DYNAMIC PROGRAMMING

EXPERIMENT :1

Aim: To find the number of ways to get a given target sum using a specified number of dice and sides.

Procedure:

☐ Create a DP table dp[dice][sum] to store ways.

☐ Initialize base case for one die.

☐ For each die, add ways from previous dice rolls.

☐ Use nested loops for dice count and sums.

☐ Display number of ways for given target.

PROGRAM:

```
def dice_throw(num_dice, num_sides, target):
    dp = [[0]*(target+1) for _ in range(num_dice+1)]
    dp[0][0] = 1

    for dice in range(1, num_dice+1):
        for t in range(1, target+1):
            for face in range(1, num_sides+1):
                if t - face >= 0:
                    dp[dice][t] += dp[dice-1][t-face]
    return dp[num_dice][target]

# Test Cases
print("Test Case 1:")
print("Number of ways to reach sum 7:", dice_throw(2,6,7))

print("Test Case 2:")
print("Number of ways to reach sum 10:", dice_throw(3,4,10))
```

OUTPUT:

```
Test Case 1:
Number of ways to reach sum 7: 6
Test Case 2:
Number of ways to reach sum 10: 6

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

RESULT:

The program successfully computes the total number of ways to reach the target sum using dynamic programming.

EXPERIMENT:2

AIM: To determine the minimum time required to process a product through two assembly lines.

PROCEDURE:

- Use dynamic programming with two arrays T1[] and T2[].
- Compute time at each station considering transfer times.
- Choose minimum time between staying or switching lines.
- Add entry and exit times.
- Return minimum total time.

PROGRAM:

```
def assembly_line(a1, a2, t1, t2, e1, e2, x1, x2):
    n = len(a1)
    T1 = [0]*n
    T2 = [0]*n

    T1[0] = e1 + a1[0]
    T2[0] = e2 + a2[0]

    for i in range(1, n):
        T1[i] = min(T1[i-1] + a1[i], T2[i-1] + t2[i-1] + a1[i])
        T2[i] = min(T2[i-1] + a2[i], T1[i-1] + t1[i-1] + a2[i])

    return min(T1[-1] + x1, T2[-1] + x2)

# Test Case
a1 = [4,5,3,2]
a2 = [2,10,1,4]
t1 = [7,4,5]
t2 = [9,2,8]
```

OUTPUT:

```
Minimum time required: 35


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

RESULT:

The minimum processing time for both assembly lines was successfully computed using dynamic programming

EXPERIMENT:3

AIM: To minimize production time across three assembly lines with transfer times and dependencies.

PROCEDURE:

- Represent station times and transfer times in matrices.
- Use DP to compute minimum time at each station for each line.
- Respect dependencies between stations.
- Compare and store minimum cumulative times.
- Return overall minimum time.

PROGRAM:

```
import sys
def three_line_schedule(times, transfer):
    n = len(times[0])
    lines = len(times)
    dp = [[0]*n for _ in range(lines)]

    for i in range(lines):
        dp[i][0] = times[i][0]

    for j in range(1, n):
        for i in range(lines):
            dp[i][j] = min(dp[k][j-1] + transfer[k][i] for k in range(lines)) +
                times[i][j]
    return min(dp[i][-1] for i in range(lines))

times = [
    [5,9,3],
    [6,8,4],
    [7,6,5]
```

OUTPUT:

```
A module you have imported isn't available at the moment. It will be available soon.

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

RESULT:

The algorithm successfully minimized total production time across three dependent assembly lines.

EXPERIMENT:4

AIM: To find the minimum path distance using matrix form.

PROCEDURE:

- Represent distances between cities in a matrix.
- Use permutations or recursion to find the shortest route.
- Apply Traveling Salesman Problem logic.
- Compute total path cost and find minimum.
- Display the shortest path distance.

PROGRAM:

```
import itertools

def min_tsp_cost(matrix):
    n = len(matrix)
    cities = list(range(n))
    start = 0
    min_cost = float('inf')
    best_path = None
    for perm in itertools.permutations(cities[1:]):  # fix 0 as start
        path = (0,) + perm + (0,)
        cost = 0
        valid = True
        for i in range(len(path)-1):
            if matrix[path[i]][path[i+1]] == 0 and path[i] != path[i+1]:
                valid = False
                break
            cost += matrix[path[i]][path[i+1]]
        if valid and cost < min_cost:
            min_cost = cost
```

OUTPUT:

```
Test Case 1: Minimum Path Distance = 80, Path = A -> B -> D -> C -> A
Test Case 2: Minimum Path Distance = 40, Path = A -> B -> C -> D -> A
Test Case 3: Minimum Path Distance = 14, Path = A -> B -> C -> D -> A

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

RESULT:

program successfully finds the minimum path cost using matrix representation.

EXPERIMENT:5

AIM: To find the shortest route for five cities using symmetric distance matrix.

PROCEDURE:

- Represent cities and distances in a matrix.
- Use permutation-based TSP solution.
- Calculate total distance for each route.
- Track minimum distance and best path.
- Display optimal route and total distance.

PROGRAM:

```
import itertools

cities = ['A','B','C','D','E']
dist = {
 ('A','B'):10, ('A','C'):15, ('A','D'):20, ('A','E'):25,
 ('B','A'):10, ('B','C'):35, ('B','D'):25, ('B','E'):30,
 ('C','A'):15, ('C','B'):35, ('C','D'):30, ('C','E'):20,
 ('D','A'):20, ('D','B'):25, ('D','C'):30, ('D','E'):15,
 ('E','A'):25, ('E','B'):30, ('E','C'):20, ('E','D'):15
}

min_path = None
min_cost = float('inf')

for perm in itertools.permutations(cities[1:]):  # Fix A as start
    path = ['A'] + list(perm) + ['A']
    cost = sum(dist[(path[i], path[i+1])] for i in range(len(path)-1))
    if cost < min_cost:
        min_cost = cost
```

OUTPUT:

```
Shortest Route: A -> B -> D -> E -> C -> A
Total Distance: 85

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

RESULT:

The shortest route and total distance for 5 cities were successfully determined using the Traveling Salesperson approach.