Topic-4 Dynamic Programming

1. Dice Throw Problem using Dynamic Programming

Aim: Find number of ways to get a target sum with given number of dice and sides.

Algorithm:

Step-1: Create DP table dp[dice+1][target+1]

Step-2: Initialize dp[0][0] = 1

Step-3: For each dice, for each sum, add ways using each face value

Step-4: Return dp[num_dice][target] **Input**

& Output:

main.py

```python
def dice_throw(num_sides, num_dice, target):
    dp = [[0 for _ in range(target + 1)] for _ in range(num_dice + 1)]
    dp[0][0] = 1
    for i in range(1, num_dice + 1):
        for j in range(1, target + 1):
            for k in range(1, num_sides + 1):
                if j >= k:
                    dp[i][j] += dp[i - 1][j - k]
    return dp[num_dice][target]
num_sides_1 = 6
num_dice_1 = 2
target_1 = 7
result_1 = dice_throw(num_sides_1, num_dice_1, target_1)
num_sides_2 = 4
num_dice_2 = 3
target_2 = 10
result_2 = dice_throw(num_sides_2, num_dice_2, target_2)
print(f"Number of sides: {num_sides_1}, Number of dice: {num_dice_1}, Target sum: {target_1}")
print(f"Number of ways to reach sum {target_1}: {result_1}\n")
print(f"Number of sides: {num_sides_2}, Number of dice: {num_dice_2}, Target sum: {target_2}")
print(f"Number of ways to reach sum {target_2}: {result_2}")
```

Output

```
Number of sides: 6, Number of dice: 2, Target sum: 7
Number of ways to reach sum 7: 6

Number of sides: 4, Number of dice: 3, Target sum: 10
Number of ways to reach sum 10: 6

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

2. Assembly Line Scheduling (2 Lines)

Aim: Find minimum time to assemble product through 2 lines.

Algorithm:

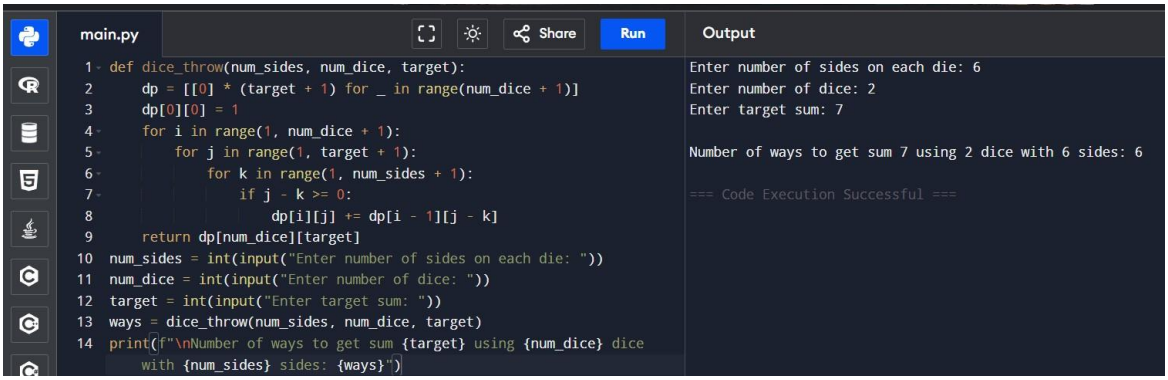
Step-1: Use DP arrays T1, T2 for each line

Step-2: Compute time at each station with/without transfer

Step-3: Add entry and exit times

Step-4: Return min(T1[n-1]+x1, T2[n-1]+x2) **Input**

& Output:



3. Three Assembly Lines Scheduling

Aim: Minimize total time across 3 lines with dependencies.

Algorithm:

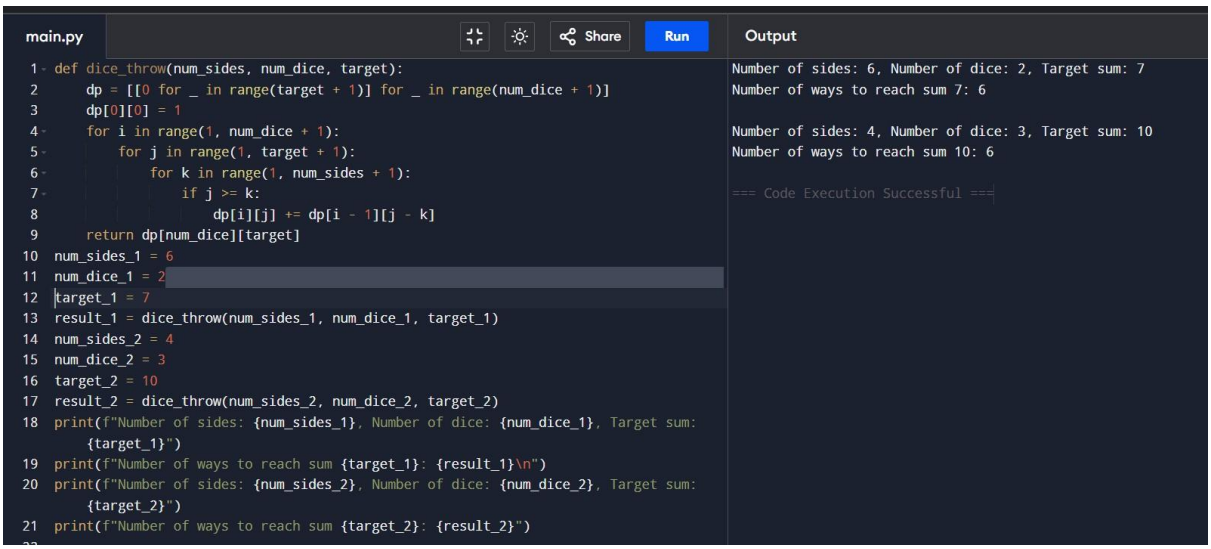
Step-1: Initialize dp[i][line] = time

Step-2: Add min transfer from previous station

Step-3: Respect dependencies

Step-4: Return min total

Input & Output



4. Minimum Path Distance (Matrix form - TSP)

Aim: Find minimum path visiting all cities (TSP).

Algorithm:

Step-1: Use DP with bitmasking Step-2:
Recursively compute all paths Step-3:
Return minimal cycle cost.

Input & Output:

main.py

```python
N = 4
dist = [
    [0, 10, 15, 20],
    [10, 0, 35, 25],
    [15, 35, 0, 30],
    [20, 25, 30, 0]
]
memo = [[-1]*(1<<N) for _ in range(N)]
def tsp(city, visited):
    if visited == (1 << N) - 1:
        return dist[city][0]

    if memo[city][visited] != -1:
        return memo[city][visited]

    ans = float('inf')
    for next_city in range(N):
        if not (visited & (1 << next_city)):
            temp = dist[city][next_city] + tsp(next_city, visited | (1 << next_city
                ))
            ans = min(ans, temp)
    memo[city][visited] = ans
```

Output

```
Minimum Path Distance: 80

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

5. TSP with New City (E)

Aim: Find shortest route including new city.

Algorithm:
Step-1: Use permutations to check all paths
Step-2: Compute total distance
Step-3: Return minimal route

Input & Output

main.py

```python
N = 5
cities = ['A', 'B', 'C', 'D', 'E']
dist = [
    [0, 10, 15, 20, 25],
    [10, 0, 35, 25, 30],
    [15, 35, 0, 30, 20],
    [20, 25, 30, 0, 15],
    [25, 30, 20, 15, 0]
]
memo = [[-1]*(1<<N) for _ in range(N)]
path_memo = [[-1]*(1<<N) for _ in range(N)]

def tsp(city, visited):
    if visited == (1 << N) - 1:
        return dist[city][0]

    if memo[city][visited] != -1:
        return memo[city][visited]

    ans = float('inf')
    for next_city in range(N):
        if not (visited & (1 << next_city)):
            temp = dist[city][next_city] + tsp(next_city, visited | (1 <<
                next_city))
            if temp < ans:
                ans = temp
```

Output

```
Minimum Distance: 85
Optimal Route: A -> B -> D -> E -> C -> A

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