# 74. Search a 2D Matrix

Medium Array Binary Search Matrix

Leetcode Link

## Problem Description

You are given a two-dimensional array, referred to as a 'matrix', where each row is sorted in ascending order, and the first element of each row is greater than the last element of the previous row. Your task is to check whether a given integer, 'target', exists within this matrix. To solve this problem efficiently, you must devise an algorithm that operates with a time complexity of `O(log(m * n))`, where `m` is the number of rows and `n` is the number of columns in the matrix. This time complexity hint suggests that we should consider a binary search approach, as it can perform searches in logarithmic time.

## Intuition

Given the properties of the matrix, we can treat it as a sorted list. The idea here is that even though the matrix has two dimensions, the sorted nature of the rows and the rule that the first element of the current row is greater than the last element of the previous row means that a logical ordering exists as if all elements were in a single sorted list.

We can then apply a binary search on this 'virtual' sorted list. First, we initialize two pointers, `left` and `right`, which represent the start and end of the possible space where the target can be located. These are initially set to point to the first and last indices of this virtual list (0 and `m * n - 1`, respectively).

In each iteration of the binary search, we calculate the middle index, `mid`. We then convert `mid` back into two dimensions, `x` and `y`, using the divmod function, to compare the matrix element at this position against the target. If the element at `[x][y]` is greater than or equal to the target, we bring the `right` pointer to `mid`, thereby discarding the second half of the search space. If the element is smaller, we increase the `left` pointer to `mid + 1`, discarding the first half of the search space. This process continues until the space is narrowed down to a single element.

Finally, we check if the element at this narrowed down position is equal to the target. If so, we return true; otherwise, false. The binary search guarantees that we will find the target if it exists in the matrix within `O(log(m * n))` time complexity.

## Solution Approach

The solution approach implements a binary search algorithm, which is a classic method used to find an element within a sorted list in logarithmic time. However, in this context, we are using binary search on a two-dimensional matrix by treating it as if it were a linear array. Here's how it's done:

- **Variables Setup**: We start by calculating the number of rows `m` and columns `n` of the matrix. We then set two pointers, `left` to 0 and `right` to `m * n - 1`, which represent the range of possible indexes where our target might be if we were to flatten the matrix into a single list.
- **While Loop**: The search process continues while `left` is less than `right`. This loop will terminate when `left` and `right` converge on the index where the target either is or would be if it were in the matrix.
- **Mid Calculation and Conversion**: Inside the loop, the mid-point index is calculated using the expression `(left + right) >> 1`, which is a bitwise operation equivalent to dividing the sum of `left` and `right` by two.
- **Index Flattening**: The mid index is then flattened into two dimensions, `x` and `y`, where `x` is the row number and `y` is the column number. This is achieved using the `divmod` function, where `mid` is divided by the number of columns `n`, `x` takes the quotient, and `y` takes the remainder of this division.
- **Search Space Halving**: Depending on the value at `matrix[x][y]`, we adjust our search space:
  - If `matrix[x][y]` is greater than or equal to the target, it means the target, if present, should be to the left, including the current mid. We then move the `right` pointer to `mid`.
  - Conversely, if `matrix[x][y]` is less than the target, the target can only be on the right, excluding the current mid. We then set the `left` pointer to `mid + 1`.
- **Final Check**: After the while loop, `left` should point to the index where the target would be if it's present. We take the `left` index and convert it back into two-dimensional indices to access the matrix element. If `matrix[left // n][left % n]` is equal to the target, we return `true`, indicating the target is present in the matrix.

This binary search algorithm effectively flattens the 2D search space into 1D, allowing us to utilize the time efficiency of binary search in a multidimensional context.

### Example Walkthrough

Let's illustrate the solution approach with a small example. Suppose we have the following matrix and we are looking for the target value `9`.

```
Matrix:
1  3  5
7  9 11

Target:
9
```

Here's what the solution would look like step by step:

1. **Variables Setup**: We start by determining that the matrix has `2` rows (`m`) and `3` columns (`n`). We set our `left` pointer to `0` and our `right` pointer to `m * n - 1`, which is `5` in this case (flattening the matrix to a single list would have indices ranging from `0` to `5`).
2. **While Loop Begins**: Since `left` (`0`) is less than `right` (`5`), we continue our search.
3. **Mid Calculation and Conversion**: We calculate the middle index with the expression `(left + right) >> 1`, which gives us `(0 + 5) >> 1 = 2`. Now we need to convert this index back to two dimensions. Dividing `2` by the number of columns (`3`) gives us a quotient of `0` (row index `x`) and a remainder of `2` (column index `y`).
4. **Index Flattening**: The element at the middle index, which is in the first row and third column of our matrix, is the number `5`. We compare this with our target `9`.
5. **Search Space Halving**:
   - Since `5` is less than `9`, we can eliminate the first half of the search space and update our `left` index to `mid + 1`, which is `3`.
6. **While Loop Continues**: With the updated `left` being `3` and `right` still `5`, we continue our search.
7. **Mid Calculation and Conversion**: Recalculate `mid` with `(left + right) >> 1`, which gives us `(3 + 5) >> 1 = 4`. This `mid` index corresponds to the second row and second column in our matrix. We divide `4` by the number of columns (`3`) to get `x = 1` and `y = 1`.
8. **Index Flattening**: The element at index `4` is `9` when looking at the matrix, which is exactly our target.
9. **Final Check**: Since `matrix[x][y]` is equal to the target `9`, we have successfully found the target in our matrix! The loop terminates and we return `true`.

By following these steps, this approach uses binary search to resolve the search problem within a two-dimensional matrix efficiently.

## Python Solution

```python
class Solution:
    def searchMatrix(self, matrix: List[List[int]], target: int) -> bool:
        # Get the number of rows and columns in the matrix
        num_rows, num_columns = len(matrix), len(matrix[0])

        # Initialize pointers for the binary search
        left, right = 0, num_rows * num_columns - 1

        # Conduct a binary search in the matrix
        while left < right:
            # Calculate the middle index between left and right
            mid = (left + right) >> 1  # Equivalent to floor division by 2 (mid = (left + right) // 2)
            # Convert the 1D representation mid back to 2D indices x and y
            row, column = divmod(mid, num_columns)
            # If the middle element is greater or equal to the target, go left
            if matrix[row][column] >= target:
                right = mid
            # If the middle element is less than the target, go right
            else:
                left = mid + 1

        # After the loop, left should point to the target element if it exists
        # Check if the target is indeed at the (left // num_columns, left % num_columns) position
        return matrix[left // num_columns][left % num_columns] == target

```

## Java Solution

```java
// Class name should start with an uppercase letter following Java naming conventions.
class Solution {

    // Method to search for a target value in a matrix.
    public boolean searchMatrix(int[][] matrix, int target) {
        // Get the number of rows and columns from the matrix.
        int rows = matrix.length, cols = matrix[0].length;

        // Initialize the left and right pointers for the binary search.
        int left = 0, right = rows * cols - 1;

        // Loop until the search space is reduced to one element.
        while (left < right) {
            // Calculate the middle point of the current search space.
            int mid = (left + right) / 2; // Shift operator can also be used (left + right) >> 1

            // Map the middle index to a 2D position in the matrix.
            int x = mid / cols, y = mid % cols;

            // Compare the middle element with the target.
            if (matrix[x][y] >= target) {
                // If the middle element is greater or equal to the target,
                // narrow the search to the left half including mid.
                right = mid;
            } else {
                // If the middle element is less than the target,
                // narrow the search to the right half excluding mid.
                left = mid + 1;
            }
        }

        // After exiting the loop, left should point to the target element, if it exists.
        // Check if the element at the 'left' position equals the target.
        return matrix[left / cols][left % cols] == target;
    }
}

```

## C++ Solution

```cpp
#include <vector>

class Solution {
public:
    // Function to search for a target value in a 2D matrix.
    bool searchMatrix(std::vector<std::vector<int>>& matrix, int target) {
        int rowCount = matrix.size();           // Number of rows in the matrix
        int colCount = matrix[0].size();        // Number of columns in the matrix
        int left = 0, right = rowCount * colCount - 1; // Set search range within the flattened matrix

        // Binary search
        while (left < right) {
            int mid = left + (right - left) / 2;  // Find the mid index
            int row = mid / colCount;             // Compute row index from mid
            int col = mid % colCount;             // Compute column index from mid

            // Compare the element at [row][col] with target
            if (matrix[row][col] >= target) {
                // If the element is greater than or equal to target, move the right boundary left
                right = mid;
            } else {
                // If the element is less than target, move the left boundary right
                left = mid + 1;
            }
        }

        // Check if the target is at the final search point
        return matrix[left / colCount][left % colCount] == target;
    }
};

```

## Typescript Solution

```typescript
// Function to perform binary search in a 2D matrix where each row and column is sorted.
// It returns true if the target number is found, otherwise returns false.
function searchMatrix(matrix: number[][], target: number): boolean {
    // Get the number of rows in the matrix
    const numRows = matrix.length;
    // Get the number of columns in the matrix
    const numCols = matrix[0].length;
    // Initialize the left pointer to start the binary search
    let left = 0;
    // Initialize the right pointer to the end of the logical 1D representation of the matrix
    let right = numRows * numCols;

    // Loop until the pointers meet, which means the target is not found if it exits the loop
    while (left < right) {
        // Calculate the mid point of the current search space
        const mid = Math.floor((left + right) / 2);
        // Compute the row index from the mid point
        const rowIndex = Math.floor(mid / numCols);
        // Compute the column index from the mid point
        const colIndex = mid % numCols;

        // If the element at mid position equals the target, return true for found
        if (matrix[rowIndex][colIndex] === target) {
            return true;
        }

        // Otherwise, move the search space depending on the comparison with the target
        if (matrix[rowIndex][colIndex] < target) {
            left = mid + 1;  // Move the left pointer to narrow the search space
        } else {
            right = mid;  // Move the right pointer to narrow the search space
        }
    }

    // If we exit the loop, the target was not found in the matrix
    return false;
}

```

## Time and Space Complexity

The time complexity of the searchMatrix function is `O(log(m*n))` where `m` is the number of rows and `n` is the number of columns in the matrix. This is because the function performs a binary search on a virtual flattened list representation of the matrix which has `m*n` elements. The search converges by halving the candidate range in each iteration.

The space complexity of the searchMatrix function is `O(1)`. This is achieved as no additional storage that scales with the input size is being used. All the operations are done in place with a few auxiliary variables that occupy constant space.