
NAME: S.DHARSHINI

REG NO: 192424258

SUB CODE: CSA0614

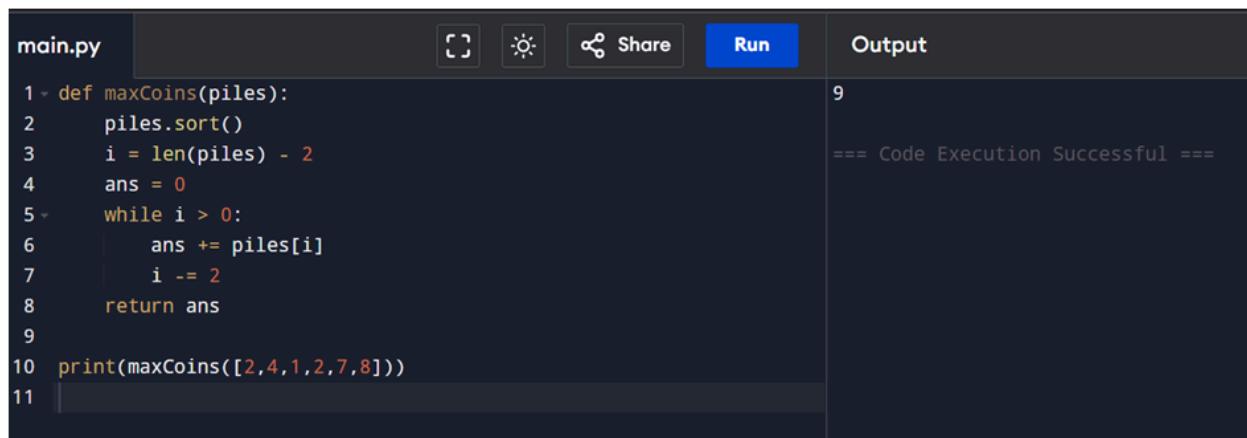
SUB NAME: DESIGN ANALYSIS AND ALGORITHM FOR APPROXIMATION PROBLEM

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
1 def maxCoins(piles):
2     piles.sort()
3     i = len(piles) - 2
4     ans = 0
5     while i > 0:
6         ans += piles[i]
7         i -= 2
8     return ans
9
10 print(maxCoins([2,4,1,2,7,8]))
11
```

The code editor interface shows the file name as "main.py". It has standard icons for copy, paste, and share, and a "Run" button which is highlighted in blue. To the right of the code area is an "Output" panel. The output shows the result of running the code: "9" followed by the message "==== Code Execution Successful ====". The code itself is a Python function named "maxCoins" that takes a list of integers as input. It first sorts the list. Then it initializes a variable "ans" to 0. It then enters a loop where it adds every second element from the end of the list to "ans" and then removes those two elements from the list by setting "i -= 2". This continues until "i" is no longer greater than 0. Finally, it returns the value of "ans". The code is run with the input [2,4,1,2,7,8] and the output is 9, indicating that the maximum number of coins that can be collected is 9.

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
```

```
1 def minCoins(coins, target):
2     coins.sort()
3     reach = 0
4     count = 0
5     i = 0
6
7     while reach < target:
8         if i < len(coins) and coins[i] <= reach + 1:
9             reach += coins[i]
10            i += 1
11        else:
12            reach += reach + 1
13            count += 1
14
15    return count
16
17 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
```

```
1 def minTime(jobs, k):
2     workers = [0]*k
3     jobs.sort(reverse=True)
4     for job in jobs:
5         workers[workers.index(min(workers))] += job
6
7
8 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
1 - def jobScheduling(start, end, profit):
2     jobs = sorted(zip(start, end, profit), key=lambda x: x[1])
3     dp = [0]*len(jobs)
4     dp[0] = jobs[0][2]
5
6     for i in range(1, len(jobs)):
7         inc = jobs[i][2]
8         for j in range(i-1, -1, -1):
9             if jobs[j][1] <= jobs[i][0]:
10                 inc += dp[j]
11                 break
12             dp[i] = max(dp[i-1], inc)
13
14
15 print(jobScheduling([1,2,3,3],[3,4,5,6],[50,10,40,70]))
16

```

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
1  n = len(graph)
2  dist = [float('inf')]*n
3  dist[src] = 0
4  visited = [False]*n
5
6
7  for _ in range(n):
8      u = min((d,i) for i,d in enumerate(dist) if not
9              visited[i])[1]
10     visited[u] = True
11     for v in range(n):
12         if graph[u][v] != float('inf'):
13             dist[v] = min(dist[v], dist[u] + graph[u][v])
14
15 INF = float('inf')
16 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
1  def dijkstra(n, edges, src, tgt):
2      g = [[] for _ in range(n)]
3      for u,v,w in edges:
4          g[u].append((v,w))
5      pq = [(0, src)]
6      dist = [float('inf')]*n
7      dist[src] = 0
8
9      while pq:
10         d,u = heapq.heappop(pq)
11         if u == tgt: return d
12         for v,w in g[u]:
13             if dist[v] > d + w:
14                 dist[v] = d + w
15                 heapq.heappush(pq, (dist[v], v))
16     return -1
17
18 print(dijkstra(6, [(0,1,7),(0,2,9),(2,5,2),(5,4,9)], 0, 4))

```

20
==== Code Execution Successful ===

RESULT : Shortest path found.**EXP 7: Huffman Coding****Aim:** To generate Huffman codes.**CODE:**

```

main.py
1  import heapq
2
3  def huffman(chars, freq):
4      heap = [[f,[c,""]] for c,f in zip(chars,freq)]
5      heapq.heapify(heap)
6      while len(heap)>1:
7          l = heapq.heappop(heap)
8          r = heapq.heappop(heap)
9          for p in l[1:]: p[1] = '0'+p[1]
10         for p in r[1:]: p[1] = '1'+p[1]
11         heapq.heappush(heap, [l[0]+r[0]] + l[1:] + r[1:])
12     return sorted(heap[0][1:])
13
14 print(huffman(['a','b','c','d'], [5,9,12,13]))

```

[['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
```

```
20
21     return result
22
23
24 # Constructing Huffman Tree
25 root = Node()
26 root.left = Node('A')
27 root.right = Node()
28 root.right.left = Node('B')
29 root.right.right = Node('C')
30
31 # Encoded string
32 encoded_string = "010011011"
33
34 # Decode
35 decoded = huffmanDecode(root, encoded_string)
36
```

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
```

```
1 def maxLoad(weights, cap):
2     weights.sort(reverse=True)
3     total = 0
4     for w in weights:
5         if total + w <= cap:
6             total += w
7     return total
8
9 print(maxLoad([10,20,30,40,50], 60))
10
```

Output

```
60
== Code Execution Successful ==
```

RESULT: Maximum weight loaded.

EXP 10: Minimum Containers

Aim: To find minimum containers required.

CODE:

```
main.py
```

```
1 def containers(weights, cap):
2     weights.sort(reverse=True)
3     cnt = 0
4     while weights:
5         load = 0
6         i = 0
7         while i < len(weights):
8             if load + weights[i] <= cap:
9                 load += weights.pop(i)
10            else:
11                i += 1
12            cnt += 1
13    return cnt
14
15 print(containers([5,10,15,20,25,30,35], 50))
16
```

Output: 3
==== Code Execution Successful ===

RESULT: Minimum containers calculated.

EXP 11: Kruskal's Algorithm

Aim: To find MST and its weight.

CODE:

```
main.py
```

```
1 def kruskal(n, edges):
2     parent = list(range(n))
3     def find(x):
4         if parent[x]!=x:
5             parent[x]=find(parent[x])
6         return parent[x]
7
8     edges.sort(key=lambda x:x[2])
9     mst, cost = [], 0
10    for u,v,w in edges:
11        if find(u)!=find(v):
12            parent[find(u)] = find(v)
13            mst.append((u,v,w))
14            cost += w
15    return mst, cost
16
17 mst, cost = kruskal(4, [(0,1,10),(0,2,6),(0,3,5),(2,3,4)])
18 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:

The screenshot shows a Jupyter Notebook interface. On the left, the code file 'main.py' is displayed with lines 29 through 44 visible. Lines 31 and 32 contain the logic for checking if two Minimum Spanning Trees (MSTs) have the same total weight. Line 37 defines an example graph with 4 nodes and 4 edges. The 'Run' button is highlighted in blue at the top right. To the right of the code area, there is an 'Output' section. It displays the text 'MST Uniqueness: Not Unique' followed by '==== Code Execution Successful'. The background of the interface is dark.

```
main.py
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)
]
```

MST Uniqueness: Not Unique
==== Code Execution Successful

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