Problem 289: Game of Life

Problem Information

Difficulty: Medium
Acceptance Rate: 0.00%
Paid Only: No

Problem Description

According to

Wikipedia's article

: "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 (taken from the above Wikipedia article):

Any live cell with fewer than two live neighbors dies as if caused by under-population.

Any live cell with two or three live neighbors lives on to the next generation.

Any live cell with more than three live neighbors dies, as if by over-population.

Any dead cell with exactly three live neighbors becomes a live cell, as if by reproduction.

The next state of the board is determined by applying the above rules simultaneously to every cell in the current state of the

m x n

grid

board

. In this process, births and deaths occur

simultaneously

.

Given the current state of the

board

,

update

the

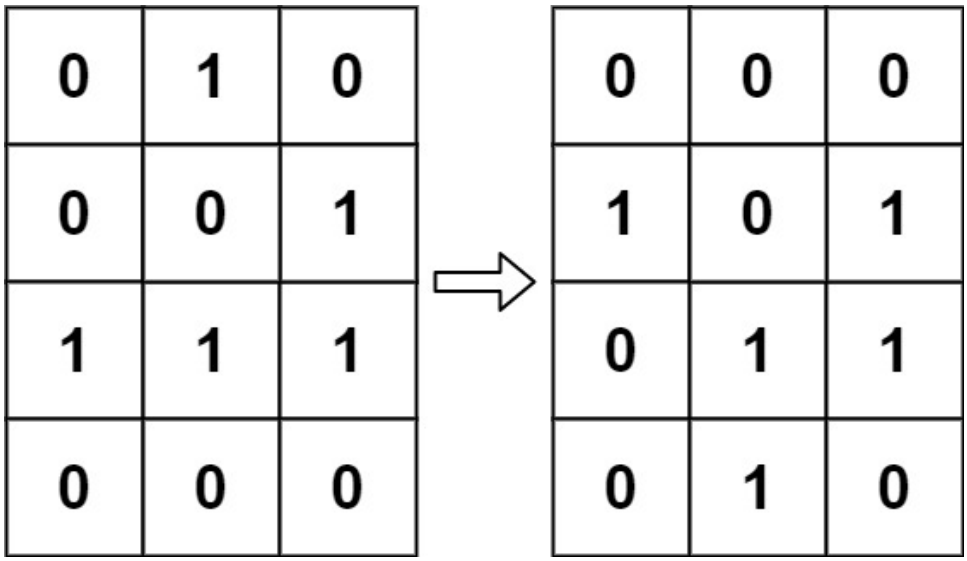
board

to reflect its next state.

Note

that you do not need to return anything.

Example 1:



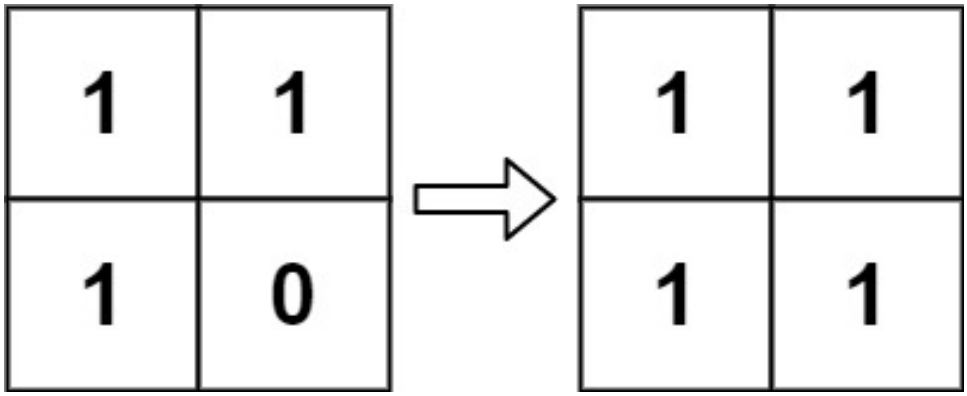
Input:

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]]

Example 2:



Input:

board = [[1,1],[1,0]]

Output:

[[1,1],[1,1]]

Constraints:

m == board.length

n == board[i].length

1 <= m, n <= 25

board[i][j]

is

0

or

1

.

Follow up:

Could you solve it in-place? Remember that the board needs to be updated simultaneously: You cannot update some cells first and then use their updated values to update other cells.

In this question, we represent the board using a 2D array. In principle, the board is infinite, which would cause problems when the active area encroaches upon the border of the array (i.e., live cells reach the border). How would you address these problems?

Code Snippets

C++:

```
class Solution {
public:
void gameOfLife(vector<vector<int>>& board) {

}
};
```

Java:

```
class Solution {
public void gameOfLife(int[][] board) {

}
}
```

Python3:

```
class Solution:
def gameOfLife(self, board: List[List[int]]) -> None:
"""
Do not return anything, modify board in-place instead.
"""
```

Python:

```
class Solution(object):
def gameOfLife(self, board):
"""
```

```
:type board: List[List[int]]
:rtype: None Do not return anything, modify board in-place instead.
"""
```

JavaScript:

```
/**
* @param {number[][]} board
* @return {void} Do not return anything, modify board in-place instead.
*/
var gameOfLife = function(board) {

};
```

TypeScript:

```
/**
Do not return anything, modify board in-place instead.
*/
function gameOfLife(board: number[][]): void {

};
```

C#:

```
public class Solution {
public void GameOfLife(int[][] board) {

}
}
```

C:

```
void gameOfLife(int** board, int boardSize, int* boardColSize) {

}
```

Go:

```
func gameOfLife(board [][]int) {

}
```

Kotlin:

```
class Solution {
fun gameOfLife(board: Array<IntArray>): Unit {

}
}
```

Swift:

```
class Solution {
func gameOfLife(_ board: inout [[Int]]) {

}
}
```

Rust:

```
impl Solution {
pub fn game_of_life(board: &mut Vec<Vec<i32>>) {

}
}
```

Ruby:

```
# @param {Integer[][]} board
# @return {Void} Do not return anything, modify board in-place instead.
def game_of_life(board)

end
```

PHP:

```
class Solution {

/**
* @param Integer[][] $board
* @return NULL
*/
function gameOfLife(&$board) {

}
```

```
}
```

Dart:

```
class Solution {
void gameOfLife(List<List<int>> board) {

}
}
```

Scala:

```
object Solution {
def gameOfLife(board: Array[Array[Int]]): Unit = {

}
}
```

Solutions

C++ Solution:

```
/*
* Problem: Game of Life
* Difficulty: Medium
* Tags: array, math
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(1) to O(n) depending on approach
*/

class Solution {
public:
void gameOfLife(vector<vector<int>>& board) {

}
};
```

Java Solution:

```
/**
* Problem: Game of Life
* Difficulty: Medium
* Tags: array, math
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(1) to O(n) depending on approach
*/

class Solution {
public void gameOfLife(int[][] board) {

}
}
```

Python3 Solution:

```
"""
Problem: Game of Life
Difficulty: Medium
Tags: array, math

Approach: Use two pointers or sliding window technique
Time Complexity: O(n) or O(n log n)
Space Complexity: O(1) to O(n) depending on approach
"""

class Solution:
def gameOfLife(self, board: List[List[int]]) -> None:
# TODO: Implement optimized solution
pass
```

Python Solution:

```
class Solution(object):
def gameOfLife(self, board):
"""
:type board: List[List[int]]
:rtype: None Do not return anything, modify board in-place instead.
"""
```

JavaScript Solution:

```
/**
* Problem: Game of Life
* Difficulty: Medium
* Tags: array, math
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(1) to O(n) depending on approach
*/

/**
* @param {number[][]} board
* @return {void} Do not return anything, modify board in-place instead.
*/
var gameOfLife = function(board) {

};
```

TypeScript Solution:

```
/**
* Problem: Game of Life
* Difficulty: Medium
* Tags: array, math
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(1) to O(n) depending on approach
*/

/**
Do not return anything, modify board in-place instead.
*/
function gameOfLife(board: number[][]): void {

};
```

C# Solution:

```
/*
* Problem: Game of Life
```

```
* Difficulty: Medium
* Tags: array, math
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(1) to O(n) depending on approach
*/

public class Solution {
public void GameOfLife(int[][] board) {

}
}
```

C Solution:

```
/*
* Problem: Game of Life
* Difficulty: Medium
* Tags: array, math
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(1) to O(n) depending on approach
*/

void gameOfLife(int** board, int boardSize, int* boardColSize) {

}
```

Go Solution:

```
// Problem: Game of Life
// Difficulty: Medium
// Tags: array, math
//
// Approach: Use two pointers or sliding window technique
// Time Complexity: O(n) or O(n log n)
// Space Complexity: O(1) to O(n) depending on approach

func gameOfLife(board [][]int) {
```

```
}
```

Kotlin Solution:

```
class Solution {
fun gameOfLife(board: Array<IntArray>): Unit {

}
}
```

Swift Solution:

```
class Solution {
func gameOfLife(_ board: inout [[Int]]) {

}
}
```

Rust Solution:

```
// Problem: Game of Life
// Difficulty: Medium
// Tags: array, math
//
// Approach: Use two pointers or sliding window technique
// Time Complexity: O(n) or O(n log n)
// Space Complexity: O(1) to O(n) depending on approach

impl Solution {
pub fn game_of_life(board: &mut Vec<Vec<i32>>) {

}
}
```

Ruby Solution:

```
# @param {Integer[][]} board
# @return {Void} Do not return anything, modify board in-place instead.
def game_of_life(board)
```

```
end
```

PHP Solution:

```
class Solution {

/**
* @param Integer[][] $board
* @return NULL
*/
function gameOfLife(&$board) {

}
}
```

Dart Solution:

```
class Solution {
void gameOfLife(List<List<int>> board) {

}
}
```

Scala Solution:

```
object Solution {
def gameOfLife(board: Array[Array[Int]]): Unit = {

}
}
```