Istanbul Technical University
Faculty of Computer and Informatics
Computer Engineering Department

BLG 335E
Analysis of Algorithms I Homework 2

Hasan Fatih Durkaya - 150200074

December 2nd, 2022

Contents

1 Introduction

In this homework, we are asked to create a system to store and organize data according to expectations. For this purpose, we have to follow some rules to manage data. This an important topic becausersituationts like this homework that includeing tens of thousands of data maybe millions are everywhere nowadays and storing it efficiently is an important problem .

2 Data Structures

In this homework I have used heap and vectors. Heap data structure is used to build a heap then apply heap sort on that because of it's lower complexity when we compare it to other sorting algorithms. Vectors are used for general storing purposes it's dynamic structure is an important properity.

2.1 Heap

Heap is a special tree-based data structure which is nearly a complete tree that has to satisfy one critical condition: Heap Properity. For instance, a max-heap for any node parent must be greater or equal to it's child. In a min-heap this case is reversed parent must be lower or equal to it's child's key value. The first node is called "root". We use a tree-based approach but we store heaps in arrays or vectors and there is relation between parent - child indexes. If we know the index of a parent we can acces to it's child without special getter function.One of the most important topics about heaps is heap sort. Heap sort is an sorting algorithm which uses heap data structure and has a time complexity O(nlogn) in all cases. This is important because it gives us the stability that we seek.

2.2 Vectors

Vectors are nearly the same with dynamic arrays with ability to manage it's size when an element is inserted or deleted. I have used vectors to store 2 different values: expected functions that we are looking for like median, firstq (wanted values) and all the dataset that contain gap,gi,grp and v (dataset).

Vectors used

```
vector<string> wanted_values;
vector<float> dataset;
```

Wanted values is string because we are not going to make an operation on them we will check just what are those on the other hand dataset is float because we have to make some operations with precision.

3 Basics of Implementation

In this part, I am going to cover critical points of implementation. We are looking for efficiency and lower computational cost, in order to achieve that there are some key points.

Adding

```
if(temp == "add"){
    if(expected_temp == "gap"){
        if(counter == 0){
                getline(fin, first_date, ',');
                getline(fin, first_hour, ',');
```

For example, when we are adding we don't want to store the values that we are not interested in. If we are looking for mean of gap we don't have store grp or gi in vector . It will cause huge amount of memory usage , to overcome this memory efficiency problem I have create an if-else structure and the type of parameter is gap we only take the first elements of the input stream after the dates. On the other hand inside if loop checks the is it the first time that we are getting inputs to get the first date and first hour to print.In this case we are just ignoring the other 3 types (grp, gi,v) of parameters because we don't need them at all.

Printing

```
for(int i = 0; i < size; i++){
    if(wanted_values[i] == "mean"){
        myfile << get_mean(dataset, counter);
                //cout<<get_mean(dataset, counter);
                if(func_count < size){
                    myfile << ",";
                    func_count++;
                }
```

Another key point is we are not calling the functions if we are adding we are calling them only if we are printing something out because if we call them in every input that might cause memory problems. We are checking wanted values vector's elements in a closed loop to call the proper function.

4 Functions

4.1 Get Mean

Function to calculate mean, this is a basic mean function. Taking input parameters the dataset and a counter to get the size. Basically adding all the values up and dividing result by size.

Algorithm 1 The calculating mean of an vector

```
function getMean(dataset, counter)
    sum
    mean
    for n < counter do
        n = n + dataset[n]
    end for
    mean = sum/counter
    return mean
end function
```

Code for Get Mean

```
float get_mean(vector<float> datas, int counter){
    float mean;
    float sum;

    for(int i = 0; i < counter; i++)
        sum = sum + datas[i];

    mean = (sum / counter);
    return mean;
}
```

Time complexity of this function is O(n) because we are linearly iterating the vector once and since there is only n elements complexity is O(n)

4.2 Get Min

Function to calculate minimum, this is a basic min function. Taking input parameters the dataset, is sorted flag and a counter to get the size. Basically if is sorted flag is 0 which means dataset is not sorted we use linear search to get the minimum value. On the other hand if flag is 1 it means dataset is already sorted so we just have to get the first value because it is the minimum

Algorithm 2 The calculating mean of a vector

```
function getMin(dataset, counter,isSorted)
    sum
    min = dataset[0]
    if isSorted == 1 then
        return min
    end if
    for n < counter do
        if min > dataset[n] then
            min = dataset[n]
        end if
    end for
    return min
end function
```

Code for Get Min

```
float get_min(vector<float> datas, float counter, int is_sorted){

    float min = datas[0];

    if(is_sorted == 1){
        return min;
    }
    else{
        for(int i = 0; i < counter; i++){
            if(min > datas[i])
                min = datas[i];
        }
        return min;
    }
}
```

Time complexity of this function is O(n) if the dataset is not sorted because we use linear search but if it is sorted complexity is O(1) since we simply return first element.

4.3 Get Max

Function to calculate maximum, this is a basic max function. Taking input parameters the dataset, is sorted flag and a counter to get the size. Basically if is sorted flag is 0 which means dataset is not sorted we use linear search to get the maximum value. On the other hand if flag is 1 it means dataset is already sorted so we just have to get the last value because it is the maximum.

Algorithm 3 The calculating max of a vector

```
function GETMAX(dataset, counter,isSorted)
    sum
    max = dataset[0]
    if isSorted == 1 then
        return dataset[counter]
    end if
    for n < counter do
        if max < dataset[n] then
            max = dataset[n]
        end if
    end for
    return max
end function
```

Codeode for Get Max

```
float get_max(vector<float>datas, double counter, int is_sorted){

    float max = datas[0];

    if(is_sorted == 1){
        return datas[counter];
    }
    else{

        for(int i = 0; i < counter; i++){

            if(max < datas[i]){
                max = datas[i];
            }
        }
        return max;
    }
}
```

Time complexity of this function is O(n) if the dataset is not sorted because we use linear search but if it is sorted complexity is O(1) since we simply return last element.

4.4 Get Standard Deviation

Function to calculate Standard Deviation. Taking input parameters the dataset and a counter to get the size. First we are getting the mean with get mean function then iterating over the dataset and subtracting mean from the value after that taking the square of the result and adding them up . At last dividing it to size minus 1 and taking the root.

Algorithm 4 The calculating standard deviation of a vector

function GETSTD(*dataset*, *counter*)
  std
  mean =GETMEAN(*dataset*,*counter*)
  for `n < counter` **do**
    if `max < dataset[n]` **then**
      $std+ = (dataset[n] - mean) * (dataset[n] - mean)$
    end if
  end for
  return $(std/counter - 1)^(1/2)$
end function

Code for Get Standard Deviation

```
float get_std(vector<float> datas, float counter){
    double mean = get_mean(datas,counter);
    double std;

    for(int i = 0; i < counter; i++){
        std += (datas[i] - mean) * (datas[i] - mean);
    }

    return sqrt(std/(counter-1));
}
```

Time complexity of this function is O(n) because again we are just linearly iterating over a vector.

4.5 Get Median

Function to calculate Median. Taking input parameters dataset and counter to get the size. First of all to get the median we should sort the vector. After the sorting operation we check if the size is odd we take the middle element if it is even, we are calculating the mean of middle two elements.Sorting algorithm is all there is and heapSort will be covered later parts.

Algorithm 5 The calculating median of a vector

```
function getMedian(dataset, counter)
    heapSort(dataset,counter)
    if size mod 2 == 0 then
        return (dataset[counter/2] + dataset[counter/2 − 1])/2
    end if
    if size mod 2 == 1 then
        return dataset[counter/2]
    end if
end function
```

Code for Get Median

```
float get_median(vector<float>datas, int size){

    heapSort(datas,size);

    if(size % 2 == 0)
        return ((datas[size/2] + datas[size/2-1])/2);
    else{
        return datas[size/2];
    }
}
```

Time complexity of this function is same as time complexity of heapSort which is nlogn , analysis of this system will be covered in later parts.

4.6 Get First Quartile

Function to calculate first quartile. First quartile is the value under which 25 percent of the data points are found , when they are sorted in increasing order. It is an important feature understand the meaning of data.

Algorithm 6 The calculating first quartile of a vector

function GETFIRSTQ($dataset$, $counter$)
  HEAPSORT($dataset$,$counter$)
  $step = 100/counter - 1$
  $left = 25/step$
  $right = left + 1$
  $lcoef = 1 - (25 - left * step)/step$
  $rcoef = 1 - lcoef$
  $quartile = (dataset[left] * lcoef) + (dataset[right_bound] * rcoef)$
end function

Code for Get First Quartile

```
float get_first_quartile(vector<float>datas, int size){

    heapSort(datas,size);

    double step = 100/double(size - 1);
    int left_bound = (25/step);

    int right_bound = (left_bound + 1);

    float l_coefficent = 1 - ((25 - left_bound * step) / (float)step);
    float r_coefficent = 1 - l_coefficent;

    float quartile = datas[left_bound]*l_coefficent+datas[right_bound]*r_co

    return quartile;
}
```

Time complexity of this function is same as time complexity of heapSort because of other parts of this function is constant time.

4.7 Get Third Quartile

Function to calculate third quartile. Third quartile is the value higher which 75 percent of the data points are found , when they are sorted in increasing order. It is an important feature understand the meaning of data.

Algorithm 7 The calculating third quartile of a vector

function GETTHIRDQ($dataset$, $counter$)
  HEAPSORT($dataset$,$counter$)
  $step = 100/counter - 1$
  $left = 75/step$
  $right = left + 1$
  $lcoef = 1 - (75 - left * step)/step$
  $rcoef = 1 - lcoef$
  $quartile = (dataset[left] * lcoef) + (dataset[right_bound] * rcoef)$
end function

Code for Get Third Quartile

```
float get_third_quartile(vector<float>datas, int size){

    heapSort(datas,size);

    double step = 100/double(size - 1);
    int left = (75/step);

    int right = (left + 1);

    float l_coef = 1 - ((75 - left_bound * step) / (float)step);
    float r_coef = 1 - l_coefficent;

    float quartile = (datas[left]*l_coef)+(datas[right]*r_coef);

    return quartile;
}
```

Time complexity of this function is same as time complexity of heapSort because of other parts of this function is constant time.

4.8 Heapify

Function to build heap data structure which satisfies heap properity. Heap structure is the essential part of heapSort.

Algorithm 8 Heapify

function HEAPIFY(*dataset*, *counter*, *subroot*)
  $left = (2 * sub_root) + 1$
  $right = (2 * sub_root) + 2$
  temp
  $largest = subroot$
  if `left < counter  dataset[left] > dataset[largest]` **then**
    $largest = left_child$
  end if
  if `right < counter  dataset[right] > dataset[largest]` **then**
    $largest = right$
  end if
  if `largest != subroot` **then**
    $temp = dataaset[subroot]$
    $dataset[subroot] = dataset[largest]$
    $dataset[largest] = temp$
    HEAPIFY(*dataset*,*counter*,*largest*)
  end if
end function

Code for Heapify

```
void heapify(vector<float> &datas, float size, int sub_root){
    int left_child = (2 * sub_root) + 1;
    int right_child = (2 * sub_root) + 2;

    float temp_var;
    float largest = sub_root;

    if(left_child < size && datas[left_child] > datas[largest])
        largest = left_child;
    if(right_child < size && datas[right_child] > datas[largest])
        largest = right_child;

    if(largest != sub_root){
        temp_var = datas[sub_root];
        datas[sub_root] = datas[largest];
        datas[largest] = temp_var;
        heapify(datas, size, largest);
    }
}
```

Time complexity of this function is n because this is the procedure where we create the the heap and we are doing this in O(n).

4.9 HeapSort

Function to sort the created heap, to use in the third and first quartile calculations.

Algorithm 9 HeapSort

```
function HEAPSORT(dataset, counter)
    temp
    for n < (counter/2)-1 do
        HEAPIFY(dataset,counter, i)
    end for
    for n < counter - 1 do
        temp = dataaset[0]
        dataset[0] = dataset[n]
        dataset[n] = temp
        HEAPIFY(dataset,counter,0)
    end for
end function
```

Code for HeapSort

```
void heapSort(vector<float> &datas, float size){
    float temp_var;

    for(int i = (size/2) - 1; i >= 0; i--)
        heapify(datas, size, i);
    for(int i = (size-1); i >= 0;i--){
        temp_var = datas[0];
        datas[0] = datas[i];
        datas[i] = temp_var;

        heapify(datas, i,0);
    }
}
```

Time complexity of this function is logn because we are sorting in a tree based-approach.

5 Experiments

In this part I will demonstrate the experiments and tests. This will shows the behaviour of the system and how our functions will treat in huge datasets.

Figure 1: First Quartile

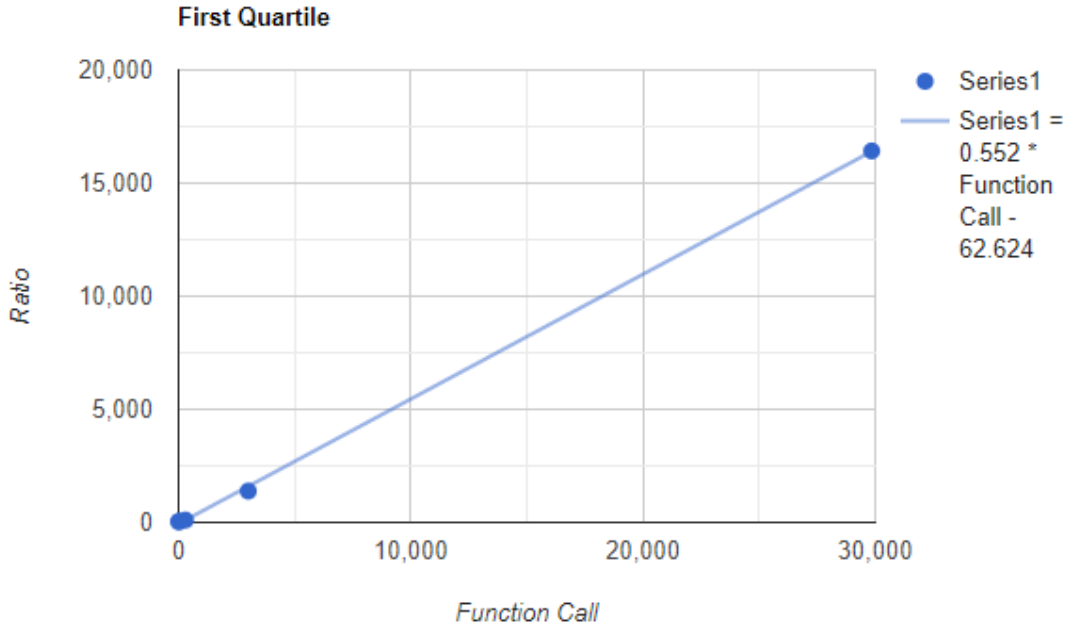


Figure 2: Third Quartile

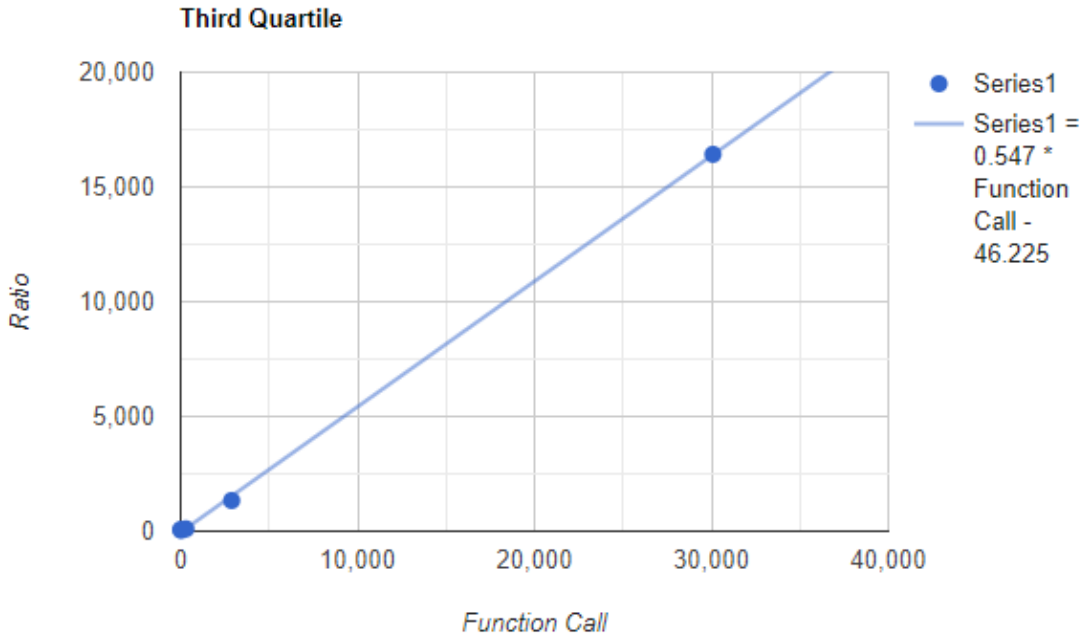


Figure 3: Max

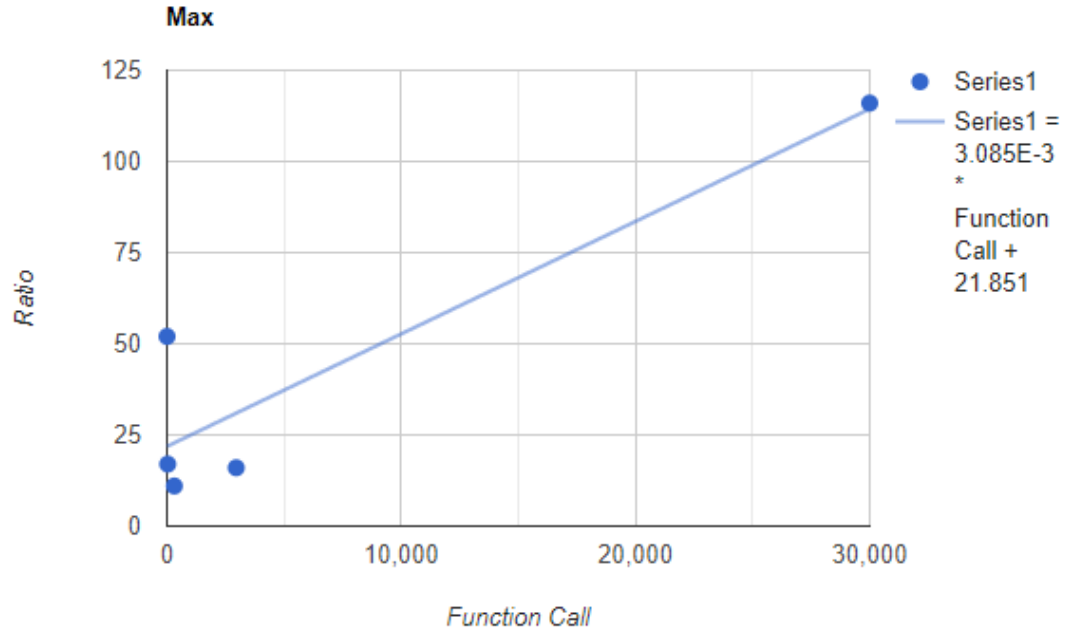


Figure 4: Min

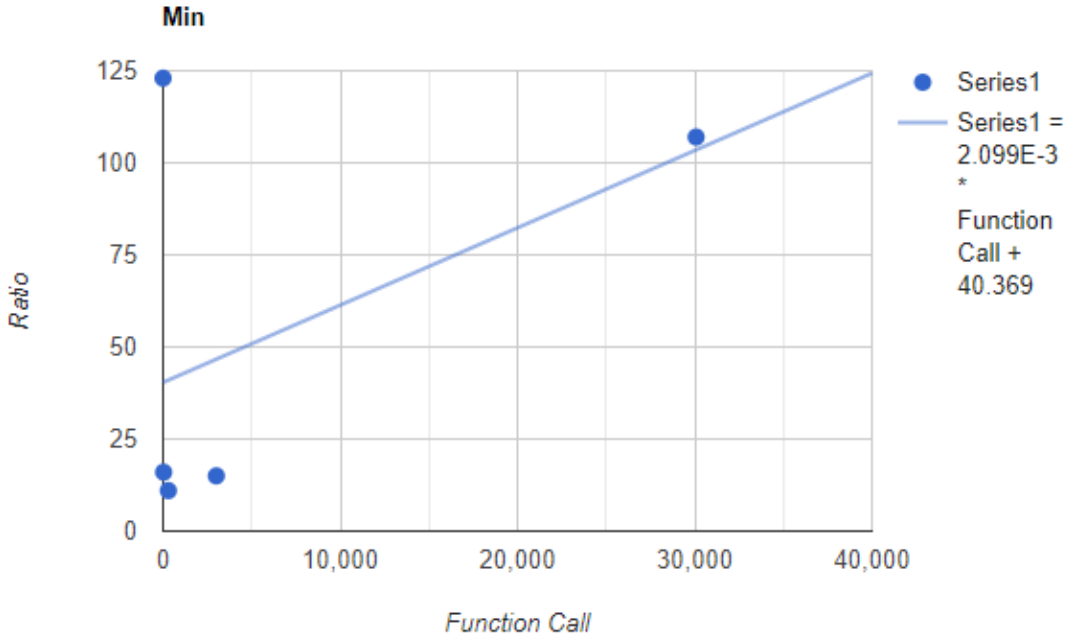


Figure 5: Mean

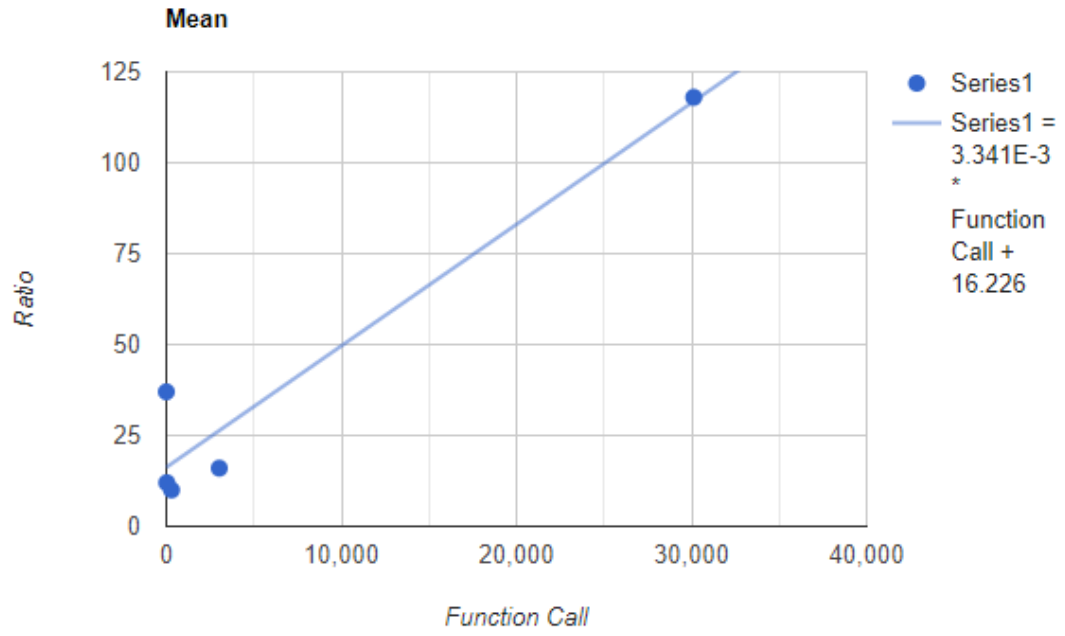


Figure 6: Standard Deviation

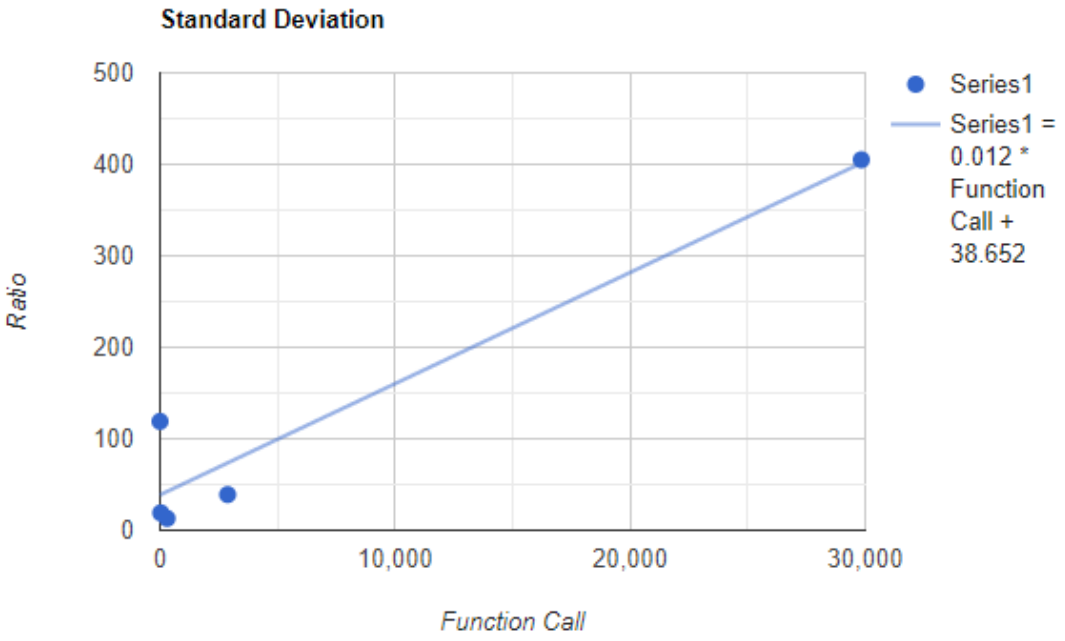
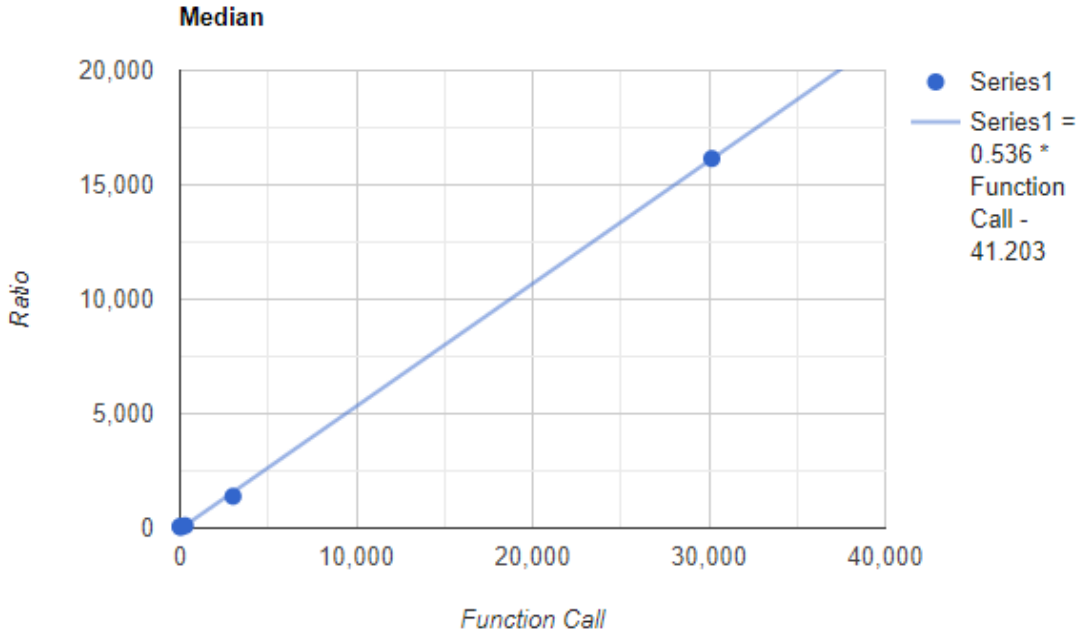


Figure 7: Median



All of the graphs are really look like linear lines because the quantities are huge and in the very small parts it changes qucikly , for example in 10 datas sample , it is quite quick but when working with 100000 datas sample the computation cost and time is large

6 Sliding Window

Sliding Window is a simple algorithm to one way system can move towards in a kernel size. For example we have ten input stream and we have a windows size 3, starting from a selected position window will start to progress the stream by sliding the window . With this approach we will have the first 3 elements and when we slide one step we will have 2,3,4 elements.

For this homework, We can assume there is 10 element array . Window will start from 0 index and ends at a special position F. We take K minimum elements among all F elements and store them in a max heap. When we created the first max-heap we will move forward one step ahead. Heap will be updated.After that again window will slide one step but this time heap will not be updated. Updating the max-heap only depends on the windows neew elements relation with the removed element of window. This procedure will continue until all the heap is processed and the selected size of max-heap will be found. Complexity analysis of this algorithm, worst case complexity of sliding window algorithm would be $O(N^2 * K)$.$Kisincludedasittakes O(K) complexity to build a heap of K elements$.