15-451/651 Algorithms, Spring 2019

Homework #2 **Due: February 5–8, 2019**

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 is due **Sunday Feb 10th**, 11:59pm, and should 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.

(25 pts) 1. **(The Counter Strikes Again!)** We saw in class that if a binary counter begins at 0, and we perform n increments, the amortized cost per increment is $O(1)$ (i.e., the total cost is $O(n)$). Suppose we want to be able to increment *and* decrement the counter.

(a) Show a sequence of n operations allowing both increments and decrements and starting from 0 that, without ever making the counter go negative, costs $\Omega(\log n)$ amortized per operation (i.e., $\Omega(n \log n)$ total cost).

To reduce the cost observed in part (a) consider a counter using the following *redundant ternary number system*. A number is represented by a sequence of *trits*, each of which is 0, +1, or -1. The value of the number represented by $t_{k-1}, \ldots, t_0$ (where each $t_i, 0 \le i \le k-1$ is a trit) is defined to be

$$\sum_{i=0}^{k-1} t_i 2^i.$$

For example, $\boxed{1}\boxed{0}\boxed{-1}$ is a representation for $2^2 - 2^0 = 3$.

To increment a ternary number: You add 1 to the low order trit. If the result is 2, then it is changed to 0, and a carry is propagated to the next trit. This process is repeated until no carry results. Decrementing a number is similar. You subtract 1 from the low order trit. If it becomes -2 then it is replaced by 0, and a borrow is propagated. Note that the same number may have multiple representations (e.g., $\boxed{1}\boxed{0}\boxed{1} = \boxed{1}\boxed{1}\boxed{-1}$) — it's a *redundant* ternary number system. The cost of an increment or a decrement is the number of trits that change in the process.

(b) Starting from 0, a sequence of n increments and decrements is done. Give a clear, coherent proof that with this representation, the amortized cost per operation is $O(1)$ (i.e., the total cost for the n operations is $O(n)$).

(25 pts) 2. **(The Quickest Quack)**

(a) We want to maintain a *max-stack*, a data structure with 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`/`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 $r \geq 1$. 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. **(An Interesting Hash Family.)**

We say that H is *ℓ-universal over range m* (or *ℓ-wise independent*) 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 ~~binary~~ 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) (10 pts) 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) (15 pts) Prove that $\mathcal{G}$ *is* 3-universal. (You can get 7 of these points by proving the weaker statement that it is 2-universal.)

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

In this problem you will write a program that takes a connected undirected graph $G = (V, E)$ where each edge has a non-negative edge weight $w(e)$, 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. The edge weights are distinct.

You must use Kruskal's algorithm, and the union-find data structure (with union-by-rank and path-compression) we introduced in lecture. When doing union-by-rank, for `link(i,j)` for two roots i and j with the same rank, make `i` the parent of `j` if $i < j$. The identity of a node in the union-find is the same as the number of the graph vertex it represents.

INPUT: The first line contains n and m the number of vertices and edges of the graph, where $2 \leq n \leq 10^3$ and $1 \leq m \leq 10^6$. The following m lines each contain a pair of vertex numbers in the range $[0, n-1]$, and a non-negative integer representing the weight of the edge. Each weight is in $[0, 10^9]$, so the total weight of the tree may be as high as $\approx 10^{12}$. (Use `long`s to store the total weight.)

OUTPUT: The first line is the weight of the minimum spanning tree. The second has the weight of the heaviest edge in this tree. The third is the sorted position of the heaviest edge in the tree, i.e., if $w(e_1) < w(e_2) < \ldots < w(e_m)$ are the weights, and e_j

is the heaviest edge in the tree then the third line is j. The fourth is the depth[1] of the node numbered $\lfloor n/2 \rfloor$ in the union-find tree data structure just after adding this heaviest edge into the MST.

Input	Output
3 3	16
0 1 9	9
1 2 7	2
0 2 10	0
4 5	8
0 1 1	5
1 2 5	4
2 3 8	1
0 3 2	
1 3 3	

Hints: Please be careful to follow the specifications above. Be sure to review the notes on union-find, which details which of the function calls trigger path compression. Also please generate some sample inputs/outputs to check your program.

[1] Depth of root = 0, depth of any other node = 1 + depth of its parent.