Q1) A Euler tour of a strongly connected, directed graph G = (V, E) is a cycle that traverses each edge of G exactly once, although it may visit a vertex more than once.
a. Show that G has a Euler tour if and only if in-degree. (v)=out-degree(v) for each vertex v € V.
b. Describe an O(E) time algorithm to and an Euler tour of G if one exists. (Hint: Merge edge-disjoint cycles.)

a). In simple cycle we only visit all the vertices in the cycle once so in-degree=out-degree=1 for all vertices of the cycle given that they are not a part of any other cycle. But in Euler tour we know that we can visit a vertex more than once. Hence, we know that number of edges entering a vertex in Euler Tour can be more than 1.
Therefore, we take Graph G. We start from vertex v and continue on the next edge and now the next vertex we do that same thing. The only restriction in edge selection is that we can can't traverse a edge twice. As we pass a vertex, we decrease the in-degree=out degree of the vertex by 1. We will eventually reach v again. As the edge selection of vertices is random it may happen that we may not cover all vertices. Hence to cover all the vertex's we backtrack to traversed vertex connected to uncovered vertex and then continue the same thing until we reach the start vertex. When all the edges are covered, we will see that in and out degrees of all vertices of the graph will be 0. This means that in and out degree of the vertices had to be equal if they are become 0 when all the edges are traversed.

b)

```
def find_euler_tour(graph):
    def dfs(VERTEX, tour):
        tour.append(node)
        for U in graph[VERTEX]:
            if U not in tour:
               dfs(U, tour)
     tour = []
     X=random vertex from graph
     dfs(x,tour)
     return tour
```

Description:

The provided code defines a function find_euler_tour that finds a Euler tour for the given graph. A Euler tour is a traversal of a graph that visits each edge exactly once. The function works by performing a depth-first search (DFS) starting from a random vertex in the graph. The DFS appends each vertex it visits to a tour list. If vertices in adjacency list of the current vertex has not yet been visited, the DFS recursively calls itself to explore that neighbour. After all vertices have been visited, the function returns the tour list.

Example:

G=(V, E)
V= [A, B, C, D]
E= [A-B, B-C, C-D, D-A]
An Euler tour for this graph would be A-B-C-D-A. The find_euler_tour function would find this tour by starting a DFS at vertex A. Then it would traverse the adjacency list of A which has B in it. Then it would go to B. Then it would traverse the adjacency list of B which has C in it. And so and so forth. The resulting tour would be A-B-C-D-A.

Time Complexity:

The time complexity will be O(E) as our algorithm traverses all the edges once.

Correctness of Algorithm:

Loop invariant: At the beginning of each iteration of the for loop, the tour contains the path from the START vertex to the current vertex.

Initialization: The tour is initialized to an empty list. The START vertex is the first vertex visited by the DFS algorithm. Therefore, the tour contains the path from the START vertex to the current vertex at the beginning of the first iteration of the for loop.

Maintenance: The DFS algorithm recursively visits all of the vertices in the graph. At each step, the current vertex is added to the tour. Therefore, the tour always contains the path from the START vertex to the current vertex.

Termination: The DFS algorithm terminates when all of the vertices in the graph have been visited. Therefore, the tour contains the path from the START vertex to all of the vertices in the graph at the end of the for loop.

Q2) Let T be the Minimum Spanning Tree of a graph G= (V, E, w). Suppose g is connected, |E|≥|V|, and all edge weights are distinct. Denote T* the MST of G and ST(G) be the set of all spanning trees of G. A second-best MST is a spanning tree T such that w(T) =min{w(T): T ϵ ST(G)-{T*}}.
a) (4 points) Show that T* is unique but that the second-best MST T2 need not be unique.
b) (4 points) Prove that G contains an edge (u,v) ϵ T* and another edge (s,t) ∉ T* such that (T*-{(u,v)}) ∪ {(s,t)} is a second-best minimum spanning tree of G.
c) (6 points) Please use Kruskal's algorithm to design an efficient algorithm to compute G's second-best MST.

a. We will prove that a minimum spanning tree T* is unique by contradiction. Let's assume that the graph G has two minimum spanning tree T1 and T2.
If T1 and T2 are MST then both of them have V-1 edges. As we know that all edge weights are distinct then there must a exist a edge (u,v) in T1 which is not present in T2. Hence if we add (u,v) to the T2 MST . Then it will create a cycle. But we know that according to the properties of MST we can have cycle. Therefore, we will remove an edge present in the cycle.
Let's consider edge (a,b) be the edge in the cycle which has the maximum weight in the cycle. Let's remove edge (a,b) from T2. But this will give us new MST T2* which will have smaller weight than T2.
But this contradicts our assumption that T2 was an MST. Hence our original assumption must have been wrong. Therefore, we can say that T1 (T*) is only unique MST in graph G.

We will prove that second best MST must not be unique by example:

G= (V, E)
V= [P, R, S, T]
E= [(P, R,3), (P, S,3), (P, T,5), (R, S,5), (R, T,5), (S, T,3)]

In this graph we can see that MST will be {P-R, P-S, S-T} which have a weight of 9. Now let's see what will the second-best MST – {P-R, S-T, R-S} which will have a weight of 11. But there is also one more second-best MST which has same weight {P-S, S-T, R-S}. As we see there are two second best MST with same weight.

Hence through this example we can say second best MST need not be unique.

b. Let’s assume that T* is MST of graph G. Since it is a MST is contains all the minimum V-1 weights of graph G. Lets take a edge (u,v) from T* and remove it from T*. Since T* is a connected removing any edge from the T* will produce two different connected graph. There might be another edge (s,t) which will join this connected graphs. But as we know that (s,t) was not in our earlier MST T* hence the wt(u,v)< wt(s,t).
Therefore when join the two different disconnected graphs using the edge (s,t) lets name that spanning tree as T**. We know that weight of T** will be more than T* as wt(u,v)< wt(s,t).Hence this T** spanning tree has weight more than T*.
How can we say that it will have less weight than any tree spanning tree except T*. We can say that by assuming that there is a another spanning tree called T” which does not have edge (u,v) or (s,t) in it and also weight more than T* MST but less T** second best MST. We can prove this wrong by contradiction. Hence for spanning T” to have a weight less than all spanning tress which do not have the edges (u,v) or (s,t). For this to be true there should exist a edge (a,b) which has a weight less than edges (u,v) or (s,t). But if there exists such as a edge (a,b) it will contradict that T* is a MST as MST only has the minimum weight edges. Hence your assumption that T’’ is weight more than T* MST but less T** second best MST is wrong.
Hence A second-best MST is a spanning tree T such that w(T) =min{w(T): T ϵ ST(G)-{T*}} is proved.

c.
```
G=(V,E)
V=[u,v,a,b]
E=[(u,v,w1),(a,b,w2)......]
E*=sorted(E)
MST= Kruskal(V,E*)
secondMSTweight= float("inf")
currentMST = None
T=none
for e in MST.edges:
        E"=E*-e
        currMST=Kruskal(V,E"):
        if(currMST.weight<secondMSTweight):
                secondMSTweight=currST
                T=currMST


return T




        Kruskal(V,E'):
        A = none
        for i in vertex V:
        MAKE-SET(i)
        for i in E': i = (u, v) ∈ E' ordered by increasing order by weight(u, v):
        if FIND-SET(u) ≠ FIND-SET(v):
                A = A ∪ {(u, v)}
                UNION (u, v)
        return A
```

Description:
First, we sort the edge according to their weight and then find the MST of G using the Kruskal algorithm. Then we all edges in MST we will remove that edge from MST. Then calculate a spanning in the graph without including that edge. We will do this for every edge in MST. And then take spanning tree which has the minimum weight among them. That spanning tree with minimum weight will be our second-best MST.

Example:

G= (V, E)
V= [P, R, S, T]
E= [(P, R,3), (P, S,3), (S, T,3), (P, T,5), (R, S,5), (R, T,5)]

MST: {P-R, P-S, S-T} Weight = 9

As we already have sorted the edges. First, we will remove one edge from Edge set which is MST. Hence, we first remove P-R in the first iteration. Then only 5 edges remain to make a Second MST. Therefore, we get a second MST = {P-S, S-T, R-S} which has weight of 11. Then in the next iteration we will exclude edge P-S and call the Kruskal's algorithm. We get one more MST with same weight {P-R, R-S, S-T}. And then in the final iteration we exclude the edge S-T. Then we get one more MST with the same weight 11. That second best MST will be {P-R, P-T, P-S}.

Time Complexity:

```
1. G=(V,E)
2. V=[u,v,a,b]
3. E=[(u,v,w1),(a,b,w2)......]
4. E*=sorted(E)…………………………………………………….Elog(E)
5. MST= Kruskal(V,E*)…………………………………………….E
6. secondMSTweight= float("inf")
7. currentweight= float("inf")
8. T=none
9. for e in MST.edges:……………………………………………V-1
10.      E"=E*-e…………………………………………………………..1
11.      currST=Kruskal(V,E"):………………………E.V-1
12.      if(currST.weight<secondMSTweight):
               secondMSTweight=currST
               T=currST


13.return T
```

Hence the time complexity will be O (Elog E+ V-1*E+E) € O(V-1*E)
We know E=V-1 As O(V-1^2) € O(V^2) Therefore the time complexity will be O(V^2)

Correctness of Algorithm:

Loop Invariant:

At every iteration of the loop, the variable 'secondMSTweight' contains the weight of the second-best MST found so far, and the variable 'T' contains the corresponding spanning tree.

Initialization:

Initially, the variable 'secondMSTweight' is initialized to 'float("inf")', which is a value higher than the weight of any spanning tree. The variable 'T' is initialized to 'none', which represents an empty spanning tree. This satisfies the loop invariant at the beginning of the loop.

Maintenance:

Throughout the loop, the algorithm removes an edge from the MST and recalculates the MST of the modified graph. If the weight of the new MST is lower than the current second-best weight, the algorithm updates both 'secondMSTweight' and 'T' accordingly. This ensures that the loop invariant is maintained after each iteration.

Termination:

Upon termination of the loop, the algorithm has considered all possible MSTs obtained by removing an edge from the original MST. Since the loop invariant holds at every iteration, the variable 'secondMSTweight' will contain the weight of the second-best MST, and the variable 'T' will contain the corresponding spanning tree.

Therefore, by establishing a loop invariant and demonstrating its initialization, maintenance, and termination, we can conclude that the provided algorithm correctly identifies the second-best MST for any connected graph G=(V,E)

Q3) Describe a modification of Dijkstra’s algorithm that runs (asymptotically) as fast as the original algorithm and assigns a binary value usp[*u*] to every vertex *u* in *G*, so that usp[*u*]=1 if and only if there is a unique shortest path from *s* to *u*. In addition to your modification, be sure to provide arguments for both the correctness and time-bound of your algorithm and an example.

```
Dijkstra(G,w,s):
Q=heap()
SG=None
processed=[ false for i in v]
usp =[0 for i in V]
#add all the vertices in the binary heap and initialise the
distance to infinity
for v in V:
    Enqueue(Q,V,float(“inf”))
usp[v]=0
usp[s]=1
Q[s]=0
for i in range(V-1):
   u=ExtractMin(Q)
   if(processed[u]==False):
       processed[u]=True
       SG=SG+u
       for v in Adj[u]:
              if(graph[u][v]! =-1 and Q[u]+w(u,v)<Q[v]:
                     Q[v]=Q[u]+w(u,v)
                     usp[v]=1
              elseif(Q[v]==Q[u]+w(u,v)):
                     usp[v]=0
Return SG
```

Text Description:

First, we make heap Q and all the vertices and add the distance of vertices from source as infinity. Similarly, we make two arrays one for if there is unique path from source or not and one for making sure the vertices are processed or not.
Then we make the distance from source to source as 0. As the minimum distance from source to source is 0. Then we will run a loop V-1 as we need V-1 edges to join V vertices. And then run the inner loop for the adjacency list of that vertex. If there is edge from u to v then we make if the current distance from source to v is less than distance of u to source + w(u,v). If it is less then we change the minimum distance of u to v.
If there is already the calculated path is already stored in Q[v] which means Q[v]=Q[u]+w(u,v). This means there are more than 1 shortest path. So usp[v]=0. If we again find a value Q[v] which less than the current we again change to usp[v] to usp[u] as curr distance will be unique. The overall time complexity doesn’t change remains same as Dijkstra algorithm.

Time Complexity:

```
Dijkstra(G,w,s):
  1. Q=heap()
  2. SG=None………………………………………………………………………..v
  3. processed=[ false for i in v]…………………………………………….v
  4. usp =[0 for i in V]……………………………………………………….v
  5. #add all the vertices in the binary heap and initialise the distance to infinity
  6. for v in V:……………………………………………………………….v
  7.     Enqueue(Q,V,float("inf"))
  8. usp[v]=0
  9. usp[s]=1
  10. Q[s]=0
  11. for i in range(V-1):……………………………………………….v
  12.   u=ExtractMin(Q)……………………………………………logv
  13.   if(processed[u]==False):
  14.      processed[u]=True
  15.      SG=SG+u
  16.      for v in Adj[u]:………………………………………………….v
  17.           if(graph[u][v]! =-1 and Q[u]+w(u,v)<Q[v]:
  18.                Q[v]=Q[u]+w(u,v)
  19.                usp[v]=1
  20.           elseif(Q[v]==Q[u]+w(u,v)):
  21.                usp[v]=0
  22. Return SG
```

The time complexity of the above algorithm is O(V-1*log V) € O(V*logV).

Example:

G= (V, E)
V= [A, B, C]
E= [(A, B,1), (A, C, 2), (B, C, 1)]

Let's assume the starting vertex is A (s = A).

The algorithm initializes distances and predecessors:

Q = [A, B, C] with distances [0, inf, inf] and usp = [1, 0, 0]

Iteration 1:

ExtractMin(Q) returns A.
Update distances: = [B, C] with distances [1, 2]
Update usp: usp = [1, 1, 1]
SG = [A]

Iteration 2:

ExtractMin(Q) returns B.
As here we can see Q[B]+w(B, C) = 2 but same as Q[c]=2 hence there will be no change.
In distance but as now the shortest path is no longer unique, we will change usp[c] to 0
Update usp: usp = [1, 1, 0]
SG = [A, B]

Iteration 3:

ExtractMin(Q) returns C.
As all the other vertex are already processed there is nothing left to do, we will just add C to SG
SG= [A, B, C]

At last the distances=[0,1,2] and usp=[1,10]

Proof Of Correctness:
Base Case:

The base case for the Dijkstra algorithm is when there is only one vertex in the graph. In this case, the algorithm simply returns the distance from the source vertex to itself, which is zero. Since we initialize all nodes with USP[v] = 1, the USP value for the source node will remain 1, which is correct.

Induction Hypothesis:

Assume that the modified Dijkstra algorithm works correctly for all graphs with n vertex, where n is a positive integer. This means that the algorithm correctly identifies the shortest path from the source node to all other vertices in the graph, and that the USP value for each node is either 1 or 0, depending on whether the shortest path to that vertex is unique or not.

Induction Step:

Consider a graph with n+1 nodes. We can prove that the modified Dijkstra algorithm works correctly for this graph by assuming that it works correctly for all graphs with n nodes, and then using that assumption to show that it also works correctly for this graph with n+1 nodes.

The modified Dijkstra algorithm works similarly to the original Dijkstra algorithm, except that it maintains an additional array USP[v] that stores whether the shortest path to each vertex v is unique or not. Initially, all USP[v] values are set to 1. The algorithm then iteratively selects the node with the shortest distance from the source node that has not yet been visited, and adds it to the set of visited nodes. The algorithm also updates the distances of the neighbors of the selected node, if their new distances would be shorter than their current distances. Additionally, the algorithm updates the USP values of the neighbors of the selected node according to the three cases mentioned:

1. d[u] + w(u,v) < d[v]: The USP value of v is updated to the USP value of u, since the shortest path to v is now unique.
2. d[u] + w(u,v) = d[v]: The USP value of v is set to 0, since the new path to v is not unique.
3. d[u] + w(u,v) > d[v]: The USP value of v is not changed, since the path through u does not yield a shorter path.

We can prove this by induction on the distance from the source node. For the node with the shortest distance from the source node, the distance stored in the output array is zero, which is correct. Since the USP value of the source node is 1, the USP value of the node with the shortest distance from the source node will also be 1, which is correct.

For a node with distance d from the source node, assume that the distance stored in the output array is correct for all nodes with distance less than d. Assume that the USP value of the node with distance d is also correct. We can then show that the distance stored in the output array for the node with distance d is also correct and that the USP value of the node with distance d is also correct by considering the way that the algorithm updates distances and USP values.

Therefore, the modified Dijkstra algorithm works correctly for all graphs with n+1 nodes, where n is a positive integer. Since the algorithm works correctly for the base case and for all graphs with n+1 nodes, we can conclude that it works correctly for all graphs with a finite number of nodes.