary them.

In the case of the maximum-finder, relevant properties of the input include the size of the list, the position of the maximum, the existence of negative and positive values, the existence of duplicates, and the ordering of values within the list. We can vary these as follows:

- the size of the list: 0, 1, 2, 3, larger.
- the position of the maximum: at the very beginning, at the very end, and somewhere in the middle.
- existence of negative and positive values: lists with all positive values, all negative values (and hence a maximum that is negative), and mixed values.
- existence of duplicates: lists with no duplicates, lists with some which are the maximum, lists with some that are not, and lists that contain just one value repeated.
- ordering within the list: lists in ascending order, lists in descending order, and lists that are in no particular order.

Include boundary values for each feature.

These are values that are valid, but are at the boundary of what is valid. It is very common for code to work on "typical" cases, but to fail on some boundary cases, so pay special attention to them. For example, when varying the size of the list, it is important to include the smallest possible list (size 0) and some that are near that size; when varying the location of the maximum, it is important to consider cases where the maximum is at either extreme end.

Make sure every line of code is executed.

Over all the test cases, every line of code should be executed at least once. For a very large program, it is difficult to verify that you have achieved 100% "coverage", so there are now tools that will automatically check coverage. If you do not have access to such a tool, you should still think about coverage: After designing your test cases, inspect the code, looking for "branches" that the test cases don't cover. (Every `if`-statement defines branches which may or may not be taken when the code is executed.) For each such branch, devise a test case that covers it and add that case to your suite of tests. If it is not possible to make the program take that branch, then you have identified a piece of code that needs to be removed.

Play "Devil's Advocate".

You probably feel proud of your code and want it to work. But when testing, you must be the enemy of your code, so to speak. When you design test cases, **your goal should be to make the program fail**, not to show that it works. Try to think of the most perverse cases — the ones your program is most likely to mess up. If it passes this kind of tough scrutiny then you can feel more confident that your program works.

It is difficult to be this tough on your own code. Even more importantly, it is hard to step away from the very mistaken assumptions that caused you to write buggy code in the first place. For example, if you didn't consider that all your input might be negative when writing a `maximum()` method, you aren't likely to think of that situation later when testing it, and so you may miss a potential bug. Because of these problems, in industrial projects the author of a program is usually not the person who tests it.

Design Your Testing Early

If you design your testing early, perhaps even before you write the code, the exercise of thinking through cases systematically will make you aware of situations that you may not otherwise think of, such as all negative inputs to a `maximum()` method. This can help you avoid bugs before they happen.

3 Breaking Down the Problem

The example we've used so far is rather simple. A maximum-finder method is very small, and requires only one method, perhaps with a few small helper methods. Testing is more complicated when the program is of a reasonable size.

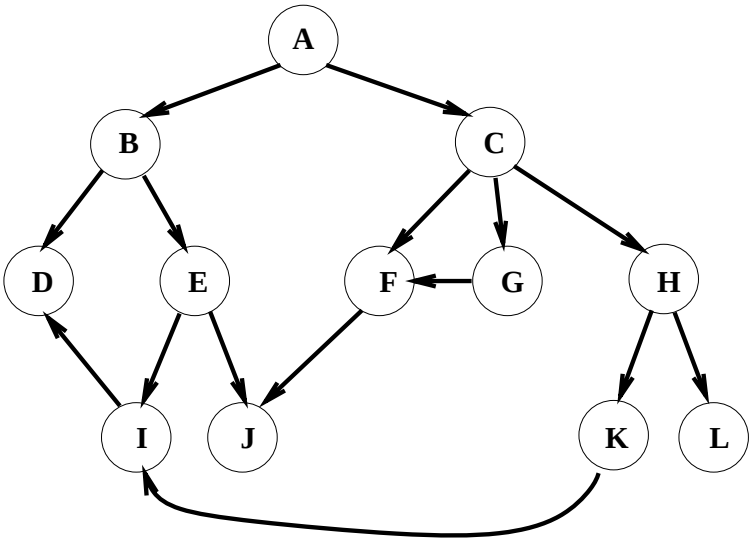
You have learned, when programming, to break down a large problem into smaller pieces (classes and methods, for example). This turns one mammoth problem into many small and manageable problems that can be tackled independently. The same lesson can be applied to testing.

Rather than test the entire program at once, test its components first. This is called **unit testing**. Once the units have been tested, they can be combined and tested in larger and larger groups, building up to testing the entire program. This is called **integration testing**. In industry, a unit might be the amount of code that a person or a team can write in a certain amount of time. With course assignments, you are working on a much smaller scale. You might consider a unit to be as small as an individual method.

I have argued that you should both write and test your code incrementally. In fact, you should interleave these tasks: Write a small unit of code, then immediately test it. For example, if you've just written a method to read in some data and build a structure to hold it, it is a very good idea to stop and test the method before writing code that performs some operation on the data structure. (Otherwise, you might find what you think is a bug in code for the operation, when in fact, the bug was in the input method.) Once you have written and tested a unit, move on to other units and to integration of the units, *testing as you go.*

You may feel reluctant to interrupt your programming in order to do testing; it can feel more productive to complete the whole program first. But this is an illusion. By working incrementally, you are likely to get to a complete, working program faster.

If we are going to test incrementally, we have to decide on the order in which to tackle it. Two general strategies are "bottom-up" and "top-down" testing.



Testing order for bottom-up testing:

- methods `D, J,` and `L`;
- methods `I`, and `F`;
- methods `E, G,` and `K`;
- methods `B`, and `H`;
- method `C`; and lastly
- method `A`.

Figure 1: A set of methods to be tested. An arrow from method `A` to method `B` indicates that `A` calls `B`.

Bottom-up Testing with Drivers

With **bottom-up** testing, we begin by testing methods that don't call any other of our own methods (although they may call Java API methods). Once they have been thoroughly tested, we can move up to testing methods that call them, working incrementally up the call tree. For example, suppose we were testing a program that contained the methods shown in figure 1. The only methods that do not call any other method are `D, J,` and `L`. These can be tested first, and in any order. Next, we can test methods that call these tested methods, but nothing else: methods `I`, and `F`. Then, we can test methods that call only `D, J, L, I,` and `F`: methods `E, G,` and `K`. Continuing in this manner, we work our way up until we finally can test method `A`. The complete test order is shown in figure 1.

In order to test an individual unit during bottom-up testing, we need a program that will call the unit and show the results. This is called a **driver program**. A driver program doesn't have to be fancy or general-purpose; it just needs to permit us to do the testing we want to do. There are several strategies you can take when writing a driver:

- Go through a fixed series of tests. Such a driver is "hard coded" to do exactly the same thing every time you run it. Figure 2 shows such a driver for a method called `quickSort()` that sorts an array of integers.

- Prompt the user (the tester) for inputs to test. Such a driver is more general. It can be a little more trouble to write than a hard-coded driver, but may be worth the trouble if you expect to vary the testing frequently. See figure 3 for an example.

- Randomly generate tests. See figure 4 for an example.

- Systematically generate tests by looping through a sequence of possibilities.
  See figure 5 for an example. To generate combinations of possibilities, we can use nested loops. For example, suppose we have a method that does something to an array `A`. Suppose the method takes one integer parameter that must be between `-A.length` and `+A.length)` and another integer parameter that must be between 0 and `A.length`-1. We can easily use a nested loop to generate all possible combinations of values for the parameters.

Top-down Testing with Stubs

Another strategy is to work from the top down during testing. But how can we test a method before having tested all the methods upon which it depends? This can be accomplished by using "stubs." A **stub** simulates the behaviour of an untested (or even unwritten) method. There are several ways to implement a stub. The simplest possible stub just prints a message saying that it was called and showing the values of the parameters. Sometimes a stub has to do a little more, for example, if the method must return a value. It may be sufficient for such a stub always return the same, fixed value. If the stub really does need to behave differently in different situations, we can make it do so by asking the user (in this case, the tester), to *tell it* what answer to return.

Testing need not proceed strictly top-down or strictly bottom-up. A mixture of the two is also possible.

4 Two Perspectives on Testing: Black Box and White Box Testing

When designing test cases, there are two points of view you can take. You can act as if all that you know about the code you are testing is its interface — you know nothing about how it is implemented. This is called **black box testing**, because you treat the code you're testing as a black box, *i.e.,* something inside of which you cannot see. Note that you can take this stance even if you do actually know about implementation details, but you must act as if you don't. With black box testing, all of your test cases are derived by systematic variation in the interface to the code.

The other point of view is that of an "insider" who knows the details of the code, and therefore knows of possibly weak points to push on. This is called **white box testing** because it is the opposite of black box testing. A much better name would be "clear box testing", since we are taking about the opposite in terms of opacity (*i.e.,* the ability to see through something).

So far, we have used both types of testing without distinguishing them. For example, the test cases shown on page 3 for the maximum-finder program are black-box cases, created without knowing anything about the implementation, but attempting complete coverage is a white-box technique. We will now work through another example in detail, devising first black-box test cases and then white-box test cases for a method.

5 Example: A Set Class

Consider a class that implements a mathematical set. Figure 6 gives the interface for this class, including specifications for each method. We'll design unit testing for the `insert()` method.

```
public static void main (String[] args) {

    int[] A1 = {87, 98, 69, 54, 65, 76, 87, 89};
    quickSort(A1);
    for (int i=0; i<A1.length; i++)
         System.out.println (A1[i]);

    int[] A2 = {1, 2, 3, 4, 5, 6, 7};
    quickSort(A2);
    for (int i=0; i<A2.length; i++)
         System.out.println (A2[i]);
}
```

Figure 2: A hard-coded driver. Only two test cases are shown; usually such a driver would contain a long sequence of tests.

```
public static void main (String[] args) throws IOException {

    BufferedReader in = new BufferedReader(
           new InputStreamReader( System.in ), 1 );
    System.out.print("How big an array do you want? ");
    int size = Integer.parseInt( in.readLine() );
    int[] A = new int[size];
    System.out.println("Please enter the array contents, one per line:");
    for (int i=0; i<size; i++)
          A[i] = Integer.parseInt( in.readLine() );
    quickSort(A);
    for (int i=0; i<size; i++)
         System.out.println (A[i]);
}
```

Figure 3: A driver that takes direction from the tester. A fancier one would put the whole thing in a loop and stop only when the user chooses to.

```
public static void main (String[] args) {

    java.util.Random generator = new java.util.Random();
    // Mod by 1000 to make sure the array size isn't too huge.
    int size = Math.abs( generator.nextInt() ) % 1000;
    int[] A = new int[size];
    for (int i=0; i<size; i++)
          A[i] = generator.nextInt();
    quickSort(A);
    for (int i=0; i<size; i++)
         System.out.println (A[i]);
}
```

Figure 4: A driver that generates random test cases. Again, a fancier one would put the whole thing in a loop and stop only when the user chooses to.

```
public static void main (String[] args) {

     java.util.Random generator = new java.util.Random();
     int[] A;
     for (int size=0; size < 1000; size++) {
          A = new int[size];
          for (int i=0; i<size; i++)
              A[i] = generator.nextInt();
          quickSort(A);
          for (int i=0; i<size; i++)
               System.out.println (A[i]);
          System.out.println("-------------");
     }
}
```

Figure 5: A driver that systematically generates test cases. This driver systematically tries different sizes of array, but uses random numbers for the array contents themselves.

```
// Implements a mathematical set of Objects.
// Two Objects are considered equal only if they are the same Object.
// (It is not enough that their equals() methods returns true.)

import java.util.Enumeration;

public interface SetInterface {

     // Inserts ‘o’ into the set.  Returns true if the insertion was successful
     // and false if it failed (because ‘o’ was already in the set).
     public boolean insert (Object o);

     // Removes ‘o’ from the set.  Returns true if the deletion was successful
     // and false if it failed (because ‘o’ was not in the set).
     public boolean remove (Object o);

     // Returns the number of elements currently in the set.
     public int size ();

     // Returns whether ‘o’ is an element of the set.
     public boolean isMember (Object o);

     // Returns an enumeration of the elements of the set.
     public Enumeration elements ();

     // Make the set empty.
     public void clear();
}
```

Figure 6: Interface for a set class.

5.1 Unit Testing of the Insert Method: Black Box Approach

Let's begin with black box testing. In order to emphasize that it is a black box, we won't reveal the implementation details for the class.

Inputs to the code

For the `maximum()` method, the only input is from the user. In general, there are at least three sources of information that can affect a method's behaviour: user input, parameter values, and the state of the object, presumably represented by instance variables. (Although an instance variable isn't really input to a method, I have called these sources of information "inputs" for lack of a better term.) In well designed code, these may be the only things that affect a method. But if other sources of information are relevant, such as a public instance variable in another class, they should be included in the testing.

The behaviour of our `insert()` method is affected by its parameter `item`, as well as by the state of the set.

Relevant properties of the inputs

The interface tells us that the item to be inserted could be any `Object`. Properties of an `Object` include its type (for example, it could be a `String`, a `Vector`, or a simple `Integer`) and its content. In this case, however, some properties of the input are `not` relevant: neither the type nor the content of the item to be added to the set makes any difference to the behaviour of the `insert()` method, except in that it may or may not be a duplicate of an item that is already in the set. So we will vary only duplicity.

Properties of the set include its size, and the type and content of its elements. Size is likely to be relevant to our testing, since boundary cases to do with size are a common source of bugs. But again, nothing about the type or the content of the set's elements makes any difference to the behaviour of `insert()`, except whether or not the new element is a duplicate.

So in summary, the relevant properties of the input are the size of the set, and whether or not the new item is a duplicate.

Systematically varying the properties

This is quite easy to do, now that we have identified the properties that need to be varied. We will be careful to include boundary values for each property.

- size of the set: empty; 1 element; 2 elements; 3 elements; more.
  If the interface acknowledged that the set could become "full" (*e.g.*, by providing an `isFull()` method, we should include a test case where we fill the set (by inserting until `isFull()` returns true) and try to insert one more item. However, the interface does not acknowledge such a possibility or let us test for it. Still, we should be skeptical and test the method's behaviour with a very large set.

- duplicity: an item identical to the new item exists in the set; the item itself exists in the set; or neither is true.
  Notice that we have been careful to test the two kinds of equality: the same object, and an object with the same content.

Case	Set content	New item	Relevant feature
1	empty set	5	inserting into empty set
2	5	6	set size 1; non-duplicate
3	5	5 (different object)	set size 1; duplicate value
4	5	5 (same object)	set size 1; duplicate object
5	5 10	6	set size 2; non-duplicate
6	5 10	5 (different object)	set size 2; duplicate value
7	5 10	5 (same object)	set size 2; duplicate object
8	5 10 15	6	set size 3; non-duplicate
9	5 10 15	5 (different object)	set size 3; duplicate value
10	5 10 15	5 (same object)	set size 3; duplicate object
11	2 3 5 8 10	6	larger set; non-duplicate
12	2 3 5 8 10	5 (different object)	larger set; duplicate value
13	2 3 5 8 10	5 (same object)	larger set; duplicate object
14	set with 1,000 randomly-generated elements	a non-duplicate element	adding to a very large set

Figure 7: Black box test cases for the `insert()` method.

Test cases

Now that we have identified appropriate categories, we simply have to select a test case that is representative of each one. In general, if there are n properties, each with on the order of k values to test, then there are k^n combinations. This can be far too many to test individually. If so, you should focus on combinations that involve border cases, while still thoroughly testing all the interesting values of each *individual* property.

In our case, we don't have to worry about the exponential number of combinations because there are only two properties to vary, with just five and three values to test respectively. In total, there are 15 combinations, so we can try them all.

One decision remaining is what type of `Object`s to put in the set and to use for the new `item`. Since we have argued that the type of each of these is irrelevant, we can choose to use any type of `Object`s in our test cases. To make things as simple as possible, we will use `Integer` for all `Object`s.

Figure 7 shows a set of test cases that meet our requirements. Note that set contents are always listed here in sorted order for the convenience of the reader; they may or may not be stored in that order by the `Set` class.

When running these test cases, we will have to confirm that the set has the correct contents after each call to `insert()`. This can be done by calling `elements()` to return the contents of the set, iterating through them, and printing each one.

At this point, we have 14 test cases. Why not 15? Two of the 14 combinations are not possible: you cannot attempt to insert a duplicate value or duplicate object into an empty set. So this leaves 13 cases. To this we have added one last cases involving a very large set.

Fourteen seems like plenty of test cases for one simple method. But we are not done!

5.2 Unit Testing the Insert Method: White Box Approach

In order to perform white box testing, we have to be able to "see inside the box". That is, we must know the implementation details of the code, in this case the `Set` class.

Assume that the set is stored in a sorted array so that searches, required for methods `isMember()`, `insert()`, and `remove()`, can be done quickly using binary search. The array begins at size 100. If it ever becomes full and another item needs to be inserted, a new array is allocated, double the current size, and all the elements are copied over to it starting from position zero. A good implementation would (1) allow the client code to choose the initial set capacity, and (2) shrink the array if it ever becomes too sparse, but our implementation doesn't have these features.

Now that we know these implementation details, we can do some testing that is more informed. If we knew even more, such as if we had access to the code, we might be able to come up with more new and relevant cases, but even with this limited information, a number of new cases will be needed.

New Relevant Categories

Because the data structure is a sorted array, we should consider where in the array the new `item` will go, or will be found if it already exists in the set. So we have a new property of the input to the code to vary: the insertion/match location. Relevant values are: at the front, the back, and somewhere in the middle.

Because we know that binary search will be used to identify the location in which to insert, we can think of particular situations that might cause problems for binary search either when the new item is a duplicate (and is therefore found by binary search) or when the item is not a duplicate and needs to be inserted:

- Odd and even sizes.
  These should be tried because it's possible to make a mistake around how the division in half is handled, especially when there is no exact middle element, as is the case in an even-sized list.

- Sets with exactly one or two items.
  There may be a bug in how the binary search gets initialized, causing it to fail on these very small lists. Note that these are not new cases — we already chose them as relevant when we designed the black box test cases. We have simply identified another reason why they are relevant.

- Search that stops at the exact middle spot, just to the left of it, and just to the right of it. (The search may stop because it finds that the new item already exists at this location, or because it finds that the new item needs to be inserted at this location.)
  Experience tells me that these cases can cause problems.

None of these cases involve a new property of the input; instead they involve new values of properties that we had already identified as relevant.

Lastly, because we know that the array size will be doubled if ever the array is going to overflow, we'll want to try some cases around this special situation. We should check insertions that take the set near the threshold, to the threshold, and just over it.

Case	Set content	New item	Relevant feature
15	2 4 6 8 10 12 14	1	odd size; inserting at front
16	2 4 6 8 10 12 14	15	odd size; inserting at back
17	2 4 6 8 10 12 14	7	odd size; inserting to left of middle
18	2 4 6 8 10 12 14	9	odd size; inserting to right of middle
19	2 4 6 8 10 12 14	2 (same object)	odd size; duplicate at front
20	2 4 6 8 10 12 14	10 (same object)	odd size; duplicate at back
21	2 4 6 8 10 12 14	8 (same object)	odd size; duplicate at middle
22	2 4 6 8 10 12 14	6 (same object)	odd size; duplicate to left of middle
23	2 4 6 8 10 12 14	10 (same object)	odd size; duplicate to right of middle
24	10 20 30 40 50 60	9	even size; inserting at front
25	10 20 30 40 50 60	61	even size; inserting at back
26	10 20 30 40 50 60	35	even size; inserting at middle
27	10 20 30 40 50 60	10 (same object)	even size; duplicate at front
28	10 20 30 40 50 60	60 (same object)	even size; duplicate at back
29	10 20 30 40 50 60	30 (same object)	even size; duplicate to left of middle
30	10 20 30 40 50 60	40 (same object)	even size; duplicate to right of middle
31	integers 1–98 inclusive	99	nearly full array
32	integers 1–99 inclusive	100	full array
33	integers 1–100 inclusive	101	array overflows and must double
34	integers 1-200 inclusive	201	array overflows and doubles again

Figure 8: White box test cases for the `insert()` method.

New Test Cases

Figure 8 shows additional test cases that cover the new categories that we have identified.

6 Presenting your Testing

You now know how to design a systematic and thorough set of test cases. This is often called a "test suite". In order to convince someone else that your program works, however, you will have to present your testing in a convincing fashion.

Annotated Test Output

The output from your test runs for the complete set of cases that you devised may be overwhelming in volume. In the pages and pages of output, there are key things to notice, and you will save the reader a lot of time by pointing them out. The simplest way to do this is to use a highlighter to highlight things on a printout and a pen to explain them. For example, on the output for test case 27, you might highlight the single 10 in the output and say "one 10; duplicate not inserted."

See figure 9 for an example of how hand-annotated test output might look. Notice that the set produced by one test case is usually not the set required for the next one. As a result, the driver program must keep creating new sets to test subsequent cases. This is inconvenient, and can be avoided to a degree by thinking about the driver when designing the test cases. For example, if

test case 5 involved inserting into a set containing the values 5 and 6 (which would still satisfy our requirement of inserting into a two-element set), then we could run this test immediately after test 2, when we have just created a set with these values in it.

A more sophisticated way to annotate test output is to give your driver the ability to recognize comments in the test input files, indicated by a special symbol such as **%**. These comments would be "echoed" in the output, thereby avoiding the need to handwrite them on the printout.

Testing Strategy

Even with 34 test cases for a simple thing like set insertion, the reader may still not believe that because your program worked on all of your test cases, it works on any *possible* example you could give it. The way to be convincing, of course, is to present the strategy behind your testing. It is, after all, what convinces you yourself. The simplest way to to explain your strategy is with tables like those above, plus a few extra remarks.

You should also cross-reference your output to the cases listed in your tables. If your driver is hard-coded, you can have it print out the case numbers as we did in figure 9. If not, you can hand write them on your printout.

7 More About Testing

This document has presented some of the most basic ideas behind good testing. There is, of course, more to learn. For example, after a piece of software has been completed and is in use, changes are often required to fix bugs, add new features, and so on. How does one test the changed software? What strategy can be used to avoid re-doing every test case? Another issue is performance testing, which is done to assess how much of some resource (such as time, or memory space) a program consumes. Here again, one must choose appropriate test cases, but now the cases concern resource use rather than correctness.

The software engineering textbooks listed below address some of these more advanced testing issues.

References

Jalote, Pankaj, *An Integrated Approach to Software Engineering*, Springer-Verlag, 1997. See chapter 9.

Pfleeger, Shari Lawrence, *Software Engineering: Theory and Practice*, Prentice Hall, 1998. See chapters 7 and 8.

Pressman, Roger S. , *Software Engineering: A Practioner's Approach*, third edition, McGraw-Hill, 1992. See chapters 18 and 19.

```
Case 1
------
Set contents before insert: empty        I will add highlighting and hand-written
Inserting:                  5            annotations on the final version.
Set contents after insert:  5
Case 2
------
Set contents before insert: 5
Inserting:                  6
Set contents after insert:  5 6
Case 3
------
Set contents before insert: 5
Inserting:                  5 (different object)
Set contents after insert:  5 5
Case 4
------
Set contents before insert: 5
Inserting:                  5 (same object)
Set contents after insert:  5
Case 5
------
Set contents before insert: 5 10
Inserting:                  6
Set contents after insert:  5 6 10
Case 6
------
Set contents before insert: 5 10
Inserting:                  5 (different object)
Set contents after insert:  5 5 10
Case 7
------
Set contents before insert: 5 10
Inserting:                  5 (same object)
Set contents after insert:  5 10
```

Figure 9: Annotated output from the first set of test cases for our `insert()` method. This output is from a hard-coded test driver, hence there is no user input. Notice that extra blanks were printed to align the integer values into a column for ease of reading. Small things like this make a big difference to the reader.