Week 3 Tutorial Sheet

(To be completed during the Week 3 tutorial class)

Objectives: The tutorials, 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. The preparation problems are not assessed. We strongly recommend that everyone tries to solve the preparation problems before the tutorial as you will benefit the most from the tutorial if you come properly prepared. However, you can still attend the tutorial if you have not solved those problems beforehand.

Instructions to Tutors: The purpose of the tutorials 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 tutorial, 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.

Weekly Implementation checklist

It will be most beneficial for your learning if you have completed this checklist **before** the tutorial.

By the end of week 3, write Python code for:

1. **Counting sort**
2. **Radix sort**

Tutorial Problems

Problem 1. **(Preparation)** Write a Python function that implements counting sort. Test your sorting function for sequences with only small elements, then sequences with large elements and observe the performance difference.

Problem 2. **(Preparation)** Show the steps taken by radix sort when sorting the integers 4329, 5169, 4321, 3369, 2121, 2099.

Problem 3. Consider the following algorithm that returns the number of occurrences of *target* in the sequence A. Identify a useful invariant that is true at the beginning of each iteration of the **while** loop. Prove that it holds, and use it to prove that the algorithm is correct.

```
function COUNT(A[1..n], target)
    count = 0
    i = 1
    while i ≤ n do
        if A[i] = target then
            count = count + 1
        end if
        i = i + 1
    end while
    return count
```

```
end function
```

Problem 4. 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 5. 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 6. 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 7. Think about and discuss with those around you why auxiliary space complexity is a useful metric. Why is it often more informative than total space complexity? For the purpose of this problem, define auxiliary space complexity as the amount of space required by an algorithm, excluding the space taken by the input, and define total space complexity as the space taken by an algorithm, including the space taken by the input.

Problem 8. 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]

Problem 9. 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 10. Consider an application of radix sort to sorting a sequence of nonempty strings of lowercase letters a to z. Each character of the strings is interpreted as a digit, hence we can understand this as radix sort operating in base-26. 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.

Supplementary Problems

Problem 11. 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.

```
function MINIMUM_ELEMENT(A[1..n])
    min = A[1]
    for i = 2 to n do
        if A[i] < min then
            min = A[i]
        end if
```

[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.

```
    end for
    return min
end function
```

Problem 12. 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.

```
function LINEAR_SEARCH(A[1..n], target)
    Set index = null
    for i = 1 to n do
        if A[i] = target then
            index = i
        end if
    end for
    return index
end function
```

Problem 13. 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 14. 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.

Problem 15. Consider the following variation on the usual implementation of insertion sort.

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

(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 16. (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 17. (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 tutorial, 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?