Median of a Stream

Given that integers are read from a data stream. Find median of elements read so for in an efficient way. For simplicity assume, there are no duplicates. For example, let us consider the stream 5, 15, 1, 3 ...

```
After reading 1st element of stream - 5 -> median - 5
After reading 2nd element of stream - 5, 15 -> median - 10
After reading 3rd element of stream - 5, 15, 1 -> median - 5
After reading 4th element of stream - 5, 15, 1, 3 -> median - 4, so on...
```

Making it clear, when the input size is odd, we take the middle element of sorted data. If the input size is even, we pick the average of the middle two elements in the sorted stream.
Note that output is the *effective median* of integers read from the stream so far. Such an algorithm is called an online algorithm. Any algorithm that can guarantee the output of *i*-elements after processing *i*-th element, is said to be ***online algorithm***. Let us discuss three solutions to the above problem

Method : Using Heaps
We can use a max heap on the left side to represent elements that are less than *effective median*, and a min-heap on the right side to represent elements that are greater than *effective median*.

After processing an incoming element, the number of elements in heaps differs utmost by 1 element. When both heaps contain the same number of elements, we pick the average of heaps root data as *effective median*. When the heaps are not balanced, we select *effective median* from the root of the heap containing more elements.

Given below is the implementation of the above method. For the algorithm to build these heaps, please read the highlighted code.


```java
// Java code to implement the approach

import java.io.*;
import java.util.*;

class GFG {

    // Function to find the median of stream of data
    public static void streamMed(int A[], int N)
    {
        // Declaring two min heap
        PriorityQueue<Double> g = new PriorityQueue<>();
        PriorityQueue<Double> s = new PriorityQueue<>();
        for (int i = 0; i < N; i++) {

            // Negation for treating it as max heap
            s.add(-1.0 * A[i]);
            g.add(-1.0 * s.poll());
            if (g.size() > s.size())
                s.add(-1.0 * g.poll());

            if (g.size() != s.size())
                System.out.println(-1.0 * s.peek());
            else
                System.out.println((g.peek() - s.peek())
                                   / 2);
        }
    }

    // Driver code
    public static void main(String[] args)
    {
        int A[] = { 5, 15, 1, 3, 2, 8, 7, 9, 10, 6, 11, 4 };
```

```
            int N = A.length;

            // Function call
            streamMed(A, N);
        }
    }
```

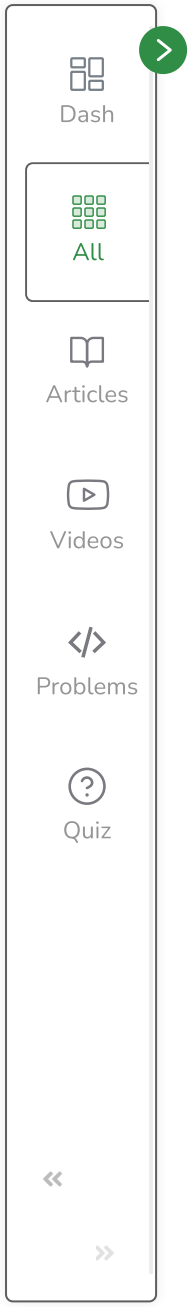

Output

```
5
10
5
4
3
4
5
6
7
6.5
7
6.5
```

Time Complexity: If we omit the way how stream was read, complexity of median finding is ***O(N log N)***, as we need to read the stream, and due to heap insertions/deletions.

Auxiliary Space: O(N)
At first glance the above code may look complex. If you read the code carefully, it is simple algorithm.

Mark as Read



Report An Issue

If you are facing any issue on this page. Please let us know.