Week 4 Tutorial Sheet

(To be completed during the Week 4 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: Aim to attempt these questions before the tutorial! It will probably not be possible to cover all questions unless the class has prepared them in advance. There are marks allocated towards active participation during the class. You **must** attempt the problems under **Assessed Preparation** section **before** your tutorial class and give your worked out solutions to your tutor at the start of the class – this is a hurdle and failing to attempt these problems before your tutorial will result in 0 mark for that class even if you actively participate in the class.

Instructions to Tutors:

1. The purpose of the tutorials is not to solve the practical exercises!
2. 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 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.

Assessed Preparation

Problem 1. 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. What are the worst-case time complexities of Quicksort assuming the following pivot choices:

(a) Select the first element of the sequence

(b) Select the minimum element of the sequence

(c) Select the median element of the sequence

(d) Select an element that is greater than exactly 10% of the others

Describe a family of inputs that cause Quicksort to exhibit its worst case behaviour for each of these pivot choices.

Tutorial Problems

Problem 3. Suppose that Bob implements Quicksort by selecting the average element of the sequence (or the closest element to it) as the pivot. Recall that the average is the sum of all of the elements divided by the number of elements. What is the worst-case time complexity of Bob's implementation? Describe a family of inputs that cause Bob's algorithm to exhibit its worst-case behaviour.

Problem 4. In the lectures, a probability argument was given to show that the average-case complexity of Quicksort is $O(n \log(n))$. Use a similar argument to show that the average-case complexity of Quickselect is $O(n)$.

Problem 5. Show the steps taken by radix sort when sorting the integers 4329, 5169, 4321, 3369, 2121, 2099.

Problem 6. Consider an application of radix sort to sorting a sequence of 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

Problem 7. Devise an algorithm that given a sequence of n unique integers and some integer $1 \leq k \leq n$, finds the k closest numbers to the median of the sequence. Your algorithm should run in $O(n)$ time. You may assume that Quickselect runs in $O(n)$ time.

Problem 8. One common method of speeding up sorting in practice is to sort using a fast sorting algorithm like Quicksort or Mergesort until the subproblems sizes are small and then to change to using insertion sort since insertion sort is fast for small, nearly sorted lists. Suppose we perform Mergesort until the subproblems have size k, at which point we finish with insertion sort. What is the worst-case running time of this algorithm?

Supplementary Problems

Problem 9. A subroutine used by Quicksort is the partitioning function which takes a list and rearranges the elements such that all elements $\leq p$ come before all elements $> p$ where p is the pivot element. Suppose I instead have $k \leq n$ pivot elements and wish to rearrange the list such that all elements $\leq p_1$ come before all elements that are $> p_1$ and $\leq p_2$ and so on..., where $p_1, p_2, ..., p_k$ denote the pivots in sorted order. The pivots are not necessarily given in sorted order in the input.

(a) Describe an algorithm for performing k-partitioning in $O(nk)$ time. Write psuedocode for your algorithm

(b) Describe a better algorithm for performing k-partitioning in $O(n\log(k))$ time. Write psuedocode for your algorithm

(c) Is it possible to write an algorithm for k-partitioning that runs faster than $O(n\log(k))$?

Problem 10. Write a Python function that implements the Randomised Quicksort algorithm (Quicksort with random pivot selection).

Problem 11. Take your Quicksort code from Problem 10 and modify it so that it implements the Quickselect algorithm where the pivot is chosen randomly.

Problem 12. Write psuedocode for a version of Quickselect that is iterative rather than recursive.

Problem 13. Modify your Quicksort code from Problem 10 so that it uses your Quickselect function from Problem 11 to select a pivot. Compare this against random pivot selection and see which one performs better for randomly generated lists.

Problem 14. (Advanced) Consider a generalisation of the median finding problem, the *weighted median*. Given n unique elements $a_1, a_2, ...a_n$ each with a positive weight $w_1, w_2, ..., w_n$ all summing up to 1, the weighted median is the element a_k such that

$$\sum_{a_i<a_k} w_i < \frac{1}{2} \qquad \text{and} \qquad \sum_{a_i>a_k} w_i \leq \frac{1}{2}$$

Intuitively, we are seeking the element whose cumulative weight is in the middle (around $\frac{1}{2}$). Explain how to modify the Quickselect algorithm so that it computes the weighted median. Give psuedocode that implements your algorithm.

Problem 15. (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 16. (Advanced) Implement the median-of-medians algorithm for Quickselect as described in the lecture notes. Use this algorithm to select a pivot for Quicksort and compare its performance against the previous pivot-selection methods.

Problem 17. (Advanced) Write an algorithm that given two sorted sequences $a[1..n]$ and $b[1..m]$ of unique integers finds the k^{th} order statistic of the union of a and b

1. Your algorithm should run in $O(\log(n)\log(m))$ time
2. Your algorithm should run in $O(\log(n)+\log(m))$ time