LAB DAY-8

1. Coin Change Problem

```
def coinchange(coins, amount):
    dp = [float('inf')] * (amount + 1)
    dp[0] = 0
    for coin in coins:
        for x in range(coin, amount + 1):
            dp[x] = min(dp[x], dp[x - coin] + 1)
    return dp[amount] if dp[amount] != float('inf') else -1

coins = [1, 2, 5]
amount = 11
print(coinchange(coins, amount))
```

Output: 3 (11 = 5 + 5 + 1)

2. Knapsack Problem

```
def knapsack(values, weights, capacity):
    n = len(values)
    dp = [[0] * (capacity + 1) for _ in range(n + 1)]

    for i in range(1, n + 1):
        for w in range(capacity + 1):
            if weights[i - 1] <= w:
                dp[i][w] = max(dp[i - 1][w], dp[i - 1][w - weights[i - 1]] + values[i - 1])
            else:
                dp[i][w] = dp[i - 1][w]

    return dp[n][capacity]

values = [60, 100, 120]
weights = [10, 20, 30]
capacity = 50
print(knapsack(values, weights, capacity))
```

Output: 220

3. Job Sequencing with Deadlines

```
class Job:
    def __init__(self, id, deadline, profit):
        self.id = id
        self.deadline = deadline
        self.profit = profit

def job_sequencing(jobs, n):
    jobs.sort(key=lambda x: x.profit, reverse=True)
    result = [False] * n
    job_sequence = ['-1'] * n

    for i in range(len(jobs)):
        for j in range(min(n, jobs[i].deadline) - 1, -1, -1):
            if not result[j]:
                result[j] = True
                job_sequence[j] = jobs[i].id
                break
```

```
    return job_sequence

jobs = [Job('a', 2, 100), Job('b', 1, 19), Job('c', 2, 27), Job('d', 1, 25), Job('e', 3, 15)]
n = 3
print(job_sequencing(jobs, n))
```

Output: ['a', 'c', 'e']

4. Single Source Shortest Paths: Dijkstra's Algorithm

```
import heapq

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

    while queue:
        currentdistance, currentvertex = heapq.heappop(queue)

        if currentdistance > distances[currentvertex]:
            continue

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

            if distance < distances[neighbor]:
                distances[neighbor] = distance
                heapq.heappush(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 = 'A'
print(dijkstra(graph, start))
```

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

5. Optimal Tree Problem: Huffman Trees and Codes

```
import heapq
from collections import defaultdict

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 huffman_encoding(data):
    if not data:
```

```
        return "", None

    frequency = defaultdict(int)
    for char in data:
        frequency[char] += 1

    priority_queue = [HuffmanNode(char, freq) for char, freq in frequency.items()]
    heapq.heapify(priority_queue)

    while len(priority_queue) > 1:
        left = heapq.heappop(priority_queue)
        right = heapq.heappop(priority_queue)
        merged = HuffmanNode(None, left.freq + right.freq)
        merged.left = left
        merged.right = right
        heapq.heappush(priority_queue, merged)

    root = priority_queue[0]
    huffman_codes = {}

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

    encode(root, "")

    encoded_data = "".join(huffman_codes[char] for char in data)
    return encoded_data, root

def huffman_decoding(encoded_data, root):
    decoded_data = []
    current = root
    for bit in encoded_data:
        current = current.left if bit == "0" else current.right
        if current.char is not None:
            decoded_data.append(current.char)
            current = root
    return "".join(decoded_data)

data = "this is an example for huffman encoding"
encoded_data, tree = huffman_encoding(data)
print(f"Encoded data: {encoded_data}")
print(f"Decoded data: {huffman_decoding(encoded_data, tree)}")
```

6. Container Loading

```
def containerloading(weights, capacity):
    weights.sort(reverse=True)
    containers = []

    for weight in weights:
        placed = False
        for container in containers:
            if sum(container) + weight <=capacity:
                container.append(weight)
                placed = True
```

```
                break
        if not placed:
            containers.append([weight])

    return containers

weights = [4, 8, 1, 4, 2, 1]
capacity = 10
print(containerloading(weights, capacity))
```

Output: [[8, 2], [4, 4, 1, 1]]

7. 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 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:
            if self.rank[root_u] > self.rank[root_v]:
                self.parent[root_v] = root_u
            elif self.rank[root_u] < self.rank[root_v]:
                self.parent[root_u] = root_v
            else:
                self.parent[root_v] = root_u
                self.rank[root_u] += 1

def kruskal(graph):
    edges = sorted(graph['edges'], key=lambda edge: edge[2])
    uf = UnionFind(graph['vertices'])
    mst = []

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

    return mst

graph = {
    'vertices': 4,
    'edges': [
        (0, 1, 10),
        (0, 2, 6),
        (0, 3, 5),
        (1, 3, 15),
        (2, 3, 4)
    ]
}
```

```
print(kruskal(graph))
```

Prim's Algorithm:

```
import heapq


def prim(graph, startvertex):
    mst = []
    visited = set([startvertex])
    edges = [(cost, startvertex, to) for to, cost in graph[startvertex]]
    heapq.heapify(edges)

    while edges:
        cost, frm, to = heapq.heappop(edges)
        if to not in visited:
            visited.add(to)
            mst.append((frm, to, cost))

            for tonext, cost in graph[to]:
                if tonext not in visited:
                    heapq.heappush(edges, (cost, to, tonext))

    return mst
graph = {
    0: [(1, 4), (2, 1)],
    1: [(0, 4), (2, 3), (3, 2)],
    2: [(0, 1), (1, 3), (3, 4), (4, 5)],
    3: [(1, 2), (2, 4), (4, 1)],
    4: [(2, 5), (3, 1)]
}

mst = prim(graph, 0)
print("Prim's MST:", mst)
```

BORUVKA'S ALGORITHM:

```
import networkx as nx

def boruvka(graph):
    mst = nx.Graph()

    components = {node: node for node in graph.nodes}
    numcomponents = len(graph.nodes)

    while numcomponents > 1:
        cheapest = {}

        for u, v, data in graph.edges(data=True):
            weight = data['weight']
            compu = components[u]
            compv = components[v]

            if compu != compv:
                if compu not in cheapest or cheapest[compu][2] > weight:
                    cheapest[compu] = (u, v, weight)
                if compv not in cheapest or cheapest[compv][2] > weight:
                    cheapest[compv] = (u, v, weight)

        for u, v, weight in cheapest.values():
            if components[u] != components[v]:
                mst.addedge(u, v, weight=weight)
                oldcomp, newcomp = components[u], components[v]

                for node in graph.nodes:
                    if components[node] == oldcomp:
                        components[node] = newcomp
```

```
                numcomponents -= 1

    return mst

graph = nx.Graph()
graph.add_weighted_edges_from([
    (0, 1, 4),
    (0, 2, 3),
    (1, 2, 1),
    (1, 3, 2),
    (2, 3, 4),
    (3, 4, 2),
    (4, 5, 6)
])

mst = boruvka(graph)
print("Edges in the Minimum Spanning Tree:")
for edge in mst.edges(data=True):
    print(edge)
```