Solution: Kth Largest Element in a Stream

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

We'll cover the following

Statement

Given an infinite stream of integers (sorted or unsorted), `nums`, design a class to find the k^{th} largest element in a stream.

> **Note**: It is the k^{th} largest element in the sorted order, not the k^{th} *distinct* element.

The class should have the following functions, inputs, and return values:

- **Init()**: It takes an array of integers and an integer k and initializes the class object.
- **Add(value)**: It takes one integer value, appends it to the stream, and calls the **Return kth largest()** function.
- **Return kth largest()**: It returns an integer value that represents the k^{th} largest element in the stream.

Constraints

- $1 \leq k \leq 10^3$
- $0 \leq$ `nums.length` $\leq 10^3$
- $-10^3 \leq$ `nums[i]` $\leq 10^3$
- $-10^3 \leq$ `value` $\leq 10^3$
- At most 10^3 calls will be made to add.
- It is guaranteed that there will be at least k elements in the array when you search for the k^{th} element.

Solution

So far, you have probably brainstormed some approaches and have an idea of how to solve this problem. Let's explore some of these approaches and figure out which one to follow based on considerations such as time complexity and any implementation constraints.

Naive approach

The naive solution is first to sort the data and then find k^{th} largest element. Insertion sort is an algorithm that can be used to sort the data as it appears. However, it also requires shifting the elements, greater than the inserted number, one place forward.

The overall time complexity of the algorithm becomes $O(n^2)$, where n is the number of elements in the data stream. The time complexity of each insertion is $O(n)$ and finding the k^{th} largest element would take $O(1)$ time, assuming we are storing the data in an array. The space complexity is $O(1)$.

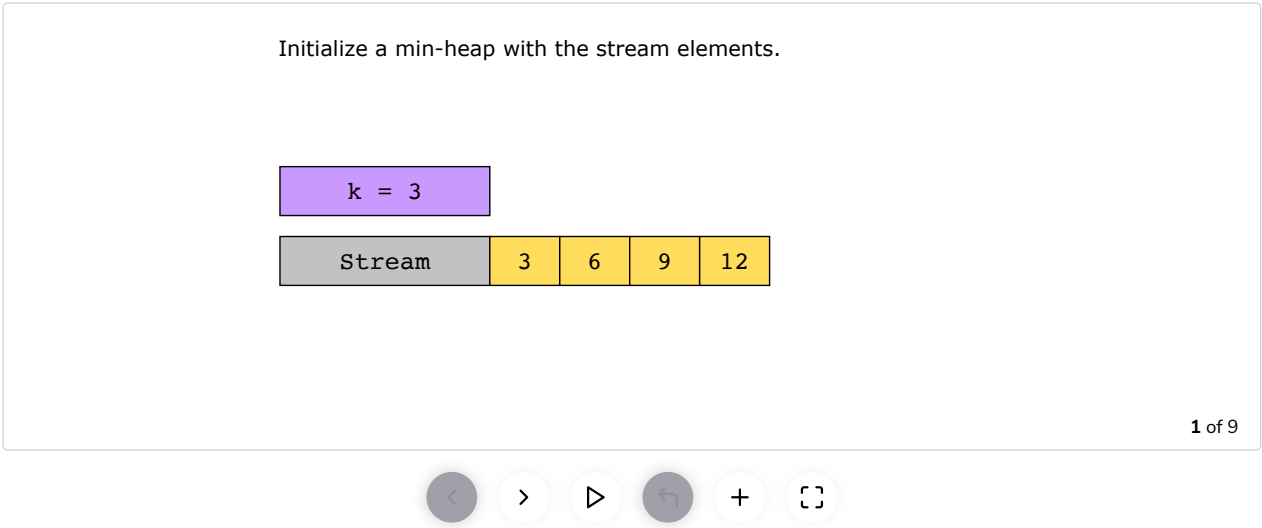
Optimized approach using Top K Elements

As new elements are added to the number stream, the k^{th} largest element keeps changing. We need to implement a class that caters to the dynamically changing numbers. The most efficient data structure for repeatedly finding the k^{th} largest number in a changing list is a heap.

We'll implement a min-heap of size k. In a min-heap, the smallest number is always at the top. This ensures that in a heap with k elements, the k^{th} largest element, is always at the top of the heap.

Heap summary:

1. Heap stores elements and can find the smallest or largest (max-heap or min-heap) element stored in $O(1)$.
2. Heap can add elements or remove the smallest (min-heap) or largest (max-heap) element in $O(\log(n))$.
3. Heap can perform insertions and removals while always maintaining the first property.



> **Note**: In the following section, we will gradually build the solution. Alternatively, you can skip straight to just the code.

Step-by-step solution construction

We'll start by defining the constructor. In the constructor, we create a min-heap using the elements from the array. Then, we pop from the heap until its size becomes equal to k.

Java

```
class KthLargest {
    public int k;
    public PriorityQueue<Integer> topKHeap;

    // constructor to initialize heap and add values in it
```

```
    public KthLargest(int k, int[] nums) {
        System.out.println("\tInitializing the heap");
        this.k = k;
        this.topKHeap = new PriorityQueue<Integer> ();
        for (int num: nums) {
            this.topKHeap.offer(num);
        }
        System.out.println("\t\tk = " + this.k);
        System.out.println("\t\tHeap: " + this.topKHeap);
        while (topKHeap.size() > k) {
            System.out.println("\t\tLength of the heap = " + this.topKHeap.size() + ", which is greater t
            System.out.print("\t\t\tPopping from the heap: " + this.topKHeap + " → ");
            topKHeap.poll();
            System.out.println(this.topKHeap);
        }
    }
    public static void main(String args[]) {
        int[] nums = { 3, 6, 9, 10 };
        System.out.println("Initial stream: " + Arrays.toString(nums));
        KthLargest KLargest = new KthLargest(3, nums);
    }
}
```

Kth Largest Element in a Stream

Next, we'll implement the *add* function that adds an incoming number to the stream. If the size of the heap exceeds k, we'll perform a pop operation until its size becomes equal to k. Since we're implementing a min-heap, the k^{th} largest element will be at the top of the heap.

Java

```
class KthLargest {
    public int k;
    public PriorityQueue<Integer> topKHeap;
    private String printHeapWithMarkers(PriorityQueue<Integer> arr, int pvalue) {
        String out = "[";
        Iterator value = arr.iterator();
        for (int i = 0; i<arr.size(); i++) {
            if (pvalue == i) {
                out += '«';
                out += value.next().toString() + '»' + ", ";
            } else if (i != arr.size() - 1)
                out += value.next().toString() + ", ";
            else
                out += value.next().toString();
        }
        out += "]";
        return out;
    }
    // constructor to initialize topKHeap and add values in it
    public KthLargest(int k, int[] nums) {
        System.out.println("\tInitializing the topKHeap");
        this.k = k;
        topKHeap = new PriorityQueue<Integer> ();
        for (int num: nums) {
            topKHeap.offer(num);
        }
        System.out.println("\t\tk = " + this.k);
```

Kth Largest Element in a Stream

We know that the k^{th} largest element is the top element of the heap. After adding the new element to the heap, we'll call the `returnKthLargest()` function. We'll return its top element.

Java

```java
import java.util.*;

class KthLargest {
    public int k;
    public PriorityQueue<Integer> topKHeap;
    private String printHeapWithMarkers(PriorityQueue<Integer> arr, int pvalue) {
        String out = "[";
        Iterator value = arr.iterator();
        for (int i = 0; i<arr.size(); i++) {
            if (pvalue == i) {
                out += '«';
                out += value.next().toString() + '»' + ", ";
            } else if (i != arr.size() - 1)
                out += value.next().toString() + ", ";
            else
                out += value.next().toString();
        }
        out += "]";
        return out;
    }
    // constructor to initialize topKHeap and add values in it
    public KthLargest(int k, int[] nums) {
        System.out.println("\tInitializing the topKHeap");
        this.k = k;
        topKHeap = new PriorityQueue<Integer> ();
        for (int num: nums) {
            topKHeap.offer(num);
        }
```

Kth Largest Element in a Stream

Just the code

Here's the complete solution to this problem:

Java

```java
import java.util.*;

class KthLargest {
    public int k;
    public PriorityQueue<Integer> topKHeap;
    private String printHeapWithMarkers(PriorityQueue<Integer> arr, int pvalue) {
        String out = "[";
        Iterator value = arr.iterator();
        for (int i = 0; i<arr.size(); i++) {
            if (pvalue == i) {
                out += '«';
                out += value.next().toString() + '»' + ", ";
            } else if (i != arr.size() - 1)
                out += value.next().toString() + ", ";
            else
                out += value.next().toString();
        }
        out += "]";
        return out;
    }
    // constructor to initialize topKHeap and add values in it
    public KthLargest(int k, int[] nums) {
        this.k = k;
        topKHeap = new PriorityQueue<Integer> ();
        for (int num: nums) {
            topKHeap.offer(num);
        }
        while (topKHeap.size() > k) {
```

Solution summary

To recap, the solution to this problem can be divided into the following parts:

- If the size of the heap $> k$, pop from the heap.
- Return the top element of the min-heap.

Time complexity

- **Constructor():** First, we are constructing a heap from the `nums` list received in the constructor. Adding n numbers one by one to a heap that was initially empty takes $O(n \log n)$. In the second step, we pop $n - k$ elements from the heap. For the maximum value of $k = n$, the pop operations take a total of $O(n \log n)$. So the time complexity of the constructor is $O(n \log n + n \log n)$, which is $O(n \log n)$.
- **Add(value):** Adding an element to a heap of size k takes $O(\log k)$ time, so the time complexity of this function is $O(\log k)$.
- **Return kth largest():** This function simply looks up and returns the heap's top element. Its time complexity is $O(1)$.

Space complexity

The space complexity will be $O(n)$ where n is the length of the input stream. The extra space involved is because of the heap. We added n elements first and then it ends up with size k, so the overall space complexity is $O(n)$.

Back
Kth Largest Element i...

Next
Reorganize String

Mark as Completed