554. Brick Wall

April 8, 2017 | 11.7K views

Previous Next

Average Rating: 4.88 (16 votes)

There is a brick wall in front of you. The wall is rectangular and has several rows of bricks. The bricks have the same height but different width. You want to draw a vertical line from the **top** to the **bottom** and cross the **least** bricks.

The brick wall is represented by a list of rows. Each row is a list of integers representing the width of each brick in this row from left to right.

If your line go through the edge of a brick, then the brick is not considered as crossed. You need to find out how to draw the line to cross the least bricks and return the number of crossed bricks.

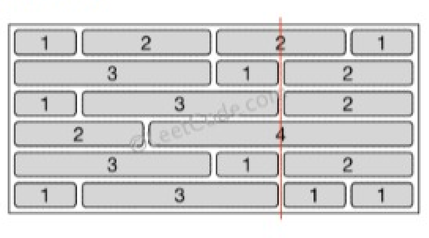
You cannot draw a line just along one of the two vertical edges of the wall, in which case the line will obviously cross no bricks.

Example:

```
Input: [[1,2,2,1],
        [3,1,2],
        [1,3,2],
        [2,4],
        [3,1,2],
        [1,3,1,1]]

Output: 2

Explanation:
```



Note:

1. The width sum of bricks in different rows are the same and won't exceed INT_MAX.
2. The number of bricks in each row is in range [1,10,000]. The height of wall is in range [1,10,000]. Total number of bricks of the wall won't exceed 20,000.

Solution

Approach #1 Brute Force [Time Limit Exceeded]

In this approach, we consider the given wall as being made up of virtual bricks each of width 1. We traverse over the width of the wall only in terms of these virtual bricks.

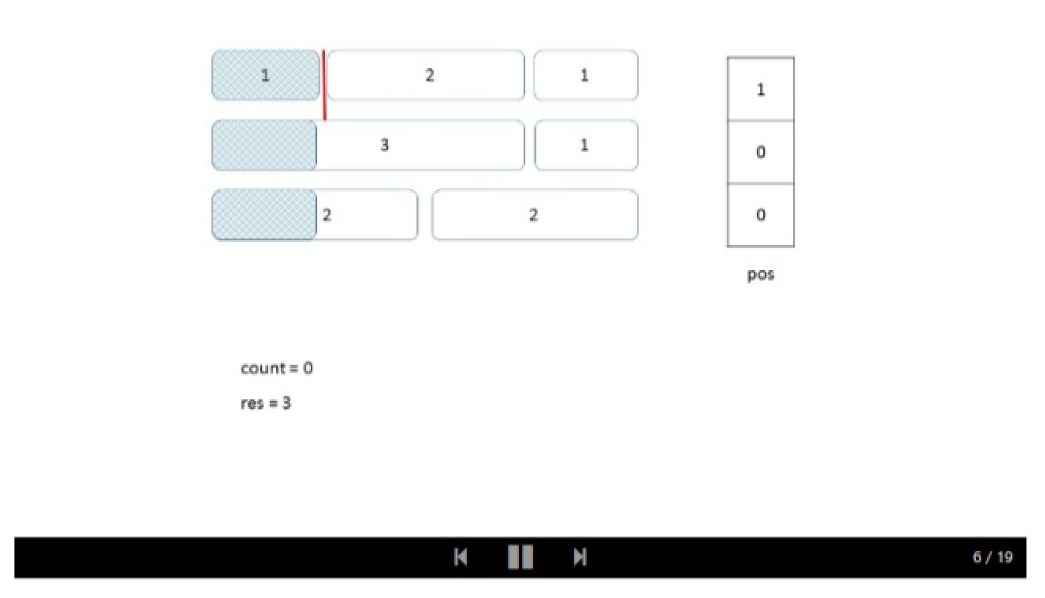
Firstly, we need to determine the total number of virtual bricks. For this, we determine the width of the given wall by summing up the widths of the bricks in the first row. This width is stored in sum. Thus, we need to traverse over the widthe sum times now in terms of 1 unit in each iteration.

We traverse over the virtual bricks in a column by column fashion. For keeping a track of the actual position at which we are currently in any row, we make use of a pos array. $pos[i]$ refers to the index of the brick in the i^{th} row, which is being treated as the virtual brick in the current column's traversal. Further, we maintain a $count$ variable to keep a track of the number of bricks cut if we draw a line down at the current position.

For every row considered during the column-by-column traversal, we need to check if we've hit an actual brick boundary. This is done by updating the brick's width after the traversal. If we don't hit an actual brick boundary, we need to increment $count$ to reflect that drawing a line at this point leads to cutting off 1 more brick. But, if we hit an actual brick boundary, we increment the value of $pos[i]$, with i referring to the current row's index. But, now if we draw a line down through this boundary, we need not increment the $count$.

We repeat the same process for every column of width equal to a virtual brick, and determine the minimum value of $count$ obtained from all such traversals.

The following animation makes the process clearer:



Java   Copy

```java
public class Solution {
    public int leastBricks(List < List < Integer >> wall) {
        int[] pos = new int[wall.size()];
        int c = 0, sum = 0, res = Integer.MAX_VALUE;
        for (int el: wall.get(0))
            sum += el;
        while (sum != 0) {
            int count = 0;
            for (int i = 0; i < wall.size(); i++) {
                List < Integer > row = wall.get(i);
                if (row.get(pos[i]) != 0)
                    count++;
                else
                    pos[i]++;
                row.set(pos[i], row.get(pos[i]) - 1);
            }
            sum--;
            res = Math.min(res, count);
        }
        return res;
    }
}
```

Complexity Analysis

- Time complexity : $O(n*m)$. We traverse over the width(n) of the wall m times, where m is the height of the wall.
- Space complexity : $O(m)$. pos array of size m is used, where m is the height of the wall.

Approach #2 Better Brute Force[Time LImit Exceeded]

Algorithm

In this approach, instead of traversing over the columns in terms of 1 unit each time, we traverse over the columns in terms of the width of the smallest brick encountered while traversing the current column. Thus, we update pos array and $sums$ appropriately depending on the width of the smallest brick. Rest of the process remains the same as the first approach.

The optimization achieved can be viewed by considering this example:

```
[[100, 50, 50],
[50, 100],
[150]]
```

In this case, we directly jump over the columns in terms of widths of 50 units each time, rather than making traversals over widths incrementing by 1 unit each time.

Java   Copy

```java
public class Solution {
    public int leastBricks(List < List < Integer >> wall) {
        int[] pos = new int[wall.size()];
        int sum = 0, res = Integer.MAX_VALUE;
        for (int el: wall.get(0))
            sum += el;
        while (sum != 0) {
            int count = 0, mini = Integer.MAX_VALUE;
            for (int i = 0; i < wall.size(); i++) {
                List < Integer > row = wall.get(i);
                if (row.get(pos[i]) != 0) {
                    count++;
                } else
                    pos[i]++;
                mini = Math.min(mini, row.get(pos[i]));
            }
            for (int i = 0; i < wall.size(); i++) {
                List < Integer > row = wall.get(i);
                row.set(pos[i], row.get(pos[i]) - mini);
            }
            sum -= mini;
            res = Math.min(res, count);
        }
        return res;
    }
}
```

Complexity Analysis

- Time complexity : $O(n*m)$. In worst case, we traverse over the length(n) of the wall m times, where m is the height of the wall.
- Space complexity : $O(m)$. pos array of size m is used, where m is the height of the wall.

Approach #3 Using HashMap [Accepted]

Algorithm

In this approach, we make use of a HashMap map which is used to store entries in the form: $(sum, count)$. Here, sum refers to the cumulative sum of the bricks' widths encountered in the current row, and $count$ refers to the number of times the corresponding sum is obtained. Thus, sum in a way, represents the positions of the bricks's boundaries relative to the leftmost boundary.

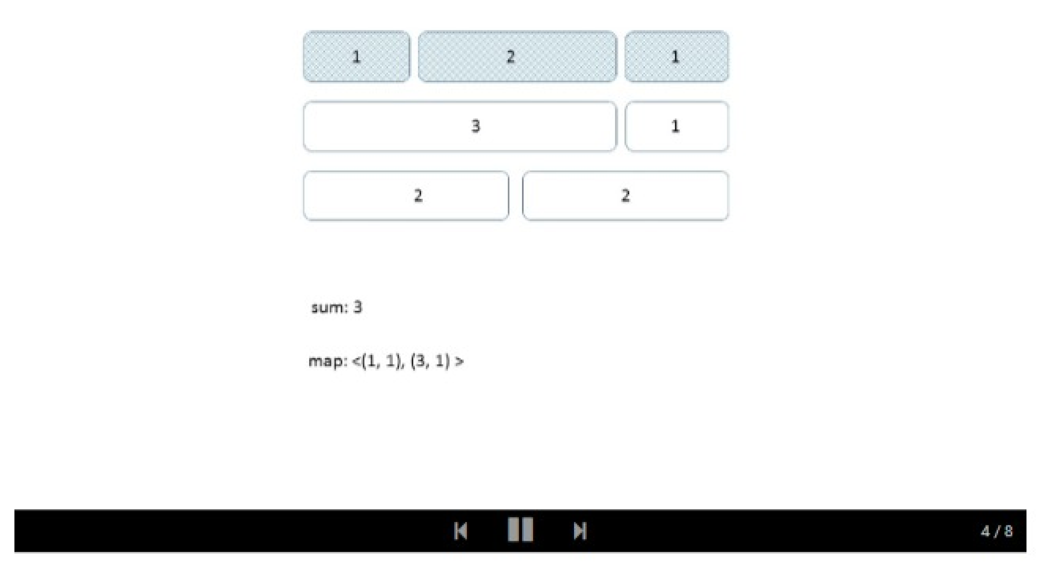
Let's look at the process first. We traverse over every row of the given wall. For every brick considered, we find the sum corresponding to the sum of the bricks' widths encountered so far in the current row. If this sum's entry doesn't exist in the map, we create a corresponding entry with an initial $count$ of 1. If the sum already exists as a key, we increment its corresponding $count$ value.

This is done based on the following observation. We will never obtain the same value of sum twice while traversing over a particular row. Thus, if the sum value is repeated while traversing over the rows, it means some row's brick boundary coincides with some previous row's brick boundary. This fact is accounted for by incrementing the corresponding $count$ value.

But, for every row, we consider the sum only upto the second last brick, since the last boundary isn't a valid boundary for the solution.

At the end, we can obtain the maximum $count$ value to determine the minimum number of bricks that need to be cut to draw a vetical line through them.

The following animation makes the process clear:



Java   Copy

```java
public class Solution {
    public int leastBricks(List < List < Integer >> wall) {
        HashMap < Integer, Integer > map = new HashMap < > ();
        for (List < Integer > row: wall) {
            int sum = 0;
            for (int i = 0; i < row.size() - 1; i++) {
                sum += row.get(i);
                if (map.containsKey(sum))
                    map.put(sum, map.get(sum) + 1);
                else
                    map.put(sum, 1);
            }
        }
        int res = wall.size();
        for (int key: map.keySet())
            res = Math.min(res, wall.size() - map.get(key));
        return res;
    }
}
```

Complexity Analysis

- Time complexity : $O(n)$. We traverse over the complete bricks only once. n is the total number of bricks in a wall.
- Space complexity : $O(m)$. map will contain atmost m entries, where m refers to the width of the wall.

Rate this article:

Previous Next