TOPIC 1 : INTRODUCTION

1. Given an array of strings words, return the first palindromic string in the array. If there is no such string, return an empty string "". A string is palindromic if it reads the same forward and backward.
   Example 1:
   Input: words = ["abc","car","ada","racecar","cool"]
   Output: "ada"
   Explanation: The first string that is palindromic is "ada".
   Note that "racecar" is also palindromic, but it is not the first.

   Example 2:
   Input: words = ["notapalindrome","racecar"]
   Output: "racecar"
   Explanation: The first and only string that is palindromic is "racecar".

   Program :-

```
def first_palindromic_string(words):
    for word in words:
        if word == word[::-1]:  # Check if the word reads the same forwards and
backwards
            return word
    return ""

# Example 1
words1 = ["abc", "car", "ada", "racecar", "cool"]
print(first_palindromic_string(words1))  # Output: "ada"

# Example 2
words2 = ["notapalindrome", "racecar"]
print(first_palindromic_string(words2))  # Output: "racecar"
```

2. You are given two integer arrays nums1 and nums2 of sizes n and m, respectively. Calculate the following values: answer1 : the number of indices i such that nums1[i] exists in nums2. answer2 : the number of indices i such that nums2[i] exists in nums1 Return [answer1,answer2].

   Example 1:
   Input: nums1 = [2,3,2], nums2 = [1,2]
   Output: [2,1]
   Explanation:

   Example 2:
   Input: nums1 = [4,3,2,3,1], nums2 = [2,2,5,2,3,6]
   Output: [3,4]
   Explanation:
   The elements at indices 1, 2, and 3 in nums1 exist in nums2 as well. So answer1 is 3.
   The elements at indices 0, 1, 3, and 4 in nums2 exist in nums1. So answer2 is 4.

Program :- def count_indices(nums1, nums2):

```
    # Convert nums2 to a set for O(1) average-time complexity checks
    set_nums2 = set(nums2)

    # Count how many elements in nums1 exist in nums2
    answer1 = sum(1 for num in nums1 if num in set_nums2)

    # Convert nums1 to a set for O(1) average-time complexity checks
    set_nums1 = set(nums1)

    # Count how many elements in nums2 exist in nums1
    answer2 = sum(1 for num in nums2 if num in set_nums1)

    return [answer1, answer2]

# Example 1
nums1 = [2, 3, 2]
nums2 = [1, 2]
print(count_indices(nums1, nums2))  # Output: [2, 1]

# Example 2
nums1 = [4, 3, 2, 3, 1]
nums2 = [2, 2, 5, 2, 3, 6]
print(count_indices(nums1, nums2))  # Output: [3, 4]
```

3. You are given a 0-indexed integer array nums. The distinct count of a subarray of nums is defined as: Let nums[i..j] be a subarray of nums consisting of all the indices from i to j such that 0 <= i <= j < nums.length. Then the number of distinct values in nums[i..j] is called the distinct count of nums[i..j]. Return the sum of the squares of distinct counts of all subarrays of nums. A subarray is a contiguous non-empty sequence of elements within an array.
   Example 1:
   Input: nums = [1,2,1]
   Output: 15
   Explanation: Six possible subarrays are:
   [1]: 1 distinct value
   [2]: 1 distinct value
   [1]: 1 distinct value
   [1,2]: 2 distinct values
   [2,1]: 2 distinct values
   [1,2,1]: 2 distinct values
   The sum of the squares of the distinct counts in all subarrays is equal to 12 + 12 + 12 + 22 + 22 + 22 = 15.

   Example 2:
   Input: nums = [1,1]
   Output: 3
   Explanation: Three possible subarrays are:
   [1]: 1 distinct value
   [1]: 1 distinct value

[1,1]: 1 distinct value
The sum of the squares of the distinct counts in all subarrays is equal to 12 + 12 + 12 = 3.

Program :-

```
def sum_of_squares_of_distinct_counts(nums):
    n = nums.length
    total_sum = 0

    for i in range(n):
        distinct_elements = set()
        for j in range(i, n):
            distinct_elements.add(nums[j])
            distinct_count = len(distinct_elements)
            total_sum += distinct_count ** 2

    return total_sum

# Example 1
nums1 = [1, 2, 1]
print(sum_of_squares_of_distinct_counts(nums1))  # Output: 15

# Example 2
nums2 = [1, 1]
print(sum_of_squares_of_distinct_counts(nums2))  # Output: 3
```

4. Given a 0-indexed integer array nums of length n and an integer k, return *the number of pairs* (i, j) *where* 0 <= i < j < n, *such that* nums[i] == nums[j] *and* (i * j) *is divisible by* k.
Example 1:
Input: nums = [3,1,2,2,2,1,3], k = 2
Output: 4
Explanation:
There are 4 pairs that meet all the requirements:
- nums[0] == nums[6], and 0 * 6 == 0, which is divisible by 2.
- nums[2] == nums[3], and 2 * 3 == 6, which is divisible by 2.
- nums[2] == nums[4], and 2 * 4 == 8, which is divisible by 2.
- nums[3] == nums[4], and 3 * 4 == 12, which is divisible by 2.
Example 2:
Input: nums = [1,2,3,4], k = 1
Output: 0
Explanation: Since no value in nums is repeated, there are no pairs (i,j) that meet all the requirements.

Program :-

```
def count_pairs(nums, k):
    n = len(nums)
    count = 0
```

```
    for i in range(n):
        for j in range(i + 1, n):
            if nums[i] == nums[j] and (i * j) % k == 0:
                count += 1

    return count

# Example 1
nums1 = [3, 1, 2, 2, 2, 1, 3]
k1 = 2
print(count_pairs(nums1, k1))  # Output: 4

# Example 2
nums2 = [1, 2, 3, 4]
k2 = 1
print(count_pairs(nums2, k2))  # Output: 0
```

5. Write a program FOR THE BELOW TEST CASES with least time complexity
   Test Cases: -
   1) Input: {1, 2, 3, 4, 5} Expected Output: 5
   2) Input: {7, 7, 7, 7, 7} Expected Output: 7
   3) Input: {-10, 2, 3, -4, 5} Expected Output: 5

Program :-

```
def find_maximum(nums):
    # Initialize the maximum element to the first element of the array
    max_val = nums[0]

    # Iterate through the array to find the maximum element
    for num in nums:
        if num > max_val:
            max_val = num

    return max_val

# Test Case 1
nums1 = [1, 2, 3, 4, 5]
print(find_maximum(nums1))  # Expected Output: 5

# Test Case 2
nums2 = [7, 7, 7, 7, 7]
print(find_maximum(nums2))  # Expected Output: 7

# Test Case 3
nums3 = [-10, 2, 3, -4, 5]
print(find_maximum(nums3))  # Expected Output: 5
```

6. You have an algorithm that process a list of numbers. It firsts sorts the list using an efficient sorting algorithm and then finds the maximum element in sorted list. Write the code for the same.
   Test Cases
   1. Empty List
      1. Input: []
      2. Expected Output: None or an appropriate message indicating that the list is empty.
   2. Single Element List
      1. Input: [5]
      2. Expected Output: 5
   3. All Elements are the Same
      1. Input: [3, 3, 3, 3, 3]
      2. Expected Output: 3

Program :-

```
def process_numbers(nums):
    if not nums:
        return "The list is empty."

    nums.sort()

    max_val = nums[-1]

    return max_val

# Test Cases

# Test Case 1: Empty List
nums1 = []
print(process_numbers(nums1))  # Expected Output: "The list is empty."

# Test Case 2: Single Element List
nums2 = [5]
print(process_numbers(nums2))  # Expected Output: 5

# Test Case 3: All Elements are the Same
nums3 = [3, 3, 3, 3, 3]
print(process_numbers(nums3))  # Expected Output: 3

# Additional Test Case 4: General Case
nums4 = [1, 3, 2, 5, 4]
print(process_numbers(nums4))  # Expected Output: 5
```

7. Write a program that takes an input list of n numbers and creates a new list containing only the unique elements from the original list. What is the space complexity of the algorithm?
   Test Cases

Some Duplicate Elements

- Input: [3, 7, 3, 5, 2, 5, 9, 2]
- Expected Output: [3, 7, 5, 2, 9] (Order may vary based on the algorithm used)

Negative and Positive Numbers

- Input: [-1, 2, -1, 3, 2, -2]
- Expected Output: [-1, 2, 3, -2] (Order may vary)

List with Large Numbers

- Input: [1000000, 999999, 1000000]
- Expected Output: [1000000, 999999]

Program :-

```
def unique_elements(nums):
    unique_set = set()
    unique_list = []

    for num in nums:
        if num not in unique_set:
            unique_set.add(num)
            unique_list.append(num)

    return unique_list

# Test Cases

# Test Case 1: Some Duplicate Elements
nums1 = [3, 7, 3, 5, 2, 5, 9, 2]
print(unique_elements(nums1))  # Expected Output: [3, 7, 5, 2, 9] (Order may vary)

# Test Case 2: Negative and Positive Numbers
nums2 = [-1, 2, -1, 3, 2, -2]
print(unique_elements(nums2))  # Expected Output: [-1, 2, 3, -2] (Order may vary)

# Test Case 3: List with Large Numbers
nums3 = [1000000, 999999, 1000000]
print(unique_elements(nums3))  # Expected Output: [1000000, 999999] (Order may vary)
```

8. Sort an array of integers using the bubble sort technique. Analyze its time complexity using Big-O notation. Write the code.

Program :-

```
def bubble_sort(nums):
    n = len(nums)
    for i in range(n):
        # Track if any swaps happen
        swapped = False
```

```
            # The last i elements are already sorted
            for j in range(0, n - i - 1):
                # Swap if the element found is greater than the next element
                if nums[j] > nums[j + 1]:
                    nums[j], nums[j + 1] = nums[j + 1], nums[j]
                    swapped = True
            # If no swaps happen, the list is already sorted
            if not swapped:
                break
        return nums

    # Test Cases

    # Test Case 1: Some Duplicate Elements
    nums1 = [3, 7, 3, 5, 2, 5, 9, 2]
    print(bubble_sort(nums1))  # Expected Output: [2, 2, 3, 3, 5, 5, 7, 9]

    # Test Case 2: Negative and Positive Numbers
    nums2 = [-1, 2, -1, 3, 2, -2]
    print(bubble_sort(nums2))  # Expected Output: [-2, -1, -1, 2, 2, 3]

    # Test Case 3: List with Large Numbers
    nums3 = [1000000, 999999, 1000000]
    print(bubble_sort(nums3))  # Expected Output: [999999, 1000000, 1000000]
```

9. Checks if a given number x exists in a sorted array arr using binary search. Analyze its time complexity using Big-O notation.
   Test Case:
   Example X={ 3,4,6,-9,10,8,9,30} KEY=10
   Output: Element 10 is found at position 5

   Example X={ 3,4,6,-9,10,8,9,30} KEY=100
   Output : Element 100 is not found

Program :-

```
def binary_search(arr, key):
    low = 0
    high = len(arr) - 1

    while low <= high:
        mid = (low + high) // 2

        # Check if key is present at mid
        if arr[mid] == key:
            return mid

        # If key is greater, ignore the left half
        elif arr[mid] < key:
```

```
            low = mid + 1

        # If key is smaller, ignore the right half
        else:
            high = mid - 1

    # If we reach here, the element was not present
    return -1

# Example Test Cases

# Test Case 1: Element is present
X1 = [3, 4, 6, -9, 10, 8, 9, 30]
X1.sort()  # Binary search requires a sorted array
KEY1 = 10
index1 = binary_search(X1, KEY1)
if index1 != -1:
    print(f'Element {KEY1} is found at position {index1}")
else:
    print(f'Element {KEY1} is not found")

# Test Case 2: Element is not present
X2 = [3, 4, 6, -9, 10, 8, 9, 30]
X2.sort()  # Binary search requires a sorted array
KEY2 = 100
index2 = binary_search(X2, KEY2)
if index2 != -1:
    print(f'Element {KEY2} is found at position {index2}")
else:
    print(f'Element {KEY2} is not found")
```

10. Given an array of integers nums, sort the array in ascending order and return it. You must solve the problem without using any built-in functions in O(nlog(n)) time complexity and with the smallest space complexity possible.

Program :-

```
def merge_sort(nums):
    if len(nums) <= 1:
        return nums

    # Split the array into two halves
    mid = len(nums) // 2
    left_half = merge_sort(nums[:mid])
    right_half = merge_sort(nums[mid:])

    # Merge the sorted halves
    return merge(left_half, right_half)

def merge(left, right):
```

```
    sorted_array = []
    i = j = 0

    # Merge the two halves while maintaining sorted order
    while i < len(left) and j < len(right):
        if left[i] < right[j]:
            sorted_array.append(left[i])
            i += 1
        else:
            sorted_array.append(right[j])
            j += 1

    # Append any remaining elements
    while i < len(left):
        sorted_array.append(left[i])
        i += 1
    while j < len(right):
        sorted_array.append(right[j])
        j += 1

    return sorted_array

# Test Cases
nums1 = [3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5]
print(merge_sort(nums1))  # Expected Output: [1, 1, 2, 3, 3, 4, 5, 5, 5, 6, 9]

nums2 = [10, -1, 2, 5, 0, 6, 4, -5]
print(merge_sort(nums2))  # Expected Output: [-5, -1, 0, 2, 4, 5, 6, 10]

nums3 = [1]
print(merge_sort(nums3))  # Expected Output: [1]

nums4 = []
print(merge_sort(nums4))  # Expected Output: []
```

11. Given an m x n grid and a ball at a starting cell, find the number of ways to move the ball out of the grid boundary in exactly N steps.

Example:

- Input: m=2,n=2,N=2,i=0,j=0 · Output: 6
- Input: m=1,n=3,N=3,i=0,j=1 · Output: 12

Program :-

```
def findPaths(m, n, N, i, j):
    MOD = 10**9 + 7

    # Create a 3D DP array with dimensions (N+1) x m x n
    dp = [[[0 for _ in range(n)] for _ in range(m)] for _ in range(N+1)]

    # Initialize the starting position
```

```
        dp[0][i][j] = 1

        # Directions for moving up, down, left, right
        directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]

        count = 0

        # Iterate through all the steps from 1 to N
        for steps in range(1, N + 1):
            for r in range(m):
                for c in range(n):
                    # If the current cell has ways to reach it
                    if dp[steps - 1][r][c] > 0:
                        # Move in all four directions
                        for dr, dc in directions:
                            nr, nc = r + dr, c + dc
                            # Check if it is out of bounds
                            if nr < 0 or nr >= m or nc < 0 or nc >= n:
                                count = (count + dp[steps - 1][r][c]) % MOD
                            else:
                                dp[steps][nr][nc] = (dp[steps][nr][nc] + dp[steps - 1][r][c]) % MOD

        return count

    # Test Cases

    # Test Case 1: m=2, n=2, N=2, i=0, j=0
    m1, n1, N1, i1, j1 = 2, 2, 2, 0, 0
    print(findPaths(m1, n1, N1, i1, j1))  # Expected Output: 6

    # Test Case 2: m=1, n=3, N=3, i=0, j=1
    m2, n2, N2, i2, j2 = 1, 3, 3, 0, 1
    print(findPaths(m2, n2, N2, i2, j2))  # Expected Output: 12
```

12. You are a professional robber planning to rob houses along a street. Each house has a certain amount of money stashed. All houses at this place are arranged in a circle. That means the first house is the neighbor of the last one. Meanwhile, adjacent houses have security systems connected, and it will automatically contact the police if two adjacent houses were broken into on the same night.
    Examples:

(i) Input : nums = [2, 3, 2]
Output : The maximum money you can rob without alerting the police is 3(robbing house 1).

(ii) Input : nums = [1, 2, 3, 1]
Output : The maximum money you can rob without alerting the police is 4 (robbing house 1 and house 3).

Program :-

```
def rob(nums):
```

```
    if not nums:
        return 0
    n = len(nums)
    if n == 1:
        return nums[0]

    def rob_linear(nums):
        if not nums:
            return 0
        n = len(nums)
        if n == 1:
            return nums[0]

        # Initialize dp arrays
        dp = [0] * n
        dp[0] = nums[0]
        dp[1] = max(nums[0], nums[1])

        for i in range(2, n):
            dp[i] = max(dp[i-1], nums[i] + dp[i-2])

        return dp[-1]

    # Two scenarios:
    # 1. Rob from first house to second-to-last house
    # 2. Rob from second house to last house

    # Scenario 1: Rob from house 0 to house n-2
    max1 = rob_linear(nums[:-1])

    # Scenario 2: Rob from house 1 to house n-1
    max2 = rob_linear(nums[1:])

    # Return the maximum of these two scenarios
    return max(max1, max2)

# Test Cases
nums1 = [2, 3, 2]
nums2 = [1, 2, 3, 1]

print("Example 1:", rob(nums1))  # Expected Output: 3
print("Example 2:", rob(nums2))  # Expected Output: 4
```

13. You are climbing a staircase. It takes n steps to reach the top. Each time you can either climb 1 or 2 steps. In how many distinct ways can you climb to the top?

Examples:

(i) Input: n=4 Output: 5

(ii) Input: n=3 Output: 3

Program :-

```
def climbStairs(n):
    if n == 0:
        return 1  # Edge case: 1 way to do nothing (stay on the ground)
    if n == 1:
        return 1  # Edge case: 1 way to reach the first step (take 1 step)

    # Initialize dp array to store number of ways to reach each step
    dp = [0] * (n + 1)
    dp[0] = 1  # 1 way to stay on the ground (do nothing)
    dp[1] = 1  # 1 way to reach the first step (take 1 step)

    # Calculate number of ways for each step up to n
    for i in range(2, n + 1):
        dp[i] = dp[i - 1] + dp[i - 2]

    return dp[n]

# Test Cases
print("Example 1:", climbStairs(4))  # Expected Output: 5
print("Example 2:", climbStairs(3))  # Expected Output: 3
```

14. A robot is located at the top-left corner of a m×n grid .The robot can only move either down or right at any point in time. The robot is trying to reach the bottom-right corner of the grid. How many possible unique paths are there?
    Examples:
    (i) Input: m=7,n=3 Output: 28
    (ii) Input: m=3,n=2 Output: 3

Program :-

```
def uniquePaths(m, n):
    # Initialize dp table with zeros
    dp = [[0] * n for _ in range(m)]

    # Base case: There is 1 way to be at the starting point
    dp[0][0] = 1

    # Fill the dp table
    for i in range(m):
        for j in range(n):
            if i > 0:
                dp[i][j] += dp[i-1][j]  # Move from above
            if j > 0:
                dp[i][j] += dp[i][j-1]  # Move from left

    # Return the number of unique paths to reach the bottom-right corner
    return dp[m-1][n-1]

# Test Cases
print("Example 1:", uniquePaths(7, 3))  # Expected Output: 28
```

```
print("Example 2:", uniquePaths(3, 2))  # Expected Output: 3
```

15. In a string S of lowercase letters, these letters form consecutive groups of the same character. For example, a string like s = "abbxxxxzyy" has the groups "a", "bb", "xxxx", "z", and "yy". A group is identified by an interval [start, end], where start and end denote the start and end indices (inclusive) of the group. In the above example, "xxxx" has the interval [3,6]. A group is considered large if it has 3 or more characters. Return the intervals of every large group sorted in increasing order by start index.

    Example 1:
    Input: s = "abbxxxxzzy"
    Output: [[3,6]]
    Explanation: "xxxx" is the only large group with start index 3 and end index 6.

    Example 2:
    Input: s = "abc"
    Output: []
    Explanation: We have groups "a", "b", and "c", none of which are large groups.

    Program :-

```
def largeGroupPositions(s):
    if len(s) < 3:
        return []

    intervals = []
    start = 0
    current_char = s[0]

    for end in range(1, len(s)):
        if s[end] == current_char:
            continue
        else:
            if end - start >= 3:
                intervals.append([start, end - 1])
            current_char = s[end]
            start = end

    # Check the last group
    if len(s) - start >= 3:
        intervals.append([start, len(s) - 1])

    return intervals

# Test Cases
print("Example 1:", largeGroupPositions("abbxxxxzzy"))  # Expected Output: [[3,6]]
print("Example 2:", largeGroupPositions("abc"))         # Expected Output: []
```

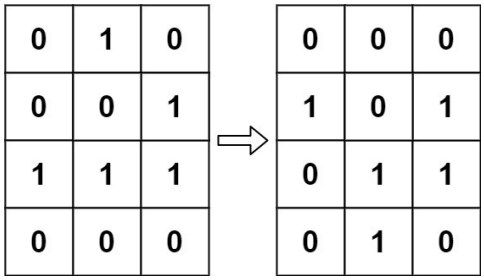
16. "The Game of Life, also known simply as Life, is a cellular automaton devised by the British mathematician John Horton Conway in 1970." The board is made up of an m x n grid of cells, where each cell has an initial state: live (represented by a 1) or dead (represented by a 0). Each cell interacts with its eight neighbors (horizontal, vertical, diagonal) using the following four rules

    Any live cell with fewer than two live neighbors dies as if caused by under-population.

    1. Any live cell with two or three live neighbors lives on to the next generation.
    2. Any live cell with more than three live neighbors dies, as if by over-population.
    3. Any dead cell with exactly three live neighbors becomes a live cell, as if by reproduction.

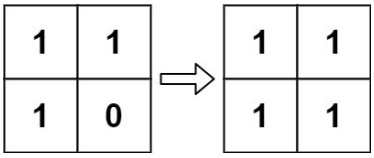
    The next state is created by applying the above rules simultaneously to every cell in the current state, where births and deaths occur simultaneously. Given the current state of the m x n grid board, return *the next state*.

    Example 1:



Input: board = [[0,1,0],[0,0,1],[1,1,1],[0,0,0]]
Output: [[0,0,0],[1,0,1],[0,1,1],[0,1,0]]

Example 2:



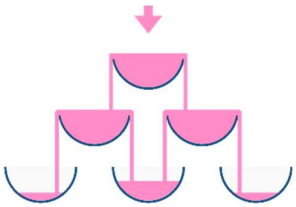
Input: board = [[1,1],[1,0]]
Output: [[1,1],[1,1]]

Program :-

```
def gameOfLife(board):
    m, n = len(board), len(board[0])

    # Directions for neighbors (8 possible directions including diagonals)
    directions = [(-1, -1), (-1, 0), (-1, 1),
                  (0, -1),          (0, 1),
                  (1, -1), (1, 0), (1, 1)]

    # Function to count live neighbors for a given cell (i, j)
    def count_live_neighbors(i, j):
        live_count = 0
```

```
        for d in directions:
            ni, nj = i + d[0], j + d[1]
            if 0 <= ni < m and 0 <= nj < n and (board[ni][nj] == 1 or board[ni][nj] == 2):
                live_count += 1
        return live_count

    # Iterate through the board and apply the rules
    for i in range(m):
        for j in range(n):
            live_neighbors = count_live_neighbors(i, j)

            if board[i][j] == 1:
                if live_neighbors < 2 or live_neighbors > 3:
                    board[i][j] = 2  # Mark as dead in next state (transition state)
            elif board[i][j] == 0:
                if live_neighbors == 3:
                    board[i][j] = -1  # Mark as alive in next state (transition state)

    # Update the board based on transition states
    for i in range(m):
        for j in range(n):
            if board[i][j] == 2:
                board[i][j] = 0
            elif board[i][j] == -1:
                board[i][j] = 1

    return board

# Test Cases
board1 = [[0,1,0],[0,0,1],[1,1,1],[0,0,0]]
print("Example 1:")
print("Input Board:")
for row in board1:
    print(row)
print("Output Board:")
result1 = gameOfLife(board1)
for row in result1:
    print(row)

board2 = [[1,1],[1,0]]
print("\nExample 2:")
print("Input Board:")
for row in board2:
    print(row)
print("Output Board:")
result2 = gameOfLife(board2)
for row in result2:
    print(row)
```

17. We stack glasses in a pyramid, where the first row has 1 glass, the second row has 2 glasses, and so on until the 100th row. Each glass holds one cup of champagne. Then, some champagne is poured into the first glass at the top. When the topmost glass is full, any excess liquid poured will fall equally to the glass immediately to the left and right of it. When those glasses become full, any excess champagne will fall equally to the left and right of those glasses, and so on. (A glass at the bottom row has its excess champagne fall on the floor.) For example, after one cup of champagne is poured, the top most glass is full. After two cups of champagne are poured, the two glasses on the second row are half full. After three cups of champagne are poured, those two cups become full - there are 3 full glasses total now. After four cups of champagne are poured, the third row has the middle glass half full, and the two outside glasses are a quarter full, as pictured below.



Now after pouring some non-negative integer cups of champagne, return how full the jth glass in the ith row is (both i and j are 0-indexed.)

Example 1:

Input: poured = 1, query_row = 1, query_glass = 1
Output: 0.00000
Explanation: We poured 1 cup of champange to the top glass of the tower (which is indexed as (0, 0)). There will be no excess liquid so all the glasses under the top glass will remain empty.

Example 2:

Input: poured = 2, query_row = 1, query_glass = 1
Output: 0.50000
Explanation: We poured 2 cups of champange to the top glass of the tower (which is indexed as (0, 0)). There is one cup of excess liquid. The glass indexed as (1, 0) and the glass indexed as (1, 1) will share the excess liquid equally, and each will get half cup of champange.

Program :-

```
def champagneTower(poured, query_row, query_glass):
    dp = [[0.0] * (r + 1) for r in range(query_row + 1)]
    dp[0][0] = poured

    for i in range(query_row):
        for j in range(len(dp[i])):
            if dp[i][j] > 1:
                excess = dp[i][j] - 1
                dp[i+1][j] += excess / 2.0
                dp[i+1][j+1] += excess / 2.0

    return min(1.0, dp[query_row][query_glass])
```

```
# Test cases
print(champagneTower(1, 1, 1))  # Output: 0.0
print(champagneTower(2, 1, 1))  # Output: 0.5
```