Solution: Find Median from a Data Stream

Let's solve the Find Median from a Data Stream problem using the Two Heaps pattern.

We'll cover the following

Statement

Our task is to implement a data structure that will store a dynamically growing list of integers and provide access to their median in $O(1)$.

Constraints:

- $-10^5 \leq$ `num` $\leq 10^5$, where `num` is an integer received from the data stream.
- There will be at least one element in the data structure before the median is computed.
- At most, 5×10^4 calls will be made to the function.

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 first sort the data and then find the median. Insertion sort is an algorithm that can be used to sort the data as it appears. This way, every time a new number is added to the stream, the numbers before that new number are already sorted, allowing insertion at the correct index. This, however, also requires us to shift 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 amortized time complexity of each insertion is therefore $O(\frac{n^2}{n})$, that is, $O(n)$. The time complexity of the function that calculates the median would be $O(1)$, assuming we are storing the data in an array. The space complexity is $O(1)$.

Optimized approach using two heaps

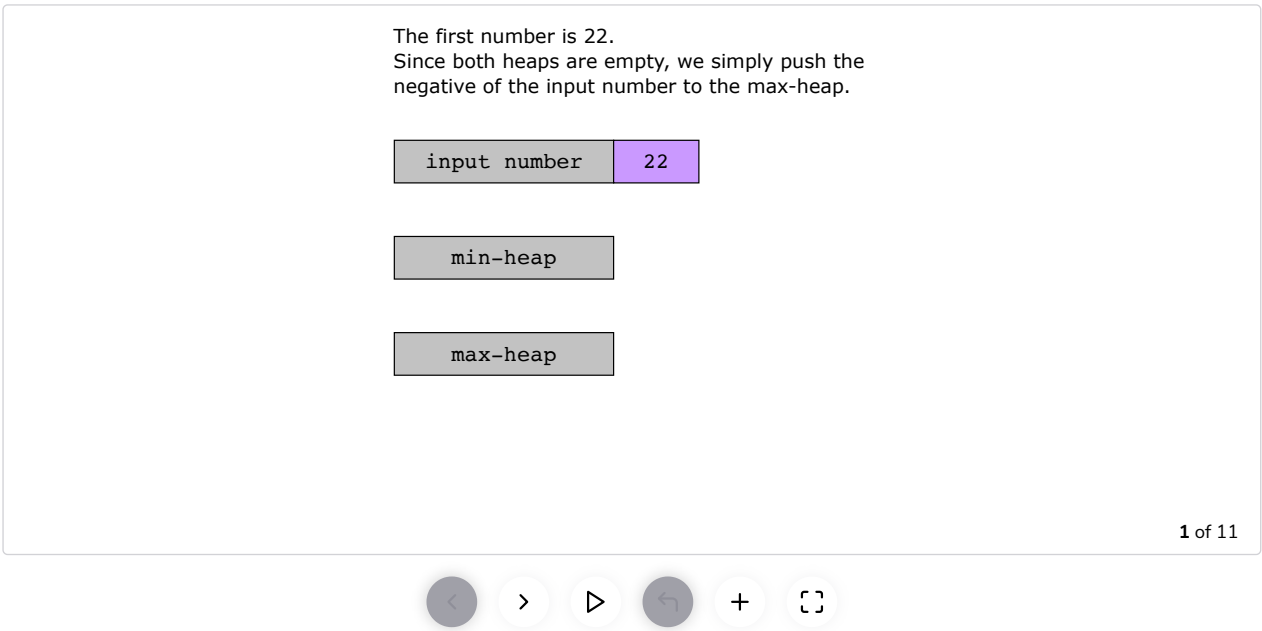
We'll assume that **x** is the median of the integers in a list. Half of the numbers in the list will be smaller than (or equal to) x, and the other half will be greater than (or equal to) x. We can divide the list into two halves. One half stores the smaller numbers, and the other half stores the larger numbers.

The median of all numbers will either be the largest number in the small list or the smallest number in the large list. If the total number of elements is even, we know that the median will be the average of these two numbers.

The most efficient data structure for repeatedly finding the smallest or largest number in a changing list is a heap. We can store the first half of the numbers in a max-heap and the second half in a min-heap. That's why the two heaps pattern is a perfect fit.

Here's how we'll implement this algorithm:

1. First, we'll store the first half of the numbers (smaller than x) in a max-heap. We use a max-heap because we want to know the largest number in the first half of the list.
2. Then, we'll store the second half of the numbers (larger than x) in a min-heap because we want to know the smallest number in the second half of the list.
3. We can calculate the median of the current list of numbers using the top element of the two heaps.



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

Java

```java
class MedianOfAStream {

  PriorityQueue<Integer> maxHeapForSmallNum; //containing first half of numbers
  PriorityQueue<Integer> minHeapForLargeNum; //containing second half of numbers

  public MedianOfAStream() {
    maxHeapForSmallNum = new PriorityQueue<>((a, b) -> b - a);
    minHeapForLargeNum = new PriorityQueue<>((a, b) -> a - b);
  }

  public void insertNum(int num) {
    if (maxHeapForSmallNum.isEmpty() || maxHeapForSmallNum.peek() >= num)
      maxHeapForSmallNum.add(num);
    else
      minHeapForLargeNum.add(num);

    // either both the heaps will have equal number of elements or max-heap will have one
    // more element than the min-heap
    if (maxHeapForSmallNum.size() > minHeapForLargeNum.size() + 1)
      minHeapForLargeNum.add(maxHeapForSmallNum.poll());
    else if (maxHeapForSmallNum.size() < minHeapForLargeNum.size())
      maxHeapForSmallNum.add(minHeapForLargeNum.poll());
  }
```

```
24
25    public double findMedian() {
26      if (maxHeapForSmallNum.size() == minHeapForLargeNum.size()) {
27        // we have even number of elements, take the average of middle two elements
28        return maxHeapForSmallNum.peek() / 2.0 + minHeapForLargeNum.peek() / 2.0;
```

Find Median from a Data Stream

Solution summary

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

- Use a min-heap to store the larger 50% of the numbers seen so far and a max-heap for the smaller 50% of the numbers.
- Add the incoming elements to the appropriate heaps.
- Calculate the median using the top elements of the two heaps.

The time complexity of the Insert Num method will change depending on how many numbers have already been received from the stream. So, the time complexity is amortized over the number of insert operations.

Each insert operation will trigger a heapify process that runs in $O(\log n)$ times, where n is the count of numbers received so far from the stream. Because of this, the cumulative complexity of a sequence of n insert operations is described by the expression $\log 1 + \log 2 + \log 3 + \ldots + \log n$.

This expression simplifies to $\log n!$, which, as per Stirling's approximation, is $O(n \log n)$. As we have performed n insert operations, the amortized time complexity of one insert operation is $O(\frac{n \log n}{n})$, that is, $O(\log n)$.

The time complexity of the Find Median method will be $O(1)$ since retrieving the top element of a heap is a constant-time operation, and we need to do at most two such retrievals.

Space complexity

The space complexity will be $O(1)$ since no additional space is used other than that which is required to store the numbers received from the data stream.

Back
Find Median from a D...

Next
Sliding Window Median

Mark as Completed