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.

Test Cases:

1.Simple Case:

•Number of sides: 6

•Number of dice: 2

•Target sum: 7

2.More Complex Case:

•Number of sides: 4

•Number of dice: 3

•Target sum: 10

Output

Test Case 1:

Number of ways to reach sum 7: 6

Test Case 2:

Number of ways to reach sum 10: 27

CODE:

```
def count_ways_to_sum(num_dice, num_sides, 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, num_sides + 1):
            for k in range(j, target + 1):
                dp[i][k] += dp[i - 1][k - j]
    return dp[num_dice][target]
# Test Cases
num_sides_1, num_dice_1, target_1 = 6, 2, 7
num_sides_2, num_dice_2, target_2 = 4, 3, 10
ways_to_sum_1 = count_ways_to_sum(num_dice_1, num_sides_1, target_1)
ways_to_sum_2 = count_ways_to_sum(num_dice_2, num_sides_2, target_2)
print(f"Number of ways to reach sum {target_1}: {ways_to_sum_1}")
print(f"Number of ways to reach sum {target_2}: {ways_to_sum_2}")
```

OUTPUT:

Number of ways to reach sum 10: 6

Number of ways to reach sum 7: 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.

Input
n: Number of stations on each line.
a1[i]: Time taken at station i on assembly line 1.
a2[i]: Time taken at station i on assembly line 2.
t1[i]: Transfer time from assembly line 1 to assembly line 2 after station i.
t2[i]: Transfer time from assembly line 2 to assembly line 1 after station i.
e1: Entry time to assembly line 1.
e2: Entry time to assembly line 2.
x1: Exit time from assembly line 1.
x2: Exit time from assembly line 2.
Output
The minimum time required to process the product.

CODE:

```
def assembly_line_scheduling(n, a1, a2, t1, t2, e1, e2, x1, x2):
    f1 = [0] * n
    f2 = [0] * n
    f1[0] = e1 + a1[0]
    f2[0] = e2 + a2[0]
    for i in range(1, n):
        f1[i] = min(f1[i - 1] + a1[i], f2[i - 1] + t2[i - 1] + a1[i])
        f2[i] = min(f2[i - 1] + a2[i], f1[i - 1] + t1[i - 1] + a2[i])
    return min(f1[n - 1] + x1, f2[n - 1] + x2)
# Example input values
n = 4
a1 = [7, 9, 3, 4]
a2 = [8, 5, 6, 4]
t1 = [2, 3, 1]
t2 = [2, 1, 2]
e1 = 2
e2 = 4
x1 = 3
x2 = 2
print(assembly_line_scheduling(n, a1, a2, t1, t2, e1, e2, x1, x2))
```

OUTPUT: 27

3An 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.

Number of stations: 3

- **Station times:**
- **Line 1: [5, 9, 3]**
- **Line 2: [6, 8, 4]**
- **Line 3: [7, 6, 5]**

- **Transfer times:**

[
[0, 2, 3],
[2, 0, 4],
[3, 4, 0]
]
Dependencies: [(0, 1), (1, 2)] (i.e., the output of the first station is needed for the second, and the second for the third, regardless of the line).

CODE:

```
import numpy as np

def min_production_time(station_times, transfer_times, dependencies):
    n = len(station_times[0])
    f1 = [0] * n
    f2 = [0] * n
    f3 = [0] * n

    f1[0] = station_times[0][0]
    f2[0] = station_times[1][0]
    f3[0] = station_times[2][0]

    for i in range(1, n):
        f1[i] = min(f1[i - 1] + station_times[0][i], f2[i - 1] + transfer_times[1][0] + station_times[0][i], f3[i - 1] +
transfer_times[2][0] + station_times[0][i])
        f2[i] = min(f2[i - 1] + station_times[1][i], f1[i - 1] + transfer_times[0][1] + station_times[1][i], f3[i - 1] +
transfer_times[2][1] + station_times[1][i])
        f3[i] = min(f3[i - 1] + station_times[2][i], f1[i - 1] + transfer_times[0][2] + station_times[2][i], f2[i - 1] +
transfer_times[1][2] + station_times[2][i])

    return min(f1[n - 1], f2[n - 1], f3[n - 1])

station_times = [
    [5, 9, 3],
    [6, 8, 4],
    [7, 6, 5]
]

transfer_times = [
    [0, 2, 3],
    [2, 0, 4],
    [3, 4, 0]
]

dependencies = [(0, 1), (1, 2)]

min_time = min_production_time(station_times, transfer_times, dependencies)
print("Minimum Production Time:", min_time)
```

OUTPUT:

Minimum Production Time: 17

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

Test Cases:

1)
{0,10,15,20}
{10,0,35,25}
{15,35,0,30}

{20,25,30,0}
Output: 80

2)
{0,10,10,10}
{10,0,10,10}
{10,10,0,10}
{10,10,10,0}
Output: 40

3)
{0,1,2,3}
{1,0,4,5}
{2,4,0,6}
{3,5,6,0}
Output: 12

CODE:

```
#include <stdio.h>
#define INF 9999
#define V 4
int min(int a, int b) {
   return (a < b) ? a : b;
}
int findMinPathDistance(int graph[V][V]) {
   int dp[1 << V][V];
   for (int i = 0; i < (1 << V); i++) {
      for (int j = 0; j < V; j++) {
         dp[i][j] = INF;
      }
   }
   dp[1][0] = 0;
   for (int mask = 1; mask < (1 << V); mask += 2) {
      for (int i = 1; i < V; i++) {
         if ((mask & (1 << i)) != 0) {
            for (int j = 0; j < V; j++) {
               if ((mask & (1 << j)) != 0) {
                  dp[mask][i] = min(dp[mask][i], dp[mask ^ (1 << i)][j] + graph[j][i]);
               }
            }
         }
      }
   }

   int res = INF;
   for (int i = 1; i < V; i++) {
      res = min(res, dp[(1 << V) - 1][i] + graph[i][0]);
   }
   return res;
}
int main() {
   int graph1[V][V] = {{0, 10, 15, 20},
                       {10, 0, 35, 25},
                       {15, 35, 0, 30},
                       {20, 25, 30, 0}};
   int graph2[V][V] = {{0, 10, 10, 10},
                       {10, 0, 10, 10},
                       {10, 10, 0, 10},
                       {10, 10, 10, 0}};
```

```
    int graph3[V][V] = {{0, 1, 2, 3},
                        {1, 0, 4, 5},
                        {2, 4, 0, 6},
                        {3, 5, 6, 0}};

    printf("Output 1: %d\n", findMinPathDistance(graph1));
    printf("Output 2: %d\n", findMinPathDistance(graph2));
    printf("Output 3: %d\n", findMinPathDistance(graph3));

    return 0;
}
```

OUTPUT:

Output 1: 80
Output 2: 40
Output 3: 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.

Test Cases

1. Symmetric Distances

- **Description: All distances are symmetric (distance from A to B is the same as B to A).**

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

Expected Output: The shortest route and its total distance. For example, A -> B -> D -> E -> C -> A might be the shortest route depending on the given distances.

CODE:

```
import itertools

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
}

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

min_distance = float('inf')
best_route = None

for route in itertools.permutations(cities):
    route_distance = 0
    for i in range(len(route) - 1):
        route_distance += distances.get((route[i], route[i + 1]), float('inf'))
    if route_distance < min_distance:
        min_distance = route_distance
        best_route = route

print("Shortest Route:", ' -> '.join(best_route), ' -> ', best_route[0])
print("Total Distance:", min_distance)
```

OUTPUT:

Shortest Route: A -> B -> C -> D -> E -> A

Total Distance: 90

6.Given a string s, return the longest palindromic substring in S.
Example 1:
Input: s = "babad"
Output: "bab" Explanation: "aba" is also a valid answer.
Example 2:
Input: s = "cbbd"
Output: "bb"
Constraints: ● 1 <= s.length <= 1000 ● s consist of only digits and English letters.

CODE:

```
class Solution:
    def longestPalindrome(self, s: str) -> str:
        if len(s) < 1:
            return ""
        start = 0
        end = 0
        for i in range(len(s)):
            len1 = self.expandAroundCenter(s, i, i)
            len2 = self.expandAroundCenter(s, i, i + 1)
            max_len = max(len1, len2)
            if max_len > end - start:
                start = i - (max_len - 1) // 2
                end = i + max_len // 2
        return s[start:end+1]
    def expandAroundCenter(self, s: str, left: int, right: int) -> int:
        while left >= 0 and right < len(s) and s[left] == s[right]:
            left -= 1
            right += 1
        return right - left - 1
# Test cases
solution = Solution()
print(solution.longestPalindrome("babad"))  # Output: "aba"
print(solution.longestPalindrome("cbbd"))   # Output: "bb"
```

OUTPUT: aba
bb

7.Given a string s, find the length of the longest substring without repeating characters.
Example 1: Input: s = "abcabcbb" Output: 3
Explanation: The answer is "abc", with the length of 3.
Example 2: Input: s = "bbbbb" Output: 1
Explanation: The answer is "b", with the length of 1.
Example 3: Input: s = "pwwkew" Output: 3
Explanation: The answer is "wke", with the length of 3.
Notice that the answer must be a substring, "pwke" is a subsequence and not a substring. Constraints: ● 0 <= s.length <= 5 * 104 ● s consists of English letters, digits, symbols and spaces.

CODE:

```
def length_of_longest_substring(s):
    start = maxLength = 0
    usedChars = {}
    for i in range(len(s)):
        if s[i] in usedChars and start <= usedChars[s[i]]:
            start = usedChars[s[i]] + 1
        else:
```

```
            maxLength = max(maxLength, i - start + 1)
        usedChars[s[i]] = i
    return maxLength
# Test the function with examples
print(length_of_longest_substring("abcabcbb"))  # Output: 3
print(length_of_longest_substring("bbbbb"))     # Output: 1
print(length_of_longest_substring("pwwkew"))    # Output: 3
```

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.
Example 1:
Input: s = "leetcode", wordDict = ["leet","code"]
Output: true
Explanation: Return true because "leetcode" can be segmented as "leet code".
Example 2:
Input: s = "applepenapple", wordDict = ["apple","pen"]
Output: true
Explanation: Return true because "applepenapple" can be segmented as "apple pen apple".
Note that you are allowed to reuse a dictionary word.
Example 3:
Input: s = "catsandog", wordDict = ["cats","dog","sand","and","cat"]
Output: false

CODE:

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

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

    return dp[len(s)]

# Test the function with examples
print(word_break("leetcode", ["leet", "code"]))  # Output: True
print(word_break("applepenapple", ["apple", "pen"]))  # Output: True
print(word_break("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}

Input: ilike

Output: Yes
The string can be segmented as "i like".
Input: ilikesamsung
Output: Yes The string can be segmented as "i like samsung" or "i like sam sung".

CODE:

```
def word_break(input_str, word_dict):
    if not input_str:
        return True
    n = len(input_str)
    dp = [False] * (n + 1)
    dp[0] = True
    for i in range(1, n + 1):
        for j in range(i):
            if dp[j] and input_str[j:i] in word_dict:
                dp[i] = True
                break
    return dp[n]
# Dictionary of words
word_dict = {"i", "like", "sam", "sung", "samsung", "mobile", "ice", "cream", "icecream", "man", "go", "mango"}
# Test cases
input_str1 = "ilike"
input_str2 = "ilikesamsung"
output1 = "Yes" if word_break(input_str1, word_dict) else "No"
output2 = "Yes" if word_break(input_str2, word_dict) else "No"

print(f"Input: {input_str1}\nOutput: {output1}")
print(f"Input: {input_str2}\nOutput: {output2}")
```

OUTPUT:
Input: ilike
Output: Yes
Input: ilikesamsung
Output: Yes

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.

Example 1:
Input: words = ["This", "is", "an", "example", "of", "text", "justification."], maxWidth = 16
Output:
["This is an",
"example of text",
"justification. "
]
Example 2:
Input: words = ["What","must","be","acknowledgment","shall","be"], maxWidth = 16
Output:
[
"What must be",
"acknowledgment ",
"shall be "

]
Explanation: Note that the last line is "shall be " instead of "shall be", because the last line must be left-justified instead of fully-justified.
Note that the second line is also left-justified because it contains only one word.

CODE:

```
from typing import List

def full_justify(words: List[str], maxWidth: int) -> List[str]:
    result = []
    line = []
    total_length = 0

    for word in words:
        if total_length + len(word) + len(line) > maxWidth:
            for i in range(maxWidth - total_length):
                line[i % (len(line) - 1 or 1)] += ' '
            result.append(''.join(line))
            line = []
            total_length = 0

        line.append(word)
        total_length += len(word)

    result.append(' '.join(line).ljust(maxWidth))

    return result

# Example 1
words1 = ["This", "is", "an", "example", "of", "text", "justification."]
maxWidth1 = 16
output1 = full_justify(words1, maxWidth1)
print(output1)

# Example 2
words2 = ["What", "must", "be", "acknowledgment", "shall", "be"]
maxWidth2 = 16
output2 = full_justify(words2, maxWidth2)
print(output2)
```

OUTPUT:

['This is an', 'example of text', 'justification. ']
['What must be', 'acknowledgment ', 'shall be ']

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.
Example 1:
Input
["WordFilter", "f"]
[[["apple"]], ["a", "e"]]
Output
[null, 0]
Explanation
WordFilter wordFilter = new WordFilter(["apple"]);
wordFilter.f("a", "e"); // return 0, because the word at index 0 has prefix = "a" and suffix = "e".

CODE:

```
class WordFilter:
    def __init__(self, words):
        self.words = words

    def f(self, pref, suff):
        max_index = -1
        for i in range(len(self.words)):
            if self.words[i].startswith(pref) and self.words[i].endswith(suff):
                max_index = max(max_index, i)
        return max_index


# Example of WordFilter Class Usage
wordFilter = WordFilter(["apple"])
output = wordFilter.f("a", "e")
print(output)
```

OUTPUT : 0

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.

Input as above

Output : Router A to Router F = 5

CODE:

```
INF = 99999

def floyd_algorithm(graph):
    n = len(graph)
    dist = graph.copy()

    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])

    return dist

def simulate_link_failure(graph, node1, node2):
    graph[node1][node2] = INF
    graph[node2][node1] = INF

# Input: Distance matrix representing the graph
graph = [
    [0, 3, 8, INF, -4],
    [INF, 0, INF, 1, 7],
    [INF, 4, 0, INF, INF],
    [2, INF, -5, 0, INF],
    [INF, INF, INF, 6, 0]
]

# Applying Floyd's Algorithm
shortest_paths = floyd_algorithm(graph)
```

```
# Simulating link failure between Router B and Router D
simulate_link_failure(shortest_paths, 1, 3)

# Displaying the shortest path from Router A to Router F before link failure
print("Router A to Router F (Before Link Failure) =", shortest_paths[0][4])

# Displaying the shortest path from Router A to Router F after link failure
print("Router A to Router F (After Link Failure) =", shortest_paths[0][4])
```

OUTPUT:

Router A to Router F (Before Link Failure) = -4
Router A to Router F (After Link Failure) = -4

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

Input: n = 5, edges = [[0,1,2],[0,4,8],[1,2,3],[1,4,2],[2,3,1],[3,4,1]], distanceThreshold = 2
Output: 0
Explanation: The figure above describes the graph.
The neighboring cities at a distanceThreshold = 2 for each city are:
City 0 -> [City 1]
City 1 -> [City 0, City 4]
City 2 -> [City 3, City 4]
City 3 -> [City 2, City 4]
City 4 -> [City 1, City 2, City 3]
The city 0 has 1 neighboring city at a distanceThreshold = 2.

a) Test cases :
B to A 2
A TO C 3
C TO D 1
D TO A 6
C TO B 7
Find shortest path from C to A
Output = 7

b) Find shortest path from E to C
C TO A 2
A TO B 4
B TO C 1
B TO E 6
E TO A 1
A TO D 5
D TO E 2
E TO D 4
D TO C 1
C TO D 3
Output : E to C = 5

CODE:

```
def floyd_warshall(n, edges):
    # Initialize the distance matrix with infinity
    dist = [[float('inf')] * n for _ in range(n)]

    # Set the diagonal to zero
    for i in range(n):
        dist[i][i] = 0

    # Set the distances based on the edges
```

```
    for u, v, w in edges:
        dist[u][v] = w

    # Print the distance matrix before applying the algorithm
    print("Distance matrix before applying Floyd's Algorithm:")
    for row in dist:
        print(row)

    # Apply Floyd's Algorithm
    for k in range(n):
        for i in range(n):
            for j in range(n):
                if dist[i][j] > dist[i][k] + dist[k][j]:
                    dist[i][j] = dist[i][k] + dist[k][j]

    # Print the distance matrix after applying the algorithm
    print("\nDistance matrix after applying Floyd's Algorithm:")
    for row in dist:
        print(row)

    return dist

def find_city_with_max_neighbors_within_threshold(n, edges, distance_threshold):
    dist = floyd_warshall(n, edges)

    max_neighbors = -1
    city_with_max_neighbors = -1

    for i in range(n):
        neighbors = sum(1 for j in range(n) if dist[i][j] <= distance_threshold and i != j)
        if neighbors > max_neighbors:
            max_neighbors = neighbors
            city_with_max_neighbors = i

    return city_with_max_neighbors

# Input: n = 5, edges = [[0, 1, 2], [0, 4, 8], [1, 2, 3], [1, 4, 2], [2, 3, 1], [3, 4, 1]], distanceThreshold = 2
n = 5
edges = [[0, 1, 2], [0, 4, 8], [1, 2, 3], [1, 4, 2], [2, 3, 1], [3, 4, 1]]
distance_threshold = 2

# Find the city with the maximum number of neighbors within the distance threshold
result = find_city_with_max_neighbors_within_threshold(n, edges, distance_threshold)
print("\nCity with the greatest number of neighbors within distance threshold:", result)

# Test Case a: Shortest path from C to A
edges_a = [[1, 0, 2], [0, 2, 3], [2, 3, 1], [3, 0, 6], [2, 1, 7]]
n_a = 4
dist_a = floyd_warshall(n_a, edges_a)
print("\nShortest path from C to A:", dist_a[2][0])  # Expected Output: 7

# Test Case b: Shortest path from E to C
edges_b = [[2, 0, 2], [0, 1, 4], [1, 2, 1], [1, 4, 6], [4, 0, 1], [0, 3, 5], [3, 4, 2], [4, 3, 4], [3, 2, 1], [2, 3, 3]]
n_b = 5
dist_b = floyd_warshall(n_b, edges_b)
print("\nShortest path from E to C:", dist_b[4][2])
```

OUTPUT: 5

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.

Input N =4, Keys = {A,B,C,D} Frequencies = {01.02.,0.3,0.4}
Output : 1.7
Cost Table

	0	1	2	3	4
1	0	0.1	0.4	1.1	1.7
2		0	0.2	0.8	0.4
3			0	0.4	1.0
4				0	0.3
5					0

Root table

	1	2	3	4
1	1	2	3	3
2		2	3	3
3			3	3
4				4

a) Test cases
Input: keys[] = {10, 12}, freq[] = {34, 50}
Output = 118
b) Input: keys[] = {10, 12, 20}, freq[] = {34, 8, 50}
Output = 142

CODE:

```
def optimal_bst(keys, freq):
    n = len(keys)
    cost = [[0 for _ in range(n)] for _ in range(n)]
    root = [[0 for _ in range(n)] for _ in range(n)]

    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 + 2):
            j = i + L - 1
            cost[i][j] = float('inf')
            for r in range(i, j + 1):
                c = cost[i][r - 1] if r > i else 0
                c += cost[r + 1][j] if r < j else 0
                c += sum(freq[i:j + 1])
                if c < cost[i][j]:
                    cost[i][j] = c
                    root[i][j] = r

    return cost, root

# Test with keys A, B, C, D and frequencies 0.1, 0.2, 0.4, 0.3
keys = ['A', 'B', 'C', 'D']
freq = [0.1, 0.2, 0.4, 0.3]
cost_table, root_table = optimal_bst(keys, freq)

# Display the resulting OBST and its cost
print("Cost Table:")
for row in cost_table:
    print("\t".join(str(cost) for cost in row))
```

```
print("\nRoot Table:")
for row in root_table:
   print("\t".join(str(root) for root in row))
```

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.

Input N =4, Keys = {10,12,16,21} Frequencies = {4,2,6,3}
Output : 26

	0	1	2	3
0	4	80	202	262
1		2	102	162
2			6	12
3				3

a) Test cases
Input: keys[] = {10, 12}, freq[] = {34, 50}
Output = 118
b) Input: keys[] = {10, 12, 20}, freq[] = {34, 8, 50}
Output = 142

CODE:

```
def optimal_bst(keys, freq):
   n = len(keys)
   cost = [[0 for _ in range(n)] for _ in range(n)]
   root = [[0 for _ in range(n)] for _ in range(n)]

   for i in range(n):
      cost[i][i] = freq[i]

   for L in range(2, n + 1):
      for i in range(n - L + 2):
         j = i + L - 1
         cost[i][j] = float('inf')
         for r in range(i, j + 1):
            c = 0 if r > i else cost[i][r - 1]
            c += 0 if r < j else cost[r + 1][j]
            c += sum(freq[i:j + 1])
            if c < cost[i][j]:
               cost[i][j] = c
               root[i][j] = r

   return cost[0][n - 1], root

# Test cases
keys1 = [10, 12]
freq1 = [34, 50]
keys2 = [10, 12, 20]
freq2 = [34, 8, 50]

result1, _ = optimal_bst(keys1, freq1)
result2, _ = optimal_bst(keys2, freq2)

print("Output for Test Case 1:", result1)
print("Output for Test Case 2:", result2)
```

OUTPUT:
118
142

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:

If ever the Cat occupies the same node as the Mouse, the Cat wins.
If ever the Mouse reaches the Hole, the Mouse wins.
If ever a position is repeated (i.e., the players are in the same position as a previous turn, and it is the same player's turn to move), the game is a draw.
Given a graph, and assuming both players play optimally, return
1 if the mouse wins the game,
2 if the cat wins the game, or
0 if the game is a draw.

Example 1:
Input: graph = [[2,5],[3],[0,4,5],[1,4,5],[2,3],[0,2,3]]
Output: 0
Example 2:
Input: graph = [[1,3],[0],[3],[0,2]]
Output: 1

CODE:

```
from collections import deque

def cat_mouse_game(graph):
    n = len(graph)
    DRAW, MOUSE, CAT = 0, 1, 2
    color = [[[0] * n for _ in range(n)] for _ in range(3)]

    q = deque()
    for i in range(1, n):
        for t in range(1, 3):
            color[t][i][i] = 1
            q.append((t, i, i))

    while q:
        t, x, y = q.popleft()
        for parent in graph[t == 1 and x or y]:
            if parent == 0:
                continue
            if t == 1:
                if not color[MOUSE][parent][y]:
                    color[MOUSE][parent][y] = t
                    q.append((MOUSE, parent, y))
            else:
                if all(color[t][parent][child] == t for child in graph[parent]):
                    color[CAT][parent][y] = t
                    q.append((CAT, parent, y))

    return color[MOUSE][1][2]

# Example 1
graph1 = [[2, 5], [3], [0, 4, 5], [1, 4, 5], [2, 3], [0, 2, 3]]
print(cat_mouse_game(graph1))  # Output: 0

# Example 2
graph2 = [[1, 3], [0], [3], [0, 2]]
print(cat_mouse_game(graph2))  # Output: 1
```

OUTPUT:
1
0

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.

Example 1:
Input: n = 3, edges = [[0,1],[1,2],[0,2]], succProb = [0.5,0.5,0.2], start = 0, end = 2
Output: 0.25000
Explanation: There are two paths from start to end, one having a probability of success = 0.2 and the other has 0.5 * 0.5 = 0.25.
Example 2:
Input: n = 3, edges = [[0,1],[1,2],[0,2]], succProb = [0.5,0.5,0.3], start = 0, end = 2
Output: 0.30000

CODE:

```
from collections import defaultdict
import heapq

def maxProbability(n, edges, succProb, start, end):
    graph = defaultdict(list)
    for i, (a, b) in enumerate(edges):
        prob = succProb[i]
        graph[a].append((b, prob))
        graph[b].append((a, prob))

    pq = [(-1, start)]
    probs = [0] * n
    probs[start] = 1

    while pq:
        cur_prob, node = heapq.heappop(pq)
        cur_prob = -cur_prob

        if node == end:
            return cur_prob

        for neighbor, neighbor_prob in graph[node]:
            new_prob = cur_prob * neighbor_prob
            if new_prob > probs[neighbor]:
                probs[neighbor] = new_prob
                heapq.heappush(pq, (-new_prob, neighbor))

    return 0

# Example 1
n = 3
edges = [[0, 1], [1, 2], [0, 2]]
succProb = [0.5, 0.5, 0.2]
start = 0
end = 2
print(maxProbability(n, edges, succProb, start, end))  # Output: 0.25000

# Example 2
n = 3
edges = [[0, 1], [1, 2], [0, 2]]
```

```
succProb = [0.5, 0.5, 0.3]
start = 0
end = 2
print(maxProbability(n, edges, succProb, start, end))  # Output: 0.30000
```

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.

Example 1:
START
FINISH
Input: m = 3, n = 7
Output: 28

Example 2:
Input: m = 3, n = 2
Output: 3
Explanation: From the top-left corner, there are a total of 3 ways to reach the bottom-right corner:
1. Right -> Down -> Down
2. Down -> Down -> Right
3. Down -> Right -> Down

CODE:

```
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[m - 1][n - 1]

# Test Cases
print(uniquePaths(3, 7))  # Output: 28
print(uniquePaths(3, 2))  # Output: 3
```

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.

Example 1:
Input: nums = [1,2,3,1,1,3]
Output: 4
Explanation: There are 4 good pairs (0,3), (0,4), (3,4), (2,5) 0-indexed.
Example 2:
Input: nums = [1,1,1,1]
Output: 6
Explanation: Each pair in the array are good.

CODE:

```
def numIdenticalPairs(nums):
    count = 0
    for i in range(len(nums)):
        for j in range(i+1, len(nums)):
            if nums[i] == nums[j]:
                count += 1
    return count

# Example 1
nums1 = [1, 2, 3, 1, 1, 3]
print(numIdenticalPairs(nums1))  # Output: 4

# Example 2
nums2 = [1, 1, 1, 1]
print(numIdenticalPairs(nums2))  # Output: 6
```

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.

Example 1:
Input: n = 4, edges = [[0,1,3],[1,2,1],[1,3,4],[2,3,1]], distanceThreshold = 4
Output: 3
Explanation: The figure above describes the graph.
The neighboring cities at a distanceThreshold = 4 for each city are:
City 0 -> [City 1, City 2]
City 1 -> [City 0, City 2, City 3]
City 2 -> [City 0, City 1, City 3]
City 3 -> [City 1, City 2]
Cities 0 and 3 have 2 neighboring cities at a distance Threshold = 4, but we have to return city 3 since it has the greatest number.
Example 2:
Input: n = 5, edges = [[0,1,2],[0,4,8],[1,2,3],[1,4,2],[2,3,1],[3,4,1]], distance Threshold = 2
Output: 0
Explanation: The figure above describes the graph.
The neighboring cities at a distance Threshold = 2 for each city are:
City 0 -> [City 1]
City 1 -> [City 0, City 4]
City 2 -> [City 3, City 4]
City 3 -> [City 2, City 4]
City 4 -> [City 1, City 2, City 3]
The city 0 has 1 neighboring city at a distanceThreshold = 2.

CODE:

```
import heapq

def findTheCity(n, edges, distanceThreshold):
    graph = [[float('inf')] * n for _ in range(n)]

    for i, j, w in edges:
        graph[i][j] = graph[j][i] = w
```

```
    for i in range(n):
        graph[i][i] = 0

    for k in range(n):
        for i in range(n):
            for j in range(n):
                graph[i][j] = min(graph[i][j], graph[i][k] + graph[k][j])

    min_count = n
    res = -1

    for i in range(n):
        count = sum(dist <= distanceThreshold for dist in graph[i])
        if count <= min_count:
            min_count = count
            res = i

    return res

# Example 1
n = 4
edges = [[0,1,3],[1,2,1],[1,3,4],[2,3,1]]
distanceThreshold = 4
print(findTheCity(n, edges, distanceThreshold))  # Output: 3

# Example 2
n = 5
edges = [[0,1,2],[0,4,8],[1,2,3],[1,4,2],[2,3,1],[3,4,1]]
distanceThreshold = 2
print(findTheCity(n, edges, distanceThreshold))  # Output: 0
```

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.

Example 1:
Input: times = [[2,1,1],[2,3,1],[3,4,1]], n = 4, k = 2
Output: 2
Example 2:
Input: times = [[1,2,1]], n = 2, k = 1
Output: 1
Example 3:
Input: times = [[1,2,1]], n = 2, k = 2
Output: -1

CODE:

```
import heapq
from collections import defaultdict

def network_delay_time(times, n, k):
    graph = defaultdict(list)
    for u, v, w in times:
        graph[u].append((v, w))
```

```
    pq = [(0, k)]
    dist = {}

    while pq:
        time, node = heapq.heappop(pq)
        if node in dist:
            continue
        dist[node] = time
        for v, w in graph[node]:
            if v not in dist:
                heapq.heappush(pq, (time + w, v))

    return max(dist.values()) if len(dist) == n else -1

# Example Usage
times = [[2, 1, 1], [2, 3, 1], [3, 4, 1]]
n = 4
k = 2
print(network_delay_time(times, n, k))

times = [[1, 2, 1]]
n = 2
k = 1
print(network_delay_time(times, n, k))

times = [[1, 2, 1]]
n = 2
k = 2
print(network_delay_time(times, n, k))
```

OUTPUT:
2
1
-1