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. Example 1: Input: piles = [2,4,1,2,7,8] Output: 9 Explanation: Choose the triplet (2, 7, 8), Alice Pick the pile with 8 coins, you the pile with 7 coins and Bob the last one. Choose the triplet (1, 2, 4), Alice Pick the pile with 4 coins, you the pile with 2 coins and Bob the last one. The maximum number of coins which you can have is: 7 + 2 = 9. On the other hand if we choose this arrangement (1, 2, 8), (2, 4, 7) you only get 2 + 4 = 6 coins which is not optimal. Example 2: Input: piles = [2,4,5] Output: 4

def maxCoins(piles):

piles.sort(reverse=True)

max\_coins = sum(piles[i] for i in range(1, 2 \* len(piles) // 3, 2))

return max\_coins

piles1 = [2, 4, 1, 2, 7, 8]

print("Output for Example 1:", maxCoins(piles1))

piles2 = [2, 4, 5]

print("Output for Example 2:", maxCoins(piles2))

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. Example 1: Input: coins = [1,4,10], target = 19 Output: 2 Explanation: We need to add coins 2 and 8. The resulting array will be [1, 2, 4, 8, 10]. It can be shown that all integers from 1 to 19 are obtainable from the resulting array, and that 2 is the minimum number of coins that need to be added to the array. Example 2: Input: coins = [1, 4, 10, 5, 7, 19], target = 19 Output: 1 Explanation: We only need to add the coin 2. The resulting array will be [1,2, 4, 5, 7, 10, 19]. It can be shown that all integers from 1 to 19 are obtainable from the resulting array, and that 1 is the minimum number of coins that need to be added to the array

def minCoinsNeeded(coins, target):

coins.sort()

count, current\_sum = 0, 0

for coin in coins:

while current\_sum + 1 < coin and current\_sum < target:

current\_sum += current\_sum + 1

count += 1

current\_sum += coin

if current\_sum >= target:

return count

while current\_sum < target:

current\_sum += current\_sum + 1

count += 1

return count

coins1 = [1, 4, 10]

target1 = 19

print("Output for Example 1:", minCoinsNeeded(coins1, target1))

coins2 = [1, 4, 10, 5, 7, 19]

target2 = 19

print("Output for Example 2:", minCoinsNeeded(coins2, target2))

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. Example 1: Input: jobs = [3,2,3], k = 3 Output: 3 Explanation: By assigning each person one job, the maximum time is 3. Example 2: Input: jobs = [1,2,4,7,8], k = 2 Output: 11 Explanation: Assign the jobs the following way: Worker 1: 1, 2, 8 (working time = 1 + 2 + 8 = 11) Worker 2: 4, 7 (working time = 4 + 7 = 11) The maximum working time is 11.

def canFinish(jobs, k, max\_time):

workers = [0] \* k

jobs.sort(reverse=True)

def backtrack(i):

if i == len(jobs):

return True

for j in range(k):

if workers[j] + jobs[i] <= max\_time:

workers[j] += jobs[i]

if backtrack(i + 1):

return True

workers[j] -= jobs[i]

if workers[j] == 0:

break

return False

return backtrack(0)

def minTimeRequired(jobs, k):

left, right = max(jobs), sum(jobs)

while left < right:

mid = (left + right) // 2

if canFinish(jobs, k, mid):

right = mid

else:

left = mid + 1

return left

jobs1 = [3, 2, 3]

k1 = 3

print("Output for Example 1:", minTimeRequired(jobs1, k1))

jobs2 = [1, 2, 4, 7, 8]

k2 = 2

print("Output for Example 2:", minTimeRequired(jobs2, k2))

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. Example 1: Input: startTime = [1,2,3,3], endTime = [3,4,5,6], profit = [50,10,40,70] Output: 120 Explanation: The subset chosen is the first and fourth job. Time range [1-3]+[3-6] , we get profit of 120 = 50 + 70. Example 2: Input: startTime = [1,2,3,4,6], endTime = [3,5,10,6,9], profit = [20,20,100,70,60] Output: 150 Explanation: The subset chosen is the first, fourth and fifth job. Profit obtained 150 = 20 + 70 + 60.

from bisect import bisect\_right

def jobScheduling(startTime, endTime, profit):

jobs = sorted(zip(startTime, endTime, profit), key=lambda x: x[1])

n = len(jobs)

dp = [0] \* (n + 1)

for i in range(1, n + 1):

s, e, p = jobs[i - 1]

prev\_index = bisect\_right([jobs[j][1] for j in range(i - 1)], s)

dp[i] = max(dp[i - 1], dp[prev\_index] + p)

return dp[n]

startTime1 = [1, 2, 3, 3]

endTime1 = [3, 4, 5, 6]

profit1 = [50, 10, 40, 70]

print("Output for Example 1:", jobScheduling(startTime1, endTime1, profit1))

startTime2 = [1, 2, 3, 4, 6]

endTime2 = [3, 5, 10, 6, 9]

profit2 = [20, 20, 100, 70, 60]

print("Output for Example 2:", jobScheduling(startTime2, endTime2, profit2))

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). Test Case 1: Input: n = 5 graph = [[0, 10, 3, Infinity, Infinity], [Infinity, 0, 1, 2, Infinity], [Infinity, 4, 0, 8, 2], [Infinity, Infinity, Infinity, 0, 7], [Infinity, Infinity, Infinity, 9, 0]] source = 0 Output: [0, 7, 3, 9, 5] Test Case 2: Input: n = 4 graph = [[0, 5, Infinity, 10], [Infinity, 0, 3, Infinity], [Infinity, Infinity, 0, 1], [Infinity, Infinity, Infinity, 0] ] source = 0 Output: [0, 5, 8, 9]

import sys

import heapq

def dijkstra\_matrix(n, graph, source):

distances = [float('inf')] \* n

distances[source] = 0

min\_heap = [(0, source)]

while min\_heap:

current\_dist, u = heapq.heappop(min\_heap)

if current\_dist > distances[u]:

continue

for v in range(n):

if graph[u][v] != float('inf') and graph[u][v] != 0:

distance = current\_dist + graph[u][v]

if distance < distances[v]:

distances[v] = distance

heapq.heappush(min\_heap, (distance, v))

return distances

n1 = 5

graph1 = [[0, 10, 3, float('inf'), float('inf')], [float('inf'), 0, 1, 2, float('inf')], [float('inf'), 4, 0, 8, 2], [float('inf'), float('inf'), float('inf'), 0, 7], [float('inf'), float('inf'), float('inf'), 9, 0]]

source1 = 0

print("Output for Test Case 1:", dijkstra\_matrix(n1, graph1, source1))

n2 = 4

graph2 = [[0, 5, float('inf'), 10], [float('inf'), 0, 3, float('inf')], [float('inf'), float('inf'), 0, 1], [float('inf'), float('inf'), float('inf'), 0]]

source2 = 0

print("Output for Test Case 2:", dijkstra\_matrix(n2, graph2, source2))

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. Test Case 1: Input: n = 6 edges = [(0, 1, 7), (0, 2, 9), (0, 5, 14), (1, 2, 10), (1, 3, 15), (2, 3, 11), (2, 5, 2), (3, 4, 6), (4, 5, 9) ] source = 0 target = 4 Output: 20 Test Case 2: Input: n = 5 edges = [(0, 1, 10), (0, 4, 3), (1, 2, 2), (1, 4, 4), (2, 3, 9), (3, 2, 7), (4, 1, 1), (4, 2, 8), (4, 3, 2)] source = 0 target = 3 Output: 8

import heapq

from collections import defaultdict

def dijkstra\_edges(n, edges, source, target):

graph = defaultdict(list)

for u, v, w in edges:

graph[u].append((v, w))

distances = [float('inf')] \* n

distances[source] = 0

min\_heap = [(0, source)]

while min\_heap:

current\_dist, u = heapq.heappop(min\_heap)

if u == target:

return current\_dist

if current\_dist > distances[u]:

continue

for v, weight in graph[u]:

distance = current\_dist + weight

if distance < distances[v]:

distances[v] = distance

heapq.heappush(min\_heap, (distance, v))

return distances[target] if distances[target] != float('inf') else -1

n1 = 6

edges1 = [(0, 1, 7), (0, 2, 9), (0, 5, 14), (1, 2, 10), (1, 3, 15), (2, 3, 11), (2, 5, 2), (3, 4, 6), (4, 5, 9)]

source1, target1 = 0, 4

print("Output for Test Case 1:", dijkstra\_edges(n1, edges1, source1, target1))

n2 = 5

edges2 = [(0, 1, 10), (0, 4, 3), (1, 2, 2), (1, 4, 4), (2, 3, 9), (3, 2, 7), (4, 1, 1), (4, 2, 8), (4, 3, 2)]

source2, target2 = 0, 3

print("Output for Test Case 2:", dijkstra\_edges(n2, edges2, source2, target2))

7. Given a set of characters and their corresponding frequencies, construct the Huffman Tree and generate the Huffman Codes for each character. Test Case 1: Input: n = 4 characters = ['a', 'b', 'c', 'd'] frequencies = [5, 9, 12, 13] Output: [('a', '110'), ('b', '10'), ('c', '0'), ('d', '111')] Test Case 2: Input: n = 6 characters = ['f', 'e', 'd', 'c', 'b', 'a'] frequencies = [5, 9, 12, 13, 16, 45] Output: [ ('a', '0'), ('b', '101'), ('c', '100'), ('d', '111'), ('e', '1101'), ('f', '1100')]

import heapq

from collections import defaultdict

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 huffman\_encoding(characters, frequencies):

heap = [Node(characters[i], frequencies[i]) for i in range(len(characters))]

heapq.heapify(heap)

while len(heap) > 1:

left = heapq.heappop(heap)

right = heapq.heappop(heap)

merged = Node(None, left.freq + right.freq)

merged.left = left

merged.right = right

heapq.heappush(heap, merged)

root = heap[0]

codes = {}

def generate\_codes(node, current\_code):

if node:

if node.char:

codes[node.char] = current\_code

generate\_codes(node.left, current\_code + "0")

generate\_codes(node.right, current\_code + "1")

generate\_codes(root, "")

return sorted(codes.items())

characters1 = ['a', 'b', 'c', 'd']

frequencies1 = [5, 9, 12, 13]

print("Output for Test Case 1:", huffman\_encoding(characters1, frequencies1))

characters2 = ['f', 'e', 'd', 'c', 'b', 'a']

frequencies2 = [5, 9, 12, 13, 16, 45]

print("Output for Test Case 2:", huffman\_encoding(characters2, frequencies2))

8. Given a Huffman Tree and a Huffman encoded string, decode the string to get the original message. Test Case 1: Input: n = 4 characters = ['a', 'b', 'c', 'd'] frequencies = [5, 9, 12, 13] encoded\_string = '1101100111110' Output: "abacd" Test Case 2: Input: n = 6 characters = ['f', 'e', 'd', 'c', 'b', 'a'] frequencies = [5, 9, 12, 13, 16, 45] encoded\_string = '110011011100101111001011' Output: "fcbade"

def huffman\_decoding(characters, frequencies, encoded\_string):

root = huffman\_encoding\_tree(characters, frequencies)

decoded\_string, node = "", root

for bit in encoded\_string:

node = node.left if bit == '0' else node.right

if node.char:

decoded\_string += node.char

node = root

return decoded\_string

def huffman\_encoding\_tree(characters, frequencies):

heap = [Node(characters[i], frequencies[i]) for i in range(len(characters))]

heapq.heapify(heap)

while len(heap) > 1:

left = heapq.heappop(heap)

right = heapq.heappop(heap)

merged = Node(None, left.freq + right.freq)

merged.left = left

merged.right = right

heapq.heappush(heap, merged)

return heap[0]

characters1 = ['a', 'b', 'c', 'd']

frequencies1 = [5, 9, 12, 13]

encoded\_string1 = '1101100111110'

print("Output for Test Case 1:", huffman\_decoding(characters1, frequencies1, encoded\_string1))

characters2 = ['f', 'e', 'd', 'c', 'b', 'a']

frequencies2 = [5, 9, 12, 13, 16, 45]

encoded\_string2 = '110011011100101111001011'

print("Output for Test Case 2:", huffman\_decoding(characters2, frequencies2, encoded\_string2))

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. Test Case 1: Input: n = 5 weights = [10, 20, 30, 40, 50] max\_capacity = 60 Output: 50 Test Case 2: Input: n = 6 weights = [5, 10, 15, 20, 25, 30] max\_capacity = 50 Output: 50

def max\_weight\_greedy(weights, max\_capacity):

weights.sort(reverse=True)

current\_weight = 0

for weight in weights:

if current\_weight + weight <= max\_capacity:

current\_weight += weight

return current\_weight

weights1 = [10, 20, 30, 40, 50]

max\_capacity1 = 60

print("Output for Test Case 1:", max\_weight\_greedy(weights1, max\_capacity1))

weights2 = [5, 10, 15, 20, 25, 30]

max\_capacity2 = 50

print("Output for Test Case 2:", max\_weight\_greedy(weights2, max\_capacity2))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. Test Case 1: Input: n = 7 weights = [5, 10, 15, 20, 25, 30, 35] max\_capacity = 50 Output: 4 Test Case 2: Input: n = 8 weights = [10, 20, 30, 40, 50, 60, 70, 80] max\_capacity = 100 Output: 6

def min\_containers\_greedy(weights, max\_capacity):

weights.sort(reverse=True)

container\_count = 0

current\_capacity = max\_capacity

for weight in weights:

if weight > current\_capacity:

container\_count += 1

current\_capacity = max\_capacity

current\_capacity -= weight

if current\_capacity < max\_capacity:

container\_count += 1

return container\_count

weights1 = [5, 10, 15, 20, 25, 30, 35]

max\_capacity1 = 50

print("Output for Test Case 1:", min\_containers\_greedy(weights1, max\_capacity1))

weights2 = [10, 20, 30, 40, 50, 60, 70, 80]

max\_capacity2 = 100

print("Output for Test Case 2:", min\_containers\_greedy(weights2, max\_capacity2))

11. Given a graph represented by an edge list, implement Kruskal's Algorithm to find the Minimum Spanning Tree (MST) and its total weight. Test Case 1: Input: n = 4 m = 5 edges = [ (0, 1, 10), (0, 2, 6), (0, 3, 5), (1, 3, 15), (2, 3, 4) ] Output: Edges in MST: [(2, 3, 4), (0, 3, 5), (0, 1, 10)] Total weight of MST: 19 Test Case 2: Input: n = 5 m = 7 edges = [ (0, 1, 2), (0, 3, 6), (1, 2, 3), (1, 3, 8), (1, 4, 5), (2, 4, 7), (3, 4, 9) ] Output: Edges in MST: [(0, 1, 2), (1, 2, 3), (1, 4, 5), (0, 3, 6)] Total weight of MST: 16

class UnionFind:

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\_mst(n, edges):

edges.sort(key=lambda x: x[2])

uf = UnionFind(n)

mst = []

total\_weight = 0

for u, v, weight in edges:

if uf.find(u) != uf.find(v):

uf.union(u, v)

mst.append((u, v, weight))

total\_weight += weight

return mst, total\_weight

n1, m1, edges1 = 4, 5, [(0, 1, 10), (0, 2, 6), (0, 3, 5), (1, 3, 15), (2, 3, 4)]

mst1, weight1 = kruskal\_mst(n1, edges1)

print("Output for Test Case 1:")

print("Edges in MST:", mst1)

print("Total weight of MST:", weight1)

n2, m2, edges2 = 5, 7, [(0, 1, 2), (0, 3, 6), (1, 2, 3), (1, 3, 8), (1, 4, 5), (2, 4, 7), (3, 4, 9)]

mst2, weight2 = kruskal\_mst(n2, edges2)

print("Output for Test Case 2:")

print("Edges in MST:", mst2)

print("Total weight of MST:", weight2)

12. Given a graph with weights and a potential Minimum Spanning Tree (MST), verify if the given MST is unique. If it is not unique, provide another possible MST. Test Case 1: Input: n = 4 m = 5 edges = [ (0, 1, 10), (0, 2, 6), (0, 3, 5), (1, 3, 15), (2, 3, 4) ] given\_mst = [(2, 3, 4), (0, 3, 5), (0, 1, 10)] Output: Is the given MST unique? True Test Case 2: Input: n = 5 m = 6 edges = [ (0, 1, 1), (0, 2, 1), (1, 3, 2), (2, 3, 2), (3, 4, 3), (4, 2, 3) ] given\_mst = [(0, 1, 1), (0, 2, 1), (1, 3, 2), (3, 4, 3)] Output: Is the given MST unique? False Another possible MST: [(0, 1, 1), (0, 2, 1), (2, 3, 2), (3, 4, 3)] Total weight of MST:7

def is\_unique\_mst(n, edges, given\_mst):

given\_mst\_weight = sum(weight for \_, \_, weight in given\_mst)

mst, mst\_weight = kruskal\_mst(n, edges)

if given\_mst\_weight != mst\_weight:

return False, mst

return True, mst

n1, m1, edges1, given\_mst1 = 4, 5, [(0, 1, 10), (0, 2, 6), (0, 3, 5), (1, 3, 15), (2, 3, 4)], [(2, 3, 4), (0, 3, 5), (0, 1, 10)]

is\_unique1, alt\_mst1 = is\_unique\_mst(n1, edges1, given\_mst1)

print("Output for Test Case 1:")

print("Is the given MST unique?", is\_unique1)

n2, m2, edges2, given\_mst2 = 5, 6, [(0, 1, 1), (0, 2, 1), (1, 3, 2), (2, 3, 2), (3, 4, 3), (4, 2, 3)], [(0, 1, 1), (0, 2, 1), (1, 3, 2), (3, 4, 3)]

is\_unique2, alt\_mst2 = is\_unique\_mst(n2, edges2, given\_mst2)

print("Output for Test Case 2:")

print("Is the given MST unique?", is\_unique2)

if not is\_unique2:

print("Another possible MST:", alt\_mst2)