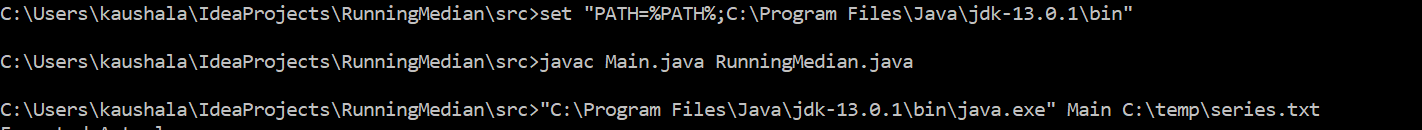
Computing Running Median

**Problem Statement:**

Finding a median of an incoming stream of data is relatively simple if you can store all the values in the memory/storage. However, if we are imposed with memory/storage constraints, finding an exact solution is difficult. If the data has some characteristics we can exploit and develop efficient specialized solutions. For example, if the stream type is integral, we can find an approximate solution if we can leverage lessons from counting sort and probability density function. This code is an attempt to do the same

**Usage:**

The code is written in Java. Provided you have the JRE setup, set the jdk path, compile and run



**Design Core:**

We can define a median such that the probability of all the numbers less than or equal to it is >= 0.5 and probability of all the numbers greater than or equal to it is >= 0.5. For every incoming number, I maintain the following

//  
// Number of times a value or  
// range of values has occurred  
//  
public int occurrenceCount;  
  
//  
// occurrenceCount / Total count of series  
//  
public double probabilityValue;  
  
//  
// Stores sum of probability of all  
// the numbers less than or equal to  
// that of a given value or range of values  
//  
public double probabilityOfNumbersLTE;  
  
//  
// Stores sum of probability of all  
// the numbers greater than or equal to  
// that of a given value or range of values  
//  
public double probabilityOfNumbersGTE;  
  
//  
// How far is this value/range from being  
// the median.  
//  
public double distanceFromMedian;

I compute distanceFromMedian as follows:

Probablity p;  
distanceFromMedian = 0;  
distanceFromMedian += p.probabilityOfNumbersGTE < 0.5 ?  
 (0.5 - p.probabilityOfNumbersGTE) : 0;  
distanceFromMedian += p.probabilityOfNumbersLTE < 0.5 ?  
 (0.5 - p.probabilityOfNumbersLTE) : 0;  
p.distanceFromMedian = distanceFromMedian;

As soon as I reach the limit to store the entries, we need to club entries whose distanceFromMedian is higher (less likely to be candidates for mean), and are consecutive, to make way for new entries. This is what we call zooming out.

We try to zoom in as much as we can on potential mean candidates. As soon as our mean starts hitting on zoomed out portion, we try to decouple the clubbed entries and zoom in.

Internally we maintain TreeMap (Which keeps keys sorted and acts as “Count Sort”) to store the values or range of values from the series as the key, and the above probability information as its value.

CountMap = new TreeMap<Range, Probablity>(new RangeCompare());

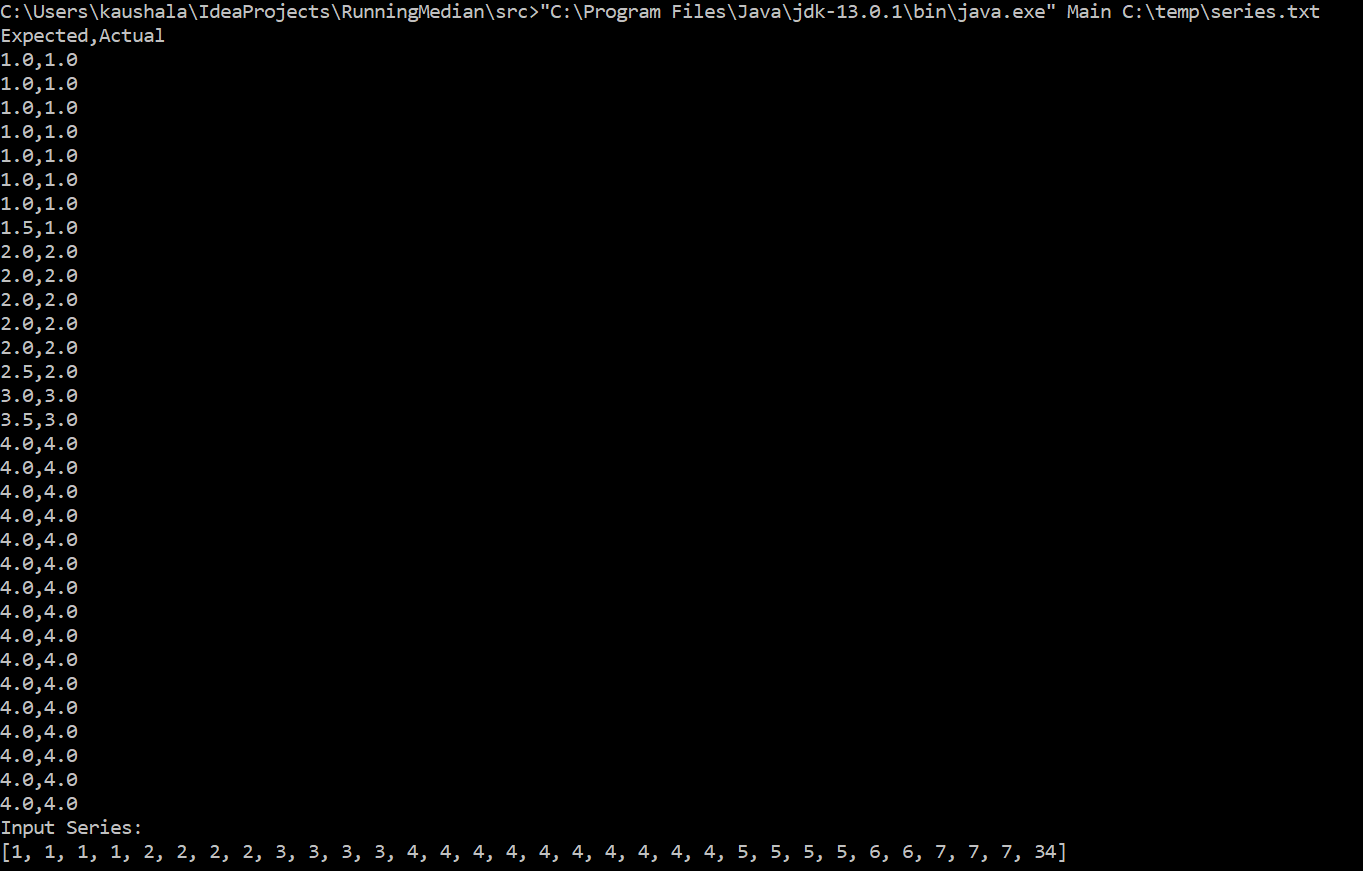
**API**

*Insert(int value)* – API to feed the integral value from the series. This call goes and check if an entry for this value already exists, or a range for this value already exists. If yes, it goes and updates the occurrence count.

If an entry for the same doesn’t exists we create a new entry and insert it. If that does cause us to go above the size limit, we begin the process of grouping the entries (By computing distance from the mean), and bring us within the size limits.

*getMedian()*- API to get the current median. The way we calculate the median, is we divide the total count of incoming values from the series by 2 and then hit our TreeMap (Count hashMap) to see which entry qualifies for it. If we do release that we have reached an entry which symbolizes a range of values, rather than a value, we begin the process of ungrouping that range, and grouping some place else. Because we now know that this has become a hot region for finding a median. Yes we are not proactive, meaning we do not do this calculation on every input, rather on a every getMedian call. Just for less processing, but that can be changed.

**Sample Usage**

****