Part A

Aim:

1. Greedy Method
2. Minimum Spanning Tree (Prim's algorithm and Kruskal's algorithm, for an undirected graph).

Prerequisite: Any programming language

Outcome: Algorithms and their implementation

Theory:

A minimum spanning tree (MST) or minimum weight spanning tree is a subset of the edges of a connected, edge-weighted undirected graph that connects all the vertices together, without any cycles and with the minimum possible total edge weight. That is, it is a spanning tree whose sum of edge weights is as small as possible.

Procedure:

1. Design algorithm and find best, average and worst-case complexity
2. Implement algorithm in any programming language.
3. Paste output

Practice Exercise:

S.no	Statement
1	Implement the Prim and Kruskal Algorithm.
2	Find the run time complexity of the above algorithm

Instructions:

1. Design, analysis and implement the algorithms.
2. Paste the snapshot of the output in input & output section.

Part B

Algorithm : PRIM's algorithm
Input : An Undirected Graph
Output : Minimum spanning tree set (minimum cost)

algorithm:

```
def prims(vertices,graph):
   visited=[False]*vertices
   visited[0]=True
   num_edges=0
   while(num_edges<vertices-1):
      min=99999
```

```
    x=0
    y=0
    for i in range(vertices):
        if visited[i]:
            for j in range(vertices):
                if( (not visited[j]) and graph[i][j]):
                    if min>graph[i][j]:
                        min=graph[i][j]
                        x=i
                        y=j

    print(x,"--",y,"-->",graph[x][y])
    visited[y]=True
    num_edges+=1
```

Code:

```
def prims():
    global vertices,graph,min_weight
    visited=[False]*vertices
    visited[0]=True
    num_edges=0
    while(num_edges<vertices-1):
        min=99999
        x=0
        y=0
        for i in range(vertices):
            if visited[i]:
                for j in range(vertices):
                    if( (not visited[j]) and graph[i][j]):
                        if min>graph[i][j]:
                            min=graph[i][j]
                            x=i
                            y=j

        print(x,"--",y,"-->",graph[x][y])
        min_weight+=graph[x][y]
        visited[y]=True
        num_edges+=1
```

```
min_weight=0

vertices=int(input('Number of vertices : '))
graph=[list(map(int,input().split())) for i in range(vertices)]
print ("Edges and their weights")
prims()
print('\nMinimum weight : ',min_weight)
```

Output:

```
PS E:\books and pdfs\sem4 pdfs\DAA lab\week8> python .\primss.py
Number of vertices : 5
0 2 0 6 0
2 0 3 8 5
0 3 0 0 7
6 8 0 0 9
0 5 7 9 0
Edges and their weights
0 -- 1 --> 2
1 -- 2 --> 3
1 -- 4 --> 5
0 -- 3 --> 6

Minimum weight :  16
PS E:\books and pdfs\sem4 pdfs\DAA lab\week8>
```

Algorithm : KRUSKAL's algorithm

Input : An Undirected Graph

Output : Minimum spanning tree set (minimum cost)

Algorithm:

Finding parent node:

```
def find(parent,u):
    if parent[u] == u:
        return u
    return find(parent, parent[u])
```

Union algorithm:

```
def union(parent, rank, x, y):
    x_ = find(parent, x)
    y_ = find(parent, y)
    if rank[x_] < rank[y_]:
        parent[x_] = y_
    elif rank[x_] > rank[y_]:
        parent[y_] = x_

    else:
        parent[y_] = x_
        rank[x_] += 1



def kruskals(graph,vertices):
    result = []
    i = 0
    num = 0
    graph = sorted(graph,key=lambda item: item[2])
    parent = []
    rank = []
    for j in range(vertices):
        parent.append(j)
        rank.append(0)

    while(num<vertices-1):
        u, v, w = graph[i]
        i = i + 1
        x = find(parent, u)
        y =find(parent, v)

        if x != y:
            num = num + 1
            result.append([u, v, w])
            union(parent, rank, x, y)

    minimumCost = 0
    print ("Edges and their weights")
```

```
    for u, v, weight in result:
        minimumCost += weight
        print("%d ~ %d --> %d" % (u, v, weight))
    print("Minimum Spanning Tree cost :" , minimumCost)
```

code:

```
def find(parent,u):
    if parent[u] == u:
        return u
    return find(parent, parent[u])

def union(parent, rank, x, y):
    x_ = find(parent, x)
    y_ = find(parent, y)
    if rank[x_] < rank[y_]:
        parent[x_] = y_
    elif rank[x_] > rank[y_]:
        parent[y_] = x_
    else:
        parent[y_] = x_
        rank[x_] += 1



def kruskals():
    global graph,vertices
    result = []
    i = 0
    num = 0
    graph = sorted(graph,key=lambda item: item[2])
    parent = []
    rank = []
    for j in range(vertices):
        parent.append(j)
        rank.append(0)

    while(num<vertices-1):
```

```
        u, v, w = graph[i]
        i = i + 1
        x = find(parent, u)
        y =find(parent, v)

        if x != y:
            num = num + 1
            result.append([u, v, w])
            union(parent, rank, x, y)

    minimumCost = 0
    print ("Edges and their weights")
    for u, v, weight in result:
        minimumCost += weight
        print("%d ~ %d --> %d" % (u, v, weight))
    print("Minimum Spanning Tree cost :" , minimumCost)


vertices=int(input('number of vertices : '))
edges=int(input('number of edges : '))
graph=[list(map(int,input().split())) for i in range(edges)]
kruskals()
```

Output:

```
PS E:\books and pdfs\sem4 pdfs\DAA lab\week8> python .\kurskals.py
number of vertices : 4
number of edges : 5
0 1 10
0 2 6
0 3 5
1 3 15
2 3 4
Edges and their weights
2 ~ 3 --> 4
0 ~ 3 --> 5
0 ~ 1 --> 10
Minimum Spanning Tree cost : 19
```

Run time complexity of Prim's algorithm:

The time complexity of the above algorithm is O(V^2) as it is implemented using an adjacency list.
But it can be reduced to O(E log V) if we use binary heap.

Therefore, the time complexity of prims algorithm using binary heap is O(E log V) where E is the number of edges and V is the number of vertices.

Space complexity of Prim's algorithm:

Using adjacency list, Space complexity s O(V^2)

Run time complexity of Kruskal's algorithm:

The complexity of the union and find algorithm is O(E log V) where E is the number of edges
Complexity for 1 sort is O(log E), so for E edges it would be O(E log E)

So total time complexity is: **O(E log E) +O(E log V) =O(E log V)**

Space complexity of Kruskal's algorithm:

O(|E| + |V|), since Disjoint Set Data Structure takes O(|V|) space to keep track of the roots of all the vertices and another O(|E|) space to store all edges in sorted manner.

Observation & Learning:

I have observed and learned that :
For Prim's algorithm,
i) The tree that we are making or growing always remains connected.
ii) Prim's Algorithm is faster for dense graphs.
iii) There are large number of edges in the graph like E = $O(V^2)$.

For Kruskal's algorithm:
i) The tree that we are making or growing always remains disconnected.
ii) Prim's Algorithm is faster for sparse graphs.
iii) There are less number of edges in the graph like E = O(V)

Conclusion:
I have successfully implemented Prim's and Kruskal's algorithms in Python programming language.