COMP1201
Algorithmics Assignment I

Alberto Tamajo

Student ID: 30696844
Email address: at2n19@soton.ac.uk

February 2020
Electronics and Computer Science Department
University of Southampton

1 Question 1

(a) In order to measure the average runtime of the Insertion sort, Shell sort and Quick sort on different sizes of array I nested two for loops. The outer loop increases the size of a random generated array of doubles by 1 from 2 to 10,000 while the inner loop allows to measure the run time of the three sorting algorithms for the same input size 20 times. The average run time of each algorithm for each input size is computed by summing the 20 run time measures for the same input size and dividing this sum by 20. The average run time values are exported into a csv file so that to plot these values against their input sizes very easily by using Excel.The Java code is in the Appendix section.

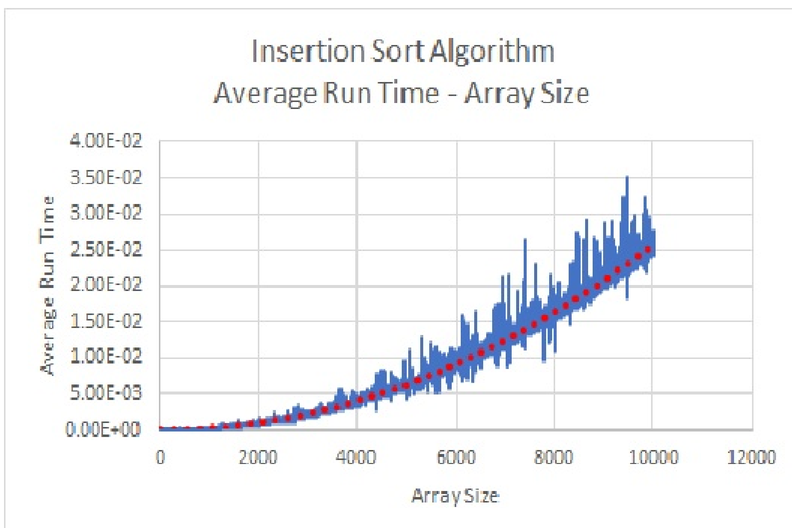

Figure 1: Insertion Sort Average Run Time Plot

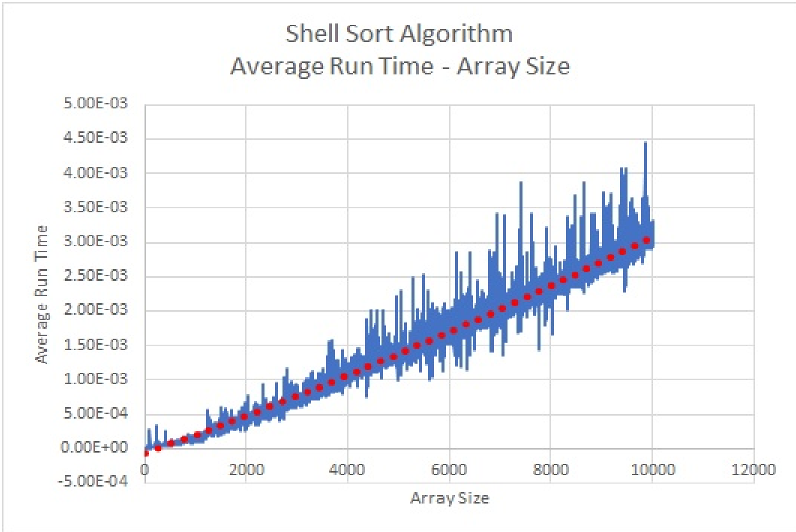

Figure 2: Shell Sort Average Run Time Plot

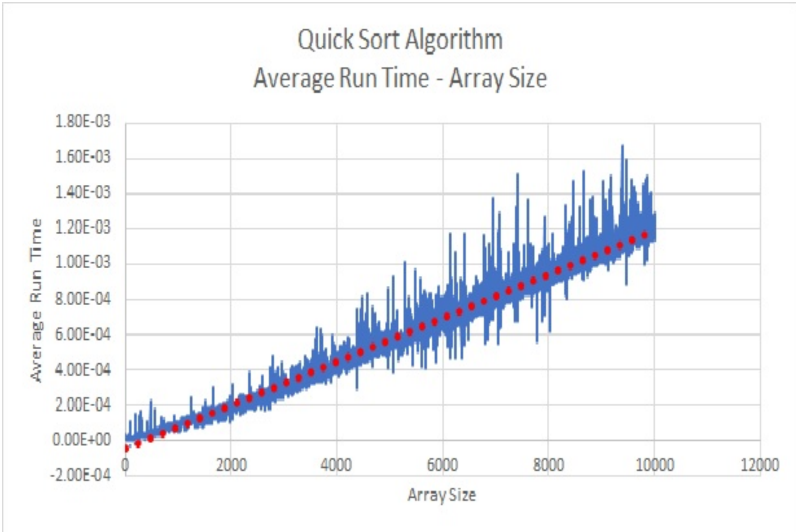

Figure 3: Quick Sort Average Run Time Plot

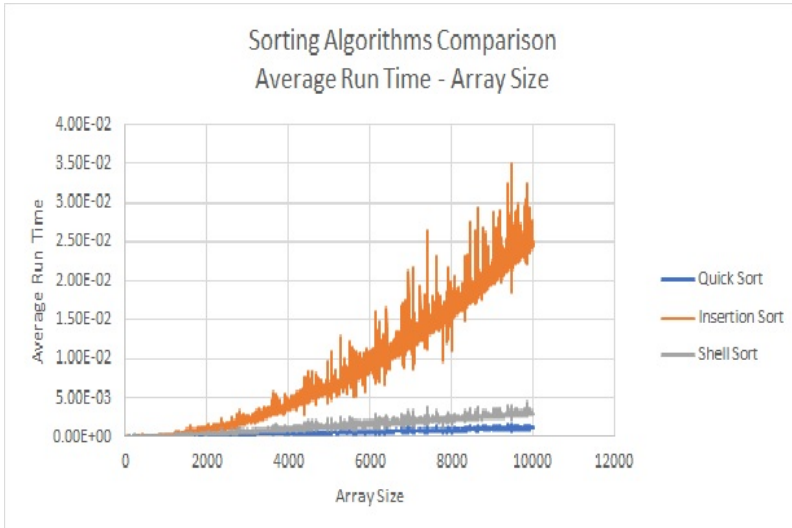

Figure 4: Insertion Sort, Shell Sort, Quick Sort comparison plot

(b) The log-log plots are below.

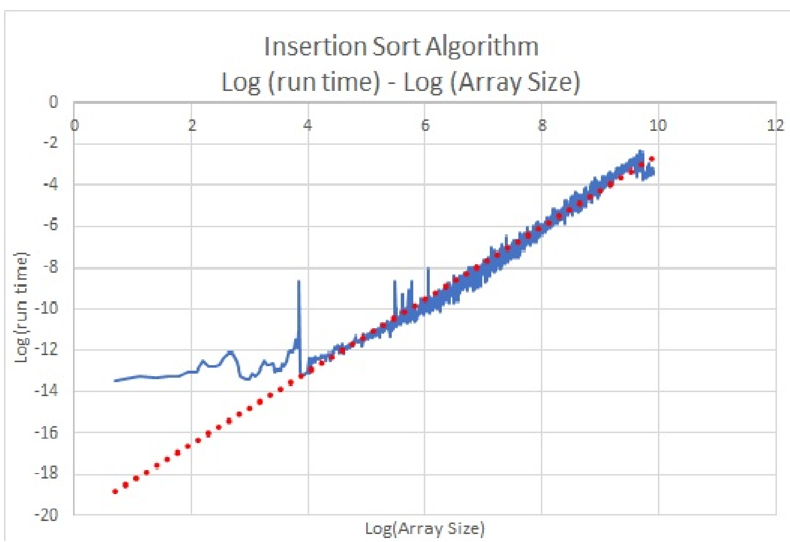

Figure 5: Insertion Sort Log(run time)-Log(Array Size) plot

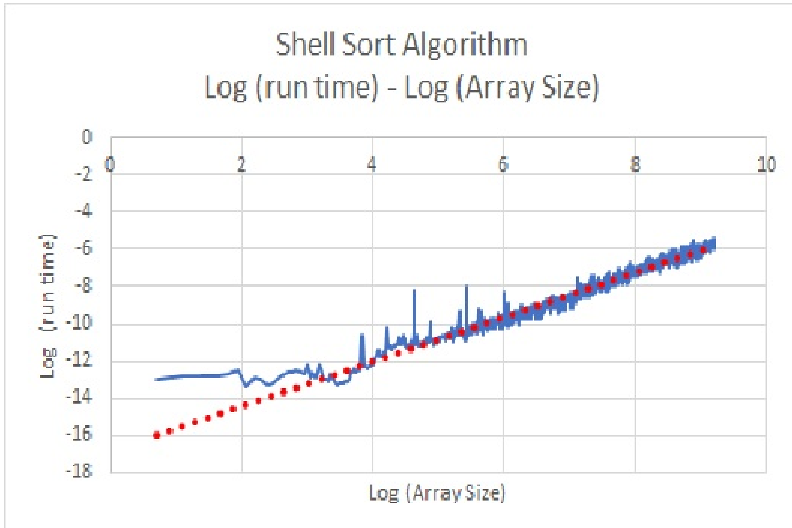

Figure 6: Shell Sort Log(run time)-Log(Array Size) plot

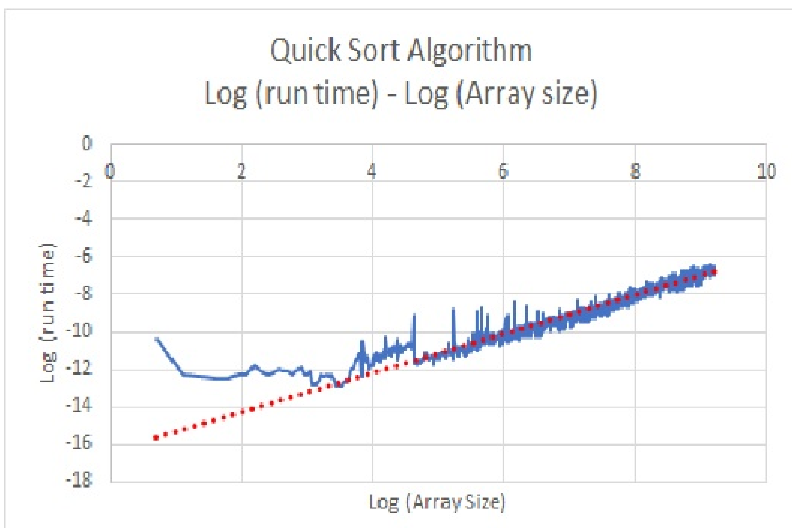

Figure 7: Quick Sort Log(run time)-Log(Array Size) plot

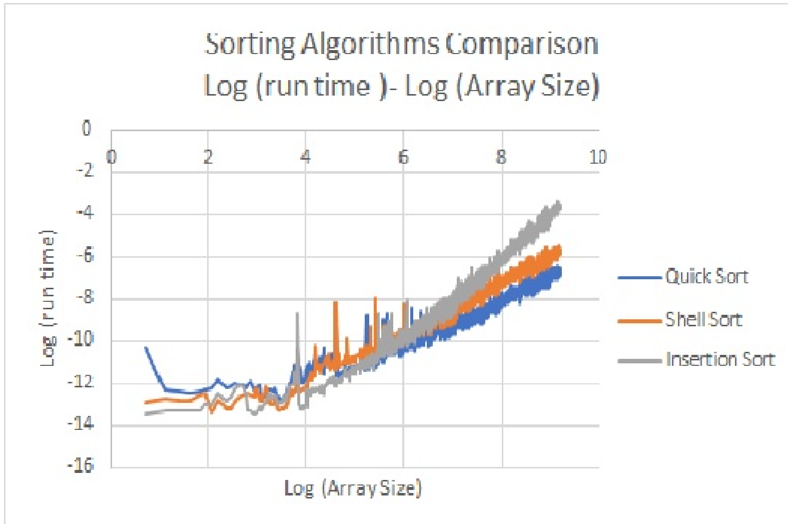

Figure 8: Insertion Sort, Shell Sort, Quick Sort Log(run time)-Log(Array Size) plot

(c) From the log-log scatter plot of the average-case time complexity of insertion sort (Fig.5) is noticeable that as the values in the x-axis (log of the input size) get larger, the graph gets closer and closer to a straight line. This leads to the fact that it is possible to describe approximately this plot with the equation $y = mx + c$ where:
$y = log(runtime)$
$x = log(inputsize)$

By exploting the fact that if a log-log plot outputs a straight line then there exists a polynomial relationship between two variables, it can be claimed that the run time of the Insertion sort algorithm is in a polynomial relationship with the input size. The degree of this polynomial relationship is equal to the slope of the straight line represented by the log-log plot.

In order to find 'm' (the slope of the straight line), I have picked 2 data points that lie almost at the end of the graph due to the fact that the larger a data point x-axis value is the more this data point is closer to the asymptotic line. I computed the slope of the line joining them and the slope turns out to be equal to 1.95. In order to gain a better understanding of the slope of this line, I also used a regression algorithm so that to find the line that best fits the data points inside the scatter plot. It turns out that the gradient of this line is equal to 1.97 which approximately is equal to 2.

Thus, since the log-log plot of the insertion sort run time against the input size outputs a straight line whose gradient is approximately equal to 2 then there exists a quadratic relationship between the two variables.
Let $y =$ run time of Insertion sort and $x =$ input size then $y = ax^2 + bx + c = \Theta(n^2)$.
In conclusion, the average run time of insertion sort is equal to $\Theta(n^2)$.

(d) Since the average-case time complexity of Insertion Sort is $\Theta(n^2)$ we can estimate that the average running time of insertion sort on an array of size 10^{10} is equal to:

$$\begin{aligned} T(10^{10}) &= \mathbf{a}10^{10^2} + \mathbf{b}10^{10} + \mathbf{c} \\ &= \mathbf{a}100,000,000,000,000,000,000 + \mathbf{b}10,000,000,000 + \mathbf{c} \end{aligned}$$

where $a > 0$ and $b, c \in \mathbb{R}$

In order to measure the run time of Insertion sort on an array of size 10^{10} in the specific context of my laptop's hardware I exploited the fact that since $T(n) = \Theta(n^2)$ then $T(n) \approx an^2$.
Consequently, it follows that $a \approx \frac{T(n)}{n^2}$.
After having picked the last data point of my plot, I used the above formula to compute a which is equal to $2.4921859e - 10$ and thus in my specific laptop the run time of Insertion sort is approximately equal to:

$T(n) \approx 2.4921859e - 10n^2$
Thus,

$$\begin{aligned} T(10^{10}) &\approx 2.4921859e - 10 * 10^{20} \\ &\approx 24921858729 \end{aligned}$$

The above value is expressed in seconds and so by converting it into years it turns out to be equal to 790 years.

2 Question 2

(a) In order to measure the average run time of the Graph Colouring algorithm for problems of size 12 to 17 I nested two for loops. The outer loops increases the number of nodes in the graph by 1 from 12 to 17 while the inner loop allows to measures the run time of the algorithm for the same input size 200 times. The average run time for each input size is computed by summing the 200 run time measures for the same input size and diving this sum by 200. The average run time values are exported in a csv file so that to plot these values against their input sizes very easily by using Excel. The Java code is in the Appendix section.

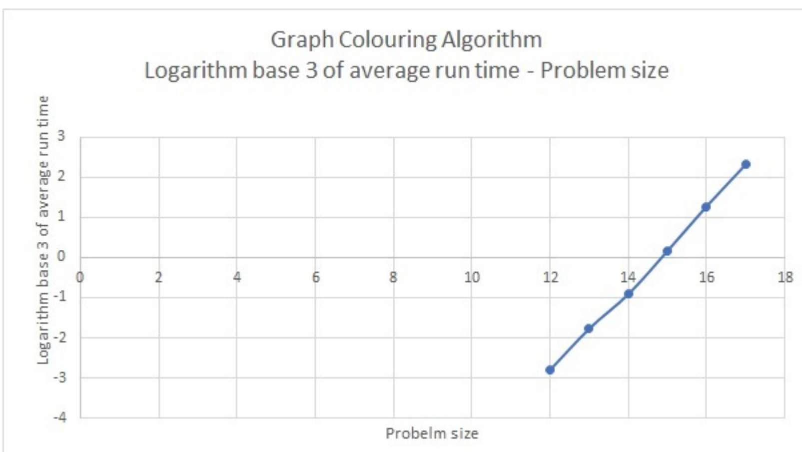

Figure 9: Graph Colouring average run time

(b) From the log-linear scatter plot of the average-case time complexity of the Graph Colouring Algorithm is noticeable that there exists roughly a linear relationship between the $log(runtime)$ and the input size. This leads to the fact that it is possible to describe approximately this plot with the equation $y = mx + c$ where:

$y = log(runtime)$
$x = inputsize$

By exploiting the fact that if a log-linear plot outputs a straight line then there exists a exponential relationship between two variables, it can be claimed that the run time of the Graph Colouring algorithm is in an exponential relationship with the input size.

This is what I expected as the Graph Colouring algorithm provided in the assignment uses a brute force approach to show the best possible colouring. In other words, it tries all possible permutations of colours of the nodes. Since, the number of colours that the algorithm uses is 3 then it follows that the total number of permutations is equal to 3^n where n is the number of nodes of the graph.

Since I expected the base of the exponential to be equal to 3, the base of the logarithm that I used to plot the log-linear graph is 3 as well. In order to confirm my suspicion, I had to find out what the value of c in the exponential expression ab^{cn} is ($b = 3$ because I used log with base 3). It turns out, by exploiting the properties of log-linear graphs, that if the plot outputs a straight line then c is equal to m which is equal to the gradient of the straight line.

In order to find m (the slope of the straight line), I have picked the last 2 data points due to the fact that the larger a data point x-axis value is the more this data point is closer to the asymptotic line. I computed the slope of the line joining them and the slope turns out to be equal to 1.03. In order to gain a better understanding of the slope of this line, I also used a regression algorithm so that to find the line that best fits the data points inside the scatter plot. It turns out that the gradient of this line is equal to 1.01 which approximately is equal to 1.

Thus, since the log-linear plot of the insertion sort run time against the input size outputs a straight line whose gradient is approximately equal to 1 then this confirm my initial suspicion. Indeed, let $y =$ run time of the graph colouring algorithm and $x =$ input size then $y = a3^n = \Theta(3^n)$. In conclusion, the average run time of the graph colouring algorithm is equal to $\Theta(3^n)$.

3 Question 3

(i) The time complexity of inserting n elements with identical keys into an initially empty binary search tree described as in the text is equal to $\Theta(n^2)$.

Proof:

- Let a binary search tree contain $k_1, k_2, ..., k_n$ nodes such that $k_1.key = k_2.key = \ldots = k_n.key$

- Let a node m be added to this binary search tree such that $m.key = k_1.key = \ldots = k_n.key$

Since constants are irrelevant in asymptotic analysis:

- Let d be equal to the maximum number of instructions to check if $m.key < k_m.key$ or $m.key > km.key$ (k_m is the m^{th} node in the tree).
- Let e be equal to the number of instructions to eventually add m
- Let $T(0) = c$ where $T(0)$ indicates the number of instructions needed to add the first element (root) to the tree.

Since $k_1.key = k_2.key = \ldots = k_n.key$ then the binary tree looks like a right skewed binary search tree:

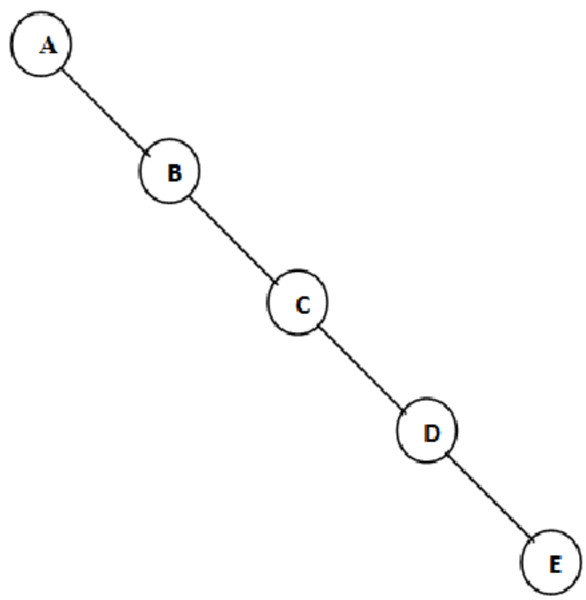

Figure 10: Right Skewed Binary Search Tree

This leads to the fact that in order to add the node m to the tree, the whole tree must be traversed.

Therefore, $T(n) = d + d + \ldots + e = dn + e = \Theta(n)$

It follows that since adding one node m to a Binary Search Tree where $m.key = k_1.key = \ldots = k_n.key$ is equal to $\Theta(n)$ then adding n elements with equal key to an empty binary search tree described as in the question text is equal to $n\Theta(n) = \Theta(n^2)$.

Proof:

- Let $Q(n)$ be the time it takes for adding n nodes with same key to an empty Binary Search Tree described as in the question text.
  It follows that:

$$
\begin{aligned}
Q(n) &= T(0) + T(1) + T(2) + \ldots + T(n-1) \\
&= c + (d+e) + (2d+e) + \ldots + [(n-1)d + e] \\
&= c + (n-1)e + [1 + 2 + 3 + \ldots + (n-1)]d \\
&= [(n-1)\frac{n}{2}]d + (n-1)e + c \\
&= [\frac{n^2-n}{2}]d + (n-1)e + c \\
&= \Theta(n^2)
\end{aligned}
$$

(ii) The time complexity of inserting n identical elements into an initially empty binary search tree, with this modified insert operation, is equal to $\Theta(nlog(n))$.

Proof:

- Let a binary search tree contain $k_1, k_2, ..., k_n$ nodes such that $k_1.key = k_2.key = \ldots = k_n.key$
- Let a node m be added to this binary search tree such that $m.key = k_1.key = \ldots = k_n.key$
- Let $T(n)$ be the time it takes to add m to such a binary search tree

It turns out that the way this binary search tree is defined leads to the fact that when adding n elements with same key, this binary search tree is balanced. Specifically, it is noticeable that given that x is a node of this binary search tree then the left subtree of x differs by at most 1 from the right subtree of x. Additionally, not only the left subtree of x and right subtree of x differ by at most 1 but also the elements of this binary search tree are added in such a way that the height is always equal to $\lceil \log_2(n+1) \rceil$ and the number of comparisons to add an element is equal to $\lfloor \log_2(n+1) \rfloor$. Since the ceiling operator and constants are irrelevant in asymptotic analysis then the height of this binary search tree is $\Theta(logn)$. Adding an element to such a binary search tree involves $\lfloor \log_2(n+1) \rfloor$ number of comparisons and so $T(n) = \Theta(logn)$. Consequently, adding n elements with same key to this binary search tree is equal to $n\Theta(logn) = \Theta(nlogn)$.

Proof:

Let $Q(n)$ be the time it takes to add n elements with same key to a binary search tree defined as in the text.

It follows that:

$$
\begin{aligned}
Q(n) &= T(0) + T(1) + T(2) + \ldots + T(n-1) \\
&= log0 + log1 + log2 + \ldots\ldots.. + log(n-1) \\
&= log((n-1)!) \\
&= \Theta((n-1)log(n-1)) \\
&= \Theta(nlog(n))
\end{aligned}
$$

(iii) The time complexity of inserting n identical elements into an initially empty binary search tree, with this new implementation, is equal to $\Theta(n)$.

The time complexity of adding one element to a Linked-List is equal to $\Theta(1)$. Therefore, adding n elements with same key to a Binary Search Tree defined as in the text is equal to $n\Theta(1)$ which is equal to $\Theta(n)$.

Proof:

- Let $T(n)$ denote the time it takes to add a node to a binary search tree containing n nodes.
- Let $T(0) = c$ where $T(0)$ denotes the time it takes to add the root to a binary search tree.
- Let $L(n)$ be the time it takes to add an element to a Linked-List such that $L(n) = d$
- Let e denote the time it takes to compare two nodes' keys.
- Let $k_1, k_2, \ldots, k_n$ be nodes to be added to a binary search tree defined as in the text such that $k_1.key = k_2.key = \ldots = k_n.key$
- Let $Q(n)$ be equal to the time it takes to add $k_1, k_2, \ldots, k_n$ to the Binary Search Tree.

It follows that:

$$
\begin{aligned}
Q(n) &= c + (e+d) + (e+d) + (e+d) + ... + (e+d) \\
&= (n-1)(e+d) + c \\
&= \Theta(n)
\end{aligned}
$$

4 Appendix

1. Test Sort Class

```java
public class TestSort {

public static void main(String[] args) throws IOException{

    ArrayList<Double> arrayAverageInsertionSortTime = new ArrayList<>();
    ArrayList<Double> arrayAverageShellSortTime = new ArrayList<>();
    ArrayList<Double> arrayAverageQuickSortTime = new ArrayList<>();

    final int MAX_ARRAY_SIZE = 10000;

    for (int arraySize=2; arraySize<= MAX_ARRAY_SIZE; arraySize++){

        double insertionSortTotalTime = 0;
        double shellSortTotalTime = 0;
        double quickSortTotalTime = 0;
        final int SAMPLE_SIZE = 20;


        for (int sampleNumber =1; sampleNumber<= SAMPLE_SIZE; sampleNumber++){

            double[] data = new double[arraySize];

            for (int n = 0; n < arraySize; n++) {
                data[n] = Math.random();
            }

            double[] data1 = (double[])data.clone();
            double[] data2 = (double[])data.clone();
            double[] data3 = (double[])data.clone();

            double time;

            System.out.println("Sample number "+ sampleNumber + " for an array of size " + arraySize);

            long time_prev = System.nanoTime();



            //INSERTION-SORT
            InsertionSort(data1);
            time = (System.nanoTime()-time_prev)/1000000000.0;
            insertionSortTotalTime += time;
            System.out.println("Insertion Sort\nTime= " + time+ "\n");

            //SHELL-SORT
```

```java
                time_prev = System.nanoTime();
                ShellSort(data2);
                time= (System.nanoTime()-time_prev)/1000000000.0;
                shellSortTotalTime += time;
                System.out.println("Shell Sort\nTime= " + time + "\n");

                //QUICK-SORT
                time_prev = System.nanoTime();
                Arrays.sort(data3);
                time = (System.nanoTime()-time_prev)/1000000000.0;
                quickSortTotalTime += time;
                System.out.println("Quick Sort\nTime= " + time + "\n");
            }

            double averageInsertionSortTime = insertionSortTotalTime/SAMPLE_SIZE;
            double averageShellSortTime = shellSortTotalTime/SAMPLE_SIZE;
            double averageQuickSortTime = quickSortTotalTime/SAMPLE_SIZE;


            arrayAverageInsertionSortTime.add(averageInsertionSortTime);
            arrayAverageShellSortTime.add(averageShellSortTime);
            arrayAverageQuickSortTime.add(averageQuickSortTime);

            System.out.println("Average time for sorting an array of " + arraySize + " elements");
            System.out.println("Insertion Sort\nAverage Time= " + averageInsertionSortTime + "\n");
            System.out.println("Shell Sort\nAverage Time= " + averageShellSortTime + "\n");
            System.out.println("Quick Sort\nAverage Time= " + averageQuickSortTime + "\n");

        }

        //Insertion Sort Normal
        BufferedWriter br = new BufferedWriter(new FileWriter("averageInsertionSort50000.csv"));
        StringBuilder sb = new StringBuilder();
        // Append from array
        int xAxis = 2;
        for (double element : arrayAverageInsertionSortTime) {
            sb.append(xAxis);
            sb.append(",");
            sb.append(element);
            sb.append(System.lineSeparator());
            xAxis++;
        }
```

```java

        br.write(sb.toString());
        br.close();

        //Insertion Sort LogLog
        br = new BufferedWriter(new FileWriter(
       "averageInsertionSortLogLog50000.csv"));
        sb = new StringBuilder();
        // Append from array
        xAxis = 2;
        for (double element : arrayAverageInsertionSortTime) {
            sb.append(Math.log(xAxis));
            sb.append(",");
            sb.append(Math.log(element));
            sb.append(System.lineSeparator());
            xAxis++;
        }

        br.write(sb.toString());
        br.close();


        //Shell Sort Normal
        br = new BufferedWriter(new FileWriter("
        averageShellSort50000.csv"));
        sb = new StringBuilder();
        // Append from array
        xAxis = 2;
        for (double element : arrayAverageShellSortTime) {
            sb.append(xAxis);
            sb.append(",");
            sb.append(element);
            sb.append(System.lineSeparator());
            xAxis++;
        }

        br.write(sb.toString());
        br.close();

        //Shell Sort LogLog
        br = new BufferedWriter(new FileWriter("
        averageShellSortLogLog.csv"));
        sb = new StringBuilder();
        // Append from array
        xAxis = 2;
        for (double element : arrayAverageShellSortTime) {
            sb.append(Math.log(xAxis));
            sb.append(",");
            sb.append(Math.log(element));
            sb.append(System.lineSeparator());
            xAxis++;
        }

        br.write(sb.toString());
        br.close();

        //Quick Sort Normal
```

```java
        br = new BufferedWriter(new FileWriter("averageQuickSort.csv"));
        sb = new StringBuilder();
        // Append from array
        xAxis = 2;
        for (double element : arrayAverageQuickSortTime) {
            sb.append(xAxis);
            sb.append(",");
            sb.append(element);
            sb.append(System.lineSeparator());
            xAxis++;
        }

        br.write(sb.toString());
        br.close();

        //Quick Sort LogLog
        br = new BufferedWriter(new FileWriter("averageQuickSortLogLog.csv"));
        sb = new StringBuilder();
        // Append from array
        xAxis = 2;
        for (double element : arrayAverageQuickSortTime) {
            sb.append(Math.log(xAxis));
            sb.append(",");
            sb.append(Math.log(element));
            sb.append(System.lineSeparator());
            xAxis++;
        }

        br.write(sb.toString());
        br.close();
    }
```

2. Graph Colouring Java Code

```
public static void main(String[] args) throws IOException{

ArrayList<Double> arrayAverageTime = new ArrayList<>();

for (int i=12; i<18; i++ ){

    final int SAMPLE_NUMBER=200;
    double totalTime=0;

    for (int samples=0; samples<SAMPLE_NUMBER; samples++){

        double time;
        Graph graph = new Graph(i, 0.5);
        long time_prev = System.nanoTime();
        Colouring colouring = graph.bestColouring(3);
        time = (System.nanoTime()-time_prev)/1000000000.0;
        totalTime += time;

    }

    double averageTotalTime=totalTime/SAMPLE_NUMBER;
    arrayAverageTime.add(averageTotalTime);
}


//into a csv file
BufferedWriter br = new BufferedWriter(new FileWriter("
averageGraphColouringAlgorithm" + ".csv"));
StringBuilder sb = new StringBuilder();
// Append from array
int xAxis = 12;
for (double element : arrayAverageTime) {
    sb.append(xAxis);
    sb.append(",");
    sb.append(element);
    sb.append(System.lineSeparator());
    xAxis++;
}

br.write(sb.toString());
br.close();

//into a csv file
br = new BufferedWriter(new FileWriter("
averageGraphColouringAlgorithmLogNorm.csv"));
sb = new StringBuilder();
// Append from array
xAxis = 12;
for (double element : arrayAverageTime) {
    sb.append(xAxis);
    sb.append(",");
    sb.append(Math.log(element)/Math.log(3));
    sb.append(System.lineSeparator());
    xAxis++;
}

```

```
    br.write(sb.toString());
    br.close();
}
```