

# GREEDY

## EXPERIMENT-1

**AIM:** To write a simple program to find the maximum number of coins you can collect from piles when three players (Alice, You, and Bob) pick coins in order from the piles.

### PROCEDURE:

- Start by taking the number of coins in each pile as input.
- Sort all the piles in ascending order.
- Divide the piles into groups of three from the largest end.
- Add up the coins you picked and display the total as the final result.

### PROGRAM:



The screenshot shows a code editor window titled "main.py". The code defines a function maxCoins that takes a list of piles as input. It sorts the piles, calculates the length of the piles list, initializes a variable "coins" to 0, and then iterates from index n to len(piles)-1, adding each pile's value to "coins". Finally, it prints the total number of coins. The code is as follows:

```
1 def maxCoins(piles):  
2     piles.sort()  
3     n = len(piles) // 3  
4     coins = 0  
5     for i in range(n, len(piles), 2):  
6         coins += piles[i]  
7     return coins  
8 piles = [2, 4, 1, 2, 7, 8]  
9 print("Maximum coins you can have:", maxCoins(piles))  
10
```

### Output:



The screenshot shows a terminal window with the title "Output". It displays the output of the program, which is "Maximum coins you can have: 9", followed by a message indicating successful code execution. The output is as follows:

```
Output  
Maximum coins you can have: 9  
==== Code Execution Successful ===
```

### Result:

Thus the program implemented successfully.

# EXPERIMENT-2

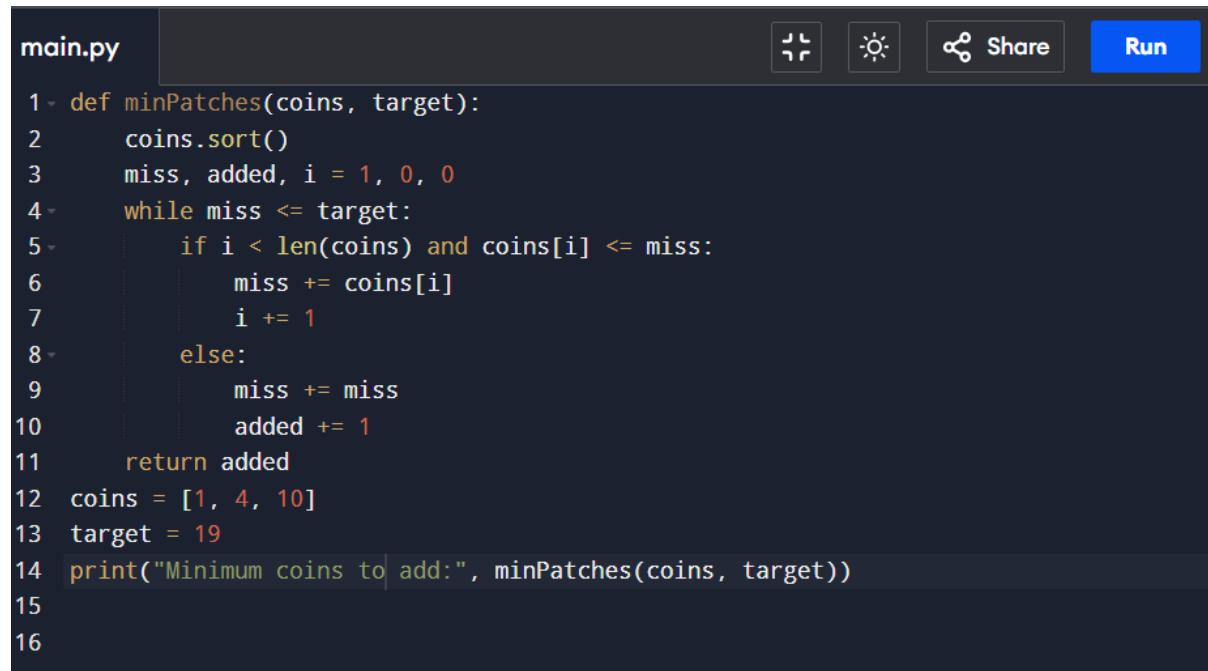
## AIM:

To write a program that finds the **minimum number of coins** that need to be added to make all integers in the range [1, target] obtainable as sums of subsequences of the given coins.

## PROCEDURE:

- Start with the given list of coins and a target value.
- Sort the coins in ascending order.
- Initialize a variable miss = 1 (the smallest value that cannot yet be formed).
- For each coin:
  - If the coin value  $\leq$  miss, extend the range of obtainable sums.
  - Otherwise, add a new coin of value miss to fill the gap.
- Repeat until miss > target.
- The count of added coins is the final answer.

## PROGRAM:



The screenshot shows a code editor window with a dark theme. The file is named "main.py". The code implements a greedy algorithm to find the minimum number of coins required to form all sums from 1 to a target value. It starts by sorting the input coins, then iterates through them, adding each to a running total "miss" if it's less than or equal to "miss". If it's greater, it adds a new coin of value "miss" to "miss" and increments the count "added". Finally, it prints the total number of coins needed.

```
main.py
1 def minPatches(coins, target):
2     coins.sort()
3     miss, added, i = 1, 0, 0
4     while miss <= target:
5         if i < len(coins) and coins[i] <= miss:
6             miss += coins[i]
7             i += 1
8         else:
9             miss += miss
10            added += 1
11    return added
12 coins = [1, 4, 10]
13 target = 19
14 print("Minimum coins to add:", minPatches(coins, target))
15
16
```

## OUTPUT:

```
Output
Minimum coins to add: 2
==== Code Execution Successful ====
```

## RESULT:

Thus the program implemented successfully.

# EXPERIMENT-3

## AIM:

To write a program that distributes jobs among workers so that the maximum working time among all workers is minimized.

## PROCEDURE:

- Start with the list of job times and the number of workers.
- Use a **backtracking** approach to assign jobs to workers one by one.
- Keep track of each worker's total working time.
- If a worker's total exceeds the current best (minimum possible maximum time), stop exploring that path.
- Continue until all jobs are assigned and record the minimum possible maximum time.
- Display the result.

## PROGRAM:

```
main.py
```

```
1 def minimumTimeRequired(jobs, k):
2     jobs.sort(reverse=True)
3     workers = [0] * k
4     res = sum(jobs)
5     def backtrack(i):
6         nonlocal res
7         if i == len(jobs):
8             for w in range(k):
9                 if workers[w] + jobs[i] < res:
10                     workers[w] += jobs[i]
11                     backtrack(i + 1)
12                     workers[w] -= jobs[i]
13                     if workers[w] == 0:
14                         break
15     backtrack(0)
16     return res
17 jobs = [3, 2, 3]
18 k = 3
19 print("Minimum possible maximum working time:", minimumTimeRequired(jobs, k))
```

## OUTPUT:

```
Output
```

```
Minimum possible maximum working time: 3
== Code Execution Successful ==
```

## RESULT:

Thus the program implemented successfully.

# EXPERIMENT-4

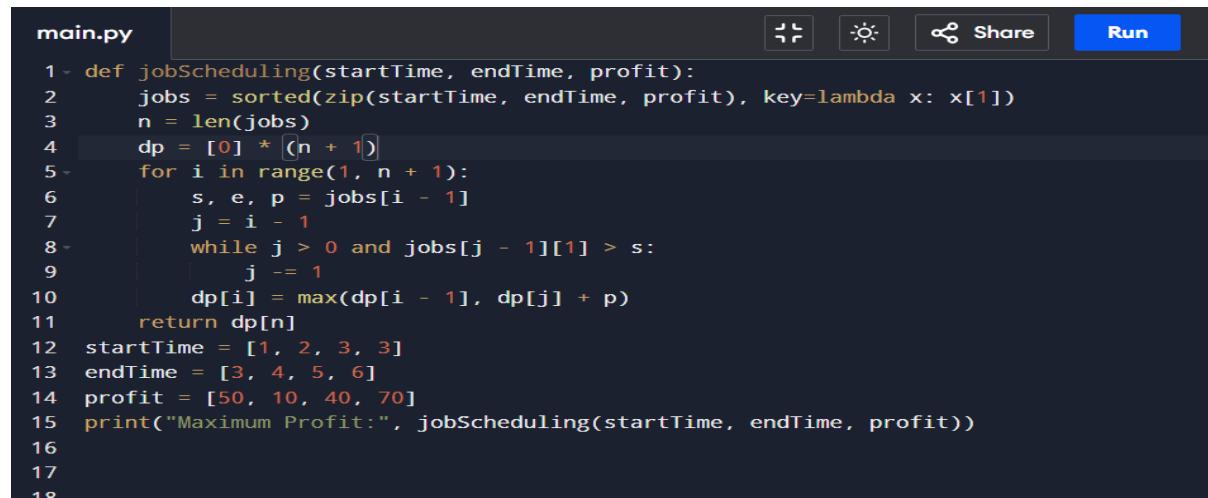
## AIM:

To write a program that finds the maximum profit from non-overlapping jobs using Weighted Job Scheduling.

## PROCEDURE:

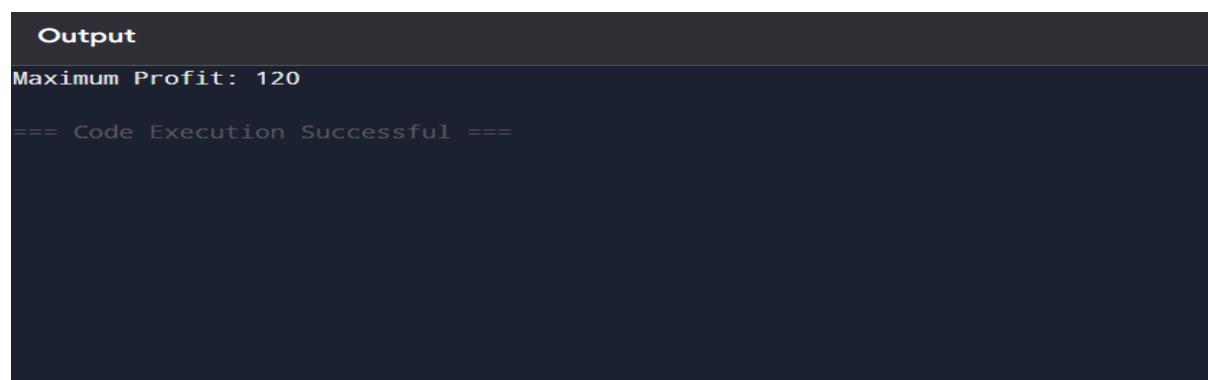
- Start with three arrays — startTime, endTime, and profit.
- Combine these into a list of jobs with their start, end, and profit values.
- Sort the jobs by **end time** to process them efficiently.
- For each job, find the last job that doesn't overlap (using binary search or loop).
- Use **Dynamic Programming** to calculate the maximum profit by including or excluding each job.
- The final value gives the maximum total profit possible without overlapping jobs.

## PROGRAM:



```
main.py
1 def jobScheduling(startTime, endTime, profit):
2     jobs = sorted(zip(startTime, endTime, profit), key=lambda x: x[1])
3     n = len(jobs)
4     dp = [0] * (n + 1)
5     for i in range(1, n + 1):
6         s, e, p = jobs[i - 1]
7         j = i - 1
8         while j > 0 and jobs[j - 1][1] > s:
9             j -= 1
10            dp[i] = max(dp[i - 1], dp[j] + p)
11    return dp[n]
12 startTime = [1, 2, 3, 3]
13 endTime = [3, 4, 5, 6]
14 profit = [50, 10, 40, 70]
15 print("Maximum Profit:", jobScheduling(startTime, endTime, profit))
16
17
18
```

## OUTPUT:



```
Output
Maximum Profit: 120
== Code Execution Successful ==
```

## RESULT:

Thus the program implemented successfully.

## EXPERIMENT-5

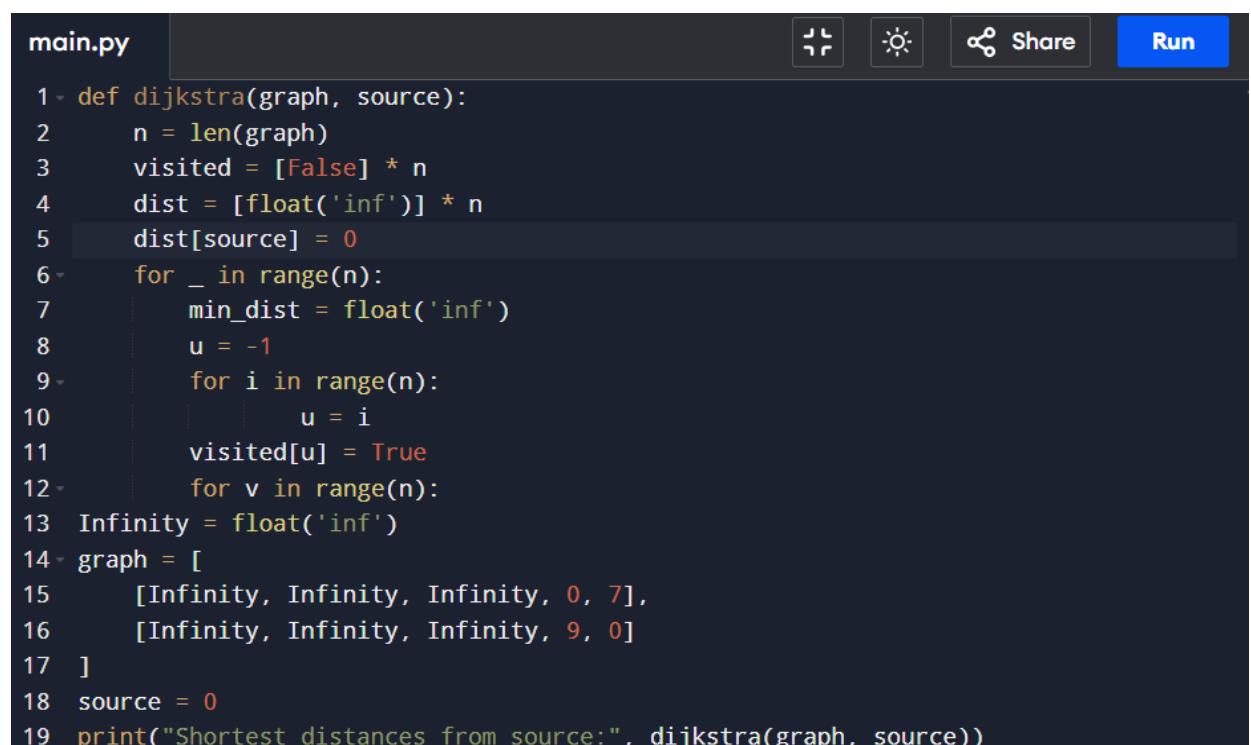
### AIM:

To write a program using Dijkstra's Algorithm to find the shortest distance from a source vertex to all other vertices in a graph represented by an adjacency matrix.

### PROCEDURE:

- Start by representing the graph using an adjacency matrix.
- Initialize a list dist[] to store the shortest distance from the source to each vertex.
- Set the distance of the source vertex as 0 and others as infinity.
- Create a visited[] list to mark vertices that are processed.
- Update the distances of its adjacent vertices if a shorter path is found.
- Repeat until all vertices are visited.
- Display the shortest distance from the source to all vertices.

### PROGRAM:



The screenshot shows a code editor window titled "main.py". The code implements Dijkstra's algorithm to find the shortest distances from a source vertex to all other vertices in a graph represented by an adjacency matrix. The code uses a list of lists for the graph, where each row represents a vertex and contains the shortest distance to all other vertices. The source vertex is set to 0, and all other vertices are initially marked as infinity. The algorithm iterates through all vertices, updating the shortest distances to their neighbors. A "Run" button is visible in the top right corner of the editor.

```
1 def dijkstra(graph, source):  
2     n = len(graph)  
3     visited = [False] * n  
4     dist = [float('inf')] * n  
5     dist[source] = 0  
6     for _ in range(n):  
7         min_dist = float('inf')  
8         u = -1  
9         for i in range(n):  
10             if dist[i] < min_dist:  
11                 min_dist = dist[i]  
12                 u = i  
13         visited[u] = True  
14         for v in range(n):  
15             if graph[u][v] != float('inf') and not visited[v]:  
16                 dist[v] = min(dist[v], graph[u][v])  
17  
18 source = 0  
19 print("Shortest distances from source:", dijkstra(graph, source))
```

## **OUTPUT:**

```
Output Cle  
Shortest distances from source: [0, 7, 3, 9, 5]  
==== Code Execution Successful ===
```

## **RESULT:**

Thus the program implemented successfully.