1. Coin Change Problem

The Coin Change Problem can be solved using dynamic programming. Here's an implementation:

```
python
Copy code
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

# Example usage:
coins = [1, 2, 5]
amount = 11
print(coinChange(coins, amount))  # Output: 3
```

2. Knapsack Problem

Here's a solution for the 0/1 Knapsack Problem using dynamic programming:

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

    for i in range(n + 1):
        for w in range(capacity + 1):
            if i == 0 or w == 0:
                dp[i][w] = 0
            elif weights[i-1] <= w:
                dp[i][w] = max(values[i-1] + dp[i-1][w-weights[i-1]], dp[i-
1][w])
            else:
                dp[i][w] = dp[i-1][w]

    return dp[n][capacity]

# Example usage:
weights = [10, 20, 30]
values = [60, 100, 120]
capacity = 50
print(knapsack(weights, values, capacity))  # Output: 220
```

3. Job Sequencing with Deadlines

Here's a greedy approach to solve the Job Sequencing with Deadlines problem:

```
python
Copy code
class Job:
    def __init__(self, job_id, deadline, profit):
        self.job_id = job_id
        self.deadline = deadline
        self.profit = profit

def jobSequencing(jobs):
    jobs = sorted(jobs, key=lambda x: x.profit, reverse=True)
    n = max(job.deadline for job in jobs)
    result = [None] * n
    job_sequence = []

    for job in jobs:
        for j in range(min(n, job.deadline) - 1, -1, -1):
            if result[j] is None:
                result[j] = job.job_id
                job_sequence.append(job.job_id)
                break

    return job_sequence

# Example usage:
jobs = [Job(1, 2, 100), Job(2, 1, 19), Job(3, 2, 27), Job(4, 1, 25), Job(5,
3, 15)]
print(jobSequencing(jobs))  # Output: [1, 3, 5]
```

4. Single Source Shortest Paths: Dijkstra's Algorithm

Here is Dijkstra's Algorithm using a priority queue:

```
python
Copy code
import heapq

def dijkstra(graph, start):
    pq = [(0, start)]
    distances = {vertex: float('inf') 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

# Example usage:
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'))  # Output: {'A': 0, 'B': 1, 'C': 3, 'D': 4}
```

5. Optimal Tree Problem: Huffman Trees and Codes

Here is the implementation of Huffman Coding:

```
python
Copy code
from heapq import heappush, heappop, heapify
from collections import defaultdict

class Node:
    def __init__(self, freq, symbol, left=None, right=None):
        self.freq = freq
        self.symbol = symbol
        self.left = left
        self.right = right
        self.huff = ''

def printNodes(node, val=''):
    newVal = val + str(node.huff)

    if node.left:
        printNodes(node.left, newVal)
    if node.right:
        printNodes(node.right, newVal)

    if not node.left and not node.right:
        print(f"{node.symbol} -> {newVal}")

def huffmanCoding(chars, freq):
    nodes = []
    for x in range(len(chars)):
        heappush(nodes, Node(freq[x], chars[x]))

    while len(nodes) > 1:
        left = heappop(nodes)
        right = heappop(nodes)

        left.huff = 0
        right.huff = 1

        newNode = Node(left.freq + right.freq, left.symbol + right.symbol, 
left, right)

        heappush(nodes, newNode)

    printNodes(nodes[0])

# Example usage:
chars = ['a', 'b', 'c', 'd', 'e', 'f']
freq = [5, 9, 12, 13, 16, 45]
huffmanCoding(chars, freq)
```

6. Container Loading

Here's a simple approach to the Container Loading problem:

```
python
```

```
Copy code
def containerLoading(weights, capacity):
    weights.sort()
    total_weight = 0
    for weight in weights:
        if total_weight + weight <= capacity:
            total_weight += weight
        else:
            break
    return total_weight

# Example usage:
weights = [2, 3, 5, 7, 1, 4, 1]
capacity = 10
print(containerLoading(weights, capacity))  # Output: 10
```

7. Minimum Spanning Tree

We can find a Minimum Spanning Tree using Kruskal's Algorithm, Prim's Algorithm, and Boruvka's Algorithm. Here are the implementations:

7.1 Kruskal's Algorithm

```
python
Copy code
class DisjointSet:
    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, vertices):
    mst = []
    edges = sorted(graph, key=lambda x: x[2])
    disjoint_set = DisjointSet(vertices)

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

# Example usage:
graph = [
    (0, 1, 10),
    (0, 2, 6),
    (0, 3, 5),
    (1, 3, 15),
    (2, 3, 4)
]
vertices = 4
print(kruskal(graph, vertices))  # Output: [(2, 3, 4), (0, 3, 5), (0, 1,
10)]
```

7.2 Prim's Algorithm

```
python
Copy code
import heapq

def prim(graph, start):
    mst = []
    visited = set()
    min_heap = [(0, start, None)]

    while min_heap:
        weight, u, prev = heapq.heappop(min_heap)
        if u in visited:
            continue
        visited.add(u)
        if prev is not None:
            mst.append((prev, u, weight))

        for v, weight in graph[u].items():
            if v not in visited:
                heapq.heappush(min_heap, (weight, v, u))

    return mst

# Example usage:
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}
}
print(prim(graph, 0))  # Output: [(0, 3, 5), (3, 2, 4), (0, 1, 10)]
```

7.3 Boruvka's Algorithm

```
python
Copy code
class DisjointSet:
    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 boruvka(graph, vertices):
    mst = []
    disjoint_set = DisjointSet(vertices)
    cheapest = [-1] * vertices

    num_trees = vertices
    mst_weight = 0

    while num_trees > 1:
        for i in range(vertices):
            cheapest[i] = -1

        for u, v, weight in graph:
            set1 = disjoint_set.find(u)
            set2 = disjoint_set.find(v)

            if set1 != set2:
                if cheapest[set1] == -1 or cheapest[set1][2] > weight:
                    cheapest[set1] = [u, v, weight]
                if cheapest[set2] == -1 or cheapest[set2][2] > weight:
                    cheapest[set2] = [u, v, weight]

        for node in range(vertices):
            if cheapest[node] != -1:
                u, v, weight = cheapest[node]
                set1 = disjoint_set.find(u)
                set2 = disjoint_set.find(v)

                if set1 != set2:
                    mst_weight += weight
                    mst.append([u, v, weight])
                    disjoint_set.union(set1, set2)
                    num_trees -= 1

    return mst

# Example usage:
graph = [
    (0, 1, 10),
    (0, 2, 6),
    (0, 3, 5),
    (1, 3, 15),
    (2, 3, 4)
]
vertices = 4
```

```
print(boruvka(graph, vertices))  # Output: [[0, 3, 5], [2, 3, 4], [0, 1,
10]]
```