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

**Output: 9**

**Program:**

def maxCoins(piles):

piles.sort(reverse=True)

coins = 0

n = len(piles) // 3

for i in range(n):

coins += piles[2 \* i + 1]

return coins

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

print(maxCoins(piles))

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.**

**Example : Input: coins = [1,4,10]**

**target = 19**

**Output: 2**

**Program:**

def min\_coins\_to\_add(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(min\_coins\_to\_add(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 : Input: jobs = [3,2,3]**

**k = 3**

**Output: 3**

**Program:**

def can\_assign(jobs, k, max\_time, workers):

if not jobs:

return True

job = jobs.pop()

for i in range(k):

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

workers[i] += job

if can\_assign(jobs, k, max\_time, workers):

return True

workers[i] -= job

if workers[i] == 0:

break

jobs.append(job)

return False

def minimum\_time\_required(jobs, k):

jobs.sort(reverse=True)

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

while left < right:

mid = (left + right) // 2

if can\_assign(jobs[:], k, mid, [0] \* k):

right = mid

else:

left = mid + 1

return left

jobs = [3, 2, 3]

k = 3

print(minimum\_time\_required(jobs, k))

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.**

**Example : Input: startTime = [1,2,3,3]**

**endTime = [3,4,5,6]**

**profit = [50,10,40,70]**

**Output: 120**

**Program:**

from bisect import bisect\_right

def job\_scheduling(startTime, endTime, profit):

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

dp = [0] \* len(jobs)

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

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

current\_profit = jobs[i][2]

j = bisect\_right([jobs[k][1] for k in range(i)], jobs[i][0]) - 1

if j != -1:

current\_profit += dp[j]

dp[i] = max(dp[i-1], current\_profit)

return dp[-1]

startTime = [1, 2, 3, 3]

endTime = [3, 4, 5, 6]

profit = [50, 10, 40, 70]

print(job\_scheduling(startTime, endTime, 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).**

**Test Case : 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]**

**Program:**

import heapq

def dijkstra(graph, source):

n = len(graph)

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

distances[source] = 0

priority\_queue = [(0, source)]

while priority\_queue:

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

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

continue

for neighbor in range(n):

weight = graph[current\_vertex][neighbor]

if weight != float('inf'):

distance = current\_distance + weight

if distance < distances[neighbor]:

distances[neighbor] = distance

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

return distances

n = 5

graph = [

[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]

]

source = 0

print(dijkstra(graph, source))

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.**

**Test Case : 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**

**Program:**

import heapq

from collections import defaultdict

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

adj\_list = defaultdict(list)

for u, v, w in edges:

adj\_list[u].append((v, w))

adj\_list[v].append((u, w))

dist = {i: float('inf') for i in range(n)}

dist[source] = 0

pq = [(0, source)]

while pq:

current\_dist, u = heapq.heappop(pq)

if u == target:

return current\_dist

if current\_dist > dist[u]:

continue

for v, weight in adj\_list[u]:

distance = current\_dist + weight

if distance < dist[v]:

dist[v] = distance

heapq.heappush(pq, (distance, v))

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

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

print(dijkstra(n, edges, source, target))

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

**Test Case : Input: n = 4 characters = ['a', 'b', 'c', 'd']**

**frequencies = [5, 9, 12, 13]**

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

**Program:**

import heapq

def huffman\_codes(characters, frequencies):

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

heapq.heapify(heap)

while len(heap) > 1:

lo = heapq.heappop(heap)

hi = heapq.heappop(heap)

for pair in lo[1:]:

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

for pair in hi[1:]:

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

heapq.heappush(heap, [lo[0] + hi[0]] + lo[1:] + hi[1:])

return sorted(heapq.heappop(heap)[1:], key=lambda p: characters.index(p[0]))

n = 4

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

frequencies = [5, 9, 12, 13]

output = huffman\_codes(characters, frequencies)

print(output)

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

**Test Case : Input: n = 4 characters = ['a', 'b', 'c', 'd']**

**frequencies = [5, 9, 12, 13]**

**encoded\_string = '1101100111110'**

**Output: "abacd"**

**Program:**

import heapq

class Node:

def \_\_init\_\_(self, freq, char=None):

self.freq = freq

self.char = char

self.left = None

self.right = None

def \_\_lt\_\_(self, other):

return self.freq < other.freq

def build\_huffman\_tree(characters, frequencies):

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

heapq.heapify(heap)

while len(heap) > 1:

left = heapq.heappop(heap)

right = heapq.heappop(heap)

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

merged.left = left

merged.right = right

heapq.heappush(heap, merged)

return heap[0] # Root of the Huffman Tree

def decode\_huffman\_tree(root, encoded\_string):

decoded\_string = []

current\_node = root

for bit in encoded\_string:

if bit == '0':

current\_node = current\_node.left

else:

current\_node = current\_node.right

if current\_node.left is None and current\_node.right is None:

decoded\_string.append(current\_node.char)

current\_node = root

return ''.join(decoded\_string)

n = 4

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

frequencies = [5, 9, 12, 13]

encoded\_string = '1101100111110'

root = build\_huffman\_tree(characters, frequencies)

decoded\_message = decode\_huffman\_tree(root, encoded\_string)

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.**

**Test Case : Input: n = 5**

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

**max\_capacity = 60**

**Output: 50**

**Program:**

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

sorted\_weights = sorted(weights, reverse=True)

total\_weight = 0

for weight in sorted\_weights:

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

total\_weight += weight

else:

break

return total\_weight

n = 5

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

max\_capacity = 60

print(max\_weight\_loaded(weights, max\_capacity))

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.**

**Test Case : Input: n = 7**

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

**max\_capacity = 50**

**Output: 4**

**Program:**

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

sorted\_weights = sorted(weights, reverse=True)

num\_containers = 0

current\_capacity = 0

for weight in sorted\_weights:

if current\_capacity + weight > max\_capacity:

num\_containers += 1

current\_capacity = weight

else:

current\_capacity += weight

if current\_capacity > 0:

num\_containers += 1

return num\_containers

n = 7

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

max\_capacity = 50

print(min\_containers(weights, max\_capacity))

1. **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 : 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**

**Program:**

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]) # Path compression

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):

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

uf = UnionFind(n)

mst\_edges = []

total\_weight = 0

for u, v, weight in edges:

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

uf.union(u, v)

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

total\_weight += weight

return mst\_edges, total\_weight

n = 4

m = 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.**

**Test Case: 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**

**Program:**

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(n, edges):

edges = sorted(edges, 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

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

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

mst, weight = kruskal(n, edges)

if weight != given\_weight:

return True, None

edges = sorted(edges, key=lambda x: x[2])

uf = UnionFind(n)

another\_mst = []

for u, v, weight in edges:

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

uf.union(u, v)

if (u, v, weight) not in given\_mst:

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

if len(another\_mst) == len(given\_mst) and sum(w for \_, \_, w in another\_mst) == given\_weight:

return False, another\_mst

return True, None

n = 4

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)]

unique, another\_mst = is\_unique\_mst(n, edges, given\_mst)

print("Is the given MST unique?", unique)

if not unique:

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