**COMP1201 Assignment 1**

**Q1:**

(a) To measure the running time of different sizes of arrays we must create, for example, three more arrays with modified sizes:

//Create more variables to store the lengths of the arrays.

int N = 1000;

int smallerN = 500;

int evenSmallerN = 100;

int biggerN = 2500;

//Create more arrays with the modified sizes

double[] smallerData = new double[smallerN];

double[] evenSmallerData = new double[evenSmallerN];

double[] biggerData = new double[biggerN];

Then we must populate the arrays, via for loops, using the *Math.random()* method.

Afterwards we must create 3 data containers (arrays) for each of the previously created arrays (data, smallerData, evenSmallerData and biggerData). The created sub-containers are copies of the arrays.

//Put all the data from the original array in these ones

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

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

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

double[] smallerData1 = (double[])smallerData.clone();

double[] smallerData2 = (double[])smallerData.clone();

double[] smallerData3 = (double[])smallerData.clone();

double[] evenSmallerData1 = (double[])evenSmallerData.clone();

double[] evenSmallerData2 = (double[])evenSmallerData.clone();

double[] evenSmallerData3 = (double[])evenSmallerData.clone();

double[] biggerData1 = (double[])biggerData.clone();

double[] biggerData2 = (double[])biggerData.clone();

double[] biggerData3 = (double[])biggerData.clone();

Moving forward, we must call each of three algorithms (insertion sort, shell sort and quick sort) for each of the sub-data containers (for data those would be data1, data2, data3).

Example for smallerData:

//Calls the first method - InsertionSort(smallerData1);

time = (System.nanoTime()-time\_prev\_smaller)/1000000000.0;

System.out.println("Insertion Sort\nTime= " + time);

time\_prev\_smaller = System.nanoTime();

//Calls the second method - ShellSort(smallerData2);

time = (System.nanoTime()-time\_prev\_smaller)/1000000000.0;

System.out.println("Shell Sort\nTime= " + time);

time\_prev\_smaller = System.nanoTime();

//Calls the third method - Arrays.sort(smallerData3);

time = (System.nanoTime()-time\_prev\_smaller)/1000000000.0;

System.out.println("Quick Sort\nTime= " + time);

In the end, if we want to display all the items (numbers) in the array (smallerData) we could use the following:

//Display all the info System.out.println("SMALLER-DATA");

System.out.println("\tPresorted\tInsertion\t\t Shell\t\t Quick");

for (int i=0; i<smallerData.length; i++) System.out.println(smallerData[i] + " " + smallerData1[i] + " " + smallerData2[i] + " " + smallerData3[i]);

(b)

(c)

Average-case time complexity of insertion sort is Θ(n^2). That is because the gradient (*which is 2*) tells us the rate of growth of the time complexity.

(d)

The average running time of insertion sort for input of size 10^10 (10 000 000 000) should be around 7 000 000 000 seconds. If we look at the log-log graph we can see that the ratio is maintained when for an input of 10 000 000 000 it takes 7 000 000 000 seconds.

**Q2:**

(a)

The new class will loop through sizes 12 to 17 and create a new graph for each size.

The class will contain only one method, which is static. This way you don’t have to instantiate it.

public class GraphExtension {  
  
 public static void estimateTime() {  
  
 long time\_prev = System.nanoTime();  
 double time;  
  
 for (int i = 12; i < 18; i++) {  
 Graph graph = new Graph(i, 0.5);  
 Colouring colouring = graph.bestColouring(3);  
 graph.show(colouring);  
 time = (System.nanoTime() - time\_prev) / 1000000000.0;  
 System.out.println(i + " points time = " + time);  
 time\_prev = System.nanoTime();  
 }  
  
 }  
}  
  
  
//The main method only contains:

GraphExtension.estimateTime();

(b)

**Q3:**

(i) Generally speaking, in order to insert an element in a binary search tree, we must traverse through the elements in a tree until we get to a node after which we can add our element. Which mean our worst case time complexity will be O(n).

Now more specifically for the question. Since we are working with keys, the values of the nodes are not important. If all the keys are identical, then we treat every one of them the same way.

E.g. If we add 5 elements to this tree:

With three nodes added, we can see that: The first one is the root. The second and third nodes are added to the right of because they are *EQUAL* to each other.

Continuing the example:

From the example we can deduce that in order for us to continue inserting elements, we need to traverse through all the previously added elements, proving that the time complexity is O(n).

(ii) Again, we are adding elements with the same keys, so their values are not of importance for the exercise. A flag is added to each node, which is changed when the node is visited while adding an element. If the value of the flag is **0** – we add to the **left**. If it is **1** – we add to the **right**.

Example: Adding 6 elements to this binary search tree

After adding three elements, we can see that with the flag feature introduced, we have a better (faster) time complexity. Continuing the example:

With this method of insertion (using the flag) we get a balanced binary search tree, which has a height of log(n). This making our time complexity be equal to O(log(n)).

(iii) If the insertion made into the BST has a key that is already in the BST, then the value under the key is added to a (singly) linked list.

Example of inserting 4 *IDENTICAL ELEMENTS*.

The elements are:

<key, value>

<a, 1>

<a, 1>

<a, 1>

<a, 1>

1 -> 1

1

If we continue to insert *IDENTICAL ELEMENTS* into this binary search tree, we would just continue adding to the (singly) linked list. Hence the time complexity of insertion into the BST is that of inserting an element into a (singly) linked list, which is O(1).

1 -> 1 -> 1 -> 1

1 -> 1 -> 1