OUR TAKE ON BIG O NOTATION AND SORTING ALGORITHMS.

Roberto Materano
xqqshia@hotmail.com
Yuarth Hernández
yuarth07@gmail.com

https://github.com/xQQsHia/OurTakeOnBigONotation

SUMMARY: *Our take on Big O Notation and Sorting Algorithms is about taking a deep look in what happens inside Selection Sort, Bubble Sort, Insertion Sort, Quick Sort and Merge Sort. It's capable of showing the execution time of all of those algorithms using 100, 1000 and 10000 elements (random integers) as input. Through of this process it will graph its results. It's all implemented in C++.*

KEYWORDS: array, bigO, c++, time, sorting algorithms.

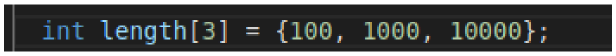
What we need to do:

1) Create an array of size 100, 1000 and 1000.
2) Fill them up with random integers.
3) Calculate each execution time for each sorting algorithm.
4) Graph the results.

STEPS:

1 CREATING ARRAYS

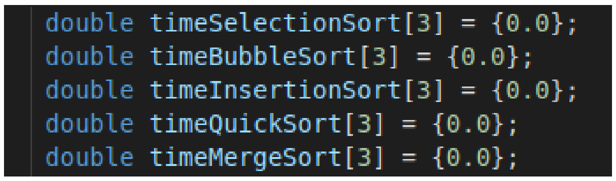
Obviously our first step is to create the arrays of size 100, 1000 and 10000. They'll be filled up later with random integers which will be sorted with the algorithms required.



```
int length[3] = {100, 1000, 10000};
```

This way we already have all the arrays.

Now we need to add new arrays to store all the times that later will be calculated.

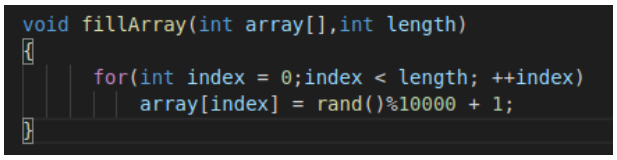


```
double timeSelectionSort[3] = {0.0};
double timeBubbleSort[3] = {0.0};
double timeInsertionSort[3] = {0.0};
double timeQuickSort[3] = {0.0};
double timeMergeSort[3] = {0.0};
```

Now we have all the arrays that we will need to store their execution time.

2 FILLING THEM UP WITH RANDOM INTEGERS

This a really important step, because this way we will make harder for our algorithms to sort the arrays.

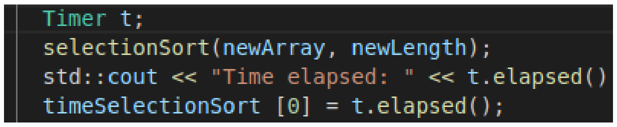


```
void fillArray(int array[],int length)
{
    for(int index = 0;index < length; ++index)
        array[index] = rand()%10000 + 1;
}
```

As it can be seen above, it is happening exactly what we want.

3 CALCULATE EACH EXECUTION TIME FOR EACH SORTING ALGORITHM

Now let's take a look of an example of how it calculates the execution time of Selection Sort.



```
Timer t;
selectionSort(newArray, newLength);
std::cout << "Time elapsed: " << t.elapsed()
timeSelectionSort [0] = t.elapsed();
```

Likewise, its execution time is stored in its correspondent array of time. The same process is done with all the algorithms.

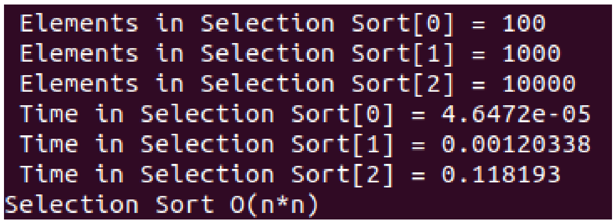
4 GRAPH THE RESULTS

So this is the most important step, because here you will notice the complexity of each sorting algorithm with their execution time for all the sizes we were asked to implement: 100, 1000 and 10000.

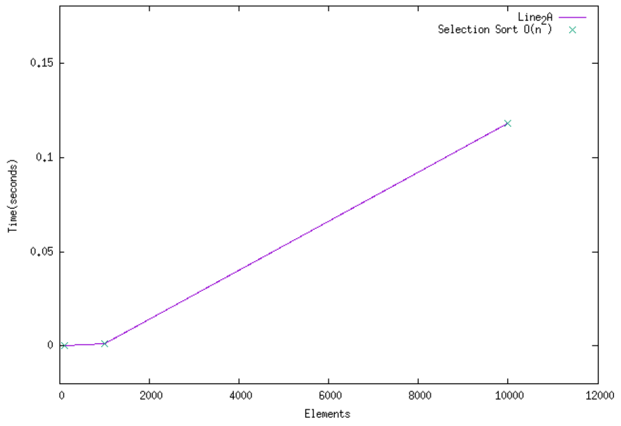
To graph all the results we used GNUPLOT.

To see the graph you need to access all the three sizes which will be shown in the graph as mere points with their respective execution times. At the same time, the points will get connected to draw up its final graph.

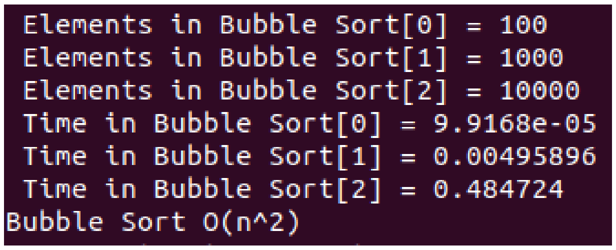
4.1 SELECTION SORT O(n^2)



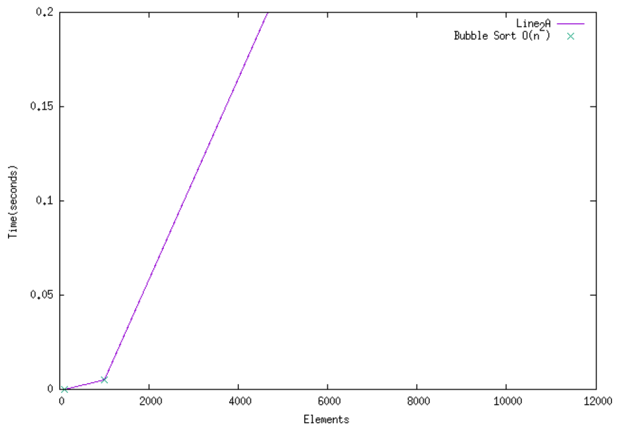
```
 Elements in Selection Sort[0] = 100
 Elements in Selection Sort[1] = 1000
 Elements in Selection Sort[2] = 10000
 Time in Selection Sort[0] = 4.6472e-05
 Time in Selection Sort[1] = 0.00120338
 Time in Selection Sort[2] = 0.118193
Selection Sort O(n*n)
```



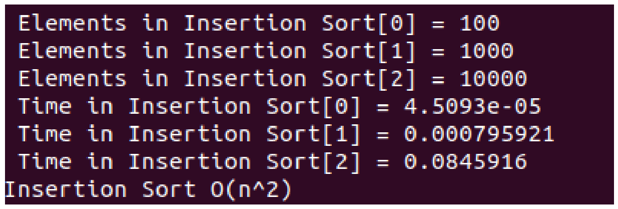
4.2 BUBBLE SORT O(n^2)



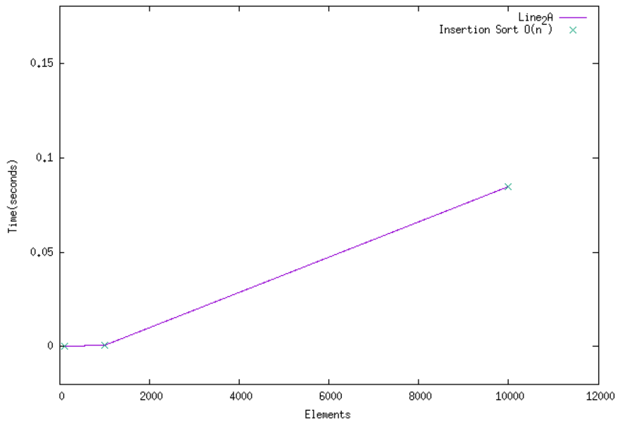
```
 Elements in Bubble Sort[0] = 100
 Elements in Bubble Sort[1] = 1000
 Elements in Bubble Sort[2] = 10000
 Time in Bubble Sort[0] = 9.9168e-05
 Time in Bubble Sort[1] = 0.00495896
 Time in Bubble Sort[2] = 0.484724
Bubble Sort O(n^2)
```



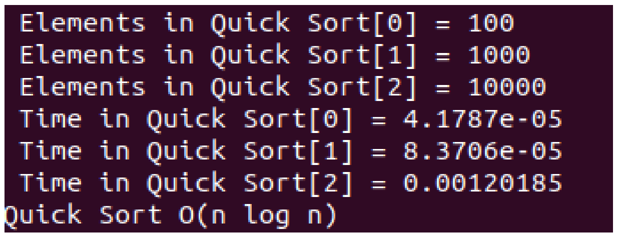
4.3 INSERTION SORT O(n^2)



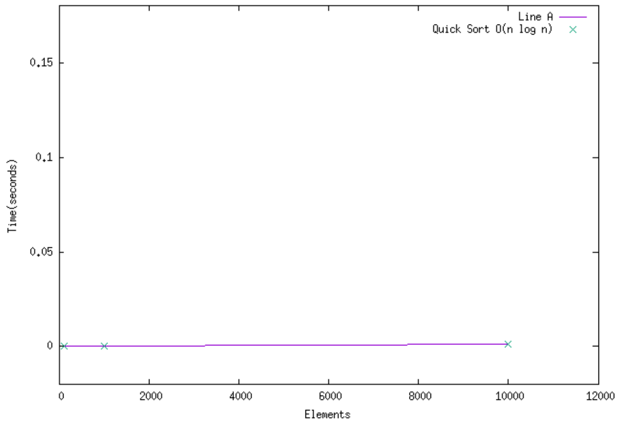
```
 Elements in Insertion Sort[0] = 100
 Elements in Insertion Sort[1] = 1000
 Elements in Insertion Sort[2] = 10000
 Time in Insertion Sort[0] = 4.5093e-05
 Time in Insertion Sort[1] = 0.000795921
 Time in Insertion Sort[2] = 0.0845916
Insertion Sort O(n^2)
```



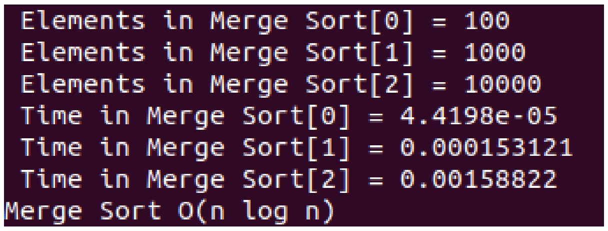
4.4 QUICK SORT O(n log n)



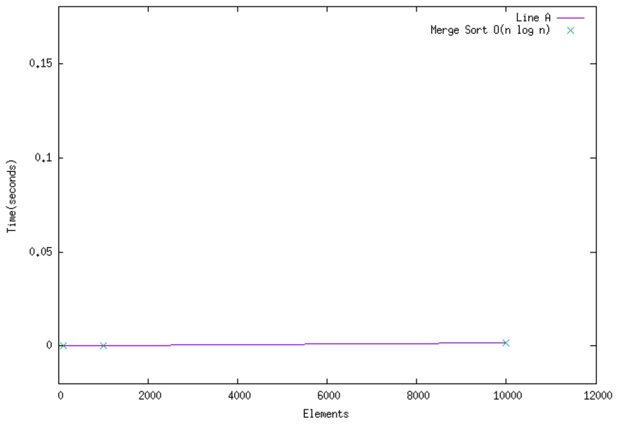
```
 Elements in Quick Sort[0] = 100
 Elements in Quick Sort[1] = 1000
 Elements in Quick Sort[2] = 10000
 Time in Quick Sort[0] = 4.1787e-05
 Time in Quick Sort[1] = 8.3706e-05
 Time in Quick Sort[2] = 0.00120185
Quick Sort O(n log n)
```



4.5 MERGE SORT O(n log n)



```
 Elements in Merge Sort[0] = 100
 Elements in Merge Sort[1] = 1000
 Elements in Merge Sort[2] = 10000
 Time in Merge Sort[0] = 4.4198e-05
 Time in Merge Sort[1] = 0.000153121
 Time in Merge Sort[2] = 0.00158822
Merge Sort O(n log n)
```



CONCLUSIONS

It is clearer now with the results that we got now, it can be concluded that those algorithms show different execution times to same array size. For example, for 10000 elements the slowest sorting algorithm was Bubble Sort with 0.484724 seconds because its complexity is O(n^2), but compared to Selection Sort and Insertion Sort which complexity are also O(n^2), they are faster resulting with 0.118193 seconds and 0.0845916 seconds respectively. On the other hand, Quick Sort was the fastest algorithm to sort all 10000 elements, with an execution time of 0.00120185 seconds, clearly its complexity is O(n log n), which mathematically means that it grows slower than O(n^2).

Now that we are in context is easiest to understand that it is really important and necessary to study the algorithmic complexity, due to this, we now can decide which algorithm must be used depending of the amounts of elements that we have as inputs or said in human terms, what we need to sort in any given context.

REFERENCES

[1] *Learn C++*. (2019). *Learncpp.com*. Retrieved 2 December 2019, from https://www.learncpp.com/

[2] *GeeksforGeeks | A computer science portal for geeks*. (2019). *GeeksforGeeks*. Retrieved 2 December 2019, from https://www.geeksforgeeks.org/

[3] (2019). *Freecodecamp.org*. Retrieved 2 December 2019, from https://www.freecodecamp.org/

Notes:

1. This is all an open source code.
2. You can check out our code at: https://github.com/xQQsHia/OurTakeOnBigONotation
3. Inside the folder that you will see in (2) there's a README that will guide you on the requirements that you need, and how to compile our program.