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:

main.py

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

Output:

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 ≤ 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:

main.py

```
def minPatches(coins, target):
    coins.sort()
    miss, added, i = 1, 0, 0
    while miss <= target:
        if i < len(coins) and coins[i] <= miss:
            miss += coins[i]
            i += 1
        else:
            miss += miss
            added += 1
    return added
coins = [1, 4, 10]
target = 19
print("Minimum coins to add:", minPatches(coins, target))
```

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

Share

Run

```python
def minimumTimeRequired(jobs, k):
    jobs.sort(reverse=True)
    workers = [0] * k
    res = sum(jobs)
    def backtrack(i):
        nonlocal res
        if i == len(jobs):
        for w in range(k):
            if workers[w] + jobs[i] < res:
                workers[w] += jobs[i]
                backtrack(i + 1)
                workers[w] -= jobs[i]
            if workers[w] == 0:
                break
    backtrack(0)
    return res
jobs = [3, 2, 3]
k = 3
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

```python
def jobScheduling(startTime, endTime, profit):
    jobs = sorted(zip(startTime, endTime, profit), key=lambda x: x[1])
    n = len(jobs)
    dp = [0] * (n + 1)
    for i in range(1, n + 1):
        s, e, p = jobs[i - 1]
        j = i - 1
        while j > 0 and jobs[j - 1][1] > s:
            j -= 1
        dp[i] = max(dp[i - 1], dp[j] + p)
    return dp[n]
startTime = [1, 2, 3, 3]
endTime = [3, 4, 5, 6]
profit = [50, 10, 40, 70]
print("Maximum Profit:", jobScheduling(startTime, endTime, profit))
```

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:

main.py Share Run

```
def dijkstra(graph, source):
    n = len(graph)
    visited = [False] * n
    dist = [float('inf')] * n
    dist[source] = 0
    for _ in range(n):
        min_dist = float('inf')
        u = -1
        for i in range(n):
                u = i
        visited[u] = True
        for v in range(n):
Infinity = float('inf')
graph = [
    [Infinity, Infinity, Infinity, 0, 7],
    [Infinity, Infinity, Infinity, 9, 0]
]
source = 0
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.