# Kth Smallest Number (hard)

We'll cover the following



## Problem Statement

Given an unsorted array of numbers, find Kth smallest number in it.

Please note that it is the Kth smallest number in the sorted order, not the Kth distinct element.

**Example 1:**

```
Input: [1, 5, 12, 2, 11, 5], K = 3
Output: 5
Explanation: The 3rd smallest number is '5', as the first two smaller numbers are [1, 2].
```

**Example 2:**

```
Input: [1, 5, 12, 2, 11, 5], K = 4
Output: 5
Explanation: The 4th smallest number is '5', as the first three smaller numbers are
[1, 2, 5].
```

**Example 3:**

```
Input: [5, 12, 11, -1, 12], K = 3
Output: 11
Explanation: The 3rd smallest number is '11', as the first two small numbers are [5, -1].
```

## Try it yourself

Try solving this question here:

Java | Python3 | JS | C++

```java
import java.util.*;

class KthSmallestNumber {

  public static int findKthSmallestNumber(int[] nums, int k) {
    // TODO: Write your code here
    return -1;
  }

  public static void main(String[] args) {
    int result = KthSmallestNumber.findKthSmallestNumber(new int[] { 1, 5, 12, 2, 11, 5 }, 3);
    System.out.println("Kth smallest number is: " + result);

    // since there are two 5s in the input array, our 3rd and 4th smallest numbers should be a '5'
    result = KthSmallestNumber.findKthSmallestNumber(new int[] { 1, 5, 12, 2, 11, 5 }, 4);
```

```
    System.out.println("Kth smallest number is: " + result);

    result = KthSmallestNumber.findKthSmallestNumber(new int[] { 5, 12, 11, -1, 12 }, 3);
    System.out.println("Kth smallest number is: " + result);
  }
}
```

Run Save Reset

This is a well-known problem and there are multiple solutions available to solve this. A few other similar problems are:

1. Find the Kth largest number in an unsorted array.
2. Find the median of an unsorted array.
3. Find the 'K' smallest or largest numbers in an unsorted array.

Let's discuss different algorithms to solve this problem and understand their time and space complexity. Similar solutions can be devised for the above-mentioned three problems.

## 1. Brute-force

The simplest brute-force algorithm will be to find the Kth smallest number in a step by step fashion. This means that, first, we will find the smallest element, then 2nd smallest, then 3rd smallest and so on, until we have found the Kth smallest element. Here is what the algorithm will look like:

Java Python3 C++ JS

```
class KthSmallestNumber {

  public static int findKthSmallestNumber(int[] nums, int k) {
    int previousSmallestNum = Integer.MIN_VALUE;
    int previousSmallestIndex = -1;
    int currentSmallestNum = Integer.MAX_VALUE;
    int currentSmallestIndex = -1;
    for (int i = 0; i < k; i++) {
      for (int j = 0; j < nums.length; j++) {
        if (nums[j] > previousSmallestNum && nums[j] < currentSmallestNum) {
          // found the next smallest number
          currentSmallestNum = nums[j];
          currentSmallestIndex = j;
        } else if (nums[j] == previousSmallestNum && j > previousSmallestIndex) {
          // found a number which is equal to the previous smallest number; since numbers can repeat,
          // we will consider 'nums[j]' only if it has a different index than previous smallest
          currentSmallestNum = nums[j];
          currentSmallestIndex = j;
          break; // break here as we have found our definitive next smallest number
        }
      }
      // current smallest number becomes previous smallest number for the next iteration
      previousSmallestNum = currentSmallestNum;
      previousSmallestIndex = currentSmallestIndex;
      currentSmallestNum = Integer.MAX_VALUE;
    }

    return previousSmallestNum;
```

Run Save Reset

### Time & Space Complexity

The time complexity of the above algorithm will be $O(N * K)$. The algorithm runs in constant space $O(1)$.

## 2. Brute-force using Sorting

We can use an in-place sort like a **HeapSort** to sort the input array to get the Kth smallest number. Following is the code for this solution:

Java Python3 C++ JS

```
import java.util.*;

class KthSmallestNumber {

  public static int findKthSmallestNumber(int[] nums, int k) {
    Arrays.sort(nums);
    return nums[k - 1];
  }

  public static void main(String[] args) {
    int result = KthSmallestNumber.findKthSmallestNumber(new int[] { 1, 5, 12, 2, 11, 5 }, 3);
```

```
    System.out.println("Kth smallest number is: " + result);

    // since there are two 5s in the input array, our 3rd and 4th smallest numbers should be a '5'
    result = KthSmallestNumber.findKthSmallestNumber(new int[] { 1, 5, 12, 2, 11, 5 }, 4);
    System.out.println("Kth smallest number is: " + result);

    result = KthSmallestNumber.findKthSmallestNumber(new int[] { 5, 12, 11, -1, 12 }, 3);
    System.out.println("Kth smallest number is: " + result);
  }
}
```

Run Save Reset

### Time & Space Complexity

Sorting will take $O(NlogN)$ and if we are not using an in-place sorting algorithm, we will need $O(N)$ space.

## 3. Using Max-Heap

As discussed in Kth Smallest Number, we can iterate the array and use a **Max Heap** to keep track of 'K' smallest number. In the end, the root of the heap will have the Kth smallest number.

Here is what this algorithm will look like:

Java Python3 C++ JS

```
import java.util.*;

class KthSmallestNumber {

  public static int findKthSmallestNumber(int[] nums, int k) {
    PriorityQueue<Integer> maxHeap = new PriorityQueue<Integer>((n1, n2) -> n2 - n1);
    // put first k numbers in the max heap
    for (int i = 0; i < k; i++)
      maxHeap.add(nums[i]);

    // go through the remaining numbers of the array, if the number from the array is smaller than the
    // top (biggest) number of the heap, remove the top number from heap and add the number from array
    for (int i = k; i < nums.length; i++) {
      if (nums[i] < maxHeap.peek()) {
        maxHeap.poll();
        maxHeap.add(nums[i]);
      }
    }

    // the root of the heap has the Kth smallest number
    return maxHeap.peek();
  }

  public static void main(String[] args) {
    int result = KthSmallestNumber.findKthSmallestNumber(new int[] { 1, 5, 12, 2, 11, 5 }, 3);
    System.out.println("Kth smallest number is: " + result);

    // since there are two 5s in the input array, our 3rd and 4th smallest numbers should be a '5'
```

Run Save Reset

### Time & Space Complexity

The time complexity of the above algorithm is $O(K * logK + (N - K) * logK)$ which is asymptotically equal to $O(N * logK)$. The space complexity will be $O(K)$ because we need to store 'K' smallest numbers in the heap.

## 4. Using Min-Heap

Also discussed in Kth Smallest Number, we can use a **Min Heap** to find the Kth smallest number. We can insert all the numbers in the min-heap and then extract the top 'K' numbers from the heap to find the Kth smallest number.

### Time & Space Complexity

Building a heap with $N$ elements will take $O(N)$ and extracting 'K' numbers will take $O(K * logN)$. Overall, the time complexity of this algorithm will be $O(N + K * logN)$ and the space complexity will be $O(N)$.

## 5. Using Partition Scheme of Quicksort

Quicksort picks a number called **pivot** and partition the input array around it. After partitioning, all numbers smaller than the pivot are to the left of the pivot, and all numbers greater than or equal to the pivot are to the right of the pivot. This ensures that the pivot has reached its correct sorted position.

We can use this partitioning scheme to find the Kth smallest number. We will recursively partition the input array and if, after partitioning, our pivot is at the `K-1` index we have found our required number; if not, we will choose one the following option:

1. If pivot's position is larger than `K-1`, we will recursively partition the array on numbers lower than the pivot.
2. If pivot's position is smaller than `K-1`, we will recursively partition the array on numbers greater than the pivot.

Here is what our algorithm will look like:

Java | Python3 | C++ | JS

```java
import java.util.*;

class KthSmallestNumber {

  public static int findKthSmallestNumber(int[] nums, int k) {
    return findKthSmallestNumberRec(nums, k, 0, nums.length - 1);
  }

  public static int findKthSmallestNumberRec(int[] nums, int k, int start, int end) {
    int p = partition(nums, start, end);

    if (p == k - 1)
      return nums[p];

    if (p > k - 1) // search lower part
      return findKthSmallestNumberRec(nums, k, start, p - 1);

    // search higher part
    return findKthSmallestNumberRec(nums, k, p + 1, end);
  }

  private static int partition(int[] nums, int low, int high) {
    if (low == high)
      return low;

    int pivot = nums[high];
    for (int i = low; i < high; i++) {
      // all elements less than 'pivot' will be before the index 'low'
```

Run | Save | Reset

### Time & Space Complexity

The above algorithm is known as QuickSelect and has a Worst case time complexity of $O(N^2)$. The best and average case is $O(N)$, which is better than the best and average case of QuickSort. Overall, QuickSelect uses the same approach as QuickSort i.e., partitioning the data into two parts based on a pivot. However, contrary to QuickSort, instead of recursing into both sides QuickSelect only recurses into one side – the side with the element it is searching for. This reduces the average and best case time complexity from $O(N * logN)$ to $O(N)$.

The worst-case occurs when, at every step, the partition procedure splits the N-length array into arrays of size '1' and 'N-1'. This can only happen when the input array is sorted or if all of its elements are the same. This "unlucky" selection of pivot elements requires $O(N)$ recursive calls, leading to an $O(N^2)$ worst-case.

Worst-case space complexity will be $O(N)$ used for the recursion stack. See details under Quicksort.

## 6. Using Randomized Partitioning Scheme of Quicksort

As mentioned above, the worst case for **Quicksort** occurs when the partition procedure splits the N-length array into arrays of size '1' and 'N-1'. To mitigate this, instead of always picking a fixed index for pivot (e.g., in the above algorithm we always pick `nums[high]` as the pivot), we can randomly select an element as pivot. After randomly choosing the pivot element, we expect the split of the input array to be reasonably well balanced on average.

Here is what our algorithm will look like (only the highlighted lines have changed):

Java | Python3 | C++ | JS

```java
import java.util.*;
import java.util.Random;

class KthSmallestNumber {

  public static int findKthSmallestNumber(int[] nums, int k) {
    return findKthSmallestNumberRec(nums, k, 0, nums.length - 1);
  }

  public static int findKthSmallestNumberRec(int[] nums, int k, int start, int end) {
```

```
    int p = partition(nums, start, end);

    if (p == k - 1)
      return nums[p];

    if (p > k - 1) // search lower part
      return findKthSmallestNumberRec(nums, k, start, p - 1);

    // search higher part
    return findKthSmallestNumberRec(nums, k, p + 1, end);
  }

  private static int partition(int[] nums, int low, int high) {
    if (low == high)
      return low;

    Random randomNum = new Random();
    int pivotIndex = low + randomNum.nextInt(high - low);
```

Run Save Reset

### Time & Space Complexity #

The above algorithm has the same worst and average case time complexities as mentioned for the previous algorithm. But choosing the pivot randomly has the effect of rendering the worst-case very unlikely, particularly for large arrays. Therefore, the **expected** time complexity of the above algorithm will be $O(N)$, but the absolute worst case is still $O(N^2)$. Practically, this algorithm is a lot faster than the non-randomized version.

## 7. Using the Median of Medians #

We can use the Median of Medians algorithm to choose a **good pivot** for the partitioning algorithm of the **Quicksort**. This algorithm finds an approximate median of an array in linear time $O(N)$. When this approximate median is used as the pivot, the worst-case complexity of the partitioning procedure reduces to linear $O(N)$, which is also the asymptotically optimal worst-case complexity of any sorting/selection algorithm. This algorithm was originally developed by Blum, Floyd, Pratt, Rivest, and Tarjan and was describe in their 1973 paper.

This is how the partitioning algorithm works:

1. If we have 5 or less than 5 elements in the input array, we simply take its first element as the pivot. If not then we divide the input array into subarrays of five elements (for simplicity we can ignore any subarray having less than five elements).
2. Sort each subarray to determine its median. Sorting a small and fixed numbered array takes constant time. At the end of this step, we have an array containing medians of all the subarray.
3. Recursively call the partitioning algorithm on the array containing medians until we get our pivot.
4. Every time the partition procedure needs to find a pivot, it will follow the above three steps.

Here is what this algorithm will look like:

Java Python3 C++ JS

```
import java.util.*;

class KthSmallestNumber {

  public static int findKthSmallestNumber(int[] nums, int k) {
    return findKthSmallestNumberRec(nums, k, 0, nums.length - 1);
  }

  public static int findKthSmallestNumberRec(int[] nums, int k, int start, int end) {
    int p = partition(nums, start, end);

    if (p == k - 1)
      return nums[p];

    if (p > k - 1) // search the lower part
      return findKthSmallestNumberRec(nums, k, start, p - 1);

    // search the higher part
    return findKthSmallestNumberRec(nums, k, p + 1, end);
  }

  private static int partition(int[] nums, int low, int high) {
    if (low == high)
      return low;

    int median = medianOfMedians(nums, low, high);
    // find the median in the array and swap it with 'nums[high]' which will become our pivot
    for (int i = low; i < high; i++) {
```

Run Save Reset

### Time & Space Complexity #

The above algorithm has a guaranteed $O(N)$ worst-case time. Please see the proof of its running time here and under "Selection-based pivoting". The worst-case space complexity is $O(N)$.

## Conclusion #

Theoretically, the Median of Medians algorithm gives the best time complexity of $O(N)$ but practically both the Median of Medians and the Randomized Partitioning algorithms nearly perform equally.

In the context of **Quicksort**, given an $O(N)$ selection algorithm using the Median of Medians, one can use it to find the ideal pivot (the median) at every step of quicksort and thus produce a sorting algorithm with $O(NlogN)$ running time in the worst-case. Though practical implementations of this variant are considerably slower on average, they are of theoretical interest because they show that an optimal selection algorithm can yield an optimal sorting algorithm.

Back
Solution Review: Problem Challenge 2

Next
Where to Go from Here

Mark as Completed

Report an Issue
Ask a Question