COMP26120
Academic Session: 2022-23

Worksheet 1: Algorithm Design Workout

This exercise is not assessed.

Learning Objectives

At the end of this worksheet you should be able to:

1. Explain why we use pseudocode as a language-independent method for describing algorithms.
2. Use pseudocode to represent algorithms and explain what given pseudocode means in English.
3. Design algorithms for some simple problems.
4. Describe one or more generic techniques for algorithmic problem-solving, such as divide-and-conquer.
5. Reason (informally) about the correctness and time complexity of algorithms e.g. to present reasoned arguments for each based on a description of the algorithms execution

Introduction

This worksheet is about examining problems, devising solutions to those problems, and then analysing those solutions. You are encouraged to be *creative* as well as thinking *analytically* about the problem and your solutions. There is never only one way to solve a problem, although there may be an optimal solution (however, it is not necessary to find this solution).

It is important to remember that you should be providing your answers in pseudocode and not any real programming language.

Before we describe the exercises, we will quickly review the key ideas you will need to use. These have already been introduced in the lectures and directed reading so you may be able to just quickly skim through them.

Algorithms

This worksheet 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 e.g. solving a **computational problem**. This goal/problem usually specifies two things: the expected **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.

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)
{
  largest ← first element of list
  for each i in list
    if (i > largest)
       largest ← i
  end for
  output: largest
}
```

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

Pseudocode

The algorithm above is described in pseudocode. 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.

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. Another common approach is to relate your problem to a known problem with a known solution, identify the differences, and adapt the known solution. This course aims to teach you common algorithmic tricks and common problems/solutions.

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[1] you met one method for *proving* correctness (Hoare Logic) but we don't take such a formal approach to correctness in this course. We will focus on well-structured, rigorous arguments with varying levels of formality and detail e.g. we will often leave gaps where something is 'obviously true'. Today's task just asks for an informal argument similar to the one given above (although 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 in this worksheet, it is intended to be easy to identify worst cases.

[1]For those who took it, for those who did not don't worry - although the notes are online.

Exercises

Select **two** problems, one from Problem Set 1 and one from Problem Set 2, and for each of your selected problems, give

1. **Two** algorithms for solving the problem, in **pseudocode**. The first algorithm can be the first thing you think of that works. The second one should be substantially different to the first and should improve upon the first one (i.e. it should use fewer operations). You may think of the best solution first, it is then your job to find a worse solution to compare it to!
2. A description why **each** algorithm is correct.
3. 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.

Please try to think these up for yourself before looking on the web. If you do need to look something up, try and learn from the way the solution was constructed.

Clearly some of these problems are more difficult than others. There are no extra points for difficultly, although some of the more challenging problems may have more obviously different solutions. Once you have devised solutions for one problem in a set you may want to have a go at another. We believe most students should be able to tackle one problem from each set in the time allocated to this worksheet. This obviously doesn't prevent you tackling more of them – or indeed saving some to look at when you are doing revision. On the semester 2 exam there will be a question that asks you to design an algorithm and justify its correctness and complexity - while this exam question will expect you to draw on techniques taught later in this course, the questions on this worksheet will give you practice thinking about algorithm design.

Problem Set 1

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. Majority Element

Input: An array of integers A, of length n.
Output: An integer k such that k appears in more than half of the positions of A if such a k exists. Otherwise “No”.

Hint: For the second algorithm you could consider trading space for time.

C. 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.

D. Computing Statistics

Input: An array of integers A, of length n
Output: The *mean*, *variance*, and *standard deviation* of the values in A

Hint: It is possible to do this in a single pass using a *recurrence relation*

E. Choose Without Replacement

Input: An array of integers A, of length n and a value k such that $k \neq n$.
Output: k unique items from A chosen randomly *(without bias)* i.e. select k things from n without replacement.

Hint: For the first algorithm you may want to just keep track of what you have chosen. For the second algorithm you will need to be cleverer than this. In the worst case analysis consider the possible relationship between n and k.

Problem set 2

F. 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 and just count the basic operations used after the sorting.

G. 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 Goodrich and Roberto, p259-262).

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

H. Balancing the See Saw

You are running a summer camp for children and are put in charge of the See Saw. This is most fun when the two sides of the See Saw are balanced but you have the issue that children are different weights. Your solution is to give each child a hat of varying weight to balance them out. To help in this task you need to devise an algorithm that takes the weights of two children and the weights of the available hats and works out which hats to give each child.

Input: A pair of numbers (*left*,*right*) and list of hat weights W of length n.
Output: A *different* pair of numbers (i, j) such that *left*+$W[i]$ =*right*+$W[j]$ or "not possible" if there are no such i and j.