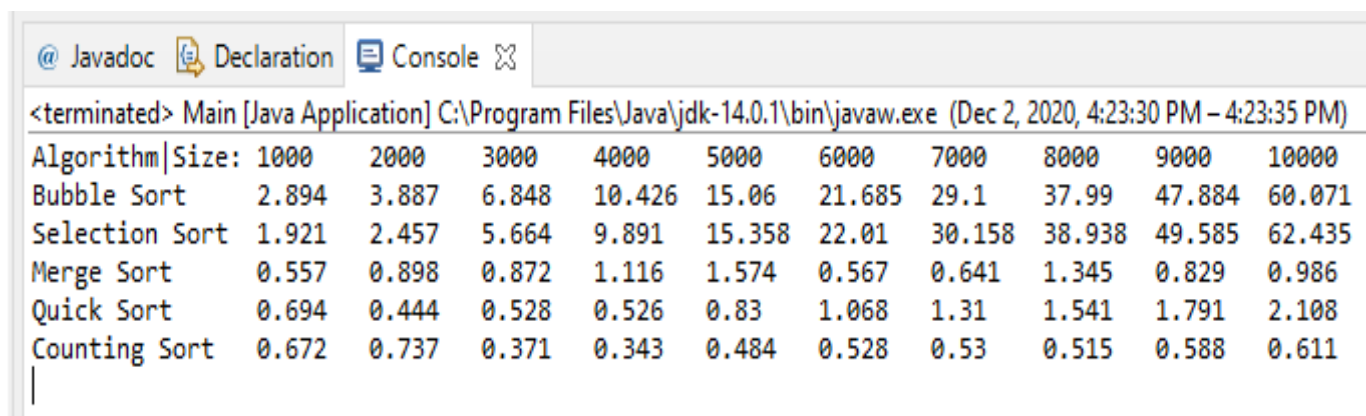


Benchmarking

The five chosen sorting algorithms were implemented in **java**. Their performances for different sample sizes (from **1000 to 10000**) of arrays having **randomly generated integers ranging from 0 to 99** were measured. Each algorithms performance was checked 10 times for each sample sizes and the **average time elapsed** during the **10 runs** was collected in **Milliseconds**. For measuring time, **System.nanoTime()** method was used at the **beginning and the end** of each run, then the **difference was calculated** and **converted** it into milliseconds.

Factors such as specification of the system (**CPU: Intel core i3 370M, Launch Date: Q3'10, No. of cores: 2, Clock frequency: 2.40GHz, CPU benchmark score: 2016, OS: windows 10 home 64 bit, RAM: 3 GB**) where the experiment performed, might have influenced the results.

The console output is as follows:



The screenshot shows a Java IDE console window with the following output:

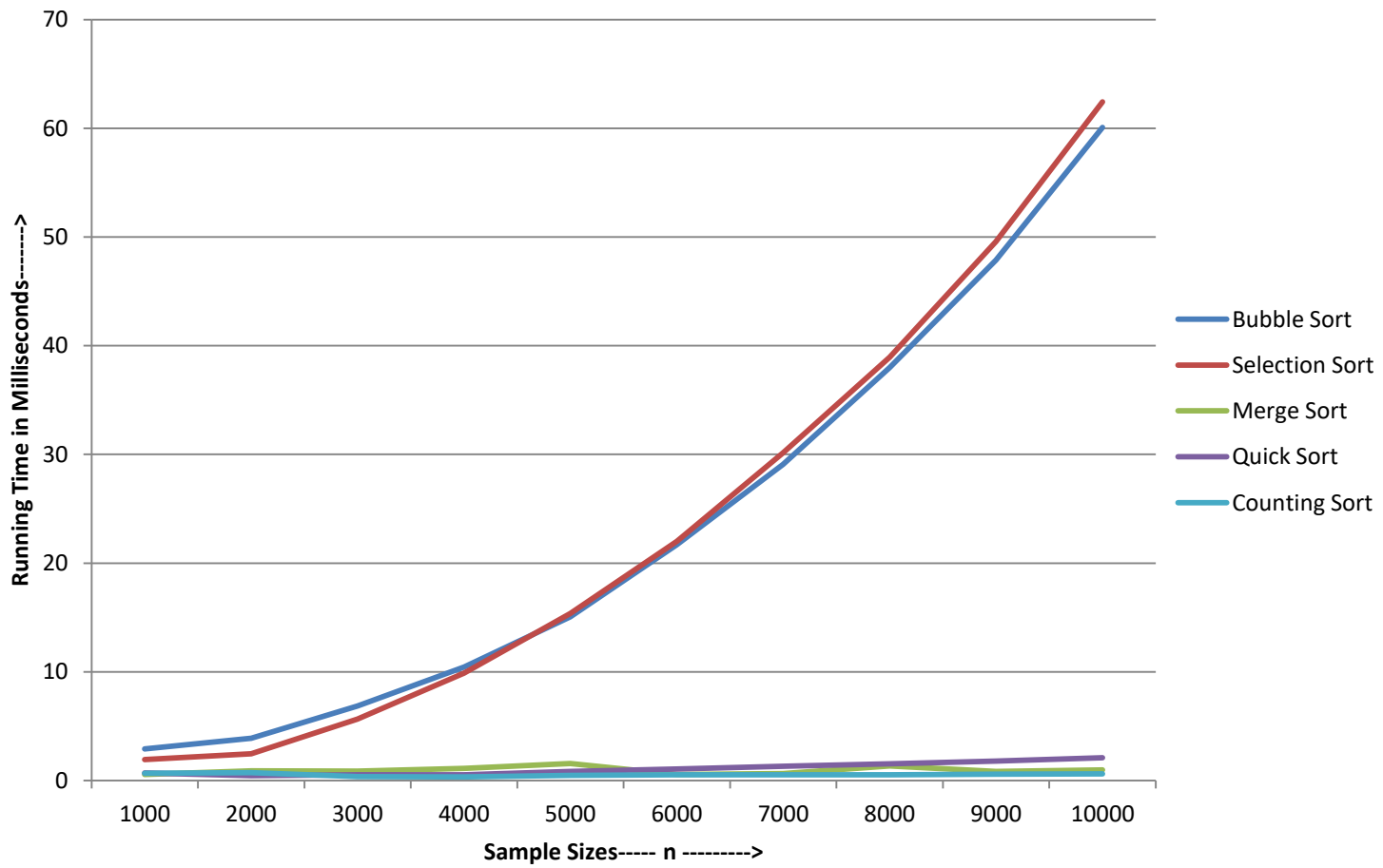
```
<terminated> Main [Java Application] C:\Program Files\Java\jdk-14.0.1\bin\javaw.exe (Dec 2, 2020, 4:23:30 PM - 4:23:35 PM)
Algorithm|Size: 1000    2000    3000    4000    5000    6000    7000    8000    9000    10000
Bubble Sort    2.894    3.887    6.848    10.426    15.06    21.685    29.1    37.99    47.884    60.071
Selection Sort  1.921    2.457    5.664    9.891    15.358    22.01    30.158    38.938    49.585    62.435
Merge Sort     0.557    0.898    0.872    1.116    1.574    0.567    0.641    1.345    0.829    0.986
Quick Sort     0.694    0.444    0.528    0.526    0.83     1.068    1.31     1.541    1.791    2.108
Counting Sort  0.672    0.737    0.371    0.343    0.484    0.528    0.53     0.515    0.588    0.611
```

Benchmarking Results of Five Sorting Algorithms in Milliseconds

For Sample Sizes 1000 to 10000

Sample Sizes ----->	1000	2000	3000	4000	5000	6000	7000	8000	9000	10000
Bubble Sort	2.894	3.887	6.848	10.426	15.06	21.685	29.1	37.99	47.884	60.071
Selection Sort	1.921	2.457	5.664	9.891	15.358	22.01	30.158	38.938	49.585	62.435
Merge Sort	0.557	0.898	0.872	1.116	1.574	0.567	0.641	1.345	0.829	0.986
Quick Sort	0.694	0.444	0.528	0.526	0.83	1.068	1.31	1.541	1.791	2.108
Counting Sort	0.672	0.737	0.371	0.343	0.484	0.528	0.53	0.515	0.588	0.611

Graphical Representation of Performance of Sorting Algorithms



Performance Analysis

Bubble Sort and Selection Sort

Bubble Sort's performance, as can be seen from the table and graphs above, is closed to $O(n^2)$, its time complexities in best, worst, and average cases. But it executed faster than **Selection Sort**, a similar category of choice for this project in sample sizes more than 4000. For sample sizes up to **4000**, **Selection Sort** is faster than **Bubble Sort**. It can be seen from the graph that **Selection Sort** also performed more or less similar to $O(n^2)$.

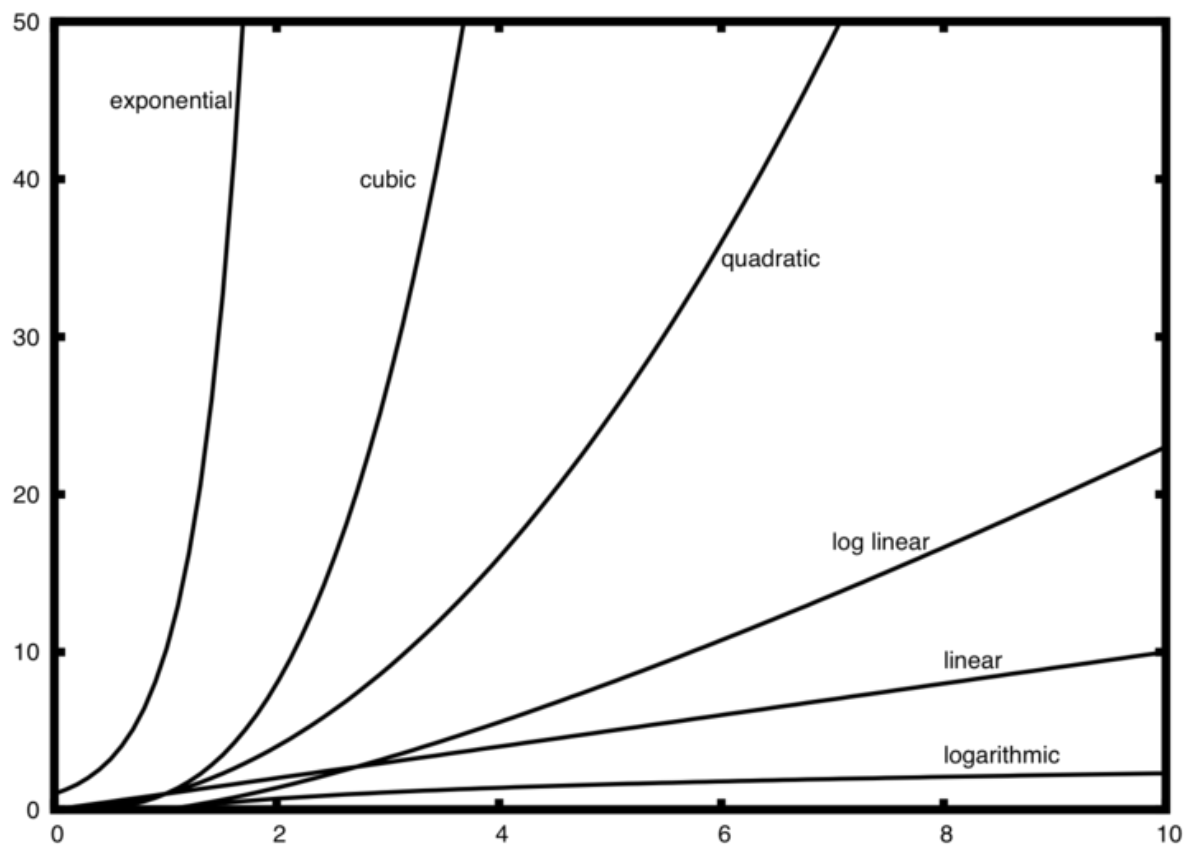
Merge Sort, Quick Sort, and Counting Sort

Among these three efficient algorithms the result is as expected; where **Counting Sort** is the fastest, more precisely, for the large sample sizes. Its graph is quite similar to $O(n + k)$. For small sample sizes **Merge Sort** and **Quick Sort** are faster than **Counting Sort** as per the data given above. While analysing the figures it can be found that **Quick Sort** is the slowest among these three, though its performance graph is almost identical to $O(n \log n)$. A similar graph is resulted for **Merge Sort** too. (Please refer the chart in next page).

To Conclude:

Counting Sort is found to be the most efficient. **Merge Sort's** efficiency is also closed to **Counting Sort**. But among the **simple algorithms**, **Bubble Sort** outperformed **Selection Sort** in this benchmarking test, when the sample sizes are greater than 4000.

Big-O Complexity Chart



(Source:

<http://interactivepython.org/lpomz/courselib/static/pythonds/AlgorithmAnalysis/BigONotation.htm>

!)