Homework 2 – COT5405 *

Kobee Raveendran

September 2021

1. **Prove or disprove that a HEAP can support the following operations in o(log n) time: (i) insert(x): insert an element x into the HEAP and (ii) `extract_min_max`: extract either the minimum or the maximum element (the HEAP won't tell if the extracted element is the min or max) [Hint: Prove by contraction via sorting lower bound].**

2. **A binomial heap can perform both insertions and extract min operations in O(log n) worst-case time. By proposing an appropriate potential function, show that the amortized cost of insertion is O(1) and the amortized cost of extract min is O(log n).**

3. **[Range Query Data Structure]: Let S be a (dynamic) set of items, where each item is associated with a unique "key" (a positive number) and a unique "weight" (a positive number). Therefore each item can be represented as a (key, weight) pair. Design an O(n) space data structure that can support the following operations on S in O(log n) time, where n is the cardinality of S: (i) insert a new item into S (ii) RangeQuery[a,b]: report the item with the highest weight among all items in S with the key within the range [a,b].**

4. **Design an O(k log k) time algorithm to find the kth smallest number from a set of n numbers arranged in the form of a binary min-heap (where n ¿¿k). Improve the time complexity to O(k log log k). Note that the trivial solution takes O(k log n) time (i.e., by simply doing extract min k times). [Hint]**

5. **Recall how we solved the following problem in class: Given a (read-only) array A[1,n] of n numbers, if it exists, find the number that appears more than n/2 times. Note that the time complexity for that approach is O(n). Also note that A is "read-only," and the auxiliary space used (i.e., to maintain two variables) is O(1). Now, let's generalize this problem: Given a (read-only) array A of n numbers and an integer $k > 1$, determine the numbers that appear more than n/k times, if they exist. Design an O(n log k) time solution that uses only O(k) auxiliary space.**

   We can solve this problem in a manner similar to the one discussed in class, just modified to track more than one possible frequently-occurring element. The algorithm would go like this:

   (a) initialize an auxiliary data structure like an associative binary search tree (or hash map to give us a better expected runtime but worse worst-case runtime) of size $k-1$ to hold the elements that can *potentially* occur more than $\frac{n}{k}$ times. Note that if $k = 2$, we'd arrive at the problem discussed in class and the array would be of size 1 (to hold the one element appearing more than $\frac{n}{2}$ times).

   (b) Like the algorithm discussed in class, walk through the array element by element, but now we track $k - 1$ elements by mapping the key (the element of the array) to a "count-like" value. The first time the element is seen, this value would be 1.

   (c) Again similar to the base algorithm, we'd increment a key's count if we see it again, but only if its key already exists in the array. If it doesn't (i.e. this is the first time we're seeing the element), we'd make a new key in the array for it (also initialized to 1).

*For some questions, I received guidance from or collaborated with: TA (office hours)

(d) In the case that the array is already filled and we can't make a new key for a new unique element, we'd instead decrement the values of *all* keys in the array by 1. And as was the case in the base algorithm, if the count of a key ever reaches 0, we remove it (this is analogous to what we did before, in that it will allow a new, potentially unique element into the "top-k-frequently-occurring" group, and previously that group had a size of 1).

(e) Continue this process as described in steps 2 - 4 (inclusive) until the end of the input array is reached. At this point, the array of $k-1$ elements will contain all the candidates for occurring more than $\frac{n}{k}$ times; however, it's not guaranteed that *every* element in that array will occur more than $\frac{n}{k}$ times (for example, the last element in the array could've entered the array just once at the very end and taken a slot thus only having a count of 1).

(f) Because of this, it's necessary to again scan the input array to confirm whether the final $k-1$ candidates actually occur more than $\frac{n}{k}$ times. We'd simply iterate over the input array again and maintain counts of only the final candidates; any that occur less than or equal to $\frac{n}{k}$ times can be safely discarded. The remaining candidates will be our final result.

This approach runs in $O(n \log k)$ time with $O(k)$ space if we use a binary search tree/sorted array approach for storing the size $k-1$ candidate pool. This is because, for every element in the array, we need to find whether its key exists by performing a binary search in the tree, which will take $O(\log k)$ time. If the key doesn't exist, we'd have to create it in the tree (which also takes $O(\log k)$ time). Incrementing/decrementing the count attribute of a key would be a constant-time operation. Thus, for all n elements in the array, we'd incur $O(\log k)$ work, giving us a runtime of $O(n \log k)$. The scan to confirm the validity of the final $k-1$ candidates would also take $O(n \log k)$ time since for all n elements, we do a binary search to determine whether it is a final candidate, and if it is, maintain a count for it.

If we instead use a hash map, we'd get an expected runtime of $O(n)$ with $O(k)$ space, as each lookup or key insertion into the size $k-1$ hash map would be amortized $O(1)$. Of course, in the worst case, the runtime would be $O(n \cdot k)$ due to the possibility of hash collisions.