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.

Example 1:

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

Output: 9

CODE:

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))

OUTPUT:

9

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

CODE:

def min\_coins\_to\_reach\_target(coins, target):

coins.sort()

current\_max = 0

count\_added = 0

i = 0

while current\_max < target:

if i < len(coins) and coins[i] <= current\_max + 1:

current\_max += coins[i]

i += 1

else:

current\_max += (current\_max + 1)

count\_added += 1

return count\_added

coins = [1, 4, 10]

target = 19

result = min\_coins\_to\_reach\_target(coins, target)

print(f"Output: {result}")

OUTPUT:

Minimum number of coins to be added: 2

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

CODE:

def canAssign(jobs, k, limit):

# Array to store the workload of each worker

workloads = [0] \* k

def backtrack(i):

if i == len(jobs):

return True

for j in range(k):

if workloads[j] + jobs[i] <= limit:

workloads[j] += jobs[i]

if backtrack(i + 1):

return True

workloads[j] -= jobs[i]

if workloads[j] == 0:

break

return False

return backtrack(0)

def minMaxWorkingTime(jobs, k):

jobs.sort(reverse=True)

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

while left < right:

mid = (left + right) // 2

if canAssign(jobs, k, mid):

right = mid # Try for a smaller possible time

else:

left = mid + 1 # Increase the working time limit

return left

jobs = [3, 2, 3]

k = 3

result = minMaxWorkingTime(jobs, k)

print(f"The minimum possible maximum working time is: {result}")

OUTPUT:

The minimum possible maximum working time is: 3

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.

CODE:

from bisect import bisect\_right

def jobScheduling(startTime, endTime, profit):

# Combine start time, end time, and profit into a single list of jobs

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

dp = [0] \* len(jobs)

start = [job[0] for job in jobs]

end = [job[1] for job in jobs]

profit = [job[2] for job in jobs]

dp[0] = profit[0]

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

last\_non\_conflicting = bisect\_right(end, start[i]) - 1

include\_profit = profit[i]

if last\_non\_conflicting != -1:

include\_profit += dp[last\_non\_conflicting]

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

return dp[-1]

startTime = [1, 2, 3, 3]

endTime = [3, 4, 5, 6]

profit = [50, 10, 40, 70]

result = jobScheduling(startTime, endTime, profit)

print(f"The maximum profit is: {result}")

OUTPUT:

The maximum profit is: 120

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]

CODE:

import heapq

def dijkstra(graph, source):

n = len(graph)

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

dist[source] = 0

pq = [(0, source)]

while pq:

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

if current\_dist > dist[u]:

continue

for v in range(n):

if graph[u][v] != float('inf'): # If there is an edge from u to v

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

if distance < dist[v]:

dist[v] = distance

heapq.heappush(pq, (distance, v)) # Push the new distance to the queue

return dist

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

shortest\_paths = dijkstra(graph, source)

print(f"The shortest distances from vertex {source} are: {shortest\_paths}")

OUTPUT:

The shortest distances from vertex 0 are: [0, 7, 3, 9, 5]

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

CODE:

import heapq

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

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

for u, v, w in edges:

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

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

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

dist[source] = 0

pq = [(0, source)] # Priority queue stores (distance, vertex)

while pq:

current\_dist, u = heapq.heappop(pq) # Get the vertex with smallest distance

if u == target:

return current\_dist # Return distance when the target is reached

if current\_dist > dist[u]:

continue # If we have already found a shorter path, skip

for v, weight in graph[u]:

distance = current\_dist + weight

if distance < dist[v]:

dist[v] = distance

heapq.heappush(pq, (distance, v)) # Push the new distance to the queue

return float('inf')

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

shortest\_path = dijkstra(n, edges, source, target)

print(f"The shortest path from vertex {source} to vertex {target} is: {shortest\_path}")

OUTPUT:

The shortest path from vertex 0 to vertex 4 is: 20

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')]

CODE:

import heapq

class HuffmanNode:

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 build\_huffman\_tree(characters, frequencies):

heap = []

for i in range(len(characters)):

node = HuffmanNode(characters[i], frequencies[i])

heapq.heappush(heap, node)

while len(heap) > 1:

left = heapq.heappop(heap)

right = heapq.heappop(heap)

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

merged.left = left

merged.right = right

heapq.heappush(heap, merged)

return heap[0]

def generate\_huffman\_codes(root):

codes = {}

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

if not node:

return

if node.char is not None:

codes[node.char] = current\_code

\_generate\_codes(node.left, current\_code + '0')

\_generate\_codes(node.right, current\_code + '1')

\_generate\_codes(root, "")

return codes

def huffman\_encoding(characters, frequencies):

# Step 1: Build Huffman Tree

huffman\_tree\_root = build\_huffman\_tree(characters, frequencies)

huffman\_codes = generate\_huffman\_codes(huffman\_tree\_root)

return huffman\_codes

n = 4

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

frequencies = [5, 9, 12, 13]

huffman\_codes = huffman\_encoding(characters, frequencies)

output = [(char, code) for char, code in huffman\_codes.items()]

print(output)

OUTPUT:

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

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"

CODE:

import heapq

class HuffmanNode:

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 build\_huffman\_tree(characters, frequencies):

heap = []

for i in range(len(characters)):

node = HuffmanNode(characters[i], frequencies[i])

heapq.heappush(heap, node)

while len(heap) > 1:

left = heapq.heappop(heap)

right = heapq.heappop(heap)

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

merged.left = left

merged.right = right

heapq.heappush(heap, merged)

return heap[0] # Return the root of the Huffman Tree

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

decoded\_message = []

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.char is not None:

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

current\_node = root # Go back to the root for the next set of bits

return ''.join(decoded\_message)

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

huffman\_tree\_root = build\_huffman\_tree(characters, frequencies)

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

return decoded\_message

n = 4

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

frequencies = [5, 9, 12, 13]

encoded\_string = '1101100111110'

decoded\_message = huffman\_decoding(characters, frequencies, encoded\_string)

print(decoded\_message)

OUTPUT:

"abacd"

9) 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

CODE:

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

weights.sort(reverse=True)

total\_weight = 0

for weight in 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

result = max\_weight(weights, max\_capacity)

print(result)

OUTPUT:

50

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

CODE:

def min\_containers(weights, max\_capacity)

weights.sort()

container\_count = 0

current\_capacity = 0

for weight in weights:

if current\_capacity + weight > max\_capacity:

container\_count += 1

current\_capacity = weight

else:

current\_capacity += weight

if current\_capacity > 0:

container\_count += 1

return container\_count

n = 7

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

max\_capacity = 50

result = min\_containers(weights, max\_capacity

print(result)

OUTPUT:

4

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

CODE:

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:

# Union by rank

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]) # Sort by the third element (weight)

uf = UnionFind(n)

mst\_edges = []

total\_weight = 0

for u, v, weight in edges:

if uf.union(u, v): # If u and v are not already connected

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)

OUTPUT:

Edges in MST: [(2, 3, 4), (0, 3, 5), (0, 1, 10)]

Total weight of MST: 19

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

CODE:

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

return True

return False

def verify\_mst(n, edges, given\_mst):

uf = UnionFind(n)

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

for u, v, weight in given\_mst:

uf.union(u, v)

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

mst\_edges = []

total\_weight = 0

edge\_count = 0

for u, v, weight in edges:

if uf.union(u, v):

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

total\_weight += weight

edge\_count += 1

if edge\_count == n - 1:

break

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

return False, []

alternative\_mst = []

uf2 = UnionFind(n)

for u, v, weight in edges:

if uf2.union(u, v):

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

if alternative\_mst != given\_mst:

return False, alternative\_mst

return True, []

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)]

is\_unique, alternative\_mst = verify\_mst(n, edges, given\_mst)

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

if not is\_unique:

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

OUTPUT:

Is the given MST unique? True