CSCI 570 - Fall 2019 - HW 3 Solution

1 Graded Problems

1. Design a data structure that has the following properties (assume n elements in the data structure, and that the data structure properties need to be preserved at the end of each operation):

   - Find median takes $O(1)$ time
   - Extract-Median takes $O(\log n)$ time
   - Insert takes $O(\log n)$ time
   - Delete takes $O(\log n)$ time

   Do the following:

   (a) Describe how your data structure will work

   (b) Give algorithms that implement the Extract-Median() and Insert() functions.

   Hint: Read this only if you really need to. Your Data Structure should use a min-heap and a max-heap simultaneously where half of the elements are in the max-heap and the other half are in the min-heap.

   Solution: We use the $\lceil n/2 \rceil$ smallest elements to build a max-heap and use the remaining $\lfloor n/2 \rfloor$ elements to build a min-heap. The median will be at the root of the max-heap and hence accessible in time $O(1)$ (we assume the case of even n, median is $(n/2)^{\text{th}}$ element when elements are sorted in increasing order).

   Insert() algorithm: For a new element x,

   (a) Compare x to the current median (root of max-heap).

   (b) If $x <$ median, we insert x into the max-heap. Otherwise, we insert x into the min-heap. This takes $O(\log n)$ time in the worst case.

   (c) If size(maxHeap) $>$ size(minHeap)+1, then we call Extract-Max() on max-heap and insert the extracted value into the min-heap. This takes $O(\log n)$ time in the worst case.

   (d) Also, if size(minHeap) >size(maxHeap), we call Extract-Min() on min-heap and insert the extracted value in to the max-heap. This takes $O(\log n)$ time in the worst case.

   Extract-Median() algorithm: Run ExtractMax() on max-heap. If after extraction, size(maxHeap) $<$ size(minHeap) then execute Extract-Min() on the min-heap and insert the extracted value into the max-heap. Again worst case time is $O(\log n)$.

2. There is a stream of integers that comes continuously to a small server. The job of the server is to keep track of k largest numbers that it has seen so far. The server has the following restrictions:

   (a) It can process only one number from the stream at a time, which means it takes a number from the stream, processes it, finishes with that number and takes the next number from the stream. It cannot take more than one number from the stream at a time due to memory restriction.

(b) It has enough memory to store up to k integers in a simple data structure (e.g. an array), and some extra memory for computation (like comparison, etc.).

(c) The time complexity for processing one number must be better than $\Theta(k)$. Anything that is $\Theta(k)$ or worse is not acceptable.

Design an algorithm on the server to perform its job with the requirements listed above.

Solution: Use a binary min-heap on the server.

(a) Do not wait until k numbers have arrived at the server to build the heap, otherwise you would incur a time complexity of $O(k)$. Instead, build the heap on-the-fly, i.e. as soon as a number arrives, if the heap is not full, insert the number into the heap and execute Heapify(). The first k numbers are obviously the k largest numbers that the server has seen.

(b) When a new number x arrives and the heap is full, compare x to the minimum number r in the heap located at the root, which can be done in $O(1)$ time. If $x \leq r$, ignore x. Otherwise, run Extract-min() and insert the new number x into the heap and call Heapify() to maintain the structure of the heap.

(c) Both Extract-min() and Heapify() can be done in $O(\log k)$ time. Hence, the overall complexity is $O(\log k)$.

3. When we have two sorted lists of numbers in non-descending order, and we need to merge them into one sorted list, we can simply compare the first two elements of the lists, extract the smaller one and attach it to the end of the new list, and repeat until one of the two original lists become empty, then we attach the remaining numbers to the end of the new list and it's done. This takes linear time. Now, try to give an algorithm using $\mathcal{O}(n \log k)$ time to merge k sorted lists (you can also assume that they contain numbers in non-descending order) into one sorted list, where n is the total number of elements in all the input lists. (Hint: Use a min-heap for k-way merging.)

   Solt. Construct a min-heap of the k sublists, that is each sublist is a node in the constructed min-heap. When comparing two sublists, compare their first elements (that is their minimum elements). The creation of this min-heap will cost $\mathcal{O}(k)$ time.

   Then we run the extract min algorithm and extract the minimum element from the root list. Then update the root sublist and heapify the min-heap according to the new minimum element in the root sublist. (If the root sublist becomes empty during this step, take any leaf sublist and make it the root and then heapify). Each extraction takes $\mathcal{O}(\log k)$ time.

   Since we extract n elements in total, the running time is $\mathcal{O}(n \log k + k) = \mathcal{O}(n \log k + k)$ (since $k < n$).

4. Suppose you are given two sets A and B, each containing n positive integers. You can choose to reorder each set however you like. After reordering, let a_i be the i-th element of set A, and let b_i be the i-th element of set B. You then receive a payoff on $\prod_{i=1}^{n} {a_i}^{b_i}$. Give an algorithm that will maximize your payoff. Prove that your algorithm maximizes the payoff,and state its running time.

   Solt. Sort both A and B in the same order (either both ascending or both descending). This takes $\mathcal{O}(nlogn)$. What's important is that the largest element of A is matched with the largest of B, the second-largest of each are matched, and so on (greedy solution).

   For purposes of the proof, fix the order of A to be in ascending order, i.e., $a_1 \leq a_2 \leq \cdots \leq a_n$; we will consider all solutions in terms of how their B arranged relative to this A. My solution has B sorted ascendingly.

   For any arbitrary solution (could be an optimal solution): $\{b_i\}$, where b_i is the element of B matched

with a_i in this solution. Suppose there exist an inversion $b_i > b_j$, for $i > j$. Now we prove that undoing this inversion will make the result no less than the original one, and has the effect of multiplying the original solutions quantity by this value:

$$\frac{{a_i}^{b_j} \times {a_j}^{b_i}}{{a_i}^{b_i} \times {a_j}^{b_j}} = \frac{{a_j}^{b_i - b_j}}{{a_i}^{b_i - b_j}}$$

Because $\frac{a_j}{a_i} \geq 1$ and $b_i - b_j > 0$, the result above should be no less than 1, i.e., every inversion relative to this solution can be removed without negatively affecting the quantity, which indicates the optimality of the greedy solution.

2 Practice Problems

1. The police department in a city has made all streets one-way. The mayor contends that there is still a way to drive legally from any intersection in the city to any other intersection, but the opposition is not convinced. A computer program is needed to determine whether the mayor is right. However, the city elections are coming up soon, and there is just enough a time to run a linear-time algorithm.

   a Formulate this as a graph problem and design a linear-time algorithm. Explain why it can be solved in linear time.

   b Suppose it now turns out that the mayors original claim is false. She next makes the following claim to supporters gathered in the Town Hall: "If you start driving from the Town Hall (located at an intersection), navigating one-way streets, then no matter where you reach, there is always a way to drive legally back to the Town Hall." Formulate this claim as a graph problem, and show how it can also be varied in linear time

   Solt.

   (a) The mayor is merely contending that the graph of the city is a strongly connected graph, denoted as G. Form a graph of the city (intersections become nodes, one-way streets become directed edges). Suppose there are n nodes and m edges. The algorithm is:

      i. Choose an arbitrary node s in G, and run BFS from s. If all the nodes are reached the BFS tree rooted at s, then go to step ii, otherwise, the mayors statement must be wrong. This operation takes time $\mathcal{O}(m+n)$.

      ii. Reverse the direction of all the edges to get the graph G^{inv}, this step takes time $\mathcal{O}(m+n)$.

      iii. Do BFS in G^{inv} starting from s. If all the nodes are reached, then the mayors statement is correct; otherwise, the mayors statement is wrong. This step takes time $\mathcal{O}(m+n)$.

      Explanation: BFS on G tests if s can reach all other nodes; BFS on G^{inv} tests if all other nodes reach s. Suppose you can reach node v by BFS on G^{inv}. It means there is a path from s to v in the G^{inv}. Now by reversing all edges of sv path, you will get a path from v to s in the G. There If these two conditions are not satised, the mayors statement is wrong obviously; if these two conditions are satised, any node v can reach any node u by going through s.

   (b) Now the mayor is contending that the intersections which are reachable from Town Hall form a strongly-connected component. Run the rst step of the previous algorithm while setting the Town Hall as s (test to see which nodes are reachable from Town Hall). Remove any nodes which are unreachable from the town hall, and this is the component which the mayor is claiming is strongly connected. Run the previous algorithm on this component to verify it is strongly connected.

2. You are given a weighted directed graph $G = (V, E, w)$ and the shortest path distances $\delta(s, u)$ from a source vertex s to every other vertex in G. However, you are not given $\pi(u)$ (the predecessor pointers). With this information, give an algorithm to find a shortest path from s to a given vertex t in $O(|V|+|E|)$

   Solution1:

(a) Starting from node t, go through the incoming edges to t one at a time to check if the condition $\delta(s,t) = w(u,t) + \delta(s,u)$ is satisfied for some $(u,t) \in E$.

(b) There must be at least one node u satisfying the above condition, and this node u must be a predecessor of node t.

(c) Repeat this check recursively, i.e. repeat the first step assuming node u was the destination instead of node t to get the predecessor to node u, until node s is reached.

Complexity Analysis: Since each directed edge is checked at most once, the complexity is $O(|V|+|E|)$. Note that constructing the adjacent list of G^{rev} in order to facilitate access to the incoming edges is needed, which is still bounded by $O(|V| + |E|)$ complexity.

Solution 2:Run a modified version of Dijkstras algorithm without using the priority queue. Specifically,

(a) Set $S = \{s\}$.

(b) While $t \notin S$

i. For each node u satisfying $u \notin S$, $(v,u) \in E$ and $v \in S$, check if the condition $\delta(s,u) = w(v,u) + \delta(s,v)$ is true.

ii. If true, add u to S, and v is the predecessor of u.

iii. EndWhile

Complexity Analysis: Since each directed edge is checked at most once, the complexity is $O(|V| + |E|)$.