1. 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("Minimum coins required:", coin_change(coins, amount))
```

2. def knapsack(values, weights, W):

```
    n = len(values)
    dp = [[0 for x in range(W + 1)] for x in range(n + 1)]

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

values = [60, 100, 120]
weights = [10, 20, 30]
W = 50
```

```
print("Maximum value in Knapsack =", knapsack(values, weights, W))
```

3. class Job:

```
    def _init_(self, id, deadline, profit):
        self.id = id
        self.deadline = deadline
        self.profit = profit

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

    for i in range(len(jobs)):
        for j in range(min(n - 1, jobs[i].deadline - 1), -1, -1):
            if result[j] is False:
                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)]
print("Job sequence:", job_sequencing(jobs))
```

4. import heapq

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

```
    while queue:
        current_distance, current_vertex = heapq.heappop(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(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_vertex = 'A'
print("Shortest paths from", start_vertex, ":", dijkstra(graph, start_vertex))
```

5. import heapq

```
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 huffman_tree(symbols, freq):
    heap = [[weight, [symbol, '']] for symbol, weight in zip(symbols, freq)]
    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))

symbols = ['A', 'B', 'C', 'D', 'E', 'F']
freq = [5, 9, 12, 13, 16, 45]
huffman_code = huffman_tree(symbols, freq)
print("Huffman Codes:")
for p in huffman_code:
    print(f"Symbol: {p[0]}, Code: {p[1]}")
```

6.
```
def container_loading(weights, capacity):
    weights.sort()
    total_weight = 0
    for weight in weights:
        if total_weight + weight <= capacity:
            total_weight += weight
```

```
        else:
            break
    return total_weight

weights = [10, 20, 30, 40, 50]
capacity = 100
print("Maximum weight loaded:", container_loading(weights, capacity))
```

7. class Graph:

```
    def _init_(self, vertices):
        self.V = vertices
        self.graph = []

    def add_edge(self, u, v, w):
        self.graph.append([u, v, w])

    def find(self, parent, i):
        if parent[i] == i:
            return i
        return self.find(parent, parent[i])

    def union(self, parent, rank, x, y):
        root_x = self.find(parent, x)
        root_y = self.find(parent, y)

        if rank[root_x] < rank[root_y]:
            parent[root_x] = root_y
        elif rank[root_x] > rank[root_y]:
            parent[root_y] = root_x
        else:
            parent[root_y] = root_x
            rank[root_x] += 1
```

```
    def kruskal_mst(self):
        result = []
        i = 0
        e = 0
        self.graph = sorted(self.graph, key=lambda item: item[2])
        parent = []
        rank = []

        for node in range(self.V):
            parent.append(node)
            rank.append(0)

        while e < self.V –
```

8. import heapq

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

    while min_heap:
        weight, current, prev = heapq.heappop(min_heap)

        if current not in visited:
            visited.add(current)
            if prev is not None:
                mst.append((prev, current, weight))

            for neighbor, wt in graph[current].items():
                if neighbor not in visited:
```

```
            heapq.heappush(min_heap, (wt, neighbor, current))

    return mst

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_vertex = 'A'
print("Edges in MST:", prim(graph, start_vertex))
```

```
9. class Graph:
    def _init_(self, vertices):
        self.V = vertices
        self.graph = []

    def add_edge(self, u, v, w):
        self.graph.append([u, v, w])

    def find(self, parent, i):
        if parent[i] == i:
            return i
        return self.find(parent, parent[i])

    def union(self, parent, rank, x, y):
        root_x = self.find(parent, x)
        root_y = self.find(parent, y)

        if rank[root_x] < rank[root_y]:
            parent[root_x] = root_y
```