Problem 723: Candy Crush

Problem Information

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

Problem Description

This question is about implementing a basic elimination algorithm for Candy Crush.

Given an

m x n

integer array

board

representing the grid of candy where

board[i][j]

represents the type of candy. A value of

board[i][j] == 0

represents that the cell is empty.

The given board represents the state of the game following the player's move. Now, you need to restore the board to a stable state by crushing candies according to the following rules:

If three or more candies of the same type are adjacent vertically or horizontally, crush them all at the same time - these positions become empty.

After crushing all candies simultaneously, if an empty space on the board has candies on top of itself, then these candies will drop until they hit a candy or bottom at the same time. No new candies will drop outside the top boundary.

After the above steps, there may exist more candies that can be crushed. If so, you need to repeat the above steps.

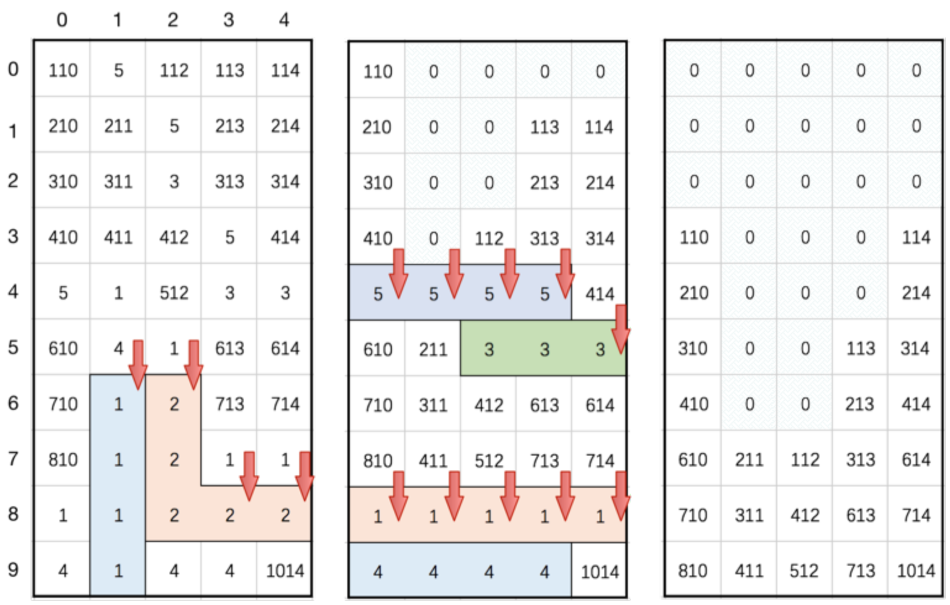
If there does not exist more candies that can be crushed (i.e., the board is stable), then return the current board.

You need to perform the above rules until the board becomes stable, then return

the stable board

.

Example 1:



	0	1	2	3	4
0	110	5	112	113	114
1	210	211	5	213	214
2	310	311	3	313	314
3	410	411	412	5	414
4	5	1	512	3	3
5	610	4	1	613	614
6	710	1	2	713	714
7	810	1	2	1	1
8	1	1	2	2	2
9	4	1	4	4	1014

Candy crush and Drop

110	0	0	0	0
210	0	0	113	114
310	0	0	213	214
410	0	112	313	314
5	5	5	5	414
610	211	3	3	3
710	311	412	613	614
810	411	512	713	714
1	1	1	1	1
4	4	4	4	1014

Candy crush and Drop

0	0	0	0	0
0	0	0	0	0
0	0	0	0	0
110	0	0	0	114
210	0	0	0	214
310	0	0	113	314
410	0	0	213	414
610	211	112	313	614
710	311	412	613	714
810	411	512	713	1014

Stable State

Input:

board = [[110,5,112,113,114],[210,211,5,213,214],[310,311,3,313,314],[410,411,412,5,414],[5,1,512,3,3],[610,4,1,613,614],[710,1,2,713,714],[810,1,2,1,1],[1,1,2,2,2],[4,1,4,4,1014]]

Output:

[[0,0,0,0,0],[0,0,0,0,0],[0,0,0,0,0],[110,0,0,0,114],[210,0,0,0,214],[310,0,0,113,314],[410,0,0,213,414],[610,211,112,313,614],[710,311,412,613,714],[810,411,512,713,1014]]

Example 2:

Input:

board = [[1,3,5,5,2],[3,4,3,3,1],[3,2,4,5,2],[2,4,4,5,5],[1,4,4,1,1]]

Output:

[[1,3,0,0,0],[3,4,0,5,2],[3,2,0,3,1],[2,4,0,5,2],[1,4,3,1,1]]

Constraints:

m == board.length

n == board[i].length

3 <= m, n <= 50

1 <= board[i][j] <= 2000

Code Snippets

C++:

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

}
};
```

Java:

```
class Solution {
public int[][] candyCrush(int[][] board) {
}
}
```

Python3:

```
class Solution:
def candyCrush(self, board: List[List[int]]) -> List[List[int]]:
```

Python:

```
class Solution(object):
def candyCrush(self, board):
"""
:type board: List[List[int]]
:rtype: List[List[int]]
"""
```

JavaScript:

```
/**
* @param {number[][]} board
* @return {number[][]}
*/
var candyCrush = function(board) {

};
```

TypeScript:

```
function candyCrush(board: number[][]): number[][] {

};
```

C#:

```
public class Solution {
public int[][] CandyCrush(int[][] board) {

}
}
```

C:

```
/**
* Return an array of arrays of size *returnSize.
* The sizes of the arrays are returned as *returnColumnSizes array.
* Note: Both returned array and *columnSizes array must be malloced, assume
caller calls free().
*/
int** candyCrush(int** board, int boardSize, int* boardColSize, int*
returnSize, int** returnColumnSizes) {

}
```

Go:

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

}
```

Kotlin:

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

}
}
```

Swift:

```
class Solution {
func candyCrush(_ board: [[Int]]) -> [[Int]] {

}
}
```

Rust:

```
impl Solution {
pub fn candy_crush(board: Vec<Vec<i32>>) -> Vec<Vec<i32>> {

}
}
```

Ruby:

```
# @param {Integer[][]} board
# @return {Integer[][]}
def candy_crush(board)

end
```

PHP:

```
class Solution {

/**
* @param Integer[][] $board
* @return Integer[][]
*/
function candyCrush($board) {

}
}
```

Dart:

```
class Solution {
List<List<int>> candyCrush(List<List<int>> board) {

}
}
```

Scala:

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

}
}
```

Elixir:

```
defmodule Solution do
@spec candy_crush(board :: [[integer]]) :: [[integer]]
def candy_crush(board) do

end
end
```

Erlang:

```
-spec candy_crush(Board :: [[integer()]]) -> [[integer()]].
candy_crush(Board) ->
.
```

Racket:

```
(define/contract (candy-crush board)
(-> (listof (listof exact-integer?)) (listof (listof exact-integer?)))
)
```

Solutions

C++ Solution:

```
/*
* Problem: Candy Crush
* Difficulty: Medium
* Tags: array
*
* 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:
vector<vector<int>> candyCrush(vector<vector<int>>& board) {

}
};
```

Java Solution:

```
/**
* Problem: Candy Crush
* Difficulty: Medium
* Tags: array
*
* 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 int[][] candyCrush(int[][] board) {

}
}
```

Python3 Solution:

```
"""
Problem: Candy Crush
Difficulty: Medium
Tags: array

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 candyCrush(self, board: List[List[int]]) -> List[List[int]]:
# TODO: Implement optimized solution
pass
```

Python Solution:

```
class Solution(object):
def candyCrush(self, board):
"""
:type board: List[List[int]]
:rtype: List[List[int]]
"""
```

JavaScript Solution:

```
/**
* Problem: Candy Crush
* Difficulty: Medium
* Tags: array
*
* 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 {number[][]}
*/
var candyCrush = function(board) {

};
```

TypeScript Solution:

```
/**
* Problem: Candy Crush
* Difficulty: Medium
* Tags: array
*
* 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
*/

function candyCrush(board: number[][]): number[][] {

};
```

C# Solution:

```
/*
* Problem: Candy Crush
* Difficulty: Medium
* Tags: array
*
* 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 int[][] CandyCrush(int[][] board) {

}
}
```

C Solution:

```
/*
* Problem: Candy Crush
* Difficulty: Medium
* Tags: array
*
* 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
*/

/**
* Return an array of arrays of size *returnSize.
* The sizes of the arrays are returned as *returnColumnSizes array.
* Note: Both returned array and *columnSizes array must be malloced, assume
caller calls free().
*/
int** candyCrush(int** board, int boardSize, int* boardColSize, int*
returnSize, int** returnColumnSizes) {

}
```

Go Solution:

```
// Problem: Candy Crush
// Difficulty: Medium
// Tags: array
//
// 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 candyCrush(board [][]int) [][]int {

}
```

Kotlin Solution:

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

}
}
```

Swift Solution:

```
class Solution {
func candyCrush(_ board: [[Int]]) -> [[Int]] {

}
}
```

Rust Solution:

```
// Problem: Candy Crush
// Difficulty: Medium
// Tags: array
//
// 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 candy_crush(board: Vec<Vec<i32>>) -> Vec<Vec<i32>> {

}
}
```

Ruby Solution:

```
# @param {Integer[][]} board
# @return {Integer[][]}
def candy_crush(board)
```

```
end
```

PHP Solution:

```
class Solution {

/**
* @param Integer[][] $board
* @return Integer[][]
*/
function candyCrush($board) {

}
}
```

Dart Solution:

```
class Solution {
List<List<int>> candyCrush(List<List<int>> board) {

}
}
```

Scala Solution:

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

}
}
```

Elixir Solution:

```
defmodule Solution do
@spec candy_crush(board :: [[integer]]) :: [[integer]]
def candy_crush(board) do

end
end
```

Erlang Solution:

```
-spec candy_crush(Board :: [[integer()]]) -> [[integer()]].
candy_crush(Board) ->
.
```

Racket Solution:

```
(define/contract (candy-crush board)
(-> (listof (listof exact-integer?)) (listof (listof exact-integer?)))
)
```