

11.

## Ball Moving Out of Grid in N Steps

### Aim

Find the number of ways a ball can move out of a grid in exactly **N steps**.

### Algorithm

1. Use **DFS with memoization (Dynamic Programming)**.
2. At each step, move in four directions (up, down, left, right).
3. If the ball goes outside the grid, count it as 1.
4. If steps run out inside the grid, return 0.
5. Use memoization (i, j, steps) to avoid recomputation.

### Input

m=2, n=2, N=2, i=0, j=0

### Output

6

The screenshot shows a code editor interface with a dark theme. On the left is the code file 'main.py' containing Python code for solving the problem. On the right is the 'Output' panel showing the execution results.

```
main.py
[ ] Share Run Output
3     dp = [[0]*n for _ in range(m)]
4     dp[i][j] = 1
5     ans = 0
6
7     for step in range(1, N+1):
8         temp = [[0]*n for _ in range(m)]
9         for x in range(m):
10            for y in range(n):
11                if dp[x][y] > 0:
12                    for dx, dy in [(-1,0),(1,0),(0,-1),(0,1)]:
13                        nx, ny = x + dx, y + dy
14                        if 0 <= nx < m and 0 <= ny < n:
15                            temp[nx][ny] = (temp[nx][ny] +
16                                dp[x][y]) % MOD
17                else:
18                    ans = (ans + dp[x][y]) % MOD
19
20
21 # Example:
22 m, n = 3, 3
23 N = 2
24 i, j = 0, 0
25 print(findWays(m, n, N, i, j)) # Output: 6
```

Output:

```
4
== Code Execution Successful ==
```

12.

## House Robber II (Circular Street)

### Aim

Find the maximum money that can be robbed without robbing two adjacent houses in a circle.

### Algorithm

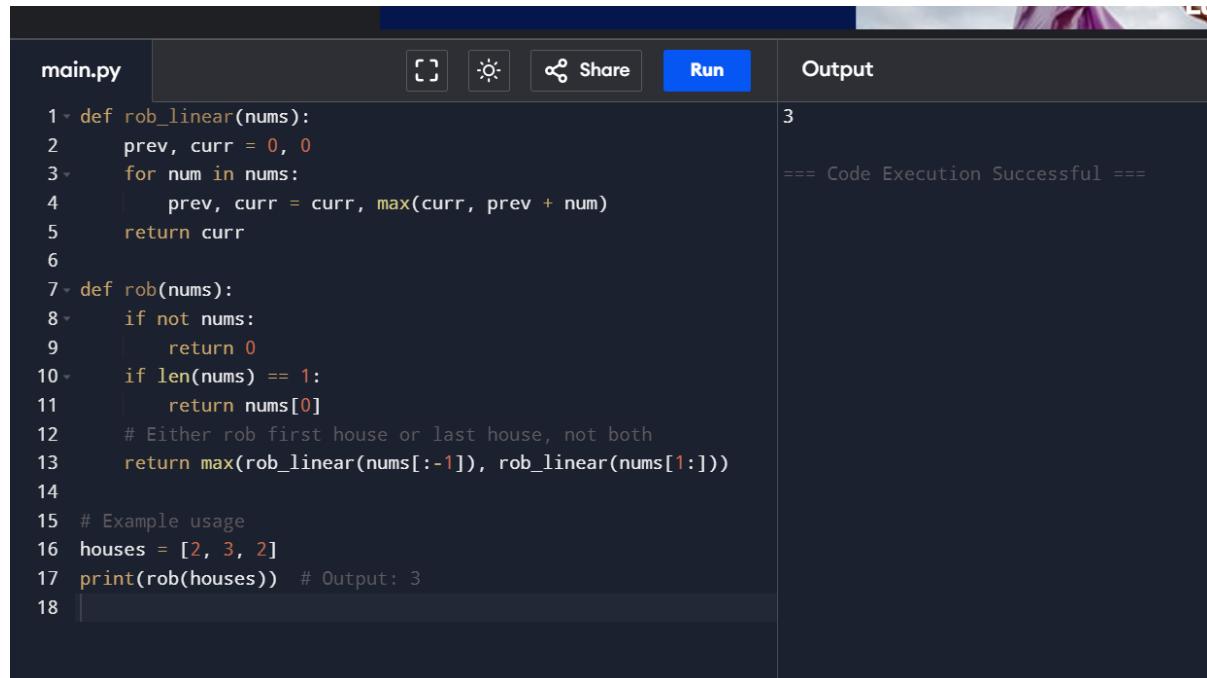
1. Normal House Robber (linear) uses DP.
2. For circular houses → two cases:
  - Exclude first house and rob from 1 to n-1.
  - Exclude last house and rob from 0 to n-2.
3. Take the maximum of both cases.

### Input

nums = [2, 3, 2]

### Output

3



The screenshot shows a code editor interface with a dark theme. On the left, there is a file named "main.py" containing the following Python code:

```
1 def rob_linear(nums):  
2     prev, curr = 0, 0  
3     for num in nums:  
4         prev, curr = curr, max(curr, prev + num)  
5     return curr  
6  
7 def rob(nums):  
8     if not nums:  
9         return 0  
10    if len(nums) == 1:  
11        return nums[0]  
12    # Either rob first house or last house, not both  
13    return max(rob_linear(nums[:-1]), rob_linear(nums[1:]))  
14  
15 # Example usage  
16 houses = [2, 3, 2]  
17 print(rob(houses)) # Output: 3  
18
```

At the top of the editor, there are several buttons: a copy icon, a share icon, a "Run" button, and a "Share" button. To the right of the code editor, there is an "Output" panel. The output shows the result of running the code: "3" followed by the message "==== Code Execution Successful ===".

13.

## Climbing Stairs

### Aim

Find distinct ways to climb n steps, moving either **1** or **2** steps at a time.

### Algorithm

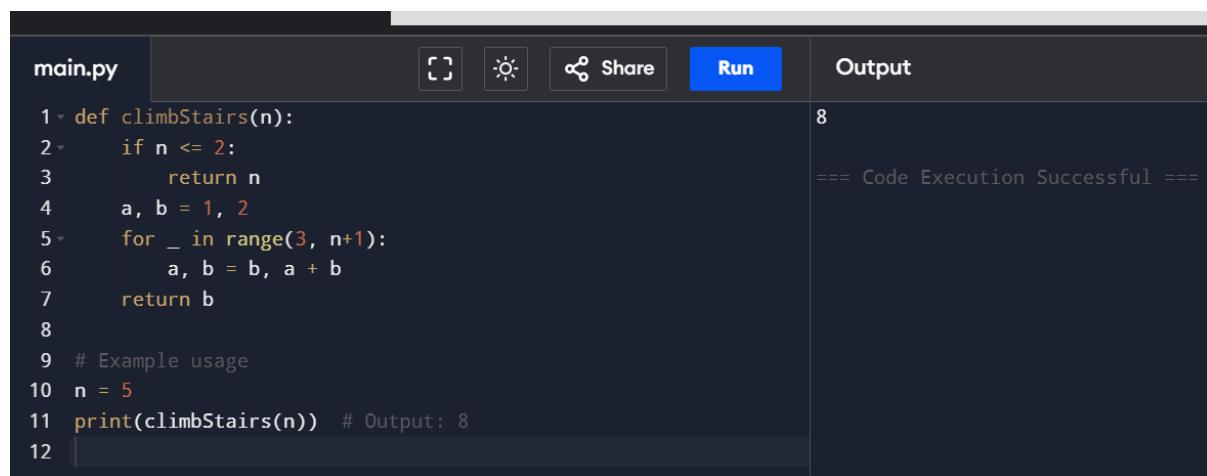
1. This is **Fibonacci sequence**:  
$$\text{ways}(n) = \text{ways}(n-1) + \text{ways}(n-2)$$
2. Use DP to avoid recomputation.

### Input

$n = 4$

### Output

5



The screenshot shows a code editor interface with a dark theme. On the left, there is a code editor window titled "main.py" containing the following Python code:

```
1 def climbStairs(n):
2     if n <= 2:
3         return n
4     a, b = 1, 2
5     for _ in range(3, n+1):
6         a, b = b, a + b
7     return b
8
9 # Example usage
10 n = 5
11 print(climbStairs(n)) # Output: 8
12
```

On the right, there is a "Run" button and an "Output" panel. The output panel shows the result of running the code: "8" and "==== Code Execution Successful ====". There are also icons for copy, paste, and share.

14.

## Robot Unique Paths

### Aim

Find the number of unique paths from **top-left** to **bottom-right** moving only **down** or **right**.

### Algorithm

1. Use **DP table**.

2. Each cell = paths from top + paths from left.

### Input

m=3, n=2

### Output

3

main.py				Run	Output
<pre>1 def uniquePaths(m, n): 2     dp = [[1]*n for _ in range(m)] 3     for i in range(1, m): 4         for j in range(1, n): 5             dp[i][j] = dp[i-1][j] + dp[i][j-1] 6     return dp[m-1][n-1] 7 8 # Example usage 9 m, n = 3, 3 10 print(uniquePaths(m, n)) # Output: 6 11</pre>				6 ==== Code Execution Successful ===	

15.

### Large Groups in String

#### Aim

Find intervals of large groups ( $\geq 3$  same consecutive characters).

#### Algorithm

1. Iterate through string with two pointers.
2. If group length  $\geq 3$ , store [start, end].

### Input

s = "abbxxxxzzy"

### Output

[[3, 6]]

The screenshot shows a code editor interface with a dark theme. On the left, the file 'main.py' is open, containing the following Python code:

```
1 def largeGroupPositions(s):
2     res = []
3     i = 0
4     while i < len(s):
5         start = i
6         while i + 1 < len(s) and s[i] == s[i+1]:
7             i += 1
8         if i - start + 1 >= 3:
9             res.append([start, i])
10        i += 1
11    return res
12
13 # Example usage
14 s = "aaabbbcccd"
15 print(largeGroupPositions(s)) # Output: [[0, 2], [3, 5], [6, 8]]
16
```

On the right, the 'Output' panel displays the results of running the code:

```
[[0, 2], [3, 5], [6, 8]]
== Code Execution Successful ==
```

16.

## Game of Life

### Aim

Compute the next state of Conway's Game of Life grid.

### Algorithm

1. For each cell, count live neighbors.
2. Apply rules:
  - $<2 \rightarrow \text{dies}$
  - $2 \text{ or } 3 \rightarrow \text{lives}$
  - $3 \rightarrow \text{dies}$
  - dead + 3 neighbors  $\rightarrow \text{live}$

### Input

```
[[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]]
```

The screenshot shows a code editor window with a dark theme. On the left, the file 'main.py' is open, containing the following code:

```
1 def gameOfLife(board):
2     m, n = len(board), len(board[0])
3
4     def count_live(i, j):
5         directions = [(-1,-1), (-1,0), (-1,1), (0,-1), (0,1), (1,-1),
6                       (1,0), (1,1)]
7         count = 0
8         for dx, dy in directions:
9             ni, nj = i + dx, j + dy
10            if 0 <= ni < m and 0 <= nj < n and abs(
11                board[ni][nj]) == 1:
12                count += 1
13
14    for i in range(m):
15        for j in range(n):
16            live_neighbors = count_live(i, j)
17            if board[i][j] == 1 and (live_neighbors < 2 or
18                                      live_neighbors > 3):
19                board[i][j] = -1 # Alive → Dead
20            if board[i][j] == 0 and live_neighbors == 3:
21                board[i][j] = 2 # Dead → Alive
22
23    for i in range(m):
24        for j in range(n):
25            if board[i][j] > 0:
```

On the right, the 'Output' tab is active, showing the result of running the code:

```
[[0, 0, 0], [1, 0, 1], [0, 1, 1], [0, 1, 0]]
== Code Execution Successful ==
```

17.

## Champagne Tower

### Aim

Find how full the query\_glass is after pouring poured cups.

### Algorithm

1. Use DP.
2. Each glass spills extra equally to next row (left and right).
3. Stop when reaching query\_row.

### Input

poured=2, query\_row=1, query\_glass=1

### Output

0.5

main.py		<b>Run</b>	<b>Output</b>
<pre>1 def champagneTower(poured, query_row, query_glass): 2     tower = [[0]*101 for _ in range(101)] 3     tower[0][0] = poured 4 5     for r in range(query_row + 1): 6         for c in range(r + 1): 7             if tower[r][c] &gt; 1: 8                 excess = (tower[r][c] - 1) / 2 9                 tower[r+1][c] += excess 10                tower[r+1][c+1] += excess 11                tower[r][c] = 1 12 13     return tower[query_row][query_glass] 14 15 # Example usage 16 poured = 10 17 query_row = 3 18 query_glass = 1 19 print(champagneTower(poured, query_row, query_glass)) # Output: 1</pre>	1 == Code Execution Successful ==		