COMP26120 Lab Exercise 1
Algorithm Design Workout

Duration: 1 lab session

For this assignment, you do not need to write any Java or C code (though you may do so, in order to check your ideas). Instead, you are asked to design some algorithms and describe them in pseudocode. You are also asked to reason about their running time and correctness. *You will hand in your work as a text file*.

Note on deadlines: To let you make best use of the help available in the labs, the deadline **for those attending the lab** will be automatically extended to the start of your next COMP26120 lab session. If you do not attend the lab, the deadline is the end of your current lab session.

Aims

To think about algorithms, not their implementation in a specific language.

Learning Outcomes

On successful completion of this exercise, a student will:

1. Have seen that the specification for an algorithm describes the allowed input(s) and the required output(s).
2. Have understood how algorithms may be described in a language-independent way using pseudocode.
3. Have designed their own algorithms for some problems.
4. Have met one or more generic techniques for algorithmic problem-solving, such as divide-and-conquer.
5. Have thought about how to show or argue that an algorithm is correct.
6. Have thought about the time complexity of an algorithm - the number of basic operations that it will use.

Summary

The task is to choose any **one** of the problems described below. You must then think of **two** different algorithms for the problem, and describe them in pseudocode. The first algorithm can be the first thing you think of that works. The second algorithm should be an improvement (if possible) in time complexity over the first one (i.e. it should use fewer operations). You should argue, convincingly, why each algorithm is correct, and finally write about their complexity (how many basic operations are used by the algorithms).

NB: overall you will have designed and described **two** algorithms.

You cannot get an extension for this lab (except for e.g. medical problems)!

To let you make best use of the help available in the labs, the deadline **for those attending the lab** will be automatically extended to the start of your next COMP26120 lab session. If you do not attend the lab, the deadline is the end of your current lab session.

Description

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 remainder of the lab.

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

An *algorithm* is a definite procedure for achieving a specific goal. When we design an algorithm, we must first understand the goal - what it is the algorithm is supposed to do. 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*.

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:

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

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 the course textbook, pages 7--8. Alternatively, look at examples of pseudocode here.

How do we think up algorithms?

You probably could have written the above algorithm...I hope so ! Hopefully, 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. This is why we are asking you to try and think of **two** algorithms for your selected problem. The first one is the bog-standard, if you like. The second one should improve it, if possible. **Please try to think these up for yourself before scouring the web.** If you do find the solution online or in a book, please acknowledge this (you will not lose marks). Try and learn from the way the solution was constructed.

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 3), and if it does we update the largest number so far (line 4). To get this all started, the largest so far is initially set to be the first element (line 1). The largest number is output in line 6.

Later in the course, we will learn about more formal methods of *proving* correctness. 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).

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 below, it is intended to be easy to identify worst cases.

The Task

Select any **one** of the following problems. For your selected problem, give

(a) **Two** algorithms for solving the problem, in pseudocode; the second one should be substantially different to the first and should improve upon the first one (i.e. reduce the complexity).
(b) A description why **each** algorithm is correct.
(c) A description of the number of operations **each** algorithm uses, explaining why you think this is the worst case. Your answer should be in terms of n unless stated otherwise in the problem.

Clearly state the letter of the problem you have chosen to tackle.

The Problems:

A. Find the "fixed point" of an array

Input: an array A of *distinct* integers in ascending order. (Remember that integers can be negative!) The number of integers in A is n.
Output: one position i in the list, such that A[i]=i, if any exists. Otherwise: "No".

Hint: for your second algorithm, you may like to read up on "binary search".

B. Max and Min of a List

Input: An array of integers A, of length *n*.
Output: The maximum and the minimum elements of the list.

For your complexity analysis, count the number of comparison operations that are used.

Hint: For the second, more efficient algorithm it is possible to use a "divide-and-conquer" algorithm. (Look this up in your course text book or elsewhere). For a list of length one, the maximum and minimum is just the one element (no comparison is needed). For a list of length two, one element is the maximum and the other the minimum (even if they are both the same), so this can be established with one comparison only. How could you join together two lists whose minimum and maximum you already know? Try to use these ideas to devise an efficient algorithm.

C. Minimum Number of Coins

Input: A list of coin values, which are 1,2,5,10,20,50,100,200. An integer *T*.
Output:The minimum number of coins needed to make the number *T* from the coins.

It is assumed that there is no limit to the number of coins available, i.e. every coin has an infinite supply. For example, for *T*=20,001. The answer would be 101. (100x the 200-value coin and 1x the 1-value coin).

Hint: One algorithm to solve this problem efficiently is to use an algorithmic technique called *dynamic programming* (this has nothing to do with computer programming, it a mathematical method). Later in the course you will do this for a related problem, but it is too difficult (and not needed) for this problem.

Instead, consider using enumeration: trying out all possible coin combinations of 1 coin, 2 coins, etc., until a combination that works is found. And for the second algorithm, consider using a "greedy" method. (Look this up in your course text-book, p259-262).

Note: The complexity of your algorithms should be expressed in terms of *T* for this problem.

D. Greatest common divisor

Input: Two positive integers *u* and *v*, where $u > v$
Output: The greatest number that divides both *u* and *v*

Hint: if you get stuck for a second algorithm, look up Euclid's algorithm. (But this does not need to be one of your methods).

Note: The complexity of your algorithms should be expressed in terms of *u* for this problem.

E. Word Cloud Problem

You may have seen word clouds in the media. Some examples are here. They visually represent the important words in a speech or written article. The words that get used most frequently are printed in a larger font, whilst words of diminishing frequency get smaller fonts. Usually, common words like "the", and "and" are excluded. However, we consider the problem of generating a word cloud for all the words. A key operation to generate a word cloud is:

Input: A list W of words, having length *n* (i.e. n words)
Output: A list of the frequencies of all the words in W, written out in any order, e.g. the=21, potato=1, toy=3, story=3, head=1

Hint: For the second algorithm you could sort the words first. You may assume that this can be done efficiently. For your efficiency calculation, just count the basic operations used after the sorting.

Marking Process

You should write a flat text file called Algorithms.txt and put it in your COMP26120/ex1 directory. Then use "submit".

Your text file should identify the problem you selected. Remember: It should give two solutions, i.e. two pieces of pseudocode. For each one, you should attempt to state why the code is correct, and also attempt to calculate the worst-case number of operations used.

The marks are awarded as follows:

```
Algorithm 1 pseudocode is reasonable attempt = 2 marks
Algorithm 1 is correct and argument is good = 1 mark
Algorithm 1 efficiency is correctly calculated = 1 mark
Algorithm 2 pseudocode is properly distinct from algorithm 1 = 2 marks
Algorithm 2 is correct and argument is good = 1 mark
Algorithm 2 efficiency is correctly calculated = 1 mark
Algorithm 2 is quite efficient, or improves Algorithm 1 = 2 marks
```