554. Brick Wall

Medium Array Hash Table

Problem Description

There's a wall constructed of multiple rows of bricks of varying widths, but each row has the same total width. Think of this like having a bunch of different sized blocks and trying to stack them up so that they all align perfectly at the ends of the wall, but between bricks, the seams could mismatch. The goal here is to draw a vertical line from the top of the wall to the bottom, going through as few bricks as possible. Here's the catch – you can't draw the line at the very edges since that would technically cross no bricks and defeats the purpose of the question. Moreover, if the line goes through the joint between two bricks, it doesn't count as crossing a brick. You are given a 2D array `wall`, which represents the layout of the wall where each inner array contains the widths of the bricks in that row.

To solve this problem, you must determine the position where the line crosses the minimum number of bricks and return that number. Imagine this like trying to find the weakest point in the wall where if you had to punch through it (without hitting the edges), you'd break the least amount of bricks.

Intuition

The solution comes down to finding the place in the wall with the most edges of bricks aligned vertically (not including the far left and far right of the wall). If you think about it, the more brick edges you have aligned in a vertical line, the fewer bricks your vertical line will cross.

What we do in the solution is essentially a counting exercise. For every row in the wall, we add up the widths of bricks and note down the running total (except the last brick because it touches the edge of the wall). Each running total represents a position where a vertical line crosses an edge between bricks, thus not crossing a brick itself. So we use a count map, a hash table, where keys are these totals (positions on the wall) and values are how many times we've seen this position across different rows. For every position (total width from the start of the wall), we increase the corresponding count.

Finally, we figure out the position with the highest count – the one that occurred most frequently across all rows, meaning our line would cross the least bricks at this position. We then subtract this maximum count from the total number of rows, giving us the least number of bricks our vertical line would cross.

Solution Approach

The key idea is to utilize a hash table to maintain the frequency of particular positions where a vertical line can cross through brick edges rather than the bricks themselves. We iterate over each row of the wall, and within each row, iterate over the width of each brick up until the second to last brick. We exclude the last brick because the edge of the last brick aligns with the edge of the wall, and the rules state we cannot draw a line there.

Here is a step-by-step breakdown of the algorithm:

1. Initialize an empty hash table `cnt` to store the frequencies of the crossed edges' positions.
2. Iterate over each row in `wall`. For each row:
   - Initialize a running sum `width` which will keep track of the position as we add up the widths of the bricks.
   - Go through each brick in the row except the last one (`range(len(row) - 1)`), adding the width of the brick to `width`.
   - For the current `width`, increment the count in `cnt`. This `width` acts as a key and represents the position where a vertical line would cross the edge between adjacent bricks.
3. After processing all rows, check if `cnt` is empty. If it is, that means every row is just one brick, so any vertical line will cross all the bricks. Therefore, return the number of rows as the number of crossed bricks.
4. If `cnt` is not empty, find the maximum frequency stored in `cnt` using `max(cnt, key=cnt.get)`. This represents the edge position that has been encountered the most and where the line will cross the least number of bricks.
5. The minimum number of crossed bricks would be the total number of rows minus the maximum frequency edge, calculated as `len(wall) - cnt[max(cnt, key=cnt.get)]`.

In this approach, the time complexity arises from iterating over all bricks once, giving O(n) complexity, where n is the total number of bricks. The space complexity is O(m), where m is the width of the wall as this dictates the maximum number of potential positions for the edges, hence the number of keys in the hash table.

Example Walkthrough

Let us consider a wall represented by the following 2D array, where each inner array contains the widths of the bricks in a particular row:

```
wall = [
  [3, 5, 1, 1],
  [2, 3, 3, 2],
  [5, 5],
  [4, 4, 2],
  [3, 3, 3, 3],
  [1, 1, 6, 1, 1]
]
```

Step-by-step walkthrough:

1. We begin by initializing a hash table `cnt` that will count the frequency of crossed edges' positions (excluding the end edges of the wall). `cnt` looks like `{}`.
2. As we iterate through the wall row by row, we will keep a running sum of the brick widths for each row, stopping before the last brick.
3. For the first row `[3, 5, 1, 1]`, the running sums of widths are:
   - After first brick: 3 (`cnt` now looks like `{3: 1}`)
   - After second brick: 3+5=8 (`cnt` now looks like `{3: 1, 8: 1}`)
   - After third brick: 8+1=9 (we don't count the last brick edge)
4. We repeat this for each subsequent row:
   - Second row `[2, 3, 3, 2]` gives us edges at 2, and 5 (`cnt` becomes `{3: 1, 8: 1, 2: 1, 5: 1}`)
   - Third row `[5, 5]` has only one internal edge at: 5 (`cnt` becomes `{3: 1, 8: 1, 2: 1, 5: 2}`)
   - Fourth row `[4, 4, 2]` contributes edges at: 4, and 8 (`cnt` becomes `{3: 1, 8: 2, 2: 1, 5: 2, 4: 1}`)
   - Fifth row `[3, 3, 3, 3]` adds edges at: 1, 4, and 7 (`cnt` becomes `{3: 1, 8: 2, 2: 1, 5: 2, 4: 2, 1: 1, 7: 1}`)
   - Last row `[1, 1, 6, 1, 1]` has edges at: 1, 2, and 8 (`cnt` becomes `{3: 1, 8: 3, 2: 2, 5: 2, 4: 2, 1: 2, 7: 1}`)
5. Now we have all the running sums in `cnt`. We don't have an empty `cnt`, meaning we have valid edge positions to consider.
6. We find the maximum frequency in `cnt`, which is 3 at position 8.
7. Finally, we calculate the minimum number of crossed bricks as the total number of rows `len(wall)` which is 6, minus the maximum frequency encountered, which is 3.
8. Therefore, drawing a vertical line at the position that corresponds to the total width of 8 (from the left) would cross the minimum number of bricks, which is $6 - 3 = 3$.

In this example, the least number of bricks that would be crossed by drawing a vertical line anywhere in the wall is 3.

Solution Implementation

Python

```python
from collections import defaultdict

class Solution:
    def leastBricks(self, wall: List[List[int]]) -> int:
        # Create a dictionary to count the frequency of each edge's position (except for the last edge of each row)
        edge_count = defaultdict(int)

        # Iterate over each row in the wall
        for row in wall:
            width = 0  # Initialize width to track the current position of edges

            # Iterate over each brick in the row, except the last brick
            for brick in row[:-1]:
                width += brick  # Increase the width by the current brick's length to find the next edge
                edge_count[width] += 1  # Increment the count of the edge at the corresponding width

        # If there are no edges counted, return the number of rows (since a vertical line would cross all rows)
        if not edge_count:
            return len(wall)

        # Find the maximum occurrence of a common edge and subtract it from the total rows.
        # This gives the minimum number of rows crossed by a vertical line.
        return len(wall) - max(edge_count.values())
```

Java

```java
// Solution to find the path through a brick wall that crosses the least number of bricks.
class Solution {

    // Function to determine the minimum number of bricks that must be crossed.
    // wall: List of lists representing the wall where each list is a row of bricks.
    public int leastBricks(List<List<Integer>> wall) {

        // A map to store the frequency of edges lining up vertically at each width index.
        Map<Integer, Integer> edgeFrequency = new HashMap<>();

        // Iterate over each row in the wall.
        for (List<Integer> row : wall) {
            int width = 0; // Tracks the cumulative width of bricks as we go along the row.

            // Iterate over the row, skipping the last brick to prevent counting the edge of the wall.
            for (int i = 0; i < row.size() - 1; i += 1) {
                width += row.get(i); // Add the width of the current brick to the total width.

                // Increase the count of the occurrence of this edge in the map.
                edgeFrequency.merge(width, 1, Integer::sum);
            }
        }

        // Find the maximum frequency of an edge lining up.
        // If no edges line up, the default maximum will be 0.
        int maxFrequency = edgeFrequency.values().stream().max(Comparator.naturalOrder()).orElse(0);

        // The minimum number of bricks crossed is the total number of rows minus the max frequency of an edge.
        return wall.size() - maxFrequency;
    }
}
```

C++

```cpp
#include <vector>
#include <unordered_map>
#include <algorithm>

/**
 * Function to find the minimum number of bricks that must be cut to create a vertical line that goes through the entire wall.
 * @param wall - 2D vector where each sub-vector represents a row of bricks in the wall
 * @return The minimum number of bricks that must be cut
 */
int leastBricks(const std::vector<std::vector<int>>& wall) {
    // Create a hash map to keep track of how many edges align at each width index
    std::unordered_map<int, int> edge_count;

    // Loop over each row of the wall
    for (const auto& row : wall) {
        int cumulative_width = 0; // Tracks the cumulative width as we add bricks

        // Iterate over the row up to the second to last brick to avoid considering the edge of the wall
        for (size_t i = 0; i < row.size() - 1; ++i) {
            cumulative_width += row[i]; // Add the width of the current brick to the cumulative width

            // Increment the edge count for the current cumulative width, defaulting to 0 for the first time
            edge_count[cumulative_width]++;
        }
    }

    // Find the maximum count of aligned edges (excluding the wall's edges)
    int max_aligned_edges = 0;
    for (const auto& count : edge_count) {
        max_aligned_edges = std::max(max_aligned_edges, count.second);
    }

    // The minimum cuts needed is the total number of rows minus the maximum number of aligned edges
    return wall.size() - max_aligned_edges;
}

// Usage example:
int main() {
    std::vector<std::vector<int>> example_wall = {
        {1, 2, 2, 1},
        {3, 1, 2},
        {1, 3, 2},
        {2, 4},
        {3, 1, 2},
        {1, 3, 1, 1}
    };

    int cuts_needed = leastBricks(example_wall);
    std::cout << cuts_needed << std::endl; // Output will be the result of calling leastBricks on example_wall

    return 0;
}
```

TypeScript

```typescript
/**
 * Function to find the minimum number of bricks that must be cut to create a vertical line that goes through the entire wall.
 * @param {number[][]} wall - 2D array where each sub-array represents a row of bricks in the wall
 * @return {number} - The minimum number of bricks that must be cut
 */
const leastBricks = (wall: number[][]): number => {
    // Create a map to keep track of how many edges align at each width index
    const edgeCount: Map<number, number> = new Map();

    // Loop over each row of the wall
    for (const row of wall) {
        let cumulativeWidth: number = 0; // Tracks the cumulative width as we add bricks

        // Iterate over the row up to the second to last brick to avoid considering the edge of the wall
        for (let i = 0; i < row.length - 1; i++) {
            cumulativeWidth += row[i]; // Add the width of the current brick to the cumulative width

            // Increment the edge count for the current cumulative width, defaulting to 0 for the first time
            edgeCount.set(cumulativeWidth, (edgeCount.get(cumulativeWidth) || 0) + 1);
        }
    }

    // Find the maximum count of aligned edges (excluding the wall's edges)
    let maxAlignedEdges: number = 0;
    for (const count of edgeCount.values()) {
        maxAlignedEdges = Math.max(maxAlignedEdges, count);
    }

    // The minimum cuts needed is the total number of rows minus the maximum number of aligned edges
    return wall.length - maxAlignedEdges;
};

// Usage example (should be removed from the final code as it's not part of the requested rewrite):
const exampleWall: number[][] = [
    [1, 2, 2, 1],
    [3, 1, 2],
    [1, 3, 2],
    [2, 4],
    [3, 1, 2],
    [1, 3, 1, 1]
];
const cutsNeeded: number = leastBricks(exampleWall);
console.log(cutsNeeded); // Output would be the result of calling leastBricks on exampleWall
```

```python
from collections import defaultdict

class Solution:
    def leastBricks(self, wall: List[List[int]]) -> int:
        # Create a dictionary to count the frequency of each edge's position (except for the last edge of each row)
        edge_count = defaultdict(int)

        # Iterate over each row in the wall
        for row in wall:
            width = 0  # Initialize width to track the current position of edges

            # Iterate over each brick in the row, except the last brick
            for brick in row[:-1]:
                width += brick  # Increase the width by the current brick's length to find the next edge
                edge_count[width] += 1  # Increment the count of the edge at the corresponding width

        # If there are no edges counted, return the number of rows (since a vertical line would cross all rows)
        if not edge_count:
            return len(wall)

        # Find the maximum occurrence of a common edge and subtract it from the total rows.
        # This gives the minimum number of rows crossed by a vertical line.
        return len(wall) - max(edge_count.values())
```

Time and Space Complexity

Time Complexity

The time complexity of the code is mainly determined by the two nested loops. The outer loop iterates through each row in the wall, which is $O(n)$ where n is the number of rows. The inner loop iterates through each brick in the row, excluding the last brick. The total number of bricks in all rows is equivalent to the total number of iterations of the inner loop. Let's denote m as the average number of bricks per row. Therefore, the total complexity would approximately be $O(n * m)$.

However, in the worst-case scenario, m (the number of bricks per row) could be proportional to the width of the wall if the bricks are very narrow. If w represents the wall's width, then the complexity could also be expressed as $O(n * w)$ in such a case.

Finally, the complexity also includes the time required to find the `max` in the defaultdict `cnt`, which, in the worst case, is $O(w)$, where w is the unique widths that are less than the wall's width. This is because, in the worst case, each possible width could have a different number of edges crossing it.

So, the overall time complexity is $O(n * m)$ or $O(n * w) + O(w)$, which simplifies to $O(n * w)$ because we drop the less significant term.

Space Complexity

The space complexity is influenced by the defaultdict `cnt`, which stores the frequency of edge crossings at each width position. In the worst case, every possible position could have an edge crossing, so the space complexity would be $O(w)$ where w is the unique widths less than the wall's width. Therefore, the space complexity is $O(w)$.