Kth Smallest Element in a Sorted Matrix

Given a *n* x *n* matrix where each of the rows and columns are sorted in ascending order, find the kth smallest element in the matrix.

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

Example:

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

return 13.
```

Note:
You may assume k is always valid, $1 \le k \le n^2$.

Solution 1

Here is the step of my solution:

1. Build a minHeap of elements from the first row.
2. Do the following operations k-1 times :
   Every time when you poll out the root(Top Element in Heap), you need to know the row number and column number of that element(so we can create a tuple class here), replace that root with the next element from the same column.

After you finish this problem, thinks more :

1. For this question, you can also build a min Heap from the first column, and do the similar operations as above.(Replace the root with the next element from the same row)
2. What is more, this problem is exact the same with Leetcode373 Find K Pairs with Smallest Sums, I use the same code which beats 96.42%, after you solve this problem, you can check with this link: https://discuss.leetcode.com/topic/52953/share-my-solution-which-beat-96-42

```
public class Solution {
    public int kthSmallest(int[][] matrix, int k) {
        int n = matrix.length;
        PriorityQueue<Tuple> pq = new PriorityQueue<Tuple>();
        for(int j = 0; j <= n-1; j++) pq.offer(new Tuple(0, j, matrix[0][j]));
        for(int i = 0; i < k-1; i++) {
            Tuple t = pq.poll();
            if(t.x == n-1) continue;
            pq.offer(new Tuple(t.x+1, t.y, matrix[t.x+1][t.y]));
        }
        return pq.poll().val;
    }
}

class Tuple implements Comparable<Tuple> {
    int x, y, val;
    public Tuple (int x, int y, int val) {
        this.x = x;
        this.y = y;
        this.val = val;
    }

    @Override
    public int compareTo (Tuple that) {
        return this.val - that.val;
    }
}
```

written by YUANGAO1023 original link here

Solution 2

```
class Solution
{
public:
 int kthSmallest(vector<vector<int>>& matrix, int k)
 {
  int n = matrix.size();
  int le = matrix[0][0], ri = matrix[n - 1][n - 1];
  int mid = 0;
  while (le < ri)
  {
   mid = (le + ri) >> 1;
   int num = 0;
   for (int i = 0; i < n; i++)
   {
    int pos = upper_bound(matrix[i].begin(), matrix[i].end(), mid) - matrix[i].be
gin();
    num += pos;
   }
   if (num < k)
   {
    le = mid + 1;
   }
   else
   {
    ri = mid;
   }
  }
  return le;
 }
};
```

written by 2997ms original link here

Solution 3

It's O(n) where n is the number of rows (and columns), not the number of elements. So it's very efficient. The algorithm is from the paper Selection in X + Y and matrices with sorted rows and columns, which I first saw mentioned by @elmirap (thanks).

The basic idea: Consider the submatrix you get by removing every second row and every second column. This has about a quarter of the elements of the original matrix. And the k-th element (k-th *smallest* I mean) of the original matrix is roughly the (k/4)-th element of the submatrix. So roughly get the (k/4)-th element of the submatrix and then use that to find the k-th element of the original matrix in O(n) time. It's recursive, going down to smaller and smaller submatrices until a trivial 2×2 matrix. For more details I suggest checking out the paper, the first half is easy to read and explains things well. Or @zhiqing_xiao's solution+explanation.

Cool: It uses variants of saddleback search that you might know for example from the Search a 2D Matrix II problem. And it uses the median of medians algorithm for linear-time selection.

Optimization: If k is less than n, we only need to consider the top-left k×k matrix. Similar if k is almost n^2. So it's even O(min(n, k, n^2-k)), I just didn't mention that in the title because I wanted to keep it simple and because those few very small or very large k are unlikely, most of the time k will be "medium" (and average $n^2/2$).

Implementation: I implemented the submatrix by using an index list through which the actual matrix data gets accessed. If [0, 1, 2, ..., n-1] is the index list of the original matrix, then [0, 2, 4, ...] is the index list of the submatrix and [0, 4, 8, ...] is the index list of the subsubmatrix and so on. This also covers the above optimization by starting with [0, 1, 2, ..., **k**-1] when applicable.

Application: I believe it can be used to easily solve theFind K Pairs with Smallest Sums problem in time O(k) instead of O(k log n), which I think is the best posted so far. I might try that later if nobody beats me to it (if you do, let me know :-).
Update: I did that now.

```
class Solution(object):
    def kthSmallest(self, matrix, k):

        # The median-of-medians selection function.
        def pick(a, k):
            if k == 1:
                return min(a)
            groups = (a[i:i+5] for i in range(0, len(a), 5))
            medians = [sorted(group)[len(group) / 2] for group in groups]
            pivot = pick(medians, len(medians) / 2 + 1)
            smaller = [x for x in a if x < pivot]
            if k <= len(smaller):
                return pick(smaller, k)
            k -= len(smaller) + a.count(pivot)
            return pivot if k < 1 else pick([x for x in a if x > pivot], k)

        # Find the k1-th and k2th smallest entries in the submatrix.
        def biselect(index, k1, k2):
```

```
            # Provide the submatrix.
            n = len(index)
            def A(i, j):
                return matrix[index[i]][index[j]]

            # Base case.
            if n <= 2:
                nums = sorted(A(i, j) for i in range(n) for j in range(n))
                return nums[k1-1], nums[k2-1]

            # Solve the subproblem.
            index_ = index[::2] + index[n-1+n%2:]
            k1_ = (k1 + 2*n) / 4 + 1 if n % 2 else n + 1 + (k1 + 3) / 4
            k2_ = (k2 + 3) / 4
            a, b = biselect(index_, k1_, k2_)

            # Prepare ra_less, rb_more and L with saddleback search variants.
            ra_less = rb_more = 0
            L = []
            jb = n   # jb is the first where A(i, jb) is larger than b.
            ja = n   # ja is the first where A(i, ja) is larger than or equal to
a.
            for i in range(n):
                while jb and A(i, jb - 1) > b:
                    jb -= 1
                while ja and A(i, ja - 1) >= a:
                    ja -= 1
                ra_less += ja
                rb_more += n - jb
                L.extend(A(i, j) for j in range(jb, ja))

            # Compute and return x and y.
            x = a if ra_less <= k1 - 1 else \
                b if k1 + rb_more - n*n <= 0 else \
                pick(L, k1 + rb_more - n*n)
            y = a if ra_less <= k2 - 1 else \
                b if k2 + rb_more - n*n <= 0 else \
                pick(L, k2 + rb_more - n*n)
            return x, y

        # Set up and run the search.
        n = len(matrix)
        start = max(k - n*n + n-1, 0)
        k -= n*n - (n - start)**2
        return biselect(range(start, min(n, start+k)), k, k)[0]
```

written by StefanPochmann original link here