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))
```

2. **Knapsack Problem**

```
def greedy_knapsack(weights, values, capacity):
   items = [(values[i] / weights[i], weights[i], values[i]) for i in range(len(weights))]

   items.sort(reverse=True, key=lambda x: x[0])

   total_value = 0
   total_weight = 0
   selected_items = []

   for ratio, weight, value in items:
      if total_weight + weight <= capacity:
         selected_items.append((weight, value))
         total_value += value
         total_weight += weight
      else:
         remain_capacity = capacity - total_weight
         fraction = remain_capacity / weight
         total_value += value * fraction
         total_weight += weight * fraction
         selected_items.append((weight * fraction, value * fraction))
         break

   return total_value, selected_items

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

max_value, selected_items = greedy_knapsack(weights, values, capacity)
print(f"Maximum value in knapsack: {max_value}")
print("Selected items (weight, value):")
for item in selected_items:
   print(item)
```

3. Job Sequencing with Deadlines

```
def jobSequencing(jobs):
    jobs.sort(key=lambda x: x[2], reverse=True)
    n = len(jobs)
    max_deadline = max(job[1] for job in jobs)
    schedule = [-1] * (max_deadline + 1)
    total_profit = 0
    for job in jobs:
        for j in range(job[1], 0, -1):
            if schedule[j] == -1:
                schedule[j] = job[0]
                total_profit += job[2]
                break

    return total_profit, schedule


jobs = [(1, 2, 100), (2, 1, 19), (3, 2, 27), (4, 1, 25), (5, 3, 15)]
print(jobSequencing(jobs))
```

4. Single Source Shortest Paths: Dijkstra's Algorithm

```
import heapq

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

    while priority_queue:
        current_distance, current_node = heapq.heappop(priority_queue)

        if current_distance > distances[current_node]:
            continue

        for neighbor, weight in graph[current_node].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}
}
start_node = 'A'
```

```
print(dijkstra(graph, start_node))
```

5. **Optimal Tree Problem: Huffman Trees and Codes**

```
import heapq
from collections import defaultdict, Counter
class Node:
  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 huffmanCoding(chars):
  freq = Counter(chars)
  priority_queue = [Node(char, f) for char, f in freq.items()]
  heapq.heapify(priority_queue)

  while len(priority_queue) > 1:
    left = heapq.heappop(priority_queue)
    right = heapq.heappop(priority_queue)
    merged = Node(None, left.freq + right.freq)
    merged.left = left
    merged.right = right
    heapq.heappush(priority_queue, merged)
  root = priority_queue[0]
  huffman_codes = {}

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

  traverse(root, '')
  return huffman_codes

chars = "aaabbc"
huffman_codes = huffmanCoding(chars)
print(huffman_codes)
```

6. **Container Loading ,**

```
def fractionalKnapsack(values, weights, W):
  n = len(values)
  index = list(range(n))
  ratio = [v/w for v, w in zip(values, weights)]
```

```
    index.sort(key=lambda i: ratio[i], reverse=True)
    max_value = 0
    for i in index:
        if weights[i] <= W:
            W -= weights[i]
            max_value += values[i]
        else:
            max_value += values[i] * W / weights[i]
            break

    return max_value

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

7. Kruskal's Algorithms

```
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(n, edges):
    ds = DisjointSet(n)
    mst = []
    edges.sort(key=lambda x: x[2])

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

```
    return mst


n = 4
edges = [(0, 1, 10), (0, 2, 6), (0, 3, 5), (1, 3, 15), (2, 3, 4)]
print(kruskal(n, edges))
```

8. **Prims Algorithm**

```
import heapq

def prim(n, graph):
    mst = []
    visited = [False] * n
    min_heap = [(0, 0, -1)]
    total_weight = 0

    while min_heap:
        weight, u, parent = heapq.heappop(min_heap)

        if visited[u]:
            continue

        visited[u] = True
        total_weight += weight
        if parent != -1:
            mst.append((parent, u, weight))

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

    return total_weight, mst

n = 4
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(n, graph))
```

9. **Boruvka's Algorithm**

```
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(n, edges):
    ds = DisjointSet(n)
    mst = []
    num_components = n
    while num_components > 1:
        cheapest = [-1] * n

        for u, v, weight in edges:
            set_u = ds.find(u)
            set_v = ds.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]
                set_u = ds.find(u)
                set_v = ds.find(v)
                if set_u != set_v:
                    ds.union(set_u, set_v)
                    mst.append((u, v, weight))
                    num_components -= 1

    return mst

n = 4
edges = [(0, 1, 10), (0, 2, 6), (0, 3, 5), (1, 3, 15), (2, 3, 4)]
print(boruvka(n, edges))
```