# 1428. Leftmost Column with at Least a One

Medium Array Binary Search Interactive Matrix

Leetcode Link

## Problem Description

The given problem presents a binary matrix with rows sorted in a non-decreasing order. That means each row transitions from 0's to 1's. The task is to find the index of the leftmost column that contains at least one '1'. If there's no such column, we return `-1`.

Access to the matrix is restricted to two operations. One is `BinaryMatrix.get(row, col)`, which returns the value at a specified row and column, and the other is `BinaryMatrix.dimensions()`, which returns the number of rows and columns.

A key constraint is that any solution must limit the number of calls to `BinaryMatrix.get` to 1000, making straightforward solutions like scanning every element inefficient and infeasible.

## Intuition

The problem is essentially asking us to find the smallest column index that contains the value '1'. Given that each row is sorted, a linear scan from the leftmost column to the rightmost in each row would be wasteful since we might traverse a lot of zeros before finding a one, especially in large matrices.

Instead, the optimal approach is to utilize binary search within each row. Binary search can quickly home in on a '1' in a sorted array by repeatedly halving the search space. For each row, we perform a binary search to find the leftmost '1'. We keep track of the smallest column index across all rows.

With the provided solution, we iterate over each row and perform a binary search. If we find a '1', we update the result to be the minimum of the current result and the column index where '1' was found.

The binary search within each row is optimized with the condition `while left < right`, which keeps narrowing down the range until `left` and `right` converge. When they converge, we check if the leftmost position contains a '1' to consider it as a candidate for our answer.

Since we are only making a couple of calls to `BinaryMatrix.get` per row (one for each step of the binary search), we stay within the limit of 1000 calls and arrive at an efficient solution.

## Solution Approach

The solution follows the Binary Search algorithm to narrow down the leftmost column that contains a '1'. Since we know that each row is sorted, this property allows us to use binary search to efficiently find the first occurrence of '1' per row, if it exists.

The process of binary search starts by initializing two pointers, `left` and `right`, which represent the range of columns we're searching in. At the start, `left` is set to 0, and `right` is set to the last column index `cols - 1`. The middle of the range, `mid`, is then calculated by averaging `left` and `right`.

We then enter a loop that continues to narrow down the range by checking if the `mid` position contains a '0' or a '1'. If a '1' is found, it means we should continue our search to the left side of `mid` to find the leftmost '1', so we set `right` to `mid`. If a '0' is found, we move our search to the right side by setting `left` to `mid + 1`. The loop stops once `left` and `right` meet, which means we have either found the leftmost '1' for that row or there is no '1' in that row.

A key detail in this implementation is that after the binary search per row, an additional check is done on the `left` position. If `BinaryMatrix.get(row, left)` returns '1', then we potentially found a new leftmost '1', and we update the result `res` accordingly. We use `res` to store the current smallest index where a '1' was found across all rows processed until that point, initializing it with `-1`.

This step is crucial as it allows us to progressively update the position of the leftmost '1' column, and by the end of the row iterations, we will have the index of the overall leftmost column with a '1'. If `res` remains `-1` by the end of the loops, it means there were no '1's in the binary matrix, so we return `-1`.

The algorithm efficiently leverages the sorted property of the rows and minimizes reads, ensuring we stay within the 1000_api_calls limit while still achieving a time complexity of O(logN), where N is the number of columns, multiplied by the number of rows M, yielding an overall time complexity of O(MlogN).

### Example Walkthrough

Let's illustrate the solution approach with a small example. Consider the binary matrix:

```
1 |
2  [0, 0, 0, 1],
3  [0, 1, 1, 1],
4  [0, 0, 1, 1],
5  [0, 0, 0, 0]
6 |
```

We need to find the leftmost column that contains a '1'. Here's how we approach the problem:

1. We start by getting the dimensions of the matrix. Let's assume dimensions are 4 rows and 4 columns.
2. We initialize our result (`res`) to -1, which would be the final answer if there's no '1' in the matrix.
3. We loop through each row and perform a binary search within that row:
   - For the first row, we initialize `left = 0` and `right = 3` (since there are 4 columns, indices start from 0).
   - We calculate the middle, `mid = (left + right) / 2`, which is initially 1. Since `BinaryMatrix.get(0, mid)` is 0, we now know that if a '1' exists in this row, it must be to the right. So we move `left` to `mid + 1`.
   - We recalculate `mid = (left + right) / 2`, which is now 2. Again, `BinaryMatrix.get(0, mid)` is 0, so we move `left` to `mid + 1`.
   - Now `left` is 3, and our loop terminates since `left >= right`.
   - After the loop, we check `BinaryMatrix.get(0, left)`, which is 1, meaning we've found a column with a '1'. We update `res` to `left`, which is 3, the current smallest index of a found '1'.
4. We repeat this for the remaining rows:
   - For the second row, our binary search will quickly converge to `left = 1`, the first '1'. We update `res` to `min(res, left)`, so now `res` becomes 1.
   - The third row also leads to updating `res` to `min(res, left)` after binary search, but since the leftmost '1' is in column 2, `res` remains 1.
   - The fourth row has no '1's, so it doesn't change `res`.
5. Finally, since we have gone through all rows, `res` holds the index of the leftmost column containing a '1', which in this example is 1.

This approach limits the number of calls to `BinaryMatrix.get` to 2-3 per row, which is well below the limit of 1000 and is efficient even for large matrices, with a final time complexity of O(MlogN).

## Python Solution

```python
class Solution:
    def leftMostColumnWithOne(self, binaryMatrix: 'BinaryMatrix') -> int:
        # Get the dimensions of the binary matrix
        rows, cols = binaryMatrix.dimensions()
        # Initialize result as -1 to represent no 1's found yet
        res = -1
        # Iterate over each row to find the leftmost column with a 1
        for row in range(rows):
            left, right = 0, cols - 1
            # Perform a binary search to find the leftmost 1 in the current row
            while left < right:
                mid = (left + right) // 2  # Use integer division for Python 3
                if binaryMatrix.get(row, mid) == 1:
                    right = mid  # Move the right pointer to the middle if a 1 is found
                else:
                    left = mid + 1  # Move the left pointer past the middle otherwise

            # After exiting the loop, check if the current leftmost index contains a 1
            if binaryMatrix.get(row, left) == 1:
                # If a 1 is found and result is -1 (first 1 found), or if the current column
                # is less than the previously recorded result, update result
                result = left if result == -1 else min(result, left)

        # Return the result, which is the index of the leftmost 1 or -1 if no 1 is found
        return result
```

## Java Solution

```java
// Define the solution class implementing the algorithm to find the leftmost column with at least one '1' in a binary matrix.
class Solution {

    // Method to find the leftmost column index with a '1'.
    public int leftMostColumnWithOne(BinaryMatrix binaryMatrix) {
        // Retrieve the dimensions of the binary matrix.
        List<Integer> dimensions = binaryMatrix.dimensions();
        int rows = dimensions.get(0); // Number of rows in the binary matrix.
        int cols = dimensions.get(1); // Number of columns in the binary matrix.

        // Initial result is set to -1 (indicating no '1' has been found yet).
        int result = -1;

        // Iterate through each row to find the leftmost '1'.
        for (int row = 0; row < rows; ++row) {
            // Initialize pointers for the binary search.
            int left = 0;
            int right = cols - 1;

            // Perform binary search in the current row to find the '1'.
            while (left < right) {
                // Calculate middle index.
                int mid = left + (right - left) / 2; // Avoids potential overflow.

                // Check the value at the middle index and narrow down the search range.
                if (binaryMatrix.get(row, mid) == 1) {
                    // '1' is found, shift towards the left (lower indices).
                    right = mid;
                } else {
                    // '0' is found, shift towards the right (higher indices).
                    left = mid + 1;
                }
            }

            // After the loop, 'left' is the position of the leftmost '1' or the first '0' after all '1's in this row.
            if (binaryMatrix.get(row, left) == 1) {
                // Update the result if a '1' was found column with a '1',
                // or if the current '1' is to the left of the previously recorded result.
                result = left;
            } else { // If it's not the first, we take the lesser (more left) of the two columns.
                result = Math.min(result, left);
            }
        }

        // Return the result. If no '1' was found, -1 will be returned, indicating such.
        return result;
    }
}
```

## C++ Solution

```cpp
class Solution {
public:
    // Function to find the leftmost column with a 1 in a binary matrix
    int leftMostColumnWithOne(BinaryMatrix& binaryMatrix) {
        // Retrieve the dimensions of the binary matrix
        vector<int> dimensions = binaryMatrix.dimensions();
        int rows = dimensions[0]; // Number of rows
        int cols = dimensions[1]; // Number of columns

        int result = -1; // Initialize result to -1 to indicate no 1 found

        // Iterate through each row to find the leftmost 1
        for (int row = 0; row < rows; ++row) {
            int left = 0;       // Starting index of the current row
            int right = cols - 1; // Ending index of the current row

            // Binary search to find the leftmost 1 in the current row
            while (left < right) {
                // Find the middle index
                int mid = left + (right - left) / 2; // The '+' 1' has been replaced with '/ 2' for clarity

                if (binaryMatrix.get(row, mid) == 1) {
                    // If a 1 is found, narrow the search to the left half
                    right = mid;
                } else {
                    // If a 0 is found, narrow the search to the right half
                    left = mid + 1;
                }
            }

            // After binary search, check if we have found a 1 at the 'left' index
            // and update the result accordingly
            if (binaryMatrix.get(row, left) == 1) {
                // If this is the first 1 found, or if the current index is smaller than previous one
                if (result == -1) {
                    result = left;  // Update the result with the current index
                } else {
                    result = min(result, left); // Update the result with the smaller index
                }
            }
        }

        // Return the result
        return result;
    }
};
```

## Typescript Solution

```typescript
// Interface representing the BinaryMatrix's methods for TypeScript type checking
interface BinaryMatrix {
    get(row: number, col: number): number;
    dimensions(): number[];
}

// Variable to hold the leftmost column with a 1 in a binary matrix
let leftMostColumnResult: number = -1;

// Function to find the leftmost column with a 1 in a binary matrix
function leftMostColumnWithOne(binaryMatrix: BinaryMatrix): number {
    // Retrieve the dimensions of the binary matrix
    let dimensions: number[] = binaryMatrix.dimensions();
    let rows: number = dimensions[0]; // Number of rows
    let cols: number = dimensions[1]; // Number of columns

    leftMostColumnResult = -1; // Initialize the result to -1 to indicate no 1 has been found

    // Iterate through each row to find the leftmost 1
    for (let row = 0; row < rows; ++row) {
        let left = 0;       // Starting index of the current row
        let right = cols - 1; // Ending index of the current row

        // Binary search to find the leftmost 1 in the current row
        while (left < right) {
            // Find the middle index
            let mid = Math.floor((left + right) / 2); // Use floor to avoid decimal indices

            if (binaryMatrix.get(row, mid) === 1) {
                // If a 1 is found, narrow the search to the left half
                right = mid;
            } else {
                // If a 0 is found, narrow the search to the right half
                left = mid + 1;
            }
        }

        // After binary search, check if we have found a 1 at the 'left' index
        // and update the result accordingly
        if (binaryMatrix.get(row, left) === 1) {
            // If this is the first 1 found, or if the current index is smaller than the previously found one
            if (leftMostColumnResult === -1 ||
                left < leftMostColumnResult) {
                leftMostColumnResult = left;  // Update the result with the current index
            } else {
                leftMostColumnResult = Math.min(leftMostColumnResult, left); // Update the result with the smaller index
            }
        }
    }

    // Return the result
    return leftMostColumnResult;
}

// Example usage:
// let matrix: BinaryMatrix = new SomeBinaryMatrixImplementation();
// console.log(leftMostColumnWithOne(matrix));
```

## Time and Space Complexity

### Time Complexity

The provided code loops through each row of the binary matrix to find the leftmost column containing a `1`. For each row, the code implements a binary search which has a logarithmic time complexity. Since the binary search is conducted for each of the `rows`, the total time complexity is `O(rows * log(cols))`.

Here's the breakdown of the time complexity:

- We iterate over each row once: `O(rows)`
- For each row, a binary search is conducted within `cols`, which takes `O(log(cols))`

Thus, the combined time complexity is `O(rows * log(cols))`.

### Space Complexity

The space complexity is `O(1)` since we are only using a constant amount of extra space. The variables `res`, `left`, `right`, `mid`, and the loop counter `row` do not depend on the size of the input and use a fixed amount of space.