1. Coin Change Problem

```
def coin_change(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(coin_change(coins, amount))
```

2. Knapsack Problem

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

    for i in range(1, n + 1):
        for w in range(1, 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]

weights = [1, 3, 4, 5]
values = [1, 4, 5, 7]
capacity = 7
```

```
print(knapsack(weights, values, capacity))
```

3. Job Sequencing with Deadlines

```
def job_sequencing(jobs, n):
    jobs.sort(key=lambda x: x[2], reverse=True)
    result = [False] * n
    job_sequence = ['-1'] * n

    for i in range(len(jobs)):
        for j in range(min(n-1, jobs[i][1]-1), -1, -1):
            if result[j] is False:
                result[j] = True
                job_sequence[j] = jobs[i][0]
                break

    return job_sequence

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

4. Single Source Shortest Paths: 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}
}
start = 'A'
print(dijkstra(graph, start))
```

5. Optimal Tree Problem: Huffman Trees and Codes

```
import heapq
from collections import defaultdict, Counter
class Node:
    def __init__(self, freq, char=None, left=None, right=None):
        self.freq = freq
        self.char = char
        self.left = left
        self.right = right
    def __lt__(self, other):
```

```
        return self.freq < other.freq
def huffman_coding(s):
    frequency = Counter(s)
    heap = [Node(freq, char) for char, freq in frequency.items()]
    heapq.heapify(heap)
    while len(heap) > 1:
        node1 = heapq.heappop(heap)
        node2 = heapq.heappop(heap)
        merged = Node(node1.freq + node2.freq, left=node1, right=node2)
        heapq.heappush(heap, merged)
    root = heap[0]
    huffman_code = {}
    def generate_code(node, current_code):
        if node.char is not None:
            huffman_code[node.char] = current_code
            return
        generate_code(node.left, current_code + "0")
        generate_code(node.right, current_code + "1")

    generate_code(root, "")
    return huffman_code
s = "huffman coding algorithm"
huffman_code = huffman_coding(s)
print(huffman_code)
```

6. Container Loading Problem

```
def container_loading(weights, capacity):
    weights.sort(reverse=True)
    total_weight = 0
    for weight in weights:
```

```
        if total_weight + weight <= capacity:
            total_weight += weight
        else:
            break
    return total_weight
weights = [4, 8, 1, 4, 2, 1]
capacity = 10
print(container_loading(weights, capacity))
```

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 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
            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 = [(weight, u, v) for u, neighbors in graph.items() for v, weight in neighbors.items()]
    edges.sort()
    uf = UnionFind(len(graph))
    mst = []
    mst_weight = 0
    for weight, u, v in edges:
        if uf.find(u) != uf.find(v):
            uf.union(u, v)
            mst.append((u, v, weight))
            mst_weight += weight
    return mst, mst_weight
graph = {
    0: {1: 1, 3: 3},
    1: {0: 1, 2: 1, 3: 3},
    2: {1: 1, 3: 1},
    3: {0: 3, 1: 3, 2: 1}
}
print(kruskal(graph))
```

8. Minimum Spanning Tree: Prim's Algorithm

```
import heapq
def prim(graph, start):
    mst = []
    visited = set([start])
    edges = [(weight, start, to) for to, weight in graph[start].items()]
    heapq.heapify(edges)
    while edges:
        weight, frm, to = heapq.heappop(edges)
        if to not in visited:
            visited.add(to)
```

```
        mst.append((frm, to, weight))

        for to_next, weight in graph[to].items():
            if to_next not in visited:
                heapq.heappush(edges, (weight, to, to_next))
    return mst
graph = {
    0: {1: 1, 3: 3},
    1: {0: 1, 2: 1, 3: 3},
    2: {1: 1, 3: 1},
    3: {0: 3, 1: 3, 2: 1}
}
print(prim(graph, 0))
```

9.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
```

```
        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 boruvka(graph, n):
    uf = UnionFind(n)
    mst = []
    mst_weight = 0
    num_components = n
    while num_components > 1:
        cheapest = [-1] * n
        for u in range(n):
            for v, weight in graph[u]:
                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))
                    mst_weight += weight
                    num_components -= 1
    return mst, mst_weight
```

```
graph = {
    0: [(1, 10), (2, 6), (3, 5)],
    1: [(0, 10), (3, 15)],
    2: [(0, 6), (3, 4)],
    3: [(0, 5), (1, 15), (2, 4)]
}
n = 4
print(boruvka(graph, n))
```