Lab walk-through #8	
Topic	**Experiments with Algorithms**

1. Introduction

Comparing the strengths and weaknesses in the algorithms you design is an essential part of software development. It helps you work out which algorithm or technique works best for the application you intend it for. Usually, implementations of algorithms are compared in terms of the running time and memory space they use.

In this lab we will look at some methods of doing this comparison using the StdDraw and Stopwatch libraries in Java.

2. Lab Objectives

By the end of this lab the student will learn how to use various Java libraries to compare running times of various algorithms.

3. Lab Setup

Before beginning this lab, you should have:

1. The Eclipse and Java Runtime Environment on your computer
2. Completed the Sorting Algorithms lab and its practice questions
3. Read Chapter 1 and 2 in your textbook
4. Before you start doing this section, in your Eclipse workspace create a project named 2XB3_Lab8 inside a package named cas.lab8.wt. Add classes to this project as you walk through this instruction.

4. Lab Exercise

To measure an algorithm's efficiency, we use just a few structural primitives (statements, conditionals, loops, nesting, and method calls) to implement algorithms, so very often the order of growth of the cost is a function of the problem size N.

The most common classifications (but not exhaustive) orders of growth are:

1. Constant – 1. The algorithm executes a fixed number of statements to finish its job. An example would be a single statement in Java that adds two numbers.
2. Logarithmic – logN. Very slightly slower than a constant-time program. The base of the logarithm is not relevant to the order of growth. An example is binary search.
3. Linear – N. Programs that spend a constant amount of time processing each piece of input data, or that are based on a single for loop.
4. Linearithmic – NlogN. Merge sort and quick sort are some common examples of this order of growth.

5. Quadratic – N^2. A typical program whose running time has order of growth N^2 has two nested for loops, used for some calculation involving all pairs of N elements. The elementary sorting algorithms selection sort and insertion sort are algorithms in this classification.
6. Cubic - N^3. Three nested loops are usually the constructs responsible for this classification.
7. Exponential – 2^N. These are extremely slow and can usually not be completed for a large problem. There exists a large class of problems for which an exponential algorithm is the best possible choice.

How we can get a particular order of growth from an algorithm is described in the next section. Also, there is a subtle difference between the order of growth of algorithms and order of growth of your implementations. Due to some implementation details that may differ for various programming languages, the programming constructs may be easier or more complicated in some languages. For large problems this difference becomes greater. Other factors also come into play. For example, CPU speed and power, operating systems used, as well as the compiler for the language.

Section 1: Comparing and Analysing Sorting Algorithms

Task 1: Import Sorting Algorithms

For this lab exercise use your Sorting Algorithms lab from last week.

1. Import your implementations from last week into a Java Project.
2. Give the Comparable interface an Integer parameter where you have used it in your implementation so that we can run, test and time these algorithms.
3. Create a new class called ExperimentsSort.java for this walkthrough in the same package as your implementation from the previous walkthrough. It will only contain a main method where you will carry out these experiments.

When we want to decide which algorithm is more efficient, we are referring to the running time of the program and its memory usage. Since memory usage is easily analysed (see page 200 in your textbook), in this lab we will focus on analysing running time. Let's do an experiment to measure time taken by a program implementing an algorithm.

There are different ways to measure time in Java. One of the easiest ways is to use the System's time measurement method called `currentTimeMillis` which returns the difference, measured in milliseconds, between the current time and midnight, January 1, 1970 UTC. This is an arbitrarily chosen date and time but it works nonetheless. The accuracy however, depends on the processor you are using, sometimes to within tens of milliseconds, but it is fine for our purposes. There is also another method, nanoTime(), but use it with care as it is known to have some issues when working on multiple cores.

We need accurate measurements to generate experimental data that we can use to formulate and check the validity of hypotheses about the relationship between running time and problem size. For this purpose, we use the Stopwatch data type shown below and in your textbook. Its `elapsedTime()` method returns the elapsed time since it was created, in seconds. The implementation is based on using the Java system's `currentTimeMillis()` method, which gives the current time in milliseconds, to save the time when the constructor is invoked, then uses it again to compute the elapsed time when `elapsedTime()` is invoked.

Task 2: Creating and Using Stopwatch

Create a class called Stopwatch.java in your package and copy and paste the code found at this link: http://algs4.cs.princeton.edu/14analysis/Stopwatch.java.html. Add the following code to your program to check the elapsed time of a method.

```
Stopwatch stopwatch = new Stopwatch ();
double startTime = stopwatch.elapsedTime();
SelectionSort.sort(array);
double endTime = stopwatch.elapsedTime();

System.out.println("Total execution time: " + (endTime - startTime) );
```

With this you can get a good estimate of how much time the statements in between the start and end times took although it can be a little inaccurate for small data sets.

There is another Stopwatch library in the apache commons library found here:
https://commons.apache.org/proper/commons-lang/javadocs/api-2.6/org/apache/commons/lang/time/StopWatch.html
This also uses `System.currentTimeMillis()` but also has other useful methods attached to it. Study and use this library to make sure you understand how to import external libraries into your Eclipse project. You will need this skill to consistently work in Java. It is a language that owes much of its fame to the vast number of libraries available for use.

Use random numbers for the information in the arrays to be sorted. This achieves a level of uniformity between all experiments. To generate a random number in Java, use the following code. Remember to import the Java utility library called Random.

```
Random random = new Random();
Integer newNumber = new Integer(random.nextInt(10000));
```

The code above generates a random number between 0(inclusively) and 10000(exclusively). Don't forget that the Comparable interface has been implemented for the type Integer and not the primitive type int. Run your experiment at least 10 times for the sorting algorithm you are testing with different amounts of inputs. Draw a table of your results. How would you improve this experiment further?

Task 3: Evaluating Execution Time of Algorithm

Create two arrays in your ExperimentsSort.java, one that contains the sizes of the inputs(x) and one that stores the corresponding times(y). Name them x and y. You will use them to print out your results and later to plot graphs.

Use a varying numbers of inputs of up to 100000 elements in your test arrays and measure the timings of your selection sort algorithm. Fill out the column T(N) in the following table for the recorded running times of your selection sort algorithm as a function of the problem size (N).

Table 1 - Selection sort times on Integer arrays of varying sizes	
N	**T(N)**
10	*Recorded time*
100	*Recorded time*
1000	*Recorded time*
10000	*Recorded time*
100000	*Recorded time*
1000000	*Estimated time*
10000000	*Estimated time*

Task 4: Making a hypothesis

Compare the growth rate of time and the growth of N as N grows up to 100000? To help answer this question, plot the data both on a normal and on a log-log scale, with the problem size *N* on the *x*-axis and the running time *T*(*N*) on the *y*-axis. You can use the instructions in Section 3 of this walk-through to plot a diagram. Can you fit a straight line of slope 2 to the log-log plot?
If yes the equation of such a line is

$$\lg (T(N)) = 2 \lg N + \lg a$$

Which is equivalent to

$$T(N) = a\, N^2$$

Use one of our data points in the Table 1 and solve the equation for *a*—for example,

$$T(10000) = a * 10000^2$$

$$166 = a * 10000^2, \text{ so } a = 1.66 * 10^{-6}$$

Can the log-log plot lead you to make the following hypothesis?

The running time of my selection sort algorithm is the following function of my array size N:

$$T(N) = 1.66 * 10^{-6} N^2$$

Note: Replace the value of a ($1.66 * 10^{-6}$) in this equation with your computed value for a.

Task 5: Prediction

Use the function in your hypothesis and make an estimate of running time for an array size of 1,000,000 and 10,000,000. Write down your estimated time in Table 1.

Task 6: Verification
Run the program this time for array sizes of 10^6 and 10^7 and measure the elapsed execution times. Are your estimated times close to the measured times? Can you verify that your hypothesis holds?

Section 2: Mathematical Model
Our main goal in doing experimental analysis of algorithms is predicting program behaviour (in this lab our focus of behaviour is on the running time). As you have seen in Task 6 above we can repeat our experiment multiple times to show that our hypothesis and our observations agree. As it is true for all scientific experiments we can never know for sure that any hypothesis is absolutely correct; we can only validate that it is consistent with our observations. To prove that a hypothesis is always valid we need a mathematical model.

Task 7: Build a mathematical model for your selection sort
Start with the selection sort algorithm, outlined here again:

```
…
for (int i = 0; i < N; i++){
      int min = i;
      for (int j = i+1; j < N; j++)
            if (less(a[j], a[min])) min = j;
      swap(a, i, min);
}
…
```

Time taken for an algorithm depends on the number of instruction calls (or in our case, method calls). We have two methods that we explicitly call in this implementation: `less` (compare method) and `swap` (movement of data method). Since we cannot control on what processors and which compiler the user chooses to run this algorithm on, let us assume that both methods take roughly the same amount of time (they are really small methods).

Given that there are n elements in the array, how many `less` method calls are there? N-1 in the first iteration of the outer loop, then N-2 in the second, and so on until there is one call in the last iteration. If you add everything together, this is a commonly used result in mathematics using binomial coefficients. The answer is (N*(N-1))/2. Expanding and simplifying this formula gets us to $N^2/2$. Even with the division of 2, what determines the size is the polynomial N^2. So we can simplify to $N*N = N^2$ calls.

What about `swap`? Since this method call is solely in the outer loop, it is called only n times in the whole sort function. So the total "time" taken is $N^2 + N$. However, N^2 is always a much larger number than N. Adding the N `swap` method calls doesn't change the time by as large a factor as the N^2 so let's simplify again and leave that out. We can say that the running time is dominated by the `less` method.

Furthermore, the execution of the sort method doesn't change no matter what the input looks like. If it is a large or a small input, it will always run in N^2 time. In other word the running time is insensitive to how the input is structured. The algorithm doesn't check first if the array is partially sorted or fully

sorted in which case it always takes the same amount of time to run, even if it is given an already sorted array. So the worst case and best case scenario for the selection sort are all the same. Clearly this is not a good solution if you have an application with hope of some partial sorting.

There is a notation called Big-O (pronounced Big-Oh) which standardises how to describe the performance of algorithms under this sort of analysis. It represents the limiting behaviour of an algorithm as the input tends towards infinity. For insertion sort, the mathematical model in Big-O notation is $O(n^2)$ with respect to execution time.

Work out the Big-O behaviour for your sorting algorithms. Take into account the worst case and best case as well the dominating method calls.

Task 8: Compare the experimental results with the results of the mathematical model

Now that you have the mathematical model for the running time of your selection sort ($O(n^2)$ you can compute how much time your algorithm will take to do the sort for different array sizes. Add one column to Table 1 ($T_2(N)$) and fill out the computed times. Draw both on a normal and on a log-log scale with the problem size N on the x-axis and the running time $T(N)$ and $T_2(N)$ on the y-axis. Do these two diagrams overlap? If no do you observe any changes as the size of N grows?

The mathematical model allows us to predict how this algorithm would perform under an arbitrary N without actually running the program for all possible values of N. By analysing the number of instructions in an implementation and building the mathematical model we can also see under what conditions we can improve the performance of an algorithm. Can you think about a sort algorithm that's running time is sub-quadratic?

Carry out the same analysis for the bubble sort and merge sort algorithms. The answers for insertion sort are in your textbook on page 250. Check that your analysis is accurate. Make sure you plot your results too. Based on this information, which one do you think is the superior algorithm for general sorting? Why?

Section 3: Plotting results on a graph

Sometimes the best way to convey information is visual representation. Use the StdDraw library to construct a graph of your results. Create a new class in the same package called StdDraw.java and copy and paste the code found on your book's website here:
http://introcs.cs.princeton.edu/java/stdlib/StdDraw.java
For extensive use of this library, refer to your book(page 43-45) as well as this page:
http://introcs.cs.princeton.edu/java/15inout/

Task 9: Plot your results on a graph using this library.

You do not need to import anything since it is in the same package as your implementation. Recall that you have created two arrays in your ExperimentsSort.java, one that contains the sizes of the inputs(x) and one that stores the corresponding times(y).

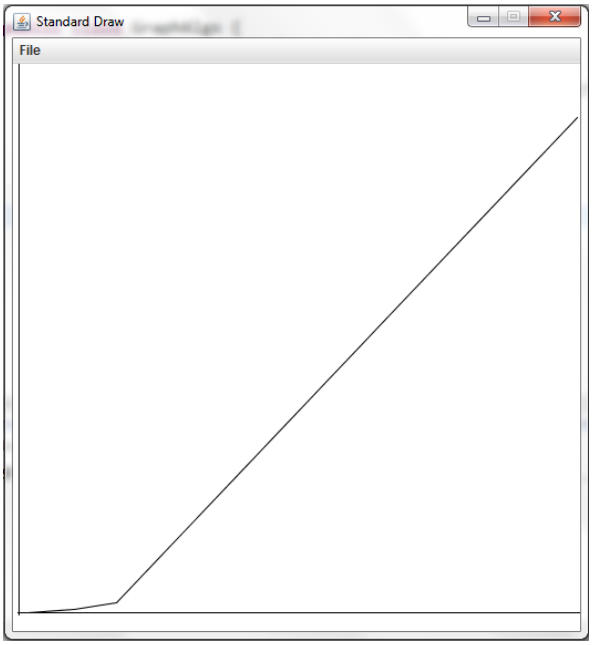
Write the following code in your ExperimentsSort.java:

```
StdDraw.setXscale(-1000, 100500);
StdDraw.setYscale(-1000, 30000);

StdDraw.line(-100, 0 ,100500, 0);//x-axis
StdDraw.line(0, -100, 0, 30000);//y-axis

for(int i = 0; i < 9; i++)
{
      StdDraw.line(x[i], y[i], x[i+1], y[i+1]);
}
```

Now run your program. Is this the shape you expected? How can you make it smoother? Check the StdDraw APIs and control methods in page 43 of your textbook.



Use StdDraw to draw your predicted function of the selection sort algorithm alongside your results using the mathematical model with a different colour. (Page 43-45 in your textbook). How would you compare these two functions?

You can now incorporate your other sorting implementations in the same way. This way, you have a visual comparison between them and can see the growth rates quite easily. You can make many decisions with this information. Depending on the amount of inputs for a specific application, you can now pick the best algorithm for a particular job. You can also verify mathematical models for an algorithm.

Task 10. Submit your work

Once all tasks are completed, you should submit your Eclipse project. This must be done before leaving the lab. Follow the instructions below for submission:

- Include a .txt file named last_name_initials.txt in the root of the project containing on separate lines: Full name, student number, any design decisions/assumptions you feel need explanation or attention.
- After checking the accuracy and completeness of your project, right-click on the name of the project, select Export->General->Archive File.
- Ensure that just your project has a check-mark beside it, and select a path to export the project to. The filename of the zipped project must follow this format: *macID_Lab8.zip.* Check the option to save the file in .zip format. Click Finish to complete the export.
- Go to Avenue and upload your zipped project to 'Lab Walk-through 8 – Lab Section X')
- IMPORTANT: You MUST export the FULL Eclipse project. Individual files (e.g. java/class files) will NOT be accepted as a valid submission.

6. Further Practice Problems

For further practice, here are some suggested questions from your textbook: 1.4.14, 1.4.15, 1.4.16, 1.4.17, 1.4.18, 1.4.19, 1.4.20, 1.4.22.