240. Search a 2D Matrix II

Dec. 15, 2017 | 86.5K views

Previous Next

Write an efficient algorithm that searches for a value in an *m* x *n* 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.

Example:

Consider the following 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]
]
```

Given target = `5`, return `true`.

Given target = `20`, return `false`.

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.

Java | Python3 | Copy

```python
class Solution:
    def searchMatrix(self, matrix, target):
        for row in matrix:
            if target in row:
                return True

        return False
```

Complexity Analysis

- Time complexity : $\mathcal{O}(nm)$

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

- Space complexity : $\mathcal{O}(1)$

  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) 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.

Java | Python3 | Copy

```python
class Solution:
    def binary_search(self, matrix, target, start, vertical):
        lo = start
        hi = len(matrix[0])-1 if vertical else len(matrix)-1

        while hi >= lo:
            mid = (lo + hi)//2
            if vertical: # searching a column
                if matrix[start][mid] < target:
                    lo = mid + 1
                elif matrix[start][mid] > target:
                    hi = mid - 1
                else:
                    return True
            else: # searching a row
                if matrix[mid][start] < target:
                    lo = mid + 1
                elif matrix[mid][start] > target:
                    hi = mid - 1
                else:
                    return True

        return False

    def searchMatrix(self, matrix, target):
        # an empty matrix obviously does not contain `target`
        if not matrix:
```

Complexity Analysis

- Time complexity : $\mathcal{O}(lg(n!))$

  It's not super obvious how $\mathcal{O}(lg(n!))$ time complexity arises from this algorithm, so let's analyze it step-by-step. The asymptotically-largest amount of work performed is in the main loop, which runs for $min(m, n)$ iterations, where m denotes the number of rows and n denotes the number of columns. On each iteration, we perform two binary searches on array slices of length $m - i$ and $n - i$. Therefore, each iteration of the loop runs in $\mathcal{O}(lg(m-i)+lg(n-i))$ time, where i denotes the current iteration. We can simplify this to $\mathcal{O}(2 \cdot lg(n-i)) = \mathcal{O}(lg(n-i))$ by seeing that, in the worst case, $n \approx m$. To see why, consider what happens when $n \ll m$ (without loss of generality); n will dominate m in the asymptotic analysis. By summing the runtimes of all iterations, we get the following expression:

  $$(1) \quad \mathcal{O}(lg(n) + lg(n-1) + lg(n-2) + \ldots + lg(1))$$

  Then, we can leverage the log multiplication rule ($lg(a) + lg(b) = lg(ab)$) to rewrite the complexity as:

  $$\begin{aligned} (2) \quad \mathcal{O}(lg(n) + lg(n-1) + lg(n-2) + \ldots + lg(1)) &= \mathcal{O}(lg(n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot 1)) \\ &= \mathcal{O}(lg(1 \cdot \ldots \cdot (n-2) \cdot (n-1) \cdot n)) \\ &= \mathcal{O}(lg(n!)) \end{aligned}$$

  Because this time complexity is fairly uncommon, it is worth thinking about its relation to the usual analyses. For one, $lg(n!) = \mathcal{O}(nlgn)$. To see why, recall step 1 from the analysis above; there are n terms, each no greater than $lg(n)$. Therefore, the asymptotic runtime is certainly no worse than that of an $\mathcal{O}(nlgn)$ algorithm.

- Space complexity : $\mathcal{O}(1)$

  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.

Java | Python3 | Copy

```python
class Solution:
    def searchMatrix(self, matrix, target):
        # an empty matrix obviously does not contain `target`
        if not matrix:
            return False

        def search_rec(left, up, right, down):
            # this submatrix has no height or no width.
            if left > right or up > down:
                return False
            # `target` is already larger than the largest element or smaller
            # than the smallest element in this submatrix.
            elif target < matrix[up][left] or target > matrix[down][right]:
                return False

            mid = left + (right-left)//2

            # Locate `row` such that matrix[row-1][mid] < target < matrix[row][mid]
            row = up
            while row <= down and matrix[row][mid] <= target:
                if matrix[row][mid] == target:
                    return True
                row += 1

            return search_rec(left, row, mid-1, down) or search_rec(mid+1, up, right, row-1)

        return search_rec(0, 0, len(matrix[0])-1, len(matrix)-1)
```

Complexity Analysis

- Time complexity : $\mathcal{O}(nlgn)$

  First, for ease of analysis, assume that $n \approx m$, as in the analysis of approach 2. Also, assign $x = n^2 = |matrix|$; this will make the master method easier to apply. Now, let's model the runtime of the divide & conquer approach as a recurrence relation:

  $$T(x) = 2 \cdot T(\frac{x}{4}) + \sqrt{x}$$

  The first term ($2 \cdot T(\frac{x}{4})$) arises from the fact that we recurse on two submatrices of roughly one-quarter size, while $\sqrt{x}$ comes from the time spent seeking along a $O(n)$-length column for the partition point. After binding the master method variables ($a = 2; b = 4; c = 0.5$) we notice that $\log_b a = c$. Therefore, this recurrence falls under case 2 of the master method, and the following falls out:

  $$\begin{aligned} T(x) &= \mathcal{O}(x^c \cdot lgx) \\ &= \mathcal{O}(x^{0.5} \cdot lgx) \\ &= \mathcal{O}((n^2)^{0.5} \cdot lg(n^2)) \\ &= \mathcal{O}(n \cdot lg(n^2)) \\ &= \mathcal{O}(2n \cdot lgn) \\ &= \mathcal{O}(n \cdot lgn) \end{aligned}$$

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

- Space complexity : $\mathcal{O}(lgn)$

  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 lgn.

Approach 4: Search Space Reduction

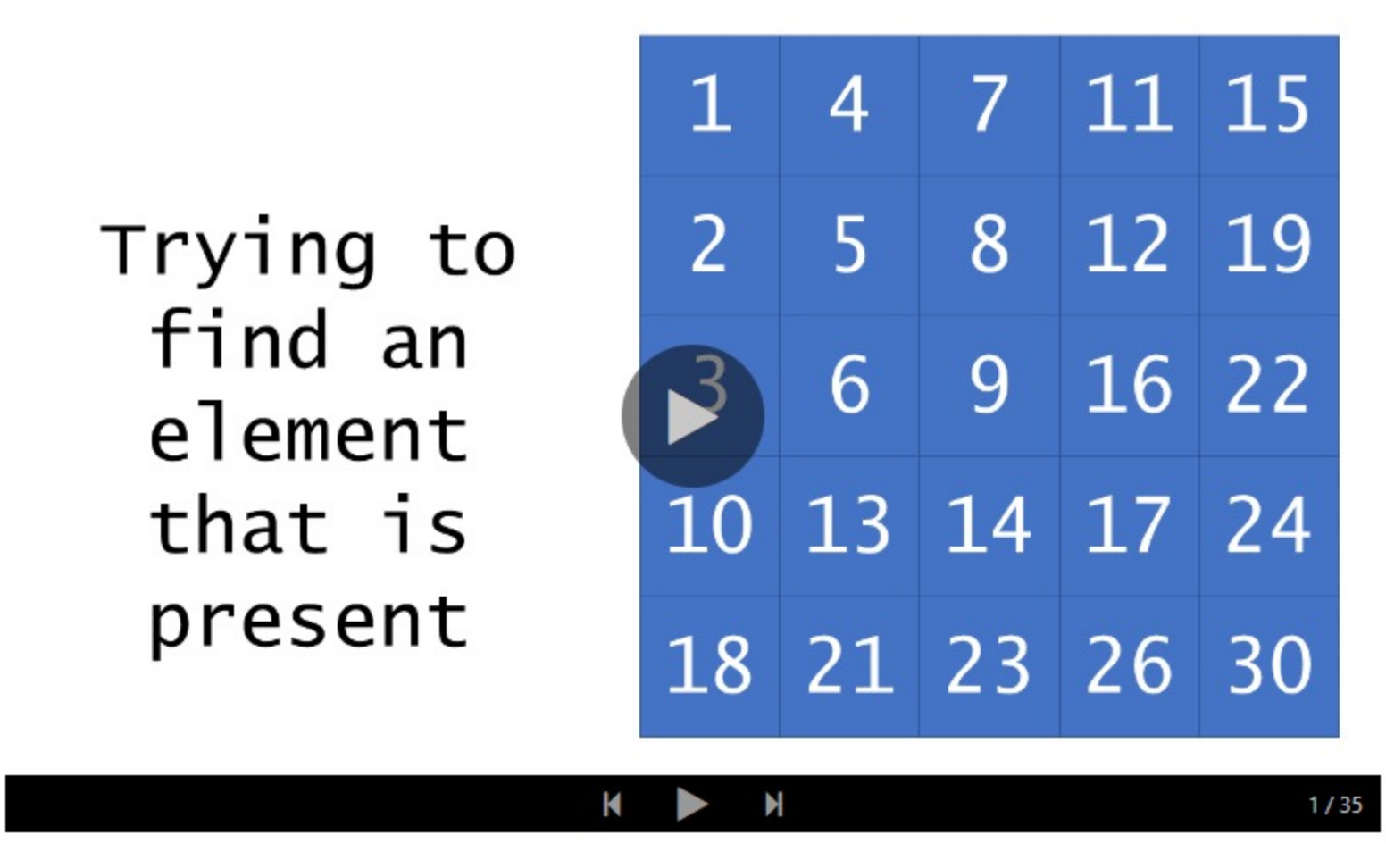
Intuition

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

Algorithm

First, we initialize a (row, col) pointer to the bottom-left of the matrix.[1] Then, until we find `target` and return `true` (or the pointer points to a (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:



Java | Python3 | Copy

```python
class Solution:
    def searchMatrix(self, matrix, target):
        # an empty matrix obviously does not contain `target` (make this check
        # because we want to cache `width` for efficiency's sake)
        if len(matrix) == 0 or len(matrix[0]) == 0:
            return False

        # cache these, as they won't change.
        height = len(matrix)
        width = len(matrix[0])

        # start our "pointer" in the bottom-left
        row = height-1
        col = 0

        while col < width and row >= 0:
            if matrix[row][col] > target:
                row -= 1
            elif matrix[row][col] < target:
                col += 1
            else: # found it
                return True

        return False
```

Complexity Analysis

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

  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 times and `col` can only be incremented n times before causing the `while` loop to terminate, the loop cannot run for more than $n+m$ iterations. Because all other work is constant, the overall time complexity is linear in the sum of the dimensions of the matrix.

- Space complexity : $\mathcal{O}(1)$

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

Analysis and solutions written by: @emptyset

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.

Previous Next