In [ ]:
```
1.coin change
```

In [8]:
```
def coinChange(coins, amount):
    coins.sort(reverse=True)
    result=0
    for coin in coins:
        count=amount//coin
        amount-=coin*count
        result+=count
    return result
coins=[1, 5, 10, 25]
amount=37
print("Minimum number of coins:",coinChange(coins,amount))
```

```
Minimum number of coins: 4
```

In [ ]:
```
2.knapsack problem
```

In [3]:
```
def fractionalKnapsack(W, arr, N):
    arr.sort(key=lambda x: x[0]/x[1], reverse=True)
    finalvalue= 0.0
    for i in range(N):
        if arr[i][1]<= W:
            W-= arr[i][1]
            finalvalue+=arr[i][0]
        else:
            finalvalue+=arr[i][0]*(W/arr[i][1])
            break
    return finalvalue
W=50
arr=[[60, 10], [100, 20], [120, 30]]
N=len(arr)
print("Maximum value in the knapsack =",fractionalKnapsack(W, arr, N))
```

```
Maximum value in the knapsack = 240.0
```

In [ ]:
```
3.job Sequencing with Deadlines
```

In [5]:
```
def job_sequencing(jobs):
    n=len(jobs)
    max_deadline=max(job[0] for job in jobs)
    jobs.sort(key=lambda x:x[1], reverse=True)
    dp=[[0]*(max_deadline+1) for_in range(n + 1)]
    for i in range(1,n+1):
        for j in range(1,max_deadline+ 1):
            if jobs[i-1][0]>j:
                dp[i][j]=dp[i-1][j]
            else:
                dp[i][j]=max(dp[i-1][j], dp[i-1][j-jobs[i-1][0]]+ jobs[i-1]
    seq=[]
    i, j=n, max_deadline
    while i>0 and j>0:
        if dp[i][j]!=dp[i-1][j]:
            seq.append(i- 1)
            j-=jobs[i-1][0]
        i-=1
    seq.reverse()
    return dp[n][max_deadline], seq
jobs=[[2, 100], [1, 19], [1, 25]]
max_profit,seq=job_sequencing(jobs)
print("Maximum profit:", max_profit)
print("Optimal sequence:", seq)
```

```
Maximum profit: 100
Optimal sequence: [0]
```

In [ ]:
```
6.cointainer loading
```

In [9]:
```
def loadContainer(items, capacity):
    sort_items=sorted(items)
    loaded_items=[]
    rema_capacity=capacity
    for item in sorted_items:
        if item<=remaining_capacity:
            loaded_items.append(item)
            rema_capacity-=item
        else:
            break
    return loaded_items
items=[50, 60, 20, 30]
capacity=100
loaded_items=loadContainer(items,capacity)
print("items loaded into  container:",loaded_items)
```

```
items loaded into  container: [20, 30, 50]
```

In [ ]:
```
minimum spanning tree
```

In [10]:
```
class DisjointSet:
    def __init__(self, n):
        self.parent = list(range(n))
    def find(self, u):
        if self.parent[u] != 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:
            self.parent[root_v] = root_u
def kruskal(n, edges):
    edges.sort(key=lambda edge: edge[2])
    ds = DisjointSet(n)
    mst = []
    for u, v, weight in edges:
        if ds.find(u) != ds.find(v):
            ds.union(u, v)
            mst.append((u, v, weight))
    return mst
edges=[(0, 1, 10), (0, 2, 6), (0, 3, 5), (1, 3, 15), (2, 3, 4)]
n = 4
mst = kruskal(n, edges)
print("Edges in MST:", mst)
```

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

In [ ]:
```
Optimal Tree Problem: Huffman Trees and Codes
```

In [11]:
```
import heapq
from collections import defaultdict

def huffman_tree(nodes):
    heap = [[weight, [symbol, ""]] for symbol, weight in nodes.items()]
    heapq.heapify(heap)
    while len(heap) > 1:
        lo = heapq.heappop(heap)
        hi = heapq.heappop(heap)
        for pair in lo[1:]:
            pair[1] = '0' + pair[1]
        for pair in hi[1:]:
            pair[1] = '1' + pair[1]
        heapq.heappush(heap, [lo[0] + hi[0]] + lo[1:] + hi[1:])
    return sorted(heapq.heappop(heap)[1:], key=lambda p: (len(p[-1]), p))

nodes = {'a': 45, 'b': 13, 'c': 12, 'd': 16, 'e': 9, 'f': 5}
huffman_tree(nodes)
```

Out[11]:
```
[['a', '0'],
 ['b', '101'],
 ['c', '100'],
 ['d', '111'],
 ['e', '1101'],
 ['f', '1100']]
```

In [ ]:
```
Single Source Shortest Paths: Dijkstra's Algorithm
```

In [13]:
```
import heapq
def dijkstra(graph, start):
    distances = {node: float('inf') for node in graph}
    distances[start] = 0
    priority_queue = [(0, start)]
    while priority_queue:
        current_distance, current_node = heapq.heappop(priority_queue)
        if current_distance > distances[current_node]:
            continue
        for neighbor, weight in graph[current_node]:
            distance = current_distance + weight
            if distance < distances[neighbor]:
                distances[neighbor] = distance
                heapq.heappush(priority_queue, (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)]
}

start_node = 'A'
distances = dijkstra(graph, start_node)
print(f"Shortest distances from {start_node}: {distances}")
```

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

In [ ]:
```
boruvka algorithm
```

In [14]:
```python
def boruvka(graph):
    parent = {}
    cheapest = {}
    for i in range(len(graph)):
        parent[i] = i
        cheapest[i] = -1
    num_trees = len(graph)
    while num_trees > 1:
        for i in range(len(graph)):
            cheapest[i] = -1
        for i in range(len(graph)):
            for j in range(len(graph[i])):
                if parent[i] != parent[j]:
                    if cheapest[parent[i]] == -1 or graph[i][cheapest[paren
                        cheapest[parent[i]] = j
        for i in range(len(graph)):
            if cheapest[i] != -1:
                if parent[i] != parent[cheapest[i]]:
                    print(f"Edge {i} - {cheapest[i]}")
                    num_trees -= 1
                    parent[parent[i]] = parent[cheapest[i]]
    return parent
graph = [
    [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]
]
boruvka(graph)
```

```
Edge 0 - 2
Edge 1 - 0
Edge 3 - 2
Edge 4 - 0
```

Out[14]:
```
{0: 2, 1: 2, 2: 2, 3: 2, 4: 2}
```

In [17]:

```
def min_key(vertices, key, mst_set):
    min_val = float('inf')
    min_index = -1
    for v in range(vertices):
        if key[v] < min_val and not mst_set[v]:
            min_val = key[v]
            min_index = v
    return min_index
def prim_mst(graph):
    vertices = len(graph)
    parent = [-1] * vertices
    key = [float('inf')] * vertices
    key[0] = 0
    mst_set = [False] * vertices
    for _ in range(vertices):
        u = min_key(vertices, key, mst_set)
        mst_set[u] = True
        print(f"Added edge: {parent[u]} - {u} Key: {key[u]}")
        for v in range(vertices):
            if graph[u][v] > 0 and not mst_set[v] and key[v] > graph[u][v]:
                parent[v] = u
                key[v] = graph[u][v]
graph = [[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]]
prim_mst(graph)
```

```
Added edge: -1 - 0 Key: 0
Added edge: 0 - 1 Key: 2
Added edge: 1 - 2 Key: 3
Added edge: 1 - 4 Key: 5
Added edge: 0 - 3 Key: 6
```

In [ ]: