main.py

Share Run

```python
def min_coins_to_add(coins, target):
    coins.sort()
    current_sum = 0
    additional_coins = 0

    for x in range(1, target + 1):
        if x > current_sum:
            additional_coins += 1
            current_sum += x
        else:
            for coin in coins:
                if coin <= current_sum:
                    current_sum += coin
                else:
                    break

    return additional_coins

coins = [1, 4, 10]
target = 19
print(min_coins_to_add(coins, target))
```

Output

```
2

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

main.py

Share Run

```python
def canDistribute(jobs, k, max_time):
    workers = [0] * k
    return backtrack(jobs, workers, 0, max_time)

def backtrack(jobs, workers, index, max_time):
    if index == len(jobs):
        return True
    for i in range(len(workers)):
        if workers[i] + jobs[index] <= max_time:
            workers[i] += jobs[index]
            if backtrack(jobs, workers, index + 1, max_time):
                return True
            workers[i] -= jobs[index]
        if workers[i] == 0:
            break
    return False

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

jobs = [3, 2, 3]
k = 3
print(minimumTimeRequired(jobs, k))
```

Output

```
3

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

main.py

Share Run

Output

```python
def jobScheduling(startTime, endTime, profit):
    jobs = sorted(zip(startTime, endTime, profit), key=lambda x: x[1])
    dp = [0] * (len(jobs) + 1)

    for i in range(1, len(jobs) + 1):

        current_profit = jobs[i - 1][2]

        j = i - 1
        while j > 0 and jobs[j - 1][1] > jobs[i - 1][0]:
            j -= 1
        dp[i] = max(dp[i - 1], current_profit + dp[j])

    return dp[-1]

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

```
120

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

main.py

Share Run

```python
def dijkstra(graph, source):
    n = len(graph)
    distances = [float('inf')] * n
    distances[source] = 0
    visited = [False] * n

    for _ in range(n):
        min_distance = float('inf')
        min_index = -1
        for v in range(n):
            if not visited[v] and distances[v] < min_distance:
                min_distance = distances[v]
                min_index = v
        visited[min_index] = True
        for v in range(n):
            if graph[min_index][v] != float('inf') and not visited[v]:
                new_distance = distances[min_index] + graph[min_index][v]
                if new_distance < distances[v]:
                    distances[v] = new_distance
    return distances
n = 5
graph = [
    [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]
]
source = 0
shortest_paths = dijkstra(graph, source)
print(shortest_paths)
```

Output

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

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

main.py

Share Run

```python
def dijkstra(n, edges, source, target):

    graph = {i: [] for i in range(n)}
    for u, v, w in edges:
        graph[u].append((v, w))
        graph[v].append((u, w))
    distances = [float('inf')] * n
    distances[source] = 0
    visited = set()
    while len(visited) < n:
        min_distance = float('inf')
        min_vertex = None
        for vertex in range(n):
            if vertex not in visited and distances[vertex] < min_distance:
                min_distance = distances[vertex]
                min_vertex = vertex
        if min_vertex is None:
            break
        visited.add(min_vertex)
        for neighbor, weight in graph[min_vertex]:
            if neighbor not in visited:
                new_distance = distances[min_vertex] + weight
                if new_distance < distances[neighbor]:
                    distances[neighbor] = new_distance
    return distances[target] if distances[target] != float('inf') else None

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

```
The shortest path from 0 to 4 is: 20

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

main.py

Share Run

```python
def max_weight(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

n = 5
weights = [10, 20, 30, 40, 50]
max_capacity = 60
result = max_weight(weights, max_capacity)
print(f"Maximum weight that can be loaded: {result}")
```

Output

```
Maximum weight that can be loaded: 50

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

main.py

Share Run

Output

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

    for weight in weights:
        if current_capacity + weight > max_capacity:
            containers += 1
            current_capacity = weight
        else:
            current_capacity += weight

    if current_capacity > 0:
        containers += 1

    return containers

n = 7
weights = [5, 10, 15, 20, 25, 30, 35]
max_capacity = 50
print(min_containers(weights, max_capacity))
```

```
4

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

main.py

Share Run Output

```python
class DisjointSet:
    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])
    ds = DisjointSet(n)
```

```
4

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

```
    mst = []
    total_weight = 0

    for u, v, weight in edges:
        if ds.find(u) != ds.find(v):
            ds.union(u, v)
            mst.append((u, v, weight))
            total_weight += weight

    return mst, total_weight

n = 4
edges = [(0, 1, 10), (0, 2, 6), (0, 3, 5), (1, 3, 15), (2, 3, 4)]
mst, total_weight = kruskal(n, edges)
print("Minimum Spanning Tree:", mst)
print("Total Weight:", total_weight)
```

Online Python Compiler - onlin

onlinegdb.com/online_python_compiler

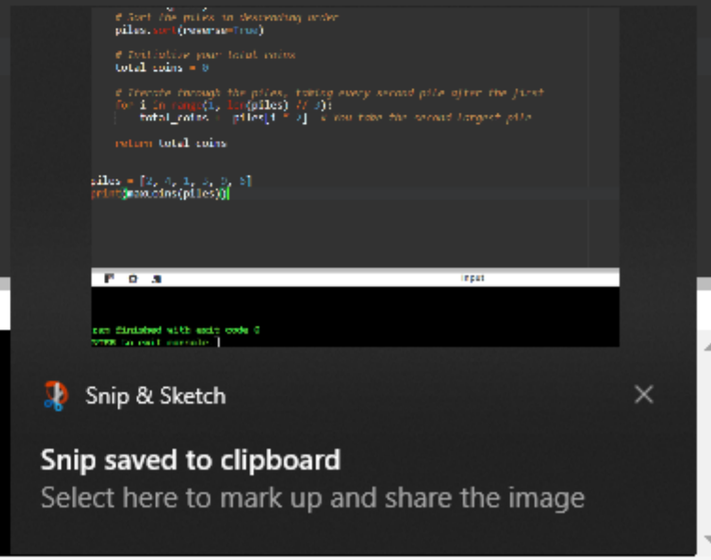
Run | Debug | Stop | Share | Save | { } Beautify | Language Python 3

main.py

```python
def maxCoins(piles):
    # Sort the piles in descending order
    piles.sort(reverse=True)

    # Initialize your total coins
    total_coins = 0

    # Iterate through the piles, taking every second pile after the first
    for i in range(1, len(piles) // 3):
        total_coins += piles[i * 2]  # You take the second largest pile

    return total_coins


piles = [2, 4, 1, 3, 9, 6]
print(maxCoins(piles))
```

input

```
4


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



Snip & Sketch

Snip saved to clipboard

Select here to mark up and share the image