DESIGN AND ANALYSIS OF ALGORITHM

NAME : MALREDDY.P

REGISTER NO: 192372015

DAY -09 PROGRAMS

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

```python
def max_coins(piles):
    piles.sort()
    total_coins = 0
    n = len(piles) // 3
    for i in range(n):
        total_coins += piles[-2 - 2 * i]
    return total_coins

print(max_coins([2, 4, 1, 2, 7, 8]))
print(max_coins([2, 4, 5]))
```

```
9
4

=== Code Execution Successful ===
```

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.

```python
def min_coins_to_add(coins, target):
    coins.sort()
    current_value = 1
    new_coins_needed = 0

    for coin in coins:
        while coin > current_value:
            new_coins_needed += 1
            current_value += current_value

        current_value += coin

        if current_value > target:
            break

    while current_value <= target:
        new_coins_needed += 1
        current_value += current_value

    return new_coins_needed

print(min_coins_to_add([1, 4, 10], 19))
print(min_coins_to_add([1, 4, 10, 5, 7, 19], 19))
```

```
2
1

=== Code Execution Successful ===
```

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.

```python
def min_max_working_time(jobs, k):
    def can_distribute(jobs, k, max_time):
        current_worker_time = 0
        workers_needed = 1
        for job in jobs:
            if current_worker_time + job > max_time:
                workers_needed += 1
                current_worker_time = job
                if workers_needed > k:
                    return False
            else:
                current_worker_time += job
        return True

    left = max(jobs)
    right = sum(jobs)
    while left < right:
        mid = (left + right) // 2
        if can_distribute(jobs, k, mid):
            right = mid
        else:
            left = mid + 1

    return left

print(min_max_working_time([3, 2, 3], 3))
print(min_max_working_time([1, 2, 4, 7, 8], 2))
```

```
3
14

=== Code Execution Successful ===
```

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

```python
def min_max_working_time(jobs, k):
    def can_distribute(jobs, k, max_time):
        current_worker_time = 0
        workers_needed = 1
        for job in jobs:
            if current_worker_time + job > max_time:
                workers_needed += 1
                current_worker_time = job
                if workers_needed > k:
                    return False
            else:
                current_worker_time += job
        return True

    left = max(jobs)
    right = sum(jobs)
    while left < right:
        mid = (left + right) // 2
        if can_distribute(jobs, k, mid):
            right = mid
        else:
            left = mid + 1

    return left

print(min_max_working_time([3, 2, 3], 3))
print(min_max_working_time([1, 2, 4, 7, 8], 2))
```

```
3
14

=== Code Execution Successful ===
```

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]

```python
import heapq

def dijkstra(n, graph, source):
    distances = [float('inf')] * n
    distances[source] = 0
    priority_queue = [(0, source)]

    while priority_queue:
        current_distance, u = heapq.heappop(priority_queue)
        if current_distance > distances[u]:
            continue
        for v in range(n):
            if graph[u][v] != float('inf'):
                distance = current_distance + graph[u][v]
                if distance < distances[v]:
                    distances[v] = distance
                    heapq.heappush(priority_queue, (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(dijkstra(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(dijkstra(n2, graph2, source2))
```

```
[0, 7, 3, 9, 5]
[0, 5, 8, 9]

=== Code Execution Successful ===
```

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.

```python
import heapq

def dijkstra(n, edges, source, target):
    adj_list = {i: [] for i in range(n)}
    for u, v, w in edges:
        adj_list[u].append((v, w))
        adj_list[v].append((u, w))

    distances = [float('inf')] * n
    distances[source] = 0
    priority_queue = [(0, source)]

    while priority_queue:
        current_distance, u = heapq.heappop(priority_queue)

        if u == target:
            return current_distance

        if current_distance > distances[u]:
            continue
        for v, weight in adj_list[u]:
            distance = current_distance + weight
            if distance < distances[v]:
                distances[v] = distance
                heapq.heappush(priority_queue, (distance, v))
    return -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 = 0
target1 = 4
print(dijkstra(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 = 0
target2 = 3
print(dijkstra(n2, edges2, source2, target2))
```

```
20
5

=== Code Execution Successful ===
```

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')].

```python
import heapq
from collections import defaultdict, Counter
class TreeNode:
    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 build_huffman_tree(characters, frequencies):
    priority_queue = [TreeNode(char, freq) for char, freq in zip(characters, frequencies)]
    heapq.heapify(priority_queue)
    while len(priority_queue) > 1:
        left = heapq.heappop(priority_queue)
        right = heapq.heappop(priority_queue)
        merged = TreeNode(None, left.freq + right.freq)
        merged.left = left
        merged.right = right
        heapq.heappush(priority_queue, merged)

    return priority_queue[0]

def generate_huffman_codes(root):
    codes = {}

    def traverse(node, code):
        if node is not None:
            if node.char is not None:
                codes[node.char] = code
            traverse(node.left, code + '0')
            traverse(node.right, code + '1')

    traverse(root, '')
    return codes

def huffman_coding(characters, frequencies):
    root = build_huffman_tree(characters, frequencies)
    codes = generate_huffman_codes(root)
    return sorted(codes.items())

characters1 = ['a', 'b', 'c', 'd']
frequencies1 = [5, 9, 12, 13]
print(huffman_coding(characters1, frequencies1))
characters2 = ['f', 'e', 'd', 'c', 'b', 'a']
frequencies2 = [5, 9, 12, 13, 16, 45]
print(huffman_coding(characters2, frequencies2))
```

```
[('a', '00'), ('b', '01'), ('c', '10'), ('d', '11')]
[('a', '0'), ('b', '111'), ('c', '101'), ('d', '100'), ('e', '1101'), ('f', '1100')]

=== Code Execution Successful ===
```

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".

```python
import heapq
class TreeNode:
    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 build_huffman_tree(characters, frequencies):
    priority_queue = [TreeNode(char, freq) for char, freq in zip(characters, frequencies)]
    heapq.heapify(priority_queue)
    while len(priority_queue) > 1:
        left = heapq.heappop(priority_queue)
        right = heapq.heappop(priority_queue)
        merged = TreeNode(None, left.freq + right.freq)
        merged.left = left
        merged.right = right
        heapq.heappush(priority_queue, merged)
    return priority_queue[0]

def decode_huffman_tree(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.left is None and current_node.right is None:
            decoded_message.append(current_node.char)
            current_node = root
    return ''.join(decoded_message)

def huffman_decode(characters, frequencies, encoded_string):
    root = build_huffman_tree(characters, frequencies)
    return decode_huffman_tree(root, encoded_string)

characters1 = ['a', 'b', 'c', 'd']
frequencies1 = [5, 9, 12, 13]
encoded_string1 = '1101100111110'
print(huffman_decode(characters1, frequencies1, encoded_string1))

characters2 = ['f', 'e', 'd', 'c', 'b', 'a']
frequencies2 = [5, 9, 12, 13, 16, 45]
encoded_string2 = '110011011100101111001011'
print(huffman_decode(characters2, frequencies2, encoded_string2))
```

```
dbcbdd
fefcbaac

=== Code Execution Successful ===
```

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.

```python
def max_weight_loaded(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

weights1 = [10, 20, 30, 40, 50]
max_capacity1 = 60
print(max_weight_loaded(weights1, max_capacity1))

weights2 = [5, 10, 15, 20, 25, 30]
max_capacity2 = 50
print(max_weight_loaded(weights2, max_capacity2))
```

```
50
30

=== Code Execution Successful ===
```

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.

```python
def min_containers(weights, max_capacity):
    weights.sort(reverse=True)
    num_containers = 0
    current_capacity = 0

    for weight in weights:
        if current_capacity + weight <= max_capacity:
            current_capacity += weight
        else:
            num_containers += 1
            current_capacity = weight
    if current_capacity > 0:
        num_containers += 1

    return num_containers

weights1 = [5, 10, 15, 20, 25, 30, 35]
max_capacity1 = 50
print(min_containers(weights1, max_capacity1))

weights2 = [10, 20, 30, 40, 50, 60, 70, 80]
max_capacity2 = 100
print(min_containers(weights2, max_capacity2))
```

```
4
5

=== Code Execution Successful ===
```

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.

ain.py Share Run

```python
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.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

n1 = 4
edges1 = [(0, 1, 10), (0, 2, 6), (0, 3, 5), (1, 3, 15), (2, 3, 4)]
mst_edges1, total_weight1 = kruskal(n1, edges1)
print("Edges in MST:", mst_edges1)
print("Total weight of MST:", total_weight1)

n2 = 5
edges2 = [(0, 1, 2), (0, 3, 6), (1, 2, 3), (1, 3, 8), (1, 4, 5), (2, 4, 7), (3, 4, 9)]
mst_edges2, total_weight2 = kruskal(n2, edges2)
print("Edges in MST:", mst_edges2)
print("Total weight of MST:", total_weight2)
```

Output

```
Edges in MST: [(2, 3, 4), (0, 3, 5), (0, 1, 10)]
Total weight of MST: 19
Edges in MST: [(0, 1, 2), (1, 2, 3), (1, 4, 5), (0, 3, 6)]
Total weight of MST: 16

=== Code Execution Successful ===
```

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:12.

```python
from heapq import heappop, heappush
from collections import defaultdict

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 is_valid_mst(n, edges, given_mst):
    total_weight_given_mst = sum(weight for _, _, weight in given_mst)
    if len(given_mst) != n - 1:
        return False, None, None

    graph = defaultdict(list)
    for u, v, weight in edges:
        graph[u].append((weight, v))
        graph[v].append((weight, u))

    uf = UnionFind(n)
    for u, v, weight in given_mst:
        if uf.find(u) == uf.find(v):
            return False, None, None
        uf.union(u, v)

    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
            if len(mst_edges) == n - 1:
                break

    if set(mst_edges) != set(given_mst):
        return True, mst_edges, total_weight
    else:
        return False, mst_edges, total_weight

n1 = 4
edges1 = [(0, 1, 10), (0, 2, 6), (0, 3, 5), (1, 3, 15), (2, 3, 4)]
given_mst1 = [(2, 3, 4), (0, 3, 5), (0, 1, 10)]
is_unique1, other_mst1, weight1 = is_valid_mst(n1, edges1, given_mst1)
print("Is the given MST unique?", is_unique1)

n2 = 5
edges2 = [(0, 1, 1), (0, 2, 1), (1, 3, 2), (2, 3, 2), (3, 4, 3), (4, 2, 3)]
given_mst2 = [(0, 1, 1), (0, 2, 1), (1, 3, 2), (3, 4, 3)]
```

```
Is the given MST unique? False
Is the given MST unique? False
Another possible MST: [(0, 1, 1), (0, 2, 1), (1, 3, 2), (3, 4, 3)]
Total weight of MST: 7

=== Code Execution Successful ===
```