240. Search a 2D Matrix II

Medium | Array | Binary Search | Divide and Conquer | Matrix | Leetcode Link

Problem Description

The problem is to create an algorithm that can efficiently search for a specific value, called `target`, within a 2-dimensional matrix. The dimensions of the matrix are `m x n`, which means it has `m` rows and `n` columns. The matrix is not just any random assortment of integers—it has two key properties that can be leveraged to make searching efficient:

1. Each row of the matrix is sorted in ascending order from left to right.
2. Each column of the matrix is sorted in ascending order from top to bottom.

Given these sorted properties of the matrix, the search should be done in a way that is more optimized than a brute-force approach which would check every element.

Intuition

To intuitively understand the solution, we should recognize that because of the row and column properties, the matrix resembles a 2D binary search problem. However, instead of splitting our search space in half each time as we would in a conventional binary search, we can make an observation:

- If we start in the bottom-left corner of the matrix (or the top-right), we find ourselves at an interesting position: moving up decreases the value (since columns are sorted in ascending order from top to bottom), and moving right increases the value (since rows are sorted in ascending order from left to right).

This starting point gives us a "staircase" pattern to follow based on comparisons:

1. If the `target` is greater than the value at our current position, we know it can't be in the current row (to the left), so we move to the right (increase the column index).
2. If the `target` is less than the value at our current position, we know it can't be in the current column (below), so we move up (decrease the row index).

By doing this, we are eliminating either a row or a column at each step, leveraging the matrix's properties to find the `target` or conclude it's not there. We keep this up until we find the `target` or exhaust all our moves (when we move out of the bounds of the matrix), in which case the `target` is not present. This is why the while loop condition in the code checks if `i >= 0 and j < n`.

The process is very much like tracing a path through the matrix that "zig-zags" closer and closer to the value if it's present. The solution here effectively combines aspects of both binary search and linear search but applied in a 2-dimensional space.

Solution Approach

The solution provided is a direct implementation of the intuitive strategy discussed previously. Here's how the solution is implemented:

- We declare a `Solution` class with a method `searchMatrix` that accepts a 2D matrix and the `target` value as parameters.

- Inside the `searchMatrix` method, we start by getting the dimensions of the matrix: `m` for the number of rows and `n` for the number of columns, which will be used for boundary checking during the search.

- We initiate two pointers, `i` and `j`, which will traverse the matrix. `i` is initialized to `m - 1`, which means it starts from the last row (bottom row), and `j` is initialized to `0`, which is the first column (leftmost column). This represents the bottom-left corner of the matrix.

- The search begins and continues as long as our pointers are within the bounds of the matrix. The `while` loop condition makes sure that `i` is never less than `0` (which would mean we've moved above the first row) and `j` is less than `n` (to ensure we don't move beyond the last column to the right).

- Within the loop, there are three cases to consider:

  1. If the element at the current position `matrix[i][j]` equals the `target`, then our search is successful, and we return `True`.
  2. If the element at `matrix[i][j]` is greater than the `target`, we must move up (decrease the value of `i`) to find a smaller element.
  3. Conversely, if `matrix[i][j]` is less than the `target`, we must move right (increase the value of `j`) in hopes of finding a larger element.
- If we exit the loop without returning `True`, it means we have exhausted all possible positions in the matrix without finding the `target`, and thus we return `False`.

This approach effectively traverses the matrix in a manner such that with each comparison, a decision can be made to eliminate either an entire row or an entire column, significantly reducing the search space and making the algorithm efficient. The worst case scenario would be traversing from the bottom-left corner to the top-right corner, which gives us a time complexity of O(m + n), where `m` is the number of rows and `n` is the number of columns.

Example Walkthrough

Let's walk through a small example to illustrate the solution approach. Consider the following 3×4 matrix and a target value to search for:

```
[ [1, 4, 7, 11],
  [2, 5, 8, 12],
  [3, 6, 9, 16] ]
```

And let's say we are searching for the target value `5`.

Using the solution approach, we start at the bottom-left corner of the matrix. This means our starting position is at the last row, first column: `matrix[2][0]`, which is `3`.

Now, we compare the target value `5` with the value at our current position:

1. `target` (5) > `matrix[2][0]` (3) so the target can't be in the current row because all values to the left are smaller. We move right to increase the value (increment `j`): now we are at `matrix[2][1]`, which is `6`.

2. `target` (5) < `matrix[2][1]` (6) so the target can't be in the current column because all values below are larger. We move up to decrease the value (decrement `i`): now we are at `matrix[1][1]`, which is `5`.

At this point, `matrix[1][1]` equals the target value `5`, so our search is successful, and we return `True`.

This approach avoided checking every single element in the matrix, instead, by leveraging the sorted nature of the rows and columns, it quickly hones in on the target with a clear strategy. Even if the target number was not present, the process would eventually move out of the matrix bounds, at which point we would return `False`, signifying that the target is not found.

Python Solution

```python
class Solution:
    def searchMatrix(self, matrix, target):
        # Rows and columns in the matrix
        num_rows, num_cols = len(matrix), len(matrix[0])

        # Start from the bottom left corner of the matrix
        row_index, col_index = num_rows - 1, 0

        # Loop until we have a valid position within the matrix bounds
        while row_index >= 0 and col_index < num_cols:
            # Check if the current element is the target
            if matrix[row_index][col_index] == target:
                return True

            # If current element is larger than target, move up to reduce value
            if matrix[row_index][col_index] > target:
                row_index -= 1
            # If current element is smaller than target, move right to increase value
            else:
                col_index += 1

        # Return False if we haven't returned True by this point
        return False

```

Java Solution

```java
class Solution {

    /**
     * Searches for a target value in a 2D matrix.
     * The matrix has the following properties:
     * - Integers in each row are sorted in ascending from left to right.
     * - The first integer of each row is greater than the last integer of the previous row.
     *
     * @param matrix 2D matrix of integers
     * @param target The integer value to search for
     * @return boolean indicating whether the target is found
     */
    public boolean searchMatrix(int[][] matrix, int target) {
        // Get the number of rows and columns in the matrix
        int rowCount = matrix.length;
        int colCount = matrix[0].length;

        // Start from the bottom-left corner of the matrix
        int row = rowCount - 1;
        int col = 0;

        // Perform a staircase search
        while (row >= 0 && col < colCount) {
            if (matrix[row][col] == target) {
                // Target is found at the current position
                return true;
            }
            if (matrix[row][col] > target) {
                // Target is less than the current element, move up
                row--;
            } else {
                // Target is greater than the current element, move right
                col++;
            }
        }

        // Target was not found in the matrix
        return false;
    }
}

```

C++ Solution

```cpp
#include <vector>
using namespace std;

class Solution {
public:
    // Searches for a target value within a 2D matrix. This matrix has the following properties:
    // Integers in each row are sorted in ascending from left to right.
    // Integers in each column are sorted in ascending from top to bottom.
    //
    // @param matrix The matrix of integers.
    // @param target The target integer to find.
    // @return True if target is found, false otherwise.
    bool searchMatrix(vector<vector<int>>& matrix, int target) {
        // Get the number of rows.
        int rows = matrix.size();
        // Get the number of columns.
        int columns = matrix[0].size();

        // Start from the bottom-left corner of the matrix.
        int currentRow = rows - 1;
        int currentColumn = 0;

        // While the position is within the bounds of the matrix...
        while (currentRow >= 0 && currentColumn < columns) {
            // If the current element is the target, return true.
            if (matrix[currentRow][currentColumn] == target) return true;

            // If the current element is larger than the target, move up one row.
            if (matrix[currentRow][currentColumn] > target) {
                --currentRow;
            }
            // If the current element is smaller than the target, move right one column.
            else {
                ++currentColumn;
            }
        }

        // If the target is not found, return false.
        return false;
    }
};

```

Typescript Solution

```typescript
/**
 * Searches for a target value in a matrix.
 * This matrix has the following properties:
 * 1. Integers in each row are sorted from left to right.
 * 2. The first integer of each row is greater than the last integer of the previous row.
 * @param matrix A 2D array of numbers representing the matrix.
 * @param target The number to search for in the matrix.
 * @return A boolean indicating whether the target exists in the matrix.
 */
function searchMatrix(matrix: number[][], target: number): boolean {
    // Get the number of rows (m) and columns (n) in the matrix
    let rowCount = matrix.length,
        columnCount = matrix[0].length;

    // Start our search from the bottom-left corner of the matrix
    let currentRow = rowCount - 1,
        currentColumn = 0;

    // Continue the search while we're within the bounds of the matrix
    while (currentRow >= 0 && currentColumn < columnCount) {
        // Retrieve the current element to compare with the target
        let currentElement = matrix[currentRow][currentColumn];

        // Check if the current element matches the target
        if (currentElement === target) return true;

        // If the current element is greater than the target,
        // move up to the previous row since all values in the current
        // row will be too large given the matrix's sorted properties
        if (currentElement > target) {
            --currentRow;
        } else {
            // If the current element is less than the target,
            // move right to the next column since all values in previous
            // columns will be too small given the matrix's sorted properties
            ++currentColumn;
        }
    }

    // If we've exited the while loop, the target is not present in the matrix
    return false;
}

```

Time and Space Complexity

Time Complexity

The time complexity of the search algorithm is `O(m + n)`, where `m` is the number of rows and `n` is the number of columns in the matrix. This is because the algorithm starts from the bottom-left corner of the matrix and moves either up (`i` decreases) or right (`j` increases) at each step. At most, it will move `m` steps upwards and `n` steps to the right before it either finds the target or reaches the top-right corner, thus completing the search.

Space Complexity

The space complexity of the algorithm is `O(1)`. This is because the algorithm uses a fixed amount of extra space (variables `i` and `j`) regardless of the size of the input matrix. It doesn't require any additional data structures that grow with the input size.