CSCI 570 - Fall 2019 - HW 3

Junzhe Liu, 2270250947

September 16, 2019

1 Graded Problems

1.1 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

1.1.1 Describe how your data structure will work

We use the $[n/2]$ smallest elements to build a max-heap and use the remaining $[n/2]$ elements to build a min-heap. The median will be at the root of the max-heap and hence accessible in time $O(1)$. This data structure could be build up by sorting the array first, and then construct two heaps respectively.

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

Suppose there are no two elements are equal, every element has different values. Insert algorithm:

- Compare the new element x to the current median (root of max-heap)
- If $x <$median, we insert x into the max-heap. Otherwise, we insert it into the min-heap. Renew the inserted heap to maintain its property. This takes $O(\log n)$ time in the worst case.
- If size(min-heap) = size(max-heap) + 2, then put $[n/2] + 1$ element (the root of min-heap) into max-heap, renew two heaps to maintain the max / min property.
- If size(max-heap) = size(min-heap) + 2, then swap median element (the root of max-heap) with the element at position $[n/2]$(the last element of max-heap), in this way, we put the (previous) median element into the min-heap. Renew two heaps takes $O(\log n)$.

Extract-Median algorithm: Suppose n is even, then max-heap and min-heap has the same size, Extract the root of max-heap (position 0) or the root of min-heap (position $[n/2] + 1$) will both do.

Suppose n is odd, then the root of the longer heap is the median. Extract element at position 0 when size(max-heap) > size(min-heap) and element at position $(n + 1)/2$ when size(max-heap) < size(min-heap).

After extraction, maintaining the extracted heap takes $O(\log n)$ operations, therefore the complexity is also $O(\log n)$.

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

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

Takes the first k numbers and construct a min-heap. Whenever it takes a new number x, compare x with the root number r, if $x > r$, then replace r with x, renew the heap. Each process takes $O(\log k)$ operations at most, thus it's better then $\Theta(k)$.

1.3 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 $O(n \log k)$ 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.)

Initially, extract the first number from each lists and construct a min-heap, the heap size is k, the construction complexity is $O(k)$. Then the algorithm runs a loop: during each iteration, we takes the root of the heap and puts it into the result array, and the put the next element of the array where the root element comes from into the heap. Extraction takes $O(1)$, and maintaining the heap takes $O(\log k)$ operations at most. Therefore the overall complexity will be:

$$O(k) + O(n) * (O(1) + O(\log k)) = O(n \log k) \tag{1}$$

1.4 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.

Algorithm 1 Maximizing Payoff

1: sort A in non-increasing order $a_1 \geq a_2 \geq \ldots \geq a_n$.
2: sort B in non-increasing order $b_1 \geq b_2 \geq \ldots \geq b_n$.
3: pair a_i with b_i, return $\prod_{i=1}^{n} a_i^{b_i}$.

Proof that this is the optimal method: We denote that the product of $a_i^{b_i}$ is P_i, that is to say $P_i = \prod_{i=1}^{n} a_i^{b_i}$. Suppose there is an optimal method O that maximizes the payoff, the optimal payoff is O_i. We'll prove that $P_i \geq O_i$, so that our algorithm is one of the optimal methods.

In each iteration, we always choose the largest number in both arrays and multiply. Therefore for $i = 1$, our algorithm will choose the largest a_i and b_i such that: $a_1 = \max a_i$ and $b_1 = \max b_i$, obviously we'll have that $P_1 = a_1^{b_1} \geq O_1$. Suppose that this relation still satisfies when $i = r : P_r \geq O_r$. When $i = r+1$, P_{r+1} chooses $a_{r+1} = \max A/\{a_1, \dots a_r\}$ and $b_{r+1} = \max B/\{b_1, \dots b_r\}$, suppose that O_{r+1} chooses $\alpha_{r+1}, \beta_{r+1}$, apparently $\alpha_{r+1} \leq a_{r+1}, \beta_{r+1} \leq b_{r+1}$. Since $P_r \geq O_r$, therefore $P_{r+1} = P_r \times a_{r+1}^{b_{r+1}} \geq O_r \times \alpha_{r+1}^{\beta_{r+1}} = O_{r+1}$.

This could be proved in another way: Suppose A, B are both sorted in a non-increasing order, and there is another method S that pairs a_1 with b_r, and a_r with b_1. Notice that $a_1 \geq a_r,\ b_1 \geq b_r$, we consider the product of two methods:

$$\frac{\text{Prod(P)}}{\text{Prod(S)}} = \frac{a_1^{b_1} \cdot a_r^{b_r}}{a_r^{b_1} \cdot a_1^{b_r}} = \left(\frac{a_1}{a_r}\right)^{b_1 - b_r} \geq 1 \tag{2}$$

We now have proved that our algorithm will give the maximized payoff.

2 Practice Problems

2.1 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.

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

Consider the intersections as nodes, and the streets connect them as directed edge, we can simplify this problem as a problem to determine whether a directed-graph is strongly-connected.

Algorithm 2 Determine a directed-graph is strongly-connected or not

1: Starting from a random node S, use BFS or DFS on graph G to find all the nodes it can reach. If it cannot reach all the nodes, then G is not strongly-connected, quit, otherwise, proceed to step 2.
2: Construct a conjugate graph $G^{'}$ by reversing all the edge directions.
3: Starting from the same node S, use BFS or DFS on $G^{'}$ to find all the nodes it can reach. If it cannot reach all the nodes, then G is not strongly connected, otherwise, it is.

In the step 1 of our algorithm, if some point N is reached, then it means that starting from node S, N could be approached via some certain route. In step 2, if some node V can be reached in $G^{'}$, then it actually means that V could reach S in the origin graph G via the exact reverse route that connects S with V in the conjugate graph $G^{'}$. If all the nodes could be found in step 1&2, then each nodes can reach each other via point S. In another word, graph G is strongly-connected.

Since this algorithm travels the graph twice, takes $O(|V| + |E|)$, construct a conjugate graph takes $O(|E|)$, therefore the overall complexity will be $O(|V| + |E|)$, it's a linear-algorithm.

2.1.2 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.

This problem can actually be transformed into question 1, the only different is that we now have given a certain point S^* rather than a random node S on the graph. The algorithm should change a little bit in order to solve this one:

Algorithm 3 Determine starting from a given point you can go to somewhere and get back

1: Starting from a given node S^*, use BFS or DFS on graph G to find all the nodes it can reach. The reachable points are saved in the node set V
2: Construct a conjugate graph $G^{'}$ by reversing all the edge directions.
3: Starting from the same node S^*, use BFS or DFS on $G^{'}$ to find all the nodes it can reach. If it cannot reach all the nodes in set V, then the claim is wrong, otherwise it's correct.

Same as the proof in question 1, this algorithm can be conducted within linear time.

2.2 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 $O(|V| + |E|)$

Starting from node t, go through the incoming edges to t one at a time to check if the condition:

$$\delta(s, t) = \delta(s, u) + w(s, u) \tag{3}$$

is satisfied for some edge $< u,\ t >\in E$. There must exist at least one node which satisfies the above condition, we denote the set which includes all the predecessor points as U_1, and take one of the nodes u_i to continue such process: test the incoming edges to u_i one at a time to see if it satisfies $\delta(s, u_i) = \delta(s, v) + w(v, u_i)$, if there does not exist such point v for our chosen u_i, then try select another u in the U_1.

By continue this process, we can obtain a chain of sets: $U_1, U_2, \ldots, U_k$. If we find there's no points satisfying the above criterion, we'll go back to the previous point set U_{k-1} and select another point and continue our algorithm.

If the source point s is included in some predecessor point set U_r, then the algorithm stops. The shortest route from s to t will be :

$$s \to u_r \to \ldots \to u_2 \to u_1 \to t \tag{4}$$

Complexity: Each edge is checked at most once, therefore the complexity is $O(|V| + |E|)$.