<https://leetcode.com/problems/kth-smallest-element-in-a-sorted-matrix/description/>

Given an n x n matrix where each of the rows and columns is sorted in ascending order, return *the* kth *smallest element in the matrix*.

Note that it is the kth smallest element **in the sorted order**, not the kth **distinct** element.

You must find a solution with a memory complexity better than O(n2).

**Example 1:**

Input: matrix = [[1,5,9],[10,11,13],[12,13,15]], k = 8

Output: 13

Explanation: The elements in the matrix are [1,5,9,10,11,12,13,13,15], and the 8th smallest number is 13

**Example 2:**

Input: matrix = [[-5]], k = 1

Output: -5

**Constraints:**

* n == matrix.length == matrix[i].length
* 1 <= n <= 300
* -109 <= matrix[i][j] <= 109
* All the rows and columns of matrix are **guaranteed** to be sorted in **non-decreasing order**.
* 1 <= k <= n2

**Follow up:**

* Could you solve the problem with a constant memory (i.e., O(1) memory complexity)?
* Could you solve the problem in O(n) time complexity? The solution may be too advanced for an interview but you may find reading [this paper](http://www.cse.yorku.ca/~andy/pubs/X+Y.pdf) fun.

**Attempt 1: 2023-11-06**

**Solution 1: Priority Queue (30 min)**

**Style  1: MinPQ**

class Solution {

public int kthSmallest(int[][] matrix, int k) {

int m = matrix.length;

int n = matrix[0].length;

PriorityQueue<Integer> minPQ = new PriorityQueue<>();

for(int i = 0; i < m; i++) {

for(int j = 0; j < n; j++) {

minPQ.offer(matrix[i][j]);

}

}

for(int i = 0; i < k - 1; i++) {

minPQ.poll();

}

return minPQ.peek();

}

}

**Style 2: MaxPQ**

class Solution {

public int kthSmallest(int[][] matrix, int k) {

int m = matrix.length;

int n = matrix[0].length;

PriorityQueue<Integer> maxPQ = new PriorityQueue<>((a, b) -> b - a);

for(int i = 0; i < m; i++) {

for(int j = 0; j < n; j++) {

if(maxPQ.size() < k) {

maxPQ.offer(matrix[i][j]);

} else if(maxPQ.peek() > matrix[i][j]) {

maxPQ.poll();

maxPQ.offer(matrix[i][j]);

}

}

}

return maxPQ.peek();

}

}

Time Complexity: O(M\*N\*logK), where M <= 300 is the number of rows, N <= 300 is the number of columns.

Space Complexity: O(K), space for heap which stores up to k elements.

**Solution 2: Binary Search (60 min)**

class Solution {

// Find lower boundary + auxiliary helper method

public int kthSmallest(int[][] matrix, int k) {

int m = matrix.length;

int n = matrix[0].length;

// The top left element will be the smallest

// the bottom right element will be the largest

int lo = matrix[0][0];

int hi = matrix[m - 1][n - 1];

while(lo <= hi) {

int mid = lo + (hi - lo) / 2;

int mid\_rank = findRank(mid, matrix, m, n);

if(mid\_rank >= k) {

hi = mid - 1;

} else {

lo = mid + 1;

}

}

return lo;

}

private int findRank(int mid, int[][] matrix, int rows, int cols) {

int count = 0;

int j = cols - 1;

for(int i = 0; i < rows; i++) {

// Alaways compare with the last element on each row

// since based on definition, on each row all elements

// are sorted in ascending order, the last element is

// the largest element on each row, and when we pre-define

// j = cols - 1, then if matrix[i][j] <= mid means all

// elements on that ith row <= mid, then count all elements

// on that row into mid rank, the except will happen only

// once, which is on one row its last element > mid, we

// cannot count all elements on that row into mid rank,

// only partial of them feasible

while(j >= 0 && matrix[i][j] > mid) {

j--;

}

count += j + 1;

}

return count;

}

}

Time Complexity: O((M+N) \* logD), where M <= 300 is the number of rows, N <= 300 is the number of columns, D <= 2\*10^9 is the difference between the maximum element and the minimum element in the matrix.

Space Complexity: O(1).

**Refer to**

<https://grandyang.com/leetcode/378/>

这道题让我们求有序矩阵中第K小的元素，**这道题的难点在于数组并不是蛇形有序的，意思是当前行的最后一个元素并不一定会小于下一行的首元素，所以我们并不能直接定位第K小的元素**，所以只能另辟蹊径。先来看一种利用堆的方法，我们使用一个最大堆，然后遍历数组每一个元素，将其加入堆，根据最大堆的性质，大的元素会排到最前面，然后我们看当前堆中的元素个数是否大于k，大于的话就将首元素去掉，循环结束后我们返回堆中的首元素即为所求:

class Solution {

public:

int kthSmallest(vector<vector<int>>& matrix, int k) {

priority\_queue<int> q;

for (int i = 0; i < matrix.size(); ++i) {

for (int j = 0; j < matrix[i].size(); ++j) {

q.emplace(matrix[i][j]);

if (q.size() > k) q.pop();

}

}

return q.top();

}

};

这题我们也可以用二分查找法来做，我们由于是有序矩阵，那么左上角的数字一定是最小的，而右下角的数字一定是最大的，所以这个是我们搜索的范围，然后我们算出中间数字mid，由于矩阵中不同行之间的元素并不是严格有序的，所以我们要在每一行都查找一下 mid，我们使用 upper\_bound，这个函数是查找第一个大于目标数的元素，如果目标数在比该行的尾元素大，则 upper\_bound 返回该行元素的个数，如果目标数比该行首元素小，则 upper\_bound 返回0, 我们遍历完所有的行可以找出中间数是第几小的数，然后k比较，进行二分查找，left 和 right 最终会相等，并且会变成数组中第k小的数字。举个例子来说吧，比如数组为:

[1 2

12 100]

k = 3

那么刚开始 left = 1, right = 100, mid = 50, 遍历完 cnt = 3，此时 right 更新为 50

此时 left = 1, right = 50, mid = 25, 遍历完之后 cnt = 3, 此时 right 更新为 25

此时 left = 1, right = 25, mid = 13, 遍历完之后 cnt = 3, 此时 right 更新为 13

此时 left = 1, right = 13, mid = 7, 遍历完之后 cnt = 2, 此时 left 更新为8

此时 left = 8, right = 13, mid = 10, 遍历完之后 cnt = 2, 此时 left 更新为 11

此时 left = 11, right = 12, mid = 11, 遍历完之后 cnt = 2, 此时 left 更新为 12

循环结束，left 和 right 均为 12，任意返回一个即可。

本解法的整体时间复杂度为 O(nlgn\*lgX)，其中X为最大值和最小值的差值，参见代码如下：

class Solution {

public:

int kthSmallest(vector<vector<int>>& matrix, int k) {

int left = matrix[0][0], right = matrix.back().back();

while (left < right) {

int mid = left + (right - left) / 2, cnt = 0;

for (int i = 0; i < matrix.size(); ++i) {

cnt += upper\_bound(matrix[i].begin(), matrix[i].end(), mid) - matrix[i].begin();

}

if (cnt < k) left = mid + 1;

else right = mid;

}

return left;

}

};

上面的解法还可以进一步优化到 O(nlgX)，其中X为最大值和最小值的差值，我们并不用对每一行都做二分搜索法，我们注意到每列也是有序的，我们可以利用这个性质，从数组的左下角开始查找，如果比目标值小，我们就向右移一位，而且我们知道当前列的当前位置的上面所有的数字都小于目标值，那么 cnt += i+1，反之则向上移一位，这样我们也能算出 cnt 的值。其余部分跟上面的方法相同，参见代码如下：

class Solution {

public:

int kthSmallest(vector<vector<int>>& matrix, int k) {

int left = matrix[0][0], right = matrix.back().back();

while (left < right) {

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

int cnt = search\_less\_equal(matrix, mid);

if (cnt < k) left = mid + 1;

else right = mid;

}

return left;

}

int search\_less\_equal(vector<vector<int>>& matrix, int target) {

int n = matrix.size(), i = n - 1, j = 0, res = 0;

while (i >= 0 && j < n) {

if (matrix[i][j] <= target) {

res += i + 1;

++j;

} else {

--i;

}

}

return res;

}

};

**Refer to**

<https://leetcode.com/problems/kth-smallest-element-in-a-sorted-matrix/solutions/1322101/c-java-python-maxheap-minheap-binary-search-picture-explain-clean-concise/>

**Solution 1: Max Heap keeps up to k elements**

* The easy approach is that we iterate all elements in the matrix and and add elements into the maxHeap. The maxHeap will keep up to k smallest elements (because when maxHeap is over size of k, we do remove the top of maxHeap which is the largest one). Finally, the top of the maxHeap is the **kth smallest element** in the matrix.
* This approach leads this problem become the same with [215. Kth Largest Element in an Array](https://leetcode.com/problems/kth-largest-element-in-an-array/solutions/1349609/), which doesn't take the advantage that the matrix is already sorted by rows and by columns.

class Solution { // 14 ms, faster than 55.67%

public int kthSmallest(int[][] matrix, int k) {

int m = matrix.length, n = matrix[0].length; // For general, the matrix need not be a square

PriorityQueue<Integer> maxHeap = new PriorityQueue<>((o1, o2) -> Integer.compare(o2, o1));

for (int r = 0; r < m; ++r) {

for (int c = 0; c < n; ++c) {

maxHeap.offer(matrix[r][c]);

if (maxHeap.size() > k) maxHeap.poll();

}

}

return maxHeap.poll();

}

}

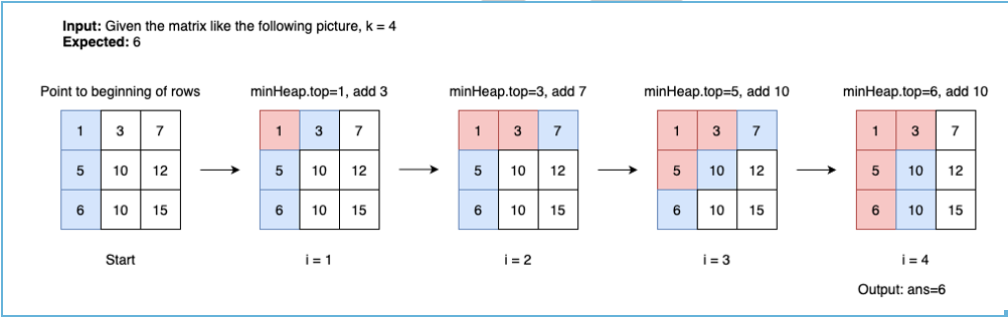
Complexity:

Time: O(M \* N \* logK), where M <= 300 is the number of rows, N <= 300 is the number of columns.

Space: O(K), space for heap which stores up to k elements.

**✔️ Solution 2: Min Heap to find kth smallest element from amongst N sorted list**

* Since each of the rows in matrix are already sorted, we can understand the problem as finding the **kth smallest element** from amongst M sorted rows.
* We start the pointers to point to the beginning of each rows, then we iterate k times, for each time ith, the top of the minHeap is the ith smallest element in the matrix. We pop the top from the minHeap then add the next element which has the same row with that top to the minHeap.



class Solution { // 18 ms, faster than 32.44%

public int kthSmallest(int[][] matrix, int k) {

int m = matrix.length, n = matrix[0].length, ans = -1; // For general, the matrix need not be a square

PriorityQueue<int[]> minHeap = new PriorityQueue<>(Comparator.comparingInt(o -> o[0]));

for (int r = 0; r < Math.min(m, k); ++r)

minHeap.offer(new int[]{matrix[r][0], r, 0});

for (int i = 1; i <= k; ++i) {

int[] top = minHeap.poll();

int r = top[1], c = top[2];

ans = top[0];

if (c + 1 < n) minHeap.offer(new int[]{matrix[r][c + 1], r, c + 1});

}

return ans;

}

}

Complexity:

Time: O(K \* logK)

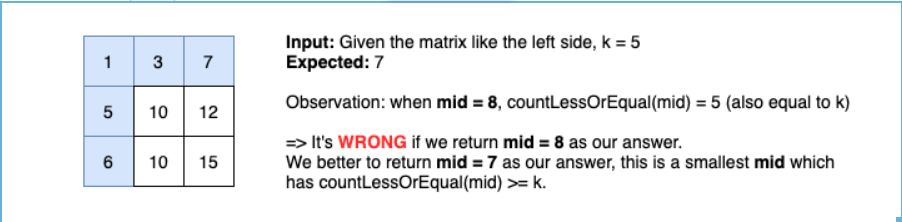
Space: O(K)

**✔️ Solution 3: Binary Search**

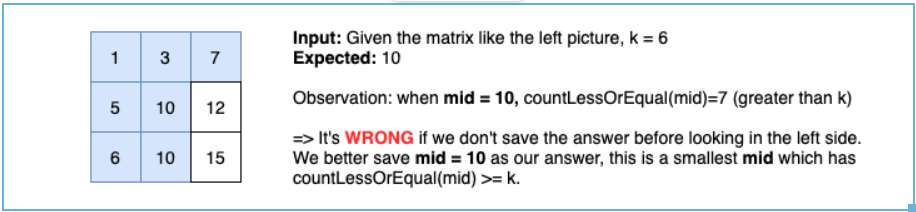
**Idea**

We binary search to find **the smallest ans** in range [minOfMatrix..maxOfMatrix] such that **countLessOrEqual(ans) >= k**, where countLessOrEqual(x) is the number of elements less than or equal to x.

Why ans must be as smallest as possible?



Why countLessOrEqual(ans) >= k but not countLessOrEqual(ans) == k?



**Algorithm**

Start with left = minOfMatrix = matrix[0][0] and right = maxOfMatrix = matrix[n-1][n-1].

Find the mid of the left and the right. This middle number is **NOT necessarily** an element in the matrix.

If countLessOrEqual(mid) >= k, we keep current ans = mid and try to find smaller value by searching in the left side. Otherwise, we search in the right side.

Since ans is the smallest value which countLessOrEqual(ans) >= k, so it's the **k th smallest element in the matrix**.

**How to count number of elements less or equal to x efficiently?**

Since our matrix is sorted in ascending order by rows and columns.

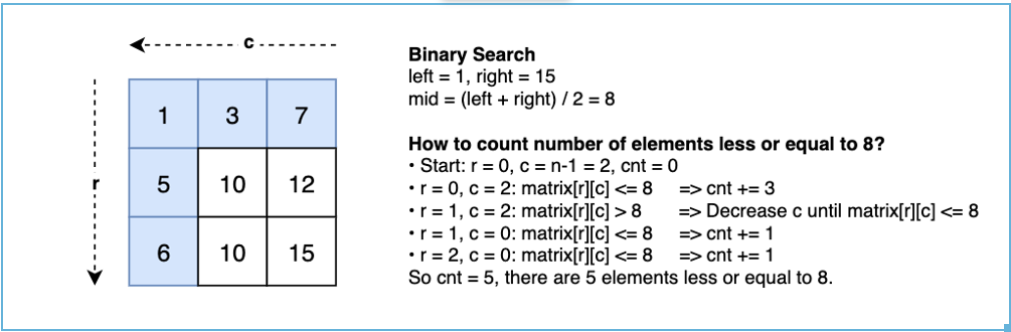
We use two pointers, one points to the rightmost column c = n-1, and one points to the lowest row r = 0.

If matrix[r][c] <= x then the number of elements in row r less or equal to x is (c+1) (Because row[r] is sorted in ascending order, so if matrix[r][c] <= x then matrix[r][c-1] is also <= x). Then we go to next row to continue counting.

Else if matrix[r][c] > x, we decrease column c until matrix[r][c] <= x (Because column is sorted in ascending order, so if matrix[r][c] > x then matrix[r+1][c] is also > x).

Time complexity for counting: O(M+N).

It's exactly the same idea with this problem: [240. Search a 2D Matrix II](https://leetcode.com/problems/search-a-2d-matrix-ii/)



**Implementation**

class Solution { // 0 ms, faster than 100%

int m, n;

public int kthSmallest(int[][] matrix, int k) {

m = matrix.length; n = matrix[0].length; // For general, the matrix need not be a square

int left = matrix[0][0], right = matrix[m-1][n-1], ans = -1;

while (left <= right) {

int mid = (left + right) >> 1;

if (countLessOrEqual(matrix, mid) >= k) {

ans = mid;

right = mid - 1; // try to looking for a smaller value in the left side

} else left = mid + 1; // try to looking for a bigger value in the right side

}

return ans;

}

int countLessOrEqual(int[][] matrix, int x) {

int cnt = 0, c = n - 1; // start with the rightmost column

for (int r = 0; r < m; ++r) {

while (c >= 0 && matrix[r][c] > x) --c; // decrease column until matrix[r][c] <= x

cnt += (c + 1);

}

return cnt;

}

}

**Complexity**

Time: O((M+N) \* logD), where M <= 300 is the number of rows, N <= 300 is the number of columns, D <= 2\*10^9 is the difference between the maximum element and the minimum element in the matrix.

Space: O(1).