Solution: Kth Largest Element in an Array

Let's solve the Kth Largest Element in an Array problem using the Top K Elements pattern.

We'll cover the following

Statement

Find the k^{th} largest element in an unsorted array.

> **Note:** We need to find the k^{th}largest element in the sorted order, not the k^{th} distinct element.

Constraints:

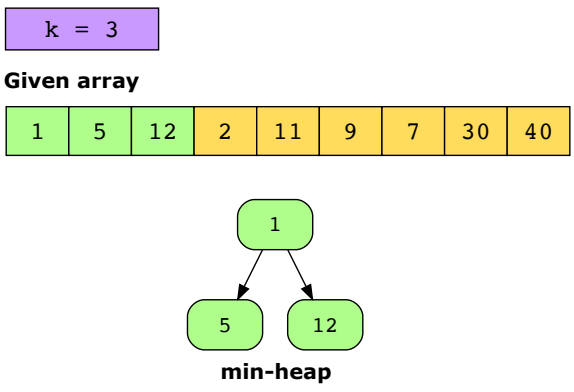
- $1 \leq$ `k` $\leq$ `nums.length` $\leq 10^4$
- $-10^4 \leq$ `nums[i]` $\leq 10^4$

Solution

Just like in K Closest Points to Origin, we can use a heap to obtain the k^{th} largest elements from our unsorted list. We now have to select the largest number, so we'll use a min-heap. We'll insert the first k elements into the min-heap.

As we know, the root is the smallest element in a min-heap. So, since we want to keep track of the k^{th} largest element, we can compare every number with the root while iterating through all the numbers in the list. If any number is greater than the root element, we'll take root out and insert the greater number. This will ensure that we always have the k largest element in the heap. In the end, we can simply return the root element as the k^{th} highest rank.

An illustration of this process is given below:



Insert the first k numbers into the heap.

Let's look at the code for the solution:

Java

```java
class KthLargestElement{
  public static int findKthLargest(int[] array, int k){

    PriorityQueue<Integer> kNumbersMinHeap = new PriorityQueue<Integer>((n1, n2) -> n1 - n2);
    // Put first k elements in the min heap
    for (int i = 0; i < k; i++)
      kNumbersMinHeap.add(array[i]);

    // Go through the remaining elements of the array, if the element from the array is greater than the
    // top (smallest) element of the heap, remove the top element from heap and add the element from arra
    for (int i = k; i < array.length; i++) {
      if (array[i] > kNumbersMinHeap.peek()) {
        kNumbersMinHeap.poll();
        kNumbersMinHeap.add(array[i]);
      }
    }

    // The root of the heap has the Kth largest element
    return kNumbersMinHeap.peek();
  }

```

```
      {1, 5, 12, 2, 11, 9, 7, 30, 20},
      {23, 13, 17, 19, 10},
      {3, 2, 5, 6, 7},
```

Kth Largest Element in an Array

Time complexity

The first loop adds k elements to the heap, which takes $O(k \log k)$ time. In the worst case, in the second loop, we may be pushing $n - k$ elements onto the heap, where n is the size of the input list. Each push and pop operation takes $\log k$ time, so $n - k$ items being pushed result in $O(n - k) \times \log k)$ time complexity.

The overall time complexity is thus $O(k \log k + (n - k) \times \log k)$, which simplifies to $O(n \log k)$.

Space complexity

The space complexity will be $O(k)$ because we are storing k elements in the heap.