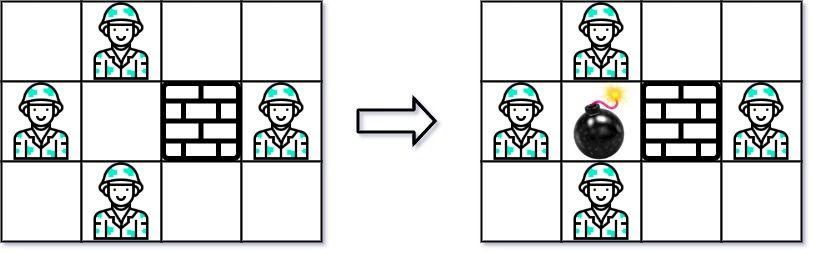
<https://leetcode.ca/all/361.html>

Given a 2D grid, each cell is either a wall 'W', an enemy 'E' or empty '0' (the number zero), return the maximum enemies you can kill using one bomb.

The bomb kills all the enemies in the same row and column from the planted point until it hits the wall since the wall is too strong to be destroyed.

Note: You can only put the bomb at an empty cell.

**Example 1:**



Input: [["0","E","0","0"],

["E","0","W","E"],

["0","E","0","0"]]

Output: 3

Explanation: For the given grid,

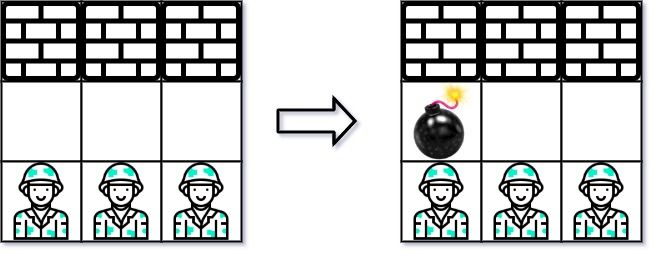
0 E 0 0

E 0 W E

0 E 0 0

Placing a bomb at (1,1) kills 3 enemies.

**Example 2:**



Input: grid = [["W","W","W"],["0","0","0"],["E","E","E"]]

Output: 1

**Constraints:**

m == grid.length

n == grid[i].length

1 <= m, n <= 500

grid[i][j] is either 'W', 'E', or '0'.

**Attempt 1: 2023-03-03**

**Solution 1: Brute Force (180 min)**

**Refer to**

<https://www.cnblogs.com/grandyang/p/5599289.html>

在论坛里看到了史蒂芬大神提出的另一种解法，感觉挺巧妙，就搬了过来。这种解法比较省空间，写法也比较简洁，需要一个 rowCnt 变量，用来记录到下一个墙之前的敌人个数。还需要一个数组 colCnt，其中 colCnt[j] 表示第j列到下一个墙之前的敌人个数。算法思路是遍历整个数组 grid，对于一个位置 grid[i][j]，对于水平方向，如果当前位置是开头一个或者前面一个是墙壁，开始从当前位置往后遍历，遍历到末尾或者墙的位置停止，计算敌人个数。对于竖直方向也是同样，如果当前位置是开头一个或者上面一个是墙壁，开始从当前位置向下遍历，遍历到末尾或者墙的位置停止，计算敌人个数。可能会有人有疑问，为啥 rowCnt 就可以用一个变量，而 colCnt 就需要用一个数组呢，为啥 colCnt 不能也用一个变量呢？原因是由遍历顺序决定的，这里是逐行遍历的，在每行的开头就统计了该行的敌人总数，所以再该行遍历没必要用数组，但是每次移动时就会换到不同的列，总不能没换个列就重新统计一遍吧，所以就在第一行时一起统计了存到数组中供后来使用。有了水平方向和竖直方向敌人的个数，那么如果当前位置是0，表示可以放炸弹，更新结果 res 即可

<https://medium.com/@rebeccahezhang/361-bomb-enemy-1b4b36d5a47a>

First naive approach: just go through each location. If it’s 0, count ‘E’ in current row and column between ‘W’. If it’s E, or W, do nothing and move forward.

The problem here is: while we scan each element in the same row, we calculate how many ‘E’ in this row n times (assume n is the number of columns). To save time, we simply store the calculation in a variable every time we scan the first element in a new row. But be careful about the ‘W’. After we hit the ‘W’, we need to recount ‘E’ and update the variable.

The same approach applies to each column. Since we are scanning through each row, we jump from one column to another column. So we need an array to store ‘E’ counts for each column.

With those two auxiliary storages, we can just update the max ‘E’ counts when we encounter ‘0’ every time.

The code looks like this (Thanks to [StefanPochmann](https://leetcode.com/stefanpochmann)’s solution)

import java.util.\*;

public class Solution {

public int maxKilledEnemies(char[][] grid) {

// iterate through the matrx by i(row), j(column)

// if it's '0', update the max so far by compare the current max and

// (enemies number in current row + enemies number in current column)

// Becareful the grid.length == 0 should be the first thing to check

// in case of array out of bound exception happens on grid[0].length

if (grid == null || grid.length == 0 || grid[0].length == 0) {

return 0;

}

int m = grid.length;

int n = grid[0].length;

int max = 0;

int rowCount = 0;

int[] colCount = new int[n];

for (int i = 0; i < m; i++) {

for (int j = 0; j < n; j++) {

// start from first row, count the enemies in the current row between two walls

// store it to avoid recompute

if (j == 0 || grid[i][j-1] == 'W') {

rowCount = 0;

for (int k = j; k < n && grid[i][k] != 'W'; k++) {

if (grid[i][k] == 'E') {

rowCount++;

}

}

}

// start from column, count the enemies in the current col between two walls

if (i == 0 || grid[i-1][j] == 'W') {

colCount[j] = 0;

for (int k = i; k < m && grid[k][j] != 'W'; k++) {

if (grid[k][j] == 'E') {

colCount[j]++;

}

}

}

// if this is a position to place the bomb, get the current max

if (grid[i][j] == '0') {

max = Math.max(max, rowCount + colCount[j]);

}

}

}

return max;

}

public static void main(String[] args) {

Solution so = new Solution();

char[][] grid = new char[][]{{'0','E','0','0'},{'E','0','W','E'},{'0','E','0','0'}};

int result = so.maxKilledEnemies(grid);

System.out.println(result);

}

}

**There is one critical point as why "if (j == 0 || grid[i][j-1] == 'W')" is correct but "if (j == 0 || grid[i][j] == 'W')" is wrong ?**

**Below is the conversation below chatGPT**

**Now if i change correct if condition and its logic as below**

if (j == 0 || grid[i][j-1] == 'W') {

rowCount = 0;

for (int k = j; k < n && grid[i][k] != 'W'; k++) {

if (grid[i][k] == 'E') {

rowCount++;

}

}

}

**into below wrong one as**

if (j == 0 || grid[i][j] == 'W') {

rowCount = 0;

for (int k = j + 1; k < n && grid[i][k] != 'W'; k++) {

if (grid[i][k] == 'E') {

rowCount++;

}

}

}

**The full wrong method code below, can you tell me why it is wrong after doing above change ?**

public int maxKilledEnemies1(char[][] grid) {

// iterate through the matrx by i(row), j(column)

// if it's '0', update the max so far by compare the current max and

// (enemies number in current row + enemies number in current column)

// Becareful the grid.length == 0 should be the first thing to check

// in case of array out of bound exception happens on grid[0].length

if (grid == null || grid.length == 0 || grid[0].length == 0) {

return 0;

}

int m = grid.length;

int n = grid[0].length;

int max = 0;

int rowCount = 0;

int[] colCount = new int[n];

for (int i = 0; i < m; i++) {

for (int j = 0; j < n; j++) {

// start from first row, count the enemies in the current row between two walls

// store it to avoid recompute

if (j == 0 || grid[i][j] == 'W') {

rowCount = 0;

for (int k = j + 1; k < n && grid[i][k] != 'W'; k++) {

if (grid[i][k] == 'E') {

rowCount++;

}

}

}

// start from column, count the enemies in the current col between two walls

if (i == 0 || grid[i-1][j] == 'W') {

colCount[j] = 0;

for (int k = i; k < m && grid[k][j] != 'W'; k++) {

if (grid[k][j] == 'E') {

colCount[j]++;

}

}

}

// if this is a position to place the bomb, get the current max

if (grid[i][j] == '0') {

max = Math.max(max, rowCount + colCount[j]);

}

}

}

return max;

}

**Answer from chatGPT:**

The change you made to the condition and logic in the incorrect version is problematic because it alters the way you calculate **rowCount**. Let's analyze the changes and understand why they are incorrect:

In the correct version:

if (j == 0 || grid[i][j-1] == 'W') {

rowCount = 0;

for (int k = j; k < n && grid[i][k] != 'W'; k++) {

if (grid[i][k] == 'E') {

rowCount++;

}

}

}

This correctly checks if the current cell is at the beginning of a row (**j == 0**) or if the cell to the left is a wall (**grid[i][j-1] == 'W'**). It then counts the number of enemies in the current row from the current cell until it encounters a wall.

Now, in the incorrect version:

if (j == 0 || grid[i][j] == 'W') {

rowCount = 0;

for (int k = j + 1; k < n && grid[i][k] != 'W'; k++) {

if (grid[i][k] == 'E') {

rowCount++;

}

}

}

The condition has changed to **grid[i][j] == 'W'**, which means it checks if the current cell is a wall, not the cell to the left. **Additionally, the loop is now starting from j + 1, which means it starts counting enemies from the cell to the right of the current cell, it will miss if the current cell is enemy.**

This change is incorrect because it doesn't properly identify the beginning of a row, and it starts counting enemies from the wrong position. As a result, the **rowCount** value will be incorrect, leading to incorrect calculations for the total number of enemies in the row when updating **max**.

To fix this, it's crucial to check the condition related to the left cell (**grid[i][j-1] == 'W'**) rather than the current cell (**grid[i][j] == 'W'**). The original condition ensures that you correctly identify the start of a row before counting enemies in that row.

This can be test out by input as below:

public static void main(String[] args) {

Solution so = new Solution();

char[][] grid = new char[][]{{'0','E','0','0'},{'E','0','W','E'},{'0','E','0','0'}};

int result = so.maxKilledEnemies(grid);

System.out.println(result);

}

When check for 2nd row as {'E','0','W','E'}, if we go with wrong logic above, we will miss calculate the first char as 'E' at the beginning of 2nd row, because we will accidently skip it since k start from wrong index as 'j + 1' rather than correct index as 'j', the start of the row shift 1 position to right because of '+ 1' on j