NAME: S.DHARSHINI

REG NO: 192424258

SUB CODE: CSA0614

SUB NAME: DESIGN ANALYSIS AND ALGORITHM FOR APPROXIMATION PROBLEM

TOPIC 5 : GREEDY ALGORITHMS

EXP 1: Maximum Coins You Can Have

Aim: To find the maximum number of coins you can collect using a greedy strategy.
CODE:

main.py

```python
def maxCoins(piles):
    piles.sort()
    i = len(piles) - 2
    ans = 0
    while i > 0:
        ans += piles[i]
        i -= 2
    return ans

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

Output

```
9

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

RESULT: Maximum coins obtained successfully.

EXP 2: Minimum Coins to Add

Aim: To find the minimum number of coins needed so all values from 1 to target are obtainable.
CODE:

main.py

```python
def minCoins(coins, target):
    coins.sort()
    reach = 0
    count = 0
    i = 0

    while reach < target:
        if i < len(coins) and coins[i] <= reach + 1:
            reach += coins[i]
            i += 1
        else:
            reach += reach + 1
            count += 1
    return count

print(minCoins([1,4,10], 19))
```

Output

```
2

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

RESULT: Minimum coins added correctly.

EXP 3: Job Assignment (Minimum Max Time)

Aim: To minimize the maximum working time among workers.
CODE:

main.py

```python
def minTime(jobs, k):
    workers = [0]*k
    jobs.sort(reverse=True)
    for job in jobs:
        workers[workers.index(min(workers))] += job
    return max(workers)

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

Output

```
11

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

RESULT: Optimal job assignment achieved.

EXP 4: Job Scheduling for Maximum Profit

Aim: To find maximum profit without overlapping jobs.
CODE:

main.py

```python
def jobScheduling(start, end, profit):
    jobs = sorted(zip(start, end, profit), key=lambda x: x[1])
    dp = [0]*len(jobs)
    dp[0] = jobs[0][2]

    for i in range(1, len(jobs)):
        inc = jobs[i][2]
        for j in range(i-1, -1, -1):
            if jobs[j][1] <= jobs[i][0]:
                inc += dp[j]
                break
        dp[i] = max(dp[i-1], inc)
    return dp[-1]

print(jobScheduling([1,2,3,3],[3,4,5,6],[50,10,40,70]))
```

Output

```
120

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

RESULT: Maximum profit calculated successfully.

EXP 5: Dijkstra (Adjacency Matrix)
Aim: To find shortest paths from source to all vertices.
CODE:

main.py

```python
    n = len(graph)
    dist = [float('inf')]*n
    dist[src] = 0
    visited = [False]*n

    for _ in range(n):
        u = min((d,i) for i,d in enumerate(dist) if not
            visited[i])[1]
        visited[u] = True
        for v in range(n):
            if graph[u][v] != float('inf'):
                dist[v] = min(dist[v], dist[u] + graph[u][v])
    return dist

INF = float('inf')
graph = [[0,10,3,INF,INF],[INF,0,1,2,INF],[INF,4,0,8,2],[INF,INF
    ,INF,0,7],[INF,INF,INF,9,0]]
```

Output

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

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

RESULT: Shortest paths found correctly.

EXP 6: Dijkstra (Edge List)

Aim: To find shortest path from source to target.

CODE:

main.py

```python
def dijkstra(n, edges, src, tgt):
    g = [[] for _ in range(n)]
    for u,v,w in edges:
        g[u].append((v,w))
    pq = [(0, src)]
    dist = [float('inf')]*n
    dist[src] = 0

    while pq:
        d,u = heapq.heappop(pq)
        if u == tgt: return d
        for v,w in g[u]:
            if dist[v] > d + w:
                dist[v] = d + w
                heapq.heappush(pq, (dist[v], v))
    return -1

print(dijkstra(6, [(0,1,7),(0,2,9),(2,5,2),(5,4,9)], 0, 4))
```

Output

```
20

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

RESULT : Shortest path found.

EXP 7: Huffman Coding

Aim: To generate Huffman codes.
CODE:

main.py

```python
import heapq

def huffman(chars, freq):
    heap = [[f,[c,""]] for c,f in zip(chars,freq)]
    heapq.heapify(heap)
    while len(heap)>1:
        l = heapq.heappop(heap)
        r = heapq.heappop(heap)
        for p in l[1:]: p[1] = '0'+p[1]
        for p in r[1:]: p[1] = '1'+p[1]
        heapq.heappush(heap, [l[0]+r[0]] + l[1:] + r[1:])
    return sorted(heap[0][1:])

print(huffman(['a','b','c','d'], [5,9,12,13]))
```

Output

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

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

RESULT: Huffman codes generated.

EXP 8: Huffman Decoding

Aim: To decode a Huffman encoded string.
CODE

main.py

```python
    return result


# Constructing Huffman Tree
root = Node()
root.left = Node('A')
root.right = Node()
root.right.left = Node('B')
root.right.right = Node('C')

# Encoded string
encoded_string = "010011011"

# Decode
decoded = huffmanDecode(root, encoded_string)

```

Output

```
Decoded String: ABACAC

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

RESULT: Encoded string decoded successfully.

EXP 9: Max Weight (Greedy)

Aim: To load maximum weight without exceeding capacity.
CODE

main.py

```python
def maxLoad(weights, cap):
    weights.sort(reverse=True)
    total = 0
    for w in weights:
        if total + w <= cap:
            total += w
    return total

print(maxLoad([10,20,30,40,50], 60))
```

Output

```
60

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

RESULT: Maximum weight loaded.

EXP 10: Minimum Containers

Aim: To find minimum containers required.
CODE:

main.py

```python
def containers(weights, cap):
    weights.sort(reverse=True)
    cnt = 0
    while weights:
        load = 0
        i = 0
        while i < len(weights):
            if load + weights[i] <= cap:
                load += weights.pop(i)
            else:
                i += 1
        cnt += 1
    return cnt

print(containers([5,10,15,20,25,30,35], 50))
```

Output

```
3

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

RESULT: Minimum containers calculated.

EXP 11: Kruskal's Algorithm

Aim: To find MST and its weight.

CODE:

main.py

```python
def kruskal(n, edges):
    parent = list(range(n))
    def find(x):
        if parent[x]!=x:
            parent[x]=find(parent[x])
        return parent[x]

    edges.sort(key=lambda x:x[2])
    mst, cost = [], 0
    for u,v,w in edges:
        if find(u)!=find(v):
            parent[find(u)] = find(v)
            mst.append((u,v,w))
            cost += w
    return mst, cost

mst, cost = kruskal(4, [(0,1,10),(0,2,6),(0,3,5),(2,3,4)])
print(mst, cost)
```

Output

```
[(2, 3, 4), (0, 3, 5), (0, 1, 10)] 19

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

RESULT: MST found successfully.

EXP 12: Check MST Uniqueness

Aim : To check whether MST is unique.

CODE:

main.py

```python
                temp_weight += w

        if temp_weight == mst_weight:
            return "Not Unique"

    return "Unique"


# Example Graph
n = 4
edges = [
    (0, 1, 1),
    (1, 2, 1),
    (2, 3, 1),
    (0, 3, 1)
]
```

Output

```
MST Uniqueness: Not Unique

=== Code Execution Successful
```

RESULT: If multiple MSTs with same total weight exist → Not Unique