Krasimir Marinov

Algorithms Tutorial 1

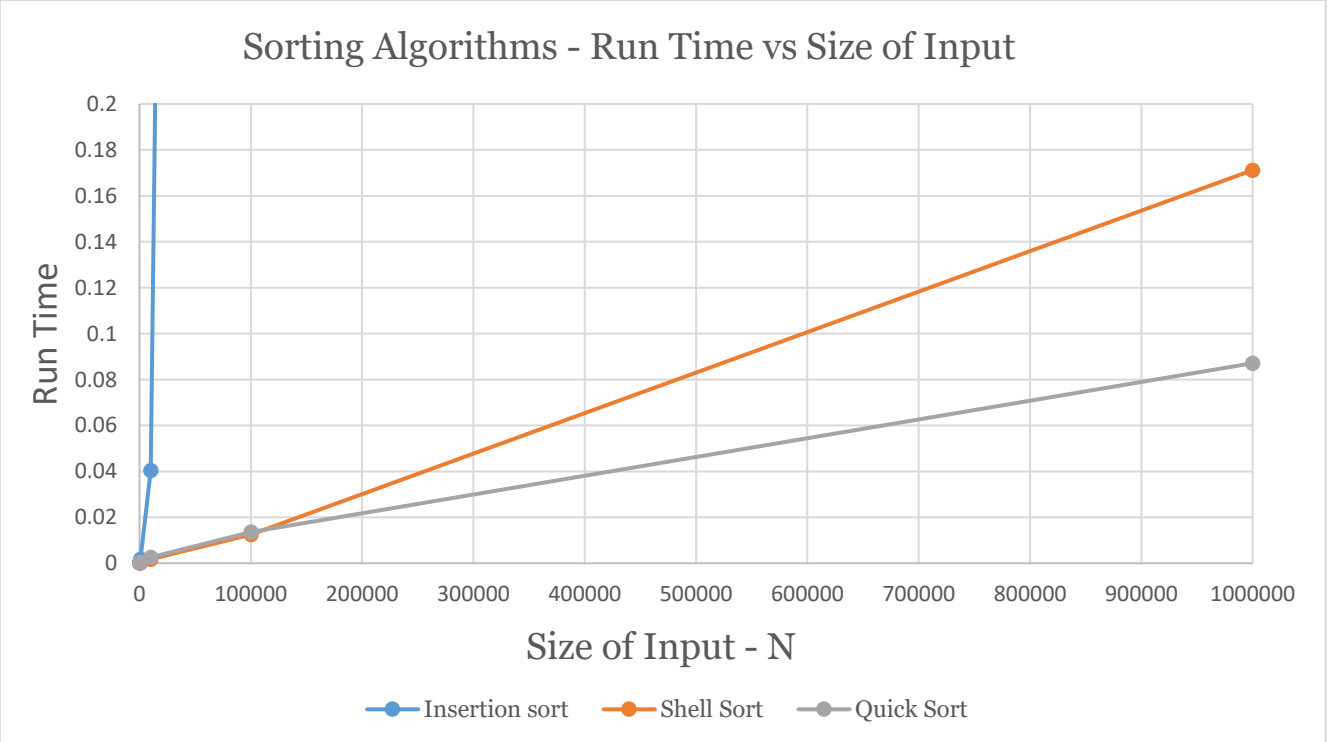
Measuring Time Complexity

Exercise 1

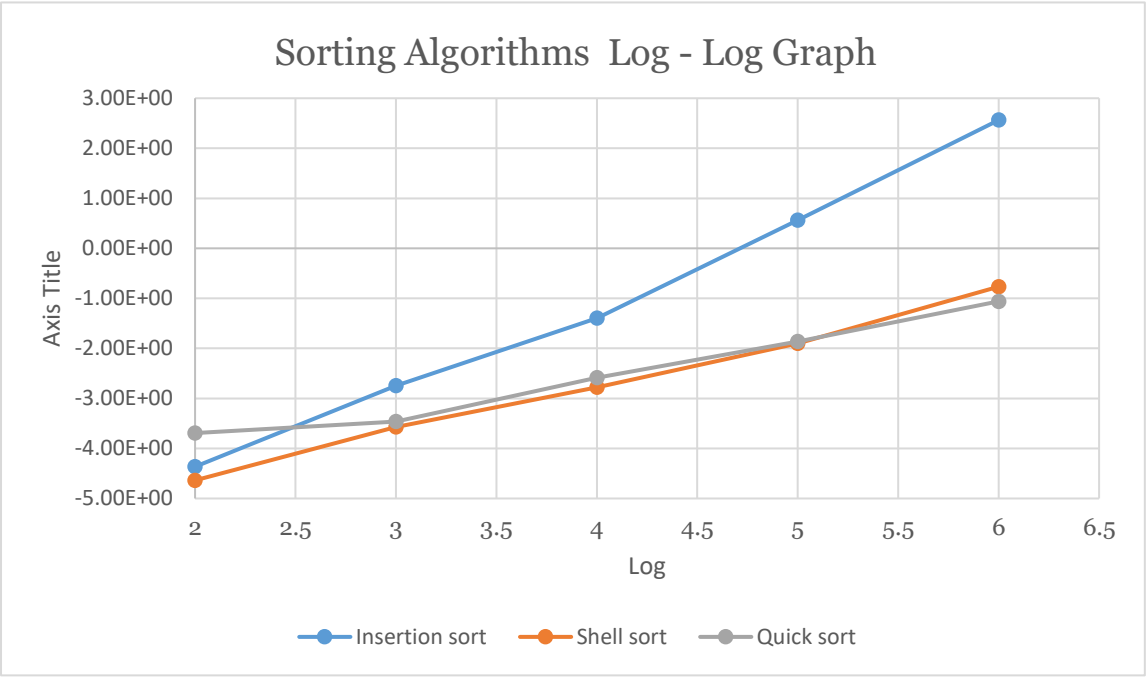
The test are done on Intel® Core™ i7-7700 CPU 3.60GHz x64- based processor with 16.0 GB installed RAM.

1. 'TestSort.java' compiles and runs successfully.
2. The program 'TestSort.java' measures' the run time of three different sorting algorithms – Insertion sort, Shell sort and Quick sort on an array with length N of random doubles.
3. I have measured the running time of each sorting algorithm 3 times on arrays from sizes of 10^2 to 10^6 and in my table I have taken the median of the three values for each algorithm.

	Insertion sort	Shell Sort	Quick Sort
Size of input	Running time		
100	0.000043236	0.00002304	0.000203093
1000	0.001786881	0.000268516	0.000344747
10000	0.04033907	0.001677654	0.002611485
100000	3.642815898	0.012493941	0.013614652
1000000	366.1900621	0.171179	0.087113696



4. We take the logarithm of the run time and the logarithm of size of input for the previously collected data and replot our graph as a log- log graph. We see that the logarithm of the run time of the sorting algorithms grows linearly with the logarithm of size of the input- n.



5. The gradient of the log-log plot for insertion sort is calculated by the formula $\frac{\Delta y}{\Delta x}$. When, calculated we see that it is approximately equal to 2.

$$gradient = \frac{\Delta y}{\Delta x} = \frac{\log_{10} T(10^6) - \log_{10} T(10^5)}{\log_{10} 10^6 - \log_{10} 10^5}$$
$$= \frac{2.56370655400465 - 0.561437223258644}{1}$$
$$= 2.002269330746$$

The gradient tells us the rate of growth of the time complexity. Therefore, the time complexity for insertion sort is $T(n) = an^c = an^2 \in O(n^2)$ where c is our gradient.

From the graph we see that the Quick Sort is faster than the Shell Sort and the Insertion Sort. However, for $n<10^5$ Shell Sort is faster than Quick Sort. Which also could be deducted from the fact that the Quick Sort has an average performance $O(n \log n)$ which is the best performance for the Shell Sort. Insertion sort has the worst average performance with $O(n^2)$

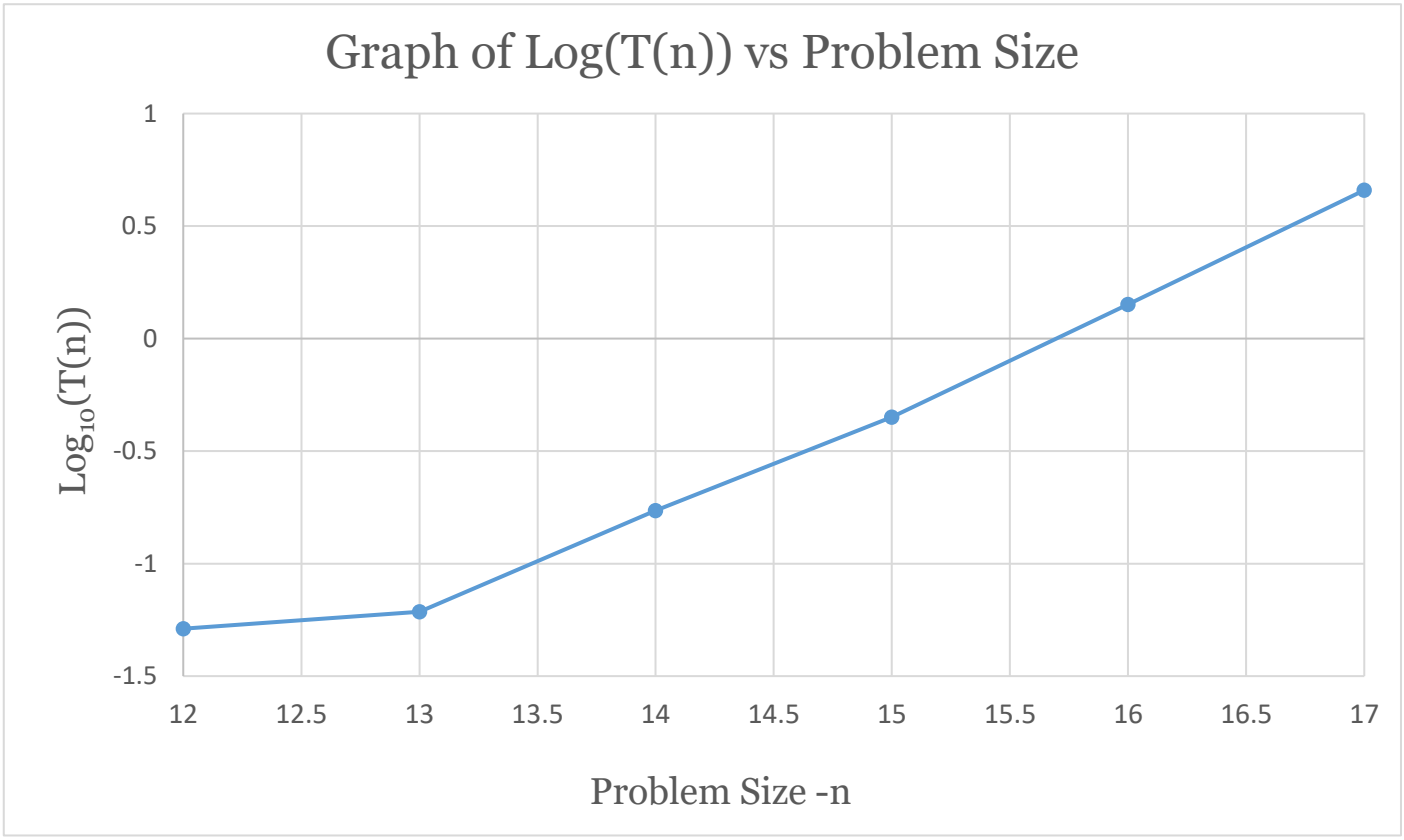
Exercise 2

1. The programs 'Graph.java', 'GraphDisplay.java' and 'Colouring.java' compile and run successfully. I write a new class called 'GraphTester.java' to measure the run time of the bestColouring() method for problems of size 12 to 17

```
/** class GraphTester used to measure the run time for problems of size 12 to 17 */
public class GraphTester {

	public static void main(String[] args) {

		// We test the run time for Graphs with sizes of 12 to 17
		for(int i=12;i<=17;i++) {

			// We declare and initialise a new graph with size i
			Graph graph = new Graph(i, 0.5);

			// time_prev used to measure the time before the call of the method
			long time_prev = System.nanoTime();
			Colouring colouring = graph.bestColouring(3);

			// double time measures the run time calculates the difference
between the current time and time_prev in seconds
			double time = (System.nanoTime()-time_prev)/1000000000.0;

			graph.show(colouring);

			// We print the run time
			System.out.println(i+" "+time);

			// We assign null to our graph object
			graph=null;

			// We call the garbage collector to collect in order every test to
start under the same circumstances
			System.gc();

		}


	}


}
```

2. After running the algorithm 10 times we take the median of the logarithm of the run time for a graph with size n. With the collected data we plot a graph of the logarithm of the run times for n versus problem size.

N	$Log_{10}(T(n))$
12	-1.2892
13	-1.21445
14	-0.76385
15	-0.34881
16	0.151823
17	0.659289

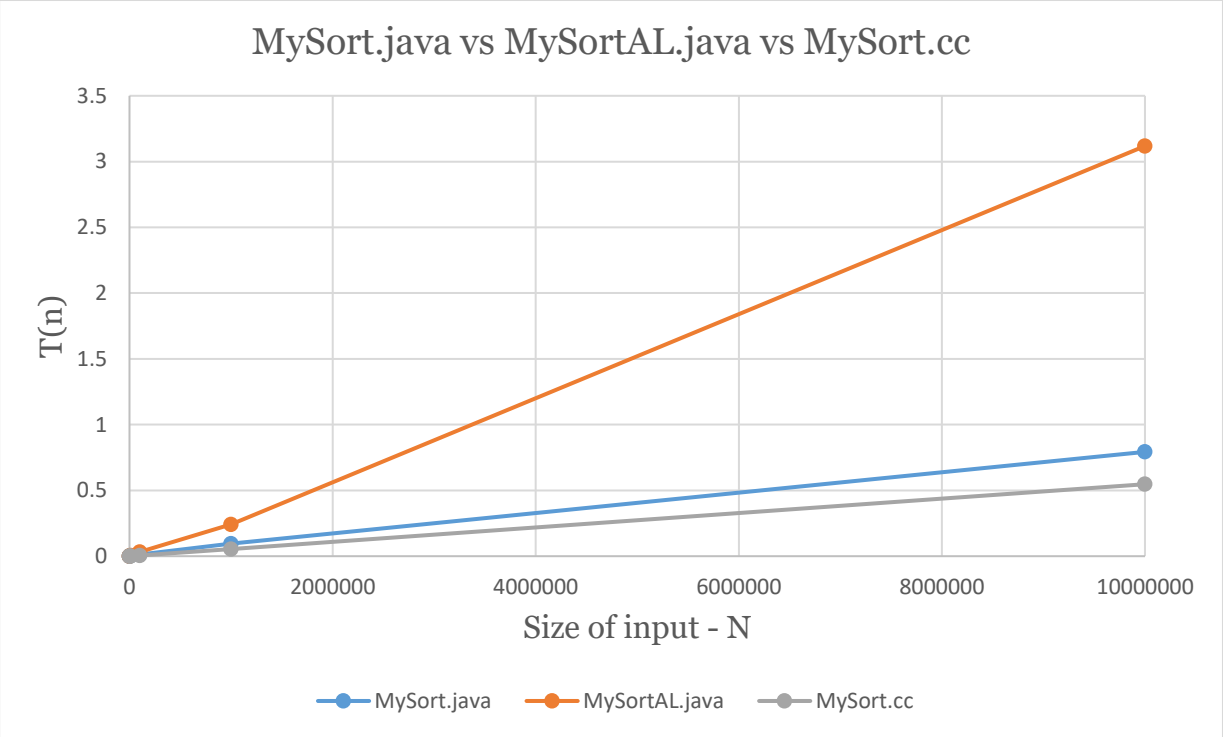


3. We plot the log–linear graph and we could see a straight line. From this, we could assume that we have an exponential function due to the fact that exponential functions plot on log-linear graphs as straight lines. The slope of the line or the gradient gives us the exponential value in the equation $\log(T(n)) = cn + \log(a)$. Therefore, we measure the gradient which is $gradient = \frac{\Delta y}{\Delta x} = \frac{\log_{10} T(17) - \log_{10} T(16)}{1} = \frac{0.659289434651352 - 0.151822570167519}{1} = 0.507466864483833 \approx 0.50.$ Thus, we could substitute c in $T(n) = a * 10^{cn}$ resulting in $T(n) = a * 10^{0.5n} \approx a * 3.16^n$. Then we could say that the rate of growth of the time complexity is exponential $T(n) \in O(2^n)$. Because it is exponential it grows faster than the sorting algorithms - Insertion Sort, Shell Sort and Quick Sort.

Exercise 3

1. We compile the program 'MySort.cc' g++ -O4 -o MySort MySort.cc and then we compile and run the programs 'MySort.java' and 'MySortAL.java'.
2. I measure the run time of each program 3 times for input of size of 10^2 to 10^7 and I take the median of the 3 attempts.

	MySort.java	MySortAL.java	MySort.cc
100	0.00012544	2.84E-04	0.00000256
1000	3.31E-04	9.68E-04	3.36E-05
10000	2.68E-03	5.63E-03	4.35E-04
100000	1.38E-02	3.32E-02	5.22E-03
1000000	9.64E-02	2.42E-01	5.49E-02
10000000	7.93E-01	3.12E+00	5.47E-01



3. From the graph we could see that MySortAL.java has the highest run time while MySort.cc is the fastest program and MySort.java is between them. MySortAL.java has the highest run time because of the access time for Array Lists which is O(n) compared to the access time for Arrays which is O(1). MySort.cc is the fastest because C/C++ produces native code to run on a particular machine while Java programs run on top of a virtual machine. However, the difference is really small.