# DAY-9 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.

main.py

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

    return total_coins

# Example usage
piles1 = [2, 4, 1, 2, 7, 8]
print(max_coins(piles1))  # Output: 9

piles2 = [2, 4, 5]
print(max_coins(piles2))  # Output: 4
```

Output

```
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.

main.py

```python
def min_coins_needed(coins, target):
    coins.sort()
    current_sum = 0
    coins_added = 0
    for coin in coins:
        while current_sum + 1 < coin:
            current_sum += current_sum + 1
            coins_added += 1
            if current_sum >= target:
                return coins_added
        current_sum += coin
        if current_sum >= target:
            return coins_added
    while current_sum < target:
        current_sum += current_sum + 1
        coins_added += 1

    return coins_added

```

Output

```
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.

main.py

```python
def can_assign(jobs, k, max_time):
    workers = [0] * k
    def backtrack(i):
        if i == len(jobs):  # All jobs have been assigned
            return True
        for j in range(k):
            if workers[j] + jobs[i] <= max_time:
                workers[j] += jobs[i]
                if backtrack(i + 1):
                    return True
                workers[j] -= jobs[i]
            if workers[j] == 0:
                break
        return False

    return backtrack(0)

def min_max_working_time(jobs, k):
    left, right = max(jobs), sum(jobs)
```

Output

```
3
11

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

main.py

```python
    while left < right:
        mid = (left + right) // 2
        if can_assign(jobs, k, mid):
            right = mid
        else:
            left = mid + 1

    return left

# Example usage
jobs1 = [3, 2, 3]
k1 = 3
print(min_max_working_time(jobs1, k1))  # Output: 3

jobs2 = [1, 2, 4, 7, 8]
k2 = 2
print(min_max_working_time(jobs2, k2))  # Output: 11
```

Output

```
3
11

=== 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.

main.py

```python
from bisect import bisect_right
def jobScheduling(startTime, endTime, profit):
    jobs = sorted(zip(endTime, startTime, profit))
    dp = [(0, 0)]
    for end, start, prof in jobs:
        i = bisect_right(dp, (start, float('inf'))) - 1
        new_profit = dp[i][1] + prof
        if new_profit > dp[-1][1]:
            dp.append((end, new_profit))

    return dp[-1][1]

# Example usage
startTime1 = [1, 2, 3, 3]
endTime1 = [3, 4, 5, 6]
profit1 = [50, 10, 40, 70]
print(jobScheduling(startTime1, endTime1, profit1))

startTime2 = [1, 2, 3, 4, 6]
```

Output

```
120
150

=== 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]

main.py | Share | Run | Output

```python
import heapq
def dijkstra(graph, source):
    n = len(graph)
    dist = [float('inf')] * n
    dist[source] = 0
    min_heap = [(0, source)]
    while min_heap:
        current_dist, u = heapq.heappop(min_heap)
        if current_dist > dist[u]:
            continue
        for v in range(n):
            if graph[u][v] != float('inf') and graph[u][v] != 0:
                new_dist = current_dist + graph[u][v]
                if new_dist < dist[v]:
                    dist[v] = new_dist
                    heapq.heappush(min_heap, (new_dist, v))
    return dist

n1 = 5
```

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

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

main.py | Share | Run | Output

```python
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(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(graph2, source2))  # Output: [0, 5, 8, 9]
```

```
[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

main.py Share Run

```python
import heapq
from collections import defaultdict, deque
def dijkstra(n, edges, source, target):
    graph = defaultdict(list)
    for u, v, w in edges:
        graph[u].append((v, w))
        graph[v].append((u, w))
    dist = [float('inf')] * n
    dist[source] = 0
    min_heap = [(0, source)]
    while min_heap:
        current_dist, u = heapq.heappop(min_heap)
        if u == target:
            return current_dist

        if current_dist > dist[u]:
            continue
        for v, weight in graph[u]:
            new_dist = current_dist + weight
            if new_dist < dist[v]:
```

Output

```
20
5

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

main.py Share Run

```python
                dist[v] = new_dist
                heapq.heappush(min_heap, (new_dist, v))

    return -1
# Test Case 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))  # Output: 20

# Test Case 2
n2 = 5
edges2 = [
```

Output

```
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
class Node:
    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):
    heap = []
    for char, freq in zip(characters, frequencies):
        heapq.heappush(heap, Node(char, freq))
    while len(heap) > 1:
        left = heapq.heappop(heap)
        right = heapq.heappop(heap)
        merged = Node(None, left.freq + right.freq)
        merged.left = left
        merged.right = right
```

```
[('a', '00'), ('b', '01'), ('c', '10'), ('d', '11')]

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

main.py

Share Run

```python
        heapq.heappush(heap, merged)
    return heap[0]

def generate_huffman_codes(root):
    huffman_codes = {}
    def generate_codes_helper(node, current_code):
        if node is None:
            return
        if node.char is not None:
            huffman_codes[node.char] = current_code
        generate_codes_helper(node.left, current_code + '0')
        generate_codes_helper(node.right, current_code + '1')
    generate_codes_helper(root, '')
    return huffman_codes

# Test Case 1
n1 = 4
characters1 = ['a', 'b', 'c', 'd']
frequencies1 = [5, 9, 12, 13]
root1 = build_huffman_tree(characters1, frequencies1)
```

Output

```
[('a', '00'), ('b', '01'), ('c', '10'), ('d', '11')]

=== 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 = '11001101110010111001011' Output: "fcbade”

main.py

Share Run

```python
            if new_dist < dist[v]:
                dist[v] = new_dist
                heapq.heappush(min_heap, (new_dist, v))

    return -1
# Test Case 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))  # Output: 20

# Test Case 2
n2 = 5
edges2 = [
```

Output

```
20
5

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

main.py

```python
        heapq.heappush(heap, merged)

    return heap[0]

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.char is not None:
            decoded_string += current_node.char
            current_node = root

    return decoded_string
```

Output

```
dbcbdd
fefcbaac

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

main.py

```python
import heapq
class Node:
    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):
    heap = []
    for char, freq in zip(characters, frequencies):
        heapq.heappush(heap, Node(char, freq))
    while len(heap) > 1:
        left = heapq.heappop(heap)
        right = heapq.heappop(heap)
        merged = Node(None, left.freq + right.freq)
        merged.left = left
        merged.right = right
```

Output

```
[('a', '00'), ('b', '01'), ('c', '10'), ('d', '11')]

=== 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

main.py

```python
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:
            continue

    return total_weight
weights1 = [10, 20, 30, 40, 50]
max_capacity1 = 60
print(max_weight_loaded(weights1, max_capacity1))  # Output: 50

# Test Case 2
weights2 = [5, 10, 15, 20, 25, 30]
max_capacity2 = 50
print(max_weight_loaded(weights2, max_capacity2))  # Output: 50
```

Output

```
60
50

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