Lab walk-through #4	
Topic	**Sort Algorithm Implementation**
Week	4 – Session 2
Week	5 – Session 1

1. Introduction

A sorting algorithm is an algorithm that puts elements of a list in a certain logical order. For example, bills are usually in chronological order; class lists are usually in alphabetical order of names or numerical order of student numbers.

As described in your Algorithms textbook "*Sorting plays a major role in commercial data processing and in modern scientific computing. Applications abound in transaction processing, combinatorial optimization, astrophysics, molecular dynamics, linguistics, genomics, weather prediction, and many other fields. Indeed, a sorting algorithm (quicksort, in Section 2.3) was named as one of the top ten algorithms for science and engineering of the 20th century*."

Sorting is also an intermediate step for solving other problems such as, searching a list. The famous binary search algorithm only achieves its logarithmic speeds by taking advantage of a sorted list. Like this, there are countless applications where sorting a data set aids in solving other more interesting problems.

2. Lab Objectives

By the end of this lab the student will learn implementation of different techniques using Java to solve a sorting problem.

3. Lab Setup

Before beginning this lab, you should have:

1. The Eclipse and Java Runtime Environment on your computer
2. Completed the JUnit lab and its practice questions
3. Read the Sorting chapter in your textbook

4. Lab Exercise

An algorithm is any well-defined computational procedure that takes some value or set of values as input and produces a value or a set of values as output. We can view the algorithm as a tool for solving a well-specified problem. The algorithm describes a specific computational **procedure** for achieving a desired input/output relationship.

Let us say we need to sort a sequence of numbers into ascending order. This is how we would formally describe the problem:

Input: A sequence of n numbers $\langle a_1, a_2, \ldots, a_n \rangle$
Output: A permutation (reordering) $\langle a'_1, a'_2, \ldots, a'_n \rangle$ of the input sequence such that $a'_1 \leq a'_2 \leq \ldots \leq a'_n$.

Can you think of a simple way to solve this problem?

Section 1: Sorting Algorithms

The first solution that probably came to your mind is finding the smallest element and putting this at the front of the list. Then finding the second smallest element and placing it at the second position in the list. and so on. This intuitive solution is sometimes called the **brute-force** implementation. In the vast collection of sorting algorithms, this is specifically known as **selection sort** since you are selecting the element you want to sort and putting it in its final position immediately.

Your textbook offers a useful template for classes that carry out sorting algorithms (P. 245). Use this template for your sorting algorithms in this walkthrough. As you can see in this template, instead of using primitive types such as integers, using the Comparable interface allows you to sort an array of any type that already has an order defined. Refer to page 247 in your textbook for a further explanation.

To start create an Eclipse project for this lab as cas2xb3_lab4 and then create a class called `SelectionSort` as defined below (main function is missing).

```
public class SelectionSort {

      private static boolean less(Comparable v, Comparable w)
      { return v.compareTo(w) < 0; }

      private static void exch(Comparable[] a, int i, int j)
      { Comparable t = a[i]; a[i] = a[j]; a[j] = t; }

      public static void sort(Comparable[] a)
      {     // Sort a[] into increasing order.
            int N = a.length; // array length
            for (int i = 0; i < N; i++)
            {     // Exchange a[i] with smallest entry in a[i+1...N).
                  int min = i; // index of minimal entr.
                  for (int j = i+1; j < N; j++)
                        if (less(a[j], a[min])) min = j;
                  exch(a, i, min);
            }
      }

      public static boolean isSorted(Comparable[] a)
      {     // Test whether the array entries are in order.
            for (int i = 1; i < a.length; i++)
                  if (less(a[i], a[i-1])) return false;
            return true;
      }
}
```

Our interest is in the `sort()` method which is the only one that will change in any of your implementations of sorting algorithms.

Initially, we start with a simple for loop that has an iteration for each element of the array. N in this loop is the length of the array. This represents the aim to fill out the sorted array with N elements.

```
for (int i = 0; i < N; i++){…
```

Next, where do we start to find the smallest element? It could be anywhere in this array. So to simplify our process, we automatically choose the first element of the unsorted array (`min = i`) and use this to compare all the other elements.

Next we construct a for loop that goes through the *remaining array* (at the start, the remaining array is the whole array) and checks if this arbitrary `min` value is smaller than any element we encounter so far. If it is, then our job is done. If it isn't, we replace the current `min` with this new `min` we've found.

```
if (less(a[j], a[min])) min = j;
```

The next step is an immediate swap of position of the `min` from our exercise above with the number of elements we have sorted. Initially this is 0 (`i` represents this number), but increases with every `min` we find (we are guaranteed to find the next min value every loop iteration).

```
exch(a, i, min);
```

This is one implementation of selection sort. Can you think of how to achieve the same thing with a while loop? This requires you to view to problem from a different angle. What would your condition be? From what you learned in Lab 3, implement a JUnit test to test this selection sort implementation and see that it works correctly.

Let us look at another simple sorting algorithm called *bubble sort*. This algorithm is also an intuitive way that some of you might have thought to sort a list. It selects two adjacent elements and checks if they are in the right order. If not, they are swapped. This procedure is repeated over the list until there are no more swaps.

Create a new class called `BubbleSort`. It should look like the `SelectionSort` example above, except the `sort`() method looks like the following:

```
public static void BubbleSort(Comparable[] num){
      int j;
      boolean sorted = true;// set flag to true to begin first pass
      while (sorted){
            sorted= false;//set flag to false awaiting a possible swap
            for(j=0;  j < num.length -1;  j++){
                  if (less(num[j+1],num[j])){
                        exch(num,j,j+1);
                        sorted = true;//shows a swap occurred
                  }
            }
      }
}
```

In this algorithm we define a boolean value called `sorted` to check whether we have made any swaps this iteration or not. We use a while loop on this variable to keep track of it. The inner for loop goes through the array of objects in pairs. The `if` statement checks whether they are out of order and if they are, they are swapped. Here the sorted variable is changed and the process is repeated. Now implement a JUnit test to check the implementation and see that it works.

As a last exercise, implement the **shell sort algorithm** that improves on bubble sort. Use the template specified in your textbook (p. 258). You will also need to implement a JUnit test for it to make sure it works correctly. Try file input and output to make your job easier every time.

For interest's sake, look at the wide array of odd sorting algorithms such as Bogobogo sort and Brick sort to see how much people have thought about the sorting problem. As a thought experiment, look back at the implementations of all these algorithms. How do you think they rank in terms of efficiency, that is, time and memory space? Which do you think is the fastest? Why? Read Chapter 2 in your textbook for more explanations. In your next lab exercise you will learn how to run experiments to compare different sorting algorithms.

6. Application of Sorting algorithms

In your text book (Section 2.5) a broad variety of sorting algorithms are being described. One interesting area is operation research (OR) where sorting algorithms come to a great help to build mathematical models to solve scheduling problems. As stated by your textbook a scheduling problem can be seen as having *N* jobs to complete, where job *j* requires t_j seconds of processing time. We need to complete all of the jobs but want to maximize customer satisfaction by minimizing the average completion time of the jobs. The *shortest processing time first* rule, where we schedule the jobs in increasing order of processing time, is known to accomplish this goal. Therefore we can sort the jobs by processing time or put them on a minimum-oriented priority queue.

Let's see how we can write a program that reads job names and processing times and prints a schedule that minimizes average completion time using the rule just described.

6.1 Creating Job ADT

We can approach this problem by coding a number of static methods to read the jobs, sort them based on the shortest processing time and print the schedule. But is this the best approach? From what you learn in Lab #2 we can extend the Java language programming power by user defined ADTs. For this problem is it a better approach if we create a Job ADT? What are the advantages of having the Job ADT? Of course if we want to reuse Job to solve other scheduling problems it makes more sense and efficient having Job ADT handy so we do not need to repeat the implementation again. In this case we know by experience that the shortest processing time first rule is just one way of tackling a scheduling problem among many. Therefore, it is very likely that we need the Job ADT again. So instead of writing static methods to solve this problem we start by creating a Job ADT.

Step 1

Recap from Lab #2, in the first step we should decide on APIs. What are the APIs that we need? At least we need the following APIs:

```
public int compareTo(Job that)
```
//compare processing times of tow jobs and returns 0 (equal), -1(this greater than that) and 1 (that greater than this)

```
public String toString()
```
//convert the job name and processing time to a String value

Step 2

We start implementing the Job ADT by deciding on instance variables. What information would be encapsulated for a job? Minimum information that we need for each instance is a job name and a job processing time. So we need two instance variables:

```
private  String jobName;
private  double processingTime;
```

Note that we made these variables private to preserve encapsulation. By doing this, what other APIs we need? Of course we want to be able to access the values of these variables. So we should have methods that return the instance variables:

```
public String getJobName(){}
public double getProcessingTime(){}
```

We should make another important design decision in this stage. Should we make the Job ADT mutable or immutable? We know that an immutable ADT is preferable as the scope of code that can change its values will be very limited. In this case, do we want to make the values of our ADT prone to change? The answer is **No** (as a practice problem, justify this answer). So we should make the following change to our instance variables:

```
private final String jobName;
private final double processingTime;
```

Step 3

What should I include in my constructor code? We know that a processing time cannot be negative. So I need to raise an exception as early as possible if this is the case (remember the fail fast rule from the lecture notes):

```
public Job(String jobName, double processingTime)
{
     if(processingTime<0)
          throw new IllegalArgumentException();
     this.jobName=jobName;
     this.processingTime=processingTime;
}
```

Step 4
My implementation of the Job ADT will look like the following

```
public class Job implements Comparable<Job>
{
	private final String jobName;
	private final double processingTime;

	public Job(String jobName, double processingTime)
	{
		if(processingTime<0)
			throw new IllegalArgumentException();
		this.jobName=jobName;
		this.processingTime=processingTime;
	}

	public String getJobName() {return jobName;}

	public double getProcessingTime() {return processingTime;}

	public int compareTo(Job that)
	{
		if(this.processingTime<that.processingTime) return -1;
		if(this.processingTime>that.processingTime) return 1;
		return 0;
	}

	public String toString()
	{
		return String.format("%s %.1f",jobName,processingTime);
	}
}
```

6.2 Client code to solve the problem

What should I include in my client code? Let's assume we are reading job names and job processing times from standard input. Now that we have the Job ADT we can read these values and store them as instances of our new Job ADT:

```
	public static void main(String[] args)
	{
		Scanner input=new Scanner(new
```

```
BufferedInputStream(System.in));
            PrintWriter output=new PrintWriter(new
OutputStreamWriter(System.out),true);
            int size=input.nextInt();
            Job[] jobs=new Job[size];
            for(int i=0;i<size;i++)
            {
                  String jobName=input.next();
                  double jobDuration=input.nextDouble();
                  jobs[i]=new Job(jobName,jobDuration);
            }
            Arrays.sort(jobs);
            output.println();
            output.println("sorted jobs");
            for(int i=0;i<size;i++)
                  output.print(jobs[i]+" ");
            output.println();
      }
```

The second part of my client code uses the system sort provided by Java. Do you know which sorting algorithm does the Java system implement and why? Can you replace the java system sort with your recently implemented bubble sort and selection sort? Which of these three implementations perform better in terms of running time?

7. Further Practice Problems

7.1 Use the newly created Job ADT and this time apply the *longest processing time first rule* to solve a load balancing scheduling problem (Exercise 2.5.13).

7.2 For further practice, here are some suggested questions from your textbook: 2.5.14, 2.5.16, 2.5.28