378. Kth Smallest Element in a Sorted Matrix

Previous Next

April 26, 2020 | 8.1K views

Average Rating: 4.05 (19 votes)

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, $1 \le k \le n^2$.

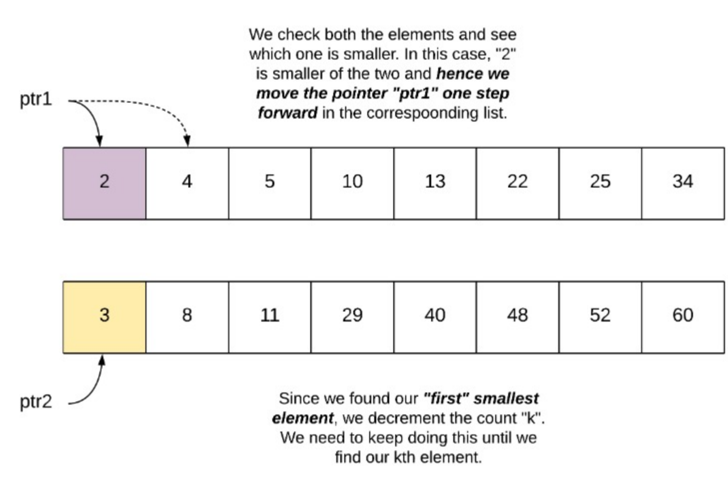
Solution

Approach 1: Min-Heap approach

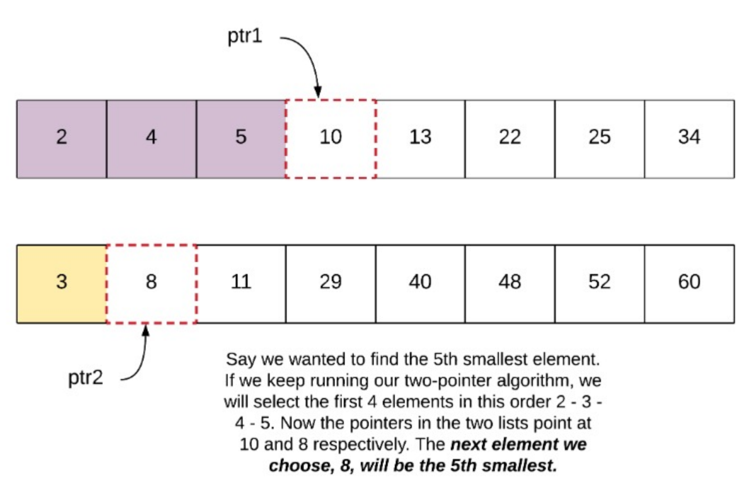
Intuition

The intuition for this approach is really simple. If you think about it, we can reframe the problem as finding the K^{th} smallest elements from amongst `N sorted lists` right? We know that the rows are sorted and so are the columns. So, we can treat each row (or column) as a sorted list in itself. Then, the problem just boils down to finding the K^{th} smallest element from amongst these `N` sorted lists. However, before we get to this problem, lets first talk about a simpler version of the problem which is to find the K^{th} smallest element from amongst 2 sorted lists. This is easy enough to solve since all we need are a pair of pointers which act as indices in the two lists.

At each step we check which element is smaller amongst the two being pointed at by the indices and progress the corresponding index accordingly. If you think about it, we just need to run the algorithm for merging two sorted lists without actually merging them. We need to keep on running this algorithm until we find our K^{th} element. Let's quickly look at how this would look like diagrammatically.



We can use this two-pointer approach for finding the K^{th} element. As explained above, at each step we check the two values and move the pointer corresponding to the smaller value forward.

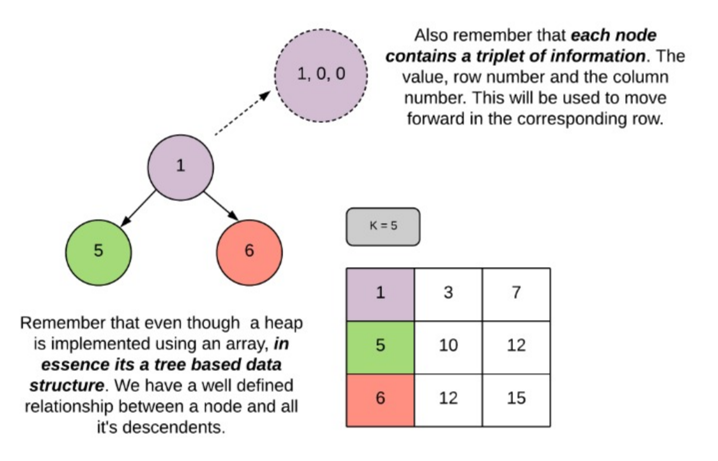


Now that was simple enough to do. In this particular problem, we have N sorted lists instead of just 2. That's what adds to the complexity. We can't really keep N different pointers now, can we? The heap data structure is perfect for this problem since at all times, we want to maintain `N` different `variables` with each of them pointing to an element in their corresponding lists. We want to be able to find the `minimum` amongst these N pointers quickly and then replace that element with the next one in its corresponding list.

The heap data structure gives us $O(1)$ access to the minimum element and $log(N)$ removal of the minimum element and addition of a new one. We just need to perform this operation `K` times to get our K^{th} smallest number. It's possible that our matrix has say 100 rows whereas we just need to find out the 5th smallest element. In this case, we can safely discard the rows in the lower part of the matrix i.e. in this case starting from the 6th row to the end of the matrix because the columns are sorted as well. So, we need the $min(N, K)$ rows essentially to find out the answer.

Algorithm

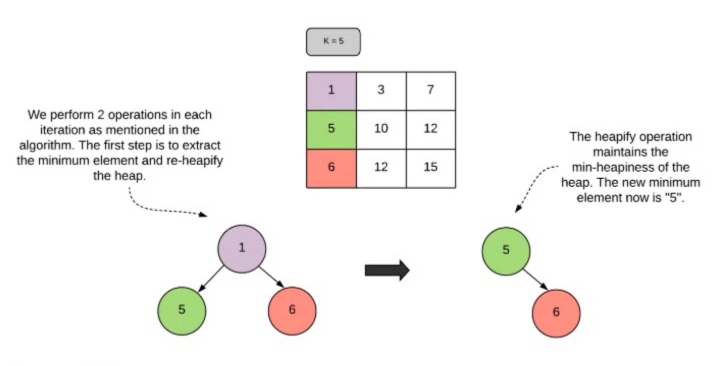
1. Initialize a min-heap called `H`.
2. For our implementation, we will be considering each row an individual list. Since the columns are sorted as well, we can easily treat each column as an individual list as well.
3. As mentioned before, we will take the first element of $min(N, K)$ rows where N represents the number of rows, and add each of these elements to the heap. It's important to know what row and column an element belongs to. Without knowing that, we won't be able to move forward in that particular list. So, apart from adding an element to the heap, we also need to add its row and column number. Hence, our min-heap will contain a triplet of information `(value, row, column)`. The heap will be arranged on the basis of the values and we will use the row and column number to add a replacement for the next element in case it gets popped off the heap.



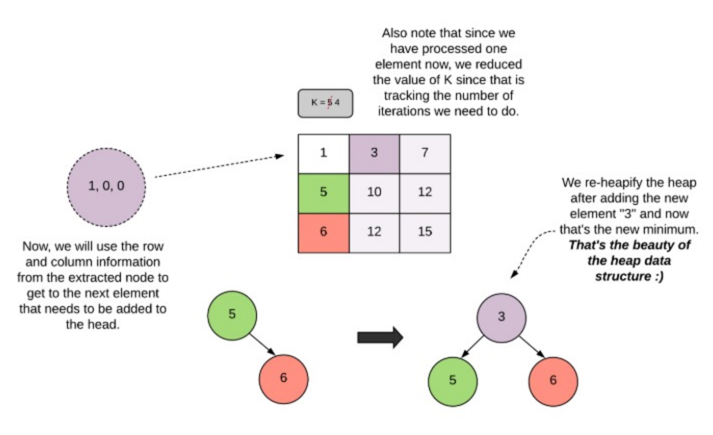
4. At this point, our heap contains $min(N, K)$ elements. Now we start a loop that goes until we iterate over `K` elements.

5. At each step, we remove the minimum element from the heap. The element will tell us which row should be further consumed. Using the row and column information we will add the next element to the heap. Specifically, if the current minimum element was from row `r` and had a column position `c`, then the next element to be added to the heap will be `(r, c+1)`.

`Extract-Min`



`Add a new element`



6. Keep on iterating till we exhaust `K` elements. The last element to be popped at the end of the loop will be the K^{th} smallest element.

Java Python3 Copy

```python
import heapq

class Solution:
    def kthSmallest(self, matrix: List[List[int]], k: int) -> int:

        # The size of the matrix
        N = len(matrix)

        # Preparing our min-heap
        minHeap = []
        for r in range(min(k, N)):

            # We add triplets of information for each cell
            minHeap.append((matrix[r][0], r, 0))

        # Heapify our list
        heapq.heapify(minHeap)

        # Until we find k elements
        while k:

            # Extract-Min
            element, r, c = heapq.heappop(minHeap)

            # If we have any new elements in the current row, add them
            if c < N - 1:
                heapq.heappush(minHeap, (matrix[r][c+1], r, c+1))
```

Complexity Analysis

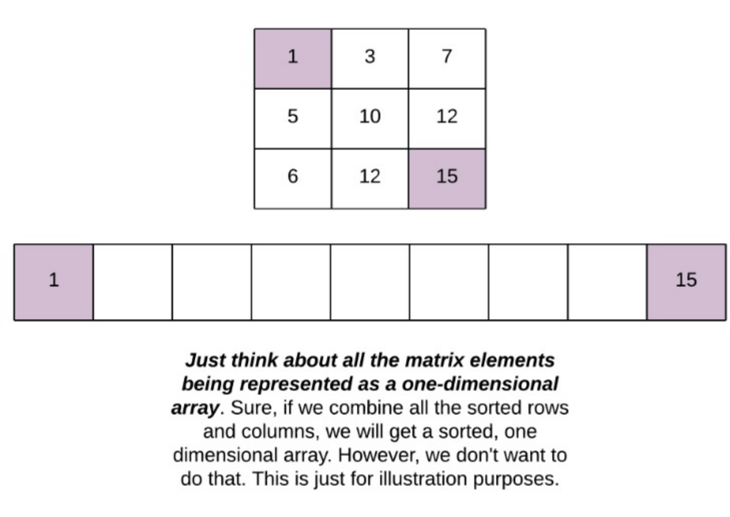
- Time Complexity: let $X = \min(K, N); X + K\log(X)$
  - Well the heap construction takes $O(X)$ time.
  - After that, we perform K iterations and each iteration has two operations. We extract the minimum element from a heap containing X elements. Then we add a new element to this heap. Both the operations will take $O(\log(X))$ time.
  - Thus, the total time complexity for this algorithm comes down to be $O(X + K\log(X))$ where X is $\min(K, N)$.
- Space Complexity: $O(X)$ which is occupied by the heap.

Approach 2: Binary Search

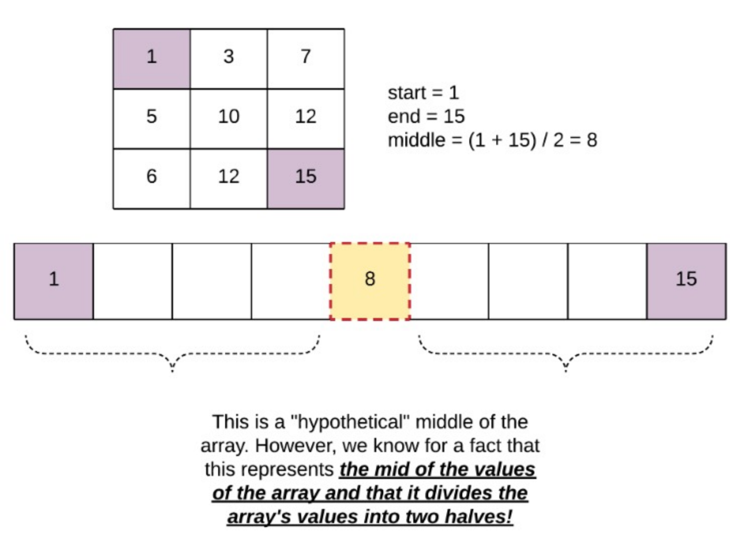
Inuition

Since each row and column of the matrix is sorted, is it possible to use Binary Search to find the K^{th} smallest number? The biggest problem to use Binary Search in this case is that we don't have a straightforward sorted array, instead we have a matrix. As we remember, in Binary Search, we calculate the middle index of the search space (`1` to `N`) and see if our required number is pointed out by the middle index; if not we either search in the lower half or the upper half. In a sorted matrix, we can't really find a middle. Even if we do consider some index as middle, it is not straightforward to find the search space containing numbers bigger or smaller than the number pointed out by the middle index.

An alternate could be to apply the Binary Search on the `number range` instead of the `index range`. As we know that the smallest number of our matrix is at the top left corner and the biggest number is at the bottom lower corner. These two number can represent the `range` i.e., the start and the end for the Binary Search. This does sound a bit counter-intuitive now, however, it will start to make sense soon. We are all accustomed to the linear array binary search algorithm. So, to delve into this idea a bit deeper, let's represent the `number range` on a single line as a one-dimensional array and see why and how binary search makes sense.



Now, let's proceed with the first step that we take in any binary search implementation. We find the middle element, right? In a normal, one-dimensional binary search, we use the indices to find the middle element. In this case, the left and the right ends of our sorted matrix are the two values. So, we use them to find the `hypothetical` middle of the matrix. The reason we call this hypothetical is because it is not necessary that the middle value will exist in the matrix. However, that is not a requirement for our algorithm.

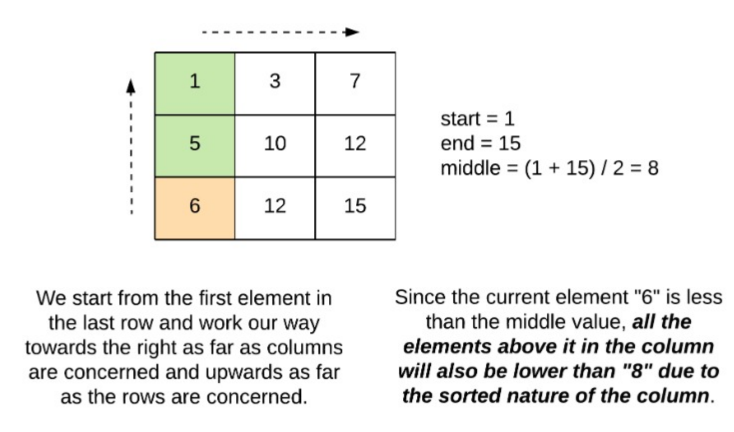


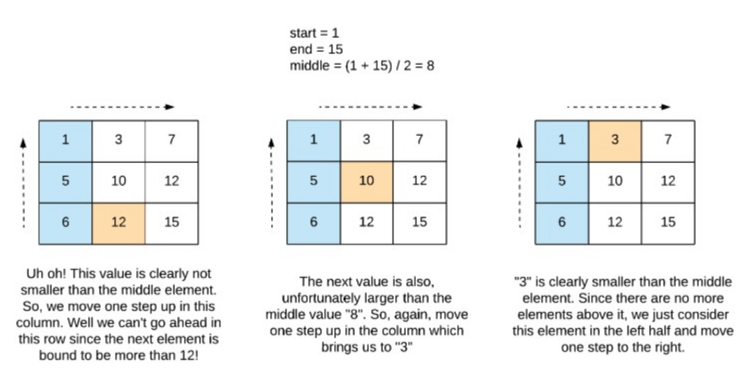
Even though this is a stupid question, it will help us to understand the overall logic here: How will you find out the K^{th} smallest number in a sorted one-dimensional array? It's simple, right? You'll just return the K^{th} element in the array. This is because you know the index K in the array contains your answer. In our example, we know the two extremes and the middle element value. However, what we don't know are the sizes of the two halves. For all we know, we could have 8 elements in the left half and just 2 on the right in the example above. We don't know!

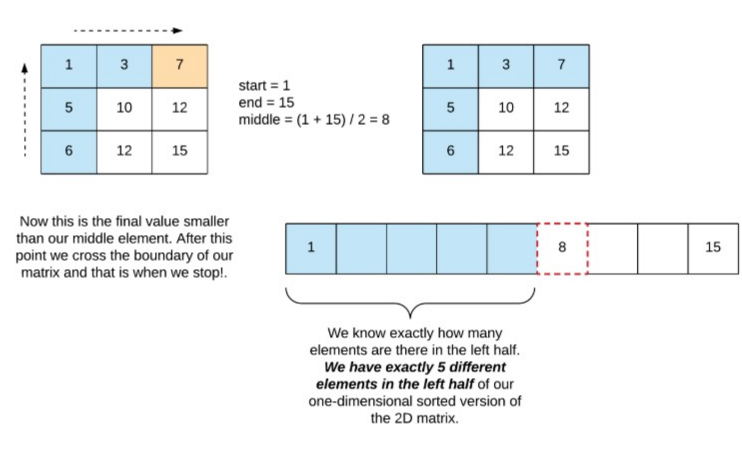
> So, after finding the middle element, we need to determine the size of the left half. Why, you might ask? Well because we want the Kth smallest element and not the largest. If the question asked us for the largest, we would be determining the size of the right half.

We already have a problem on LeetCode that is about searching for an element in a sorted 2D matrix. However, we don't want to search for our middle element. We want to `count the number of elements` in the left half of the number range of which we know the middle element and the two extremes as well. As it turns out, this can be done in $O(N + N) = O(N)$ time where N represents the number of rows and columns. We will make use of the sorted nature of the matrix and count all the elements we need without actually iterating over them.

> For example, if an element in the cell [i, j] is smaller than our middle element, then it means that all the elements in the column "j" before this element i.r. (i-1) other elements in this column are also going to be less than the middle element. Why? Because the columns are sorted as well!

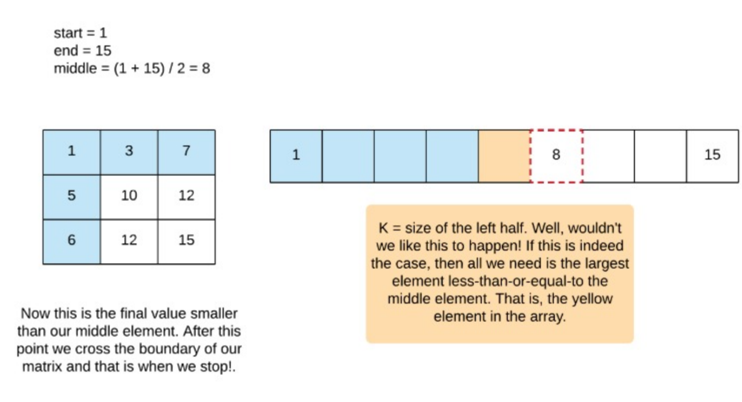




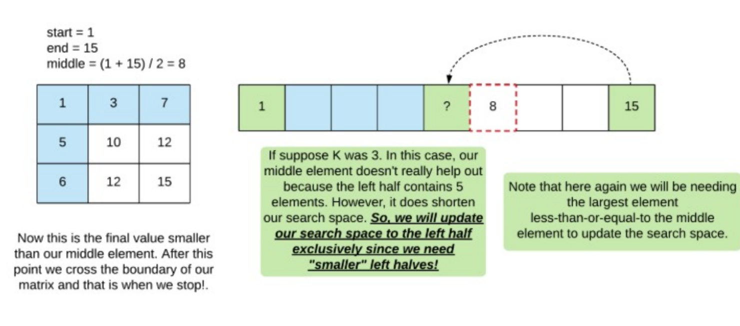


What do we do with this newfound information of ours? Well, that depends on the value of K. So, let's consider our 3 different cases here.

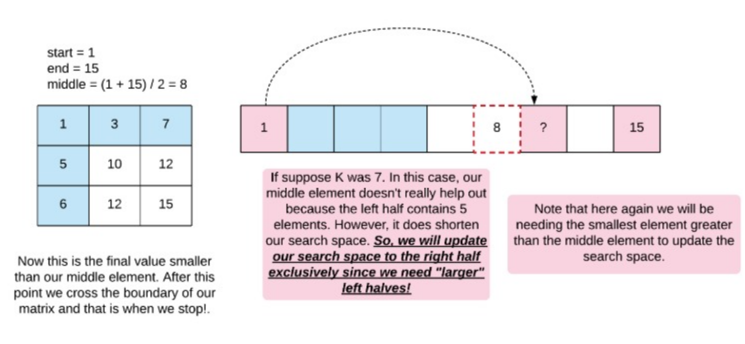
`Case 1: The size of the left half is EQUAL to K`



`Case 2: The size of the left half is LESS than K`



`Case 3: The size of the left half is MORE than K`



So, as we can see from the above figures, when we are trying to determine the size of our left-half, we also need to keep track of two values. The largest element less-than-or-equal-to the middle element and the smallest element greater than the middle. For an index-based binary search, this wouldn't be a problem since `middle+1` and `middle-1` would represent these entries :).

Algorithm

- Start the binary search with `start = matrix[0][0]` and `end = matrix[N-1][N-1]`
- Find the `middle` of the start and the end. This middle number is `NOT` necessarily an element in the matrix.
- Count all the numbers smaller than or equal to middle in the matrix. As the matrix is sorted, we can do this in $O(N)$. Note that this is determining the size of the left-half of the array.
- While counting, we need to keep track of the `smallest number greater than the middle` (let's call it `R`) and at the same time the `biggest number less than or equal to the middle` (let's call it `L`). These two numbers will be used to adjust the `number range` for the binary search in the next iteration.
- If the count is equal to `K`, `L` will be our required number as it is the `biggest number less than or equal to the middle`, and is definitely present in the matrix.
- If the count is less than `K`, we can update `start = R` to search in the higher part of the matrix
- If the count is greater than `K`, we can update `end = L` to search in the lower part of the matrix in the next iteration.

Java Python3 Copy

```python
class Solution:

    def countLessEqual(self, matrix, mid, smaller, larger):

        count, n = 0, len(matrix)
        row, col = n - 1, 0

        while row >= 0 and col < n:
            if matrix[row][col] > mid:

                # As matrix[row][col] is bigger than the mid, let's keep track of the
                # smallest number greater than the mid
                larger = min(larger, matrix[row][col])
                row -= 1

            else:

                # As matrix[row][col] is less than or equal to the mid, let's keep track of the
                # biggest number less than or equal to the mid

                smaller = max(smaller, matrix[row][col])
                count += row + 1
                col += 1

        return count, smaller, larger

    def kthSmallest(self, matrix: List[List[int]], k: int) -> int:
```

Complexity Analysis

- Time Complexity: $O(N \times log(\text{Max} - \text{Min}))$
  - Let's think about the time complexity in terms of the normal binary search algorithm. For a one-dimensional binary search over an array with N elements, the complexity comes out to be $O(log(N))$.
  - For our scenario, we are kind of defining our binary search space in terms of the minimum and the maximum numbers in the array. Going by this idea, the complexity for our binary search should be $O(log(\text{Max} - \text{Min}))$ where Max is the maximum element in the array and likewise, Min is the minimum element.
  - However, we update our search space after each iteration. So, even if the maximum element is super large as compared to the remaining elements in the matrix, we will bring down the search space considerably in the next iterations. But, going purely by the extremes for our search space, the complexity of our binary search in search of K^{th} smallest element will be $O(log(\text{Max} - \text{Min}))$.
  - In each iteration of our binary search approach, we iterate over the matrix trying to determine the size of the left-half as explained before. That takes $O(N)$.
  - Thus, the overall time complexity is $O(N \times log(\text{Max} - \text{Min}))$
- Space Complexity: $O(1)$ since we don't use any additional space for performing our binary search.

Although there are a series of posts about the Binary Search approach in our discussions section, the one I consulted personally was this. Most of the code and some portions of the explanation are taken from the post. I just tried to add a bit more visual flair to further explain things.

Also, Stefan Pochmann has another great approach based on a research paper for this very problem. Do check it out!

Rate this article:

Previous Next

Comments: 7 Sort By

Type comment here... (Markdown is supported)

Preview Post

NP-Complete ★2 May 28, 2020 12:41 PM

Could someone explain why the worst-case complexity for the binary search in Approach 2 is logarithmic if we are not always strictly partitioning the search space in half during each iteration?

2 Share Reply

hyl0327 ★9 May 14, 2020 1:18 AM

In Approach 1, we may use `heapreplace` when `c < N - 1`. It's supposed to be more efficient than `heappop` plus `heappush` (see https://docs.python.org/3/library/heapq.html#heapq.heapreplace).

It'll be something like:

Read More

1 Share Reply

Pavan161 ★9 June 14, 2020 9:34 AM

Simple and concise solution using maxHeap.

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

Read More

0 Share Reply

SHOW 2 REPLIES

yuli88 ★6 June 7, 2020 8:38 AM

For Approach 1, the following condition is sufficient to make it work, right?
(1) each of the rows are sorted;
(2) the first colum is sorted.

0 Share Reply

misrat ★24 May 22, 2020 3:52 AM

Will Approach 1 be accepted in an interview ?

0 Share Reply

SHOW 1 REPLY

NotADwarf ★14 May 17, 2020 2:23 PM

In addition to what @hyl0327 mentions we could also change the way the heap is created in python. The default heap constructor used in the solution above ignores the fact that the rows/columns are already sorted. There is an alternative way to construct the heap - by using *heapq.merge*. We can also use the *nsmallest* function to get the k smallest element. So the python code can be simplified to a one line solution:

Read More

0 Share Reply

SanketSingh ★10 May 2, 2020 2:04 AM

Found this great video on youtube explaining the Approach2
https://youtu.be/G5wLN4UweAM
I hope it will help you guys.

Read More

-1 Share Reply