2125. Number of Laser Beams in a Bank

Medium Array Math String Matrix

Leetcode Link

Problem Description

The problem provides a representation of a bank's floor plan using a binary string array `bank`, where each string represents a row in the bank. A '0' indicates an empty cell and a '1' indicates the presence of a security device. Laser beams are formed between security devices on different rows, but only if there are no security devices on the rows between them. Your task is to determine the total number of laser beams in the bank.

To better understand the problem, consider the following points:

- The floor plan is visualized as a grid, where each row is a string from the array, and each character in the string corresponds to a cell in that row.
- A laser beam can only form vertically between devices in columns where there happens to be a device in both rows with no devices in the intervening rows.
- Multiple beams can form in the same column if there are more than two devices on that column but separated by rows without devices.

The objective is to calculate the total number of these beams. Imagine stacking the rows on top of each other: a laser beam is a vertical line connecting '1's but only if the slice of column between them has no other '1's.

Intuition

To come up with the solution, we must track security devices along each row and identify potential beam formations. The approach is as follows:

- Go through each row of the bank and count the number of security devices ('1's).
- Keep track of the number of devices from the last row that had devices. Initially, this is zero since we haven't encountered any devices.
- As we continue to check each row, when we find a row with devices, we can potentially form laser beams with the devices from the previously encountered row. The number of new beams formed is the product of the number of devices in the current row and the number of devices in the previously encountered row.
- We then update the 'last' variable to hold the count of devices in the current row, as this will be used to calculate beams with the next row that contains devices.

Following this logic, the algorithm tallies the number of beams as we iterate over the rows. This systematic approach ensures that all possible beam connections are counted without missing any viable combinations or double-counting beams.

Solution Approach

The given Python implementation begins by initializing two variables: `last` to hold the count of devices from the last row that contained devices, and `ans` to accumulate the total number of laser beams.

The code then iterates over each row (`b`) in the bank:

```
for b in bank:
    ...
```

For each row, the code uses the `count` method to determine the number of '1's in it which translates to the number of devices in that row:

```
if (t := b.count('1')) > 0:
```

This count is assigned to `t`, and the use of the walrus operator (`:=`) helps condense the assignment and the condition into one line.

When the number of devices `t` is greater than zero, meaning the current row has devices, two things happen:

1. The code calculates the number of laser beams formed with the previous row by multiplying `last` (the count from the last row with devices) by `t` (the count from the current row):

```
ans += last * t
```

2. It then updates the `last` variable to `t`, so that it holds the current count for the next iteration:

```
last = t
```

The rationale behind this is simple: laser beams only form between devices from different rows. Thus, the current row can only form beams with a previous row, not with itself or rows it hasn't 'seen' yet.

After iterating through all the rows, `ans` holds the total number of beams, and the code returns this value:

```
return ans
```

The algorithm's complexity is O(m * n), where *m* is the number of rows and *n* is the number of columns (or the length of each row), as it needs to count the '1's in each row.

This solution is efficient and avoids unnecessary calculations by not looking at the individual positions of devices within the rows. It leverages the fact that any two devices in the same column from different rows would form beams with all pairs of rows between them that don't contain any devices.

Example Walkthrough

Let us consider a small example to illustrate the solution approach.

Suppose we have the following `bank` array representing our bank's floor plan:

```
["011",
"000",
"001"]
```

This array represents a bank with three rows and three columns. Each '1' is a security device, and each '0' is an empty space.

Using the intuition and solution approach:

1. We initialize `last` to 0 (since we have not processed any rows yet) and `ans` to 0 (as our accumulator for laser beams).
2. We start iterating through the rows:
   - For the first row (`"011"`), there are 2 security devices. With `last` initially being 0, no beams can be formed yet, so we simply set `last` to 2.
   - For the second row (`"000"`), there are no security devices. Nothing happens here since we need at least one security device to form a laser beam.
   - For the third row (`"001"`), there is 1 security device. We find the product of `last` (which is 2 from the first row) and the count of devices in the current row (1), resulting in 2 laser beams formed. We add this to `ans`, giving us `0 + (2 * 1) = 2` beams.
   - We update `last` to 1, reflecting the number of devices in the third row.
3. There are no more rows, so we conclude the example with `ans` being 2. There are 2 laser beams formed between the devices in the first and third rows.

Thus the final output for this example bank is 2 laser beams.

Python Solution

```python
class Solution:
    def numberOfBeams(self, bank: List[str]) -> int:
        # Variable to store the count of devices (1s) in the last non-empty row
        last_count = 0

        # Variable to store the total number of beams formed
        total_beams = 0

        # Iterate over each row in the bank
        for row in bank:
            # Count the number of devices ('1's) in the current row
            current_count = row.count('1')

            # If the current row is not empty, proceed with calculations
            if current_count > 0:
                # Add the number of beams formed between the last non-empty row and the current row
                total_beams += last_count * current_count

                # Update the last_count to be the current row's count for the next iteration
                last_count = current_count

        # Return the total number of beams formed
        return total_beams
```

Java Solution

```java
class Solution {
    // Method to calculate the number of beams created by laser security systems in the bank.
    public int numberOfBeams(String[] bank) {
        // Initial count for the previous row that had at least one security device.
        int previousCount = 0;
        // Initialize the total number of beams to 0.
        int totalBeams = 0;

        // Iterate over each row in the bank.
        for (String row : bank) {
            // Initialize the count of security devices in the current row.
            int currentCount = 0;
            // Convert the current row to a character array for iteration.
            char[] devices = row.toCharArray();
            for (char device : devices) {
                // Check if the current spot has a security device (denoted by '1').
                if (device == '1') {
                    // Increment the count of devices in the current row.
                    ++currentCount;
                }
            }
            // If the current row has at least one security device, calculate beams.
            if (currentCount > 0) {
                // Add the number of beams formed between the previous and current row to totalBeams.
                totalBeams += previousCount * currentCount;
                // Update previousCount to be the count of the current row for the next iteration.
                previousCount = currentCount;
            }
        }
        // Return the total number of beams formed in the bank.
        return totalBeams;
    }
}
```

C++ Solution

```cpp
class Solution {
public:
    // Function to calculate the number of laser beams that can be formed in the security bank.
    int numberOfBeams(vector<string>& bank) {
        int totalBeams = 0;     // This variable will store the total number of beams.
        int previousDevices = 0; // This will hold the count of devices (ones) in the previous row.

        // Iterate over each row in the 'bank' grid.
        for (auto& row : bank) {
            int currentDevices = 0; // Count the number of devices (ones) in the current row.
            // Iterate through each character in the row to count the devices.
            for (char& device : row) {
                if (device == '1') { // If the current character is '1', increment the counter.
                    ++currentDevices;
                }
            }
            // If there are devices in the current row, calculate the beams between this and the previous row with devices.
            if (currentDevices > 0) {
                totalBeams += previousDevices * currentDevices; // Beams are formed between the rows.
                previousDevices = currentDevices; // Update the previousDevices count for the next iteration.
            }
        }
        return totalBeams; // Return the total number of beams formed.
    }
};
```

Typescript Solution

```typescript
/**
 * Calculate the number of beams produced by security lasers in a bank vault.
 * Lasers are installed on the ceiling (represented by '1's) and the beams connect with cameras across the vault.
 * Each row in the array represents a ceiling of the bank, '1' denotes a laser and '0' denotes an empty space.
 * Beams are formed between rows of lasers without interfering with other beams.
 * @param {string[]} bank - array of strings representing the bank's ceiling layout.
 * @return {number} The total number of beams created by the security system.
 */
function numberOfBeams(bank: string[]): number {
    // Initialize variables to keep track of the last row's laser count and the total number of beams
    let lastLaserCount = 0;
    let totalBeams = 0;

    // Iterate through each row in the bank array
    for (const row of bank) {
        let currentLaserCount = 0; // Counter for the number of lasers in the current row

        // Iterate through each character in the row to count the number of lasers
        for (const value of row) {
            if (value === '1') {
                currentLaserCount++;
            }
        }

        // If the current row has lasers, calculate the beams with the previous row and update the lastLaserCount
        if (currentLaserCount !== 0) {
            totalBeams += lastLaserCount * currentLaserCount;
            lastLaserCount = currentLaserCount;
        }
    }

    // Return the calculated total number of beams
    return totalBeams;
}
```

Time and Space Complexity

Time Complexity

The time complexity of the code is `O(mn)`, where `m` is the number of rows in the `bank` list and `n` is the number of columns in each row. This is because the code iterates over each row with a loop (which runs `m` times), and within each iteration, it performs a count operation with `b.count('1')`, which, in the worst-case scenario, runs in `O(n)` time for each row if all elements need to be checked (where `n` can be seen as the maximum possible length of the binary strings representing the devices). The other operations inside the loop are constant time operations, therefore, they do not change the overall time complexity.

Space Complexity

The space complexity of the code is `O(1)`. The solution uses a constant amount of extra space regardless of the input size. Variables `last`, `ans`, and `t` are reused for each row, hence, the space used does not grow with the size of the input (`bank` list).