Walls and Gates

You are given a *m x n* 2D grid initialized with these three possible values.

1. `-1` - A wall or an obstacle.
2. `0` - A gate.
3. `INF` - Infinity means an empty room. We use the value `2^31 - 1 = 2147483647` to represent `INF` as you may assume that the distance to a gate is less than `2147483647`.

Fill each empty room with the distance to its *nearest* gate. If it is impossible to reach a gate, it should be filled with `INF`.

For example, given the 2D grid:

```
INF  -1  0  INF
INF INF INF  -1
INF  -1 INF  -1
  0  -1 INF INF
```

After running your function, the 2D grid should be:

```
  3  -1   0   1
  2   2   1  -1
  1  -1   2  -1
  0  -1   3   4
```

Solution 1

Push all gates into queue first. Then for each gate update its neighbor cells and push them to the queue.

Repeating above steps until there is nothing left in the queue.

```
public class Solution {
    public void wallsAndGates(int[][] rooms) {
        if (rooms.length == 0 || rooms[0].length == 0) return;
        Queue<int[]> queue = new LinkedList<>();
        for (int i = 0; i < rooms.length; i++) {
            for (int j = 0; j < rooms[0].length; j++) {
                if (rooms[i][j] == 0) queue.add(new int[]{i, j});
            }
        }
        while (!queue.isEmpty()) {
            int[] top = queue.remove();
            int row = top[0], col = top[1];
            if (row > 0 && rooms[row - 1][col] == Integer.MAX_VALUE) {
                rooms[row - 1][col] = rooms[row][col] + 1;
                queue.add(new int[]{row - 1, col});
            }
            if (row < rooms.length - 1 && rooms[row + 1][col] == Integer.MAX_VALU
E) {
                rooms[row + 1][col] = rooms[row][col] + 1;
                queue.add(new int[]{row + 1, col});
            }
            if (col > 0 && rooms[row][col - 1] == Integer.MAX_VALUE) {
                rooms[row][col - 1] = rooms[row][col] + 1;
                queue.add(new int[]{row, col - 1});
            }
            if (col < rooms[0].length - 1 && rooms[row][col + 1] == Integer.MAX_V
ALUE) {
                rooms[row][col + 1] = rooms[row][col] + 1;
                queue.add(new int[]{row, col + 1});
            }
        }
    }
}
```

written by chase1991 original link here

Solution 2

```
public class Solution {
int[][] dir ={{0,1},{0,-1},{1,0},{-1,0}};
public void wallsAndGates(int[][] rooms) {
    for(int i=0;i<rooms.length;i++){
        for(int j=0;j<rooms[0].length;j++){
            if(rooms[i][j]==0)
                bfs(rooms,i,j);
        }
    }
}
public void bfs(int[][] rooms,int i,int j){
    for(int[] d:dir){
        if(i+d[0]>=0 && i+d[0]<rooms.length && j+d[1]>=0 && j+d[1]<rooms[0].lengt
h && rooms[i+d[0]][j+d[1]]>rooms[i][j]+1){
            rooms[i+d[0]][j+d[1]]=rooms[i][j]+1;
            bfs(rooms,i+d[0],j+d[1]);
        }
    }
}
```

}

written by DREAM123 original link here

Solution 3

```
void wallsAndGates(vector<vector<int>>& rooms) {
    const int row = rooms.size();
    if (0 == row) return;
    const int col = rooms[0].size();
    queue<pair<int, int>> canReach;  // save all element reachable
    vector<pair<int, int>> dirs = {{1, 0}, {0, 1}, {-1, 0}, {0, -1}}; // four dir
ections for each reachable
    for(int i = 0; i < row; i++){
        for(int j = 0; j < col; j++){
            if(0 == rooms[i][j])
                canReach.emplace(i, j);
        }
    }
    while(!canReach.empty()){
        int r = canReach.front().first, c = canReach.front().second;
        canReach.pop();
        for (auto dir : dirs) {
            int x = r + dir.first,  y = c + dir.second;
            // if x y out of range or it is obstasle, or has small distance aread
y
            if (x < 0 || y < 0 || x >= row || y >= col || rooms[x][y] <= rooms[r]
[c]+1) continue;
            rooms[x][y] = rooms[r][c] + 1;
            canReach.emplace(x, y);
        }
    }
}
```

written by lchen77 original link here