Solution: Sliding Window Median

Let's solve the Sliding Window Median problem using the Two Heaps pattern.

We'll cover the following

Statement

Given an integer array, `nums`, and an integer, `k`, there is a sliding window of size `k`, which is moving from the very left to the very right of the array. We can only see the `k` numbers in the window. Each time the sliding window moves right by one position.

Given this scenario, return the median of the each window. Answers within 10^{-5} of the actual value will be accepted.

Constraints:

- $1 \leq$ `k` $\leq$ `nums.length` $\leq 10^5$
- $-2^{31} \leq$ `nums[i]` $\leq 2^{31} - 1$

Solution

So far, you've 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 to use nested loops to traverse over the array. The outer loop ranges over the entire array, and the nested loop is used to iterate over windows of k elements. For each window, we'll first sort the elements and then compute the median. We'll append this value to the median list and move the window one step forward.

The above algorithm will have a total time complexity of $O(n-k+1) \cdot O(k \log k)$, that is, $O((n-k) \cdot (k \log k))$:

- Traversal: $O(n-k+1)$, where k is the window size and n is the number of elements in the array.
- Sorting: Assuming we perform sorting using quicksort, the complexity would be $O(k \log k)$.

The space complexity would be $O((n-k) \cdot \log k)$ because the space complexity of quicksort is $O(\log k)$.

Optimized approach using two heaps

Since the median, let’s call it x, is the middle value in a list of sorted elements, we know that half of the elements will be smaller than (or equal to) x, and the other half will be greater than (or equal to) x.

We can divide the list into two halves. One half will store the smaller numbers—let’s call it `smallList`. The other half will store the larger numbers—let’s call it `largeList`. If the total number of elements is odd, we keep the $\lceil \frac{n}{2} \rceil^{th}$ element in `smallList`. The median will be the largest number in `smallList` if the number of elements is odd. Otherwise, if the total number of elements is even, the median will be the mean of the largest number in `smallList` and the smallest number in `largeList`. The best data structure for finding the smallest or largest number in a list of numbers is a heap.

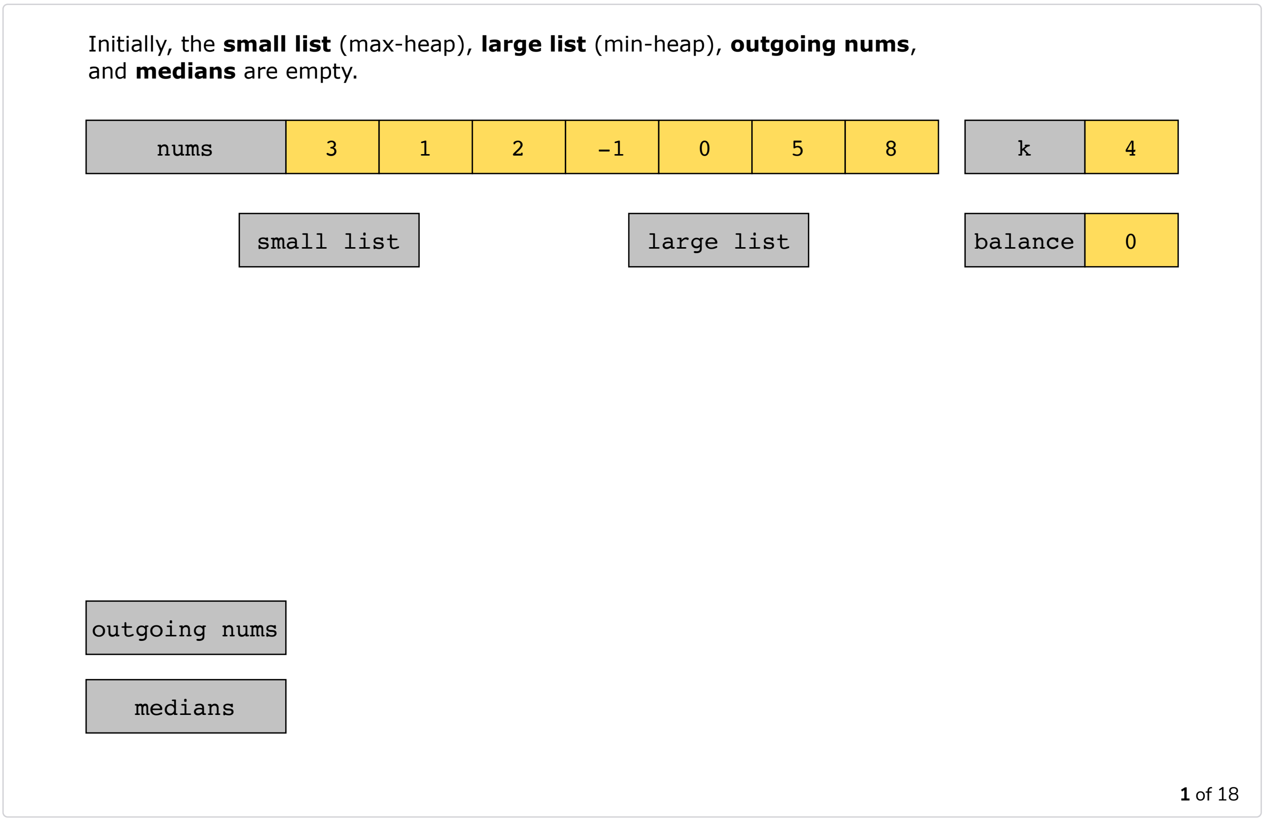
- We store the first half of the numbers of the window in a max-heap. We use a max-heap because we want to know the largest number in the first half of the window.
- We use a min-heap to store the second half of the numbers, since we want to know the smallest number in the second half of the window.

Having said that, the two heaps pattern is a perfect fit for this problem.

While sliding the window over the array, as soon as an element leaves the window, we need to remove it form the heap, too. Removing an element from the heap takes $O(m)$ time, where m is the size of the heap. This is the time spent in finding the element in the heap. This makes our sliding function very costly. Therefore, we don’t remove an element right away. Instead, we maintain a hash map to keep track of the elements to be removed from the heaps. We only remove an element from the heap if it’s at the top of the heap, which reduces the time complexity to $O(logn)$. In addition, if the element to be removed is not at the top of the heap, it doesn’t disturb the calculation of the medians.

Here’s what the algorithm will do:

1. Add `k` elements to `smallList`. Since we’re implementing a max-heap, we’ll multiply each element with -1.
2. Transfer the top $\lfloor \frac{k}{2} \rfloor$ elements from `smallList` to `largeList`. We’ll multiply each element by -1 again to convert it to its original value.
3. Append the median to the median list. In case of an odd window size, the median is the top element of `smallList`. Otherwise, it’s the mean of the top elements of the two lists.
4. We use a `balance` variable to check if $\lceil \frac{k}{2} \rceil$ members belonging to the sliding window are present in the max-heap (`smallList`). If there’s an extra element, we transfer the top element from the max-heap to the min-heap. The second highest element then springs to the top of the max-heap. Similarly, if more than $\lfloor \frac{k}{2} \rfloor$ elements end up in the min-heap (`largeList`), we restore the balance by popping the smallest element from the min-heap and adding it to the max-heap.
5. Move the window one step forward and add the outgoing element to the hash map. If the top element of the small list or the large list is present in the hash map with a frequency greater than 0, we remove it from the respective heap.
6. Repeat steps 3–5 until all the numbers in the `nums` have been processed.


Initially, the **small list** (max-heap), **large list** (min-heap), **outgoing nums**, and **medians** are empty.

Let's look at the code for this solution below:

Java

```java
class SlidingWindow{
    public static double[] medianSlidingWindow(int[] nums, int k) {
        // To store the medians
        List<Double> medians = new ArrayList<Double>();

        // To keep track of the numbers that need to be removed from the heaps
        HashMap<Integer, Integer> outgoingNum = new HashMap<>();

        // Max heap
        PriorityQueue<Integer> smallList = new PriorityQueue<>(Collections.reverseOrder());

        // Min heap
        PriorityQueue<Integer> largeList = new PriorityQueue<>();

        // Initialize the max heap
        for (int i = 0; i < k; i++)
            smallList.offer(nums[i]);

        // Transfer the top 50% of the numbers from max heap to min heap
        for (int i = 0; i < k / 2; i++)
            largeList.offer(smallList.poll());

        // Variable to keep the heaps balanced
        int balance = 0;

        int i = k;
        while (true) {
```

Solution summary

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

- Populate max-heap with `k` elements.
- Transfer $\lfloor \frac{k}{2} \rfloor$ elements from the max-heap to the min-heap.
- If the window size is odd, the median is the top of the max-heap. Otherwise, it's the mean of the top elements of the two heaps.
- Move the window forward and add the outgoing number in the hash map, which is used to track the outgoing numbers.
- Rebalance the heaps if they have more elements.
- If the top element of the max-heap or the min-heap is present in the hash map with a frequency greater than 0, this element is irrelevant. We remove it from the respective heap and the hash map.
- Repeat the process until all elements are processed.

Time complexity

The time spent in creating the heaps is $O(klogk)$. As we process all elements in the loop, it runs for $O(n-k)$ time, where n is the size of the input array. Inside the loop, we push and pop elements from the heaps, which takes $O(log(n-k))$ time. This is because the size of a heap can grow up to $O(n-k)$ in the worst case. Therefore, the total time complexity of the algorithm above is $O(klogk) + O((n-k) * log(n-k))$. As $k \leq n$, the time complexity can be represented as $O(nlogn)$.

Space complexity

As the size of a heap can grow up to $O(n-k)$ in the worst case, the space occupied by it is $O(n-k)$. Similarly, the space occupied by the hash map will also be $O(n-k)$. As $k \leq n$, the space complexity of the above algorithm is $O(n)$.

← Back

Next →