Homework 4

3] Let $G = (V, E)$ be the graph and (u, v) be the edge that is deleted inorder to form G_1.
Suppose after deletion the spanning tree T still remains connected then we wont do anything as it will now be the ~~Spanning~~ minimum spanning tree for G_1 as well.

But if T is not connected then T would be but naturally divided into T_1 and T_2.
Now we will find all the edges that will connect T_1 and T_2. After which we will find the one with minimum weight and add that edge to the minimum spanning tree (T) as that is the required minimum spanning tree.

Time Complexity:-

First we have to find: T_1 and T_2 after which we have to find all the edges.

lastly traverse through them to find the minimum one.
So, the overall time complexity would O(|V|+|E|).

4] (1) c] A-B

(2) b] B-E

(3) c] 20

6] Let graph G(V,E) be the graph with n servers and G_1 be the graph (V_1, E_1) where S be not present.
∴ So, G_1 won't be having S and all its adjacent edges.
We then either BFS or DFS to check whether G_1 is connected or not. Stating at any server.

If G_1 is a connected graph:
Compute the MST using any MST algorithm on G_1.
Add the edge between S and it's adjacent edges which has the minimum cost to MST.

Else:
Impossible to compute.

By doing so, in a way we are ensuring that S would be a leaf ~~node~~ in the MST.

7] (i) This statement is False.
Let's say there ~~is~~ an edge from S to C (S, C) which weighs 1 ~~unit~~, we have one more edge & C to t which weighs 2 and one edge from (S, t) which weighs 3.
So, there are now two shortest paths from S to t both weighing 3.

(ii) This statement is also false.
Let there be a shortest path from (S, t) which has 2 edges and each of them ~~are~~ of cost 1. Let there be one more edge connecting S and t having cost 3. Now if we increase the cost by 1 each, then the shortest ~~one~~ path will have a cost cost of 5 which

is not the multiple of k

(iii) This Statement maybe true when we assume that after decreasing the weights by k, the weights are still positive. Let's say the edges value form paths from s to t decrease by K, so in that case the shortest path would also decrease it's cost by atmost "K" as it is one of the paths.

If we consider a case where by decreasing the weights we might end up with negative weights. If we do get negative weights and let there also be a cycle in our graph which includes a path from s to t. So, the shortest path will end up going in that cycle infinite times. and so, our cost don't be decreasing by atmost k times.

(iv) This Statement is false. Let's say there are 2 paths from

s to t. One having one edge (b) which costs 5 and the other having 2 edges (a, c) each of 3 cost. S → t would

S → (b) → t this would be the shortest path. When we square the costs, edge b's cost becomes 25 whereas edges a and c became 9 each. So now, the shortest path is from edges a and c. Thus on squaring the ~~edo~~ shortest path changed.

5] We will use a max priority queue instead of a min as we want maximum bandwidth. relax step to we will also modify the insert step in the ~~min~~ priority queue because the path quality is the minimum of edges weights. All the above changes won't be affecting the correctness of Dijkstra's so exploring the vertices would be like Dijkstra's.

2] Inorder to make sure that the graph G becomes a MST, we can use the cycle property 'k' times. This means that we keep BFS till the time we don't find a cycle in the graph and once we do we remove the edge having highest cost. Till ~~we continue doing this~~ now we have removed one edge from G and we are also ensuring that it is connected thus not violating the MST property. Now, if we continue to do this 'k' times, we will end up getting another graph G' which in fact is a tree and it is a MST.
The time complexity for each iteration would be O(m+n) for BFS and subsequent check to remove the edge having high cost. Here $m \leq n+k$ so ~~the~~ O(n) is the time complexity.

1] We will use max heap to store first half of the elements and min heap to store the second half. The median would be the root of max heap and hence we can access it in O(1) time. Here we are assuming ~~to~~

n to be even and median to be n/2 when the array is sorted in increasing order.

Insert():

len(Maxheap) = 0
len(Minheap) = 0

If x < median, we insert in max-heap. Everytime we insert an element in max-heap, we increase it's length by 1. Otherwise we insert in minheap and increase it's length by 1. This would take O(logn) in worst case.

If size(maxheap) > size(minheap) + 1, then we remove one element from maxheap and decrease it's size by 1. We then add the removed element in minheap and increase it's size by 1. This would also take O(logn) in worst case.

If size(minheap) > size(maxheap), we remove one element from minheap and decrease it's size

by 1. We add the extracted element in max heap and increase it's length by 1. This would also take O(logn) time.

Find Median ():

If [len(max heap) + len (min heap)] is even, then return (sum of roots of max heap and min heap by 2) as the median.

Else if len(max heap) > len(min heap). Return the root of max heap as media

Else return the root of min heap as median