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.

7/2 = 3.5
(4)

If n is even,

Build a max heap with $n/2$ smallest element and a min heap with $n/2$ largest elements. The mean of the two roots will be the median

If n is odd.

Buid a max heap with $\lceil n/2 \rceil$ smallest element and a min heap with $\lfloor n/2 \rfloor$ largest elements. The root of max-heap will be the median.

```
Initialize:
    maxHeap = new MaxHeap()
    minHeap = new MinHeap()
    maxlen=0
    minlen=0

Insert(x):
    if x < maxHeap.root() or maxHeap.isEmpty():
        maxHeap.insert(x)
        maxlen+=1.
    else:
        minHeap.insert(x).
        minlen-=1.

    if maxlen > minlen+1:
        value = maxHeap.extractMax().
        minHeap.insert(value)
        maxlen-=1
        minlen+=1
    else if minlen > maxlen:
        value = minHeap.extractMin()
        maxHeap.insert(value)
```

O(logn)

O(logn)

maxlen += 1.
minlen -= 1.

∴ Total O(logn) in worst case for insert.

Find-Median():

O(1)

```
if (maxlen+minlen)%2 == 0:
    return (maxHeap.root() + minHeap.root())/2.
else if maxlen > minlen:
    return maxHeap.root().
else:
    return minHeap.root()
```

∴ O(1) to find median

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.

Step1. Remove S and all edges connected to S from G to create new G'

Step2. Perform DFS on G' to check connectivity.

If G' not connected:

print("it is impossible to create a spanning tree with S having only one neighbour")

Exit().

else:

use prime's algorithm to find the MST of G'. → Step3 go to

Step3: Among all edges that were connected to S in the original graph G, find the edge with the minimum cost. let this edge be (S,v), where v is a vertex in G'.

Step4: Add (S,v) to the MST of G' to incorporate S as a leaf node.

Step5: The resulting graph is the spanning tree with minimized maintenance costs where S is connected to exactly one other vertex.

Step6: Calculate the total maintenance cost.

3. [15 points] Prove or disprove the following:

- 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.
- 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.

(1) True.

Proof by contradiction.

Suppose T is not a minimum cost spanning tree of G, despite every edge in spanning tree T belonging to some MSTs of G.

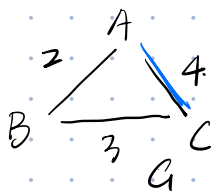
1. Fact: A minimum cost spanning tree of a graph $G=(V,E)$ is a subset of E that connects all vertices in V together, without any cycles, and with the minimum possible total edge cost.

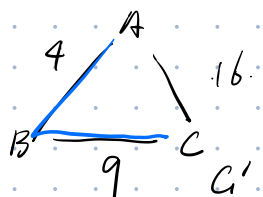
2. From assumption, we know each edge in T is part of some MST of G. which means each edge in T is the minimum cost edge for connecting its vertices considering the fact of MSTs.

3. Assume T is not an MST implies there exists another spanning tree T' of G with a lower total cost than T. However, since T contains only edges that are part of some MSTs, every edge in T is already a minimum cost edge across some cut of the graph.

4. For T' to have a lower total cost than T, T' must include at least one edge with a lower cost than an edge in T for the same cut, which contradicts the assumption that all edges in T are part of some MSTs and thus are the minimum cost edges across their respective cuts.

(2) False.





G: Shortest path from A to C $w(AC) = 4$

G': Shortest path from A to C $w'(AB) + w'(BC) = 13$

They are different.

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.

use max PQ instead of min PQ.

```
S = Null
Initialize max priority queue Q with
  all nodes V where d(v) is the key value.
  ( All d(v)'s are set to 0, except
     for s where d(s) = ∞ )
while S ≠ V
    v = Extract_Max(Q)
    S = S ∪ {v}
    for each vertex u ∈ Adj(v):
        if d(u) < min(d(v), weight(v,u))
            d(u) = max(d(u), min(d(v), weight(v,u)))
            Increase_key(Q, u, d(u))
    end for
end while
```

different from Dijkstra. Initially all nodes have 0. except the starting points is set to ∞.

↳ the bandwidth of a path is the minimum of the bandwidths of the edges on that path.

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.

Use min heap.

Initialize a min heap of size k and store the first k numbers from the stream of integers. insert operation.

=> O(log k) for processing one number.

Compare the (k+1)th number with the root of min-heap.

=> O(1) for processing one number.

If (k+1)th is larger, replace it.

extract-min operation followed by insertion.

=> O(log k) for processing one number.

else

continue

∴ Overall, the time complexity for processing one number is O(log 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.

Step 1: Apply Dijkstra's algorithm starting from the source vertex s to find the shortest paths to all other vertices in graph. $\Rightarrow O(m + n\log n)$ by Fibonacci heap.
(m: edges, n: vertices)

Step 2: Reverse the direction of all edges in the graph and then apply Dijkstra starting from destination vertex t. which finds the shortest path from all vertices to t in the original graph. $\Rightarrow O(m+n\log n)$

Step 3: For each edge $(u, v) \in E$ in graph, set the weight of (u,v) to zero, and calculate the sum of the shortest path $s \to u$, and the shortest path $v \to t$. Do it for every edge to determine which edge, when set to zero, yields the shortest overall path from s to t.

$\Rightarrow m \times O(1) = O(m)$

$\therefore$ time complexity $= 2O(m+n\log n) + O(m)$

since $O(m)$ is dominated by $O(m+n\log n)$

My algorithm has the same run time complexity as Dijkstra's algorithm.

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.

We assign a cost of 2 to each insert operation. This cost is higher than the actual cost of inserting a single node but is chosen to cover the potential future work of merging trees. In this case 1 unit for the insert itself and 1 unit for merge operation.

Not every insert will cause a merge. However, when a merge occurs, it uses the pre-allocated unit from past inserts that did not require a merge.

The binary structure of binomial trees ensures that for heap of size n, there are at most $\log n + 1$ trees, each corresponding to a bit set in the binary representation of n. Since a merge corresponds to adding 1 in binary arithmetic, the number of merges that any single insert can cause is limited to the number of bits in n, and the work for these merges has been pre-paid by the assigned cost of 2 per insert.

Thus, over n inserts, this leads to a constant amortized cost per insert, as the total cost is linear in $O(n)$, making the amortized cost. $O(n)/n = O(1)$.