University of Waterloo
CS240 - Winter 2021
Assignment 2

There are written questions and a programming question in this assignment.

- the deadline for **all written questions, from Problems 1 to 6 inclusive,** is Wednesday February 10, 5pm;
- the deadline to submit your file `kWayMergeSort.cpp` **from Problem 7** is Wednesday February 24, 5pm.

The integrity of the grade you receive in this course is very important to you and the University of Waterloo. As part of every assessment in this course you must read and sign an Academic Integrity Declaration (AID) before you start working on the assessment and submit it **before the deadline of February 10** along with your answers to the assignment; i.e. **read, sign and submit A02-AID.txt now or as soon as possible**. The agreement will indicate what you must do to ensure the integrity of your grade. If you are having difficulties with the assignment, course staff are there to help (provided it isn't last minute).

The Academic Integrity Declaration must be signed and submitted on time or the assessment will not be marked.

Please read `http://www.student.cs.uwaterloo.ca/~cs240/w21/guidelines/guidelines.pdf` for guidelines on submission. **Each written question solution must be submitted individually to MarkUs as a PDF** with the corresponding file names: a2q1.pdf, a2q2.pdf, ..., a2q6.pdf.

It is a good idea to submit questions as you go so you aren't trying to create several PDF files at the last minute. **Remember, late assignments will not be marked but can be submitted to MarkUs after the deadline for feedback if you email cs240@uwaterloo.ca and let the ISAs know to look for it.**

Notes:

- Logarithms are in base 2, if not mentioned otherwise.
- If you use MSD or LSD radix sort in this assignment, you are free to chose any value for radix R you wish. If you chose radix R other than 10 (standard decimal representation), you must account for the time it takes to transform a number in decimal representation to base R.

Problem 1 [2 + 4 + 4 marks]

a) Let H be a max-heap of size 13 with unique elements. On which level(s) of the heap can the 3rd largest item be located? On which level(s) of the heap can the smallest item be located? Recall that the topmost level in the heap is 0.

b) Module 2 describes an implementation of *heapify* that calls *fix-down* for array indices in non-increasing order from $parent(last(n))$ down to 0. Consider *badHeapify* which calls *fix-down* on the same array indices, but in the reverse order. Prove that the algorithm below is incorrect. To prove this, provide an array A of size 6 such that *badHeapify*(A) fails to put A in heap order. Draw the corresponding binary tree for output array A and explain why this tree is not a heap. Assume max-heap. Thus the order of *fix-down* operations is critical for the correctness of the *heapify* algorithm.

```
BadHeapify(A)
A: an array
1.    n ← A.size()
2.    for i ← 0 to parent(last(n))
3.          fix-down(A, n, i)
```

c) Suppose H is a max-heap storing keys. Assume that the heap is implemented as a binary tree. Write an algorithm *HeapSearch*(v, k) that takes as an input the root node of the heap v, and a key k and returns the node of the heap storing k, if k is present in the heap, or $NULL$ if k is not in the heap. Let s be the number of keys in the heap that are larger than k. Your algorithm must have complexity $O(1 + s)$.

Problem 2 [5 marks]

You are given an array A of size n storing integers. Write an algorithm $reorder(A)$ that reorders the elements of the array so that odd numbers are in odd positions while even numbers are in even positions. If there are more even elements than odd ones in A (or vice-versa) then those additional elements should be placed at the end of the array. For example, with an initial array $A = [47, 50, 78, 76, 7, 60]$ the result could be: $A = [50, 47, 78, 7, 76, 60]$. Your algorithm must be in-place and must run in $O(n)$ time.

Problem 3 [4 + 4 marks]

Consider the algorithm below, where $coinFlip()$ returns an integer uniformly sampled from $\{0, 1\}$.

```
HalfMin(A, m)
A: an array A of size n, 0 ≤ m < n
1.   if m = 0 return A[0]
2.   middle = m/2
3.   a ← A[middle]
4.   i ← middle + 1
5.   while i ≤ m
6.        if A[i] < a
7.             a = A[i]
8.        i ← i + 1
9.   if coinFlip() = 0
10.       return a
11.  else
12.       b = HalfMin(A, m/2)
13.       return min{a, b}
```

a) Derive the recurrence equation $T(n)$ for computing the expected number of comparisons of elements in array A (i.e. comparisons at line 6) during the execution of $HalfMin(A, n-1)$.

b) Solve the recurrence equation you derived in part (a). You can assume divisibility as convenient.

Problem 4 [4+3] marks

You have found a treasure chest filled with $n \geq 3$ coins. Exactly 2 coins are counterfeit, the rest are genuine. All genuine coins weigh the same and the two counterfeit coins weigh the same, but a counterfeit coin weighs less than a genuine coin. Your task is to separate the genuine coins from the counterfeit coins. To accomplish this you will compare the weight of pairs of subsets of the coins using a balance scale. The outcome of one weighing will determine that each subset of coins weighs the same, or that one or the other subset of coins weighs more.

a) Give a precise (not big-Omega) lower bound for the number of weighings required in the worst case to determine which coins are genuine and which are counterfeit for $n \geq 3$. Briefly justify. Bounds in $o(\log n)$ will not be eligible for partial credit.

b) Describe an algorithm called `FindGenuine` to determine the genuine coins when n = 4. Use the names C_1, C_2, C_3, C_4 for the four coins, and the function

$$CompareWeight\left(\{first\_subset; second\_subset\}\right),$$

which returns 1 if $first\_subset$ weighs more, 0 if both subsets weigh the same and -1 if $second\_subset$ weighs more. Your function should return the set of genuine coins. Give an exact worst-case analysis of the number of weightings required by your algorithm. For full marks, this should match exactly the lower bound from Part (a).

Problem 5 [6 marks]

Let $A[0...n-1]$ be an unsorted array of positive integers, in the range of $(0, ..., n^5)$. Entries in A are not necessarily unique. Design an algorithm that tests whether there are two numbers in A that are exactly ten apart, i.e. $|A[i] - A[j]| = 10$ for some indexes $i, j \in \{0, 1, ...n-1\}$. For example, the algorithm should return "yes" if the array is $A = [4, 1, 10, 6, 5, 11]$, because numbers 1 and 11 are 10 units apart. It should return "no" on $A = [10, 5, 13, 1, 14, 2]$. The worst-case run-time of your algorithm must be $O(n)$. You can use $O(n)$ additional space. Describe the idea of your algorithm in plain language (you may also give pseudo-code if you wish), and briefly justify correctness.

Problem 6 [1+2+2+(4)+4 marks]

In this problem we will look at an alternative average case analysis for *quicksort*1. Recall that in this version of quicksort, the last array element is the pivot. Assume that the input array A is of size n and contains all distinct integers in the range form 0 to $n-1$. Instead of Hoare's partition, we use the first version of partition from module03, which is stable: if a and b stay in the same subarray after partition, then their order is the same as before partition. For example, array $A = [4, 1, 3, 0, 2]$ is partitioned into $[1, 0]$ and $[4, 3]$. Recall that running time of *quicksort*1 is proportional to the number of comparisons performed during partition. We will derive $O(n \log n)$ bound in the average case by counting average number of comparisons.

a) Let $A = [4, 2, 5, 0, 6, 1, 3]$. List all the pivots for *quicksort*1, in the order of their occurrence.

b) Let $A = [4, 2, 5, 0, 6, 1, 3]$. During the execution of *quicksort*1 on this array, will 1 be compared to 6? Will 4 be compared to 6?

c) Let $n = 4$. List all permutations of the input for which *quicksort*1 will compare 1 to 3. You can use wildcard '*' to denote a set of inputs. For example $[*, *, *, 2]$ denotes a subset of inputs where the last element is 2 and all other elements are in arbitrary order. Do you notice any pattern to these permutations? Pay attention to where 2, the only element between 1 and 3, is positioned.

d) (bonus) Let $0 \leq i < j < n$. Prove that there are $\frac{2n!}{j-i+1}$ permutations where i will be compared to j.

e) Show that the average number of comparisons in *quicksort*1 over all input instances is $O(n \log n)$. You can use the result from (d) even if you did not prove it and the fact that $\sum_{i=1}^{i=k} \frac{1}{i} \leq \log k + 1$.

Problem 7 Programming question [11+1 marks]

Submission deadline for this question is Wednesday, February 24, 5pm.

`MSDRadixSort` has a disadvantage of many recursive calls. We can modify `MSDRadixSort` to stop recursing once the subarray size gets sufficiently small, and run insertion sort at the end of your algorithm. For this problem, you are to implement this modified version of `MSDRadixSort`. Your method should stop making recursive calls when the current array size is smaller than or equal to `threshold`. Note `threshold=1` means standard, unmodified MSDRadixSort.

You may use C++ vectors and any member functions for vectors. No other std library data structures are allowed. You are not allowed to use any built-in functions for sorting.

We give you (on the assignment webpage) a starter file `MSDRadixInsertionSort.cpp`. You can compile it (using the `c++17` standard), but notice that the body of the function `MSDSort` is empty. Do not change its signature. You should complete this file and submit it once you are done (you can write new functions).

Function `MSDSort` takes as an input a vector `A` of unsigned long integers to sort, integer `m` which specifies the maximum number of digits in any number in array `A` (set to 10 currently), integers `l` and `r` which specify between which array indexes, inclusively, to sort, `d` which is the next digit to sort by (leading digit is 0), `threshold` which is used to stop recursing, and finally, `total_num_calls`, an integer (passed by reference) which tracks the total number of calls to `MSDRadixSort`. Your implementation of `MSDRadixSort` should set `total_num_calls` to the correct number. For example, on array `A[123, 232, 021, 320, 210, 230, 101]` and with `m=3`, `total_num_calls = 9`.

The `main` function reads from `cin` the number of integers to sort on the first line, the value of parameter `threshold` on the second line, and then the integers to sort written on a separate lines. You can assume that each line contains a single integer written in base 10. The `main` function calls `MSDRadixSort` (which does nothing for the moment), and prints the array after execution, and the value of `total_num_calls` on the last line. We also provide a sample input / output (you can pipe `input1.txt` into your program; once everything is implemented, you should obtain what is in `output1.txt`). Do not change anything in the main function you submit (in particular, leave the `#ifndef` and `#endif` we put there).

Run your program on a randomly generated array of size $100,000$ for each of $threshold \in \{1, 5, 10, 50, 100, 1000, 10000\}$. Use the included function `getRandomArray` to generate a random array. In the main function, you can set `read_input_from_cin` to `false` to switch from reading from `cin` to generating a random array. Note and report the running times in each case. What is the optimal value of `threshold`? Submit the results of this experiment in file 'experiment.pdf'.

Programming part is worth 11 marks. The experimental plots are worth 1 mark.