Algorithmics

Davide Gamba - ID: 30426243 - email: dg3g18@soton.ac.uk

Tutorial 1 - Measuring Time Complexity

The first part of this report focuses on measuring the time complexity of different array-sorting algorithms, namely insertion sort, Quicksort and shell sorting.
The second part deals with the measurement of time complexity of a Graph-Colouring Solver.
The third and last part, compares three different sorting algorithms

---

1. Hardware

The specifics of the computer used to run the programs that led to the generation of this report's data, are the following:

- **Model:** MacBook Pro (15-inch, Mid 2012)
- **Processor:** 2.6 GHz Intel Core i7
- **Memory:** 8 GB 1600 MHz DDR3

2. Measuring the time complexity of sort

The first program that was run was TestSort.java, which generates an array of size N and fills it with random values. After that, the values inside the array are cloned in three separate arrays, the program then runs each of the resulting arrays through one of the three available methods.

Each of the methods implements a different algorithm that is used to sort the values of the array, from the smallest to the largest. The first method uses insertion sort, the second one uses shell sort and the third one uses Quicksort.

After the execution of each method, the time that it took to sort the array is recorded and printed to the user.

Once that all of the three methods have been executed, the program prints

the values of each sorted array in a table-like manner to the user.

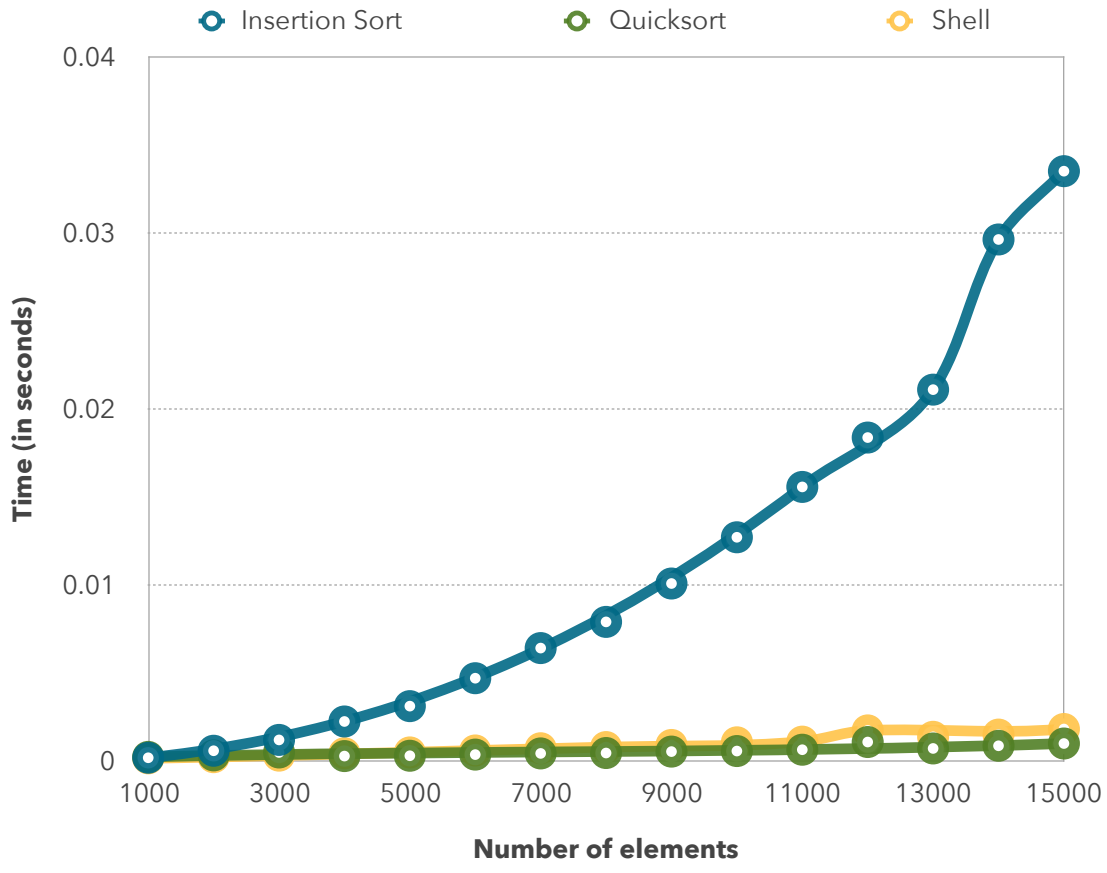
Measuring the time complexity

Complying with the specifics of the Tutorial, TestSort.java was modified in order to collect data on the time complexity of the three sort algorithms.

Three different arrays were declared in order to record the data on time for each algorithm. The size of this arrays was determined by a variable "timesExecuted" that was set to 100.

The instructions inside the main method were enclosed in a for loop that ran for "timesExecuted" times, each time the program was executed by the console.

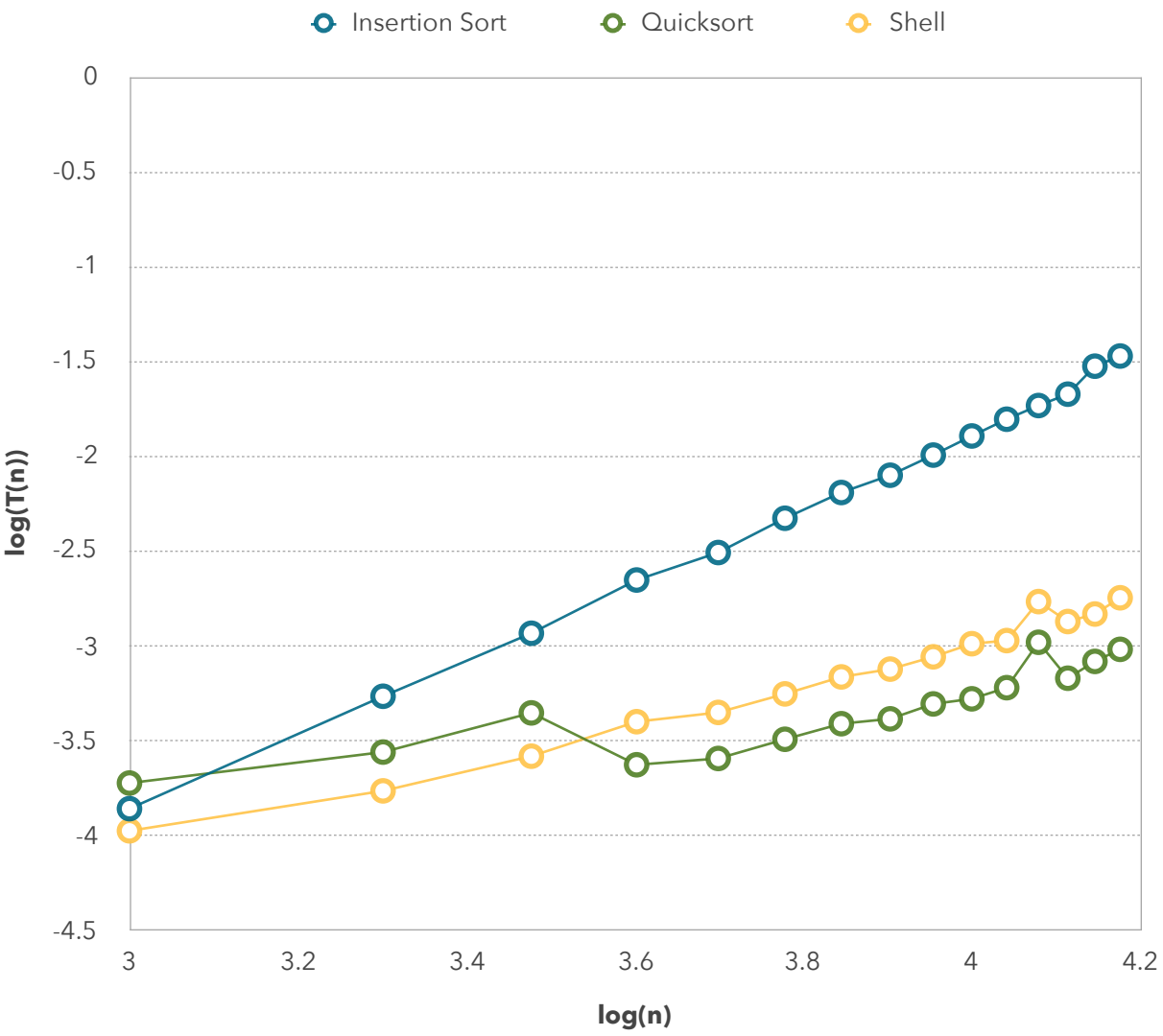
At the end of the for loop, the median value of the time taken for the execution of the sort algorithms was recorded in the graph.



By observing the graph we can see that the time taken by Insertion Sort grows quadratically as the number of elements in the array increases.

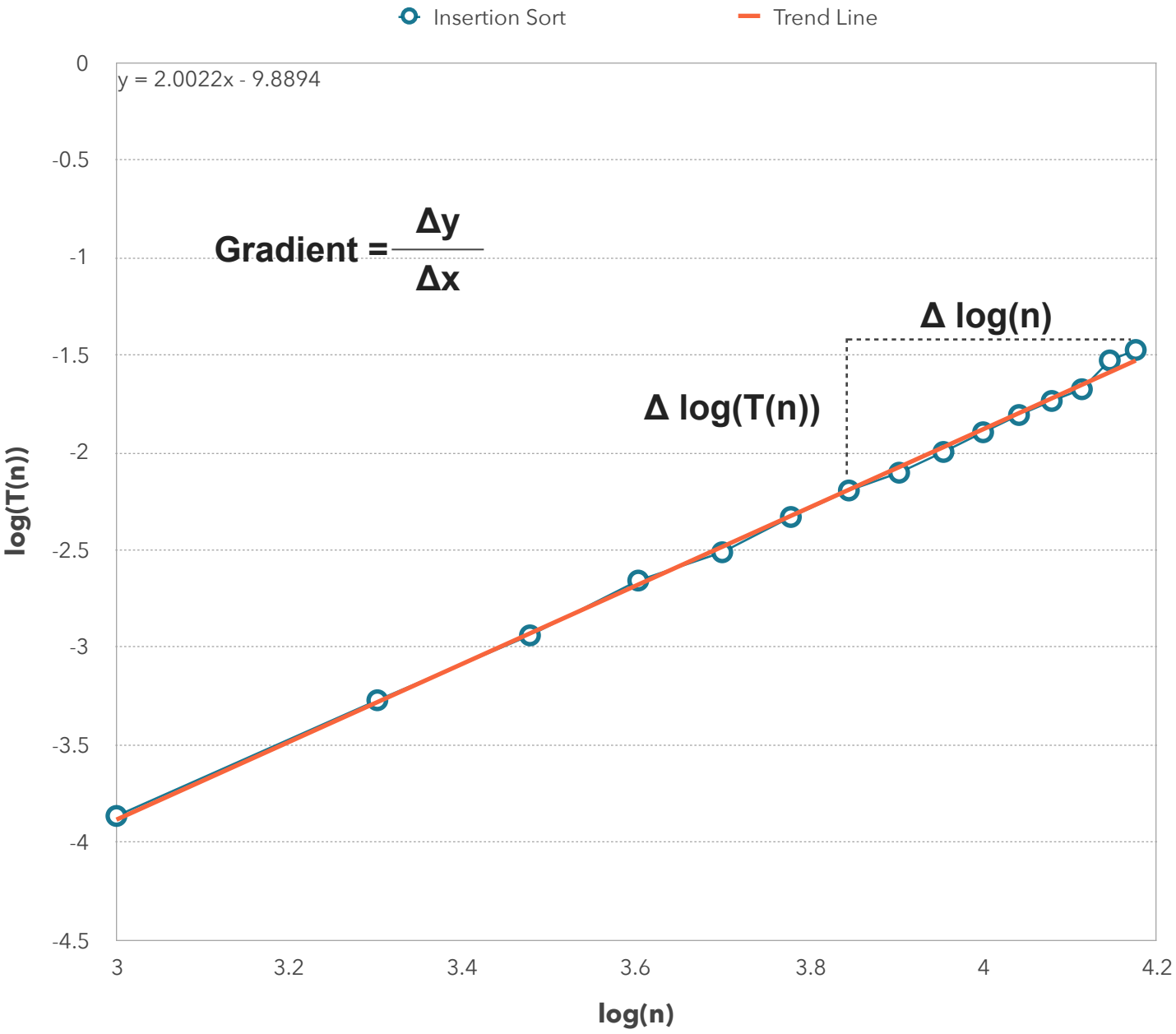
Which means that it takes time roughly proportional to n^2. This means that even though Quicksort and Shell Sort take linear time (O(n)) to be executed, the whole program will run in quadratic time, $O(n^2)$.

Since the program runs in $O(n^2)$, then we find the logarithm of the run time grows linearly with the logarithm of size of the input. We can therefore replot the graph as a log-log graph.



At this point, we can estimate the time complexity of Insertion sort by calculating the gradient (or slope) of the function.

Since the function is not perfectly linear, an average of the gradient between each point is necessary to compute an adequate estimate of the time complexity of Insertion Sort.



The resulting gradient average $\tilde{G}$ is calculated with by using the following:

$$G_i = \frac{\Delta y_i}{\Delta x_i} = \frac{y_{i+1} - y_i}{x_{i+1} - x_i}$$

$$\tilde{G} = \frac{1}{n} \sum_{i=1}^{n-1} G_i$$

Which in the specific case resulted in $\tilde{G}$ being:

$$\tilde{G} = 2.13827040510465 \approx 2.0$$

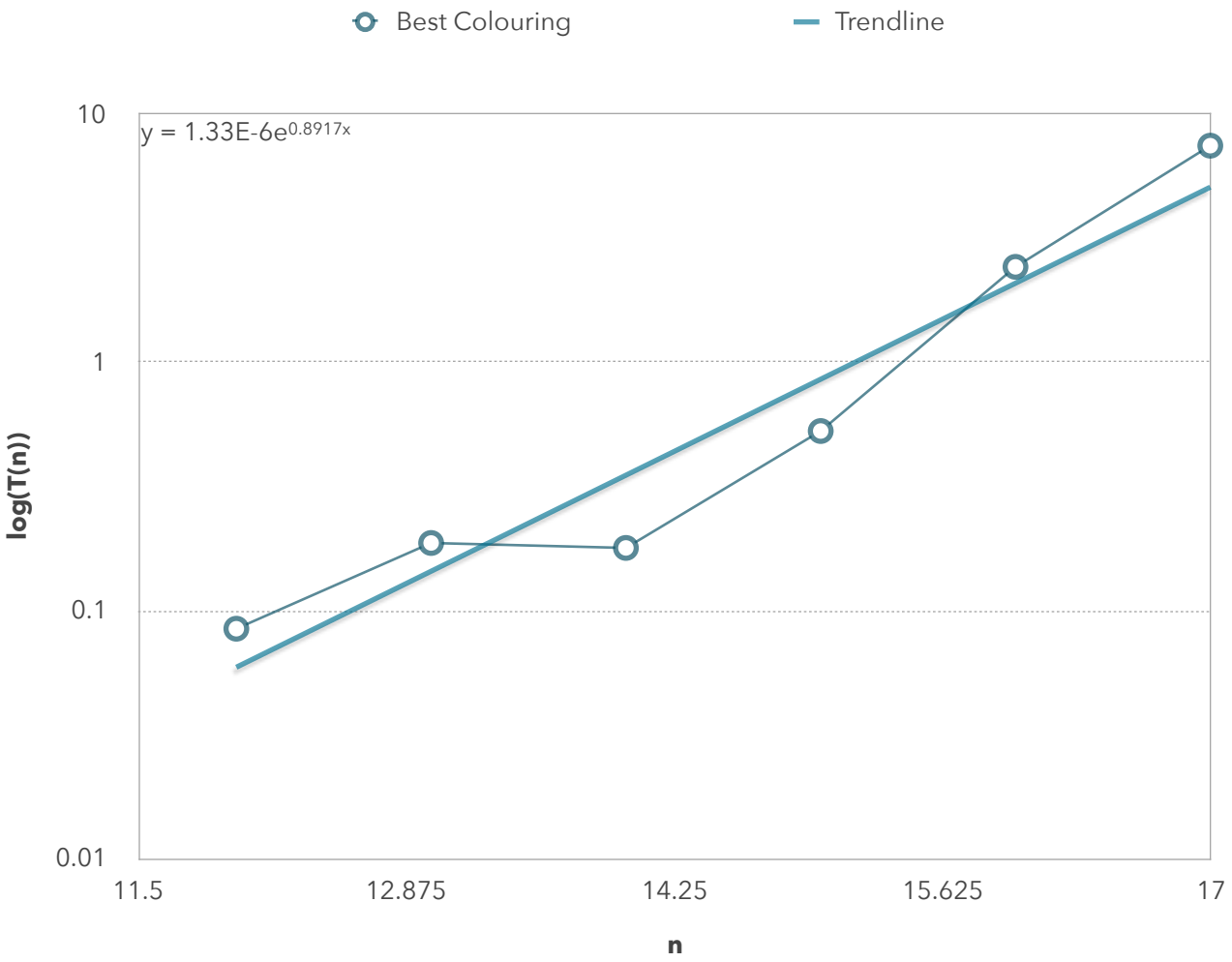
The time complexity of Insertion Sort is therefore quadratic: **O(n²)**

2. Measuring Complexity of Graph-Colouring Solver

After copying, compiling and executing the three java classes given in the Tutorial text: Graph.java, GraphDisplay.java and Colouring.java, a new class was created called Time.java, the code of which is shown in the picture below:

```java
/**
 * Is used to calculate the time that the program took
 * to execute a certain piece of code.
 */
public class Time {


    double timeTaken = 0.0;

    /**
     * The parameters use numbers generated by System.nanoTime()
     * to calculate the time that has passed
     * since the beginning of the execution of the code
     * @param startTime the starting time of the execution of the code
     * @param endTime the time in which the code has ended its execution
     * @return the time it took to execute the code (in seconds)
     */
    public double measureTime(long startTime, long endTime){

        timeTaken = (endTime - startTime) / 1000000000.0;

        return timeTaken;
    }
}
```

The time measurements relative to the execution of the algorithm with values of nodes from 12 and 17 were then recorded and are shown in the graph below.



log(T(n)) = 0.8917n + log(1.33E-6e)

The equation can be reduced to:

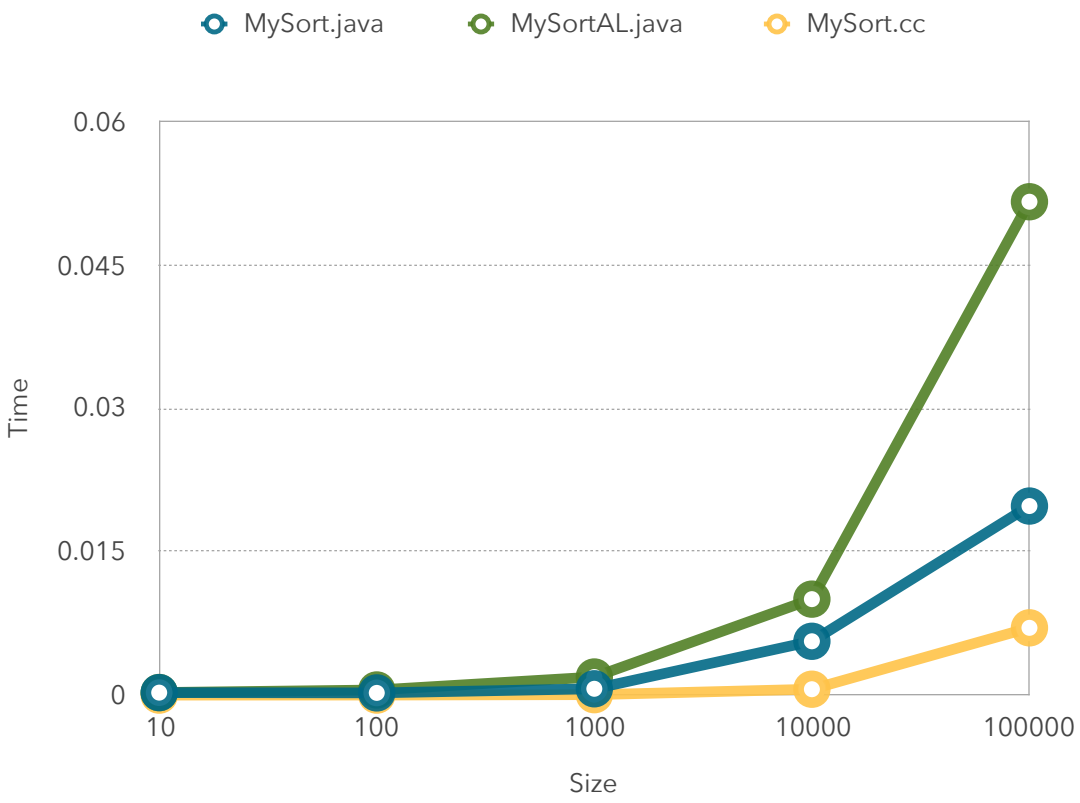
$y = 1.33E\text{-}6e^{0.8917n}$

From this equation we get that the time complexity of the algorithm is exponential and therefore runs in **$O(e^n)$**.

The difference between the sort and the graph coloring algorithms, is that sort runs in quadratic time and is therefore faster than graph coloring which runs in exponential time.

3. Java and C++

Three programs: MySort.java, MySortAL.java and MySort.cc were run, each containing a different sorting algorithm. Their runtimes were recorded for arrays of sizes from 10 to 10^5 and plotted in the graph shown below.



As we can observe, the most efficient algorithm was MySort.cc, with MySort.java being the second. MySortAL.java, however, was the worst performing algorithm. While the graph only shows the time taken by the algorithms for arrays with a maximum of 10^5 elements, all three of the algorithms was tested also with arrays of size 10^7 and 10^8.

More specifically, it was actually impossible to calculate the runtime of MySortAL.java for an array of size 10^8, as the program would quickly run out of memory, before finishing the sorting. Even by increasing the allocated memory to 1GB, the algorithm was still saturating the RAM.