Coin Change Problem

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

    for coin in coins:
        for i in range(coin, amount + 1):
            dp[i] = min(dp[i], dp[i - coin] + 1)

    return dp[amount] if dp[amount] != float('inf') else -1

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

Knapsack Problem

```
def knapsack(values, weights, capacity):
    n = len(values)
    dp = [[0 for _ in range(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] = dp[i - 1][w]
            else:
                dp[i][w] = max(dp[i - 1][w], values[i - 1] + dp[i - 1][w - weights[i - 1]])

    return dp[n][capacity]

values = [60, 100, 120]
```

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

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

Job Sequencing with Deadlines

```
def job_sequencing_with_deadlines(arr, t):
    n = len(arr)
    arr.sort(key=lambda x: x[2], reverse=True)
    result = [False] * t
    job = ['-1'] * t

    for i in range(n):
        for j in range(min(t - 1, arr[i][1] - 1), -1, -1):
            if result[j] is False:
                result[j] = True
                job[j] = arr[i][0]
                break

    return job
```

Single Source Shortest Paths: Dijkstra's Algorithm

```
import heapq

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

```
    while queue:
        current_distance, current_node = heapq.heappop(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(queue, (distance, neighbor))

    return distances
```

Optimal Tree Problem: Huffman Trees and Codes

```
from heapq import heappush, heappop, heapify
from collections import defaultdict

def huffman_tree(freq):
    heap = [[weight, [symbol, ""]] for symbol, weight in freq.items()]
    heapify(heap)
    while len(heap) > 1:
        lo = heappop(heap)
        hi = heappop(heap)
        for pair in lo[1:]:
            pair[1] = '0' + pair[1]
        for pair in hi[1:]:
            pair[1] = '1' + pair[1]
        heappush(heap, [lo[0] + hi[0]] + lo[1:] + hi[1:])
    return sorted(heappop(heap)[1:], key=lambda p: (len(p[-1]), p))
```

```
# Example Usage
freq = {'a': 16, 'b': 9, 'c': 12, 'd': 5, 'e': 13, 'f': 45}
huff_tree = huffman_tree(freq)
print("Symbol\tFrequency\tHuffman Code")
for p in huff_tree:
    print(f"{p[0]}\t{freq[p[0]]}\t\t{p[1]}")
```

Container Loading

```
        def container_loading(containers, items):
    # Your code here
    Pass

     def calculate_total_volume(items):
    # Your code here
    Pass
```

Minimum Spanning Tree

```
from collections import defaultdict

def min_spanning_tree(graph):
    parent = dict()
    rank = dict()

    def make_set(vertice):
        parent[vertice] = vertice
        rank[vertice] = 0

    def find(vertice):
```

```
        if parent[vertice] != vertice:
            parent[vertice] = find(parent[vertice])
        return parent[vertice]

    def union(vertice1, vertice2):
        root1 = find(vertice1)
        root2 = find(vertice2)
        if root1 != root2:
            if rank[root1] > rank[root2]:
                parent[root2] = root1
            else:
                parent[root1] = root2
                if rank[root1] == rank[root2]: rank[root2] += 1

    for vertice in graph['vertices']:
        make_set(vertice)

    minimum_spanning_tree = set()
    edges = list(graph['edges'])
    edges.sort()
    for edge in edges:
        weight, vertice1, vertice2 = edge
        if find(vertice1) != find(vertice2):
            union(vertice1, vertice2)
            minimum_spanning_tree.add(edge)

    return minimum_spanning_tree

# Example graph representation
graph = {
    'vertices': ['A', 'B', 'C', 'D', 'E', 'F'],
```

```
    'edges': set([
        (1, 'A', 'B'),
        (5, 'A', 'C'),
        (3, 'A', 'D'),
        (4, 'B', 'C'),
        (2, 'B', 'D'),
        (1, 'C', 'D'),
        (6, 'C', 'E'),
        (4, 'D', 'E'),
        (5, 'D', 'F'),
        (3, 'E', 'F')
    ])
}

# Finding the Minimum Spanning Tree of the example graph
mst = min_spanning_tree(graph)
print("Minimum Spanning Tree:")
for edge in mst:
    print(edge)
```

Kruskal's Algorithms,

```
            # Kruskal's Algorithm implementation
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_parent(self, parent, i):
        if parent[i] == i:
            return i
        return self.find_parent(parent, parent[i])

    def union(self, parent, rank, x, y):
```

Prims Algorithm

```
                from collections import defaultdict
from heapq import *

def prim(graph, start):
    mst = []
    visited = set([start])
    edges = [(cost, start, to) for to, cost in graph[start]]
    heapify(edges)

    while edges:
        cost, frm, to = heappop(edges)
        if to not in visited:
            visited.add(to)
            mst.append((frm, to, cost))
            for to_next, cost in graph[to]:
                if to_next not in visited:
                    heappush(edges, (cost, to, to_next))

    return mst

# Example Usage
graph = defaultdict(list)
graph[0] = [(1, 7), (2, 8)]
```

```
graph[1] = [(0, 7), (2, 5), (3, 3)]
graph[2] = [(0, 8), (1, 5), (3, 6)]
graph[3] = [(1, 3), (2, 6)]

minimum_spanning_tree = prim(graph, 0)
print(minimum_spanning_tree)
```

Boruvka's Algorithm

```
    # Boruvka's Algorithm implementation
def boruvka(graph):
  mst = []
  trees = [{node} for node in graph.nodes]

  while len(trees) > 1:
    cheapest_edge = {}
    for edge in graph.edges:
      tree1 = next((tree for tree in trees if edge[0] in tree), None)
      tree2 = next((tree for tree in trees if edge[1] in tree), None)
      if tree1 != tree2:
        cost = graph.weights[edge]
        if cost < cheapest_edge.get(tree1, (None, float('inf')))[1]:
          cheapest_edge[tree1] = (edge, cost)
        if cost < cheapest_edge.get(tree2, (None, float('inf')))[1]:
          cheapest_edge[tree2] = (edge, cost)

    for tree, (edge, cost) in cheapest_edge.items():
      mst.append(edge)
      trees.remove(tree)
      new_tree = tree.union(next(tree for tree in trees if edge[0] in tree or edge[1] in tree))
      trees = [t for t in trees if t != tree]
      trees.append(new_tree)
```

return mst