University of Waterloo
CS240 Spring 2021
Assignment 2

Due: Wednesday, June 9 at 5:00pm

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 before you start working on the assessment and submit it **before the deadline of June 9th** along with your answers to the assignment; i.e. **read, sign and submit A02-AcInDe.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/s21/guidelines.pdf` for guidelines on submission. **Each question 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.
Late assignments, however, can be reviewed for **feedback only** upon request to the ISAs at cs240@uwaterloo.ca.
Note: you may assume all logarithms are base 2 logarithms: $\log = \log_2$.

Problem 1 [4 + 6 = 10 marks]

Consider a heap implemented with an array. Let the kth ancestor of node at index i of the array be the ancestor of i that is separated from i by k edges. For example, the parent of i is the 1st ancestor of i.

a) Prove by induction that the kth ancestor of node at index i, if present, is stored at index $\left\lfloor \frac{i+1}{2^k} - 1 \right\rfloor$. You can use the following equality without a proof: if n, m are integers, and n is a positive integer, then for a real number x,

$$\left\lfloor \frac{x-m}{n} \right\rfloor = \left\lfloor \frac{\lfloor x \rfloor - m}{n} \right\rfloor.$$

b) Suppose we have a max-heap implemented with an array H and storing n keys. Design an algorithm $isAncestor(x, H, i)$ which takes as an input heap array H, a valid index

i in the array and x. This algorithm should return true if key x is stored at one of the ancestors of node i, false otherwise. The running time of your algorithm should be $O(\log \log n)$. You can use the result from part (a) even if you did not do part (a). You can assume that you have values of 2^k for $1 \le k \le \log n$ pre-computed, so that computation of 2^k takes only $O(1)$ time.

Problem 2 [3+3+5+4=15 Marks]

One potential pitfall of *QuickSort* is that it does not necessarily perform well if there are many repeated elements.

a) Assume that you call *QuickSort* on an array of size n where all elements are the same. Derive (with an explanation) an asymptotically tight bound on the run-time, presuming you always use the simple partition-algorithm, listed below.

```
partition(A, p)
A: array of size n,   p: integer s.t. 0 ≤ p < n
      Create empty lists small, equal, and large.
      v ← A[p]
      for each element x in A
            if x < v append x to small
            else if x > v append x to large
            else append x to equal
      i ← size(small)
      j ← size(equal)
      Overwrite A[0...i−1] by elements in small
      Overwrite A[i...i+j−1] by elements in equal
      Overwrite A[i+j...n−1] by elements in large
      return i
```

b) Assume that you call *QuickSort* on an array of size n where all elements are the same. Derive (with an explanation) an asymptotically tight bound on the run-time, presuming you use Hoare's partition-algorithm from class (listed below).

partition(A, p)
A: array of size n, p: integer s.t. $0 \le p < n$
1. *swap*$(A[n-1], A[p])$
2. $i \leftarrow -1$, $j \leftarrow n-1$, $v \leftarrow A[n-1]$
3. **loop**
4.   **do** $i \leftarrow i+1$ **while** $A[i] < v$
5.   **do** $j \leftarrow j-1$ **while** $j \ge i$ and $A[j] > v$
6.   **if** $i \ge j$ **then break** (goto 9)
7.   **else** *swap*$(A[i], A[j])$
8. **end loop**
9. *swap*$(A[n-1], A[i])$
10. **return** i

c) One possible improvement to *QuickSort* is to modify *partition* so that it returns three subsets: The left part has items $< v$, the middle part has items $= v$, and the right part has items $> v$. Describe how to modify Hoare's algorithm to achieve this. In particular, fill in the pseudo-code (and explain it) for the following stub:

ThreeWayPartition(A, p)
A: array of size n, p: integer s.t. $0 \le p < n$
$v \leftarrow A[p]$
...
return(i, j)
// $A[0..i-1]$ has items $< v$, $A[i..j]$ has items $= v$, $A[j+1..n-1]$ has items $> v$

Your algorithm must have worst-case run-time $O(n)$ and be *in-place*, i.e., use $O(1)$ additional space.

d) Describe how *ThreeWayPartition* can be useful for *QuickSort* when repeated elements are allowed. In particular, what modifications would you make to the following pseudo-code from class?

QuickSort1(A)
A: array of size n
if $n \le 1$ **then return**
$p \leftarrow$ *choosePivot1*(A)
$i \leftarrow$ *partition*(A, p)
QuickSort1$(A[0, 1, \ldots, i-1])$
QuickSort1$(A[i+1, \ldots, n-1])$

Your modifications should be such that if the input array has k distinct elements, then the *worst-case* run-time of the algorithm is $O(kn)$. Argue that this holds.

Problem 3 [8 marks]

A student designed a data structure and named it an `almost-priority-queue`. This data structure allows two operations: `insert` and `extract_almost_Max`, where `extract_almost_Max` outputs either the largest priority or the second largest priority item. Also, `extract_almost_Max` does not tell you whether it extracted the largest or second largest priority item. In case the data structure has only one element, `extract_almost_Max` extracts that element. The student claims that the worst case running time of both `insert` and `extract_almost_Max` is $o(\log n)$. Prove that the student has made a mistake in the running time analysis of their data structure.
Hint: Sorting takes $\Omega(n \log n)$ time.

Problem 4 [3+3+5=11 marks]

Consider the algorithm below, where *random(n)* returns an integer from the set of $\{0, 1, 2, \ldots, n-1\}$ uniformly and at random. Array A stores non-repeating integers in the range $\{0, 1, 2, \ldots, n-1\}$, and k is an integer between 0 and $n-1$.

$ArrayAlg(A, n, k)$
A: array of size n

    $i \leftarrow random(n)$
    if $A[i] == k$ **then return** i
    else
        for $j = 0$ **to** n **do**
            print('*')
        $ArrayAlg(A, n, k)$

a) What is the best-case running time of *ArrayAlg*?

b) What is the worst-case running time of *ArrayAlg*?

c) Let $T(n)$ be the expected running time of *ArrayAlg*. Write a recurrence relation for $T(n)$ and then solve it. Express your answer using Θ notation.

Problem 5 [6 marks]

Let $R_1, \ldots, R_n$ be n axis-aligned rectangles in the plane for which the corners are points in the $n \times n$-grid. Thus, for each rectangle R_i the four corners are points where both coordinates are integers in $\{1, \ldots, n\}$. Degenerate rectangles (i.e. rectangles of height or width zero) are allowed. Give an algorithm to sort $R_1, \ldots, R_n$ by increasing area in $O(n)$ time.

Problem 6 [6 marks]

Let A be an array of size n storing numbers $0, 1$. It is known that the array starts with 0 ends with 0, and that all 1's are consecutive. A valid example is $A = [0, 1, 1, 1, 0, 0]$. Give an exact (not asymptotic) lower bound on the number of comparisons required to find the smallest index i and largest index j s.t. $A[i] = 1$ and $A[j] = 1$. You can assume that $n \geq 3$.