Counting Rooms

You are given a map of a building, and your task is to count the number of its rooms.
The size of the map is n x m squares, and each square is either floor or wall.
You can walk left, right, up, and down through the floor squares.

Input

The first input line has two integers n and m: the height and width of the map.

Then there are n lines of m characters describing the map.
Each character is either . (floor) or # (wall).

Output

Print one integer: the number of rooms.

Constraints

1 <= n,m <= 1000

Example

Input:

```
5 8
########
#..#...#
####.#.#
#..#...#
########
```

Output:

```
3
```

Problem Breakdown

Lets understand the problem first

You're given a 2D grid of . (floor) and # (walls).

You can move up, down, left, or right, but only through . cells.

Your task is to count how many separate rooms exist in the building.

```
########
#..#...#
####.#.#
#..#...#
########
```

Why is this a graph problem

Any problem where entities are connected and you need to explore reachability between them can usually be represented as a graph.

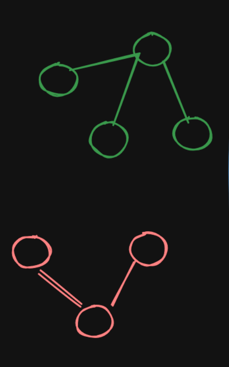
Each cell in the grid becomes a node, and each valid move to an adjacent cell is an edge.

This is called implicit graph construction — we don't store an adjacency list, we use the grid itself to traverse.

How do we know it's a connected components problem?

Because we are grouping tiles that are all connected.
Each such group is a connected component — and each component = one room.

What are connected components?



A connected component in a graph is a group of nodes where:

Every node is reachable from every other node in the group.

There is no connection to nodes outside the group.

How do we solve connected components problems?

We use graph traversal (DFS or BFS):

Start traversal from an unvisited node.

Mark all nodes in that component.

Each new traversal = new component.

Graph Traversal.

DFS	BFS
Uses recursion to go deep into the connected tiles.	Uses a queue to explore level by level.
Mark each tile as visited.	Avoids recursion depth issues in large grids.
Count the number of times DFS starts → this is the number of rooms.	

Algorithm

```cpp
#include <bits/stdc++.h>
using namespace std;

#define endl '\n'

int n, m;
vector<vector<char>> grid;
vector<vector<bool>> visited;

const vector<pair<int, int>> directions = {{-1, 0}, {0, -1}, {1, 0}, {0, 1}};

void dfs(int x, int y) {
    visited[x][y] = true;

    for (auto [dx, dy] : directions) {
        int nextX = x + dx;
        int nextY = y + dy;

        // Check if the next cell is within bounds, is a floor ('.'), and not visited
        if (nextX >= 0 && nextX < n && nextY >= 0 && nextY < m &&
            grid[nextX][nextY] == '.' && !visited[nextX][nextY]) {
            dfs(nextX, nextY);
        }
    }
}

void solve() {
    cin >> n >> m;

    grid.resize(n, vector<char>(m));
    visited.assign(n, vector<bool>(m, false));

    for (int i = 0; i < n; ++i)
        for (int j = 0; j < m; ++j)
            cin >> grid[i][j];

    int rooms = 0;

    for (int i = 0; i < n; ++i)
        for (int j = 0; j < m; ++j)
            if (grid[i][j] == '.' && !visited[i][j]) {
                bfs(i, j);
                rooms++;
            }

    cout << rooms << endl;
}

int main() {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);
    solve();
    return 0;
}
```

Read the grid

For each cell in the grid:
    a. If it's a floor ('.') and not visited:
        i. Start DFS/BFS from that cell.
        ii. Mark all connected floor tiles as visited.
        iii. Increment room counter.

Time Complexity

$O(N \times M)$

we visit every cell at most once.

Space Complexity

$O(N * M)$ for visited grid.

DFS: O(stack size) up to $N * M$

BFS: O(queue size) up to $N * M$