Day 44 / 100 :

Topic - matrix

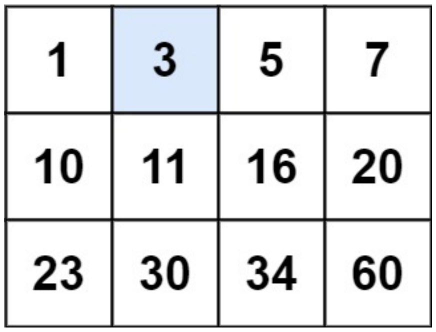
1️⃣ Problem statement: Search a 2D Matrix (medium)

You are given an m x n integer matrix with the following two properties:

- Each row is sorted in non-decreasing order.
- The first integer of each row is greater than the last integer of the previous row.

Given an integer target, return true if the target is in matrix or false otherwise.
You must write a solution in O(log(m * n)) time complexity.

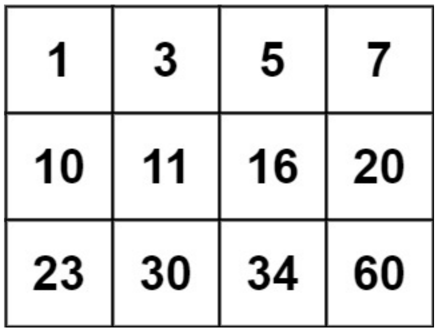
Example 1:



1	3	5	7
10	11	16	20
23	30	34	60

Input: matrix = [[1,3,5,7], [10,11,16,20], [23,30,34,60]], target = 3
Output: true

Example 2:



1	3	5	7
10	11	16	20
23	30	34	60

Input: matrix = [[1,3,5,7], [10,11,16,20], [23,30,34,60]], target = 13
Output: false

Solutions :

Approach 1 - Binary search

all the elements are sorted and we need to try and find a specific element, right? That is a no-brainer: we need to go with a binary search approach.

Provided neither the matrix or the rows are empty, since a nice problem like this needs to get burdened by pointless requirements that boil down to the chore of writing an extra line to remove silly edge cases <_<

Now, the relatively tricky part here is that we do not have all of them in a single flat container, but in a matrix. We will actually then want to do 2 binary searches: one to find the row to be searched, one to search it directly.

To do so, I declared 4 variables:

- row, meant to store which row will be searched, once we found one;
- l, r will store the extremes of our search, initialised to 0 and m.size() - 1, respectively;
- mid will store the on going average of l and r.

First round of binary search:

- we go on until l < r [note that the first one has to be a strict comparison, unlike for the second search];
- at each iteration we will compute mid as (l + r) / 2;
- we will then compare and find that if t is bigger than the last element of the row we are checking (m[mid].back() < t), it is time to raise l up to mid + 1 and loop again;
- specularly, if t is smaller than the first element of the row we are checking (m[mid][0] > t), it is time to lower r down to mid - 1 and loop again;
- finally, if neithere condition is correct, we it means that m[mid][0] <= t && m[mid].back() >= t, which implies we are already checking the right row, so we just assign l = mid and break out of the loop.

Be as it is, once we are done iterating, we will store the value of the correct row to search in row, reset l and r and move on with a similar logic to parse this time on a more traditional way if the element is in the row or not: a match will have us return true.

If we exit the loop without any joy, we return false.

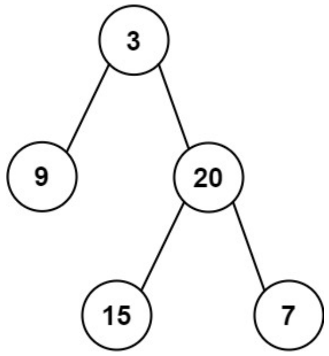
The code:

```
class Solution {
public:
    bool searchMatrix(vector<vector<int>> &m, int t) {
        // pointless edge case we still have to consider
             if (!m.size() || !m[0].size()) return false;
        // support variables
             int row, l = 0, r = m.size() - 1, mid;
        while (l < r) {
            mid = (l + r) / 2;
            // moving l up if needed
            if (m[mid].back() < t) l = mid + 1;
            // moving r down if needed
            else if (m[mid][0] > t) r = mid - 1;
            // we found our row!
            else {
                l = mid;
                break;
            };
        }
        // storing the value of the new found row
        row = l;
        // resetting l and r to run a binary search on the rows
        l = 0;
        r = m[0].size() - 1;
        while (l <= r) {
            mid = (l + r) / 2;
            // moving l up if needed
            if (m[row][mid] < t) l = mid + 1;
            // moving r down if needed
            else if (m[row][mid] > t) r = mid - 1;
            // we found our value!
            else return true;
        }
        return false;
    }
};
```

2 Problem statement: Construct Binary tree from in order and postorder traversal (medium)

Given two integer arrays inorder and postorder where inorder is the inorder traversal of a binary tree and postorder is the postorder traversal of the same tree, construct and return the binary tree.

Example 1:



Input: inorder = [9,3,15,20,7], postorder = [9,15,7,20,3]

Output: [3,9,20,null,null,15,7]

Example 2:

Input: inorder = [-1], postorder = [-1]

Output: [-1]

Intuition:

To construct a binary tree from inorder and postorder traversal arrays, we first need to understand what each of these traversals represents.
Inorder traversal visits the nodes in ascending order of their values, i.e., left child, parent, and right child. On the other hand, postorder traversal visits the nodes in the order left child, right child, and parent.
Knowing this, we can say that the last element in the postorder array is the root node, and its index in the inorder array divides the tree into left and right subtrees. We can recursively apply this logic to construct the entire binary tree.

Approach:

1. Start with the last element of the postorder array as the root node.
2. Find the index of the root node in the inorder array.
3. Divide the inorder array into left and right subtrees based on the index of the root node.
4. Divide the postorder array into left and right subtrees based on the number of elements in the left and right subtrees of the inorder array.
5. Recursively construct the left and right subtrees.

Complexity:

- Time complexity:
  The time complexity of this algorithm is O(n), where n is the number of nodes in the tree. We visit each node only once.
- Space complexity:
  The space complexity of this algorithm is O(n). We create a hashmap to store the indices of the inorder traversal, which takes O(n) space. Additionally, the recursive call stack can go up to O(n) in the worst case if the binary tree is skewed.

```
class Solution {
public:
    TreeNode* buildTree(vector<int>& inorder,
vector<int>& postorder) {
        unordered_map<int, int> index;
        for (int i = 0; i < inorder.size(); i++) {
            index[inorder[i]] = i;
        }
        return buildTreeHelper(inorder, postorder, 0,
inorder.size() - 1, 0, postorder.size() - 1, index);
    }

    TreeNode* buildTreeHelper(vector<int>& inorder,
vector<int>& postorder, int inorderStart, int
inorderEnd, int postorderStart, int postorderEnd,
unordered_map<int, int>& index) {
        if (inorderStart > inorderEnd || postorderStart
> postorderEnd) {
            return nullptr;
        }
        int rootVal = postorder[postorderEnd];
        TreeNode* root = new TreeNode(rootVal);
        int inorderRootIndex = index[rootVal];
        int leftSubtreeSize = inorderRootIndex -
inorderStart;
        root->left = buildTreeHelper(inorder,
postorder, inorderStart, inorderRootIndex - 1,
postorderStart, postorderStart + leftSubtreeSize - 1,
index);
        root->right = buildTreeHelper(inorder,
postorder, inorderRootIndex + 1, inorderEnd,
postorderStart + leftSubtreeSize, postorderEnd - 1,
index);
        return root;
    }
};
```