NAME : TEJASWINI M

REG NO. : 192411079

COURSE CODE : CSA0613

COURSE NAME : DESIGN AND ANALYSIS OF ALGORITHMS FOR OPTIMAL APPLICATIONS

SLOT : A

TOPIC 4 DYNAMIC PROGRAMMING

1. **You are given the number of sides on a die (num_sides), the number of dice to throw (num_dice), and a target sum (target). Develop a program that utilizes dynamic programming to solve the Dice Throw Problem.**

AIM:

To develop a dynamic programming program that calculates the number of ways to achieve a target sum using a given number of dice with a fixed number of sides.

ALGORITHM:

1. Read the number of sides, number of dice, and target sum.
2. Create a DP table where dp[i][j] represents the number of ways to get sum j using i dice.
3. Initialize dp[0][0] = 1.
4. For each die from 1 to num_dice:
    - For each possible sum from 1 to target:
        - For each face value from 1 to num_sides:
            - Update dp[i][j] += dp[i-1][j-face] if j-face ≥ 0.
5. The result will be stored in dp[num_dice][target].
6. Print the result.

CODE:

```
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 face in range(1, num_sides + 1):
                if j - face >= 0:
                    dp[i][j] += dp[i - 1][j - face]

    return dp[num_dice][target]
```

INPUT:

```
print("Test Case 1 Output:", dice_throw(6, 2, 7))
print("Test Case 2 Output:", dice_throw(4, 3, 10))
```

OUTPUT:

```
Test Case 1 Output: 6
Test Case 2 Output: 6
```

2. **In a factory, there are two assembly lines, each with n stations. Each station performs a specific task and takes a certain amount of time to complete. The task must go through each station in order, and there is also a transfer time for switching from one line to another. Given the time taken at each station on both lines and the transfer time between the lines, the goal is to find the minimum time required to process a product from start to end.**

AIM:

To find the minimum time required to process a product through two assembly lines using Dynamic Programming, considering station times, transfer times, entry times, and exit times.

ALGORITHM:

1. Read the number of stations n.
2. Initialize two arrays dp1 and dp2 to store the minimum time to reach each station on line 1 and line 2.
3. Set

   dp1[0] = e1 + a1[0]

   dp2[0] = e2 + a2[0]
4. For each station i from 1 to n-1:

   dp1[i] = min(dp1[i-1] + a1[i], dp2[i-1] + t2[i-1] + a1[i])

   dp2[i] = min(dp2[i-1] + a2[i], dp1[i-1] + t1[i-1] + a2[i])

5. Add exit times:

   result = min(dp1[n-1] + x1, dp2[n-1] + x2)

6. Print the result.

CODE:

```
def assembly_line_scheduling(a1, a2, t1, t2, e1, e2, x1, x2, n):
    dp1 = [0] * n
    dp2 = [0] * n

    dp1[0] = e1 + a1[0]
    dp2[0] = e2 + a2[0]

    for i in range(1, n):
        dp1[i] = min(dp1[i-1] + a1[i], dp2[i-1] + t2[i-1] + a1[i])
        dp2[i] = min(dp2[i-1] + a2[i], dp1[i-1] + t1[i-1] + a2[i])

    return min(dp1[n-1] + x1, dp2[n-1] + x2)
```

INPUT:

```
a1 = [4, 5, 3, 2]
a2 = [2, 10, 1, 4]
t1 = [7, 4, 5]
t2 = [9, 2, 8]
e1 = 10
e2 = 12
x1 = 18
x2 = 7
n = 4
```

OUTPUT:

```
Minimum Time: 35
```

3. **An automotive company has three assembly lines (Line 1, Line 2, Line 3) to produce different car models. Each line has a series of stations, and each station takes a certain amount of time to complete its task. Additionally, there are transfer times between lines, and certain dependencies must be respected due to the sequential nature of some tasks. Your goal is to minimize the total production time by determining the optimal scheduling of tasks across these lines, considering the transfer times and dependencies.**

AIM:

To minimize the total production time by optimally scheduling tasks across three assembly lines while considering station times, inter-line transfer times, and sequential dependencies using Dynamic Programming.

ALGORITHM:

1. Let dp[l][s] represent the minimum time to complete station s on line l.
2. Initialize the first station of each line directly from the given station times.
3. For each next station, compute:

   dp[l][s] = min(dp[k][s-1] + transfer[k][l]) + time[l][s]

   where k ranges over all lines.
4. The answer is the minimum value in the last station column.
5. This ensures dependencies are respected and transfer times are included.

CODE:

```
# DP table
dp = [[math.inf] * n_stations for _ in range(n_lines)]

# Initialization for first station
for l in range(n_lines):
    dp[l][0] = times[l][0]

# Fill DP table
for s in range(1, n_stations):
    for l in range(n_lines):
        dp[l][s] = min(dp[k][s-1] + transfer[k][l] for k in range(n_lines)) + times[l][s]

# Result
min_time = min(dp[l][n_stations-1] for l in range(n_lines))

print("DP Table:", dp)
print("Minimum Production Time:", min_time)
```

INPUT:

```
# Input data
times = [
    [5, 9, 3],  # Line 1
    [6, 8, 4],  # Line 2
    [7, 6, 5]   # Line 3
]
transfer = [
    [0, 2, 3],
    [2, 0, 4],
    [3, 4, 0]
]

n_lines = 3
n_stations = 3
```

OUTPUT:

DP Table: [[5, 14, 17], [6, 14, 18], [7, 13, 18]]

Minimum Production Time: 17

4. Write a c program to find the minimum path distance by using matrix form.

AIM:

To find the minimum path distance from a given distance matrix using a brute-force approach.

ALGORITHM:

1. Store the matrix.
2. Generate all permutations of cities except the starting city (0).
3. For each permutation, calculate total travel cost.
4. Add cost to return back to starting city.
5. Track the minimum cost.
6. Print the minimum path distance.

CODE:

```
import itertools
import sys
def calculate_cost(graph, path):
    cost = 0
    prev = 0  # start from city 0
    for city in path:
        cost += graph[prev][city]
        prev = city
    cost += graph[prev][0]  # return to start
    return cost
def find_min_path(graph):
    n = len(graph)
    cities = list(range(1, n))
    min_cost = sys.maxsize
    for perm in itertools.permutations(cities):
        cost = calculate_cost(graph, perm)
        if cost < min_cost:
            min_cost = cost
    return min_cost
```

INPUT:

```
# Test Case 1
graph1 = [
   [0,10,15,20],
   [10,0,35,25],
   [15,35,0,30],
   [20,25,30,0]
]
# Test Case 2
graph2 = [
   [0,10,10,10],
   [10,0,10,10],
   [10,10,0,10],
   [10,10,10,0]
]
# Test Case 3
graph3 = [
   [0,1,2,3],
   [1,0,4,5],
   [2,4,0,6],
   [3,5,6,0]
]
print("Test Case 1 Output:", find_min_path(graph1))
print("Test Case 2 Output:", find_min_path(graph2))
print("Test Case 3 Output:", find_min_path(graph3))
```

OUTPUT:

Test Case 1 Output: 80

Test Case 2 Output: 40

Test Case 3 Output: 14

5. Assume you are solving the Traveling Salesperson Problem for 4 cities (A, B, C, D) with known distances between each pair of cities. Now, you need to add a fifth city (E) to the problem.

AIM:

To find the shortest possible route for the Traveling Salesperson Problem (TSP) when a fifth city (E) is added, using a brute-force approach for symmetric distances.

ALGORITHM:

1. Store all cities in a list.
2. Store the symmetric distances between every pair of cities.
3. Fix the starting city as A (to avoid duplicate cyclic paths).
4. Generate all permutations of the remaining cities.
5. For each permutation:
    - Form a complete route starting and ending at A.
    - Calculate the total distance of the route.

6. Keep track of the route with the minimum total distance.

7. Output the shortest route and its total distance.

CODE:

```
def get_distance(a, b):
    if a == b:
        return 0
    if (a, b) in distances:
        return distances[(a, b)]
    return distances[(b, a)]

best_distance = float('inf')
best_route = None

for perm in itertools.permutations(cities[1:]):  # Fix A as start
    route = ['A'] + list(perm) + ['A']
    total_cost = 0

    for i in range(len(route) - 1):
        total_cost += get_distance(route[i], route[i+1])

    if total_cost < best_distance:
        best_distance = total_cost
        best_route = route

print("Shortest Route:", " -> ".join(best_route))
print("Minimum Distance:", best_distance)
```

INPUT:

```
cities = ['A', 'B', 'C', 'D', 'E']

distances = {
    ('A','B'):10, ('A','C'):15, ('A','D'):20, ('A','E'):25,
    ('B','C'):35, ('B','D'):25, ('B','E'):30,
    ('C','D'):30, ('C','E'):20,
    ('D','E'):15
}
```

OUTPUT:

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

6. Given a string s, return the longest palindromic substring in S.

AIM:

To find the longest palindromic substring in a given string using an efficient approach.

ALGORITHM:

1. If the string length is less than 2, return the string itself.

2. For each character in the string, consider it as the center of a palindrome.
3. Expand around the center for two cases:
    - Odd-length palindrome (single center)
    - Even-length palindrome (two centers)
4. Track the maximum length palindrome found.
5. Return the substring with the maximum length.

CODE:

```
def longestPalindrome(s):
    if len(s) < 2:
        return s

    start = 0
    max_len = 1

    def expandAroundCenter(left, right):
        nonlocal start, max_len
        while left >= 0 and right < len(s) and s[left] == s[right]:
            current_len = right - left + 1
            if current_len > max_len:
                start = left
                max_len = current_len
            left -= 1
            right += 1

    for i in range(len(s)):
        expandAroundCenter(i, i)     # Odd length
        expandAroundCenter(i, i+1)   # Even length

    return s[start:start + max_len]
```

INPUT:

```
# Test Cases
print(longestPalindrome("babad"))
print(longestPalindrome("cbbd"))
```

OUTPUT:

```
bab
bb
```

7. Given a string s, find the length of the longest substring without repeating characters.

AIM:

To find the length of the longest substring in a given string that contains no repeating characters.

ALGORITHM:

1. Use a sliding window technique with two pointers (left and right).
2. Maintain a set to store unique characters in the current window.
3. Move the right pointer and add characters to the set if they are not already present.
4. If a duplicate character is found, remove characters from the left until the duplicate is removed.
5. Keep track of the maximum window size at each step.
6. Return the maximum length obtained.

CODE:

```
def longestUniqueSubstring(s):
    char_set = set()
    left = 0
    max_length = 0

    for right in range(len(s)):
        while s[right] in char_set:
            char_set.remove(s[left])
            left += 1

        char_set.add(s[right])
        max_length = max(max_length, right - left + 1)

    return max_length
```

INPUT:

```
# Test Cases
print(longestUniqueSubstring("abcabcbb"))
print(longestUniqueSubstring("bbbbb"))
print(longestUniqueSubstring("pwwkew"))
```

OUTPUT:

```
3
1
3
```

8. **Given a string s and a dictionary of strings wordDict, return true if s can be segmented into a space-separated sequence of one or more dictionary words. Note that the same word in the dictionary may be reused multiple times in the segmentation.**

AIM:

To determine whether a given string can be segmented into a sequence of one or more valid dictionary words using dynamic programming.

ALGORITHM:

1. Let n be the length of the string s.
2. Create a boolean DP array dp of size n+1 where dp[i] is true if the substring s[0…i-1] can be segmented using the dictionary.
3. Initialize dp[0] = true (empty string can always be segmented).
4. For each index i from 1 to n:
    - Check all j from 0 to i-1.
    - If dp[j] is true and s[j:i] exists in the dictionary, then set dp[i] = true and break.
5. Return dp[n] as the final result.

CODE:

```
def wordBreak(s, wordDict):
    wordSet = 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 wordSet:
                dp[i] = True
                break

    return dp[n]
```

INPUT:

```
# Test Cases
print(wordBreak("leetcode", ["leet", "code"]))
print(wordBreak("applepenapple", ["apple", "pen"]))
print(wordBreak("catsandog", ["cats", "dog", "sand", "and", "cat"]))
```

OUTPUT:

```
True
True
False
```

9. **Given an input string and a dictionary of words, find out if the input string can be segmented into a space-separated sequence of dictionary words.Consider the following dictionary { i, like, sam, sung, samsung, mobile, ice, cream, icecream, man, go, mango}**

AIM:

To check whether a given input string can be segmented into a space-separated sequence of valid dictionary words.

ALGORITHM:

1. Store all dictionary words in a set for fast lookup.
2. Create a boolean DP array dp of size n+1, where n is the length of the string.
3. Set dp[0] = true (empty string can be segmented).
4. For each position i from 1 to n:
    - For each position j from 0 to i-1:
        - If dp[j] is true and substring s[j:i] exists in the dictionary, then set dp[i] = true and break.
5. If dp[n] is true, the string can be segmented. Otherwise, it cannot.

CODE:

```
def wordBreak(s, wordDict):
    wordSet = 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 wordSet:
                dp[i] = True
                break

    return dp[n]
```

INPUT:

```
# Dictionary
dictionary = ["i", "like", "sam", "sung", "samsung", "mobile",
              "ice", "cream", "icecream", "man", "go", "mango"]


# Test Cases
print(wordBreak("ilike", dictionary))
print(wordBreak("ilikesamsung", dictionary))
```

OUTPUT:

```
True
True
```

10. Given an array of strings words and a width maxWidth, format the text such that each line has exactly maxWidth characters and is fully (left and right) justified. You should pack your words in a greedy approach; that is, pack as many words as you can in each line. Pad extra spaces ' ' when necessary so that each line has exactly maxWidth characters. Extra spaces between words should be distributed as evenly as possible. If the number of spaces on a line does not divide evenly between words, the empty slots on the left will be assigned more spaces than the slots on the right. For the last line of text, it should be left-justified, and no extra space is inserted between words. A word is defined as a character sequence consisting of non-space characters only. Each word's length is guaranteed to be greater than 0 and not exceed maxWidth. The input array words contains at least one word.

AIM:

To format a given array of words into fully justified text with each line exactly maxWidth characters, distributing spaces evenly using a greedy approach.

ALGORITHM:

1. Initialize an empty list res to store the justified lines.

2. Use a pointer to iterate through the words and pack as many words as possible into each line without exceeding maxWidth.
3. For each line:
    - Calculate total spaces needed: maxWidth - sum(length of words in line).
    - Distribute spaces evenly between words. If extra spaces remain, assign more to the left slots.
    - For the last line or lines with a single word, left-justify by adding spaces to the right.
4. Append the justified line to res and continue with the next set of words.
5. Return the list res containing all justified lines.

CODE:

```
def fullJustify(words, maxWidth):
    res = []
    n = len(words)
    i = 0
    while i < n:
        # Determine the words in the current line
        line_len = len(words[i])
        j = i + 1
        while j < n and line_len + 1 + len(words[j]) <= maxWidth:
            line_len += 1 + len(words[j])
            j += 1
        line_words = words[i:j]
        num_words = j - i
        total_chars = sum(len(word) for word in line_words)
        spaces_needed = maxWidth - total_chars
        if j == n or num_words == 1:  # last line or single word
            line = ' '.join(line_words)
            line += ' ' * (maxWidth - len(line))
        else:
            spaces_between = spaces_needed // (num_words - 1)
            extra_spaces = spaces_needed % (num_words - 1)
            line = ''
            for k in range(num_words - 1):
                line += line_words[k]
                line += ' ' * (spaces_between + (1 if k < extra_spaces else 0))
            line += line_words[-1]
        res.append(line)
        i = j
    return res
```

INPUT:

```
# Test Case
words = ["This","is","an","example","of","text","justification."]
maxWidth = 16
justified_text = fullJustify(words, maxWidth)
for line in justified_text:
    print(f'"{line}"')
```

OUTPUT:

```
"This   is   an"
"example  of text"
"justification.  "
```

11. Design a special dictionary that searches the words in it by a prefix and a suffix. Implement the WordFilter class: WordFilter(string[] words) Initializes the object with the words in the dictionary.f(string pref, string suff) Returns the index of the word in the dictionary, which has the prefix pref and the suffix suff. If there is more than one valid index, return the largest of them. If there is no such word in the dictionary, return -1.

AIM:

To design a special dictionary that allows efficient searching of words by both prefix and suffix. Implement a class WordFilter which supports initialization with a list of words and a function f(pref, suff) to return the index of the word that matches the given prefix and suffix.

ALGORITHM:

1. Initialize the WordFilter class with a list of words.
2. Use a hash map (dictionary) to store all possible combinations of (prefix, suffix) for each word along with its index.
3. For each word at index i:
    - Generate all prefixes of the word.
    - Generate all suffixes of the word.
    - Store (prefix, suffix) → i in the dictionary. If duplicates occur, overwrite to keep the largest index.

4. Implement the function f(pref, suff) which checks if (pref, suff) exists in the dictionary.
    - If yes, return the corresponding index.
    - If no, return -1.

CODE:

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

    def f(self, pref, suff):
        return self.pref_suff_map.get((pref, suff), -1)
```

INPUT:

```
# Example Usage
wordFilter = WordFilter(["apple"])
print(wordFilter.f("a", "e"))  # Output: 0
```

OUTPUT:

```
0
```

12. Implement Floyd's Algorithm to find the shortest path between all pairs of cities. Display the distance matrix before and after applying the algorithm. Identify and print the shortest path

AIM:

To implement Floyd's Algorithm (Floyd-Warshall) to find the shortest paths between all pairs of cities (or nodes) in a weighted graph. Display the distance matrix before and after applying the algorithm and identify the shortest path between given cities.

ALGORITHM:

1. Initialize a 2D distance matrix dist where dist[i][j] represents the distance from city i to city j.
    - If there is no direct edge, set distance to infinity.
    - Set dist[i][i] = 0 for all i.
2. Update the matrix with given edges (direct distances).
3. Apply Floyd-Warshall Algorithm:
    - For each intermediate vertex k:
        - For each pair of vertices (i, j):
            - Update dist[i][j] = min(dist[i][j], dist[i][k] + dist[k][j])
4. After processing all vertices, dist[i][j] contains the shortest distance between city i and city j.
5. To find the shortest path between specific cities, refer to dist[source][destination].

CODE:

```
INF = float('inf')
def floyd_warshall(n, edges):
    # Initialize distance matrix
    dist = [[INF] * n for _ in range(n)]
    for i in range(n):
        dist[i][i] = 0
    # Fill in direct edges
    for u, v, w in edges:
        dist[u][v] = w
    print("Distance matrix before Floyd-Warshall:")
    for row in dist:
        print(row)
    # Floyd-Warshall algorithm
    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-Warshall:")
    for row in dist:
        print(row)
    return dist
```

INPUT:

```
# Example Input (Test Case a)
n = 4
edges = [
  [0,1,3],
  [1,2,1],
  [1,3,4],
  [2,3,1]
]

dist_matrix = floyd_warshall(n, edges)

# Shortest path from City 0 to City 3
print(f"\nShortest distance from City 0 to City 3 = {dist_matrix[0][3]}")
```

OUTPUT:

```
Distance matrix before Floyd-Warshall:
[0, 3, inf, inf]
[inf, 0, 1, 4]
[inf, inf, 0, 1]
[inf, inf, inf, 0]

Distance matrix after Floyd-Warshall:
[0, 3, 4, 5]
[inf, 0, 1, 2]
[inf, inf, 0, 1]
[inf, inf, inf, 0]

Shortest distance from City 0 to City 3 = 5
```

13. Write a Program to implement Floyd's Algorithm to calculate the shortest paths between all pairs of routers. Simulate a change where the link between Router B and Router D fails. Update the distance matrix accordingly. Display the shortest path from Router A to Router F before and after the link failure.

AIM:

To implement Floyd's Algorithm to compute the shortest paths between all pairs of routers, simulate a link failure, update the distance matrix, and display the shortest path from Router A to Router F before and after the failure.

ALGORITHM:

1. Initialize a distance matrix dist for all routers:
   - dist[i][i] = 0
   - dist[i][j] = weight of edge between i and j if it exists, otherwise infinity.
2. Apply Floyd-Warshall Algorithm:
   - For each intermediate router k, update:
     dist[i][j] = min(dist[i][j], dist[i][k] + dist[k][j])
3. Save dist for before link failure.

4. Simulate link failure by setting the failed link's distance to infinity:
    - For example, B-D fails → dist[B][D] = dist[D][B] = INF
5. Re-run Floyd-Warshall to recompute distances after the link failure.
6. Display the shortest path from Router A to Router F both before and after the failure.

CODE:

```
INF = float('inf')

def floyd_warshall(n, 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

# Copy for before failure
import copy
dist_before = copy.deepcopy(dist)
dist_before = floyd_warshall(n, dist_before)

print(f"Shortest distance from Router A to Router F before failure = {dist_before[0][5]}")

# Simulate link failure B-D
dist[1][3] = INF
dist[3][1] = INF

# Recompute shortest paths after failure
dist_after = floyd_warshall(n, dist)
print(f"Shortest distance from Router A to Router F after B-D failure = {dist_after[0][5]}")
```

INPUT:

```
# Routers A=0, B=1, C=2, D=3, E=4, F=5
n = 6
# Initial distances
dist = [
    [0, 1, 5, INF, INF, INF], # A
    [1, 0, 2, 1, INF, INF],   # B
    [5, 2, 0, INF, 3, INF],   # C
    [INF, 1, INF, 0, 1, 6],   # D
    [INF, INF, 3, 1, 0, 2],   # E
    [INF, INF, INF, 6, 2, 0]  # F
]
```

OUTPUT:

```
Shortest distance from Router A to Router F before failure = 5
Shortest distance from Router A to Router F after B-D failure = 8
```

14. Implement Floyd's Algorithm to find the shortest path between all pairs of cities. Display the distance matrix before and after applying the algorithm. Identify and print the shortest path

AIM:

To implement Floyd's Algorithm to find the shortest paths between all pairs of cities, display the distance matrix before and after the algorithm, and identify specific shortest paths between selected cities.

ALGORITHM:

1. Initialize a distance matrix dist with:
    - dist[i][i] = 0
    - dist[i][j] = weight of edge between i and j if it exists, otherwise infinity.
2. Apply Floyd-Warshall Algorithm:
    - For each intermediate city k, update:
      dist[i][j] = min(dist[i][j], dist[i][k] + dist[k][j])
3. After computation, display the distance matrix.
4. Count neighbors for each city within the distanceThreshold.
5. Identify and print the shortest path between the requested city pairs.

CODE:

```
INF = float('inf')
def floyd_warshall(n, dist):
    # Create a copy to store predecessors for path reconstruction
    pred = [[-1 if dist[i][j]==INF else i for j in range(n)] for i in range(n)]
    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]
                    pred[i][j] = pred[k][j]
    return dist, pred
def reconstruct_path(pred, start, end):
    path = []
    if pred[start][end] == -1:
        return path
    at = end
    while at != start:
        path.append(at)
        at = pred[start][at]
    path.append(start)
    path.reverse()
    return path
```

```
# Initialize distance matrix
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  # assuming undirected graph

print("Distance matrix before Floyd-Warshall:")
for row in dist:
    print(row)

dist_after, pred = floyd_warshall(n, dist)

print("\nDistance matrix after Floyd-Warshall:")
for row in dist_after:
    print(row)

# Count neighbors within distanceThreshold
for city in range(n):
    neighbors = [i for i in range(n) if i!=city and dist_after[city][i] <= distanceThreshold]
    print(f"City {city} -> {neighbors}")

# Find city with minimum neighbors within threshold
min_neighbors = min(len([i for i in range(n) if i!=city and dist_after[city][i] <= distanceThreshold]) for city in range(n))
city_with_min = [city for city in range(n) if len([i for i in range(n) if i!=city and dist_after[city][i] <= distanceThreshold]) == min_neighbors]
print(f"City with minimum neighbors at threshold {distanceThreshold}: {city_with_min[0]}")

# Shortest path examples:
# a) C to A (2->0)
start, end = 2, 0
path = reconstruct_path(pred, start, end)
print(f"Shortest path from C to A: {path}, Distance = {dist_after[start][end]}")
```

INPUT:

```
# Test Case 1: n=5, edges as given
n = 5
edges = [[0,1,2],[0,4,8],[1,2,3],[1,4,2],[2,3,1],[3,4,1]]
distanceThreshold = 2
```

OUTPUT:

```
Distance matrix before Floyd-Warshall:
[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 Floyd-Warshall:
[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]
City 0 -> [1]
City 1 -> [0, 4]
City 2 -> [3, 4]
City 3 -> [2, 4]
City 4 -> [1, 2, 3]
City with minimum neighbors at threshold 2: 0
Shortest path from C to A: [2, 1, 0], Distance = 5
Shortest path from E to C: [4, 3, 2], Distance = 2
```

15. Implement the Optimal Binary Search Tree algorithm for the keys A,B,C,D with frequencies 0.1,0.2,0.4,0.3 Write the code using any programming language to construct the OBST for the given keys and frequencies. Execute your code and display the resulting OBST and its cost. Print the cost and root matrix.

AIM:

To implement the Optimal Binary Search Tree (OBST) algorithm for a given set of keys and their frequencies, construct the OBST, and calculate its minimum cost.

ALGORITHM:

1. Let n be the number of keys. Define cost[i][j] as the cost of the optimal BST that contains keys from i to j.
2. Initialize cost[i][i-1] = 0 for all i (empty subtrees).
3. Define freq_sum(i, j) as the sum of frequencies from key i to key j.
4. For lengths l = 1 to n, compute cost[i][j] as:
5. cost[i][j] = min (cost[i][r-1] + cost[r+1][j] + freq_sum(i,j)) for all r from i to j

Keep track of root[i][j] = r which gives the root key of the subtree.

6. The minimum cost of the OBST is cost[1][n].

CODE:

```
def optimal_bst(keys, freq):
    n = len(keys)
    cost = [[0 for _ in range(n+2)] for _ in range(n+2)]
    root = [[0 for _ in range(n+1)] for _ in range(n+1)]
    # sum of frequencies from i to j
    def freq_sum(i, j):
        return sum(freq[i-1:j])
    for i in range(1, n+2):
        cost[i][i-1] = 0
    for l in range(1, n+1):  # length of chain
        for i in range(1, n-l+2):
            j = i + l - 1
            cost[i][j] = float('inf')
            for r in range(i, j+1):
                c = cost[i][r-1] + cost[r+1][j] + freq_sum(i,j)
                if c < cost[i][j]:
                    cost[i][j] = c
                    root[i][j] = r
    print("Minimum cost of OBST:", cost[1][n])
    print("\nCost Table:")
    for i in range(1, n+1):
        print(cost[i][1:n+1])
    print("\nRoot Table:")
    for i in range(1, n+1):
        print(root[i][1:n+1])
    return cost, root
```

INPUT:

```
# Test Case 1
keys = ['A','B','C','D']
freq = [0.1,0.2,0.4,0.3]
optimal_bst(keys, freq)

# Test Case 2
keys2 = [10,12]
freq2 = [34,50]
optimal_bst(keys2, freq2)

# Test Case 3
keys3 = [10,12,20]
freq3 = [34,8,50]
optimal_bst(keys3, freq3)
```

OUTPUT:

```
Minimum cost of OBST: 1.7

Cost Table:
[0.1, 0.4, 1.1, 1.7]
[0, 0.2, 0.8, 1.4000000000000001]
[0, 0, 0.4, 1.0]
[0, 0, 0, 0.3]

Root Table:
[1, 2, 3, 3]
[0, 2, 3, 3]
[0, 0, 3, 3]
[0, 0, 0, 4]
Minimum cost of OBST: 118

Cost Table:
[34, 118]
[0, 50]

Root Table:
[1, 2]
[0, 2]
Minimum cost of OBST: 142

Cost Table:
[34, 50, 142]
[0, 8, 66]
[0, 0, 50]

Root Table:
[1, 1, 3]
[0, 2, 3]
[0, 0, 3]
```

16. **Consider a set of keys 10,12,16,21 with frequencies 4,2,6,3 and the respective probabilities. Write a Program to construct an OBST in a programming language of your choice. Execute your code and display the resulting OBST, its cost and root matrix.**

AIM:

To construct an Optimal Binary Search Tree (OBST) for a given set of keys and their frequencies, and to compute the minimum search cost, cost matrix, and root matrix.

ALGORITHM:

1. Let n be the number of keys. Define cost[i][j] as the cost of the optimal BST for keys from i to j.
2. Initialize cost[i][i-1] = 0 for all i (empty subtrees).
3. Define freq_sum(i, j) as the sum of frequencies from key i to key j.
4. For lengths l = 1 to n, compute:
5. cost[i][j] = min (cost[i][r-1] + cost[r+1][j] + freq_sum(i,j)) for r in [i, j]. Store the root r in root[i][j].
6. The minimum cost is cost[1][n].

CODE:

```
def optimal_bst(keys, freq):
    n = len(keys)
    cost = [[0 for _ in range(n+2)] for _ in range(n+2)]
    root = [[0 for _ in range(n+1)] for _ in range(n+1)]
    def freq_sum(i, j):
        return sum(freq[i-1:j])
    for i in range(1, n+2):
        cost[i][i-1] = 0
    for l in range(1, n+1):  # length of chain
        for i in range(1, n-l+2):
            j = i + l - 1
            cost[i][j] = float('inf')
            for r in range(i, j+1):
                c = cost[i][r-1] + cost[r+1][j] + freq_sum(i,j)
                if c < cost[i][j]:
                    cost[i][j] = c
                    root[i][j] = r
    print("Minimum cost of OBST:", cost[1][n])
    print("\nCost Table:")
    for i in range(1, n+1):
        print(cost[i][1:n+1])
    print("\nRoot Table:")
    for i in range(1, n+1):
        print(root[i][1:n+1])
    return cost, root
```

INPUT:

```
# Test Case 1
keys = [10, 12, 16, 21]
freq = [4, 2, 6, 3]
optimal_bst(keys, freq)

# Test Case 2
keys2 = [10, 12]
freq2 = [34, 50]
optimal_bst(keys2, freq2)

# Test Case 3
keys3 = [10, 12, 20]
freq3 = [34, 8, 50]
optimal_bst(keys3, freq3)
```

OUTPUT:

```
Minimum cost of OBST: 26

Cost Table:
[4, 8, 20, 26]
[0, 2, 10, 16]
[0, 0, 6, 12]
[0, 0, 0, 3]

Root Table:
[1, 1, 3, 3]
[0, 2, 3, 3]
[0, 0, 3, 3]
[0, 0, 0, 4]
Minimum cost of OBST: 118

Cost Table:
[34, 118]
[0, 50]

Root Table:
[1, 2]
[0, 2]
Minimum cost of OBST: 142

Cost Table:
[34, 50, 142]
[0, 8, 66]
[0, 0, 50]

Root Table:
[1, 1, 3]
[0, 2, 3]
[0, 0, 3]
```

17. **A game on an undirected graph is played by two players, Mouse and Cat, who alternate turns. The graph is given as follows: graph[a] is a list of all nodes b such that ab is an edge of the graph. The mouse starts at node 1 and goes first, the cat starts at node 2 and goes second, and there is a hole at node 0. During each player's turn, they must travel along one edge of the graph that meets where they are. For example, if the Mouse is at node 1, it must travel to any node in graph[1]. Additionally, it is not allowed for the Cat to travel to the Hole (node 0).Then, the game can end in three ways:**

AIM:

To determine the outcome of the Cat and Mouse game on an undirected graph, assuming both players play optimally.

ALGORITHM:

1. Model the game as states (mouse_pos, cat_pos, turn).
2. Use **Breadth-First Search (BFS)** or **Dynamic Programming** to propagate known results:
    - If mouse is at hole (node 0), mouse wins.
    - If cat and mouse are at the same node, cat wins.
    - If a state repeats, it is a draw.
3. Initialize all terminal states (mouse at hole → mouse wins, cat catches mouse → cat wins).
4. Propagate the results backward using **topological order** of game states.
5. Return the outcome for the starting state (mouse=1, cat=2, turn=1).

CODE:

```
from collections import deque
def catMouseGame(graph):
    n = len(graph)
    DRAW, MOUSE, CAT = 0, 1, 2
    # Initialize degrees
    degree = [[[0, 0] for _ in range(n)] for _ in range(n)]
    for m in range(n):
        for c in range(n):
            degree[m][c][0] = len(graph[m])  # mouse turn
            degree[m][c][1] = len(graph[c]) - (0 in graph[c])  # cat turn can't go to hole
    # Result table: 0=draw, 1=mouse win, 2=cat win
    result = [[[DRAW, DRAW] for _ in range(n)] for _ in range(n)]
    queue = deque()
```

```
    # Terminal states
    for i in range(n):
        for t in [0,1]:
            result[0][i][t] = MOUSE
            queue.append((0, i, t, MOUSE))
            if i != 0:
                result[i][i][t] = CAT
                queue.append((i, i, t, CAT))
    # BFS propagation
    while queue:
        mouse, cat, turn, res = queue.popleft()
        prev_turn = 1 - turn
        if turn == 0:  # mouse just moved, so cat's turn previously
            prev_positions = [(m, cat, prev_turn) for m in graph[mouse]]
        else:  # cat just moved
            prev_positions = [(mouse, c, prev_turn) for c in graph[cat] if c != 0]
        for m, c, t in prev_positions:
            if result[m][c][t] != DRAW:
                continue
            # If the current player can win, set result
            if res == (MOUSE if t == 0 else CAT):
                result[m][c][t] = res
                queue.append((m, c, t, res))
            else:
                degree[m][c][t] -= 1
                if degree[m][c][t] == 0:
                    # All moves lead to opponent win, so opponent wins
                    result[m][c][t] = CAT if t == 0 else MOUSE
                    queue.append((m, c, t, result[m][c][t]))
    return result[1][2][0]
```

INPUT:

```
# Test Cases
graph1 = [[2,5],[3],[0,4,5],[1,4,5],[2,3],[0,2,3]]
graph2 = [[1,3],[0],[3],[0,2]]
print(catMouseGame(graph1))  # Output: 0
print(catMouseGame(graph2))  # Output: 1
```

OUTPUT:

```
0

1
```

18. **You are given an undirected weighted graph of n nodes (0-indexed), represented by an edge list where edges[i] = [a, b] is an undirected edge connecting the nodes a and b with a probability of success of traversing that edge succProb[i]. Given two nodes start and end, find the path with the maximum probability of success to go from start to end and return its success probability. If there is no path from start to end, return 0. Your answer will be accepted if it differs from the correct answer by at most 1e-5.**

AIM:

To find the path with the **maximum probability of success** in an undirected weighted graph from a start node to an end node.

ALGORITHM:

1. Model the graph using an adjacency list where each edge has a success probability.
2. Use a **modified Dijkstra's algorithm** with a **max-heap** (priority queue) to maximize probability:
    - Instead of minimizing distance, keep track of maximum probability to reach each node.
3. Initialize the probability of reaching the start node as 1 and all others as 0.
4. Pop the node with the highest probability from the heap, update its neighbors:
    - For neighbor v of u, prob[v] = max(prob[v], prob[u] * edge_prob[u][v])
    - Push neighbor v into heap if probability improves.
5. Continue until heap is empty or end node is reached.
6. Return the probability of reaching the end node.

CODE:

```
import heapq
def maxProbability(n, edges, succProb, start, end):
    # Build graph
    graph = [[] for _ in range(n)]
    for (u, v), p in zip(edges, succProb):
        graph[u].append((v, p))
        graph[v].append((u, p))
    # Max heap with negative probabilities
    heap = [(-1.0, start)]
    prob = [0.0] * n
    prob[start] = 1.0
    while heap:
        p, node = heapq.heappop(heap)
        p = -p
        if node == end:
            return p
        if p < prob[node]:
            continue
        for nei, edge_prob in graph[node]:
            new_prob = p * edge_prob
            if new_prob > prob[nei]:
                prob[nei] = new_prob
                heapq.heappush(heap, (-new_prob, nei))
    return 0.0
```

INPUT:

 # Test Cases

n1 = 3

edges1 = [[0,1],[1,2],[0,2]]s

succProb1 = [0.5,0.5,0.2]

start1, end1 = 0, 2

n2 = 3

edges2 = [[0,1],[1,2],[0,2]]

succProb2 = [0.5,0.5,0.3]

start2. end2 = 0. 2

OUTPUT:

0.25

0.3

19. **There is a robot on an m x n grid. The robot is initially located at the top-left corner (i.e., grid[0][0]). The robot tries to move to the bottom-right corner (i.e., grid[m - 1][n - 1]). The robot can only move either down or right at any point in time. Given the two integers m and n, return the number of possible unique paths that the robot can take to reach the bottom-right corner. The test cases are generated so that the answer will be less than or equal to 2 * 10 9.**

AIM:

To find the number of **unique paths** a robot can take from the top-left corner to the bottom-right corner of an m x n grid, moving only **right** or **down**.

ALGORITHM:

1. Use **dynamic programming (DP)** to store the number of ways to reach each cell.
2. Let dp[i][j] represent the number of unique paths to reach cell (i, j).
3. Base cases:
    - dp[0][0] = 1 (starting cell)
    - First row dp[0][j] = 1 (can only move right)
    - First column dp[i][0] = 1 (can only move down)

4. For other cells:
    - dp[i][j] = dp[i-1][j] + dp[i][j-1]
    - (Paths from top + paths from left)
5. Return dp[m-1][n-1] as the total number of unique paths.

CODE:

```
def uniquePaths(m, n):
    # Initialize dp array
    dp = [[1]*n for _ in range(m)]

    # Fill dp array
    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[m-1][n-1]
```

INPUT:

```
# Test Cases
print(uniquePaths(3, 7))
print(uniquePaths(3, 2))
```

OUTPUT:

```
28
3
```

20. Given an array of integers nums, return the number of good pairs. A pair (i, j) is called good if nums[i] == nums[j] and i < j.

AIM:

Given an array of integers nums, return the number of good pairs. A pair (i, j) is called good if nums[i] == nums[j] and i < j.

ALGORITHM:

1. Initialize a **counter** to 0.
2. Use a **dictionary** (hashmap) to store the frequency of each number encountered so far.

3. For each number in the array:
    - If it has been seen before freq[num] times, it forms freq[num] new good pairs with previous occurrences.
    - Increment freq[num] by 1.
4. Return the total counter value.

CODE:

```
def numGoodPairs(nums):
    from collections import defaultdict
    freq = defaultdict(int)
    count = 0

    for num in nums:
        count += freq[num]
        freq[num] += 1

    return count
```

INPUT:

```
# Test Cases
print(numGoodPairs([1,2,3,1,1,3]))
print(numGoodPairs([1,1,1,1]))
```

OUTPUT:

```
4
6
```

21. **There are n cities numbered from 0 to n-1. Given the array edges where edges[i] = [fromi, toi, weighti] represents a bidirectional and weighted edge between cities fromi and toi, and given the integer distanceThreshold. Return the city with the smallest number of cities that are reachable through some path and whose distance is at most distanceThreshold, If there are multiple such cities, return the city with the greatest number. Notice that the distance of a path connecting cities i and j is equal to the sum of the edges' weights along that path.**

AIM:

To find the city with the **smallest number of reachable cities** within a given distance threshold. If multiple cities have the same count, return the **city with the greatest number**.

ALGORITHM:

1. Initialize a **distance matrix** dist of size n x n and set initial distances to infinity (INF) except dist[i][i] = 0.
2. Update the distance matrix with the **direct edge weights** from the input edges.
3. Apply **Floyd-Warshall algorithm** to compute shortest paths between all pairs of cities:
    - For each intermediate city k, update dist[i][j] = min(dist[i][j], dist[i][k] + dist[k][j]).
4. For each city i, count the number of cities j such that dist[i][j] <= distanceThreshold and i != j.
5. Track the city with the **minimum count**. If there's a tie, choose the city with the **largest index**.

CODE:

```
def findTheCity(n, edges, distanceThreshold):
    INF = float('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
    # Floyd-Warshall
    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]
    min_count = n
    city_ans = 0
    for i in range(n):
        count = sum(1 for j in range(n) if i != j and dist[i][j] <= distanceThreshold)
        if count <= min_count:
            min_count = count
            city_ans = i  # choose greatest index in case of tie
    return city_ans
```

INPUT:

```
# Test Cases
print(findTheCity(4, [[0,1,3],[1,2,1],[1,3,4],[2,3,1]], 4))
print(findTheCity(5, [[0,1,2],[0,4,8],[1,2,3],[1,4,2],[2,3,1],[3,4,1]], 2))
```

OUTPUT:

```
3
0
```

22. You are given a network of n nodes, labeled from 1 to n. You are also given times, a list of travel times as directed edges times[i] = (ui, vi, wi), where ui is the source node, vi is the target node, and wi is the time it takes for a signal to travel from source to target. We will send a signal from a given node k. Return the minimum time it takes for all the n nodes to receive the signal. If it is impossible for all the n nodes to receive the signal, return -1.

AIM:

To determine the **minimum time** it takes for a signal to reach all nodes in a directed weighted network starting from a given node. If some nodes are unreachable, return -1.

ALGORITHM:

1. Model the network as a **graph** using an adjacency list.
2. Use **Dijkstra's algorithm** (shortest path from source) to find the minimum travel time from the starting node k to all other nodes.
3. Track the **maximum distance** among all reachable nodes.
4. If any node is unreachable (distance = ∞), return -1. Otherwise, return the maximum distance.

CODE:

```
import heapq
def networkDelayTime(times, n, k):
    graph = {i: [] for i in range(1, n+1)}
    for u, v, w in times:
        graph[u].append((v, w))
    # Min-heap for Dijkstra
    heap = [(0, k)]  # (time, node)
    dist = {}
    while heap:
        time, node = heapq.heappop(heap)
        if node in dist:
            continue
        dist[node] = time
        for nei, wt in graph[node]:
            if nei not in dist:
                heapq.heappush(heap, (time + wt, nei))
    if len(dist) != n:
        return -1
    return max(dist.values())
```

INPUT:

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

OUTPUT:

```
2
1
-1
```