15-451/651 Algorithms, Spring 2018

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

(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.)

Solution: The upper and lower bounds are exactly $\lceil \log_2 2n \rceil$. For brevity, however, we will assume here that n is a power of 2. Also, to make the math simpler, let's index the arrays beginning with 1.

(a) Compare $A[n/2]$ to $B[n/2]$. Say that $A[n/2] < B[n/2]$. Notice that the only elements that can possibly be smaller than $A[n/2]$ are $A[1] \ldots A[n/2-1]$ and $B[1] \ldots B[n/2-1]$ (which is a total of just $n-2$ elements). So, $A[n/2]$ is less than the median overall. This means we can delete $A[1] \ldots A[n/2]$ as possible answers. Also, $B[n/2]$ must be at least the nth smallest overall since $A[1] \ldots A[n/2]$ and $B[1] \ldots B[n/2-1]$ *are* smaller than it. This means that nothing larger than it could possibly be the right answer, so we can delete $B[n/2+1] \ldots B[n]$. So, we recursively find the $(n/2)$th smallest element in the union of the arrays $A' = A[n/2+1] \ldots A[n]$ and $B' = B[1] \ldots B[n/2]$. Notice that we deleted the same number of elements above and below the median, so we continue to be searching for the median recursively.

We used one comparison to cut the problem size in half. As a base case, when $n = 1$, we take one final comparison to find the element we want. So this solves to $T(n) = 1 + \lg n = \lg(2n)$.

(b) For the lower bound, if the algorithm makes at most t comparisons, then there are at most 2^t possible outputs it can produce (since there are at most 2^t possible series of answers possible to the comparisons made). So, if $2^t < 2n$ then there must be some location out of the $2n$ that is never output by the algorithm. But this means the algorithm cannot be correct because we can also show that for any index i, there exist arrays A and B such that $A[i]$ is the unique correct answer, and similarly arrays A and B such that $B[i]$ is the unique correct answer. For example, consider filling A and B with the numbers $\{1, 2, \ldots, 2n\}$. To make $A[i]$ be the correct answer, put n into $A[i]$, fill $A[1], \ldots, A[i-1]$ with entries less than n, fill $A[i+1], \ldots, A[n]$ with entries greater than n, and then put whatever is left into B. So, $\lceil \log_2 2n \rceil$ is a lower bound.

(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.)

Solution: Have an array with a top pointer as in the usual array-based implementation of a stack, but give each element of the array two fields: one for the item and one for the max so far. When pushing a new item x, we set `A[++top].item = x` as usual, and then update the maximum by setting `A[top].max = max(A[top-1].max, x)`. By induction, we maintain the invariant that `A[i].max` is the maximum out of `A[0].item`, ..., `A[i].item` for all values of i from 0 to `top`. (And this invariant is clearly maintained on a pop as well). Thus, the `return-max` operation can just return `A[top].max`.

(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 $O(n)$.

Solution: we implement the queue using two stacks exactly as in the problem from Quiz #1, using `push` and `pop` operations from part (a) to implement the dump operation (that dumps one stack into the other). This ensures that both stacks maintain the invariant from part (a), namely `A[i].max` is the maximum out of `A[0].item`, ..., `A[i].item`. To implement `return-max` we just call `return-max` on each of the two stacks and return the larger value.

(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.)

Solution: We just use the queue from part (b). For the first r entries, we just do an `enqueue(x)` and `return-max`. For the rest we do `enqueue(x)`, `dequeue()` and then `return-max`.

(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.

Solution: For instance, consider the following 4 keys of length 2 (so $n = 2$): 00, 11, 01, 10. Whatever T is, these keys have the property that $g_T(00) \oplus g_T(11) \oplus g_T(01) \oplus g_T(10) = 0$ since each of the four entries in T is xor'ed twice. This means that the hash of any one of the strings can be determined from the hash of the other three, so there is no way that all 4-tuples of hash-codes are equally likely.

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.

Solution: To show why $\mathcal{G}$ is 3-universal, consider three distinct keys x, y, z. We now consider two cases:

Case 1: there is some index i such that all three keys differ in that index. We now argue as follows. Consider choosing all of T except for $T_{i,*}$. Thus, filling in this row of T will determine the hash codes for x, y, and z. We first fill in T_{i,x_i}: since each value for this

entry produces a different value for $g_T(x)$, we have that x is equally likely to hash to all possible b-bit values. Now $g_T(x)$ is fixed. We now fill in T_{i,y_i}: again, since each value for this entry produces a different value for $g_T(y)$, we have that y is equally likely to hash to all possible b-bit values, even conditioning on everything done so far. This means that x and y's hash values are independent. Now $g_T(x)$ and $g_T(y)$ are fixed. We now fill in T_{i,z_i}: again, since each value for this entry produces a different value for $g_T(z)$, we have that z is equally likely to hash to all possible b-bit values, even given everything so far. So all triples of hash values are equally likely.

Case 2: there is no index i such that all three keys differ in that index. In this case, choose some index i such that $x_i \neq y_i$ (this must exist since $x \neq y$). We know z_i matches one of the other two, so say without loss of generality that $z_i = y_i$. Choose some other index j such that $z_j \neq y_j$. We know that x_j matches one of those two values so say without loss of generality that $x_j = y_j$. We can now argue as in Case 1. First fill in all of T except for $T_{i,x_i}, T_{i,y_i}, T_{j,z_j}$. Now, filling in T_{i,x_i} determines x's hash (all equally likely), then filling in T_{i,y_i} determines y's hash (all equally likely, no matter what x hashed to), then filling in T_{j,z_j} determines z's hash value (all equally likely, no matter what x, y hashed to)