HW4

▼ **Question 1**

Problem 1

[10 points] 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 $\mathcal{O}(1)$ time
- Insert takes $\mathcal{O}(\log n)$ time

Do the following:

1. Describe how your data structure will work.
2. Give algorithms that implement the Find-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

Consider the following algorithm -

```
public static class MedianPriorityQueue {
    PriorityQueue<Integer> left;
    PriorityQueue<Integer> right;

    public MedianPriorityQueue() {
      left = new PriorityQueue<>(Collections.reverseOrder());
      right = new PriorityQueue<>();
    }

    public void Insert(int val) {
      if (right.size() > 0 && val > right.peek()) {
          right.add(val);
      } else {
          left.add(val);
      }

      if (left.size() - right.size() == 2) {
          right.add(left.remove());
      } else if (right.size() - left.size() == 2) {
          left.add(right.remove());
      }
    }

    public int Find_Median() {
      if (this.size() == 0) {
          return -1;
      }
      if (left.size() >= right.size()) {
          return left.remove();
      }
```

```
      return right.remove();
    }

    public int peek() {
      if (this.size() == 0) {
          System.out.println("Underflow");
          return -1;
      }
      if (left.size() >= right.size()){
          return left.peek();
      }
      return right.peek();
    }

    public int size() {
      return left.size() + right.size();
    }
  }
```

Working of data structure explained

1. The data structure is named MedianPriorityQueue and is made using a Min-Heap and a Max-Heap simultaneously. (PriorityQueue java implementation has been used for implementing the min and max heap.
2. We will start by inserting elements to the min-heap. We will insert data into the max-heap only when the the difference in the size of the min-heap and the max-heap exceeds 2. Same will be the case when inserting data into the min-heap. This ensures equal distribution of data into both the heaps.
3. For a new value, if the max-heap size is > 0 and the value of to be inserted in greater than the root of max-heap, data will be inserted in the min-heap. Otherwise, data is inserted in the min-heap by default
4. We know, insertion in a PQ (PriorityQueue) takes $O(log(n))$ time. Since MedianPriorityQueue uses a PQ itself internally, the insertion time will be $O(log(n))$.
5. In order to find the median, we will just peek the max-heap to get the median. Data retrieval takes constant time for PQ. Hence $O(1)$ time will be taken for finding the median using the MedianPriorityQueue data structure.

> 💡 Note: Assumption has been made that in case when n is even, the median will be smaller of the 2 middle nodes.

▼ Question 2

Problem 2

[10 points] 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:

- 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 $\mathcal{O}(k)$. Anything that is $\mathcal{O}(k)$ or worse is not acceptable.

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

Solution

We define a min_heap (array implementation) with the size being k. The server will process only one number at a time (after k elements have are already present in the heap) and the min_heap will always contain k largest numbers in the array.

Since a heap is a complete binary tree, a new element will always be added at the end of the array. Let's say that a new element is added at index i. Now, the following holds true -

> 💡 parent is present at $i/2$
>
> left child is present at $2 * i + 1$
>
> right child is present at $2 * i + 2$

As this is a min_heap and we add a new element at index i, we check whether $arr[i]$ is greater than $arr[i/2]$ and if it is, we swap the elements at indices i and $i/2$ which is a constant time operation. Hence, processing a single number takes $O(1)$ constant time due to the swap operation.

Pseudocode

```
We create a min_heap with max size as k.
Procedure: build_heap() {
  insert elements to the server one by one (as per the server restrictions) in an array implementation of the heap.
  // let this array be appropriately named, min_heap[k], where k is the size of the array
  when an element is inserted at position (k - 1) we check whether the number that has been added, satisfies the heap property i.e
  if (min_heap[k - 1] > min_heap[(k - 1) / 2]) {
    swap(min_heap[k - 1], min_heap[(k - 1) / 2]) // O(1) constant time required for this operation.
  }
}
```

▼ **Question 3**

Problem 3

[10 points] Suppose you are given a connected graph G, with edge costs that are all distinct. Prove that G has a unique minimum spanning tree.

Solution

Proof by contradiction -

Let there by 2 minimum spanning trees T and T' with all edge costs as unique in the graph G. Now, since the MSTs are unique with same number of edges, there must be an edge e' in T' which does not exist in T. Adding this edge e' in T, we get a cycle and T is no longer an MST (since T was an MST and MSTs are only minimally acyclic - meaning that addition of a single edge will lead to cycle creation in the MST). Let this cycle be C. Let e be the edge in T with the most weight, in which case, e should not be a part of any MST (as an MST is created only when the edge with most weight is removed from the graph - Cycle Property). This contradicts our assumption that edge e must be present in at least one of T or T'.

This contradiction means that there is no edge different in T and T' which means T = T' and hence G has a unique minimum spanning tree when all the edge costs are unique.

▼ **Question 4**

Problem 4

[10 points] Let us say that a graph $G = (V, E)$ is a near tree if it is connected and has at most $n+8$ edges, where $n = |V|$. Give an algorithm with running time $\mathcal{O}(n)$ that takes a near tree G with costs on its edges, and returns a minimum spanning tree of G. You may assume that all edge costs are distinct.

Solution

Cycle Property - It states that in a graph G if there exists a cycle C , then C cannot contain an edge e which has the maximum weight. - As proved in Kleinberg and Tardos (4.20)

We have to make use of the Cycle property 9 times, which will be done in the following way -

1. Perform a BFS traversal on the graph G to find a cycle. Once a cycle is found, remove the most expensive edge e.
2. This does not change the identity of the minimum spanning tree and the graph is still connected - by the Cycle property as mentioned below.
3. The total number of edges have not reduced by 1
4. There are $n + 7$ edges remaining now.

Repeating steps 1 through 4, 8 more times, leaves us with $n - 1$ edges. G is still connected and is a minimum snapping tree (as per the Cycle Property)

Time complexity analysis

Each iteration takes $O(m + n)$ for BFS and finding the most expensive edge in the cycle for removal. Since $m <= n + 8$, we consider the worst case scenario to identify the Big-Oh time complexity - we get $O(n + 8 + n)$ = $O(2n + 8)$ which is the same as $O(n)$

▼ **Question 5**

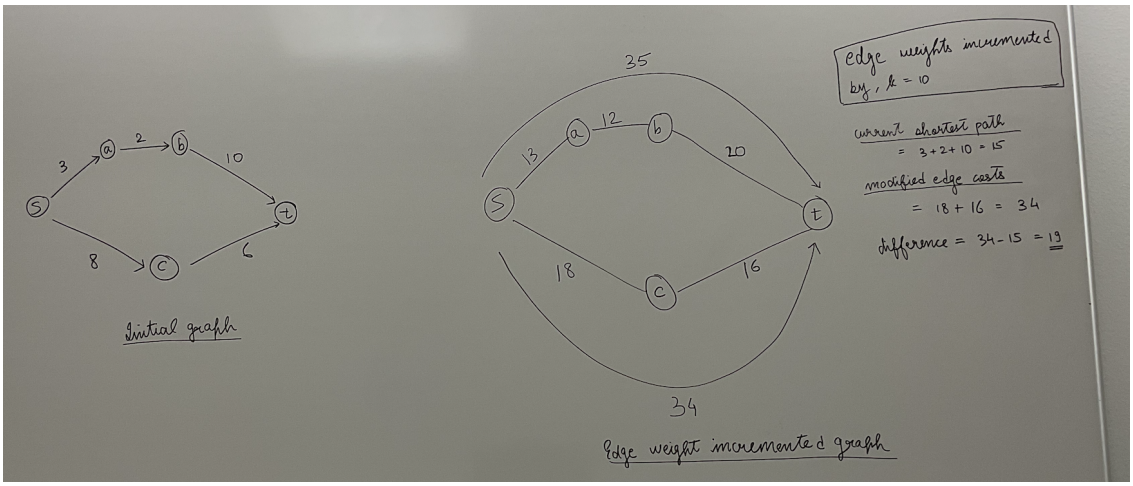
Problem 5

[20 points] Given a connected graph $G = (V, E)$ with positive edge weights and two nodes s, t in V, prove or disprove with explanations:

1. If all edge weights are unique, then there is a single shortest path between any two nodes in V.
2. If each edge's weight is increased by k, the shortest path cost will increase by a multiple of k.
3. If the weight of some edge e decreases by k, then the shortest path cost will decrease by at most k.
4. If each edge's weight is replaced by its square, i.e., w to w^2, then the shortest path will be the same as before but with different costs.

Solution

1. **FALSE** - The addition of multiple edges can sum out to be equal. Consider the following weights for 2 different and unique paths - (3 + 2 + 10 = 15) and (8 + 7 = 15). In this case, there are 2 shortest paths with the same distance from source to destination.
2. **FALSE** - Counter example given in the screenshot below - not only does the shortest path change from $s - a - b - t$ to $s - c - t$, but the value shortest value changes from 15 to 34 when $k = 10$. The difference is 19 which is not a multiple of k



3. **TRUE -** This can be divided into 2 scenarios:
   a. If e is in the shortest path - Let the original shortest path cost be C. Now, since e has been decremented by e' where $e' = e - k$, the total cost also changes to C' where $C' = C - k$, which means that the cost of the shortest path changes by a maximum of k units. In this case, the shortest path remains the same.
   b. If e is NOT in the shortest path - the shortest path cost decreases by a maximum of k. There is a possibility that the shortest path also changes.

Let there be 2 shortest paths in a graph, G.
with costs as c_1 & c_2.

$c_1 = a + b + c$ ——— ①

$c_2 = d + e$ ——— ②

Assumption: $c_1 \leq c_2$ ——— ③

According to question: one of the edges is decreased by k. Let that edge be e

So, $e' = e - k$

This means, from ① & ②

$c_1 \leq c_2$

$a + b + c \leq d + e'$

$a + b + c \leq d + e - k$

$c_1 \leq c_2 - k$

$k \leq c_2 - c_1$

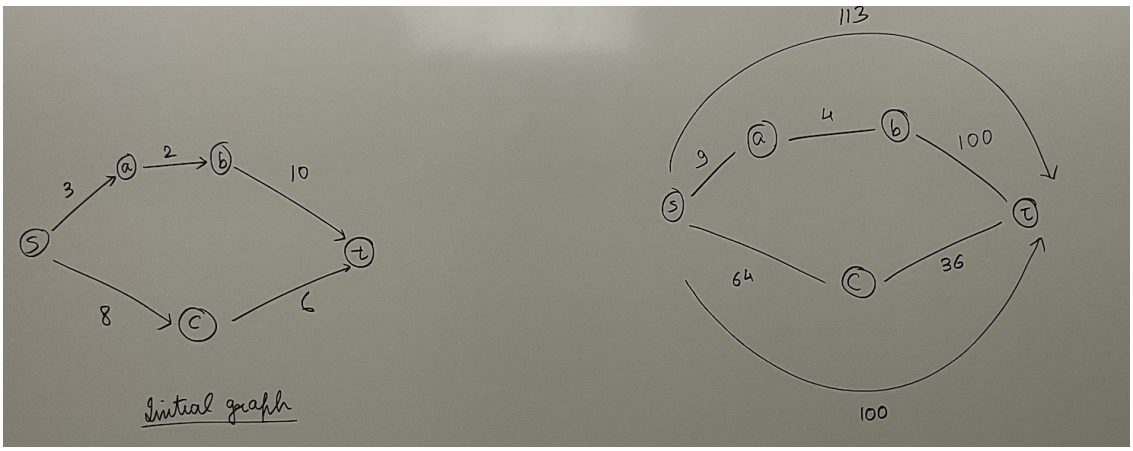
But, according to question, $k > 0$

This leads to change in equality sign.

$\therefore k \geq c_2 - c_1$

& c_2 is now the shortest path

4. **TRUE -** Since all edges are positive (stated in the question) and unique (stated in the question as well), squaring all the edge's weights will increase the edge weights uniformly. Hence, the shortest path remains the same but the shortest path cost changes. Example in the screenshot below - The shortest path before and after the squaring off of the edges remains the same ($s - c - t$) but the cost of this path changes (14 initially, versus 100 after squaring the edges)



▼ **Question 6**

Problem 6

[**16 points**] Consider a directed, weighted graph G where all edge weights are positive. You have one Star, which allows you to change the weight of any one edge to zero. In other words, you may change the weight of any one edge to zero. Propose an efficient method based on Dijkstra's algorithm to find a lowest-cost path between two vertices s and t, given that you may set one edge weight to zero. (**Note:** you will receive 10 points if your algorithm is efficient. You will receive full points (16 points) if your algorithm has the same run time complexity as Dijkstra's algorithm.)

Solution

Consider the following algorithm -

We will have to run the Dijkstra's algorithm twice. Once on the original graph with source as s and next time, with the graph (edge direction reversed) with the source node as t. Let $D(i)$ be the shortest distance from source s to a vertex i. Similarly, let $D_{rev}(i)$ is the shortest distance from a node i to another source t.

Looping through each edge e that connects vertices $v1$ and $v2$, add $D(v1)$ and $D_{rev}(v2)$, which is essentially the shortest path $s \rightarrow v1 \rightarrow v2 \rightarrow t$ (when weight of edge e is zero). The minimum of all these sum values is the shortest path from source s to destination t.

It is better to choose a path which contains a zero weight edge rather than the one where the zero-weight edge is not a part of the shortest path. In case of a path with the zero-weight edge, the shortest path value will be -

s → v1 → v2 → t, where v1 → v2 path cost is zero.

So, shortest path value = $D(v1)$ + $D_{rev}(v2)$

Time Complexity Analysis

Using Dijkstra's Algorithm twice = $O(n + E \cdot log(V))$, assuming E = number of edges and V = number of vertices. The complexity of our algorithm is the same as Dijkstra's.