Search in a row and column-wise sorted matrix

Problem Statement: You have been given a 2-D array **'mat'** of size **'N x M'** where **'N'** and **'M'** denote the number of rows and columns, respectively. The elements of each row and each column are sorted in non-decreasing order.
But, the first element of a row is not necessarily greater than the last element of the previous row (if it exists).
You are given an integer **'target'**, and your task is to find if it exists in the given 'mat' or not.

Examples

```
Example 1:
Input Format: N = 5, M = 5, target = 14
mat[] =

Result: true
Explanation: Target 14 is present in the cell (3, 2)(0-based indexing) of the matrix. So, the answer
is true.

Example 2:
Input Format: N = 3, M = 3, target = 12,

mat[] =

Result: false
Explanation: As target 12 is not present in the matrix, the answer is false.
```

Disclaimer: *Don't jump directly to the solution, try it out yourself first.*

Brute Force Approach
Algorithm / Intuition

One key point to notice here is that the first element of a row is not necessarily greater than the last element of the previous row (if it exists). This means the matrix is not necessarily entirely sorted although each row and column is sorted in non-decreasing order.

The extremely naive approach is to get the answer by checking all the elements of the given matrix. So, we will traverse the matrix and check every element if it is equal to the given 'target'.

Algorithm:

1. We will use a loop(say i) to select a particular row at a time.
2. Next, for every row, we will use another loop(say j) to traverse each column.
3. Inside the loops, we will check if the element i.e. matrix[i][j] is equal to the 'target'. If we found any matching element, we will return true.
4. Finally, after completing the traversal, if we found no matching element, we will return false.

Code

```java
import java.util.*;

public class tUf {
    public static boolean searchElement(ArrayList<ArrayList<Integer>> matrix, int target) {
        int n = matrix.size(), m = matrix.get(0).size();

        // traverse the matrix:
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < m; j++) {
                if (matrix.get(i).get(j) == target)
                    return true;
            }
        }
        return false;
    }

    public static void main(String[] args) {
        ArrayList<ArrayList<Integer>> matrix = new ArrayList<>();
        matrix.add(new ArrayList<>(Arrays.asList(1, 4, 7, 11, 15)));
        matrix.add(new ArrayList<>(Arrays.asList(2, 5, 8, 12, 19)));
        matrix.add(new ArrayList<>(Arrays.asList(3, 6, 9, 16, 22)));
        matrix.add(new ArrayList<>(Arrays.asList(10, 13, 14, 17, 24)));
        matrix.add(new ArrayList<>(Arrays.asList(18, 21, 23, 26, 30)));

        boolean result = searchElement(matrix, 8);
        System.out.println(result ? "true" : "false");
    }
}
```

Output: true

Complexity Analysis

Time Complexity: O(N X M), where N = given row number, M = given column number.
Reason: In order to traverse the matrix, we are using nested loops running for n and m times respectively.

Space Complexity: O(1) as we are not using any extra space.

Better Approach
Algorithm / Intuition

We are going to use the Binary Search algorithm to optimize the approach.

The primary objective of the Binary Search algorithm is to efficiently determine the appropriate half to eliminate, thereby reducing the search space by half. It does this by determining a specific condition that ensures that the target is not present in that half.

The question specifies that each row in the given matrix is sorted. Therefore, to determine if the target is present in a specific row, we don't need to search column by column. Instead, we can efficiently use the binary search algorithm.

Algorithm:

1. We will use a loop(say i) to select a particular row at a time.
2. Next, for every row, i, we will check if it contains the target using binary search.
    1. After applying binary search on row, i, if we found any element equal to the target, we will return true. Otherwise, we will move on to the next row.
3. Finally, after completing all the row traversals, if we found no matching element, we will return false.

Code
```java
import java.util.*;

public class tUf {
    public static boolean binarySearch(ArrayList<Integer> nums, int target) {
        int n = nums.size(); //size of the array
        int low = 0, high = n - 1;

        // Perform the steps:
        while (low <= high) {
            int mid = (low + high) / 2;
            if (nums.get(mid) == target) return true;
            else if (target > nums.get(mid)) low = mid + 1;
            else high = mid - 1;
        }
        return false;
    }

    public static boolean searchElement(ArrayList<ArrayList<Integer>> matrix, int target) {
        int n = matrix.size();
        int m = matrix.get(0).size();

        for (int i = 0; i < n; i++) {
            boolean flag = binarySearch(matrix.get(i), target);
            if (flag == true) return true;
        }
        return false;
    }


    public static void main(String[] args) {
        ArrayList<ArrayList<Integer>> matrix = new ArrayList<>();
        matrix.add(new ArrayList<>(Arrays.asList(1, 4, 7, 11, 15)));
        matrix.add(new ArrayList<>(Arrays.asList(2, 5, 8, 12, 19)));
        matrix.add(new ArrayList<>(Arrays.asList(3, 6, 9, 16, 22)));
        matrix.add(new ArrayList<>(Arrays.asList(10, 13, 14, 17, 24)));
        matrix.add(new ArrayList<>(Arrays.asList(18, 21, 23, 26, 30)));

        boolean result = searchElement(matrix, 8);
        System.out.println(result ? "true" : "false");
    }
}
```
Output: true

Complexity Analysis

Time Complexity: O(N*logM), where N = given row number, M = given column number.
Reason: We are traversing all rows and it takes O(N) time complexity. And for all rows, we are applying binary search. So, the total time complexity is O(N*logM).

Space Complexity: O(1) as we are not using any extra space.

Optimal Approach
Algorithm / Intuition

We can enhance this method by adjusting how we move through the matrix. Let's take a look at the four corners: (0, 0), (0, m-1), (n-1, 0), and (n-1, m-1). By observing these corners, we can identify variations in how we traverse the matrix.

Assume the given ‘target’ = 14 and given matrix =

1	4	7	11	15
2	5	8	12	19
3	6	9	16	22
10	13	14	17	24
18	21	23	26	30

Observations:

- **Cell (0, 0):** Assume we are starting traversal from (0, 0) and we are searching for 14. Now, this row and column are both sorted in increasing order. So, we cannot determine, how to move i.e. row-wise or column-wise. That is why, ***we cannot start traversal from (0, 0).***
- **Cell (0, m-1):** Assume we are starting traversal from (0, m-1) and we are searching for 14. Now, in this case, the row is in decreasing order and the column is in increasing order. Therefore, if ***we start traversal from (0, m-1), in the following way, we can easily determine how we should move.***
  - **If matrix[0][m-1] > target:** We should move row-wise.
  - **If matrix[0][m-1] < target:** We need bigger elements and so we should move column-wise.
- **Cell (n-1, m-1):** Assume we are starting traversal from (n-1, m-1) and we are searching for 14. Now, this row and column are both sorted in decreasing order. So, we cannot determine, how to move i.e. row-wise or column-wise. That is why, ***we cannot start traversal from (n-1, m-1).***
- **Cell (n-1, 0):** Assume we are starting traversal from (n-1, 0) and we are searching for 14. Now, in this case, the row is in increasing order and the column is in decreasing order. Therefore, if ***we start traversal from (n-1, 0), in the following way, we can easily determine how we should move.***
  - **If matrix[n-1][0] < target:** We should move row-wise.
  - **If matrix[n-1][0] > target:** We need smaller elements and so we should move column-wise.

From the above observations, it is quite clear that we should start the matrix traversal from either the cell (0, m-1) or (n-1, 0).

Note: Here in this approach, we have chosen the cell (0, m-1) to start with. You can choose otherwise.

Using the above observations, we will start traversal from the cell (0, m-1) and every time we will compare the target with the element at the current cell. After comparing we will either eliminate the row or the column accordingly like the following:

- **If current element > target:** We need the smaller elements to reach the target. But the column is in increasing order and so it contains only greater elements. So, we will eliminate the column by decreasing the current column value by 1(*i.e. col--*) and thus we will move row-wise.
- **If current element < target:** In this case, We need the bigger elements to reach the target. But the row is in decreasing order and so it contains only smaller elements. So, we will eliminate the row by increasing the current row value by 1(*i.e. row++*) and thus we will move column-wise.

Algorithm:

1. As we are starting from the cell (0, m-1), the two variables i.e. ‘row’ and ‘col’ will point to 0 and m-1 respectively.
2. We will do the following steps until row < n and col >= 0(*i.e. while(row < n && col >= 0)*):
   1. **If matrix[row][col] == target:** We have found the target and so we will return true.
   2. **If matrix[row][col] > target:** We need the smaller elements to reach the target. But the column is in increasing order and so it contains only greater elements. So, we will eliminate the column by decreasing the current column value by 1(*i.e. col--*) and thus we will move row-wise.

3. **If matrix[row][col] < target:** In this case, We need the bigger elements to reach the target. But the row is in decreasing order and so it contains only smaller elements. So, we will eliminate the row by increasing the current row value by 1(*i.e. row++*) and thus we will move column-wise.

3. If we are outside the loop without getting any matching element, we will return false.

Code

```
import java.util.*;

public class tUf {
    public static boolean searchElement(ArrayList<ArrayList<Integer>> matrix, int target) {
        int n = matrix.size();
        int m = matrix.get(0).size();
        int row = 0, col = m - 1;

        //traverse the matrix from (0, m-1):
        while (row < n && col >= 0) {
            if (matrix.get(row).get(col) == target) return true;
            else if (matrix.get(row).get(col) < target) row++;
            else col--;
        }
        return false;
    }

    public static void main(String[] args) {
        ArrayList<ArrayList<Integer>> matrix = new ArrayList<>();
        matrix.add(new ArrayList<>(Arrays.asList(1, 4, 7, 11, 15)));
        matrix.add(new ArrayList<>(Arrays.asList(2, 5, 8, 12, 19)));
        matrix.add(new ArrayList<>(Arrays.asList(3, 6, 9, 16, 22)));
        matrix.add(new ArrayList<>(Arrays.asList(10, 13, 14, 17, 24)));
        matrix.add(new ArrayList<>(Arrays.asList(18, 21, 23, 26, 30)));

        boolean result = searchElement(matrix, 8);
        System.out.println(result ? "true" : "false");
    }
}
```

Output: true

Complexity Analysis

Time Complexity: O(N+M), where N = given row number, M = given column number.
Reason: We are starting traversal from (0, M-1), and at most, we can end up being in the cell (M-1, 0). So, the total distance can be at most (N+M). So, the time complexity is O(N+M).

Space Complexity: O(1) as we are not using any extra space.