Coin Change Problem

```
def coin_change(coins, amount):
    dp = [float('inf')] * (amount + 1)
    dp[0] = 0  # Base case



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


print(coin_change([1, 2, 5], 11))  # Output: 3
print(coin_change([2], 3))         # Output: -1
```

Knapsack Problem

```
def knapsack_01(W, weights, values):
    n = len(weights)
    dp = [[0 for _ in range(W + 1)] for _ in range(n + 1)]
        for i in range(1, n + 1):
        for w in range(1, W + 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][W]


W = 50
weights = [10, 20, 30]
```

```
values = [60, 100, 120]
print(knapsack_01(W, weights, values))  # Output: 220
```

Job Sequencing with Deadlines

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


def job_sequencing_with_deadlines(jobs):
    jobs.sort(key=lambda x: x.profit, reverse=True)
    n = len(jobs)
    result = [False] * n  # To keep track of free time slots
    job_sequence = [-1] * n  # To store result (sequence of jobs)
    max_profit = 0



    for job in jobs:


        for j in range(min(n, job.deadline) - 1, -1, -1):
            if result[j] is False:
                result[j] = True
                job_sequence[j] = job.job_id
                max_profit += job.profit
                break



    job_sequence = [job_id for job_id in job_sequence if job_id != -1]
    return job_sequence, max_profit
```

```
jobs = [
    Job(1, 4, 20),
    Job(2, 1, 10),
    Job(3, 1, 40),
    Job(4, 1, 30)
]

sequence, profit = job_sequencing_with_deadlines(jobs)
print(f"Job sequence: {sequence}")  # Output: Job sequence: [3, 1]
print(f"Max profit: {profit}")      # Output: Max profit: 60
```

Single Source Shortest Paths: Dijkstra's Algorithm

```
import heapq

def dijkstra(graph, source):

    n = len(graph)



    distances = {vertex: float('infinity') for vertex in graph}
    distances[source] = 0



    priority_queue = [(0, source)]

    while priority_queue:
        current_distance, current_vertex = heapq.heappop(priority_queue)
```

```
        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(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}
}
source = 'A'
distances = dijkstra(graph, source)
print(f"Shortest distances from vertex {source}:")
for vertex, distance in distances.items():
    print(f"{vertex}: {distance}")
```

Optimal Tree Problem: Huffman Trees and Codes

```
import heapq
from collections import defaultdict

class Node:
    def __init__(self, frequency, symbol, left=None, right=None):
```

```
self.frequency = frequency
        self.symbol = symbol
        self.left = left
        self.right = right
        self.huff = ''
 def __lt__(self, other):
        return self.frequency < other.frequency
def print_codes(node, val=''):
    new_val = val + str(node.huff)
if node.left:
        print_codes(node.left, new_val)
    if node.right:
        print_codes(node.right, new_val)

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

def huffman_coding(characters, frequencies):
    nodes = []
    for x in range(len(characters)):
        heapq.heappush(nodes, Node(frequencies[x], characters[x]))

    while len(nodes) > 1:
        left = heapq.heappop(nodes)
        right = heapq.heappop(nodes)
 left.huff = 0
        right.huff =
        new_node = Node(left.frequency + right.frequency, left.symbol + right.symbol, left, right)
        heapq.heappush(nodes, new_node)
```

```
    print_codes(nodes[0])

characters = ['a', 'b', 'c', 'd', 'e']

frequencies = [45, 13, 12, 16, 9]

huffman_coding(characters, frequencies)
```