1. Show how to modify the Prim-Jarník algorithm to run in $O(n^2)$time.

Solution:

A Spanning tree is defined as a tree with N nodes and N - 1 edges.

The Prim-Jarnk algorithm may be modified to run in $O(n^2)$ time.

We will employ three arrays:

key: This array contains the weight/cost of the MST (initialized to INT MAX except for index 0 which is set to zero).

MST: This is a boolean array that specifies whether or not a node is already a member of MST (initialized to false except for the index 0 which is true).

Parent: This identifies the parent of a certain node in the MST (initialized to -1)

Steps:

1. Assuming we begin with node 0, index 0 in the key array is set to zero (because it is the first node in the MST). We locate the index/node in the key array with the lowest weight. Then we find all of its nearby edges and choose the one with the least weight.

2. At the same time, we assign this node's parent to node '0' and mark it as true (showing that it is now a member of the MST).

3. After that, we'd keep looking for the one with the lowest weight in the key array that isn't in the MST (Notice that this is where we ensure that we pick up the node with minimum weight and we do not choose an edge that might cause a cycle)

4. We repeat this approach until all nodes have joined the MST.

The time complexity of the above algorithm will be $O(n^2)$ and the space complexity is$O(n)$.

2.
Suppose you are a manager in the IT department for the government of a corrupt dictator, who has a collection of computers that need to be connected together to create a communication network for his spies. You are given a weighted graph, G, such that each vertex in G is one of these computers and each edge in G is a pair of computers that could be connected with a communication line. It is your job to decide how to connect the computers. Suppose now that the CIA has approached you and is willing to pay you various amounts of money for you to choose some of these edges to belong to this network (presumably so that they can spy on the dictator). Thus, for you, the weight of each edge in G is the amount of money, in U.S. dollars, that the CIA will pay you for using that edge in the communication network. Describe an efficient algorithm, therefore, for finding a ***maximum spanning tree*** in G, which would maximize the money you can get from the CIA for

connecting the dictator's computers in a spanning tree. What is the running time of your algorithm?

Solution:

We can use the modified version of the Prim-Jarník algorithm technique to calculate the maximum spanning tree

We will employ three arrays:

key: This array contains the weight/cost of the MST (initialized to INT MIN except for index 0 which is set to zero).

MST: This is a boolean array that specifies whether or not a node is already a member of MST (initialized to false except for the index 0 which is true).

Parent: This identifies the parent of a certain node in the MST (initialized to -1)

In addition to these three arrays, we will be using a priority queue implemented via max heap to calculate the maximum spanning tree.

Let us assume we start with the index 0. We had set key[0] = 0, parent[0] = -1 and we have added node 0 to the priority queue.

After that, we are retrieving the value from the priority queue and store it in the variable let's say x. We are iterating over all the neighbors of x and we are checking if the mset[neighbour_x] is equal to false and the weight of the edge is greater than key[neighbour_x]. If yes, then set parent[neighbor _x] = x and key[neighbour_x] = weight of the edge and add that node along with its weight in the priority queue.

Repeat the above step until the priority queue is empty.

We have a class Node that has two variables v and weight v is the neighbor of the node and weight as the name suggest is the weighted edge.

```
Node(int _v, int _w) {

        v = _v;

        weight = _w;

}
```

Below is the code snippet for the priority queue operation

```
 while (!pq.isEmpty()) {

            int u = pq.poll().getV();
```

```
        mstSet[u] = true;


        for (Node it: adj.get(u)) {
            if (mstSet[it.getV()] == false && it.getWeight() < key[it.getV()]) {
                parent[it.getV()] = u;
                key[it.getV()] = it.getWeight();
                pq.add(new Node(it.getV(), key[it.getV()]));
            }
        }
    }
```

The time complexity of the above algorithm is O(N log N) and the space complexity is O(n)

3.

Solution:

Because T is a tree, adding another edge to T will result in precisely one cycle.

The greatest weight edge of a cycle cannot occur in any least spanning tree of the graph due to the cycle characteristic of minimum spanning trees.

As a result, the technique first adds edge e' to T to generate a network with a single cycle. Then, in T, select the edge with the highest weight and delete it. The generated graph will be linked and have n-1 edges (as one edge is added and one is removed from the original minimum spanning tree). As a result, T' is the updated minimal spanning tree of the graph, with all other edge weights remaining the same as in the original graph.

The time complexity of the above algorithm is O(n + m) where n is the number of vertices and m is the number of edges and the space complexity will be O(n).

4.

Let N be a flow network with n vertices and m edges. Show how to compute an augmenting path with the largest residual capacity in O((n+m) log n) time

Solution:

The amount of flow that may be raised on an edge (u,v) on an augmenting path without breaking the capacity limitation is referred to as its residual capacity. Increasing network traffic by adding residual capabilities to edges along the augmenting path.

It is necessary to discover the lowest of residual capacities in order to locate the enhancing road with the greatest residual capacity (i.e., the path with the most flow or fattest path).

If c_m is the smallest of the residual capacities of the edges along the augmenting route p, then c_m is the maximum or biggest residual capacity that may be raised on each edge along the augmenting path.

An algorithm similar to Dijkstra's method will assist us in determining the minimal residual capacity. Dijkstra's algorithm, in general, determines the shortest path. That is, the path's edge lengths must be as short as possible.

Our modified Dijkstra's method discovers the path's edges with the lowest residual capacity (c_{min}). On each edge of the augmenting path, c_{min} will be the greatest or biggest residual capacity that may be raised.

The time complexity for the modified Dijkstra algorithm will be O(nlogn) and the space complexity O(n).

5.

Consider the previous exercise, but suppose the city of Irvine, California, changed its dog-owning ordinance so that it still allows for residents to own a maximum of three dogs per household, but now restricts each resident to own at most one dog of any given breed, such as poodle, terrier, or golden retriever. Describe an efficient algorithm for assigning puppies to residents that provides for the maximum number of puppy adoptions possible while satisfying the constraints that each resident will only adopt puppies that he or she likes, that no resident can adopt more than three puppies, and that no resident will adopt more than one dog of any given breed.

Solution:

We will make a set P of vertices of size n, where each vertice represents a pet, and a set R of residents of Irvine of size n, where each vertex represents an Irvine city resident. Create an s and t node to represent the source and destination nodes, respectively. To ensure that a pet is allocated to just one person, establish an edge with capacity 1 from s to every vertex in P, ensuring that no more than one unit flow passes from a pet vertex and therefore a pet is assigned to only one citizen.

To guarantee that every citizen adopts a pet that he or she likes, we must ensure that each citizen gets at least one pet of a specific species. Instead of building an edge (u,v) from pet u to citizen v, we will generate s_v number of intermediary nodes for each citizen v, where s_v is the number of pet species that citizen v likes.

Thus, if citizen v likes 5 different species of pets, we will create 5 intermediate nodes for citizen v and 5 vertices w1,w2,w3,w4, and w5, where each vertex from w1 to w5 denotes different species of dogs liked by v and there will be an each from w1 to w5 towards vertex v with capacity 1 unit to ensure that citizen v does not receive more than 1 dog of particular species. If pet u is in species w and liked by citizen v, there will be an edge of capacity 1 unit connecting vertex u in set P to vertex w.

So for each citizen v, if he likes a pet u of species w then we will create an edge (u,w) of capacity 1 unit and an edge (w,v) of capacity 1 unit.

The time complexity of the above algorithm will be O(NlogN) and the space complexity will be O(N).

6.

A limousine company must process pickup requests every day, for taking customers from their various homes to the local airport. Suppose this company receives pickup requests from n locations and there are n limos available, where the distance of limo i to location j is given by a number, d_{ij}. Describe an efficient algorithm for computing a dispatchment of the

n limos to the n pickup locations that minimize the total distance traveled by all the limos.

Solution:

We can use the Floyd Warshall Algorithm to solve this problem

The Floyd-Warshall Algorithm finds the shortest path between all pairs of vertices in a weighted graph. This approach is applicable to both directed and undirected weighted graphs. However, it does not work for graphs with negative cycles (where the sum of the edges in a cycle is negative).

Below are the steps to find the minimum distance for each of the pairs.

1. Make an n*n-dimensional matrix A0, where n is the number of vertices. The indexes for the row and column are I and j, respectively. The graph's vertices are I and j.

Each cell A[i][j] is filled with the distance between the ith and jth vertex. If no path exists from the ith vertex to the jth vertex, the cell is left infinite.

2. Now, using matrix A0, generate matrix A1. The elements in the first column and row remain unchanged. The remaining cells are filled as follows.

Let k be the shortest route vertex between source and destination. k is the initial vertex in this stage. If (A[i][j] > A[i][k] + A[k][j]), A[i][j] is filled with (A[i][k] + A[k][j]).
In other words, if the direct distance between the source and the destination is larger than the path through the vertex k, the cell is filled with A[i][k] + A[k][j].

k is vertex 1 in this stage. This vertex k is used to compute the distance from the source vertex to the destination vertex.
Repeat step 2 for each of the nodes in the graphs.

When we perform above step for the last node it gives the shortest path between each pair of vertices.

The time complexity of the above algorithm is $O(n^3)$ and the space complexity is $O(n^2)$**.**