Find Median from Data Stream

Problem Level: Hard

Problem Description:

You are given a stream of N integers. For every i^{th} integer added to the running list of integers, print the resulting median.

Sample Input 1:

```
6
6 2 1 3 7 5
```

Sample Output 1:

```
6 4 2 2 3 4
```

Explanation:

```
S = {6}, median = 6
S = {6, 2} → {2, 6}, median = 4
S = {6, 2, 1} → {1, 2, 6}, median = 2
S = {6, 2, 1, 3} → {1, 2, 3, 6}, median = 2
S = {6, 2, 1, 3, 7} → {1, 2, 3, 6, 7}, median = 3
S = {6, 2, 1, 3, 7, 5} → {1, 2, 3, 5, 6, 7}, median = 4
```

Approach to be followed:

To solve this problem, we need to use 2 heaps. Let's say we are taking input for the i^{th} element. If we have the previous i - 1 numbers sorted, then we can easily add the number to its appropriate place, we will retain the sorted list, as well as find the median. So we need to figure out a way to add this number to its correct place in the appropriate time.

Let's say we had two sorted arrays. The first array holds the smaller half of the numbers in decreasing order. The second array contains the larger half of the numbers in increasing order. Now, after taking input for the i^{th} number, we can easily decide which half this number belongs to and add it there in the appropriate place. If any of the arrays becomes much larger than the other array, we can remove the first element from that array and add it to the appropriate place of the other array.

Now let's look at the indices of the array where we need to access at any moment for getting the median. If the total number of elements is odd, then we need the first element of the array with the higher number of elements. If the total number of elements is even, then we need the average of the first elements of both the arrays. So the only indices we need to access while getting the median and adding the elements are the first index of both of the arrays. To achieve this efficiently, we shall use heaps.

We will use a max heap for storing data of the smaller half of the numbers and a min heap for storing data of the larger half of the numbers. Let's see how we will add the numbers and get the medians.

Steps:

1. While adding the i^{th} number we will check the following conditions and add accordingly.
    a. If the number is greater than or equal to the max element of the max heap then it surely belongs to the larger half of the numbers i.e. the min heap. So we will add it there.
    b. Otherwise, we will add it to the max heap.
2. It may happen that one of the heaps becomes much larger than the other if we add the numbers in this way.
3. To stop this situation from taking place, check the size of the heaps after adding every number.
    a. If the difference between the number of elements of the two heaps becomes more than one, pop the top element from the heap with more elements and push that element to the other heap.
4. This way, the difference can never be more than one.
5. To get the median after adding the i^{th} number we will check if i^{th} is odd or even.
    a. If it is odd, then surely one of the heap has one more element than the other. The median will be the top element of that heap then.
    b. If i^{th} is even, then the two heaps must have the same number of elements. So the median will be the average of the top elements of the two heaps.

Pseudo Code:

```
function printRunningMedian(arr[], n)

    declare minHeap, maxHeap


    // For each element in the data stream

    loop from i = 0 till i < n

        if maxHeap.size() and arr[i] greater than maxHeap.top()

            minHeap.push(arr[i])

        else

            maxHeap.push(arr[i])


        if absolute of (maxHeap.size() - minHeap.size()) greater than 1

            if maxHeap.size() greater than minHeap.size()

                temp = maxHeap.top()

                maxHeap.pop()

                minHeap.push(temp)

            else

                temp = minHeap.top()

                minHeap.pop()

                maxHeap.push(temp)


        median

        totalSize = maxHeap.size() + minHeap.size()


        // When number of elements is odd

        if totalSize % 2 equals 1)

            if maxHeap.size() greater than minHeap.size()

                median = maxHeap.top()

            else

                median = minHeap.top()
```

```
        // When number of elements is even
        else
            median = 0
            if maxHeap is not empty:
                median = median + maxHeap.top()
            if minHeap is not empty:
                median = median + minHeap.top()
                    median = median / 2

        print median
    end loop
end function
```

Time Complexity: O(N * log(N)), where **N** denotes the number of elements in the array. Time complexity to insert an element in a heap is **log(n)**. So to insert n elements, the overall time complexity is **O(N * log(N))**.