Search in a sorted 2D 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 are sorted in non-decreasing order. Moreover, the first element of a row is 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 = 3, M = 4, target = 8,
mat[] =
1 2 3 4
5 6 7 8
9 10 11 12
Result: true
Explanation: The 'target'  = 8 exists in the 'mat' at index (1, 3).

Example 2:
Input Format: N = 3, M = 3, target = 78,
mat[] =
1 2 4
6 7 8
9 10 34
Result: false
Explanation: The 'target' = 78 does not exist in the 'mat'. Therefore in the output, we see 'false'.
```

Brute Force Approach
Algorithm / Intuition

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 find any matching element, we will return true.
4. Otherwise, after completing the traversal, we will return false.

Code

```
#include <bits/stdc++.h>
using namespace std;

bool searchMatrix(vector<vector<int>>& matrix, int target) {
    int n = matrix.size(), m = matrix[0].size();

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

int main()
{
    vector<vector<int>> matrix = {{1, 2, 3, 4}, {5, 6, 7, 8}, {9, 10, 11, 12}};
    searchMatrix(matrix, 8) == true ? cout << "true\n" : cout << "false\n";
}
```

**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.

To make the time complexity even better, we won't use binary search on every row. We'll focus only on the particular row where the target might be located.

**How to check if a specific row is containing the target:**

If the target lies between the first and last element of the row, i (*i.e. matrix[i][0] <= target && target <= matrix[i][m-1]*), we can conclude that the target might be present in that specific row.

Once we locate the potentially relevant row containing the 'target', we need to confirm its presence. To accomplish this, we will utilize the Binary search algorithm, effectively reducing the time complexity.

**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.
    1. **If matrix[i][0] <= target && target <= matrix[i][m-1]:** If this condition is met, we can conclude that row i has the possibility of containing the target.
    So, we will apply binary search on row i, and check if the 'target' is present. If it is present, we will return true from this step. Otherwise, we will return false.
3. Otherwise, after completing the traversal, we will return false.

Code

```
#include <bits/stdc++.h>
using namespace std;

bool binarySearch(vector<int>& 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[mid] == target) return true;
        else if (target > nums[mid]) low = mid + 1;
        else high = mid - 1;
    }
    return false;
}


bool searchMatrix(vector<vector<int>>& matrix, int target) {
    int n = matrix.size();
    int m = matrix[0].size();

    for (int i = 0; i < n; i++) {
        if (matrix[i][0] <= target && target <= matrix[i][m - 1]) {
            return binarySearch(matrix[i], target);
        }
    }
    return false;
}

int main()
{
    vector<vector<int>> matrix = {{1, 2, 3, 4}, {5, 6, 7, 8}, {9, 10, 11, 12}};
    searchMatrix(matrix, 8) == true ? cout << "true\n" : cout << "false\n";
}
```

**Output:** true

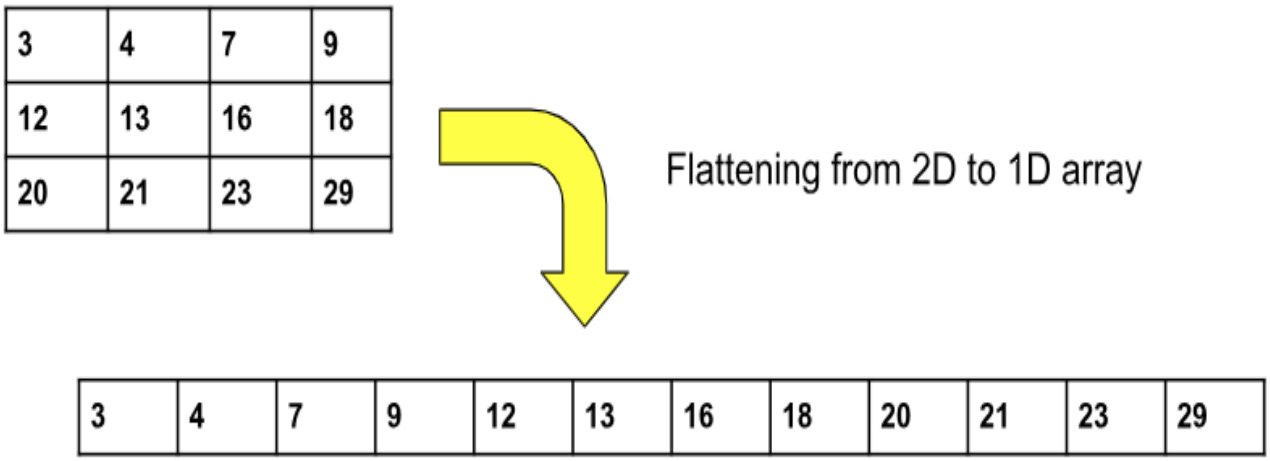
Complexity Analysis

**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. But for all rows, we are not applying binary search rather we are only applying it once for a particular row. That is why the time complexity is O(N + logM) instead of O(N*logM).

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

Optimal Approach
Algorithm / Intuition

If we flatten the given 2D matrix to a 1D array, the 1D array will also be sorted. By utilizing binary search on this sorted 1D array to locate the 'target' element, we can further decrease the time complexity. The flattening will be like the following:




But if we really try to flatten the 2D matrix, it will take O(N x M) time complexity and extra space to store the 1D array. In that case, it will not be the optimal solution anymore.

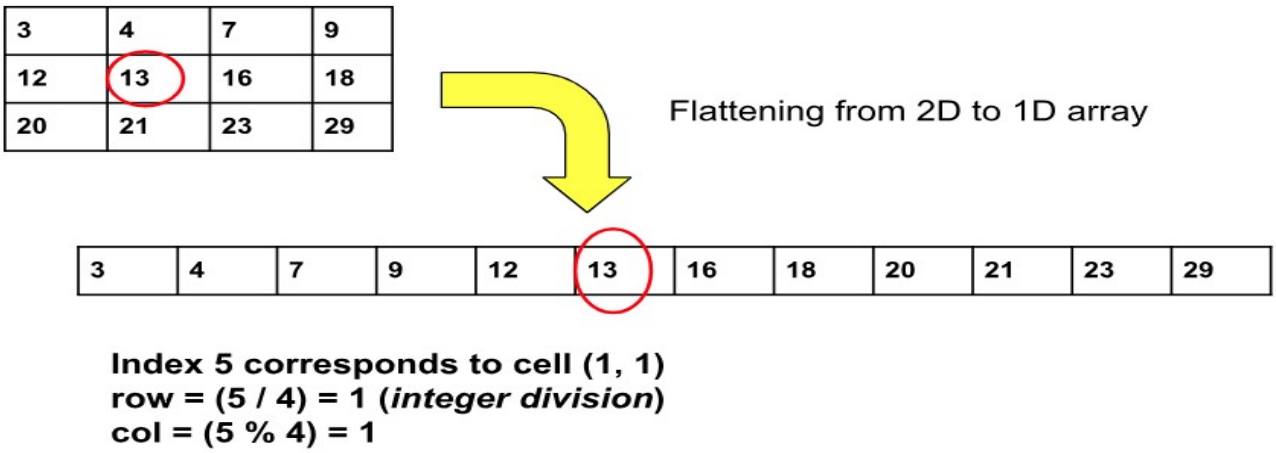
**How to apply binary search on the 1D array without actually flattening the 2D matrix:**

If we can figure out how to convert the index of the 1D array into the corresponding cell number in the 2D matrix, our task will be complete. In this scenario, we will use the binary search with the indices of the imaginary 1D array, ranging from 0 to (NxM)-1(*total no. of elements in the 1D array = NxM*). When comparing elements, we will convert the index to the cell number and retrieve the element. Thus we can apply binary search in the imaginary 1D array.

**How to convert 1D array index to the corresponding cell of the 2D matrix:**

We will use the following formula:

If index = i, and no. of columns in the matrix = m, the index i corresponds to the cell with
**row = i / m** and **col = i % m**. More formally, the cell is (i / m, i % m)(*0-based indexing*).




The range of the indices of the imaginary 1D array is [0, (NxM)-1] and in this range, we will apply binary search.

**Algorithm:**

1. **Place the 2 pointers i.e. low and high:** Initially, we will place the pointers. The pointer low will point to 0 and the high will point to (NxM)-1.

2. **Calculate the ‘mid’:** Now, inside the loop, we will calculate the value of ‘mid’ using the following formula:
   **mid = (low+high) // 2 ( ‘//’ refers to integer division)**
3. **Eliminate the halves based on the element at index mid:** To get the element, we will convert index ‘mid’ to the corresponding cell using the above formula. Here no. of columns of the matrix = M.
   **row = mid / M, col = mid % M.**
   1. **If matrix[row][col] == target:** We should return true here, as we have found the ‘target’.
   2. **If matrix[row][col] < target:** In this case, we need bigger elements. So, we will eliminate the left half and consider the right half (low = mid+1).
   3. **If matrix[row][col] > target:** In this case, we need smaller elements. So, we will eliminate the right half and consider the left half (high = mid-1).
4. Steps 2-3 will be inside a while loop and the loop will end once low crosses high
   (*i.e. low > high*). If we are out of the loop, we can say the target does not exist in the matrix. So, we will return false.

Code

```
#include <bits/stdc++.h>
using namespace std;

bool searchMatrix(vector<vector<int>>& matrix, int target) {
    int n = matrix.size();
    int m = matrix[0].size();

    //apply binary search:
    int low = 0, high = n * m - 1;
    while (low <= high) {
        int mid = (low + high) / 2;
        int row = mid / m, col = mid % m;
        if (matrix[row][col] == target) return true;
        else if (matrix[row][col] < target) low = mid + 1;
        else high = mid - 1;
    }
    return false;
}

int main()
{
    vector<vector<int>> matrix = {{1, 2, 3, 4}, {5, 6, 7, 8}, {9, 10, 11, 12}};
    searchMatrix(matrix, 8) == true ? cout << "true\n" : cout << "false\n";
}
```

**Output:** true

Complexity Analysis

**Time Complexity:** O(log(NxM)), where N = given row number, M = given column number.
**Reason:** We are applying binary search on the imaginary 1D array of size NxM.

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