In [1]:

```
#Dijkstra's Algorithm
import heapq

def dijkstra(graph, start):
    pq = [(0, start)]
    distances = {vertex: float('infinity') for vertex in graph}
    distances[start] = 0

    while pq:
        current_distance, current_vertex = heapq.heappop(pq)

        if current_distance > distances[current_vertex]:
            continue

        for neighbor, weight in graph[current_vertex].items():
            distance = current_distance + weight

            if distance < distances[neighbor]:
                distances[neighbor] = distance
                heapq.heappush(pq, (distance, neighbor))

    return distances
graph = {
    'A': {'B': 1, 'C': 4},
    'B': {'A': 1, 'C': 2, 'D': 5},
    'C': {'A': 4, 'B': 2, 'D': 1},
    'D': {'B': 5, 'C': 1}
}

print(dijkstra(graph, 'A'))
```

```
{'A': 0, 'B': 1, 'C': 3, 'D': 4}
```

In [4]:

```python
#Huffman Codes
from heapq import heappush, heappop, heapify
from collections import defaultdict, Counter

class Node:
    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 build_huffman_tree(s):
    freq = Counter(s)
    heap = [Node(char, freq) for char, freq in freq.items()]
    heapify(heap)

    while len(heap) > 1:
        left = heappop(heap)
        right = heappop(heap)
        merged = Node(None, left.freq + right.freq)
        merged.left = left
        merged.right = right
        heappush(heap, merged)

    return heap[0]

def huffman_codes(node, prefix="", code={}):
    if node:
        if node.char is not None:
            code[node.char] = prefix
        huffman_codes(node.left, prefix + "0", code)
        huffman_codes(node.right, prefix + "1", code)
    return code

def encode(s, code):
    return ''.join(code[char] for char in s)

def decode(encoded_str, root):
    decoded_str = ""
    node = root
    for bit in encoded_str:
        node = node.left if bit == '0' else node.right
        if node.char:
            decoded_str += node.char
            node = root
    return decoded_str

s = "alice"
root = build_huffman_tree(s)
code = huffman_codes(root)
encoded_str = encode(s, code)
decoded_str = decode(encoded_str, root)

print("Original string:", s)
print("Encoded string:\n", encoded_str)
print("Decoded string:\n", decoded_str)
```

```
Original string: alice
Encoded string:
 111001100110
Decoded string:
 alice
```

In [5]:

```python
#Container Loading Problem
def container_loading(weights, capacity):
    weights.sort(reverse=True)
    containers = 0

    while weights:
        current_weight = 0
        i = 0
        while i < len(weights):
            if current_weight + weights[i] <= capacity:
                current_weight += weights[i]
                weights.pop(i)
            else:
                i += 1
        containers += 1

    return containers

weights = [4, 8, 1, 4, 2, 1]
capacity = 10
print("Number of containers needed:", container_loading(weights, capaci
```

```
Number of containers needed: 2
```

In [6]:

```python
#Minimum Spanning Tree (Kruskal's Algorithm)
class UnionFind:
    def __init__(self, n):
        self.parent = list(range(n))
        self.rank = [0] * n

    def find(self, u):
        if u != self.parent[u]:
            self.parent[u] = self.find(self.parent[u])
        return self.parent[u]

    def union(self, u, v):
        root_u = self.find(u)
        root_v = self.find(v)
        if root_u != root_v:
            if self.rank[root_u] > self.rank[root_v]:
                self.parent[root_v] = root_u
            else:
                self.parent[root_u] = root_v
                if self.rank[root_u] == self.rank[root_v]:
                    self.rank[root_v] += 1

def kruskal(n, edges):
    uf = UnionFind(n)
    mst = []
    edges.sort(key=lambda x: x[2])

    for u, v, weight in edges:
        if uf.find(u) != uf.find(v):
            uf.union(u, v)
            mst.append((u, v, weight))

    return mst

n = 4
edges = [(0, 1, 10), (0, 2, 6), (0, 3, 5), (1, 3, 15), (2, 3, 4)]
mst = kruskal(n, edges)
print("Edges in MST:", mst)
```

```
Edges in MST: [(2, 3, 4), (0, 3, 5), (0, 1, 10)]
```

In [7]:
```python
#Minimum Spanning Tree (Boruvka's Algorithm)
class UnionFind:
    def __init__(self, n):
        self.parent = list(range(n))
        self.rank = [0] * n

    def find(self, u):
        if u != self.parent[u]:
            self.parent[u] = self.find(self.parent[u])
        return self.parent[u]

    def union(self, u, v):
        root_u = self.find(u)
        root_v = self.find(v)
        if root_u != root_v:
            if self.rank[root_u] > self.rank[root_v]:
                self.parent[root_v] = root_u
            else:
                self.parent[root_u] = root_v
                if self.rank[root_u] == self.rank[root_v]:
                    self.rank[root_v] += 1

def boruvka(n, edges):
    uf = UnionFind(n)
    mst = []
    num_components = n

    while num_components > 1:
        cheapest = [-1] * n

        for u, v, weight in edges:
            set_u = uf.find(u)
            set_v = uf.find(v)
            if set_u != set_v:
                if cheapest[set_u] == -1 or cheapest[set_u][2] > weight
                    cheapest[set_u] = (u, v, weight)
                if cheapest[set_v] == -1 or cheapest[set_v][2] > weight
                    cheapest[set_v] = (u, v, weight)

        for node in range(n):
            if cheapest[node] != -1:
                u, v, weight = cheapest[node]
                if uf.find(u) != uf.find(v):
                    uf.union(u, v)
                    mst.append((u, v, weight))
                    num_components -= 1

    return mst

n = 4
edges = [(0, 1, 10), (0, 2, 6), (0, 3, 5), (1, 3, 15), (2, 3, 4)]
mst = boruvka(n, edges)
print("Edges in MST:", mst)
```

```
Edges in MST: [(0, 3, 5), (0, 1, 10), (2, 3, 4)]
```

In [ ]: