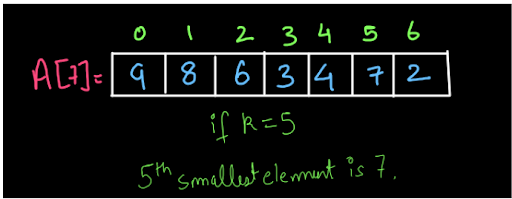
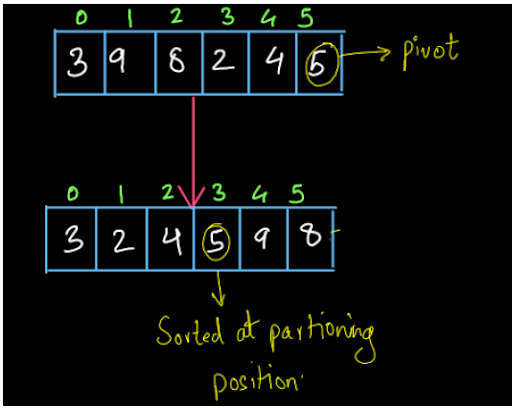
**1. Problem Discussion :**

We are given an array of integers, and we have to find the kth smallest element in the array using the Quick Select algorithm. For instance, have a look at the diagram given below:

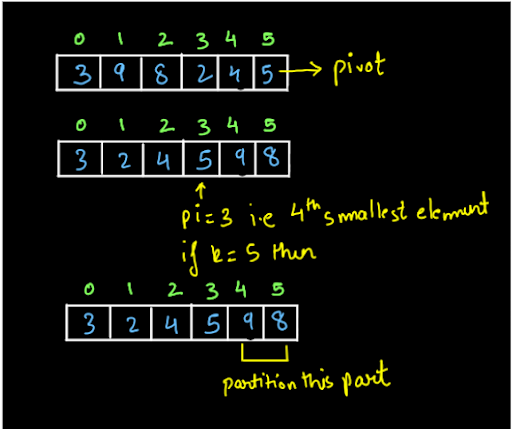
****

Because we were given k=5, the array's 5th smallest element is 7. The fascinating aspect of this problem is that it requires you to use the Quick Select technique to locate the kth smallest element. What makes this algorithm unique? We will be able to solve this problem in linear time complexity using the Quick Select Algorithm. How? That is exactly what you must consider!!! Don't worry, we've covered everything you need to know about this process in this post and the solution video. However, we encourage that you first try to solve the problem on your own before moving on to the answer.

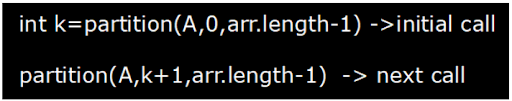
● Partition the array: The process for partitioning an array was already covered in the PARTITION AN ARRAY and QUICK SORT questions. The same technique is required here. As indicated in the picture below, we'll partition the array using the final element as a pivot element:

****

● So, after partitioning the array, the pivot element is in its sorted position, which means that all elements less than the pivot are before it in the array, and all elements greater than the pivot are after it. Check the Partition Position: If the partition position after partition is equal to k-1, the partition will be stopped and the element will be returned to the partition position. Otherwise, we'll pick which section of the array to partition next based on the value of partition position. This will be a call that recurses. Take a look at the illustration below:

****

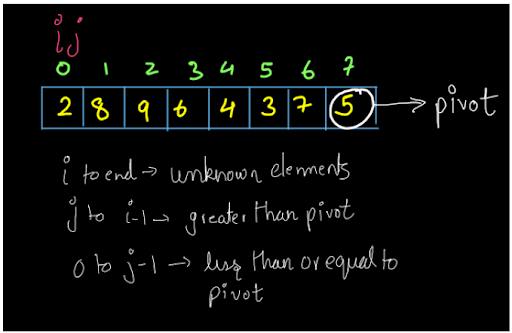
● We'll split in the right area of the array to acquire the 5th smallest element because partitioning got us the 4th smallest element (because partition index pi=3) and we were looking for the 5th smallest element. The initial and subsequent calls, based on the preceding example, would have been:

****

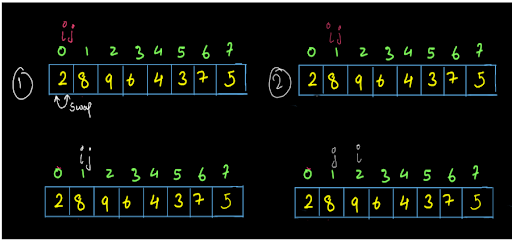
● This is neither the actual code, nor does it depict how the call will be made. Consider the code in the example above and try to solve the problem on your own before moving on to the solution's explanation.

**2. Approach :**

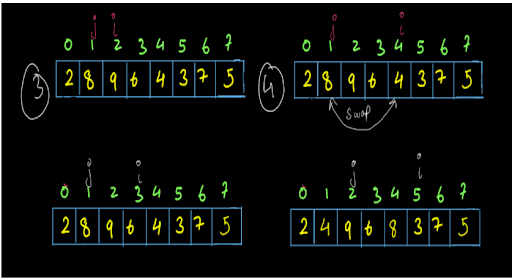
If you already know how to partition an array and how to solve the PARTITION AN ARRAY and QUICK SORT issues, you can skip this section because it is the same process. If you haven't studied this procedure before, we recommend watching the PARTITION AN ARRAY video and reading the complete procedure from here to fully comprehend the procedure.

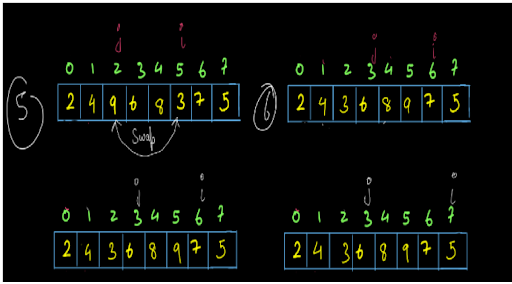
****

The variables I and 'j' are preserved at the beginning of the array in this case. The terms are spelled out below. These are the conditions: The members between I and the last index of the array I to end) must be unknown elements, which means we don't know their values in comparison to the pivot, i.e. whether they are smaller or larger. The items between j and i-1 are those that are larger (in terms of values) than the pivot. The items from 0 to j-1 will be less or equal to the pivot (in terms of values). While partitioning the array, we'll strive to keep these conditions in mind. The situation is as follows at first: Since i=0, all elements from I to the end are unknowns, i.e. we don't know anything about them right now. We don't have any items greater than pivot right now because j is also zero, and the same is true for smaller elements, i.e. we don't have any smaller Partition Procedure: If you already know how to partition an array and how to solve the PARTITION AN ARRAY and QUICK SORT issues, you can skip this section because it is the same process. If you haven't studied this procedure before, we recommend watching the PARTITION AN ARRAY video and reading the complete procedure from here to fully comprehend the procedure. Iterations:

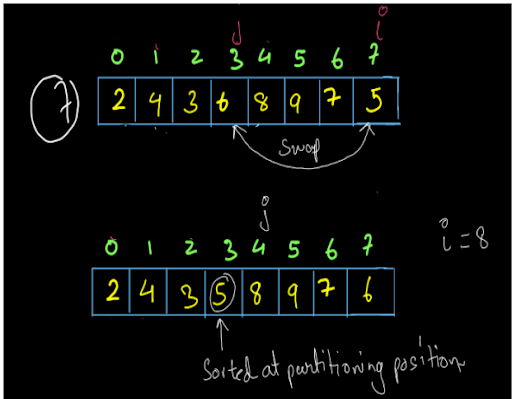
****

The figure above depicts the first and second iterations. We swapped arr[i] and arr[j] because arr[i] pivots, but I and 'j' were in the same position (i=j=0), so the swapping isn't visible (the array after swapping is shown below the first array and the next iteration is on the right side in the diagram). Then, after swapping, both the I and 'j' variables are incremented. Since arr[i]>pivot, no swapping occurs in the second iteration, and only I increments. In the same way, iterations 3-6 are depicted in the diagrams below. We suggest that you go through all of the iterations on your own.

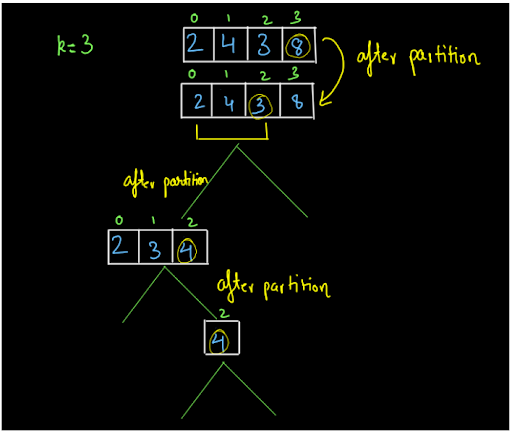
****

****

We've arrived at the final iteration. I have reached the end of the array, and j has arrived at index 3. We'll stop iterating now that I have reached arr.length. If you look at the diagram below, you'll note that pivot=5 is sorted at its partitioning position, which is j-1. As a result, the final partitioning position is j-1.

****

So, this is the entire partitioning procedure. I hope you got it. If you want to see this process in detail, you may refer to the video here. Now if we recursively perform the above algorithm till the time we get pivot element position equal to k then we will get our ans as in the sorted array our kth smallest element will be at kth position. Lets see the recursive tree of an example :

****

We used a tiny array to make the technique easier to understand. If we need to identify the third largest member in this array, we must first split it, with the final element serving as the pivot. Then, because the partition index (pi) we got was pi=3 and we're looking for the third-smallest element, we only have to partition the left half of the array this time, leaving the rest of the array alone, we only have to partition the left side of the array this time. We obtained 3 at its sorted position after the second partition, and pi=1. As a result, this time we must partition on the right side. We got pi=2 on the third partition and the third-smallest element, 4, on the fourth partition.

**3. Code :**

ConsoleJava

import java.io.\*;

import java.util.\*;

public class Main {

public static int quickSelect(int[] arr, int lo, int hi, int k) {

int pivot = arr[hi];

int pi = partition(arr, pivot, lo, hi);

if (pi > k)

{

return quickSelect(arr, lo, pi - 1, k);

}

else if (pi < k)

{

return quickSelect(arr, pi + 1, hi, k);

}

return pivot;

}

public static int partition(int[] arr, int pivot, int lo, int hi) {

System.out.println("pivot -> " + pivot);

int i = lo, j = lo;

while (i <= hi) {

if (arr[i] <= pivot) {

swap(arr, i, j);

i++;

j++;

} else {

i++;

}

}

System.out.println("pivot index -> " + (j - 1));

return (j - 1);

}

// used for swapping ith and jth elements of array

public static void swap(int[] arr, int i, int j) {

System.out.println("Swapping " + arr[i] + " and " + arr[j]);

int temp = arr[i];

arr[i] = arr[j];

arr[j] = temp;

}

public static void print(int[] arr) {

for (int i = 0; i < arr.length; i++) {

System.out.print(arr[i] + " ");

}

System.out.println();

}

public static void main(String[] args) throws Exception {

Scanner scn = new Scanner(System.in);

int n = scn.nextInt();

int[] arr = new int[n];

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

arr[i] = scn.nextInt();

}

int k = scn.nextInt();

System.out.println(quickSelect(arr, 0, arr.length - 1, k - 1)); //we passed k-1 as the kth smallest element will be index k-1 in the array

}

}

**4. Analysis:**

Time Complexity:

Average case time complexity will be O(n log n) as partitioning will happen in O(n) and on an average we are performing that log n times. Worst case time complexity will be O(n^2) when the array will be sorted. for better understanding of worst case time complexity read Quick Sort article.

Space Complexity:

We are not using any extra space in this program so our space complexity will be O(1).