Algorithms and Data Structures
Week 2: Correctness and performance of programs

Abstract

Summary: This week we'll be concentrating on correctness of programs; we'll look at using pre- and post-conditions and loop invariants for capturing key correctness criteria. We'll reflect on how these are related to the JUnit tests that you write.

We'll also have a look at performance of programs, covering some of the important theoretical concepts in the study of complexity analysis. We'll reflect on how they relate to the experimental performance of the algorithms you write.

· **Book reference**: Drozdek chapter 2, section 2.1– 2.2;

· **Resources**: Week 2 archived eclipse program bundle.

· **Workshop:** In the workshop this week please look at exercises 1, 2, 3 and 4 at the end of these notes.

  There is a submission box open in iLearn for these solutions. **Submission is entirely optional but strongly recommended.** Although there are no marks for submission, recall that the Week 6 in class test will be formed from a selection of the exercises you have been asked to study in these workshops.

· **Extension questions:** Exercise 10 at the end of these notes. (This is extension only — it will not be part of the test.)

Correctness of programs

A program *invariant* is a relationship between the program variables *which does not change*. An invariant is used to establish the correctness of loops. A program *variant* is a function of the program variables which does change. It is used to establish that your loop terminates. Loops are said to be partially correct if the invariant establishes that provided they do terminate then the correct values are computed; loops are said to be totally correct of they are partially correct and they terminate.

The dishwasher problem: invariants in everyday life

This section poses a small everyday problem illustrating how to use invariants in logical thinking; the latter is crucial for good algorithm design!

> In a busy household dishes are periodically added to the dishwasher until it fills up, at which point it can be switched on

to clean the dishes inside. After cleaning the dishes should be removed ready for a new load.

The social dynamics of the household environment mean that the dishwasher is rarely completely emptied after it has been through its washing cycle, so that sometimes dirty dishes are added when some clean dishes are still inside. This can happen frequently if the person emptying the dishwasher is interrupted in their task. Separating the dirty dishes from the clean ones is not an easy task, and if this happens the whole load has to be rewashed.

Is there a foolproof protocol which ensures that dishes never become mixed up? You are not allowed to inspect all the dishes (see below what you are allowed to do). How would you prove that your protocol is correct?

This is a classic example of a "distributed system" where the overall tasks are made up of "atomic" subtasks, carried out independently by a number of agents. Other examples of distributed systems include e.g. internet protocols, and hybrid systems (cars, trains etc.). Distributed systems are notoriously difficult to get right because the independent agent behaviour can interfere with each other leading to annoying results (as in the dishwasher example).

The dishwasher atomic actions are the following:

- Add a dirty dish (if there is room and the dishwasher is off);
- Remove any dish (if there is one inside and the dishwasher is off);
- Switch the dishwasher on;
- Add dishwasher detergent (if the dispenser is empty and the dishwasher is off);
- Check to see whether the dishwasher is on;
- Check to see whether the dispenser is empty;
- Check to see whether the dishwasher is full or empty.

Problem: Design a small algorithm which consists of the above actions so that the clean and dirty dishes never get mixed up.

[Hint: Make that an invariant property and decide which action can break the invariant and what special conditions to impose to ensure this cannot happen.]

Simple invariants for loops

In your programming the same ideas apply.

```
int i=0;  int sum=0;
while ( i < N ) {
   i= i+1;
  sum= sum + i;
}
```

Trace through the algorithm and tabulate i, `sum` and $\sum_{0 \leq j \leq i} j$ to show that `sum` $==\sum_{0 \leq j \leq i} j$ is indeed an invariant.

Using your invariants

Once you've found a program invariant, to use it to check that your program is correct, you need to do three things.

1. Check that the invariant is true for the initialisation of the variables (just before the first iteration);
2. Check that if the invariant holds, and the loop guard holds (ie another iteration is possible), then the invariant is still true after the iteration;
3. Check that if the guard is false then the invariant implies (eg is the same as) the post condition.

Computing x^N

We'll look at two implementations of raising to the power problem. The first is easy to understand but slow, the second is harder to understand but fast. We'll apply some invariant reasoning to ensure we get it right.

Given x and N, compute x raised to the power of N.

- if $x = 2$ and $N = 3$, output 8
- if $x = 10$ and $N = 5$, output 100000

```
   int power (int x, int N) {
    // POST: output x^N
   int answer= 1;
   for (int i= 0; i < N; i++)
     answer= answer*x
  return answer;
}
```

Complexity $O(N)$.

There's a much faster way to do this, but care must be taken to get it right. Consider the following rules.

Rule 1: If N is even then $x^N = x^{N/2} \times x^{N/2}$.

Rule 2: If N is odd then $x^N = x^{N-1} \times x$.

To use this let a and b be variables which record whether we are using Rule 1 (on a) or Rule 2 (on b). In general we need to make sure that a, b satisfy the following equation $a^n \times b = x^N$, and we'll write a program to make sure that this becomes a loop invariant!

```
int fastPower (int x, int N) {
// PRE: x > 0; N >= 0
// POST: returns x^N

     int a= x; int b= 1; int n= N;
     while (n > 0) {
       if (n%2 == 0) {a= a*a; n= n/2;}
       else  {b= a*b; n= n-1;}
     }
     return b;
}
```

Complexity: $O(logN)$.

Correctness of `fastPower`

How do we know this is right? Assuming that $a^n \times b = x^N$ is a program invariant, what we want to check is that if it is true before we enter the loop for the first time, then we just need to check that it implies the post condition on termination.

It is true initially, because we let $a = x$, $b = 1$ and $n = N$, and check that $x^N = a^n \times b = x^N \times 1$ is true (which is is since $x^N \times 1 = x^N$). On termination we observe that $n = 0$, and the invariant in this case reduces to $x^N = a^0 \times b = 1 \times b = b$. Thus on termination we know that $b = x^N$ which is the correct value to return to satisfy the postcondition.

Using variants

A variant is an integer valued function of the program variables, that should either always increase, or always decrease after each complete iteration. If it is a decreasing variant, it should never decrease below zero; if it is an increasing variant then there should be a point beyond which it cannot increase. If exactly one of these properties hold of your variants then your program is guaranteed to terminate. If you find that your chosen variant does not always decrease then your program might go into an infinite loop.

Case study: VLSI chips design

As we have seen, invariants are particularly good for intricate algorithms which have been introduced to improve performance. Consider the following problem.

> A VLSI chip manufacturer has n supposedly identical VLSI chips that are potentially capable of testing each other. His test jig accommodates two chips at a time, and the result is either that they are the same, or that they are different.
>
> We have been hired to design an algorithm to distinguish **good** chips from **bad** ones, using as few comparisons as possible — this is because the test jig is slow, and the manufacturer wishes to speed up overall development. The trouble with **bad** ones is that they might test the same OR different. However all **good** do test the same (as each other), and all**bad** test different from all **good** . Because of the manufacturing process we know that more than half of the manufactured chips fall into the "good" category.

A "brute force" solution to this problem takes each pair of chips and tests them against all the others, selecting those which test "same" for more than half. This involves $O(n^2)$ comparisons.

We can do much better – in fact solving the problem in $O(n)$ comparisons– by thinking harder.

Let S be the complete set of chips to test; Let C be a subset of S. We want to ensure that all items in C are the same as each other (i.e. are either all good OR all bad).

Observe that if $C = S$ then because we know that S has more than half of the "good" transistors, and that C contains transistors that are the same as each other that in this case S can only contain good chips.

Here are the steps in the pseudocode; we use the invariant to check that it works.

1. If C is empty, then add any element of S to it — we know that any chip is the same as itself.
2. If C is not empty, then select some element c from C and some element s from $S \backslash C$, and test them in the test jig.
3. If $s = c$ then add s to C;
4. If $s != c$ then remove both s (from S) and c (from C).
5. Repeat from step 2.

The *invariant* for this program is that there are always at least $|S|/2 + 1$ good chips in S and, all chips in C are the same as eachother (i.e. are either all good or all bad). (The first condition implies that S is nonempty: why?)

The *variant* is the size of $S\backslash C$ (i.e. $|S\backslash C|$).

We will discuss these ideas with respect to other examples in lectures, including those found in the sample programs for week 2.

Correctness and JUnit testing

The problem of testing is that even if your programs pass all the tests you write, it doesn't mean that they work in all cases! The pre- and postconditions for your methods should try to cover all cases and actually should be considered as the specification of what the implementation is trying to live up to. Where such specifications are available you can improve your tests by trying to make them match up with representative cases described by the specification.

For example in `fastPower` the pre-condition `x >= 0; N >= 0` has infinitely possible representations and we can't test them all. But only really four of them are representative: the various combinations of `x == 0`, `x > 0` and `N == 0`, `N > 0`; in particular the cases where `x` or `N` are somewhat special and these are the places where we frequently find errors in programs.

Does the given implementation of `fastPower` do the right thing in all cases?

Theory of time complexity

How fast a particular algorithm runs on a computer depends on things like the clock speed of the computer, how big the input is, and the design of the algorithm.

Time complexity is the measurement of efficiency of a particular implementation of a problem, independent of the clock speed and input. (As you have seen from your own experimental performance testing, it is not easy to get an accurate measurement.)

We use Big Oh notation to give a quantitative indication of the efficiency of a program relative to the size of the input. We use worst-case analysis as a way to compare different implementations of the same problem: thus we only compare them for their respective worst (ie slowest) inputs, and only for very large values.

Big Oh notation is used to estimate the gross features of the growth of time complexity relative to the input size. The most common patterns are:

- *linear time* – $O(n)$
  Examples: Simple linear searches e.g. in arrays or lists; comparison of list-like structures.
- *quadratic time* – $O(n^2)$
  Examples: Simple sorting e.g. Bubble sort, selection sort, but also quick sort.
- *logarithmic time* – $O(\log n)$
  Examples include searching on sorted structures, e.g. Binary search on a sorted array.
- *exponential time* – $O(2^n)$
  Many of the most difficult problems are exponential which means that for large input values the computing time becomes infeasible. Digital security relies on the existence of such problems.
- *n-log n time* – $O(n \log n)$
  Examples include some of the faster sorts, such as merge sort (but not quick sort!).

Big "Oh" arithmetic

Let $f(n), g(n)$ be integer-valued functions.

Define $O(f(n)) \ll O(g(n))$ if there are constants $k \geq 0$ and $N \geq 0$ such that for all $n \geq N$, we have $k * g(n) \geq f(n)$.

In this definition we don't care about the actual value of the constants. That is because for very large N the value of $f(N)$ in comparison with $g(n)$ will be determined by their dominating terms. So:

$$O(\log n) \ll O(n) \ll O(n \log n) \ll O(n^2) \ll O(2^n)$$

The following will help you figure out $O(f(n))$ for a given function $f(n)$.

- Constants are all equivalent to $O(1)$
- If $O(f(n)) \ll O(g(n))$ then $O(f(n) + g(n)) = O(g(n)$
- Multiplication is easy: $O(f(n) \times g(n)) = O(f(n)) \times O(g(n))$
- Constants dont matter: $O(k \times f(n) + c) = O(f(n))$
- The largest growing term is the only one that matters. So for example, $O(n \times n + log(n)) = O(n \times n)$

Computing big Oh

Decide which are the most expensive operations. Compute approximately how many times they are used in your implementation.

The following patterns will help you. In these examples we assume that the array lookups are the most expensive.

Linear time

Notice that the distance to finishing is reduced by a constant amount, with possibly only a constant number of inner iterations, at each outer iteration.

```
· for(int i=0; i < n; i++)  x= A[i];
· for(int i=n; i > 0; i--)  x= A[i];
· for (int i=0; i < n ; i= i+2)  x= A[i];
·  for (int i= 0; i < n ; i++)
     for (int j= 0; j < 2  ; j++) {x=  A[i][j];}

·  for (int i=1; i < n ; i=2*i)
     for (int j=i; j < 2*i; j++) {x=  A[i][j];}
```

Log time

Notice that the distance to finishing is reduced by approximately halved each time; in the case of a nested loop there are only a constant number of inner iterations at each outer iteration.

```
· for(int i=1; i < n; i=2*i)  x= A[i];
· for(int i=n; i > 0; i=i/2)  x= A[i];
· for (int i=1; i < n ; i= 3*i)  x= A[i];
·  for (int i= 1; i < n ; i=i*2)
     for (int j= 0; j < 2  ; j++) {x=  A[i][j];}
```

Quadratic time

There are normally nested iterations, with the number of inner iterations determined proportional to the number of outer iterations that have occurred.

```
· for(int i=1; i < n*n; i++)  x= A[i];
·  for (int i= 1; i < n ; i++)
     for (int j= 0; j < n  ; j++) { x=  A[i]; }

· for (int i= 0; i < n ; i++)
     for (int j= 0; j < i  ; j++) { x=  A[i]; }

·  for (int i=n; i > 0 ; i--)
       for (int j= i; j > 0  ; j--) { x=  A[i]; }
```

N log N time

A combination of halving the distance to finishing and doing n inner iterations each outer iteration. (Or the other way about!)

```
· for (int i= 0; i < n ; i++)
     for (int j= 1; j < n  ; j=2*j) { x=  A[i][j]; }

· for (int i= 1; i < n ; i=2*i)
     for (int j= 0; j < n  ; j++) { x=  A[i][j]; }

· for (int i=1; i < n ; i=2*i)
       for (int j= i; j < n  ; j++) { x=  A[i][j]; }
```

Case study: Breaking RSA cryptography

In the news recently some research analysing some of the practical problems involved in generating keys for RSA public key encryption. We will look at how the researchers managed to break thousands of private keys using some of the simple techniques of the sort that you have seen before.

The idea is to use the greatest common divisor, which has faster complexity than exponential, because of Euclid's algorithm.

```
int gcdIterative (int M, int N) {
// POST: Returns the greatest common divisor of M and N
int n= N; int m= M;
while (m!=n) {
 if (m > n) m= m-n;
 else n= n-m;
}
return n;
}
```

This is correct because of the invariant: $gcd(m, n) = gcd(M, N)$. This is $O(N + M)$.

Practice questions

1. In this activity you must work collaboratively with a friend to develop correct code for the programming tasks set out below. They are related to the linked list class discussed in lectures.

   One person must take on the role of programmer, and the other the role of tester. Swap roles so that everyone gains the experience both as a tester and a programmer.

In the role of programmer you must look at the specification and write a method in the linked list class that implements it. In the role of tester, without writing the code (!) you must write a set of JUnit tests that a correctly-written program would pass.

Make sure that the testers and programmers do not collaborate whilst writing the code and tests, but when both teams are done try out the tests on the code. Look at eachother's work and comment, wrt the following questions: Did the code pass the tester's tests? Did the tests cover all the main cases?

- Write a program that takes two linked lists (l1 and l2) and interleaves them so that the returned list is the first node from l1, followed by the first node from l2, then the second node from l1 followed by the second node from l2 etc. If any nodes are left after the interleaving then they should be appended.

  `public SLList interleave (SLList l1, SLList l2);`
- Write a program that takes a linked list l1 of integers, and an integer x and removes all the instances of x from the list, leaving the remaining nodes connected in the same relative order as they were before.

  `public SLList removeALL (SLList l1, int x);`

What to submit:

Submit the code and tests written in the group exercise along with short answers to the above questions. Please include comments describing your design and the and tests.

You may submit either the tests or the code depending on which role you took.

If you cannot make it to class and therefore are working on your own, please submit solutions for one of the programs and write both the code and tests for your choice.

2. How much does the ice in a hockey rink weigh?

   For this question, getting exactly the right answer is not the point (although it's good when it's in the "ball park"). Questions like these help you to practise thinking about a problem and to define the assumptions that need to be made to give a reasonable answer. In your group, discuss some assumptions you can make to enable you to do a "back of the envelope" calculation. In your submission list the assumptions that you made in order to come up with your answer.

   Note also that in "real life" most problems are ill defined (such as this one) and to be a good problem solver and critical thinker, you need to be able to formulate the problem in a way that gives something of a meaningful (if not always useful!) answer.

3. Here is an implementation of Binary Search that unfortunately Annabelle did in a hurry and so didn't have time to test it. It does assume that the input array A is sorted and that the key is an item which might (or might not) be an item occurring in it.

   (a) In the case that key does appear in A, the integer returned should be the smallest index such that A[index] == key.

   (b) In the case that key does not appear in A then the smallest index should be returned such that everything to the left of index is strictly less than key, and everything to the right is at least key. (Note this is a neat way of saying the same thing as 1 in the case that key is in A ....)

   For this exercise you are asked to perform a code review to help Annabelle improve her design. Concentrate in particular on the coding style, use of variables, comments (or lack thereof), correctness and performance. If you think there might be circumstances under which the program does not compute the right answer then provide an example and try to point to the place in the code that causes the problem. (Hint: consider the cases below.) Write a few tests to highlight problems you find.

   In your answer point out the problems and places for improvement. (Be polite but firm!) Show me the tests that you used to find the problems, and make sure that in the corrected code they now work.

```
static int binarySearch(char A[], char key) {
int f= 0;
int l= A.length;

while(f < l) {
  int mid= (l-f)/2;
  int x= A[mid];
  if (x == key) return mid;
  if (x < key) f= mid;
  else l= mid;
}

return f;
}
```

   - when A is empty
   - when it's non-empty, and everywhere less than key
   - non-empty and everywhere more than key
   - non-empty and key is not there but would fit somewhere inside
   - key is there exactly once

- key is there more than once

For an extra challenge, once you've cleaned up Annabelle's code, can you find an invariant for the loop which would help convince her that you've cleaned it up correctly?

4. You have been asked to implement the software for a large company's employee database. The database should store the employees' names, employee id's, their roles, salaries, holiday allowances etc. The company has a huge number of employees but a low turnover (employees rarely join or leave). However on a day-to-day basis a large number of employee records need to be looked up, checked and revised.

   Discuss the sequential data structure that you would choose to ensure that the average processing time over a month is as low as possible.

   When you have decided how the data structure should be arranged, think about the resources you need for adding, deleting and looking up an item. State which operations (out of adding/deleting/looking up) take the most or least time, suggesting a Big Oh estimate for each relative to the arrangement of data you have selected.

5. Work out the big Oh equivalent of the following functions.

   (a) $x^2 + 100x$

   (b) $x \times (4x + 20)$

   (c) 400

   (d) $x^2 + \log x + 11$

6. Consider the following small program:

```
int SumBetween (int X, int Y)
//  pre  :  X <= Y
//  post : returns the sum X + (X+1) + ... + Y

{
   int sum = 0;
   for (int i=X; i!=Y+1; i++)
      sum += i;
}
```

   (a) State the loop invariant.

   (b) Show that the loop invariant is established by the initialization.

   (c) Verify that the loop invariant is preserved by each iteration of the loop.

(d) Show that the loop invariant and the termination condition imply the post-condition.

(e) Assuming that addition is the most expensive operation, what is the complexity in big oh notation of this program?

7. Consider the code for binary search given in this week's activities. Study the specification for this method so that you understand exactly what it is supposed to achieve.

   Write some JUnit tests so that if the binary search were correct, the code would pass your tests. In fact the code given contains (at least) three errors (that I know about). Try to make your tests uncover the problems.

   Hard: Write an invariant for the recursive method which if satisfied would prove that the program is correct. Try to make it help you detect how to correct the program.

   [Note: This question is about writing tests to match the specification; when you report that you have found an error, you must state how your JUnit test detected the error.]

8. Consider the implementation below for Insertion Sort, which sorts an array into descending order.

   (a) Trace the algorithm for the array `A = {2,1,5,0,6}`, and `n=5`; make sure you keep track of the array after each inner and outer iteration. In your trace, draw a rectangle around the array after each complete iteration of the OUTER loop (see (++) in the code).

   (b) Verify that the array portion `A[0..j-1]` is sorted after each complete iteration of the outer loop, i.e. underline that portion of the array and check that it is sorted. (Note that `A[0..j-1]` means "the portion of the array between, and including, indices `0` and `j-1`.)

   (c) Now check that the invariant for the outer loop is established (made to be true) by the initialisation of the variable j; i.e. substitute the initial values of the relevant variables into the invariant statement and observe that it is true.

   (d) Finally deduce that the postcondition is satisfied after the outer loop has terminated.

   [Hint: what is the value of `j` on termination? Substitute into the invariant.]

```
void insertion_sort(int array[], int n)
 // PRE: n is the length of array
 // POST: array is sorted
{
     int i, j, key;
       for(j = 1; j < n; j++)    //Notice starting with 1 (not 0)
```

```
      // Invariant for the outer loop: array[0..j-1] is sorted, descending order
        {
            key = array[j];
            for(i = j - 1; (i >= 0) && (array[i] < key); i--)
            {     //Move smaller values up one position
               array[i+1] = array[i];
             }
            array[i+1] = key;    //Insert key into proper position (++)
        }
       return;
}
```

9. One of the tricks to designing algorithms is to reuse strategies from similar problems. The following question concerns an application of binary search to a different problem. The reason we can use binary search is because the problem can be thought of as searching an ordered array of integers, even though we don't actually implement the array.

   The integer square root of an integer n is defined to be the integer m such that $m^2 \leq n < (m+1)^2$.

   Use a binary search strategy to write a function to find the integer square root.

   [Hint: Consider the sorted array `[0, 1, 2, ...., n]`, and look for the element that matches the rule for being integer square root.]

10. Recall the pseudocode from lectures for identifying a good VLSI chip. The idea can be reused to finding the majority item in a list, where the majority is defined by the element in a list of size n which occurs at least $n/2+1$ times. Write a linear-time program to find the majority element in an array. What is the invariant of your loop? What is the variant of your loop? Explain why your design has linear time performance.

    Write JUnit tests to check your program.

11. A celebrity is someone that everyone knows, but who knows no one. Find an efficient algorithm for determining whether or not a group of n people contains a celebrity, based on asking questions of the form "Excuse me x, do you know y?" What is the time complexity of your solution?

    [Hint: what can you can conclude about the celebrity-ness of x or y if x answers "yes"; what can you conclude if he says "no".]