HOMEWORK 4

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):**
    a. **Find median takes O(1) time**
    b. **Insert takes 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.**

   Ans:
   Description:
   To design this data structure, we will have to take two heaps, one min heap and one max heap. The min heap would store the larger half of the elements while max heap to store the smaller half of the elements. By doing so, the min heap's root element will be the smallest element amongst the larger elements and max heap's root element will be the largest amongst the smaller elements.
   The Insert() function will insert every element in the heaps according to the previous median defined. If the element is less than or equal to the max-heap's root, it is inserted into the max-heap, else, it is inserted into the min-heap.
   We then balance the heaps so that the difference between their size is atmost 1.
   The FindMedian() function checks the size of the heaps. If there are odd number of elements, we display the root node of max heap. If there are even elements, the median is the average of the root nodes of both the heaps.

   Time Complexity:
   The Insert() function inserts elements in min heap and max heap. Insertion of elements in heaps has O(log(n)) time complexity.
   FindMedian() function peeks the root element of the max heap. This has O(1) time complexity as it takes constant time to peek at a element in a heap.
   Hence, the total complexity of this algorithm is O(log(n)).

   Algorithm:
   Create two heaps: min and max heap.
   Define Insert function. This function takes an element, compares it with the current median, and inserts it in one of the two heaps accordingly. If the element is smaller than the root node of the max heap, then it is inserted in the max heap, or else it goes in the min heap.
   Next, check whether the heaps are balanced or not. If not, extract the element from the heap that has more elements and insert it in the heap that has lesser elements. We do this until the difference is either 0 or 1.
   Then, define FindMedian function. In this function, the size of the heaps are compared.

If the length of max heap is equal to the length of min heap, there are even number of total elements. Here, we print the average of the root nodes of both the heaps ((root node of max heap + root node of min heap) / 2).
If the length of either of the heaps is greater than the other, then there are odd number of total elements. Here, we print the root node of max heap.

2. **[20 points] A network of n servers under your supervision is modeled as an undirected graph G = (V, E) where a vertex in the graph corresponds to a server in the network and an edge models a link between the two servers corresponding to its incident vertices. Assume G is connected. Each edge is labeled with a positive integer that represents the cost of maintaining the link it models. Further, there is one server (call its corresponding vertex as S) that is not reliable and likely to fail. Due to a budget cut, you decide to remove a subset of the links while still ensuring connectivity. That is, you decide to remove a subset of E so that the remaining graph is a spanning tree. Further, to ensure that the failure of S does not affect the rest of the network, you also require that S is connected to exactly one other vertex in the remaining graph. Design an algorithm that given G and the edge costs efficiently decides if it is possible to remove a subset of E, such that the remaining graph is a spanning tree where S is connected to exactly one other vertex and (if possible) finds a solution that minimizes the sum of maintenance costs of the remaining edges.**

   Ans:
   In this algorithm, we check if the graph is a spamming tree or not, add additional checks to ensure that the unreliable server S is connected to exactly one other vertex.
   Algorithm:
   Sort the edges in an ascending order. This gives us the sorted list of edges with respect to their costs.
   Choose the graph that only includes unreliable server to start with. This server should remain connected to the network.
   Initialize a for loop to iterate through all the edges. In this loop, we check if adding each link would create a loop in the network or not. If not, and the unreliable server is not already connected to two other servers, then append it to the graph. Update the unreliable server S with respect to its connections. This is continued till all the servers are connected without forming a loop.
   Then we check if the graph is a spanning tree or not. Check if all the servers are connected with minimum cost no loops. If it is a true, then print the tree, else, print "Not a valid tree".

   Hence, this algorithm checks whether if it is possible to remove a subset of E, such that the remaining graph is a spanning tree where S is connected to exactly one other vertex, while also giving us the minimized sum of maintenance costs of remaining edges.

3. **[15 points] Prove or disprove the following:**
   a. **T is a spanning tree on an undirected graph G = (V, E). Edge costs in G are NOT guaranteed to be unique. If every edge in T belongs to SOME minimum cost spanning trees in G, then T is itself a minimum cost spanning tree.**

   Ans:
   False.
   Counterexample:
   Consider a graph G with three vertices {A, B, C}.
   It has three edges with non-unique weights:
   (A-B, 2)
   (B-C, 1)
   (A-C, 2).
   Now, let T be the spanning tree with edges (A-B, 2) and (B-C, 1).

   In this case, T is not a minimum cost spanning tree because there exists another spanning tree with a smaller total weight, which is the tree formed by edges (B-C, 1) and (A-C, 2).
   Hence, the statement that “if every edge in T belongs to some minimum cost spanning trees in G, then T is itself a minimum cost spanning tree” is proven false by this counterexample.

   b. **Consider two positively weighted graphs G = (V, E, w) and G’ = (V, E, w’) with the same vertices V and edges E such that, for any edge e in E, we have w’(e) = w(e)^2 For any two vertices u, v in V, any shortest path between u and v in G’ is also a shortest path in G.**

   Ans:
   False.
   Counterexample:
   Graph G = (V, E, w):
   V = {A, B, C, D, E}
   E = {(A, B), (B, C), (A, C)}
   w((A, B)) = 1
   w((B, C)) = 1
   w((C, E)) = 1
   w((A, D)) = 1
   w((D, E)) = 1.5
   For the path A to E, the smallest path this A to D and D to E = 1 + 1.5 = 2.5.

   Graph G’ = (V, E, w’):
   w’(e) = w(e)^2
   V = {A, B, C, D, E}
   E = {(A, B), (B, C), (A, C)}

After squaring the weights:
w((A, B)) = 1
w((B, C)) = 1
w((C, D)) = 1
w((A, D)) = 1
w((D, E)) = 2.25
Now, the path A to D and D to E is not the shortest path anymore as the total is 3.25 while the path A to B, B to C, C to E is 3.
The shortest path for the graph G' is different than the shortest path for the graph G.

Hence the statement "For any two vertices u, v in V, any shortest path between u and v in G' is also a shortest path in G." is proved false by this counterexample.

4. **[15 points] A new startup FastRoute wants to route information along a path in a communication network, represented as a graph. Each vertex represents a router and each edge a wire between routers. The wires are weighted by the maximum bandwidth they can support. FastRoute comes to you and asks you to develop an algorithm to find the path with maximum bandwidth from any source s to any destination t. As you would expect, the bandwidth of a path is the minimum of the bandwidths of the edges on that path; the minimum edge is the bottleneck. Explain how to modify Dijkstra's algorithm to do this.**

Ans:
Algorithm:
Create a min heap to store the minimum bandwidth.
Initialize a variable to track the minimum bandwidth at any given point.
Initialize a for loop till all the neighboring vertices are checked. In this loop, add these values (minimum bandwidth) to the min heap instead of edge weights by taking the minimum of the 2 bandwidths that are being checked.
End Loop.
Start with vertex s. Extract the highest minimum bandwidth value from the heap.
Initialize a for loop to traverse through all the neighboring vertices till we reach the destination t (or till the min heap is empty). In this loop, calculate the potential minimum bandwidth on the path to v by taking the minimum of the current minimum bandwidth of the extracted vertex and the bandwidth of the edge connecting them.
If the potential minimum bandwidth is greater than the current minimum bandwidth of v, update v's minimum bandwidth and add v to the min heap.
End Loop
The minimum bandwidth found for the destination vertex t is the maximum bandwidth along the path from s to t. Hence, print that path.

In this algorithm, we modify the Dijkstra's algorithm to find the path with maximum bandwidth from any source s to destination d. Dijkstra's algorithm gives us the shortest path using the sum of edge weights. But that doesn't guarantee to get the path that has the maximum bandwidth to the network. To find this, we have to consider path that has minimum bandwidth. Hence, we focus on the path with minimum bandwidth than focusing on path that has the smallest sum of edge weights.

5. **[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 O(k). Anything that is O(k) or worse is not acceptable. Design an algorithm on the server to perform its job with the requirements listed above.**

   Ans:
   For this problem, we use min heap. Min heap would have the smallest element as its root node. Hence, it will be easier for us to compare the new integers with the smallest element, and if the new integer is larger, we replace it with the smallest element of the heap.
   As the integers come continuously to the server, if the length of the heap is less than k, it would be processed directly into the heap. If the length is equal to k, then the integer would be checked with the smallest element of the min heap (root element). If it is larger than the root element, the integer will replace it.
   We print the heap min heap that stores the k largest numbers

   Algorithm:
   Create a min heap
   Define NewNumber function to get the new integer than has come in.
   In this function, first check if the length of min heap is equal to k or not. If no, then insert the new number into the heap. If yes, then compare the new number with the root element.
   If the root element is smaller than the new number, then replace the root element with this number.
   Print the min heap.

Time Complexity:
The time complexity of this algorithm is O(log(k)) due to insertion of k elements into the heap. This is better than O(k) as log(k) is a smaller time period than k.

6. **[15 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 from node s to node t, given that you may set one edge weight to zero. Note: you will receive 10 points if your algorithm is efficient. This means your method must do better than the naive solution, where the weight of each node is set to 0 per time and the Dijkstra's algorithm is applied every time for lowest-cost path searching. You will receive full points (15 points) if your algorithm has the same run time complexity as Dijkstra's algorithm.**

   Ans:
   Here, we run Dijkstra's algorithm twice. First time we run it to find the shortest path of the original graph. Second time, we run it on the modified graph (with one edge weight set to zero) to update the distances.

   Algorithm:
   Initialize a min heap to store the nodes according to their distance.
   Initialize a while loop till the min heap is empty. In this loop, pop the root node from min heap (this gives us the minimum distance at that point in min heap). Calculate the minimum distance from each neighbor node. If this distance is less than the current distance, insert this distance to min heap.
   End Loop.
   Update the weight of one edge to zero.
   Run the above process again for the update weight.
   Compare both the distances. If the distance with the modified edge weight is smaller, it means setting one edge weight to zero has resulted in a shorter path from 's' to 't'.

7. **[10 points] When constructing a binomial heap of size n using n insert operations, what is the amortized cost of the insert operation? Find using the accounting method.**

   Ans:
   Accounting method to find amortized cost of constructing a binomial heap of size n using n insert operations:
   We assign 2 tokens to each element that is been inserted.
   During the first insert operation, 2 tokens are used: 1 for creation and 1 stored as credit. During the second insert operation, 2 tokens are charged again, and 1 token is

used for the operation. The remaining 1 token is added to the credit of the first tree, making it 2 tokens in credit.
Similarly, for n insertions, we have $1+2+3+\ldots+2^{(n-1)}$ of total credit. This is a geometric series that adds up to 2^n-1.
Hence, the total credit after n insert operations is 2^n - 1, and since each operation costs 1 token, the amortized cost per insert operation is (2^n - 1) / n, which is O(1).