DAY-2 PROGRAMS

1. 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

main.py Share Run Output

```python
def findPaths(m, n, N, i, j):
    dp = [[[0 for _ in range(n)] for _ in range(m)] for _ in
        range(N + 1)]
    directions = [(1, 0), (-1, 0), (0, 1), (0, -1)]
    for step in range(1, N + 1):
        for r in range(m):
            for c in range(n):
                for dr, dc in directions:
                    nr, nc = r + dr, c + dc
                    if 0 <= nr < m and 0 <= nc < n:
                        dp[step][r][c] += dp[step - 1][nr][nc]
                    else:
                        dp[step][r][c] += 1
    return dp[N][i][j]

# Test Cases
print(findPaths(2, 2, 2, 0, 0))  # Output: 6
print(findPaths(1, 3, 3, 0, 1))  # Output: 12

```

```
6
12

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

2. 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).

```python
def rob(nums):
    def simple_rob(nums):
        rob1, rob2 = 0, 0
        for n in nums:
            new_rob = max(rob1 + n, rob2)
            rob1 = rob2
            rob2 = new_rob
        return rob2
    if len(nums) == 1:
        return nums[0]
    return max(simple_rob(nums[:-1]), simple_rob(nums[1:]))

# Test Cases
print(rob([2, 3, 2]))  # Output: 3
print(rob([1, 2, 3, 1]))  # Output: 4

```

```
3
4

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

3. 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

```python
def climbStairs(n):
    if n == 1:
        return 1
    if n == 2:
        return 2
    first, second = 1, 2
    for i in range(3, n + 1):
        third = first + second
        first = second
        second = third

    return second

print(climbStairs(4))  # Output: 5
print(climbStairs(3))  # Output: 3
```

```
5
3

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

4. 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

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

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

```
28
3

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

5. 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.

```python
def largeGroupPositions(s):
    result = []
    start = 0
    for i in range(1, len(s) + 1):
        if i == len(s) or s[i] != s[i - 1]:

            if i - start >= 3:
                result.append([start, i - 1])

            start = i

    return result

# Test Cases
print(largeGroupPositions("abbxxxxzzy"))  # Output: [[3, 6]]
print(largeGroupPositions("abc"))         # Output: []
print(largeGroupPositions("aaa"))         # Output: [[0, 2]]
print(largeGroupPositions("abcdddeeeeaabbbcd"))  # Output: [[3,
```

```
[[3, 6]]
[]
[[0, 2]]
[[3, 5], [6, 9], [12, 14]]

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

6. "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 underpopulation. 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 overpopulation. 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]].

main.py Share Run Output

```python
def gameOfLife(board):
    m, n = len(board), len(board[0])
    next_state = [[0] * n for _ in range(m)]
    directions = [(-1, -1), (-1, 0), (-1, 1), (0, -1), (0, 1),
        (1, -1), (1, 0), (1, 1)]
    def count_live_neighbors(x, y):
        count = 0
        for dx, dy in directions:
            nx, ny = x + dx, y + dy
            if 0 <= nx < m and 0 <= ny < n and board[nx][ny] ==
                1:
                count += 1
        return count
    for i in range(m):
        for j in range(n):
            live_neighbors = count_live_neighbors(i, j)

            # Apply the rules of the game
            if board[i][j] == 1:  # Cell is currently live
```

```
[[0, 0, 0], [1, 0, 1], [0, 1, 1], [0, 1, 0]]
[[1, 1], [1, 1]]

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

main.py

```
                if live_neighbors == 3:
                    next_state[i][j] = 1  # Cell becomes live
    for i in range(m):
        for j in range(n):
            board[i][j] = next_state[i][j]
board1 = [
    [0, 1, 0],
    [0, 0, 1],
    [1, 1, 1],
    [0, 0, 0]
]
board2 = [
    [1, 1],
    [1, 0]
]
gameOfLife(board1)
print(board1)
gameOfLife(board2)
print(board2)
```

Output

```
[[0, 0, 0], [1, 0, 1], [0, 1, 1], [0, 1, 0]]
[[1, 1], [1, 1]]

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

7. 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 j th glass in the i th 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 champagne.

```
def champagneTower(poured: int, query_row: int, query_glass:
    int) -> float:
    dp = [[0] * (r + 1) for r in range(query_row + 1)]
    dp[0][0] = poured
    for r in range(query_row):
        for c in range(r + 1):
            excess = (dp[r][c] - 1.0) / 2.0
            if excess > 0:
                dp[r + 1][c] += excess
                dp[r + 1][c + 1] += excess
    return min(1, dp[query_row][query_glass])

# Example usage
poured = 2
query_row = 1
query_glass = 1
result = champagneTower(poured, query_row, query_glass)
print(f"The glass at row {query_row}, glass {query_glass} is
    {result} full.")
```

```
The glass at row 1, glass 1 is 0.5 full.

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