# Question

Write an efficient algorithm that searches for a target value in an m x n integer matrix. The 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.

**Example 1:**



**Input:** 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]], target = 5

**Output:** true

**Example 2:**



**Input:** 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]], target = 20

**Output:** false

**Constraints:**

* m == matrix.length
* n == matrix[i].length
* 1 <= n, m <= 300
* -109 <= matix[i][j] <= 109
* All the integers in each row are **sorted** in ascending order.
* All the integers in each column are **sorted** in ascending order.
* -109 <= target <= 109

# Solution

#### **Approach 1: Brute Force**

**Intuition**

As a baseline, we can search the 2D array the same way we might search an unsorted 1D array -- by examining each element.

**Algorithm**

The algorithm doesn't really do anything more clever than what is explained by the intuition; we loop over the array, checking each element in turn. If we find it, we return true. Otherwise, if we reach the end of the nested for loop without returning, we return false. The algorithm must return the correct answer in all cases because we exhaust the entire search space.

|  |
| --- |
| class Solution {  public boolean searchMatrix(int[][] matrix, int target) {  for (int i = 0; i < matrix.length; i++) {  for (int j = 0; j < matrix[0].length; j++) {  if (matrix[i][j] == target) {  return true;  }  }  }  return false;  }  } |

**Complexity Analysis**

* Time complexity : \mathcal{O}(nm)O(*nm*)

Becase we perform a constant time operation for each element of an n\times m*n*×*m* element matrix, the overall time complexity is equal to the size of the matrix.

* Space complexity : \mathcal{O}(1)O(1)

The brute force approach does not allocate more additional space than a handful of pointers, so the memory footprint is constant.

#### **Approach 2: Binary Search**

**Intuition**

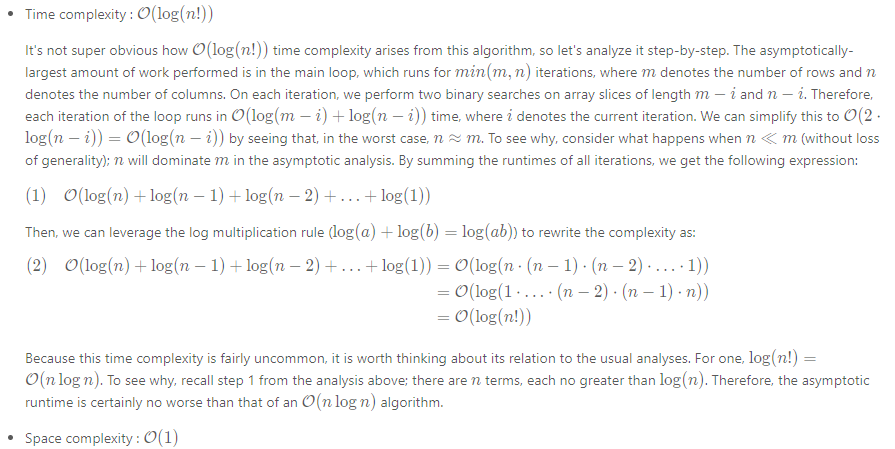
The fact that the matrix is sorted suggests that there must be some way to use binary search to speed up our algorithm.

**Algorithm**

First, we ensure that matrix is not null and not empty. Then, if we iterate over the matrix diagonals, we can maintain an invariant that the slice of the row and column beginning at the current (row, col)(*row*,*col*) pair is sorted. Therefore, we can always binary search these row and column slices for target. We proceed in a logical fashion, iterating over the diagonals, binary searching the rows and columns until we either run out of diagonals (meaning we can return False) or find target (meaning we can return True). The binarySearch function works just like normal binary search, but is made ugly by the need to search both rows and columns of a two-dimensional array.

|  |
| --- |
| class Solution {  private boolean binarySearch(int[][] matrix, int target, int start, boolean vertical) {  int lo = start;  int hi = vertical ? matrix[0].length-1 : matrix.length-1;  while (hi >= lo) {  int mid = (lo + hi)/2;  if (vertical) { // searching a column  if (matrix[start][mid] < target) {  lo = mid + 1;  } else if (matrix[start][mid] > target) {  hi = mid - 1;  } else {  return true;  }  } else { // searching a row  if (matrix[mid][start] < target) {  lo = mid + 1;  } else if (matrix[mid][start] > target) {  hi = mid - 1;  } else {  return true;  }  }  }  return false;  }  public boolean searchMatrix(int[][] matrix, int target) {  // an empty matrix obviously does not contain `target`  if (matrix == null || matrix.length == 0) {  return false;  }  // iterate over matrix diagonals  int shorterDim = Math.min(matrix.length, matrix[0].length);  for (int i = 0; i < shorterDim; i++) {  boolean verticalFound = binarySearch(matrix, target, i, true);  boolean horizontalFound = binarySearch(matrix, target, i, false);  if (verticalFound || horizontalFound) {  return true;  }  }    return false;  }  } |

**Complexity Analysis**



Because our binary search implementation does not literally slice out copies of rows and columns from matrix, we can avoid allocating greater-than-constant memory.

#### **Approach 3: Divide and Conquer**

**Intuition**

We can partition a sorted two-dimensional matrix into four sorted submatrices, two of which might contain target and two of which definitely do not.

**Algorithm**

Because this algorithm operates recursively, its correctness can be asserted via the correctness of its base and recursive cases.

Base Case

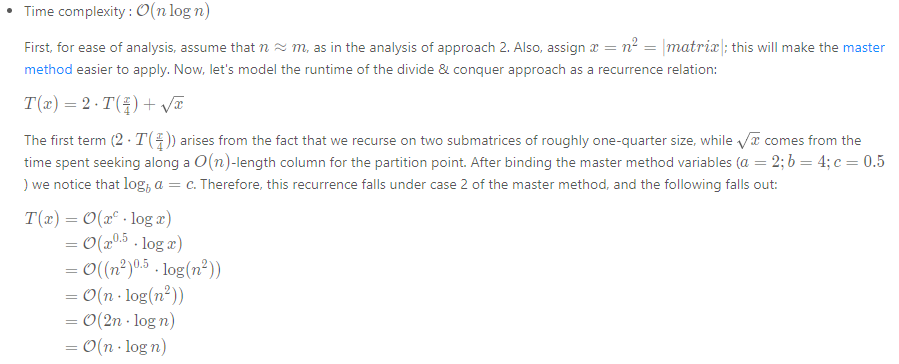
For a sorted two-dimensional array, there are two ways to determine in constant time whether an arbitrary element target can appear in it. First, if the array has zero area, it contains no elements and therefore cannot contain target. Second, if target is smaller than the array's smallest element (found in the top-left corner) or larger than the array's largest element (found in the bottom-right corner), then it definitely is not present.

Recursive Case

If the base case conditions have not been met, then the array has positive area and target could potentially be present. Therefore, we seek along the matrix's middle column for an index row such that *matrix*[*row*−1][*mid*]<*target*<*matrix*[*row*][*mid*]  (obviously, if we find target during this process, we immediately return true). The existing matrix can be partitioned into four submatrice around this index; the top-left and bottom-right submatrice cannot contain target (via the argument outlined in Base Case section), so we can prune them from the search space. Additionally, the bottom-left and top-right submatrice are sorted two-dimensional matrices, so we can recursively apply this algorithm to them.

|  |
| --- |
| class Solution {  private int[][] matrix;  private int target;  private boolean searchRec(int left, int up, int right, int down) {  // this submatrix has no height or no width.  if (left > right || up > down) {  return false;  // `target` is already larger than the largest element or smaller  // than the smallest element in this submatrix.  } else if (target < matrix[up][left] || target > matrix[down][right]) {  return false;  }  int mid = left + (right-left)/2;  // Locate `row` such that matrix[row-1][mid] < target < matrix[row][mid]  int row = up;  while (row <= down && matrix[row][mid] <= target) {  if (matrix[row][mid] == target) {  return true;  }  row++;  }  return searchRec(left, row, mid-1, down) || searchRec(mid+1, up, right, row-1);  }  public boolean searchMatrix(int[][] mat, int targ) {  // cache input values in object to avoid passing them unnecessarily  // to `searchRec`  matrix = mat;  target = targ;  // an empty matrix obviously does not contain `target`  if (matrix == null || matrix.length == 0) {  return false;  }  return searchRec(0, 0, matrix[0].length-1, matrix.length-1);  }  } |

**Complexity Analysis**



Extension: what would happen to the complexity if we binary searched for the partition point, rather than used a linear scan?

* Space complexity : \mathcal{O}(\log n)O(log*n*)

Although this approach does not fundamentally require greater-than-constant addition memory, its use of recursion means that it will use memory proportional to the height of its recursion tree. Because this approach discards half of matrix on each level of recursion (and makes two recursive calls), the height of the tree is bounded by \log nlog*n*.

#### **Approach 4: Search Space Reduction**

**Intuition**

Because the rows and columns of the matrix are sorted (from left-to-right and top-to-bottom, respectively), we can prune \mathcal{O}(m)O(*m*) or \mathcal{O}(n)O(*n*) elements when looking at any particular value.

**Algorithm**

First, we initialize a (row, col)(*row*,*col*) pointer to the bottom-left of the matrix.[[1]](https://leetcode.com/problems/search-a-2d-matrix-ii/solution/#fn1) Then, until we find target and return true (or the pointer points to a (row, col)(*row*,*col*) that lies outside of the dimensions of the matrix), we do the following: if the currently-pointed-to value is larger than target we can move one row "up". Otherwise, if the currently-pointed-to value is smaller than target, we can move one column "right". It is not too tricky to see why doing this will never prune the correct answer; because the rows are sorted from left-to-right, we know that every value to the right of the current value is larger. Therefore, if the current value is already larger than target, we know that every value to its right will also be too large. A very similar argument can be made for the columns, so this manner of search will always find target in the matrix (if it is present).

Check out some sample runs of the algorithm in the animation below:

1 / 35

|  |
| --- |
| **class Solution {**  **public boolean searchMatrix(int[][] matrix, int target) {**  **// start our "pointer" in the bottom-left**  **int row = matrix.length-1;**  **int col = 0;**  **while (row >= 0 && col < matrix[0].length) {**  **if (matrix[row][col] > target) {**  **row--;**  **} else if (matrix[row][col] < target) {**  **col++;**  **} else { // found it**  **return true;**  **}**  **}**  **return false;**  **}**  **}** |

**Complexity Analysis**

* Time complexity : \mathcal{O}(n+m)O(*n*+*m*)

The key to the time complexity analysis is noticing that, on every iteration (during which we do not return true) either row or col is is decremented/incremented exactly once. Because row can only be decremented m*m* times and col can only be incremented n*n* times before causing the while loop to terminate, the loop cannot run for more than n+m*n*+*m* iterations. Because all other work is constant, the overall time complexity is linear in the sum of the dimensions of the matrix.

* Space complexity : \mathcal{O}(1)O(1)

Because this approach only manipulates a few pointers, its memory footprint is constant.

#### **Footnotes**

1. This would work equally well with a pointer initialized to the top-right. Neither of the other two corners would work, as pruning a row/column might prevent us from achieving the correct answer. [↩︎](https://leetcode.com/problems/search-a-2d-matrix-ii/solution/#fnref1)