

main.py

```
1 - def maxCoins(piles):
2     piles.sort()
3     total_coins = 0
4     n = len(piles) // 3
5     for i in range(n):
6         total_coins += piles[-(2 + i)]
7     return total_coins
8
9 # Example usage
10 print(maxCoins([2, 4, 1, 2, 7, 8]))
11 print(maxCoins([2, 4, 5]))
12
```



Share



Run

Output

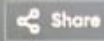
11

4

=== Code Execution Successful ===



main.py



Output

Clear

```
1- def min_coins_to_add(coins, target):
2-     coins.sort()
3-     current_sum = 0
4-     count = 0
5-     for coin in coins:
6-         while current_sum + 1 < coin and current_sum < target:
7-             count += 1
8-             current_sum += current_sum + 1
9-             current_sum += coin
10-        while current_sum < target:
11-            count += 1
12-            current_sum += current_sum + 1
13-        return count
14-
15- # Example usage
16- coins = [1, 4, 10]
17- target = 19
18- print(min_coins_to_add(coins, target)) # Output: 2
19-
```

2

=== Code Execution Successful ===

Minimum Maximum Working Time of Workers

3

=== Code Execution S

```
3- def minimumTimeRequired(jobs, k):
4-     def canDistribute(maxTime):
5-         workers = [0] * k
6-         return backtrack(0, workers, maxTime)
7-
8-     def backtrack(i, workers, maxTime):
9-         if i == len(jobs):
10-            return True
11-         for j in range(k):
12-             if workers[j] + jobs[i] <= maxTime:
13-                 workers[j] += jobs[i]
14-                 if backtrack(i + 1, workers, maxTime):
15-                     return True
16-                 workers[j] -= jobs[i]
17-             if workers[j] == 0: # No need to try further if this worker is still idle
18-                 break
19-         return False
20-
21-     left, right = max(jobs), sum(jobs)
22-     while left < right:
23-         mid = (left + right) // 2
24-         if canDistribute(mid):
25-             right = mid
26-         else:
27-             left = mid
28-     return left
29-
30- # Example usage
31- jobs = [3, 2, 3]
32- k = 3
33- print(minimumTimeRequired(jobs, k)) # Output: 3
```



```

1 def jobScheduling(startTime, endTime, profit):
2     jobs = sorted(zip(startTime, endTime, profit), key=lambda x: x[1])
3     dp = [0] * (len(jobs) + 1)
4
5     def binarySearch(index):
6         low, high = 0, index - 1
7         while low <= high:
8             mid = (low + high) // 2
9             if jobs[mid][1] <= jobs[index][0]:
10                 low = mid + 1
11             else:
12                 high = mid - 1
13         return high
14
15     for i in range(1, len(jobs) + 1):
16         incl_profit = jobs[i - 1][2]
17         l = binarySearch(i - 1)
18         if l != -1:
19             incl_profit += dp[l + 1]
20         dp[i] = max(incl_profit, dp[i - 1])
21
22     return dp[-1]
23
24 # Example usage
25 startTime = [1, 2, 3, 3]
26 endTime = [3, 4, 5, 6]
27 profit = [50, 10, 40, 70]
28 print(jobScheduling(startTime, endTime, profit))

```

120

=== Code Execution

main.py

```
1 import sys
2
3 def dijkstra(graph, source):
4     n = len(graph)
5     distances = [sys.maxsize] * n
6     distances[source] = 0
7     visited = [False] * n
8
9     for _ in range(n):
10         min_distance = sys.maxsize
11         min_index = -1
12
13         for v in range(n):
14             if not visited[v] and distances[v] < min_distance:
15                 min_distance = distances[v]
16                 min_index = v
17
18         visited[min_index] = True
19
20         for v in range(n):
21             if (graph[min_index][v] > 0 and not visited[v] and
22                 distances[min_index] + graph[min_index][v] < distances[v]):
23                 distances[v] = distances[min_index] + graph[min_index][v]
24
25     return distances
26
27 # Test Case 1
28 n = 5
29 graph = [[0, 10, 3, float('inf'), float('inf')],
30          [float('inf'), 0, 1, 2, float('inf')],
31          [float('inf'), 4, 0, 8, 2],
32          [float('inf'), float('inf'), float('inf'), 0, 7],
33          [float('inf'), float('inf'), float('inf'), 9, 0]]
34 source = 0
35
36 output = dijkstra(graph, source)
37 print(output)
```

input

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

```
...Program finished with exit code 0
Press ENTER to exit console.
```




```
1 import heapq
2
3 def dijkstra(n, edges, source, target):
4     graph = {i: [] for i in range(n)}
5     for u, v, w in edges:
6         graph[u].append((v, w))
7         graph[v].append((u, w))
8
9     min_heap = [(0, source)]
10    distances = {i: float('inf') for i in range(n)}
11    distances[source] = 0
12
13    while min_heap:
14        current_distance, current_vertex = heapq.heappop(min_heap)
15
16        if current_vertex == target:
17            return current_distance
18
19        if current_distance > distances[current_vertex]:
20            continue
21
22        for neighbor, weight in graph[current_vertex]:
23            distance = current_distance + weight
24
25            if distance < distances[neighbor]:
26                distances[neighbor] = distance
27                heapq.heappush(min_heap, (distance, neighbor))
28
29    return distances[target]
30
31 # Test Case
32 n = 6
33 edges = [(0, 1, 7), (0, 2, 9), (0, 5, 14), (1, 2, 10), (1, 3, 15),
34          (2, 3, 11), (2, 5, 2), (3, 4, 6), (4, 5, 9)]
35 source = 0
36 target = 4
37 print(dijkstra(n, edges, source, target))
```

20

=== Code Exec

main.py



Output

Clear

```
1 import heapq
2 from collections import defaultdict
3
4 class Node:
5     def __init__(self, char, freq):
6         self.char = char
7         self.freq = freq
8         self.left = None
9         self.right = None
10
11     def __lt__(self, other):
12         return self.freq < other.freq
13
14 def huffman_codes(characters, frequencies):
15     heap = [Node(char, freq) for char, freq in zip(characters, frequencies)]
16     heapq.heapify(heap)
17     while len(heap) > 1:
18         left = heapq.heappop(heap)
19         right = heapq.heappop(heap)
20         merged = Node(None, left.freq + right.freq)
21         merged.left = left
22         merged.right = right
23         heapq.heappush(heap, merged)
24     codes = {}
25     def generate_codes(node, current_code):
26         if node:
27             if node.char is not None:
28                 codes[node.char] = current_code
29             generate_codes(node.left, current_code + '0')
30             generate_codes(node.right, current_code + '1')
31     generate_codes(heap[0], '')
32     return [(char, codes[char]) for char in characters]
33
34 # Test Case
35 n = 4
36 characters = ['a', 'b', 'c', 'd']
37 frequencies = [5, 9, 12, 13]
38 output = huffman_codes(characters, frequencies)
39 print(output)
```

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

=== Code Execution Successful ===



29°C

Partly cloudy

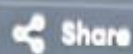


Search

ENG
IN

19:15

19-10-2024



Run

Output

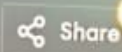
```
1- class Node:
2-     def __init__(self, char, freq):
3-         self.char = char
4-         self.freq = freq
5-         self.left = None
6-         self.right = None
7-
8- def build_huffman_tree(characters, frequencies):
9-     nodes = [Node(char, freq) for char, freq in zip(characters, frequencies)]
10-    while len(nodes) > 1:
11-        nodes = sorted(nodes, key=lambda x: x.freq)
12-        left = nodes[0]
13-        right = nodes[1]
14-        merged = Node(None, left.freq + right.freq)
15-        merged.left = left
16-        merged.right = right
17-        nodes = nodes[2:] + [merged]
18-    return nodes[0]
19-
20- def decode_huffman_tree(root, encoded_string):
21-     decoded_string = []
22-     current_node = root
23-     for bit in encoded_string:
24-         current_node = current_node.left if bit == '0' else current_node.right
25-         if current_node.char:
26-             decoded_string.append(current_node.char)
27-             current_node = root
28-     return ''.join(decoded_string)
29-
30- n = 4
31- characters = ['a', 'b', 'c', 'd']
32- frequencies = [5, 9, 12, 13]
33- encoded_string = '1101100111110'
34-
35- huffman_tree = build_huffman_tree(characters, frequencies)
36- output = decode_huffman_tree(huffman_tree, encoded_string)
37- print(output) # Output: "abacd"
```

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

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


main.py

```
1 - def max_weight(weights, max_capacity):
2     weights.sort(reverse=True)
3     total_weight = 0
4
5     for weight in weights:
6         if total_weight + weight <= max_capacity:
7             total_weight += weight
8
9     return total_weight
10
11 # Test Case 1
12 n1 = 5
13 weights1 = [10, 20, 30, 40, 50]
14 max_capacity1 = 60
15 output1 = max_weight(weights1, max_capacity1)
16 print(output1) # Output: 50
```



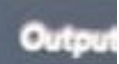
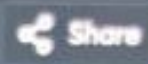
Share

Run

Output

60

=== Code Execution Successful ===



```
1 - def min_containers(weights, max_capacity):
2     weights.sort(reverse=True)
3     containers = 0
4     current_capacity = 0
5
6     for weight in weights:
7         if current_capacity + weight > max_capacity:
8             containers += 1
9             current_capacity = weight
10        else:
11            current_capacity += weight
12
13    if current_capacity > 0:
14        containers += 1
15
16    return containers
17
18 # Test Case 1
19 n = 7
20 weights = [5, 10, 15, 20, 25, 30, 35]
21 max_capacity = 50
22 result = min_containers(weights, max_capacity)
23 print(result)
```

4

=== Code

```

- class DisjointSet:
2-     def __init__(self, n):
3-         self.parent = list(range(n))
4-         self.rank = [0] * n
5-     def find(self, u):
6-         if self.parent[u] != u:
7-             self.parent[u] = self.find(self.parent[u])
8-         return self.parent[u]
9-     def union(self, u, v):
10-         root_u = self.find(u)
11-         root_v = self.find(v)
12-         if root_u != root_v:
13-             if self.rank[root_u] > self.rank[root_v]:
14-                 self.parent[root_v] = root_u
15-             elif self.rank[root_u] < self.rank[root_v]:
16-                 self.parent[root_u] = root_v
17-             else:
18-                 self.parent[root_v] = root_u
19-                 self.rank[root_u] += 1
20
21-     def kruskal(n, edges):
22-         edges.sort(key=lambda x: x[2])
23-         ds = DisjointSet(n)
24-         mst = []
25-         total_weight = 0
26
27-         for u, v, weight in edges:
28-             if ds.find(u) != ds.find(v):
29-                 ds.union(u, v)
30-                 mst.append((u, v, weight))
31-                 total_weight += weight
32
33-         return mst, total_weight
34
35- n = 4
36- edges = [(0, 1, 10), (0, 2, 6), (0, 3, 5), (1, 3, 15), (2, 3, 4)]
37- mst, total_weight = kruskal(n, edges)
38- print("Edges in MST:", mst)
39- print("Total weight of MST:", total_weight)

```

4

=== Code Exec

main.py

```
1 class Graph:
2     def __init__(self, vertices):
3         self.V = vertices
4         self.graph = []
5     def add_edge(self, u, v, w):
6         self.graph.append((w, u, v))
7     def find_parent(self, parent, i):
8         if parent[i] == i:
9             return i
10        return self.find_parent(parent, parent[i])
11    def union(self, parent, rank, x, y):
12        xroot = self.find_parent(parent, x)
13        yroot = self.find_parent(parent, y)
14        if rank[xroot] < rank[yroot]:
15            parent[xroot] = yroot
16        elif rank[xroot] > rank[yroot]:
17            parent[yroot] = xroot
18        else:
19            parent[yroot] = xroot
20            rank[xroot] += 1
21
22    def is_unique_mst(self, given_mst):
23        parent = []
24        rank = []
25        for node in range(self.V):
26            parent.append(node)
27            rank.append(0)
28        mst_weight = 0
29        for u, v, w in given_mst:
30            mst_weight += w
31            self.union(parent, rank, u, v)
32        return mst_weight
33
34 n = 4
35 edges = [(0, 1, 10), (0, 2, 6), (0, 3, 5), (1, 3, 15), (2, 3, 4)]
36 given_mst = [(2, 3, 4), (0, 3, 5), (0, 1, 10)]
37 g = Graph(n)
38 for edge in edges:
39     g.add_edge(*edge)
40 unique = g.is_unique_mst(given_mst)
41 print(f"Is the given MST unique? {unique}")
```

Output

▲ Is the given MST unique? 19

=== Code Execution Successful ===