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:

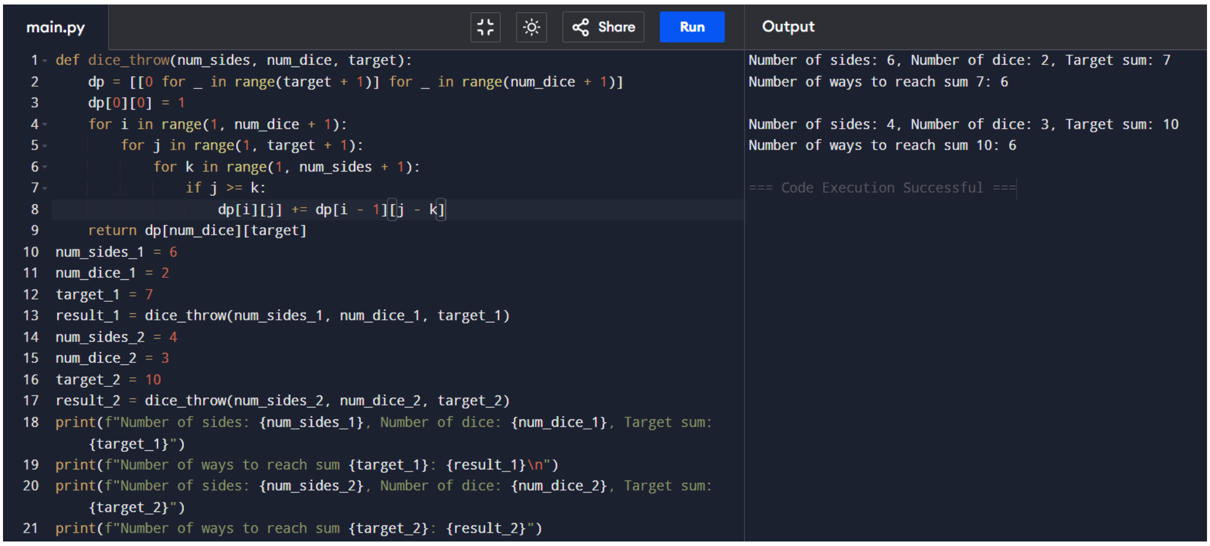
Step-1: Create DP table dp[dice+1][target+1]

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

Share

Run

Output

```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}")
```

```
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:

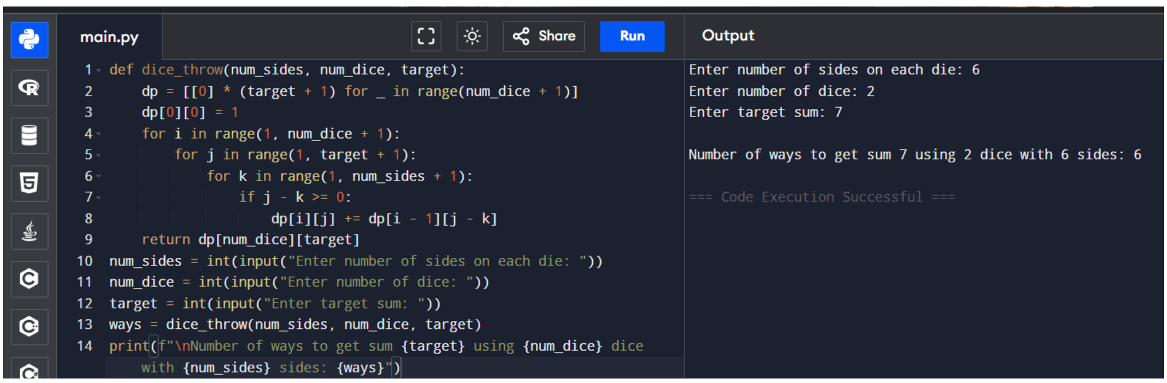
Step-1: Use DP arrays T1, T2 for each line

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:


main.py

Share

Run

Output

```python
def dice_throw(num_sides, num_dice, target):
    dp = [[0] * (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 >= 0:
                    dp[i][j] += dp[i - 1][j - k]
    return dp[num_dice][target]
num_sides = int(input("Enter number of sides on each die: "))
num_dice = int(input("Enter number of dice: "))
target = int(input("Enter target sum: "))
ways = dice_throw(num_sides, num_dice, target)
print(f"\nNumber of ways to get sum {target} using {num_dice} dice with {num_sides} sides: {ways}")
```

```
Enter number of sides on each die: 6
Enter number of dice: 2
Enter target sum: 7

Number of ways to get sum 7 using 2 dice with 6 sides: 6

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

3. Three Assembly Lines Scheduling

Aim: Minimize total time across 3 lines with dependencies.

Algorithm:

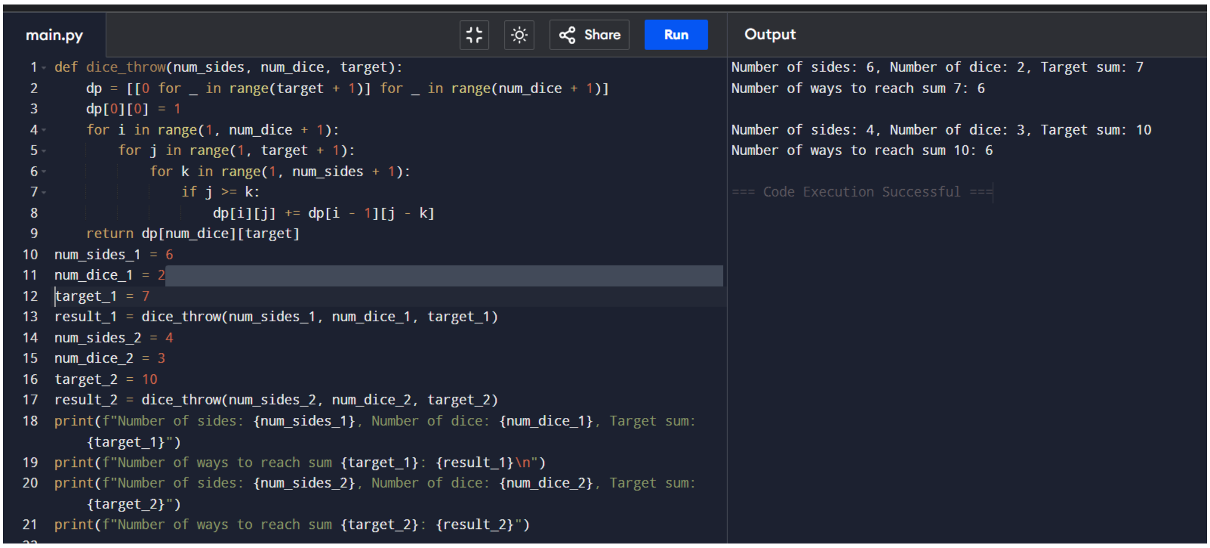
Step-1: Initialize dp[i][line] = time

Step-2: Add min transfer from previous station

Step-3: Respect dependencies

Step-4: Return min total

Input & Output


main.py

```
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 ===
```

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 ===
```

6. Longest Palindromic Substring

Aim: To find the longest substring in a given string s that is a palindrome.

Algorithm:

Step-1: Initialize a function to expand around center.

Step-2: For each character in the string, expand around it (odd and even centers).

Step-3: Keep track of the longest palindrome found.

Step-4: Return the substring between start and end indexes.

Input & Output

main.py

```python
def longest_palindrome(s):
    start = end = 0
    for i in range(len(s)):
        l, r = i, i
        while l >= 0 and r < len(s) and s[l] == s[r]:
            if r - l > end - start:
                start, end = l, r
            l -= 1
            r += 1
        l, r = i, i + 1
        while l >= 0 and r < len(s) and s[l] == s[r]:
            if r - l > end - start:
                start, end = l, r
            l -= 1
            r += 1
    return s[start:end+1]
print(longest_palindrome("babad"))
print(longest_palindrome("cbbd"))
```

Output

```
bab
bb

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

7. Longest Substring Without Repeating Characters

Aim:

To find the length of the longest substring without repeating characters.

Algorithm:

Step-1: Use sliding window with set to store characters.

Step-2: Move right pointer and add characters until repetition occurs.

Step-3: Move left pointer to remove repeated characters.

Step-4: Track maximum length found.

Input & Output

main.py

```python
def length_of_longest_substring(s):
    char_set = set()
    l = 0
    max_len = 0
    for r in range(len(s)):
        while s[r] in char_set:
            char_set.remove(s[l])
            l += 1
        char_set.add(s[r])
        max_len = max(max_len, r - l + 1)
    return max_len
print(length_of_longest_substring("abcabcbb"))
print(length_of_longest_substring("bbbbb"))
print(length_of_longest_substring("pwwkew"))
```

Output

```
3
1
3

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

8. Word Break Problem (Segment String into Dictionary Words)

Aim:

To check if the string s can be segmented into dictionary words.

Algorithm:

Step-1: Create a DP array of size n+1 initialized with False.

Step-2: Set dp[0] = True.

Step-3: For each index i, check all j < i.

Step-4: If dp[j] is True and substring s[j:i] in dictionary, set dp[i] = True.

Step-5: Return dp[n] **as final answer.**

Input & Output:

main.py

```python
def wordBreak(s, wordDict):
    n = len(s)
    dp = [False] * (n + 1)
    dp[0] = True
    for i in range(1, n + 1):
        for j in range(i):
            if dp[j] and s[j:i] in wordDict:
                dp[i] = True
                break
    return dp[n]
print("Input: s = 'leetcode', wordDict = ['leet','code']")
print("Output:", wordBreak("leetcode", ["leet", "code"]))
print("Input: s = 'applepenapple', wordDict = ['apple','pen']")
print("Output:", wordBreak("applepenapple", ["apple", "pen"]))
print("Input: s = 'catsandog', wordDict = ['cats','dog','sand','and','cat']")
print("Output:", wordBreak("catsandog", ["cats", "dog", "sand", "and", "cat"]))
```

Output

```
Input: s = 'leetcode', wordDict = ['leet','code']
Output: True
Input: s = 'applepenapple', wordDict = ['apple','pen']
Output: True
Input: s = 'catsandog', wordDict = ['cats','dog','sand','and','cat']
Output: False

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

9. Word Break Problem

Aim: To determine if a given string can be segmented into a sequence of valid dictionary words.

Algorithm:

Step-1:Use dynamic programming with a boolean array dp where dp[i] indicates if substring s[0:i] can be segmented.

Step-2:Initialize dp[0] = True (empty string is segmentable).

Step-3:For each index i, check all j < i to see if dp[j] is True and s[j:i] is in the dictionary.

Step-4:Return dp[len(s)].

Input & Output

main.py

```python
def word_break(s, wordDict):
    word_set = set(wordDict)
    n = len(s)
    dp = [False] * (n + 1)
    dp[0] = True

    for i in range(1, n+1):
        for j in range(i):
            if dp[j] and s[j:i] in word_set:
                dp[i] = True
                break
    return dp[n]

# Test Cases
wordDict = ["i","like","sam","sung","samsung","mobile","ice","cream","icecream"
    ,"man","go","mango"]
print(word_break("ilike", wordDict))        # Output: True
print(word_break("ilikesamsung", wordDict)) # Output: True
```

Output

```
True
True

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

10. Text Justification

Aim: To format a list of words such that each line has exactly maxWidth characters and is fully justified.

Algorithm:

Step-1: Use a greedy approach to pack as many words as possible in one line.

Step-2: Calculate spaces to distribute evenly between words.

Step-3: For the last line or lines with one word, left-justify.

Step-4: Build each line by concatenating words and spaces accordingly.

Input & Output

main.py

```python
def full_justify(words, maxWidth):
    res, curr, num_of_letters = [], [], 0

    for word in words:
        if num_of_letters + len(word) + len(curr) > maxWidth:
            for i in range(maxWidth - num_of_letters):
                curr[i % (len(curr)-1 or 1)] += ' '
            res.append(''.join(curr))
            curr, num_of_letters = [], 0
        curr.append(word)
        num_of_letters += len(word)

    # Last line
    res.append(' '.join(curr).ljust(maxWidth))
    return res

# Test Case
words1 = ["This", "is", "an", "example", "of", "text", "justification."]
print(full_justify(words1, 16))
```

Output

```
['This    is    an', 'example  of text', 'justification.

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

11. Word Filter (Prefix & Suffix Search)

Aim: To design a dictionary that efficiently returns the index of a word having a given prefix and suffix.

Algorithm:

Step-1: Preprocess all words and store combinations of prefix#suffix in a dictionary with the word index.

Step-2: For f(pref, suff), check pref#suff in the dictionary and return the largest index.

Input & Output:

main.py

```python
class WordFilter:
    def __init__(self, words):
        self.lookup = {}
        for index, word in enumerate(words):
            for i in range(len(word)+1):
                for j in range(len(word)+1):
                    self.lookup[word[:i] + '#' + word[j:]] = index

    def f(self, pref, suff):
        return self.lookup.get(pref + '#' + suff, -1)
wf = WordFilter(["apple"])
print(wf.f("a", "e"))
```

Output

```
0

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

12. Floyd's Algorithm (All-Pairs Shortest Path)

Aim: To find the shortest path between all pairs of cities.

Algorithm:

Step-1: Initialize distance matrix with given edges.

Step-2: Apply Floyd’s algorithm:

for k in range(n):

for i in range(n):

for j in range(n):

dist[i][j] = min(dist[i][j], dist[i][k] + dist[k][j])

Step-3: Print distance before and after.

Step-4: Print shortest path.

Input & Output:

main.py Share Run

```python
INF = float('inf')
def floyd_warshall(n, edges, distanceThreshold):
    dist = [[INF] * n for _ in range(n)]
    for i in range(n):
        dist[i][i] = 0
    for u, v, w in edges:
        dist[u][v] = w
        dist[v][u] = w
    print("Distance Matrix Before Floyd's Algorithm:")
    for row in dist:
        print(row)
    for k in range(n):
        for i in range(n):
            for j in range(n):
                if dist[i][k] + dist[k][j] < dist[i][j]:
                    dist[i][j] = dist[i][k] + dist[k][j]
    print("\nDistance Matrix After Floyd's Algorithm:")
    for row in dist:
        print(row)
    reachable_counts = []
    for i in range(n):
        count = 0
        for j in range(n):
            if i != j and dist[i][j] <= distanceThreshold:
                count += 1
        reachable_counts.append(count)
```

Output

```
Distance Matrix Before Floyd's Algorithm:
[0, 3, inf, inf]
[3, 0, 1, 4]
[inf, 1, 0, 1]
[inf, 4, 1, 0]

Distance Matrix After Floyd's Algorithm:
[0, 3, 4, 5]
[3, 0, 1, 2]
[4, 1, 0, 1]
[5, 2, 1, 0]

Reachable cities count within threshold 4 : [2, 3, 3, 2]
City with smallest number of reachable cities: 3

Output: 3

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

13. Floyd’s Algorithm (Routers with Link Failure)

Aim: To find shortest paths between routers and update when link fails.

Algorithm:

Step-1: Initialize adjacency matrix for routers.

Step-2: Run Floyd’s algorithm.

Step-3: Print path A→F.

Step-4: Remove link (set INF), rerun algorithm, print new path.

Input & Output:

main.py

```python
import math
def floyd_warshall(dist):
    n = len(dist)
    for k in range(n):
        for i in range(n):
            for j in range(n):
                if dist[i][k] + dist[k][j] < dist[i][j]:
                    dist[i][j] = dist[i][k] + dist[k][j]
    return dist
INF = math.inf
routers = ['A', 'B', 'C', 'D', 'E', 'F']
dist = [
    [0, 2, INF, 1, INF, INF],
    [2, 0, 3, 2, INF, INF],
    [INF, 3, 0, 4, 2, INF],
    [1, 2, 4, 0, 3, 6],
    [INF, INF, 2, 3, 0, 1],
    [INF, INF, INF, 6, 1, 0]
]
print("Shortest paths BEFORE link failure:")
before = floyd_warshall([row[:] for row in dist])
for row in before:
    print(row)
a_to_f_before = before[0][5]
print("Shortest path from Router A to F before failure =", a_to_f_before)
```

Output

```
Shortest paths BEFORE link failure:
[0, 2, 5, 1, 4, 5]
[2, 0, 3, 2, 5, 6]
[5, 3, 0, 4, 2, 3]
[1, 2, 4, 0, 3, 4]
[4, 5, 2, 3, 0, 1]
[5, 6, 3, 4, 1, 0]
Shortest path from Router A to F before failure = 5

Shortest paths AFTER link failure:
[0, 2, 5, 1, 4, 5]
[2, 0, 3, 3, 5, 6]
[5, 3, 0, 4, 2, 3]
[1, 3, 4, 0, 3, 4]
[4, 5, 2, 3, 0, 1]
[5, 6, 3, 4, 1, 0]
Shortest path from Router A to F after failure = 5

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

14. Floyd's Algorithm (Custom Graph)

Aim: To find the shortest path between given cities using Floyd's algorithm.

Algorithm:
Step-1: Initialize distance matrix using edges.
Step-2: Apply Floyd's algorithm.
Step-3: Print required shortest path.

Input & Output:

main.py

```python
import math
def floyd_warshall(n, edges, distanceThreshold):
    INF = math.inf
    dist = [[INF] * n for _ in range(n)]
    for i in range(n):
        dist[i][i] = 0
    for u, v, w in edges:
        dist[u][v] = w
        dist[v][u] = w
    print("Distance matrix before applying Floyd's Algorithm:")
    for row in dist:
        print(row)
    for k in range(n):
        for i in range(n):
            for j in range(n):
                if dist[i][k] + dist[k][j] < dist[i][j]:
                    dist[i][j] = dist[i][k] + dist[k][j]
    print("\nDistance matrix after applying Floyd's Algorithm:")
    for row in dist:
        print(row)
    city_counts = []
    for i in range(n):
        count = 0
        for j in range(n):
            if i != j and dist[i][j] <= distanceThreshold:
                count += 1
```

Output

```
Distance matrix before applying Floyd's Algorithm:
[0, 2, inf, inf, 8]
[2, 0, 3, inf, 2]
[inf, 3, 0, 1, inf]
[inf, inf, 1, 0, 1]
[8, 2, inf, 1, 0]

Distance matrix after applying Floyd's Algorithm:
[0, 2, 5, 5, 4]
[2, 0, 3, 3, 2]
[5, 3, 0, 1, 2]
[5, 3, 1, 0, 1]
[4, 2, 2, 1, 0]

Reachable cities count within threshold = 2
City 0: 1 cities
City 1: 2 cities
City 2: 2 cities
City 3: 2 cities
City 4: 3 cities

City with smallest reachable cities = 0

Output: 0
=== Code Execution Successful ===
```

15. Optimal Binary Search Tree

Aim: To construct OBST with minimal search cost.

Algorithm:

Step-1: Initialize cost[i][i] = freq[i].
Step-2: Fill table using dynamic programming.
Step-3: For each range, find root giving minimum cost.
Step-4: Display cost and root matrices.

Input & Output:

main.py

```python
def optimal_bst(keys, freq, n):
    cost = [[0 for _ in range(n+2)] for _ in range(n+2)]
    root = [[0 for _ in range(n+2)] for _ in range(n+2)]
    for i in range(1, n+2):
        cost[i][i-1] = 0
    for length in range(1, n+1):
        for i in range(1, n-length+2):
            j = i + length - 1
            cost[i][j] = float('inf')
            total = sum(freq[i-1:j])
            for r in range(i, j+1):
                c = cost[i][r-1] + cost[r+1][j] + total
                if c < cost[i][j]:
                    cost[i][j] = c
                    root[i][j] = r
    return cost, root
keys = ['A', 'B', 'C', 'D']
freq = [0.1, 0.2, 0.4, 0.3]
n = len(keys)
cost, root = optimal_bst(keys, freq, n)
print("Cost Matrix:")
for i in range(1, n+1):
    print(cost[i][1:n+1])
print("\nRoot Matrix:")
for i in range(1, n+1):
    print(root[i][1:n+1])
```

Output

```
Cost Matrix:
[0.1, 0.4, 1.1, 1.7]
[0, 0.2, 0.8, 1.4]
[0, 0, 0.4, 1.0]
[0, 0, 0, 0.3]

Root Matrix:
[1, 2, 3, 3]
[0, 2, 3, 3]
[0, 0, 3, 3]
[0, 0, 0, 4]

Minimum cost of OBST: 1.7

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

Q16. Optimal Binary Search Tree (OBST)

AIM:

Construct an OBST using given keys and frequencies and find its minimum cost.

Algorithm:

- Step-1: Input keys and frequencies.
- Step-2: Compute cumulative frequency sums.
- Step-3: Use DP to compute cost and root matrices.
- Step-4: Return final cost and root matrix.

Input & Output:

main.py

```python
def optimal_bst(keys, freq):
    n = len(keys)
    cost = [[0]*n for _ in range(n)]
    root = [[0]*n for _ in range(n)]
    sum_freq = [0]*(n+1)
    for i in range(1, n+1):
        sum_freq[i] = sum_freq[i-1] + freq[i-1]
    for i in range(n):
        cost[i][i] = freq[i]
        root[i][i] = i
    for L in range(2, n+1):
        for i in range(n-L+1):
            j = i+L-1
            cost[i][j] = float('inf')
            total = sum_freq[j+1] - sum_freq[i]
            for r in range(i, j+1):
                c = (0 if r == i else cost[i][r-1]) + (0 if r == j else cost[r
                    +1][j]) + total
                if c < cost[i][j]:
                    cost[i][j] = c
                    root[i][j] = r
    print("Minimum Cost:", cost[0][n-1])
    print("Cost Matrix:")
    for row in cost: print(row)
    print("Root Matrix:")
    for row in root: print(row)
```

Output

```
Minimum Cost: 26
Cost Matrix:
[4, 8, 20, 26]
[0, 2, 10, 16]
[0, 0, 6, 12]
[0, 0, 0, 3]
Root Matrix:
[0, 0, 2, 2]
[0, 1, 2, 2]
[0, 0, 2, 2]
[0, 0, 0, 3]

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

Q17. Mouse and Cat Game

Aim: Determine winner (Mouse, Cat, or Draw) in a turn-based graph game.

Algorithm:

Step-1: Represent game states.

Step-2: Apply BFS/DP to track possible outcomes.

Step-3: Detect repetition/draw states.

Step-4: Return result: 1(Mouse), 2(Cat), 0(Draw).

Input & Output:

main.py

```python
from collections import deque
def catMouseGame(graph):
    n = len(graph)
    DRAW, MOUSE, CAT = 0, 1, 2
    color = [[[DRAW]*3 for _ in range(n)] for _ in range(n)]
    degree = [[[0]*3 for _ in range(n)] for _ in range(n)]
    for m in range(n):
        for c in range(n):
            degree[m][c][1] = len(graph[m])
            degree[m][c][2] = len([x for x in graph[c] if x != 0])
    q = deque()
    for i in range(n):
        for t in [1,2]:
            if i != 0:
                color[0][i][t] = MOUSE; q.append((0,i,t,MOUSE))
                color[i][i][t] = CAT; q.append((i,i,t,CAT))
    while q:
        m,c,t,res = q.popleft()
        for mm,cc,tt in parents(graph,m,c,t):
            if color[mm][cc][tt] != DRAW: continue
            if tt == res or all(color[x][y][tt] == res for x,y,_ in parents(graph
                ,mm,cc,tt)):
                color[mm][cc][tt] = res
                q.append((mm,cc,tt,res))
    return color[1][2][1]
def parents(graph, m, c, t):
```

Output

```
0

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

Q18. Maximum Probability Path

Aim: Find path from start to end with max success probability.

Algorithm:

Step-1: Represent graph.

Step-2: Use modified Dijkstra with probabilities.

Step-3: Track maximum probability.

Step-4: Return max probability.

Input & Output:

main.py

```python
import heapq
def maxProbability(n, edges, succProb, start, end):
    g = [[] for _ in range(n)]
    for (u,v),p in zip(edges, succProb):
        g[u].append((v,p)); g[v].append((u,p))
    prob = [0]*n; prob[start]=1
    heap=[(-1,start)]
    while heap:
        p,u = heapq.heappop(heap)
        p = -p
        if u == end: return p
        for v,w in g[u]:
            if p*w > prob[v]:
                prob[v]=p*w
                heapq.heappush(heap, (-prob[v], v))
    return 0.0
print(maxProbability(3, [[0,1],[1,2],[0,2]], [0.5,0.5,0.2], 0, 2))
```

Output

```
0.25

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

Q19. Unique Paths

Aim: Count unique paths from top-left to bottom-right in grid.

Algorithm:

Step-1: Initialize dp table.

Step-2: Fill first row/col with 1.

Step-3: Fill others with sum of top and left cells.

Step-4: Return dp[m-1][n-1].

Input & Output:

main.py

```python
def uniquePaths(m, n):
    dp = [[1]*n for _ in range(m)]
    for i in range(1,m):
        for j in range(1,n):
            dp[i][j] = dp[i-1][j]+dp[i][j-1]
    return dp[-1][-1]

print(uniquePaths(3,7))
```

Output

```
28

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

Q20. Good Pairs

Aim: Count pairs (i, j) such that nums[i] == nums[j] and i < j.

Algorithm:

Step-1: Use dictionary for frequency count.

Step-2: For each count, add count*(count-1)//2 to result.

Step-3: Return total pairs.

Input & Output:

main.py

```python
def numIdenticalPairs(nums):
    from collections import Counter
    count = Counter(nums)
    return sum(v*(v-1)//2 for v in count.values())

print(numIdenticalPairs([1,2,3,1,1,3]))
```

Output

```
4

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

Q21. City with Smallest Reachable Cities

Aim: Find city with fewest neighbors within distance threshold.

Algorithm:

Step-1: Initialize distance matrix (Floyd-Warshall).

Step-2: Update shortest paths.

Step-3: Count reachable cities for each.

Step-4: Return city with smallest count (largest index if tie).

Input & Output:

main.py

```python
def findTheCity(n, edges, threshold):
    INF = 10**9
    dist = [[INF]*n for _ in range(n)]
    for i in range(n): dist[i][i]=0
    for u,v,w in edges: dist[u][v]=dist[v][u]=w
    for k in range(n):
        for i in range(n):
            for j in range(n):
                if dist[i][k]+dist[k][j]<dist[i][j]:
                    dist[i][j]=dist[i][k]+dist[k][j]
    res, min_count = -1, n+1
    for i in range(n):
        count = sum(dist[i][j]<=threshold for j in range(n))-1
        if count<=min_count:
            min_count=count; res=i
    return res

print(findTheCity(4, [[0,1,3],[1,2,1],[1,3,4],[2,3,1]], 4))
```

Output

```
3

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

Q22. Network Delay Time

Aim: Find minimum time for all nodes to receive signal from node k.

Algorithm:

Step-1: Build adjacency list.

Step-2: Use Dijkstra's algorithm.

Step-3: Track max distance.

Step-4: Return max or -1 if unreachable.

Input & Output:

main.py Share Run

```python
import heapq
def networkDelayTime(times, n, k):
    g = [[] for _ in range(n+1)]
    for u,v,w in times: g[u].append((v,w))
    dist = [float('inf')]*(n+1)
    dist[k] = 0
    heap = [(0,k)]
    while heap:
        d,u = heapq.heappop(heap)
        if d>dist[u]: continue
        for v,w in g[u]:
            if d+w<dist[v]:
                dist[v]=d+w
                heapq.heappush(heap,(dist[v],v))
    ans = max(dist[1:])
    return ans if ans<float('inf') else -1

print(networkDelayTime([[2,1,1],[2,3,1],[3,4,1]], 4, 2))
```

Output

```
2

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