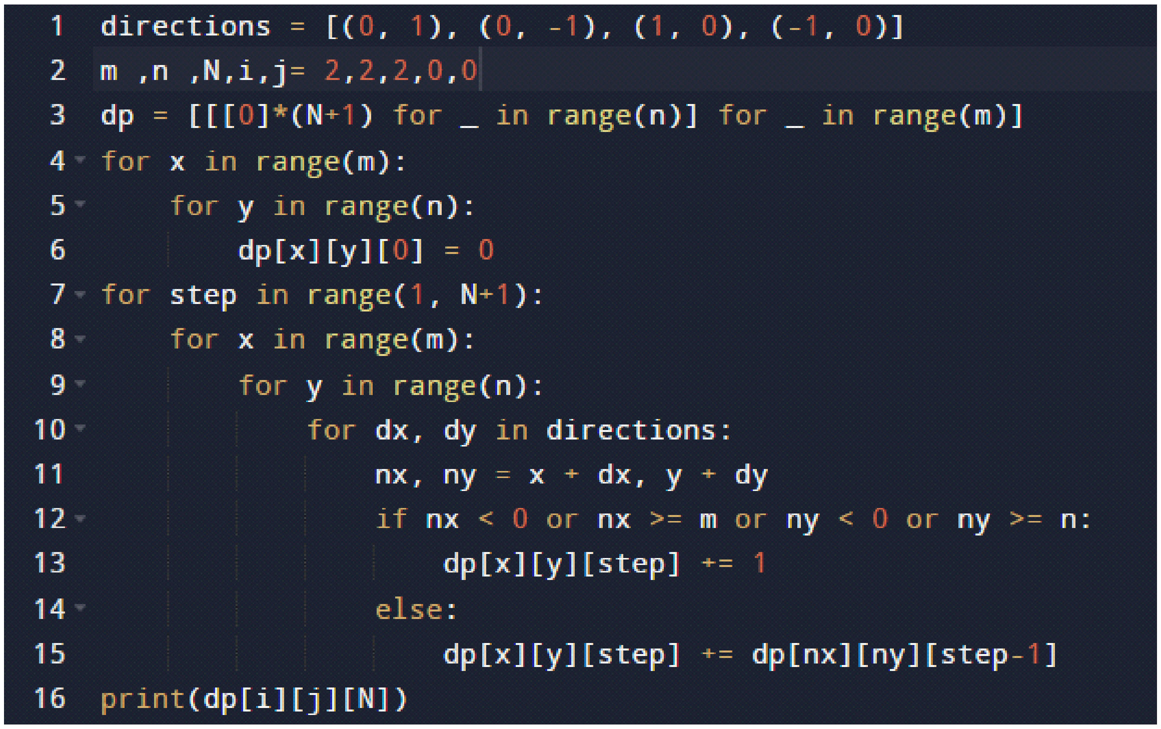
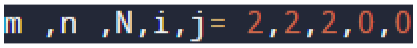
11. Given an m x n grid and a ball at a starting cell, find the number of ways to move the ball out of the grid boundary in exactly N steps.
Example:
· Input: m=2,n=2,N=2,i=0,j=0 · Output: 6
· Input: m=1,n=3,N=3,i=0,j=1 · Output: 12

AIM**:** the aim is to move the ball can move in four directions up down left and right
ALGORITHM:
STEP1**:** initialize the dp table with zeroes.
STEP2:get dp[start-x][start-y][0]=1 where (start_x,start-y) is starting cell
STEP3: for each cell(i,j)
PROGRAM:



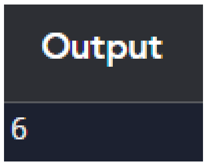
```
directions = [(0, 1), (0, -1), (1, 0), (-1, 0)]
m ,n ,N,i,j= 2,2,2,0,0
dp = [[[0]*(N+1) for _ in range(n)] for _ in range(m)]
for x in range(m):
    for y in range(n):
        dp[x][y][0] = 0
for step in range(1, N+1):
    for x in range(m):
        for y in range(n):
            for dx, dy in directions:
                nx, ny = x + dx, y + dy
                if nx < 0 or nx >= m or ny < 0 or ny >= n:
                    dp[x][y][step] += 1
                else:
                    dp[x][y][step] += dp[nx][ny][step-1]
print(dp[i][j][N])
```

INPUT:



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

OUTPUT:



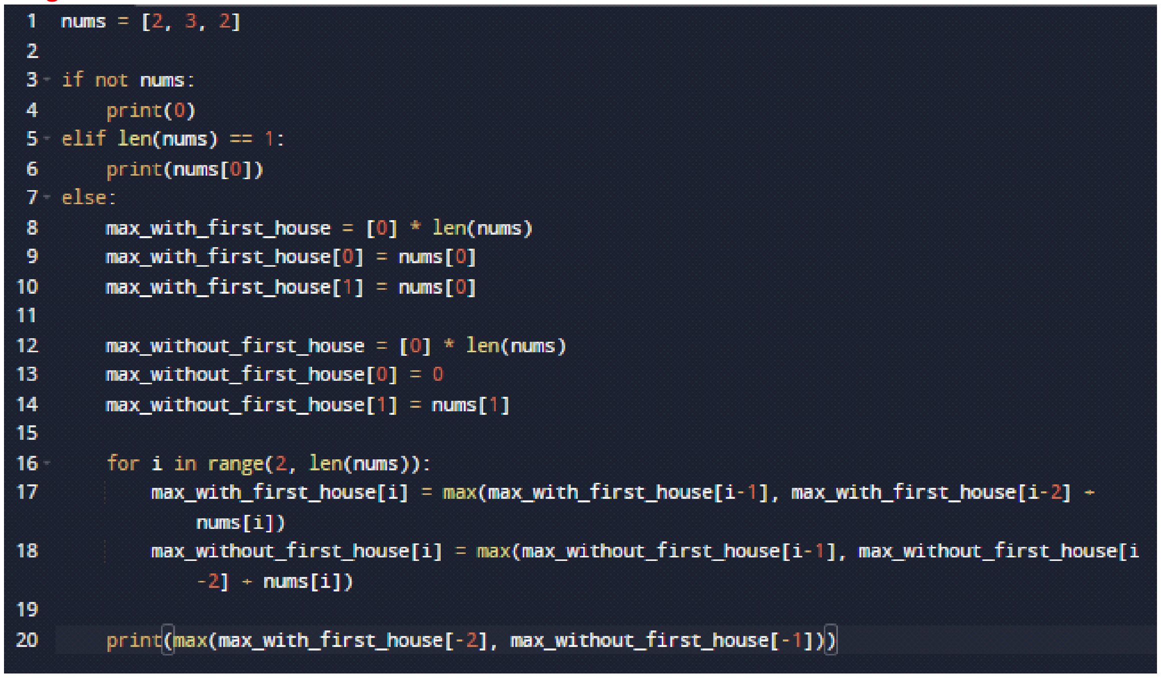
```
Output

6
```

TIMECOMPLEXITY:t(n)=O(m*n*N)
RESULT: the function 'num-ways-to-exist grid returns the number of ways to move the ball out the grid boundary in exactly N steps.

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

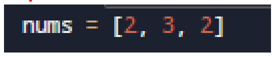
means the first house is the neighbor of the last one. Meanwhile, adjacent houses have security systems connected, and it will automatically contact the police if two adjacent houses were broken into on the same night.
Example:
Input : nums = [2, 3, 2]
Output : The maximum money you can rob without alerting the police is 3(robbing house 1).

Aim:The goal is to maximize the total amount of money stolen with bringing the security system.
Algorithm:
step1:intilize 'dp1" and 'dp2' with zeros.
step2:for each house'i' fro, 1 to 'n'.
step3:amount of money that can be stolen is the maxims of dp[n-1] and dp[n-1].
Program:



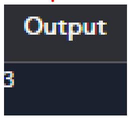
```
nums = [2, 3, 2]

if not nums:
    print(0)
elif len(nums) == 1:
    print(nums[0])
else:
    max_with_first_house = [0] * len(nums)
    max_with_first_house[0] = nums[0]
    max_with_first_house[1] = nums[0]

    max_without_first_house = [0] * len(nums)
    max_without_first_house[0] = 0
    max_without_first_house[1] = nums[1]

    for i in range(2, len(nums)):
        max_with_first_house[i] = max(max_with_first_house[i-1], max_with_first_house[i-2] +
            nums[i])
        max_without_first_house[i] = max(max_without_first_house[i-1], max_without_first_house[i
            -2] + nums[i])

    print(max(max_with_first_house[-2], max_without_first_house[-1]))
```

Input:



```
nums = [2, 3, 2]
```

Output:



```
Output
3
```

Time complexity:T(n)=O(n)
Result: the function 'rob house' returns the maximum amount of monry that si stolen without triggering the security system.

13. You are climbing a staircase. It takes n steps to reach the top. Each time you can either

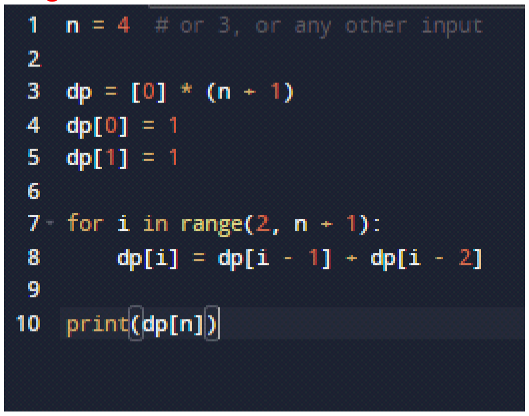
climb 1 or 2 steps. In how many distinct ways can you climb to the top?

Examples:

(i) Input: n=4 Output: 5

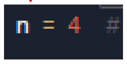
(ii) Input: n=3 Output: 3

Aim:The aim is to find the number of distincts ways to climb to topof stairscase with 4steps.
Algorithm:
step1:Let 'dp[p]' to the number of distints ways to climb 'i' the step.
step2:'dp[p]'=dp[i-1]+dp[i-2].
step3:Intilize dp[0]=1 & dp[1]=1.
step4:return dp[n].
Program:



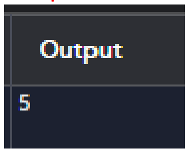
```python
n = 4  # or 3, or any other input

dp = [0] * (n + 1)
dp[0] = 1
dp[1] = 1

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

print(dp[n])
```

Input:



```
n = 4  #
```

Output:



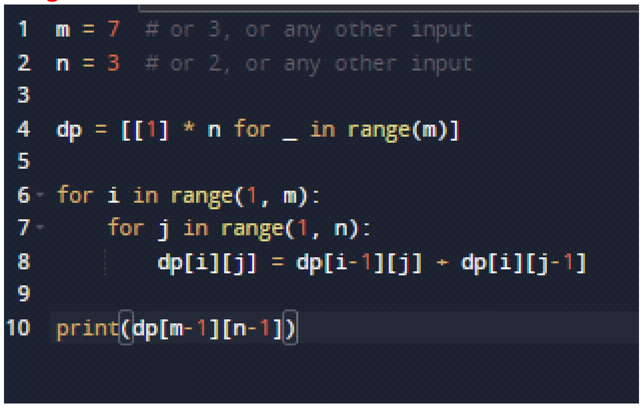
```
Output
5
```

Time complexity:T(n)=O(n)
Result: The number of distinct ways to climb to top of stairscase with 'n' steps.

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

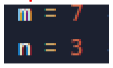
Aim:The aim is to find the number of unie part for robot to reach bottom right corner of m*n grid.
Algorithm:
step1:Let 'dp[p]' to the number of uniue parth to read cell.
step2:'dp[i][j]'=dp[i-1][j]+dp[i][j-1].

step3:intlize 'dp[0][j]=1' and dp[i][j].
step4:return dp(m-1)(n-1).
Program:



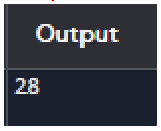
```
m = 7  # or 3, or any other input
n = 3  # or 2, or any other input

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]

print(dp[m-1][n-1])
```

Input:



```
m = 7
n = 3
```

Output:



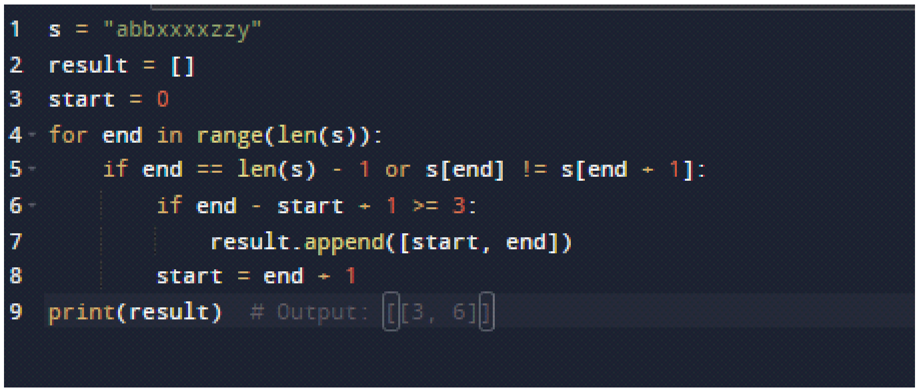
```
Output
28
```

Time complexity:T(n)=O(m*n)
Result: the number of   uniue part for robot to reach bottom right corner.

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.

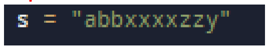
Example 1: Input: s = "abbxxxxzzy"
Output: [[3,6]]
Explanation: "xxxx" is the only large group with start index 3 and end index 6.

Aim: To return the interval of every large group sorted increasing order by sort index.
Algorithm:
step1:Intlize 'result=[]',start=0 and count=1.
step2:Iterate through 's'.
step3:after iterating check if 'count'.
step4:return 'result' sorted by index.
Program:



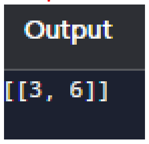
```python
s = "abbxxxxzzy"
result = []
start = 0
for end in range(len(s)):
    if end == len(s) - 1 or s[end] != s[end + 1]:
        if end - start + 1 >= 3:
            result.append([start, end])
        start = end + 1
print(result)  # Output: [[3, 6]]
```

Input:



```python
s = "abbxxxxzzy"
```

output:



```
Output
[[3, 6]]
```

Time complexity:T(n)=O(n).

Result:The function returns a list at internals of every large group sorted in ascending order.

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 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]]

Aim: The aim is given the current state of an m*n grid towards.

Algorithm:

step1:Create a new m*n grid to store the next state of board.

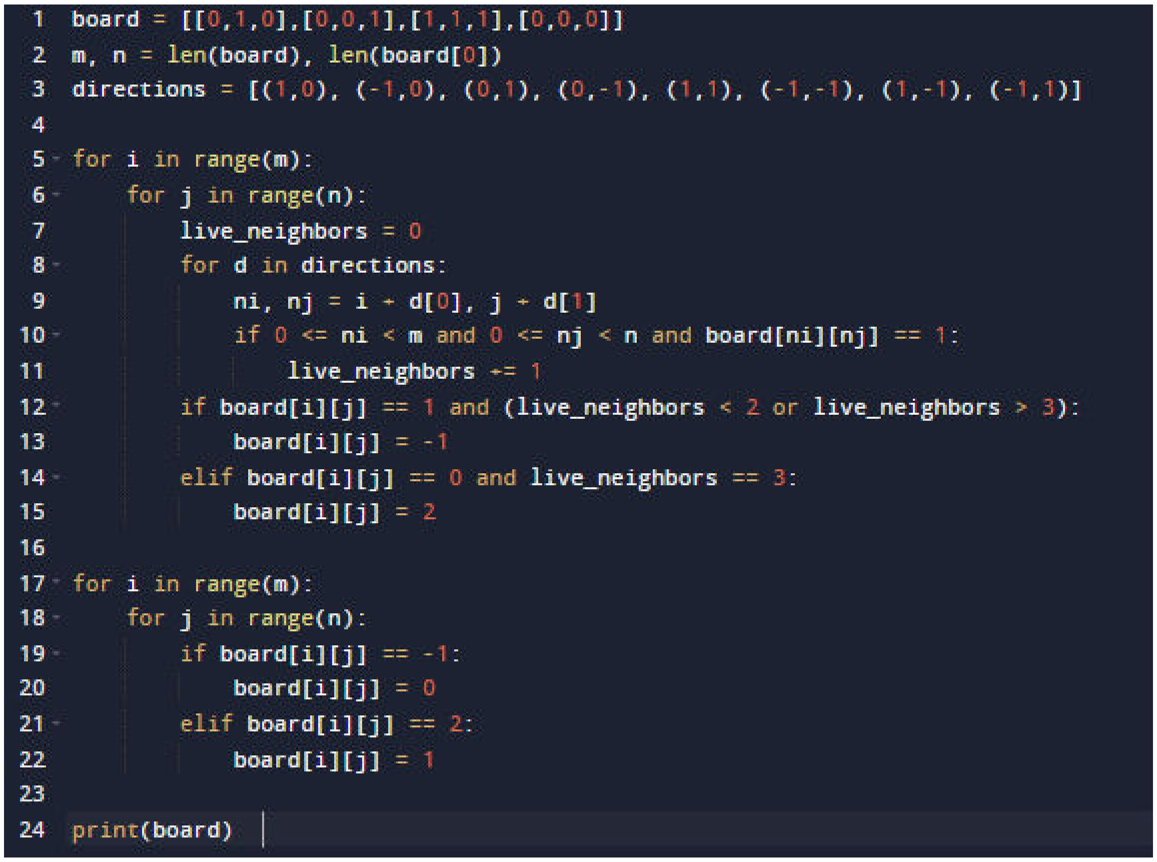
step2:Iterate through each cell in the current.

step3:count line neighbour and apply the rules.

step4:store the next state in the new grid.

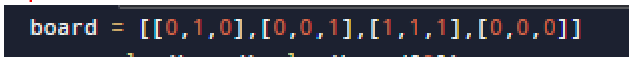
step5:return the next state as the next state of board.

Program:



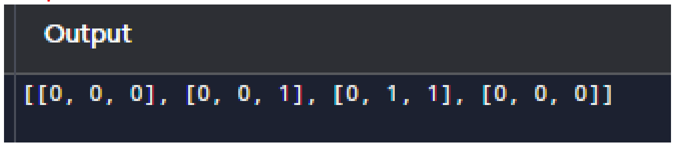
```python
board = [[0,1,0],[0,0,1],[1,1,1],[0,0,0]]
m, n = len(board), len(board[0])
directions = [(1,0), (-1,0), (0,1), (0,-1), (1,1), (-1,-1), (1,-1), (-1,1)]

for i in range(m):
    for j in range(n):
        live_neighbors = 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:
                live_neighbors += 1
        if board[i][j] == 1 and (live_neighbors < 2 or live_neighbors > 3):
            board[i][j] = -1
        elif board[i][j] == 0 and live_neighbors == 3:
            board[i][j] = 2

for i in range(m):
    for j in range(n):
        if board[i][j] == -1:
            board[i][j] = 0
        elif board[i][j] == 2:
            board[i][j] = 1

print(board)
```

Input:



```python
board = [[0,1,0],[0,0,1],[1,1,1],[0,0,0]]
```

output:



```
Output

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

Time complexity:T(n)=O(m*n).

Result:The program return returns the next state of board as new m*n grid where each cell has a value of O.

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

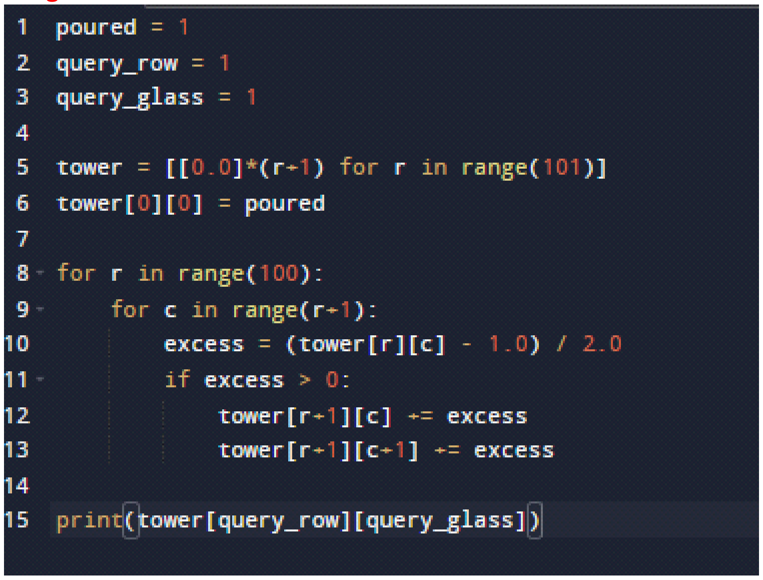
ith row is (both i and j are 0-indexed.)
Input: poured = 1,
query_row = 1, query_glass = 1
Output: 0.00000
Explanation: We poured 1 cup of champange to the top glass of the tower (which is indexed as (0, 0)). There will be no excess liquid so all the glasses under the top glass will remain empty.

Aim: The aim is to find the amount of champanze in the glass of ith row after pouring a certain number.

Algorithm:

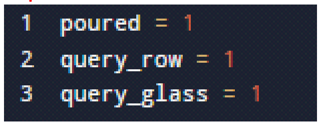
step1:Initiazation.
step2:pouring champanze.
step3:for each row'i' fro, 0 to 100.
step4:Query return 'min[glass [query_row,[query-glass][i,O]

Program:



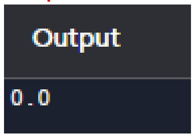
```python
poured = 1
query_row = 1
query_glass = 1

tower = [[0.0]*(r+1) for r in range(101)]
tower[0][0] = poured

for r in range(100):
    for c in range(r+1):
        excess = (tower[r][c] - 1.0) / 2.0
        if excess > 0:
            tower[r+1][c] += excess
            tower[r+1][c+1] += excess

print(tower[query_row][query_glass])
```

Input:



```python
poured = 1
query_row = 1
query_glass = 1
```

output:



```
Output
0.0
```

Time complexity:T(n)=O(n^2).

Result:The function 'champinaze tower' return the amount of champinze in jth class of ith row after powring certain number of rows.