1. **T**here 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.

Input: piles = [2,4,1,2,7,8]

Output: 9

def maxCoins(piles):

piles.sort()

total\_coins = 0

n = len(piles) // 3

for i in range(n):

total\_coins += piles[-(i + 1)]

return total\_coins

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

print(maxCoins(piles)) # Output: 9

1. 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.

Input: coins = [1,4,10], target = 19

Output: 2

def minCoins(coins, target):

coins.sort()

current\_sum = 0

added\_coins = 0

for coin in coins:

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

added\_coins += 1

current\_sum += (current\_sum + 1)

current\_sum += coin

if current\_sum >= target:

break

while current\_sum < target:

added\_coins += 1

current\_sum += (current\_sum + 1)

return added\_coins

coins = [1, 4, 10]

target = 19

print(minCoins(coins, target))

1. 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.

jobs = [3, 2, 3]

k = 3

left = max(jobs)

right = sum(jobs)

while left < right:

mid = (left + right) // 2

worker\_count = 0

current\_time = 0

possible = True

for job in jobs:

if current\_time + job > mid:

worker\_count += 1

current\_time = job

if worker\_count >= k:

possible = False

break

else:

current\_time += job

if possible:

right = mid

else:

left = mid + 1

print(left)

1. 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.

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.

startTime = [1, 2, 3, 3]

endTime = [3, 4, 5, 6]

profit = [50, 10, 40, 70]

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

n = len(jobs)

dp = [0] \* n

for i in range(n):

dp[i] = jobs[i][2]

for j in range(i):

if jobs[j][1] <= jobs[i][0]:

dp[i] = max(dp[i], dp[j] + jobs[i][2])

max\_profit = max(dp)

print(max\_profit)

1. 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).

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]

import sys

n = 5

Infinity = float('inf')

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

distances = [Infinity] \* n

distances[source] = 0

visited = [False] \* n

for \_ in range(n):

min\_distance = Infinity

min\_index = -1

for v in range(n):

if not visited[v] and distances[v] < min\_distance:

min\_distance = distances[v]

min\_index = v

visited[min\_index] = True

for v in range(n):

if (graph[min\_index][v] != Infinity and not visited[v] and

distances[min\_index] + graph[min\_index][v] < distances[v]):

distances[v] = distances[min\_index] + graph[min\_index][v]

print(distances)

1. 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.

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

import sys

import heapq

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

graph = {i: [] for i in range(n)}

for u, v, w in edges:

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

graph[v].append((u, w)) # If the graph is undirected

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

distances[source] = 0

priority\_queue = [(0, source)] # (distance, vertex)

while priority\_queue:

current\_distance, current\_vertex = heapq.heappop(priority\_queue)

if current\_vertex == target:

break

if current\_distance > distances[current\_vertex]:

continue

for neighbor, weight in graph[current\_vertex]:

distance = current\_distance + weight

if distance < distances[neighbor]:

distances[neighbor] = distance

heapq.heappush(priority\_queue, (distance, neighbor))

result = distances[target]

print(result if result != float('inf') else "No path")

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

Input: n = 4 characters = ['a', 'b', 'c', 'd'] frequencies = [5, 9, 12, 13]

Output: [('a', '110'), ('b', '10'), ('c', '0'), ('d', '111')]

import heapq

n = 4

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

frequencies = [5, 9, 12, 13]

heap = [[freq, [char, ""]] for char, freq in zip(characters, frequencies)]

heapq.heapify(heap)

while len(heap) > 1:

low1 = heapq.heappop(heap)

low2 = heapq.heappop(heap)

for pair in low1[1:]:

pair[1] = '0' + pair[1]

for pair in low2[1:]:

pair[1] = '1' + pair[1]

heapq.heappush(heap, [low1[0] + low2[0]] + low1[1:] + low2[1:])

huffman\_codes = sorted(heap[0][1:], key=lambda p: (len(p[-1]), p))

print(huffman\_codes)

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

Input: n = 4 characters = ['a', 'b', 'c', 'd'] frequencies = [5, 9, 12, 13]

encoded\_string = '1101100111110'

Output: "abacd"

import heapq

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

frequencies = [5, 9, 12, 13]

encoded\_string = '1101100111110'

heap = [[freq, char] for char, freq in zip(characters, frequencies)]

heapq.heapify(heap)

while len(heap) > 1:

low1 = heapq.heappop(heap)

low2 = heapq.heappop(heap)

for pair in low1[1:]:

pair = (pair[0], '0' + pair[1])

for pair in low2[1:]:

pair = (pair[0], '1' + pair[1])

heapq.heappush(heap, [low1[0] + low2[0]] + low1[1:] + low2[1:])

huffman\_codes = {code: char for char, code in heap[0][1:]}

decoded\_message = ""

current\_code = ""

for bit in encoded\_string:

current\_code += bit

if current\_code in huffman\_codes:

decoded\_message += huffman\_codes[current\_code]

current\_code = ""

print(decoded\_message)

1. 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.

Input: n = 5 weights = [10, 20, 30, 40, 50] max\_capacity = 60

Output: 50

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

max\_capacity = 60

weights.sort(reverse=True)

total\_weight = 0

for weight in weights:

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

total\_weight += weight

print(total\_weight)

1. 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.

Input: n = 7 weights = [5, 10, 15, 20, 25, 30, 35] max\_capacity = 50

Output: 4

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

max\_capacity = 50

weights.sort(reverse=True)

container\_count = 0

current\_capacity = 0

for weight in weights:

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

current\_capacity += weight

else:

container\_count += 1

current\_capacity = weight

if current\_capacity > 0:

container\_count += 1

print(container\_count)

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

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

return True

return False

def kruskal(n, edges):

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

uf = UnionFind(n)

mst\_edges = []

total\_weight = 0

for u, v, weight in edges:

if uf.union(u, v):

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

total\_weight += weight

if len(mst\_edges) == n - 1:

break

return mst\_edges, total\_weight

n,m = 4,5

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

mst\_edges, total\_weight = kruskal(n, edges)

print("Edges in MST:", mst\_edges)

print("Total weight of MST:", total\_weight)

1. 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.

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

n, m = 4, 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)]

mst\_weight = sum(weight for u, v, weight in given\_mst)

mst\_edges\_set = set((min(u, v), max(u, v)) for u, v, weight in given\_mst)

alternative\_mst = []

found\_alternative = False

for u, v, weight in edges:

if (min(u, v), max(u, v)) not in mst\_edges\_set:

total\_weight = weight + sum(w for \_, \_, w in alternative\_mst)

if total\_weight < mst\_weight:

alternative\_mst.append((u, v, weight))

elif total\_weight == mst\_weight:

alternative\_mst.append((u, v, weight))

found\_alternative = True

break

is\_unique = len(alternative\_mst) == 0

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

if not is\_unique:

print("Another possible MST:", alternative\_mst)