UIT2402 -- ADVANCED DATA STRUCTURES AND ALGORITHM ANALYSIS LAB

EX NO : 3

1.Given the following graph as an adjacency matrix or edge list, implement Kruskal's

and Prim's algorithms to find the Minimum Spanning Tree (MST)

Edge	Weight
A-B	4
A-C	1
B-C	3
B-D	2
C-D	5
C-E	6
D-E	7

Write a python program for implementing above task.

ALGORITHM:

Prim's Algorithm

- Initialize an empty Minimum Spanning Tree (MST).
- Select any arbitrary starting node as the initial vertex.

- Mark the starting node as visited.
- Initialize a list of edges connected to the visited nodes.
- Select the edge with the smallest weight that connects a visited node to an unvisited node.
- Add this edge to the MST and mark the new node as visited.
- Update the list of edges to include newly discovered edges from the visited node.
- Repeat the process of selecting the smallest edge that connects to an unvisited node.
- Continue until all vertices are included in the MST.
- The final selected edges form the Minimum Spanning Tree.

Kruskal's Algorithm

- Sort all edges in ascending order based on their weight.
- Initialize an empty Minimum Spanning Tree (MST).
- Create a disjoint-set data structure where each vertex is its own parent.
- Iterate through the sorted edges one by one.
- For each edge, check if adding it creates a cycle using the union-find method.
- If no cycle is formed, add the edge to the MST.

- Perform a union operation to merge the sets of the two vertices connected by the added edge.
- Continue adding edges until the MST contains (V - 1) edges.
- If an edge creates a cycle, discard it and move to the next edge.
- The final set of selected edges forms the Minimum Spanning Tree.

CODING:

```
class Graph:
    def __init__(self,vertices):
        self.vertices=vertices
        self.edges=[]
        self.adj_list={v:[] for v in vertices}
    def add_edge(self,u,v,weight):
        self.edges.append((weight,u,v))
        self.adj_list[u].append((weight,v))
        self.adj_list[v].append((weight,u))
    def find(self,parent,i):
        if parent[i]==i:
            return i
        return self.find(parent,parent[i])
```

```
def union(self,parent,rank,x,y):
    root_x=self.find(parent,x)
    root_y=self.find(parent,y)
    if rank[root_x]<rank[root_y]:
        parent[root_x]=root_y
    elif rank[root_x]>rank[root_y]:
        parent[root_y]=root_x
    else:
        parent[root_y]=root_x
        rank[root_x]+=1
def kruskal_mst(self):
    self.edges.sort()
    parent={v: v for v in self.vertices}
    rank={v:0 for v in self.vertices}
    mst=[]
    for weight,u,v in self.edges:
        root_u=self.find(parent,u)
        root_v=self.find(parent,v)
        if root_u!=root_v:
            mst.append((u,v,weight))
            self.union(parent,rank,root_u,root_v)
    return mst
```

```
    def prim_mst(self):
        start_vertex=self.vertices[0]
        mst=[]
        visited=set()
        min_edges=[(0,start_vertex,None)]
        while min_edges:
            min_edges.sort()
            weight,u,parent=min_edges.pop(0)
            if u not in visited:
                visited.add(u)
                if parent is not None:
                    mst.append((parent,u,weight))
                for edge_weight,v in self.adj_list[u]:
                    if v not in visited:
                        min_edges.append((edge_weight,v,u))
        return mst
graph = Graph(["A","B","C","D","E"])
graph.add_edge("A","B",4)
graph.add_edge("A","C",1)
graph.add_edge("B","C",3)
graph.add_edge("B","D",2)
graph.add_edge("C","D",5)
```

```
graph.add_edge("C","E",6)
graph.add_edge("D","E",7)
print("Kruskal's MST:",graph.kruskal_mst())
print("Prim's MST:",graph.prim_mst())
```

OUTPUT:

```
Kruskal's MST: [('A', 'C', 1), ('B', 'D', 2), ('B', 'C', 3), ('C', 'E', 6)]
Prim's MST: [('A', 'C', 1), ('C', 'B', 3), ('B', 'D', 2), ('C', 'E', 6)]
```

2. Given an input string, implement Huffman Coding to do the following:

- Generate the Huffman Tree based on character frequencies.
- Generate Huffman Codes for each character.
- Encode the input string using the Huffman Codes.

Write a python program to implement above task.

ALGORITHM:

- Calculate the frequency of each character in the input string.
- Create a Huffman node for each character and store them in a list.
- Sort the list based on character frequencies in ascending order.
- Remove the two nodes with the smallest frequencies and merge them into a new node with their combined frequency.
- Insert the new node back into the sorted list while maintaining order.
- Repeat steps 4 and 5 until only one node remains, which becomes the root of the Huffman tree.
- Traverse the tree to assign binary codes to each character, using ‘0’ for left branches and ‘1’ for right branches.
- Replace each character in the input string with its corresponding Huffman code to generate the encoded string.
- Use the Huffman tree to decode the encoded string by traversing from the root to leaf nodes based on binary sequences.
- Display the Huffman tree, character frequencies, generated codes, encoded string, and decoded output

CODING:

```
class HuffmanNode:
    def __init__(self,char,freq):
```

```
        self.char=char
        self.freq=freq
        self.left=None
        self.right=None
    def __lt__(self, other):
        return self.freq<other.freq
def calculate_frequencies(text):
    frequency={}
    for char in text:
        if char in frequency:
            frequency[char]+=1
        else:
            frequency[char]=1
    return frequency
def build_huffman_tree(text):
    frequency=calculate_frequencies(text)
    heap=[HuffmanNode(char,freq) for char,freq in
frequency.items()]
    while len(heap)>1:
        heap.sort(key=lambda node:node.freq)
        left=heap.pop(0)
        right=heap.pop(0)
```

```
        merged=HuffmanNode(None,left.freq+right.freq)
        merged.left=left
        merged.right=right
        heap.append(merged)
    return heap[0]
def
generate_huffman_codes(node,current_code,huffman_codes
):
    if node is None:
        return
    if node.char is not None:
        huffman_codes[node.char]=current_code

generate_huffman_codes(node.left,current_code+"0",huffma
n_codes)

generate_huffman_codes(node.right,current_code+"1",huff
man_codes)
def print_huffman_tree(node,level=0,prefix="Root: "):
    if node is not None:
        print("  "*level+prefix+(f"{node.char}:{node.freq}" if
node.char else f"*:{node.freq}"))
        print_huffman_tree(node.left,level+1,"L--- ")
        print_huffman_tree(node.right,level+1,"R--- ")
```

```
def huffman_encoding(text):
    root=build_huffman_tree(text)
    huffman_codes={}
    generate_huffman_codes(root,"",huffman_codes)
    encoded_text="".join(huffman_codes[char] for char in text)
    return root,huffman_codes,encoded_text
txt="abcaabdceef"
root,huffman_codes,encoded_string=huffman_encoding(txt)
print("Character Frequency")
for char,freq in sorted(calculate_frequencies(txt).items()):
    print(f"{char}: {freq}")
print("\nCharacter Huffman Code")
for char,code in sorted(huffman_codes.items()):
    print(f"{char}: {code}")
print("\nHuffman Tree:")
print_huffman_tree(root)
print("\nEncoded String:",encoded_string)
```

OUTPUT:

```
Character Frequency
a: 3
b: 2
c: 2
d: 1
e: 2
f: 1

Character Huffman Code
a: 10
b: 110
c: 111
d: 010
e: 00
f: 011

Huffman Tree:
Root: *:11
  L--- *:4
    L--- e:2
    R--- *:2
      L--- d:1
      R--- f:1
  R--- *:7
    L--- a:3
    R--- *:4
      L--- b:2
      R--- c:2

Encoded String: 10110111101011001011100000011
```