COMP26120
Academic Session: 2022-23

Worksheet 1: Algorithm Design Workout
Model Solution

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:

```
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

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".

Solution:
A very simple algorithm is iteration. A slight modification of this adds a special early stopping case e.g.

```
for i from 0 to length(A)
    if A[i] = i then return true
    if A[i] > i then return false
return false
```

Correctness: If we return true then it is clear that this is the correct result. We also need to show that we do not miss any `true` results. The only time we exit the loop early is if $A[i] > i$. In this case, due to A being sorted, it is not possible for $A[j] = j$ some index $j > i$ as the elements of A are distinct e.g. it is necessarily the case that $A[i+1] > i+1$ and so on (this is an inductive argument).
Complexity: In the worst case we iterate through the whole of A hence $O(n)$.

Given that the list is already sorted, an obvious better algorithm is a divide-and-conquer binary search approach.

```
left = 0; right = length(A)-1
while left ≠ right
    middle = floor((left+right)/2)
    If A[middle] = middle then return true
    If A[middle] < middle then left = middle+1
    Else right = middle−1
Return false
```

Correctness: This argument is slightly more complicated than the standard binary search correctness argument as we are not looking for a specific element in the array. Again, if we return true then it is clear that this is the correct result. We also cannot miss any true results. We rely on the distinctness of the elements again and the same result as above e.g. if $A[i] > i$ then $A[i+1] > i+1$. This can also be applied in the symmetric $A[i] < i$ case to show that in both cases the solution cannot occur in the part of the array that is being ignored.

Complexity: In the worst case we have $\lfloor log_2 n \rfloor + 1$ operations, which is $O(log\ n)$. It is important to understand where the $log_2\ n$ comes from in these kinds of solutions. The Wikpedia page on Binary Search provides the standard explanation based on the height of the *search tree* explored by the algorithm.

B. Majority Element

Leetcode 169

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.

Solution:
There are a number of possible solutions here. The main ones seem to be:

- Naive - Multiple iterations counting numbers of occurrences
- Hash-counting
- Sort and then look in the middle
- Divide-and-conquer - find the majority in sub-arrays then combine
- The Boyer-Moore single pass algorithm e.g. https://en.wikipedia.org/wiki/Boyer%E2%80%93Moore_majority_vote_algorithm
- Probabilistic (choose elements at random)

The probabilistic approach isn't guaranteed to give the correct answer (but can be made to give the probability that the answer is correct) so let's ignore that one. Boyer-Moore is fun but I wouldn't expect you to come up with that by yourself (it's $O(n)$). You can see the sorting and divide-and-conquer solutions described here with code: https://leetcode.com/problems/majority-element/solution/. I'll describe the first two.

The naive solution is to to check if each element is the majority e.g.:

```
for i from 0 to length(A)
    count := 0
    for j from 0 to length(A)
        if A[j] = A[i] then count++
    if count>length(A)/2 then return A[i]
return "No"
```

Correctness: This is trivial. If we return k then we've checked that k appears in more than half the positions. If we return false then none of the elements it a majority element.
Complexity: This is $O(n^2)$ as we have a nested loop but in practice we could achieve better than that by giving up early on some iterations - if the remaining part of the array is less than `length(A)/2-count` then there's no way for that element to be the majority element.

For the hash counting algorithm we just need two passes. One where we use a hash table to count the number of occurences of each element and the second pass to find the one with the largest count.

```
map := empty map
```

```
majority_value := A[0]; majority_count := 0

for i from 0 to length(A)

    map[A[i]] := 1 + (0 if A[i] in the map or map(A[i]) otherwise)
    if map[A[i]] > majority_count then
        majority_count = map[A[i]]
        majority_value = A[i]
```

- `if majority_count > length(A)/2 then return majority_value`

```
return "No"
```

Correctness: Firstly, we need to convince ourselves that `majority_count` and `majority_value` do store the count/value of the element that appears most often in the array (it doesn't matter if this is not unique). We can see that we are correctly tracking the count of each element via the map and that we keep track of the largest count so far. Secondly, we then argue that if k is a majority element then it must also be the most frequent element so it is either `majority_value` or a majority element does not exist. Hence, we can just check whether `majority_value` occurs more than `length(A)/2` times.

Complexity: Iterating once with constant hash table operations - $O(n)$. The memory complexity is going to be $O(n)$ due to the way hash tables store things internally.

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.

Solution:
The simple solution is brute-force, trying every value.

```
for i from v down to 1

    if i divides u and i divides v then return i
```

Correctness: As we try every value (including 1) we're guaranteed to find a number that divides both u and v. Because we check from largest to smallest it will be the greatest such number.
Complexity: $O(v)$ as in the worst case we check every number from v to 1.

Euclid's algorithm is better. There are two variants. One where we subtract the smaller of the two numbers on each step and one where we take the remainder on each step. The second is more efficient:

```
while v ≠ 0
    t:=v
    v := u mod v
    u:=t
return u
```

Correctness: The correctness argument is based on the fact that the GCD of two numbers remains the same if the smallest one is subtracted from the largest one (that's what the subtraction version of the algorithm does). Taking the remainder is the same as subtracting as many times as we can.

Complexity: Computing this precisely is complicated. If you look it up you'll see that it's $5 \log_{10} u$. The argument is actually quite beautiful but I'm not going to repeat it here (see https://en.wikipedia.org/wiki/Euclidean_algorithm#Number_of_steps). What you should be able to see is that there's something "logarithmy" going on here due to the remainder step.

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*

Solution:
The simple solution is to pass through A twice - once for the sample mean and once for the standard deviation. I'm not going to write that out as it should be straightforward. The complexity is obviously $O(2n)$.

The better solution is to use a single pass with a recurrence relation. For example, Welford's algorithm (see http://en.wikipedia.org/wiki/Algorithms_for_calculating_variance) exploits a neat recurrence relation (that is a little tricky to come up with by yourself). The complexity then becomes $O(n)$.

Isn't $O(n) = O(2n)$? Yes, in theory, but in practice there's obviously a measurable difference!

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.

Solution:
The simple solution is to choose at random from [1,N], mark things as you remove them, and if you come across something already marked then choose again. Obviously correct. Complexity is more difficult - at round i the probability of having to choose again is $N - i/N$. When K is small this probability is small and time complexity is close to O(K). If K is almost N then time complexity approaches $O(N^2)$.

To avoid coming across something you've seen before we can modify the above so that whenever you select something you swap it with the last element in your range and then select from [1,N-1]. This is correct as at each step we know that the part of the array we are not selecting from only contains things already selected. We achieve $O(K)$ time with constant space so this is likely to be the winner.

Randomly shuffle the list and then select the first K items. To be correct we need this to be unbiased e.g. randomly generate a number per item in the list and sort based on this. Complexity is dominated by sorting.

A more complicated idea is to create a random binary string of length N with exactly K 1-bits. The 1-bits then represent the numbers to select. The problem then becomes doing this in an unbiased way e.g. flipping a coin N times isn't going to work as you're not guaranteed to get K heads. This is a potential solution that we would welcome your thoughts on!

Tudor Bujdei-Leonte who took this unit in 2021-2022 has suggested the following pseudo-code for this problem. It has similarities with a couple of the approaches above but isn't identical to any of them. It will have $O(N)$ time and $O(2N)$ (so $O(N)$) space.

```
current_n = n
current_k = k
results = []
for i = 1 to n
    if (random() < current_k/current_n): // random() uniformly chooses a number
                                         // between 0 and 1
        results.append(A[i])
        current_k = current_k - 1
    current_n = current_n - 1
return results
```

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.

Solution:
A naive solution to this problem is to maintain an array of the seen words and use this to index into an array of word counts. So for each word you first need to find the index into this seen array and then update the relevant count in the count array. For example:

```
seen_array = empty
nseen = 0
count_array = empty
for each word in input list of length n
  i = 0
  while (word != seen_array[i] and i less than nseen)
    i = i+1
  if(i less than nseen)
    count[i] = count[i]+1
  else
    seen_array[nseen] = word
    count[nseen] = count[nseen]+1
    nseen = nseen + 1
for (i = 0 to nseen-1)
    print (seen_array[i], count_array[i])
```

The correctness is straightforward. The complexity comes from searching for the word in the `seen_array`. In the worst case all words are new and we make i comparisons on each step, which ultimately gives us a $O(n^2)$ complexity - although it is somewhat smaller in practice.

Another approach is to using a hash table to record the `count_array`, removing the need for the `seen_array` and the need to search it. If we assume hash table operations are constant we get $O(n)$.

If we sort A first then we only need to scan the array once, counting the blocks of identical words. So if we don't count the cost of sorting we have $O(n)$. If we do then sorting will dominate.

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.

Solution:
The naive solution is a brute-force one e.g.

```
For i in 1 to T
    try every combination of i coins
       if a combination is found that has total = T, stop.
```

This is correct as we don't miss any combinations. We have $O(8^T)$ operations in the worst case - which is bad for large T.

However, we can take a *greedy* approach here. In fact, we'll see this example again when we explore greedy algorithms properly later in the course. The high-level pseudocode is:

```
Set V=T
   Until V=0, do
    Take the largest value coin and subtract its value from V
```

Correctness comes because for coins $1, 2, 5, 10, 20, 50, 100, 200$ each subsequent coin's value is at least double the previous one's. This property ensures that we cannot make a total in fewer coins by taking smaller valued ones in combination. This follows by thinking about "carrying" in arithmetic. e.g.

```
10s 5s 2s 1s
0   2  1  1 = 13 (4 coins)
but we could have carried the two 5s, which gives
1   0  1  1 = 13 (3 coins)
for fewer coins.
```

Making sure we have carried everything always results in fewer coins. For complexity, we need at most 8 coins (1 of each) to make any total less than 400. For larger numbers, we will take the 200 coin until we get below 400 and then start taking smaller coins. This will require T/200 + 8 coins in the worst case. So the algorithm is $O(T)$.

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.

Solution:
The straightforward solution here is to loop through list W twice checking each possible (i, j):

```
for i in 0 to n
  for j in 0 to n
    if i!=j and left + W[i] = right + W[j] then return (i,j)
return not possible
```

Correctness. This is straightforward as we check each possible solution.
Complexity. This is $O(n^2)$ as we both loops go up to n.

There's a possible solution that involves sorting but the linear solution involves remembering. The idea is to create a set of all possible resultant weights of the left child wearing each hat and then check each hat on the right child and lookup whether that weight had been seen on the left child. This depends on being able to efficiently look up the weights we saw with the left child when considering the right child. This can be given in pseudocode as

```
Let L be an empty set
Let M be an empty map
for i from 0 to n
   l = W[i] + left
   add l to L and set M[l] to i
for i from 0 to n
  r = W[i] + right
  if r in L then
    if M[r] != i
      return (M[r],i)
return not possible
```

Correctness. In the first iteration we precompute all possible weights resulting from the left child wearing a hat. On the second iteration we consider all hats that the right child could wear. If the resulting weight is the same as a previously seen weight then we have a solution.

Complexity. We iterate through n twice only performing constant operations (assuming set and map operations are constant e.g. we use a hashmap and an amortised argument). Therefore, the complexity is $O(2n) = O(n)$.