Week 3 Applied Sheet

Objectives: The applied sessions, in general, give practice in problem solving, in analysis of algorithms and data structures, and in mathematics and logic useful in the above.

Instructions to the class: You should actively participate in the class.

Instructions to Tutors: The purpose of the applied class is not to solve the practical exercises! The purpose is to check answers, and to discuss particular sticking points, not to simply make answers available.

Supplementary problems: The supplementary problems provide additional practice for you to complete after your applied class, or as pre-exam revision. Problems that are marked as **(Advanced)** difficulty are beyond the difficulty that you would be expected to complete in the exam, but are nonetheless useful practice problems as they will teach you skills and concepts that you can apply to other problems.

Problems

Problem 1. Describe a simple modification that can be made to any comparison-based sorting algorithm to make it stable. How much space and time overhead does this modification incur?

Problem 2. A subroutine used by Mergesort is the merge routine, which takes two sorted lists and produces from them a single sorted list consisting of the elements from both original lists. In this problem, we want to design and analyse some algorithms for merging many lists, specifically $k \geq 2$ lists.

(a) Design an algorithm for merging k sorted lists of total size n that runs in $O(nk)$ time

(b) Design a better algorithm for merging k sorted lists of total size n that runs in $O(n \log(k))$

(c) Is it possible to write a comparison-based algorithm that merges k sorted lists that is faster than $O(n \log(k))$?

Problem 3. Consider an application of radix sort to sorting a sequence of non-empty strings of lowercase letters a to z in alphabetical order (each character of the strings can be interpreted as being base-26 for running the internal counting sort rounds). Radix sort is traditionally applied to a sequence of equal length elements, but we can modify it to work on variable length strings by simply padding the shorter strings with empty characters at the end.

(a) What is the time complexity of this algorithm? In what situation is this algorithm very inefficient?

(b) Describe how the algorithm can be improved to overcome the problem mentioned in (a). The improved algorithm should have worst-case time complexity $O(n)$, where n is the sum of all of the string lengths, i.e. it should be optimal.

Problem 4. Write an in-place algorithm that takes a sequence of n integers and removes all duplicate elements from it. The relative order of the remaining elements is not important. Your algorithm should run in $O(n \log(n))$ time and use $O(1)$ auxiliary space (i.e. it must be in-place).

Problem 5. The engineers in your company believe they have created a phone that is ultra-resistant against falls (perhaps even falls from as high as 150m). To test that hypothesis, your company built two prototypes and asked you to perform the test to determine the maximum height in meters (without considering fractions) that the phone can be dropped from without breaking.

There is an unknown integer value $0 \leq x \leq 150$ such that if the prototype is dropped from heights up to x meters it will not break, while if it is dropped from heights of $x+1$ meters or higher it will break. Your job is to determine x. If one prototype is broken, it cannot be used in the tests anymore.

As a computer science student, you want to optimise your work. For doing so, you plan to develop an algorithm for determining the heights you should drop the prototype from at each iteration. No matter what is the value of

$0 \leq x \leq 150$, your algorithm should be correct and determine the value of x. For an algorithm A and an unknown integer $0 \leq x \leq 150$, let $\mathsf{Drop}(A, x)$ denote the number of times you need to drop a prototype if you use algorithm A and x is the threshold. Let $\mathsf{Drop}(A)$ denote the worst-case performance of A, which is given by the maximum of $\mathsf{Drop}(A, x)$ over all possible $0 \leq x \leq 150$.

For an optimal deterministic algorithm A that has $\mathsf{Drop}(A)$ as small as possible, what is the value of $\mathsf{Drop}(A)$?

[Hint: Since you have two prototypes, it is possible to do better than linear search. But as you only have two prototypes, you need to be very careful if only one prototype remains intact (and thus the search cannot be as efficient as binary search).]

Problem 6. Devise an efficient online algorithm[1] that finds the smallest k elements of a sequence of integers. Write pseudocode for your algorithm. [Hint: Use a data structure that you have learned about in a previous unit]

Supplementary Problems

Problem 7. Write pseudocode for insertion sort, except instead of sorting the elements into non-decreasing order, sort them into non-increasing order. Identify a useful invariant of this algorithm.

Problem 8. Consider the following algorithm that returns the minimum element of a given sequence A. Identify a useful invariant that is true at the beginning of each iteration of the **for** loop. Prove that it holds, and use it to show that the algorithm is correct.

```
1: function MINIMUM_ELEMENT(A[1..n])
2:     min = A[1]
3:     for i = 2 to n do
4:         if A[i] < min then
5:             min = A[i]
6:     return min
```

Problem 9. Consider the problem of finding a target value in a sequence (not necessarily sorted). Given below is pseudocode for a simple linear search that solves this problem. Identify a useful loop invariant of this algorithm and use it to prove that the algorithm is correct.

```
1: function LINEAR_SEARCH(A[1..n], target)
2:     Set index = null
3:     for i = 1 to n do
4:         if A[i] = target then
5:             index = i
6:     return index
```

Problem 10. Write an iterative Python function that implements binary search on a sorted, non-empty list, and returns the position of the key, or None if it does not exist.

(a) If there are multiple occurrences of the key, return the position of the **final** one. Identify a useful invariant of your program and explain why your algorithm is correct

(b) If there are multiple occurrences of the key, return the position of the **first** one. Identify a useful invariant of your program and explain why your algorithm is correct

Problem 11. Devise an algorithm that given a sorted sequence of distinct integers $a_1, a_2, ..., a_n$ determines whether there exists an element such that $a_i = i$. Your algorithm should run in $O(\log(n))$ time.

[1] In this case, online means that you are given the numbers one at a time, and at any point you need to know which are the smallest k.

Problem 12. Consider the following variation on the usual implementation of insertion sort.

```
1: function FAST_INSERTION_SORT(A[1..n])
2:     for i = 2 to n do
3:         Set key = A[i]
4:         Binary search to find max k < i such that A[k] ≤ key
5:         for j = i downto k + 1 do
6:             A[j] = A[j − 1]
7:         A[k] = key
```

(a) What is the number of comparisons performed by this implementation of insertion sort?

(b) What is the worst-case time complexity of this implementation of insertion sort?

(c) What do the above two facts imply about the use of the comparison model (analysing a sorting algorithm's complexity by the number of comparisons it does) for analysing time complexity?

Problem 13. (Advanced) Consider the problem of sorting one million 64-bit integers using radix sort.

(a) Write down a formula in terms of b for the number of operations performed by radix sort when sorting one million 64-bit integers in base b.

(b) Using your preferred program (for example, Wolfram Alpha), plot a graph of this formula against b and find the best value of b, the one that minimises the number of operations required. How many passes of radix sort will be performed for this value of b?

(c) Implement radix sort and use it to sort one million randomly generated 64-bit integers. Compare various choices for the base b and see whether or not the one that you found in Part (b) is in fact the best.

Problem 14. (Advanced) When we analyse the complexity of an algorithm, we always make assumptions about the kinds of operations we can perform, how long they will take, and how much space that we will use. We call this the *model of computation* under which we analyse the algorithm. The assumptions that we make can lead to wildly different conclusions in our analysis.

An early model of computation used by computed scientists was the *RAM*, the Random-Access Machine. In the RAM model, we assume that we have unlimited memory, consisting of *registers*. Each register has an *address* and some contents, which can be any integer. Additionally, integers are used as pointers to refer to memory addresses. A RAM is endowed with certain operations that it is allowed to perform in constant time. The total amount of space used by an algorithm is the total size of all of the contents of the registers used by it.

(a) State some unrealistic aspects of the RAM model of computation.

(b) Explain why the definition of an in-place algorithm being those which use $O(1)$ auxiliary space is near worthless in this description of the RAM model.

(c) In the Week 2 Applied, we discussed fast algorithms for computing Fibonacci numbers. In particular, we saw that $F(n)$ can be computed using matrix powers, which can be computed in just $O(\log(n))$ multiplications. If a RAM is endowed with all arithmetic operations, then we can compute $F(n)$ in $O(\log(n))$ time using this algorithm. If instead the RAM is only allowed to perform addition, subtraction, and bitwise operations in constant time, we can still simulate multiplication using any multiplication algorithm (for example, Karatsuba multiplication). Explain why in this model, it is impossible to compute $F(n)$ in $O(\log(n))$ time with this algorithm.

A model that is more commonly used in modern algorithm and data structure analysis is the *word RAM* (short for word Random-Access Machine). In the word RAM model, every word is a fixed-size w-bit integer, where w is a parameter of the model. We can perform all standard arithmetic and bitwise operations on w-bit integers in constant time. The total amount of space used by an algorithm is the number of words that it uses

(d) What is the maximum amount of memory that can be used by a w-bit word RAM?

(e) Suppose we wish to solve a problem whose input is a sequence of size n. What assumption must be made about the model for this to make sense?

(f) Discuss some aspects that the word RAM still fails to account for in realistic modern computers

(g) Does the word RAM model allow us to compute $F(n)$, the n^{th} Fibonacci number, in $O(\log(n))$ time?