University of Waterloo
CS240 - Spring 2026
Assignment 2

Due Date: Tuesday June 9, 5pm

Please read the following link for guidelines on submission:

`https://student.cs.uwaterloo.ca/~cs240/assignments.phtml#guidelines`

Late Policy: Assignments are due at 5:00pm, with a grace period until 7:59pm.

Question 1 [1+4+5 marks]

We want to design a variant of MergeSort that divides the array A into k subarrays rather than two, with the division taking place at indices n/k, $2n/k$, ..., $(k-1)n/k$. Here, k is a fixed constant, which we will take of the form $k = 2^s$ (s integer), to simplify some calculations.

In this problem, *we only count comparisons between elements in our arrays*, and *we give upper bounds on the worst-case runtime*. We will deviate slightly from the asymptotic point of view that hides constants: we want to understand the impact of the choice of k, so you will have to give explicit inequalities, not big-Os.

(a) With $k = 2$, show that you can merge two sorted arrays A_1 and A_2 into a single sorted array A using at most n element comparisons, if we assume that A_1 and A_2 both have length $n/2$ (you can assume n even).

(b) More generally, for k of the form $k = 2^s$, show that you can merge k sorted arrays $A_1, \ldots, A_k$ into a single sorted array A using at most $ns = n\log_2(k)$ element comparisons, if we assume that $A_1, \ldots, A_k$ all have length n/k (you can assume n is a multiple of k).

(c) Give the variant of MergeSort that subdivides the input array into k subarrays, and give an explicit (not asymptotic) upper bound on the number of element comparisons it does. You can assume that n is a power of k.

Question 2 [1+3+4+5 marks]

We want to implement a *double-ended priority queue*. This is a collection where we can insert elements, access the minimum or maximum and remove the minimum or maximum; our goal is do to insert, delete-min, delete-max in worst-case time $O(\log(n))$ (if there are n elements in the collection) and find-min, find-max in constant time.

To support these requirements, we use two heaps H_1 (a min-heap) and H_2 (a max-heap) stored as arrays. At all times, these heaps should contain the same set of elements, but stored

in a different order. You should also use two arrays T_1 and T_2 that specify the correspondence between the elements of H_1 and H_2: $T_1[i]$ gives the index of $H_1[i]$ in H_2, and conversely $T_2[i]$ gives the index of $H_2[i]$ in H_1.

(a) Explain how to implement find-min and find-max, and justify the runtime.

(b) Give an algorithm to update the arrays T_1 and T_2 if we swap the elements of indices i and j in H_1 (we do not worry whether swapping these two elements breaks the heap condition in H_1). Give (and justify) the cost of this operation, and a brief justification of its correctness.

(c) Give an algorithm to insert a new key in the data structure. Give (and justify) the worst-case cost of this operation, and a brief justification of correctness. Big-Os are sufficient.

(d) Give algorithms to do delete-min and delete-max. Give (and justify) the worst-case cost of these operations, and a brief justification of correctness. Big-Os are sufficient.

Question 3 [2+4+5 marks]

Consider n distinct keys $k_0, \ldots, k_{n-1}$ stored in an array A of length n, with array indices ranging from 0 to $n-1$. We consider the problem of finding the first key in A that satisfies a certain condition `test`(k), using the following algorithm:

find(A)

1: $n = \text{length}(A)$
2: **for** $i = 0, \ldots, n-1$ **do**
3: **if** `test`$(A[i])$ **then**
4: **return** $A[i]$
5: **end if**
6: **end for**
7: **return** "not found"

In the questions below, we are interested in the *average* cost of this procedure, where the average is taken over the $n!$ permutations σ of $\{0, \ldots, n-1\}$. For the cost analyses, we only count how many times we call `test` at step 3 of the algorithm.

(a) Suppose that no key in the array satisfies `test`. What is the average number of tests we do? Give the exact number (and justify your answer).

(b) Suppose now that there are exactly $s \leq n$ keys in A that satisfy `test`. For j in $\{0, \ldots, n-s\}$, prove that there are exactly

$$\binom{n-s}{j} j! s (n-j-1)!$$

permutations σ of $\{0, \ldots, n-1\}$ for which in the corresponding array A,

- `test`$(A[0]), \ldots,$ `test`$(A[j-1])$ are all false
- `test`$(A[j])$ is true

How many tests do we do for these permutations? What about $j > n - s$?

(c) Still under the assumption of the previous question, give the average number of key comparisons we do, in terms of n and s (we want the exact value). We recommend you use the following equalities

$$\sum_{j=0}^{n-s}(j+1)\binom{n-j-1}{s-1} = \sum_{j=0}^{n-s}(j+1)\frac{(n-j-1)!}{(n-j-s)!(s-1)!} = \binom{n+1}{s+1},$$

which you don't have to prove (the first one is trivial, the second one not so much). You can do this questions without completing the previous one.

The following exercises are provided for additional practice and are not part of the graded portion of the assignment. While these questions will not be evaluated, we strongly encourage you, and expect you, to work through them to deepen your understanding of the material. You are welcome to upload your solutions to Crowdmark for your own record-keeping and organization; however, no grades or feedback will be provided for these submissions. You may also ask questions about these exercises during office hours or on Piazza.

Question 4

A *stable* sorting algorithm is one in which the relative order of all identical entries is the same in the output as it was in the input. Suppose for instance that we sort pairs of keys / values (k, v), using only the key k for comparisons, and that objects $(k, v), (k, v')$ with identical keys are allowed to show up; then, if (k, v) comes before (k, v') in the input, it should come before (k, v') in the output.

Is heapsort, as seen in class, a stable sorting algorithm? If so, prove it; if not, give a counter-example.

Question 5

Suppose we are working with a max heap, represented as an array, and we want to remove an element from it that is not necessarily the root. We are given the index i of this element in the array. Describe an algorithm that performs this task, give the pseudo-code, analyse its worst-case complexity in terms of the number n of elements in the heap (give a big-O), and briefly justify correctness (you can take for granted that the algorithms for insert and delete in a heap are correct without reproving them, of course). Note: if your algorithm runs in time $\Omega(n)$ in the worst case, you will not receive full marks.

Question 6

Let A be an array of n distinct integers. An *inversion* is a pair of indices (i, j) such that $i < j$ and $A[i] > A[j]$.

(a) Determine the maximum number and minimum number of inversions in an array of n distinct integers. Explain what the arrays that attain these maxima and minima look like.

(b) Given a pair of distinct indices (i, j), determine the number of permutations for which (i, j) is an inversion (do not just give the number; we want a justification).

(c) Determine the average number of inversions in an array of n distinct integers. The average is computed over all $n!$ permutations of the n integers in A. Hint: indicator variables indexed by i, j.

(d) Give the average runtime of insertion sort (as a Theta bound), assuming that in size n you run it over all $n!$ permutations of $1, \ldots, n$.