Reading Material
Lab Exercise 1: Algorithm Design Workout

Before setting out the task in detail, we briefly introduce some of the basic concepts you will need. As part of this, an example of a problem and a single model solution to it is given. This should give you all you need for completing the lab.

1 Algorithms

This lab is designed to make you think about algorithms. So what exactly is an algorithm?

An algorithm is a definite procedure for achieving a specific goal. The goal usually specifies two things: the **input** to the algorithm, and the desired **output**. An algorithm that reliably takes any allowed input and spits out the required output is a *correct* algorithm. The details of how it does it are unimportant to its correctness, as long as it does it reliably.

The key thing to remember is that an algorithm is not the same as a piece of (machine) code, since an algorithm is more abstract than a program; it is a higher-level description.

2 An example problem

For example, we might require an algorithm to find the largest number in a list of integers. The input here is any non-empty (finite) list of integers. The output is the largest number. If there is more than one largest number, this does not matter, we will still return it just once.

An algorithm for doing this is as follows:

```
1 find_largest_in_list (list)
2 {
3   largest ← first element of list
4   for each i in list
5     if (i > largest)
6       largest ← i
7   end for
8   output: largest
9 }
```

Figure 1: An example algorithm for finding the largest element in a given list

3 Pseudocode

The algorithm above is described in pseudocode. Basically, the aim of pseudocode is clarity and precision in defining an algorithm. Pseudocode does not need to be machine-readable (it is not in a specific language), so some leeway in syntax is allowed, i.e., you are allowed to use short bits of english as long as they are unambiguous, given the context.

To understand more about pseudocode, consult Goodrich and Roberto, pages 7–8. Alternatively, look at examples of pseudocode here.

4 How do we think up algorithms?

For most problems it is easy to think of one way to do it. You just need to think logically about what needs to get done, and think of a logical way of organizing it. But to design an efficient algorithm is often more demanding. You might need to make use of an algorithmic trick, such as divide-and-conquer, dynamic programming, or greedy search (all things you will learn more about during the course). Or you might need to think more laterally.

5 Correctness Arguments (Informal)

Q. Is the above algorithm for finding the largest integer *correct*?

We could argue that it is, as follows.

To find the largest number in a (finite) list, it is necessary and sufficient to check every element once. (This seems self-evident). As we check each element, we just need to see if it exceeds the largest number encountered so far (line 5), and if it does we update the largest number so far (line 6). To get this all started, the largest so far is initially set to be the first element (line 3). The largest number is output in line 8.

In COMP11212 you met one method for *proving* correctness (Hoare Logic) and this (and other methods) will be mentioned again later in the course. But today's task just asks for an informal argument similar to the one given above (though they may need to be a bit more complicated if the algorithm is longer).

6 Complexity of Algorithms

To understand something about how long an algorithm will take to run, we often analyse the number of basic operations it will perform. We may count different kinds of operations depending on what the problem is (for example, comparisons, memory accesses, additions, or multiplications), or any combination of these.

Q. How many basic operations does the algorithm for finding the largest integer use?

A. The dominant (or most frequent) basic operation it uses is comparison, so I will count these. It compares every element in the list against the variable, "largest". This is n comparisons for a list of length n.

Note that the answer explains what operation is being counted and why. It would (obviously) be wrong to count multiplications here as the algorithm doesn't use them. We must count the thing it does most (the dominant operation(s)), as that gives the best idea of how long the algorithm will take to run.

Also note that the answer is given in terms of n, the size of the input given to the algorithm. We will usually want to express the complexity of an algorithm in this way, in terms of the input size (or in terms of some number given in the input). This is because the complexity (number of basic operations used) is a function of n. **What we want is a worst case analysis.**

For the above problem, the algorithm always uses exactly n comparisons for an input of size n. But for many problems the state of the input affects the number of operations needed to calculate the output. For example, in sorting, many sorting algorithms are affected by whether the input is already sorted or nearly sorted. Some algorithms are very fast if the input is already sorted, some are very slow. What we usually want to know is how does the algorithm perform (how many operations it uses) in the worst case.

Sometimes it is hard to think about the worst case, but mostly it is easy. For the list of problems given in this lab, it is intended to be easy to identify worst cases.