Previous Next

723. Candy Crush

Nov. 4, 2017 | 19.1K views

Average Rating: 4.50 (18 votes)

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

Given a 2D integer array `board` representing the grid of candy, different positive integers `board[i][j]` represent different types of candies. A value of `board[i][j] = 0` represents that the cell at position `(i, j)` 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:

1. 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.
2. 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.)
3. After the above steps, there may exist more candies that can be crushed. If so, you need to repeat the above steps.
4. If there does not exist more candies that can be crushed (ie. the board is *stable*), then return the current board.

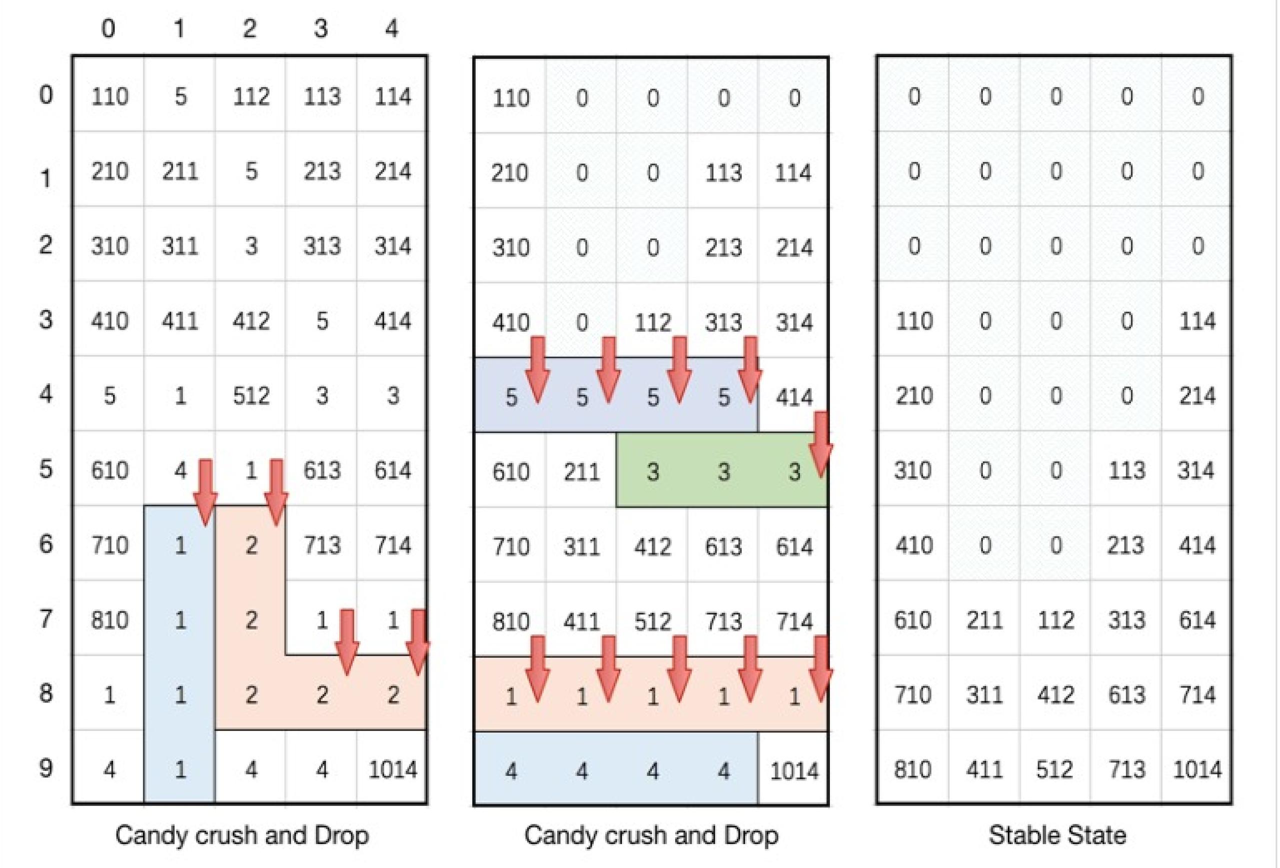
You need to perform the above rules until the board becomes stable, then return the current board.

Example:

```
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],[310,0,0,113,3
14],[410,0,0,213,414],[610,211,112,313,614],[710,311,412,613,714],[810,411,512,713,
1014]]

Explanation:
```



Note:

1. The length of `board` will be in the range [3, 50].
2. The length of `board[i]` will be in the range [3, 50].
3. Each `board[i][j]` will initially start as an integer in the range [1, 2000].

Approach #1: Ad-Hoc [Accepted]

Intuition

We need to simply perform the algorithm as described. It consists of two major steps: a crush step, and a gravity step. We work through each step individually.

Algorithm

Crushing Step

When crushing, one difficulty is that we might accidentally crush candy that is part of another row. For example, if the board is:

```
123
145
111
```

and we crush the vertical row of `1`s early, we may not see there was also a horizontal row.

To remedy this, we should flag candy that should be crushed first. We could use an auxillary `toCrush` boolean array, or we could mark it directly on the board by making the entry negative (ie. `board[i][j] = -Math.abs(board[i][j])`)

As for how to scan the board, we have two approaches. Let's call a *line* any row or column of the board.

For each line, we could use a sliding window (or `itertools.groupby` in Python) to find contiguous segments of the same character. If any of these segments have length 3 or more, we should flag them.

Alternatively, for each line, we could look at each width-3 slice of the line: if they are all the same, then we should flag those 3.

After, we can crush the candy by setting all flagged `board` cells to zero.

Gravity Step

For each column, we want all the candy to go to the bottom. One way is to iterate through and keep a stack of the (uncrushed) candy, popping and setting as we iterate through the column in reverse order.

Alternatively, we could use a sliding window approach, maintaining a read and write head. As the read head iterates through the column in reverse order, when the read head sees candy, the write head will write it down and move one place. Then, the write head will write zeroes to the remainder of the column.

We showcase the simplest approaches to these steps in the solutions below.

Java Python Copy

```python
class Solution(object):
    def candyCrush(self, board):
        R, C = len(board), len(board[0])
        todo = False

        for r in xrange(R):
            for c in xrange(C-2):
                if abs(board[r][c]) == abs(board[r][c+1]) == abs(board[r][c+2]) != 0:
                    board[r][c] = board[r][c+1] = board[r][c+2] = -abs(board[r][c])
                    todo = True

        for r in xrange(R-2):
            for c in xrange(C):
                if abs(board[r][c]) == abs(board[r+1][c]) == abs(board[r+2][c]) != 0:
                    board[r][c] = board[r+1][c] = board[r+2][c] = -abs(board[r][c])
                    todo = True

        for c in xrange(C):
            wr = R-1
            for r in xrange(R-1, -1, -1):
                if board[r][c] > 0:
                    board[wr][c] = board[r][c]
                    wr -= 1
            for wr in xrange(wr, -1, -1):
                board[wr][c] = 0

        return self.candyCrush(board) if todo else board
```

Complexity Analysis

- Time Complexity: $O((R*C)^2)$, where R, C is the number of rows and columns in `board`. We need $O(R*C)$ to scan the board, and we might crush only 3 candies repeatedly.
- Space Complexity: $O(1)$ additional complexity, as we edit the board in place.

Analysis written by: @awice.

Rate this article:

Previous Next

Comments: 15

Sort By

Type comment here... (Markdown is supported)

Preview Post

sstcurry 67 December 7, 2017 11:05 PM

Great solution!

It's *O((R*C)^2)* complexity because each function call scans the board three times so it's *3(R*C)*. If we only crush 3 candies each time, the function will be called *(R*C)/3* times. Multiply those two terms together you get *O((R*C)^2)*.

25 Share Reply

SHOW 4 REPLIES

akhil1311 117 October 10, 2019 4:49 AM

This is just an implementation heavy problem; not an interesting one.

13 Share Reply

vnk01 18 March 3, 2019 1:02 AM

Horrible code structure & no modular design at all...
(To be cleared, I like the explanation but wonder why it comes to a big mess in the actual implementation)

12 Share Reply

GupiBagha 41 February 18, 2020 3:44 AM

Please dont use variable names like 'r' and 'R'. Makes code unreadable, will never pass code reviews (and also be frowned upon during interviews)

1 Share Reply

vegito2002 1183 January 11, 2018 12:32 AM

My **similar solution** though more verbose.

1 Share Reply

sschangi 183 November 6, 2017 11:12 AM

@awice I have two questions: Firstly, although the function may call itself recursively, is it still true that we say it is a O(1) space? Secondly, would you please elaborate on time complexity O((R*C)^2). I do not get the square power completely.

Thanks

1 Share Reply

SHOW 1 REPLY

fightForPuppy 64 January 24, 2019 4:41 AM

I only have one doubt: for the example in the description, why board[8][0] which with value 1 didn't get crushed for the first round.

0 Share Reply

SHOW 1 REPLY

jasperjoe 315 April 30, 2020 7:02 AM

Was someone copying your code? http://storypku.com/2017/11/leetcode-question-723-candy-crush/

0 Share Reply

yhossam95 22 April 16, 2020 11:47 PM

Space complexity is O(RC*) not O(1) because you can do up to O(R*C) scans and for each scan you'll have an a function saved in the call stack. An iterative implementation would've been O(1)

0 Share Reply

blackspinner 79 April 16, 2020 3:34 AM

Did @awice wrote this code?

0 Share Reply

‹ 1 2 ›