<https://leetcode.com/problems/minesweeper/description/>

Let's play the minesweeper game ([Wikipedia](https://en.wikipedia.org/wiki/Minesweeper_(video_game)), [online game](http://minesweeperonline.com/))!

You are given an m x n char matrix board representing the game board where:

'M' represents an unrevealed mine,

'E' represents an unrevealed empty square,

'B' represents a revealed blank square that has no adjacent mines (i.e., above, below, left, right, and all 4 diagonals),

digit ('1' to '8') represents how many mines are adjacent to this revealed square, and

'X' represents a revealed mine.

You are also given an integer array click where click = [clickr, clickc] represents the next click position among all the unrevealed squares ('M' or 'E').

Return *the board after revealing this position according to the following rules*:

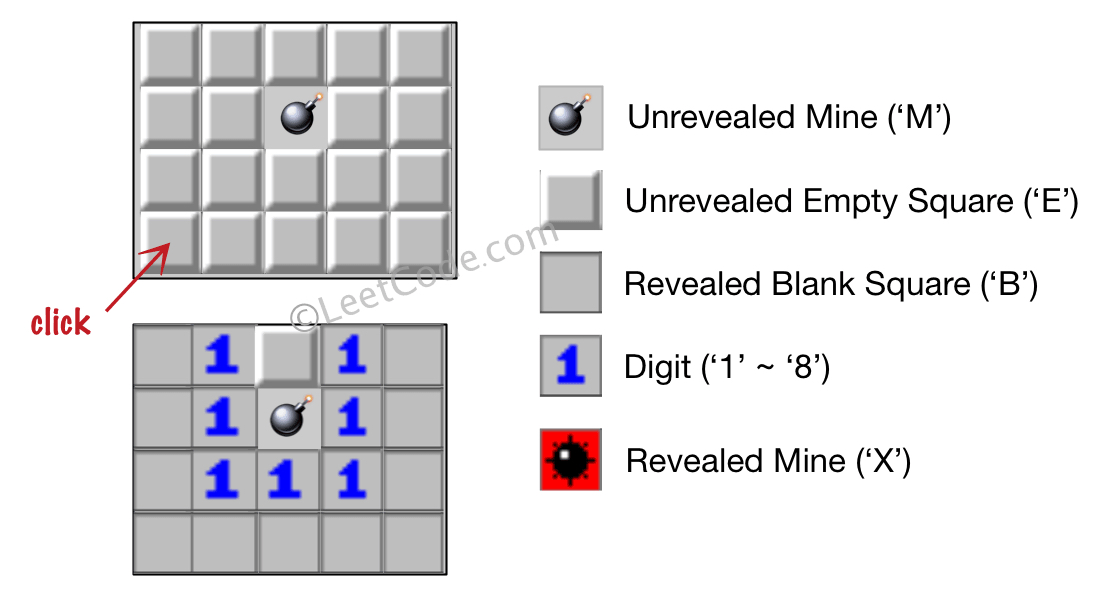
If a mine 'M' is revealed, then the game is over. You should change it to 'X'.

If an empty square 'E' with no adjacent mines is revealed, then change it to a revealed blank 'B' and all of its adjacent unrevealed squares should be revealed recursively.

If an empty square 'E' with at least one adjacent mine is revealed, then change it to a digit ('1' to '8') representing the number of adjacent mines.

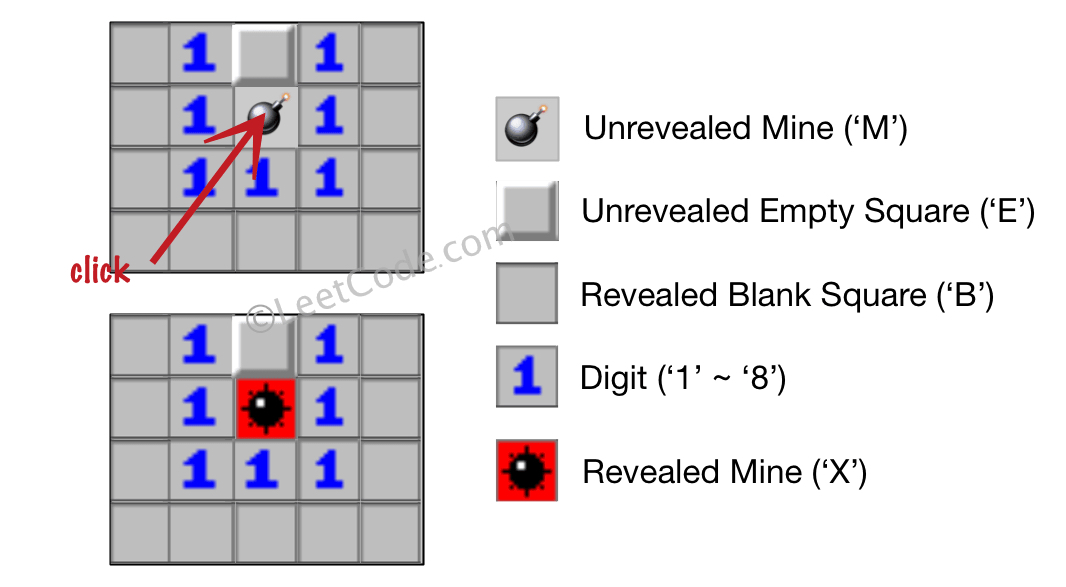
Return the board when no more squares will be revealed.

**Example 1:**



**Input:** board = [["E","E","E","E","E"],["E","E","M","E","E"],["E","E","E","E","E"],["E","E","E","E","E"]], click = [3,0]**Output:** [["B","1","E","1","B"],["B","1","M","1","B"],["B","1","1","1","B"],["B","B","B","B","B"]]

**Example 2:**



**Input:** board = [["B","1","E","1","B"],["B","1","M","1","B"],["B","1","1","1","B"],["B","B","B","B","B"]], click = [1,2]**Output:** [["B","1","E","1","B"],["B","1","X","1","B"],["B","1","1","1","B"],["B","B","B","B","B"]]

**Constraints:**

m == board.length

n == board[i].length

1 <= m, n <= 50

board[i][j] is either 'M', 'E', 'B', or a digit from '1' to '8'.

click.length == 2

0 <= clickr < m

0 <= clickc < n

board[clickr][clickc] is either 'M' or 'E'.

**Attempt 1: 2025-09-30**

**Solution 1: DFS (60 min)**

class Solution {

    int[] dx = new int[] {0, 1, 1, 1, 0, -1, -1, -1};

    int[] dy = new int[] {1, 1, 0, -1, -1, -1, 0, 1};

    public char[][] updateBoard(char[][] board, int[] click) {

        int i = click[0];

        int j = click[1];

        if(board[i][j] == 'M') {

            board[i][j] = 'X';

            return board;

        }

        int m = board.length;

        int n = board[0].length;

        helper(board, m, n, i, j);

        return board;

    }

    private void helper(char[][] board, int m, int n, int i, int j) {

        if(i < 0 || i >= m || j < 0 || j >= n || board[i][j] != 'E') {

            return;

        }

        int mineCount = countMines(board, m, n, i, j);

        if(mineCount > 0) {

            board[i][j] = (char) (mineCount + '0');

        } else {

            board[i][j] = 'B';

            for(int k = 0; k < 8; k++) {

                helper(board, m, n, i + dx[k], j + dy[k]);

            }

        }

    }

    private int countMines(char[][] board, int m, int n, int i, int j) {

        int count = 0;

        for (int k = 0; k < 8; k++) {

            int x = i + dx[k];

            int y = j + dy[k];

                count++;

            }

        }

        return count;

    }

}

Time Complexity: O(m \* n)

Space Complexity: O(m \* n), worst-case recursion depth (a long snake-shaped board)

**Refer to Kimi**

Below is a “build-it-yourself” walk-through that shows exactly how the DFS part of the solution is discovered and why every line is there.

We start with the rules, translate each rule into code shape, and finally stitch the shapes together into the compact DFS you saw.

Restate the problem in plain English

Board cells can be:

'M' – unrevealed mine

'E' – unrevealed empty

'B' – revealed blank (no adjacent mines)

'1'..'8' – revealed digit (that many adjacent mines)

'X' – revealed mine (only when you click on it)

You click (r,c).

If it is 'M' → turn it into 'X' and stop.

If it is 'E' → reveal it according to minesweeper rules, possibly revealing a whole region.

Translate the rules into algorithmic steps

STEP-0 Handle the mine-click special case.

STEP-1 Count how many mines are in the 8 neighbour cells.

STEP-2 If count > 0 → write the digit and stop.

STEP-3 If count == 0 → write 'B' and repeat STEP-1..3 for every neighbour that is still 'E'.

STEP-3 is literally the DFS (or BFS) “flood-fill”.

Skeleton of the DFS function

We need a helper that, given a coordinate (r,c):

returns immediately if out of bounds or not 'E' (already visited or mine).

counts adjacent mines.

either writes a digit and returns, or writes 'B' and recurses.

Draft:

dfs(r,c):

if (r,c) invalid or board[r][c] != 'E' → return

mines = count(r,c)

if mines > 0

board[r][c] = (char)('0' + mines)

return

board[r][c] = 'B'

for each of 8 directions (dr,dc)

dfs(r+dr, c+dc)

That is the entire DFS logic.

Everything else is bookkeeping.

Implement count() without duplication

count() is a tiny utility that loops over the same 8 directions and increments when it sees 'M' (or 'X', because once a mine is revealed it becomes 'X' but is still a mine).

Integrate with the public API

LeetCode gives you

public char[][] updateBoard(char[][] board, int[] click)

so we:

check the click cell for 'M' → explode it and return fast.

otherwise call dfs(click[0], click[1]) once; the DFS will reveal the whole connected region.

Complexity analysis while we are at it

Each cell is visited at most once and when visited we do O(1) work (8 neighbour checks).

Hence:

Time: O(m \* n)

Space: O(m \* n) worst-case recursion depth (a long snake-shaped board).

If you are afraid of stack overflow you can swap the DFS for a BFS queue; the rest of the logic is identical.

Full annotated code again, now with inline comments

class Solution {

// 8 directions: top-left, top, top-right, left, right, bottom-left, bottom, bottom-right

private static final int[][] DIRS = {

{-1,-1},{-1,0},{-1,1},

{ 0,-1}, { 0,1},

{ 1,-1},{ 1,0},{ 1,1}

};

public char[][] updateBoard(char[][] board, int[] click) {

int r = click[0], c = click[1];

if (board[r][c] == 'M') { // RULE-0: click on mine → explode

board[r][c] = 'X';

return board;

}

dfs(board, r, c); // RULE-1..3: reveal region

return board;

}

// ---------- DFS core ----------

private void dfs(char[][] b, int r, int c) {

// guard-clause: out of bounds OR already visited / not empty

if (r < 0 || r >= b.length || c < 0 || c >= b[0].length || b[r][c] != 'E')

return;

int mines = countAdjacentMines(b, r, c);

if (mines > 0) { // RULE-2: digit cell

b[r][c] = (char) ('0' + mines);

return; // stop recursion here

}

// RULE-3: blank cell → mark and flood

b[r][c] = 'B';

for (int[] d : DIRS)

dfs(b, r + d[0], c + d[1]);

}

// ---------- helper ----------

private int countAdjacentMines(char[][] b, int r, int c) {

int cnt = 0;

for (int[] d : DIRS) {

int nr = r + d[0], nc = c + d[1];

&& (b[nr][nc] == 'M' || b[nr][nc] == 'X')) // 'X' is a revealed mine

cnt++;

}

return cnt;

}

}

Take-away pattern

Any reveal/flood problem on a grid can be solved with this exact template:

if cell not in expected hidden state → return

compute local constraint (here = count mines)

if constraint forces a terminal value → write it and return

else mark revealed value and recurse/neighbour-enqueue

Swap the local-constraint function and you have surrounded regions, number of islands, minesweeper, or any other grid-propagation task.

**Why no need a 'visited' check in Minesweeper ?**

We don't need a separate 'visited' array in the Minesweeper DFS because we are updating the board in place and we only recurse on cells that are 'E' (unrevealed empty cells). Once we reveal a cell by changing it to either 'B' (blank) or a digit (1-8), it will no longer be 'E', so in the DFS, we check for 'E' and thus won't process the same cell again.

Let me break it down:

**We only start DFS on 'E' cells.** If a cell is not 'E', we skip it in the recursion.

**We change the cell's value during the DFS.** When we visit an 'E' cell, we either:

Set it to a digit (if there are adjacent mines) and stop, or

Set it to 'B' and then recurse to its neighbors.

**The condition board[i][j] != 'E' in the DFS function acts as our visited check.** Once we change the cell from 'E' to something else, we won't process it again.

This is why we don't need an explicit visited array. The board itself tracks which cells have been visited by the fact that they are no longer 'E'.

However, note that in the BFS solution, we also only add 'E' cells to the queue, and we mark them as visited by changing the board value (so we don't enqueue the same cell multiple times). The same principle applies.

Let me illustrate with an example:

Suppose we have a board and we click on an 'E' cell that has no adjacent mines. We set it to 'B'. Then we look at its 8 neighbors. For each neighbor that is 'E', we recursively call DFS. When we look at that same cell again from a neighbor, it's now 'B' (not 'E') so we skip it.

This ensures that we don't get into an infinite loop.

So the key is: we are using the board's state to track whether a cell has been visited (by changing it from 'E' to another state).

Great question! The lack of an explicit visited array in Minesweeper DFS is actually a clever optimization. Here's why it's not needed:

**🎯 The Implicit Visited Mechanis**m

The key insight is that **we modify the board state itself to serve as our visited check**.

**How It Works:**

private void dfs(int i, int j) {

// This condition acts as both boundary check AND visited check:

if (i < 0 || i >= m || j < 0 || j >= n || board[i][j] != 'E')

return;

// If we get here, the cell is guaranteed to be unrevealed ('E')

// We immediately change the cell state, marking it as "visited"

int minesCount = countMines(i, j);

if (minesCount > 0) {

board[i][j] = (char) (minesCount + '0'); // Now it's '1'-'8'

} else {

board[i][j] = 'B'; // Now it's 'B' (blank)

// Continue recursion to neighbors...

}

}

**🔄 State Transitions as Visited Marker**s

**Initial State:** 'E' (Empty, unrevealed)

**Final States:** 'B', '1', '2', ... '8'

Once a cell changes from 'E' to any other state, it will never satisfy board[i][j] != 'E' again, so the DFS will skip it.

**📊 Comparison: With vs Without Explicit Visite**d

**Without Explicit Visited (Current Solution):**

if (board[i][j] != 'E') return; // Uses board state as visited check

**With Explicit Visited (Alternative):**

if (visited[i][j] || board[i][j] != 'E') return;

visited[i][j] = true;

**🧪 Let's Trace Why This Work**s

**Example:** Click on [0,0] which has no adjacent mines

Step 1: dfs(0,0) - finds 0 mines, sets board[0][0] = 'B'

Step 2: Recursively calls dfs(0,1)

Step 3: dfs(0,1) - finds 0 mines, sets board[0][1] = 'B'

Step 4: dfs(0,1) recursively calls dfs(0,0) again

Step 5: dfs(0,0) sees board[0][0] = 'B' (not 'E') → RETURNS IMMEDIATELY

**⚡ Why This is More Efficient**

**Saves Memory**: No extra O(m×n) visited array

**Saves Time**: No need to initialize/maintain separate data structure

**Cleaner Code**: Fewer variables to track

**Natural Fit**: The game rules already require state changes

**🚨 When You WOULD Need Explicit Visite**d

You'd need explicit visited tracking if:

// Scenario 1: When you can't modify the original board

if (weCannotChangeTheBoardState) {

// Then we'd need boolean[][] visited

}

// Scenario 2: When multiple states can lead to reprocessing

if (cellCanBeRevisitedInDifferentStates) {

// Then state change alone isn't sufficient

}

// Scenario 3: When you need to track visitation separately from state

if (weNeedToKnowJustVisitHistory) {

// Not just current state

}

**🎮 Game Logic Alignmen**t

This approach perfectly aligns with the actual Minesweeper game:

Once revealed, a cell never goes back to unrevealed state

The visual change on the board prevents you from clicking it again

The recursion naturally follows the "cascade reveal" behavior

**💡 Key Takeawa**y

The line board[i][j] != 'E' does double duty:

**Content check**: Is this an unrevealed empty cell?

**Visited check**: Have we processed this cell before?

This elegant solution works because the problem domain naturally provides a way to mark cells as "processed" through state changes, eliminating the need for separate bookkeeping.

It's a great example of how understanding the problem domain can lead to cleaner, more efficient algorithms!