DAA DAY 9

1. There are 3n piles of coins of varying size, you and your friends will take piles of coins as follows: In each step, you will choose any 3 piles of coins (not necessarily consecutive). Of your choice, Alice will pick the pile with the maximum number of coins. You will pick the next pile with the maximum number of coins. Your friend Bob will pick the last pile. Repeat until there are no more piles of coins. Given an array of integers piles where piles[i] is the number of coins in the ith pile. Return the maximum number of coins that you can have.

```
def maxCoins(piles):
  piles.sort(reverse=True)
  max_coins = 0
  for i in range(1, len(piles) * 2 // 3, 2):
    max_coins += piles[i]
  return max_coins
piles = [2,4,1,2,7,8]
print(maxCoins(piles))
```

2. You are given a 0-indexed integer array coins, representing the values of the coins available, and an integer target. An integer x is obtainable if there exists a subsequence of coins that sums to x. Return the minimum number of coins of any value that need to be added to the array so that every integer in the range [1, target] is obtainable. A subsequence of an array is a new non-empty array that is formed from the original array by deleting some (possibly none) of the elements without disturbing the relative positions of the remaining elements.

```
def minCoins(coins, target):
  coins.sort()
  count = 0  # The number of coins we need to add
  current_sum = 0  # This represents the current maximum sum we can obtain
  for coin in coins:
    while current_sum + 1 < coin:
      count += 1
```

```
      current_sum += current_sum + 1
      if current_sum >= target:
        return count
    current_sum += coin
    if current_sum >= target:
      return count
  while current_sum < target:
    count += 1
    current_sum += current_sum + 1
  return count
coins = [1, 3]
target = 6
print(minCoins(coins, target))
```

3. You are given an integer array jobs, where jobs[i] is the amount of time it takes to complete the ith job. There are k workers that you can assign jobs to. Each job should be assigned to exactly one worker. The working time of a worker is the sum of the time it takes to complete all jobs assigned to them. Your goal is to devise an optimal assignment such that the maximum working time of any worker is minimized. Return the minimum possible maximum working time of any assignment.

```
def canAssign(jobs, k, max_time, workers):
  if not jobs:
    return True
  current_job = jobs.pop()
  for i in range(k):
    if workers[i] + current_job <= max_time:
      workers[i] += current_job
      if canAssign(jobs, k, max_time, workers):
```

```
            return True
          workers[i] -= current_job
        if workers[i] == 0:
          break
    jobs.append(current_job)
    return False
def minimumTimeRequired(jobs, k):
    left, right = max(jobs), sum(jobs)
    jobs.sort(reverse=True)
    while left < right:
        mid = (left + right) // 2
        if canAssign(jobs[:], k, mid, [0] * k):
          right = mid
        else:
          left = mid + 1
    return left
jobs = [3, 2, 3]
k = 3
print(minimumTimeRequired(jobs, k))
```

4. We have n jobs, where every job is scheduled to be done from startTime[i] to endTime[i], obtaining a profit of profit[i]. You're given the startTime, endTime and profit arrays, return the maximum profit you can take such that there are no two jobs in the subset with overlapping time range. If you choose a job that ends at time X you will be able to start another job that starts at time X.

```
from bisect import bisect_right
def jobScheduling(startTime, endTime, profit):
    jobs = sorted(zip(startTime, endTime, profit), key=lambda x: x[1])
```

```python
    dp = [0] * len(jobs)

    dp[0] = jobs[0][2]

    for i in range(1, len(jobs)):

        current_profit = jobs[i][2]

        index = bisect_right([jobs[j][1] for j in range(i)], jobs[i][0]) - 1

        if index != -1:

            current_profit += dp[index]

        dp[i] = max(dp[i-1], current_profit)

    return dp[-1]

startTime = [1, 2, 3, 4, 6]

endTime = [3, 5, 10, 6, 9]

profit = [20, 20, 100, 70, 60]

print(jobScheduling(startTime, endTime, profit))  # Output: 150
```

5. Given a graph represented by an adjacency matrix, implement Dijkstra's Algorithm to find the shortest path from a given source vertex to all other vertices in the graph. The graph is represented as an adjacency matrix where graph[i][j] denote the weight of the edge from vertex i to vertex j. If there is no edge between vertices i and j, the value is Infinity (or a very large number).

```python
import heapq

import sys

def dijkstra(graph, src):

    n = len(graph)

    dist = [sys.maxsize] * n

    dist[src] = 0

    pq = [(0, src)]  # Priority queue to get the vertex with the minimum distance

    while pq:

        current_dist, u = heapq.heappop(pq)
```

```
        if current_dist > dist[u]:
            continue
        for v in range(n):
            if graph[u][v] and dist[u] + graph[u][v] < dist[v]:
                dist[v] = dist[u] + graph[u][v]
                heapq.heappush(pq, (dist[v], v))
    return dist
graph = [
    [0, 10, 20, sys.maxsize, sys.maxsize],
    [10, 0, 30, 50, 10],
    [20, 30, 0, 20, 33],
    [sys.maxsize, 50, 20, 0, 2],
    [sys.maxsize, 10, 33, 2, 0]
]
source_vertex = 0
distances = dijkstra(graph, source_vertex)
print(distances)
```

6. Given a graph represented by an edge list, implement Dijkstra's Algorithm to find the shortest path from a given source vertex to a target vertex. The graph is represented as a list of edges where each edge is a tuple (u, v, w) representing an edge from vertex u to vertex v with weight w.

```
import heapq
import sys
from collections import defaultdict
def dijkstra(edges, n, src, target):
    graph = defaultdict(list)
    for u, v, w in edges:
```

```
        graph[u].append((v, w))
        graph[v].append((u, w))
    dist = {i: sys.maxsize for i in range(n)}
    dist[src] = 0
    pq = [(0, src)]
    while pq:
        current_dist, u = heapq.heappop(pq)
        if u == target:
            return current_dist
        for v, weight in graph[u]:
            if current_dist + weight < dist[v]:
                dist[v] = current_dist + weight
                heapq.heappush(pq, (dist[v], v))
    return -1  # Return -1 if the target is unreachable
edges = [
    (0, 1, 10), (0, 2, 5),
    (1, 2, 2), (1, 3, 1),
    (2, 1, 3), (2, 3, 9),
    (2, 4, 2), (3, 4, 4),
    (4, 0, 7), (4, 3, 6)
]
n = 5
src = 0
target = 3
print(dijkstra(edges, n, src, target))
```

7. Given a set of characters and their corresponding frequencies, construct the Huffman Tree and generate the Huffman Codes for each character.

```
import heapq

from collections import defaultdict, Counter

class Node:

  def __init__(self, char=None, freq=0):

    self.char = char

    self.freq = freq

    self.left = None

    self.right = None

  def __lt__(self, other):

    return self.freq < other.freq

def huffman_codes(frequencies):

  heap = [Node(char, freq) for char, freq in frequencies.items()]

  heapq.heapify(heap)

  while len(heap) > 1:

    left = heapq.heappop(heap)

    right = heapq.heappop(heap)

    merged = Node(freq=left.freq + right.freq)

    merged.left = left

    merged.right = right

    heapq.heappush(heap, merged)

  def generate_codes(node, prefix="", codebook={}):

    if node:

      if node.char is not None:

        codebook[node.char] = prefix
```

```
        generate_codes(node.left, prefix + "0", codebook)
        generate_codes(node.right, prefix + "1", codebook)
    return codebook
  root = heapq.heappop(heap)
  return generate_codes(root)
frequencies = Counter({'a': 5, 'b': 9, 'c': 12, 'd': 13, 'e': 16, 'f': 45})
huffman_code = huffman_codes(frequencies)
print(huffman_code)
```

8. Given a Huffman Tree and a Huffman encoded string, decode the string to get the original message.

```
class Node:
  def __init__(self, char=None, left=None, right=None):
    self.char = char
    self.left = left
    self.right = right
def decode_huffman(root, encoded_str):
  decoded_str = []
  current = root
  for bit in encoded_str:
    if bit == '0':
      current = current.left
    else:  # bit == '1'
      current = current.right
    if current.char is not None:
      decoded_str.append(current.char)
      current = root  # Reset to the root for the next character
```

```
    return ''.join(decoded_str)
root = Node()
root.left = Node()
root.right = Node('A')
root.left.left = Node('B')
root.left.right = Node('C')
encoded_str = "0110"  # Represents "BC"
decoded_message = decode_huffman(root, encoded_str)
print(decoded_message)
```

9. Given a list of item weights and the maximum capacity of a container, determine the maximum weight that can be loaded into the container using a greedy approach. The greedy approach should prioritize loading heavier items first until the container reaches its capacity.

```
def max_weight_greedy(weights, max_capacity):
    weights.sort(reverse=True)  # Sort weights in descending order
    total_weight = 0
    for weight in weights:
        if total_weight + weight <= max_capacity:
            total_weight += weight
        else:
            break
    return total_weight
weights = [10, 20, 30, 40, 50]
max_capacity = 100
print(max_weight_greedy(weights, max_capacity))
```

10. Given a list of item weights and a maximum capacity for each container, determine the minimum number of containers required to load all items using a greedy approach. The greedy approach should prioritize loading items into the current container until it is full before moving to the next container.

```
def min_containers(weights, capacity):
    weights.sort(reverse=True)  # Sort weights in descending order
    containers = []
    for weight in weights:
        placed = False
        for i in range(len(containers)):
            if containers[i] + weight <= capacity:
                containers[i] += weight
                placed = True
                break
        if not placed:
            containers.append(weight)  # Need a new container
    return len(containers)
weights = [10, 20, 30, 40, 50]
capacity = 100
print(min_containers(weights, capacity))
```

11. Given a graph represented by an edge list, implement Kruskal's Algorithm to find the Minimum Spanning Tree (MST) and its total weight.

```
import heapq
class UnionFind:
    def __init__(self, size):
        self.parent = list(range(size))
        self.rank = [0] * size
```

```python
    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):
        rootU = self.find(u)
        rootV = self.find(v)
        if rootU != rootV:
            if self.rank[rootU] > self.rank[rootV]:
                self.parent[rootV] = rootU
            elif self.rank[rootU] < self.rank[rootV]:
                self.parent[rootU] = rootV
            else:
                self.parent[rootV] = rootU
                self.rank[rootU] += 1
def kruskal(n, edges):
    uf = UnionFind(n)
    edges.sort(key=lambda x: x[2])
    mst_weight = 0
    mst_edges = []
    for u, v, weight in edges:
        if uf.find(u) != uf.find(v):
            uf.union(u, v)
            mst_edges.append((u, v, weight))
            mst_weight += weight
    return mst_edges, mst_weight
```

```
edges = [
   (0, 1, 10), (0, 2, 6), (0, 3, 5),
   (1, 3, 15), (2, 3, 4)
]
n = 4
mst_edges, mst_weight = kruskal(n, edges)
print("Edges in MST:", mst_edges)
print("Total weight of MST:", mst_weight)
```