11. Ball Out of Grid in N Steps

AIM:
To find the total number of ways a ball can move out of an **m x n grid** starting from a given cell (i, j) in **exactly N steps**.

ALGORITHM (DFS + Memoization):

1. Start at (i, j) with N steps remaining.
2. If the ball moves out of boundary → count as 1 way.
3. If no steps remain → count as 0.
4. Move the ball in all **4 directions** (up, down, left, right) recursively, reducing remaining steps by 1.
5. Store results in a **memoization dictionary** to avoid recomputation.
6. Sum the ways from all directions and return as the total count.

PYTHON CODE:

```
def ways(m,n,N,i,j,memo={}):
    if i<0 or i>=m or j<0 or j>=n: return 1
    if N==0: return 0
    if (i,j,N) in memo: return memo[(i,j,N)]
    memo[(i,j,N)] = ways(m,n,N-1,i+1,j,memo)+ways(m,n,N-1,i-1,j,memo)+ways(m,n,N-1,i,j+1,memo)+ways(m,n,N-1,i,j-1,memo)
    return memo[(i,j,N)]


m,n,N = map(int,input("Enter m n N: ").split())
i,j = map(int,input("Enter i j: ").split())
print("Ways:", ways(m,n,N,i,j))
```

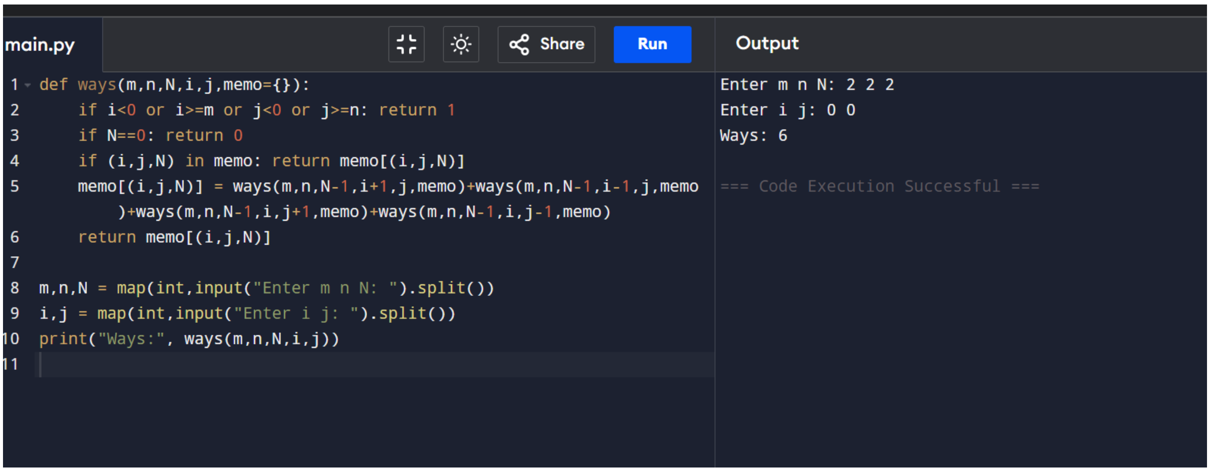
INPUT:

Enter m n N: 2 2 2

Enter i j: 0 0

OUTPUT:

Ways: 6



12. House Robber II (Circle Houses)

AIM:

To determine the **maximum money** a robber can steal from houses arranged in a **circle** without alerting the police (cannot rob two adjacent houses).

ALGORITHM (Dynamic Programming):

1. If there is only 1 house → rob it.
2. Divide the circle problem into **two linear cases**:
   - Rob houses **0 to n-2**
   - Rob houses **1 to n-1**
3. Use **DP formula** for linear houses: dp[i] = max(dp[i-1], dp[i-2]+nums[i]).
4. Take the **maximum** of the two cases → answer.

PYTHON CODE:

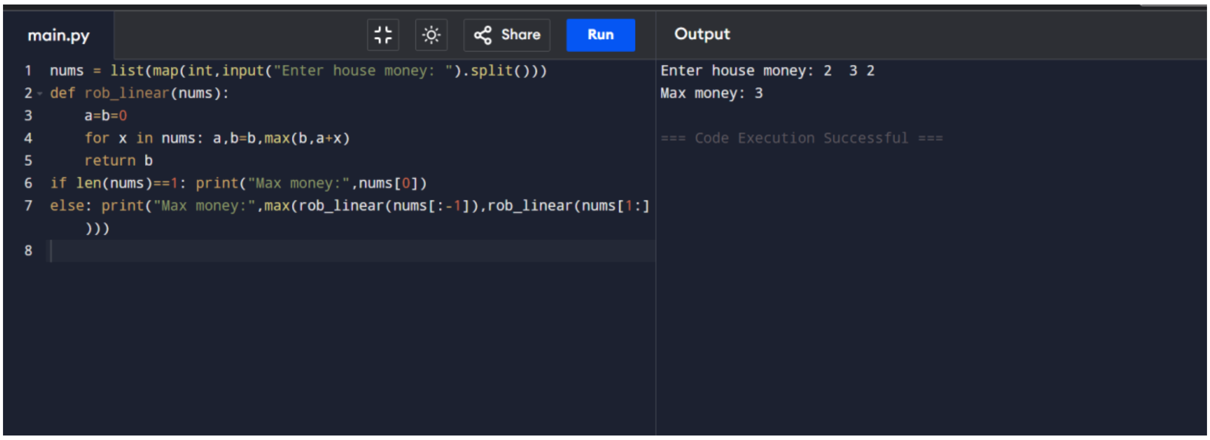
```
nums = list(map(int,input("Enter house money: ").split()))
def rob_linear(nums):
    a=b=0
    for x in nums: a,b=b,max(b,a+x)
    return b
if len(nums)==1: print("Max money:",nums[0])
else: print("Max money:",max(rob_linear(nums[:-1]),rob_linear(nums[1:])))
```

INPUT:

Enter house money: 2 3 2

OUTPUT:

Max money: 3



13. Climbing Stairs

AIM:
To find the number of **distinct ways** to reach the top of a staircase with n steps, moving either 1 or 2 steps at a time.

ALGORITHM (Dynamic Programming / Fibonacci):

1. Let ways[i] represent ways to reach step i.
2. Base cases: ways[0] = 1, ways[1] = 1.
3. Recurrence: ways[i] = ways[i-1] + ways[i-2].
4. Compute up to step n → ways[n] is the answer.

PYTHON CODE:

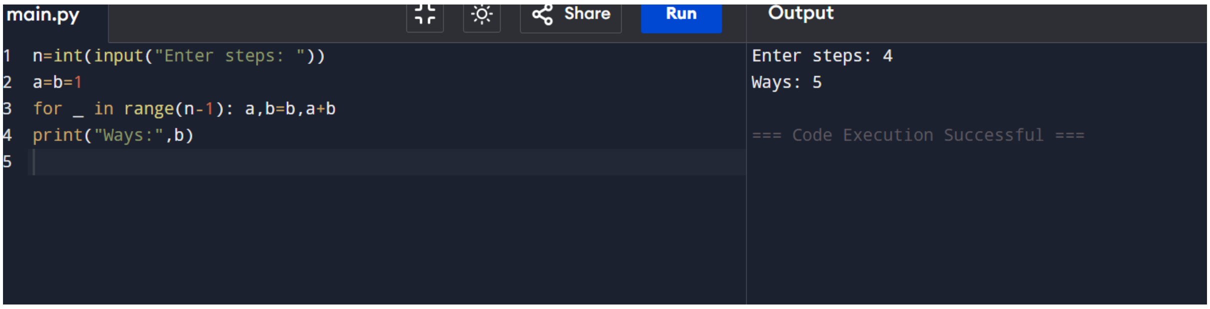
```
n=int(input("Enter steps: "))

a=b=1

for _ in range(n-1): a,b=b,a+b

print("Ways:",b)
```

INPUT:

Enter steps: 4

OUTPUT:

Ways: 5



14. Unique Paths in Grid (Robot)

AIM:
To count the total number of unique paths a robot can take from top-left to bottom-right in an m x n grid, moving only down or right.

ALGORITHM (Combinatorics / DP):

1. Total moves = (m-1) downs + (n-1) rights → total moves = m+n-2.
2. Choose positions for down moves → C(m+n-2, m-1) combinations.
3. Return this value as the number of unique paths.

PYTHON CODE:

```
from math import comb

m,n=map(int,input("Enter m n: ").split())

print("Unique paths:",comb(m+n-2,m-1))
```

INPUT:

Enter m n: 7 3

OUTPUT:

Unique paths: 28

15. Large Groups in String

AIM:

To identify intervals of all large groups (consecutive same characters of length ≥ 3) in a string.

ALGORITHM:

1. Initialize start = 0.
2. Traverse the string:
   - If current character ≠ next character or end of string:
     - If (i - start + 1) ≥ 3 → record interval [start, i].
     - Update start = i+1.
3. Return the list of intervals.

PYTHON CODE:

```
s=input("Enter string: ")

res=[];start=0

for i in range(len(s)):

   if i==len(s)-1 or s[i]!=s[i+1]:

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

      start=i+1

print("Large groups:",res)
```

INPUT:

Enter string: abbxxxxzzy

OUTPUT:

Large groups: [[3,6]]

16. Game of Life

AIM:
To compute the next state of a cellular automaton grid based on Conway's Game of Life rules.

ALGORITHM:

1. For each cell (i, j), count live neighbors in 8 directions.
2. Apply rules:
    - Live cell with <2 or >3 neighbors → dies (0).
    - Live cell with 2 or 3 neighbors → lives (1).
    - Dead cell with exactly 3 neighbors → becomes live (1).
3. Update the entire grid simultaneously.

PYTHON CODE:

```
m=int(input("Rows: "))
board=[list(map(int,input().split())) for _ in range(m)]
n=len(board[0])
dirs=[(-1,-1),(-1,0),(-1,1),(0,-1),(0,1),(1,-1),(1,0),(1,1)]
new=[[0]*n for _ in range(m)]
for i in range(m):
  for j in range(n):
    live=sum(0<=i+dx<m and 0<=j+dy<n and board[i+dx][j+dy] for dx,dy in dirs)
    if board[i][j]==1 and live in [2,3]: new[i][j]=1
    if board[i][j]==0 and live==3: new[i][j]=1
print("Next state:")
for row in new: print(row)
```

INPUT:

```
Rows: 4
0 1 0
0 0 1
1 1 1
0 0 0
```

OUTPUT:

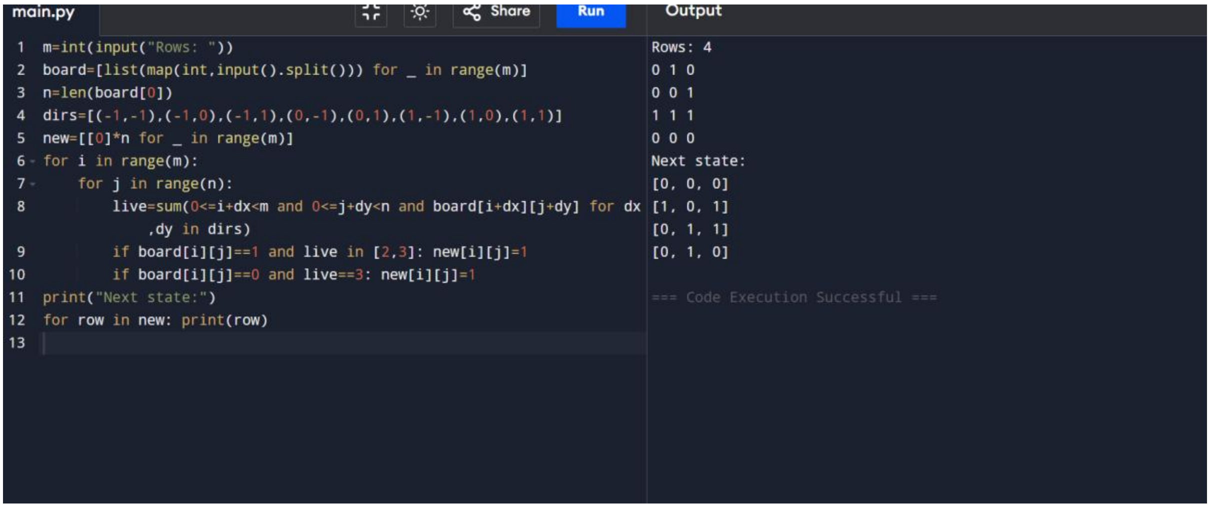
Next state:

[0, 0, 0]

[1, 0, 1]

[0, 1, 1]

[0, 1, 0]



17. Champagne Tower

AIM:
To determine how full a glass is in a pyramid of glasses after pouring a certain number of cups of champagne.

ALGORITHM (Simulation / DP):

1. Initialize a 2D array dp representing glasses, top glass gets poured cups.
2. Traverse row by row:
   - If dp[i][j] > 1, calculate excess = (dp[i][j]-1)/2.
   - Distribute excess equally to dp[i+1][j] and dp[i+1][j+1].
3. For queried glass (query_row, query_glass), return min(1, dp[query_row][query_glass]).

PYTHON CODE:

```
poured=int(input("Poured cups: "))

query_row=int(input("Query row: "))

query_glass=int(input("Query glass: "))

dp=[[0]*(query_row+2) for _ in range(query_row+2)]
```

```
dp[0][0]=poured
for r in range(query_row+1):
    for c in range(r+1):
        if dp[r][c]>1:
            excess=(dp[r][c]-1)/2
            dp[r+1][c]+=excess
            dp[r+1][c+1]+=excess
print("Glass fullness:",min(1,dp[query_row][query_glass]))
```

INPUT:

Poured cups: 1

Query row: 1

Query glass: 1

OUTPUT:

Glass fullness: 0.0