Algorithmics Tutorial Sheet 1

Measure the time complexity of sort

Name: Palak Jain
Soton email address: ppj1u18@soton.ac.uk
ID: 30012627

Throughout the tutorial sheet, some parts of code are highlighted. These parts are the altered/newly added parts of the provided code, which have been highlighted for greater clarity.

1. Compile and run 'TestSort.java'

2. What does 'TestSort.java' do?

TestSort.java generates an array of 1000 random doubles and sorts them using Insertion sort, Shell sort and Quick sort (the java default). It also measures the time taken by each sorting algorithm to complete the sorting and prints this time.

TestSort.java

```
import java.util.Arrays;
import java.io.*;

public class TestSort {
   public static void main(String[] args) {
          int times = 10000;
          double[] insertionSortTime = new double[times];
          double[] shellSortTime = new double[times];
          double[] quickSortTime = new double[times];

                int N = 1000;
                for(int j=0; j<times; j++){
                      double[] data = new double[N];

                      for (int i = 0; i < N; i++)
                         data[i] = Math.random();

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

                      long time_prev = System.nanoTime();
                      InsertionSort(data1);
                      double time = (System.nanoTime()-time_prev)/1000000000.0;
                      System.out.println("Insertion Sort\nTime= " + time);
```

```
                    insertionSortTime[j] = time;

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

                    shellSortTime[j] = time;

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

                    quickSortTime[j] = time;

                System.out.println("Insertion Sort Median time = " +median(insertionSortTime));
                System.out.println("Shell Sort \n Median time = " + median(shellSortTime));
                System.out.println("Quick Sort \n Median time = " + median(quickSortTime));

        ……..

    public static double median(double[] a) {
            Arrays.sort(a);
            return a[a.length/2];
    }
}
```

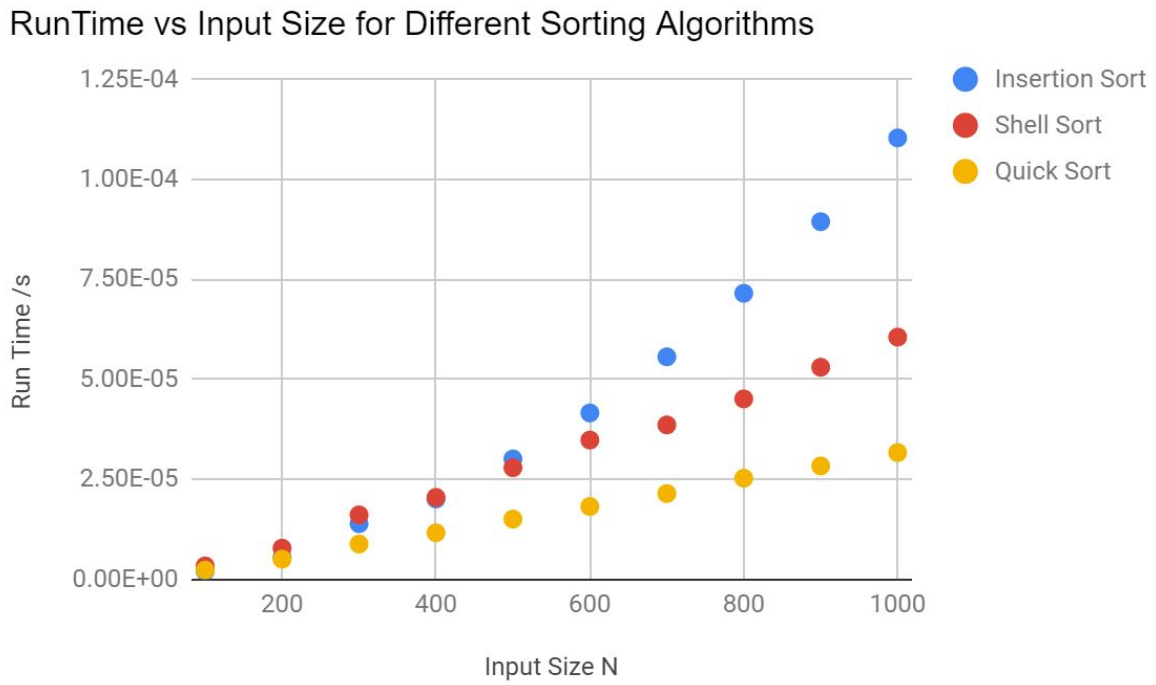
3. Modify TestSort to measure the time complexity on different sizes of arrays. Plot a graph of the run time versus size of input.

The TestSort program has been modified slightly, and this modified version is pasted above.

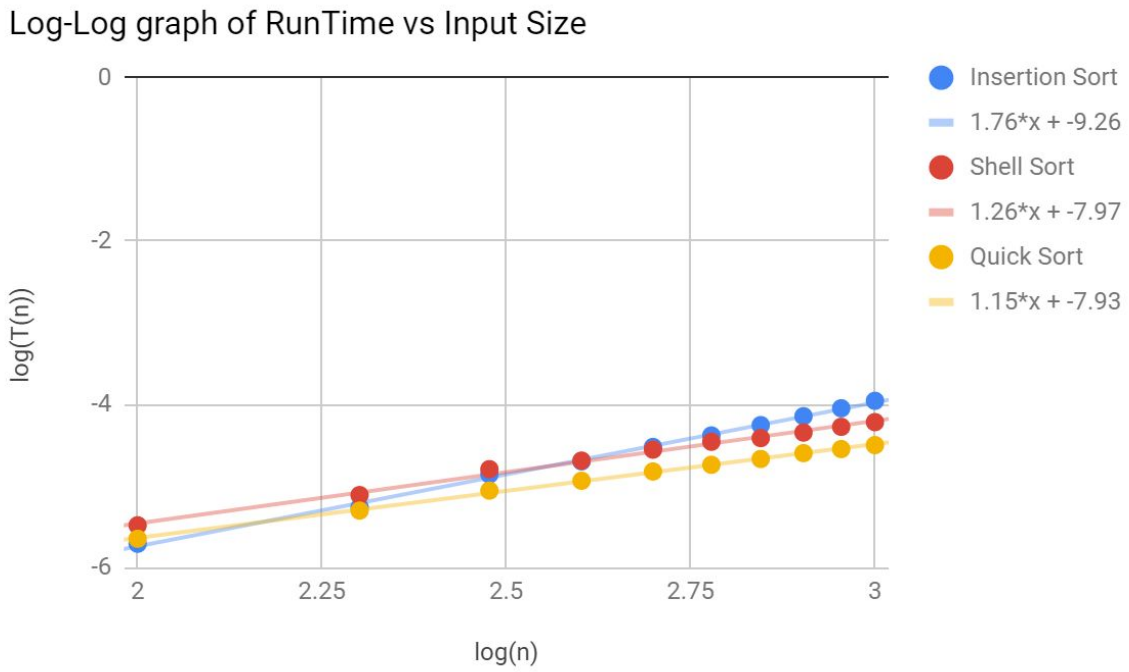
Another array of size 10000 is created and each sorting algorithm is run that many times for input size starting from 100 and until 1000, with an increment of 100 data values each time. The 10000 values for each sorting algorithm are recorded and the median method is called for each input size to find the median of the values. Taking the median of such large number of data values greatly increases the reliability of results and reduces the effect of outliers on the recorded data.

The graph below records the median runtimes for each input size. It shows approximate linear relationships between the Run Time and Input Size for the different sorting algorithms. For smaller input sizes, there is not a significant difference between the run times of the sorting algorithms. However, for large input sizes(approaching 1000 data values), we can clearly see from the graph that the Insertion Sort algorithm takes the most amount of time, while the

QuickSort algorithm takes the least. The diversion between the graph grows with input size as QuickSort gets relatively faster than Shell Sort, which becomes relatively faster than Insertion Sort.



4. If a program runs in $O(n^a)$ (i.e. $T = cn^a$) then we find the logarithm of the run time grows linearly with the logarithm of size of the input $log(T)$. Take the logarithm of the run time and array size for the previous data and replot your graph as a log-log graph.



5.Estimate the time complexity for insertion sort by measuring the gradient of the log-log plot.

We know that $T(n) = cn^a$
Taking log of both sides, we get: $log(T(n)) = alog(n) + log(c)$, which is the form of the lines of best fit. Thus, the gradient a of the lines determines the order of time complexity. As can be seen from the line of best fit for the insertion sort algorithm, from the graph above, its gradient is 1.76, which rounds to the whole number 2. Thus, the time complexity for the algorithm is approximately $O(n^2)$.

6. Write a new class to measure the run time of this program for problems of size 12 to 17. Plot a graph of the logarithm of the run times versus problem size, i.e. n not log(n). From your measurements estimate the time complexity---is this what you would expect? How does the time complexity of this algorithm compare with sort?

Class TestSort1 was created to calculate the run time of the graph coloring program. It has one method- the measureTime() method, which, records the time taken for the algorithm to run. It creates an array of size 6 to store the values for 12-16 nodes. For each node, it creates an instance of graph, with a probability factor 0.5. The time is started using the System.nanoTime() method, the bestColouring() method is called, then the time is stopped and recorded in an array. The main method in Graph class is altered; as new instance of Test Sort1 is created and the measureTime() method is called on it.

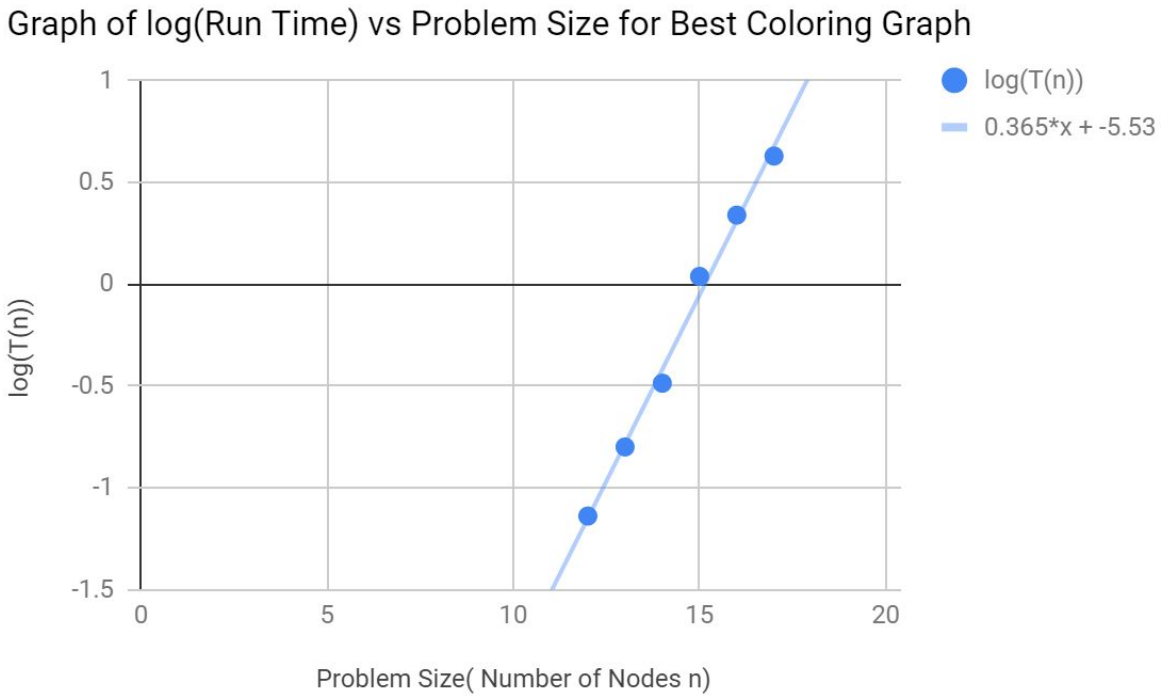
TestSort1.java

```
import java.util.Arrays;
import java.io.*;

public class TestSort1 {
	Graph graph;
	int i;
	public void measureTime(){
		int N = 6;
		double[] data = new double[N];

		for(i=12; i<=17; i++){
			graph = new Graph(i, 0.5);
			long time_prev = System.nanoTime();
			graph.bestColouring(3);
			double time = (System.nanoTime()-time_prev)/1000000000.0;
			System.out.println("Time taken for" + i +"nodes is" + time);
			data[i-12] = time;
		}
	}
}
```

Graph.java

```
import java.util.*;
import java.io.*;
import java.awt.Color;

public class Graph {
…….
        public static void main(String[] args) {
                TestSort1 testSort = new TestSort1();
                testSort.measureTime();
        }
……..
}
```

The graph below shows a linear relationship between the problem size and $log(T(n))$. As $log(T(n)) = cn + log(a)$, the line of best fit of the graph is $log(T(n)) = 0.365n - 5.53$, where 0.365 is c, and -5.53 is $log(a)$. Thus $T(n) = 10^{(0.365n-5.53)}$. So $T(n) = 10^{-5.53} * 10^{0.365n}$. Therefore, $T(n) = 10^{-5.53} * 2.31^n$. Therefore, the time complexity is exponential. Now, this is expected, as the graph of run time vs problem size is exponential, $T(n)$ is of exponential form; $T(n) = ae^{cn}$. The sorting algorithms have a polynomial time complexity whereas this algorithm has an exponential time complexity. As the exponential function is steeper than the polynomial functions, it is slower and therefore has a higher time complexity.



7. Compile the codes (to compile C++ use > g++ -O4 -o MySort MySort.cc). They all take an argument which is the number of elements to sort. Compare their run times (you might want to modify the program to do this more efficiently).

The sorting programs are altered so that 10000 values are stored for each input size and the median is calculated for each input size. The time is started using System.nanoTime(), the sort method is called and the time is stopped, again using the same method. The same is done with the c++ code. The clock() method is used to start and stop time in c++.

MySort.java

```
import java.io.*;
import java.util.*;

public class MySort{
	public static void main(String [] args){
		int times = 10000;
		double[] sortTime = new double[times];

		for(int j=0;j < times; j++){
			int N = Integer.parseInt(args[0]);
			double[] data = new double[N];

			for (int i=0; i<N; i++){
				data[i] = Math.random();
			}
			long time_prev = System.nanoTime();
			Arrays.sort(data);
			double time = (System.nanoTime()-time_prev)/1000000000.0;

			sortTime[j] = time;
			System.out.println("Sort Time= " + time);
			/* for (double d: data)
			System.out.println(d); */
		}

		System.out.println(median(sortTime));
	}

	public static double median(double[] a) {
		Arrays.sort(a);
		return a[a.length/2];
	}
}
```

MySortAL.java

```
import java.io.*;
import java.util.*;

public class MySortAL{
	public static void main(String [] args ){
		int times = 10000;
		double[] sortTime = new double[times];

		for(int j=0;j < times; j++){
			int N = Integer.parseInt(args[0]);
			List<Double> data = new ArrayList<Double>(N);

			for (int i=0; i<N; i++){
				data.add(Math.random());
			}

			long time_prev = System.nanoTime();
			Collections.sort(data);
			double time = (System.nanoTime()-time_prev)/1000000000.0;
			sortTime[j] = time;
			System.out.println("Sort Time= " + time);
			/* for (double d: data)
			System.out.println(d); */
		}

		System.out.println(median(sortTime));
	}

	public static double median(double[] a) {
		Arrays.sort(a);
		return a[a.length/2];
	}
}
```

MySort.cc

```
#include <iostream>
#include <fstream>
#include <iterator>
#include <vector>
#include <algorithm>
#include <iostream>
#include <stdio.h>

using namespace std;
```

```
int main(int argc, char *argv[]){
	int N;
	clock_t t;
	double time_taken;

	sscanf(argv[1], "%d", &N);

	vector<double> runTime(10000);

	vector<double> data(N);

	for(int j=0; j<10000;j++){
		for(unsigned int i=0; i<N; i++) {
			data[i] = rand()/(RAND_MAX+1.0)
		}
		t = clock();
		sort(data.begin(), data.end());
		t = clock() - t;
		time_taken = ((double)t)/CLOCKS_PER_SEC; // in seconds
		runTime[j] = time_taken;
	}

	printf("MySort.cc took %f seconds to execute \n", time_taken);
	return 0;
	sort(runTime.begin(), runTime.end());
	cout<< runTime[runTime.size()/2]<<"\n";
}
```

As can be seen from the graph, the relationship between Run Time and Input Size is linear; MySort.cc is the fastest program, followed by MySort.java, and lastly MySortAL.java which is significantly slower than both.