15-451/651 Algorithms, Spring 2018

Homework #2 **Due: February 6–9, 2018**

This is an oral presentation assignment. Again, there are three regular problems (#1-#3) and one programming problem (#4). You should work in groups of three. The sign-up sheet will be online soon (details on Piazza) and your group should sign up for a 1-hour slot. Each person in the group must be able to present every regular problem. The TA/Professor will select who presents which regular problem. You are not required to hand anything in at your presentation, but you may if you choose.

The programming problem will be submitted to autolab, similar to HW#1. You will not have to present anything orally for the programming problem. You can discuss the problem with your group-mates, but must write the program by yourself. Please do not copy.

The bonus problem B2 will be released soon.

(25 pts) 1. **(A Median of Sorts.)** Let A and B be two *sorted* arrays of n elements each. We can easily find the median element in A — it is just the element in the middle — and similarly we can easily find the median element in B. (For any k, define the median of $2k$ elements as the element that is greater than $k-1$ elements and less than k elements.) However, suppose we want to find the median element overall — i.e., the nth smallest in the *union* of A and B. How quickly can we do that? You may assume that all the $2n$ elements in A union B are distinct.

Your job is to give tight upper and lower bounds for this problem. Specifically, for some function $f(n)$,

(a) Give an algorithm whose worst-case running time (measured in terms of number of comparisons) is $O(f(n))$, and

(b) Give a lower bound showing that any comparison-based algorithm must make $\Omega(f(n))$ comparisons in the worst case.

(c) Now, get rid of the O and Ω to make your bounds *exactly* tight in terms of the number of comparisons needed for this problem. I.e., show upper and lower worst-case bounds that are *exactly the same function*, not even off by one.

Some hints: You may wish to try small cases. For the lower bound, you should think of the output of the algorithm as being the location of the desired element (e.g, "A[33]") rather than the element itself. How many different possible outputs are there?

An aside: a solution to part (c) subsumes part (a) and (b). However, we think it is helpful when solving a problem like this to first think about the growth rate before getting into the exact constants. (And you can get partial credit.)

(25 pts) 2. **(Duck Soup: a Stueue? Or a Quack?)**

(a) We want to maintain a max-stack, which is a data structure that supports the following operations.

- `push`(number x), pushes x on the stack
- `pop`, pops the top element off the stack
- `return-max`, returns the maximum number among the elements still on the stack. (Does not push or pop anything.)

How would you implement a max-stack, while maintaining constant time per operation (worst-case).

Clarifications: You are allowed to allocate infinitely large arrays, etc. On an empty stack, return `NULL` on a `pop` or `return-max`. (Similar clarifications apply to the following parts.)

(b) We want to now maintain a max-queue, which is what you'd expect given the previous definition. It supports the following operations.

- `enqueue`(number x), adds x to the end of the queue
- `dequeue`, removes the element at the front of the queue
- `return-max`, returns the maximum number among the elements still in the queue. (Does not enqueue or dequeue anything.)

Show how to use two max-stacks to implement a max-queue. This max-queue should take $O(1)$ amortized time per operation. That is, a sequence of n operations (consisting of some number of `enqueue`, `dequeue` and `return-max` operations in any order) should take total time $O(n)$.

(c) Consider the streaming setting, where you are making one pass over the stream $a_1, a_2, \ldots, a_n$, with each a_i being a number. You are given a number $r \geq 0$. For each time t, when you see a_t, you want to output the maximum value among the past r elements $a_{t-r+1}, a_{t-r+2}, \ldots, a_t$. E.g., if $r = 4$ and the input stream is

$$3, 17, 3, 9, 2, 0, 7, 2, 8, 9, 1, 8, 4, 5, 9$$

then the output stream should be

$$3, 17, 17, 17, 17, 9, 9, 7, 8, 9, 9, 9, 9, 8, 9$$

Give an algorithm that makes one pass over a stream of n numbers to produce the desired output stream, and the total time taken is $O(n)$. You are allowed enough space to store $O(r)$ elements. (Note that taking time $O(rn)$ would be trivial by just storing the most recent r elements and recomputing the max from scratch each time – you want to do better than that.)

(25 pts) 3. **(Every Day I am Hashin'.)**

We say that H is *ℓ-universal over range m* if for every fixed sequence of ℓ distinct keys $\langle x_1, x_2, \ldots, x_\ell \rangle$, if we choose a hash function h at random from H, the sequence $\langle h(x_1), h(x_2), \ldots, h(x_\ell) \rangle$ is equally likely to be any of the m^ℓ sequences of length ℓ with elements drawn from $\{0, 1, \ldots, m-1\}$. It's easy to see that if H is 2-universal then it is universal. (*Check for yourself!*)

Consider a universe U of strings $s = s_1, s_2, \ldots, s_n$ of length n from an alphabet of size k. (Each character is an integer in $\{0, 1, \ldots, k-1\}$.) Hence $|U| = k^n$. Assume that $m = 2^b$.

An interesting universal family $\mathcal{G}$ (of functions from U to $\{0, \ldots, m-1\}$) can be obtained as follows. First, generate a 2-dimensional table T of b-bit random numbers; recall that $b = \lg(m)$. The first index of $T_{i,j}$ is in the range $[1, n]$ and the second index is in the range $[0, k-1]$. Now define the hash function $g_T()$ as follows:

$$g_T(s) = \bigoplus_{i=1}^{n} T_{i,s_i}$$

where "$\bigoplus$" represents the bitwise-xor function (recall, each $T_{i,j}$ is a b-bit string). The output of $g_T(s)$ is a b-bit string which is then interpreted as a number in $\{0, \ldots, m-1\}$. Note that since each choice of the table T gives a hash function g_T, and T is specified by $n \cdot k \cdot b$ bits, the family $\mathcal{G}$ consists of 2^{nkb} functions.

(a) Prove that $\mathcal{G}$ is *not* 4-universal.

Hint: *To show that $\mathcal{G}$ is not 4-universal, you should exhibit 4 distinct keys $\langle x_1, x_2, x_3, x_4 \rangle$ such that if you were told the values of $g_T(x_1)$, $g_T(x_2)$, and $g_T(x_3)$, you could infer the value of $g_T(x_4)$ uniquely (without knowing anything else about T). This will mean that not all 4-tuples of hash-values are equally likely, since the first 3 entries in the tuple $\langle g_T(x_1), g_T(x_2), g_T(x_3), g_T(x_4) \rangle$ determined the 4th entry. You can do this using $n = 2$ and $k = 2$.*

(b) Prove that $\mathcal{G}$ *is* 3-universal.

(25 pts) 4. **Programming: MSTs and Union-Find**

In this problem you will write a program that takes an undirected graph $G = (V, E)$ where each edge has a non-negative edge length $\ell(e)$, and outputs the weight of the minimum spanning tree, and the weight of the heaviest edge in this tree. The graph has no parallel edges or self-loops.

You must use Kruskal's algorithm, and the union-find data structure (with union-by-rank and path-compression) we introduced in lecture.

INPUT: The first line contains n and m the number of vertices and edges of the graph, where $1 \leq n \leq 1000$ and $1 \leq m \leq 10000$. The following m lines each contain a pair of vertex numbers in the range $[0, n-1]$, and a non-negative integer representing the length of the edge.

OUTPUT: The first line is the weight of the minimum spanning tree. The second has the weight of the heaviest edge in this tree.

Input	Output
3 3 0 1 9 1 2 7 0 2 10	16 9
4 5 0 1 1 1 2 4 2 3 2 0 3 3 1 3 1	4 2