**TOPIC 5 : GREEDY**

* 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):

# Sorting is not needed as we assume piles are given in the required order

total\_coins = 0

i = 1 # Index to pick the second largest in each triplet

while i < len(piles):

total\_coins += piles[i]

i += 3 # Move to the next triplet

return total\_coins

# Example usage:

print(maxCoins([2, 4, 5])) # Output: 4

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 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

.

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.

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 minimumMaxWorkingTime(jobs, k):

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

        # Use backtracking to check if we can assign jobs within max\_time

        workers = [0] \* k

        return dfs(jobs, workers, 0, max\_time)

    def dfs(jobs, workers, idx, max\_time):

        if idx >= len(jobs):

            return True

        job\_time = jobs[idx]

        for i in range(len(workers)):

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

                workers[i] += job\_time

                if dfs(jobs, workers, idx + 1, max\_time):

                    return True

                workers[i] -= job\_time

                # Early pruning: if workers[i] is 0, further workers[i+1:] will also be 0

                if workers[i] == 0:

                    break

        return False

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

    while left < right:

        mid = (left + right) // 2

        if can\_assign(jobs, k, mid):

            right = mid

        else:

            left = mid + 1

    return left

# Example usage:

print(minimumMaxWorkingTime([3, 2, 3], 3))  # Output: 3

print(minimumMaxWorkingTime([1, 2, 4, 7, 8], 2))  # Output: 11

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 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.

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 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]

def dijkstra(graph, n, source):

# Step 1: Initialize distance array with infinity and source with 0

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

dist[source] = 0

# Step 2: Initialize visited array

visited = [False] \* n

# Step 3: Process vertices

for \_ in range(n):

# Find vertex with the minimum distance not yet visited

min\_dist = float('inf')

min\_vertex = -1

for v in range(n):

if not visited[v] and dist[v] < min\_dist:

min\_dist = dist[v]

min\_vertex = v

# Mark the selected vertex as visited

visited[min\_vertex] = True

# Update distances for neighbors of the selected vertex

for v in range(n):

if graph[min\_vertex][v] != float('inf'): # There is an edge from min\_vertex to v

new\_dist = dist[min\_vertex] + graph[min\_vertex][v]

if new\_dist < dist[v]:

dist[v] = new\_dist

return dist

# Test Case 1

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]

]

n1 = 5

source1 = 0

print(dijkstra(graph1, n1, source1)) # Output: [0, 7, 3, 9, 5]

# Test Case 2

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]

]

n2 = 4

source2 = 0

print(dijkstra(graph2, n2, source2)) # Output: [0, 5, 8, 9]

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 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

1. 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')]

1. 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"

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 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(weights, max\_capacity):

# Step 1: Sort weights in descending order

weights.sort(reverse=True)

# Step 2: Initialize variables

current\_load = 0

# Step 3: Iterate and load items until capacity is exceeded

for weight in weights:

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

current\_load += weight

else:

break

# Step 4: Return the maximum weight loaded

return current\_load

# Test Case 1

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

max\_capacity1 = 60

print(max\_weight(weights1, max\_capacity1)) # Output: 50

# Test Case 2

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

max\_capacity2 = 50

print(max\_weight(weights2, max\_capacity2)) # Output: 50

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 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

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 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

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 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