## 1. Introduction

In this lab activity, we'll work with three algorithms to sort data, to experiment with them, and measure their respective *average execution times*. In particular, we will work with the following sorting algorithms: *BubbleSort*, *SelectionSort,* and *InsertionSort*. These are three of the most common simple algorithms for sorting. Assuming *n* values to be sorted, the time complexities of these algorithms are as follows:

| Algorithm | Average Execution Time | Worst-case Execution Time |
|---|---|---|
| **Bubble Sort** | $O(n^2)$ | $O(n^2)$ |
| **Selection Sort** | $O(n^2)$ | $O(n^2)$ |
| **Insertion Sort** | $O(n^2)$ | $O(n^2)$ |

## 2. The Three Algorithms

The three algorithms that we study here are based on simple strategies that one would expect to be the first ideas that come to the mind of any person thinking in designing an algorithm for sorting.

The main idea of the Bubble Sort and the Selection Sort algorithms is the following. Assume that `arr` is the array to sort in non-decreasing order (we may simply say "increasing").

```
1. Repeat the following for i=n, n-1, ..., 2:
    a. Reorganize the first i elements in the array arr so that the maximum among them is
       placed in position i-1 of the array (the last position of the first i positions).
```

Note that what this scheme guarantees is that the maximum value among the first i position will end up at position i in the array on each iteration of loop in 1. It does not guarantee that all the positions will be sorted. But the execution of the loop, decreasing the value of i from n down to 2, trivially guarantees that, at the end, the array will be sorted in non-decreasing order.

**Bubble Sort**. What the Bubble Sort algorithm does to reorganize the first i elements is to test consecutive pairs of values in positions 0 and 1, 1 and 2, 2 and 3, and so on until i-2 and i-1. Whenever the first value in the pair is greater than the second value, a swapping operation for those two values is executed. This will guarantee that the maximum value, among those first i elements will be carried upward to end up at position i-1 of the array. Based on this, the above scheme is refined as follows:

```
1. Repeat the following for j=0, 1, ..., i-2
    a. if (arr[j] > arr[j+1])
           swapArrayElements(arr, j, j+1)     // interchanges values arr[j] and arr[j+1]
```

**Selection Sort.** The main idea that the Selection Sort uses to reorganize the first i values is the following:

```
1. Find mpi = the index of the position in arr that contains the largest value among the first i
   elements in arr.
2. If mpi != i-1
    a. Swap the two values at positions mpi and i-1 in arr.
```

**Insertion Sort.** The main idea of the Insertion Sort algorithm is the following:

```
1. Repeat the following for i=2, 3, ..., n:
    a. Given that elements in the first i-1 elements (those at positions 0..i-2) are sorted (increasing)
       whenever a new iteration of this loop begins, insert into them the i-th element (the one at position
       i-1), moving one element to the right as needed until the final position for that element is found
       with the final goal to have sorted the elements in the first i positions of the array; so, one more
```

position is guaranteed to be sorted after each iteration.

The above scheme is refined as follows:

```
1. Repeat for i=2, ..., n
   a. v = arr[i-1]   // value to insert
   b. j = i-2
   c. while j >= 0 and v < arr[j] do:   // looking for where to insert it
      i.  arr[j+1] = arr[j]
      ii. j--
   d. arr[j+1] = v      // once placed in arr[j+1], then elements in positions 0..i-1 are sorted
```

In this activity, we work with generic implementations of these three sorting algorithms, test them, and experiment with their execution times.

## 3. Implementation of the Three Sorting Algorithms

Let's work on the implementation of the three algorithms; then, we will work on the experimentation framework. We want to achieve code reusability as possible, and also to implement generic versions of these algorithms. For this, we will base the comparisons needed in objects that are **comparators**.

**NOTE**: Techniques similar to the ones used here shall be used in throughout the course to implement hierarchies of different implementations for some important types of collections.

The idea is to implement a hierarchy of classes in which the classes at the bottom correspond to objects that specialize in sorting an array. The top of the hierarchy is the following Java interface:

```
/**
  * This type of object specializes in sorting an array of elements of type E. The sorting shall be based on
  * an order relation given by a  comparator object. Sorts the elements of the array in non-decreasing
  * order as per the associated comparator.
  * @param <E> Generic type of elements
 */
public interface Sorter<E> {
    void sort(E[] arr, Comparator<E> cmp);    // to sort arr using comparator cmp
    String getName();                         // to get the name of the strategy
}
```

Some intermediate abstract classes are implemented to support better code-reusability. These are the following two classes:

```
public abstract class AbstractSorter<E> implements Sorter<E> {
   protected E[] arr;           // the array to sort
   protected Comparator<E> cmp; // the comparator to use
   private String name;         // name of the strategy

   protected static class DefaultComparator<E> implements Comparator<E> {
      public int compare(E o1, E o2) {
         try {
            return ((Comparable<E>) o1).compareTo(o2);
         } catch (ClassCastException e) {
           throw new IllegalArgumentException("Data type is not Comparable");
         }
      }
   }
   public AbstractSorter(String name) {
      this.name = name;
   }
   /** When this method is applied to the Sorter object, it will initiate
    * the sorting process for the specified array using the given comparator.
    * @param arr The array to sort.
    * @param cmp The comparator to use. If null, then the default comparator
    * is used.
    */
   public void sort(E[] arr, Comparator<E> cmp) {
      this.arr = arr;
      this.cmp = (cmp != null ? cmp : new DefaultComparator<E>());
      auxSort();      // this method implements the final algorithm to sort in corresponding subclass
   }
   public String getName() {
      return name;
   }
   protected void swapArrayElements(int i, int j) {
      // pre: 0 <= i, j < arr.length;
      E temp = arr[i];  arr[i] = arr[j]; arr[j] = temp;
   }
   /** This method needs to be implemented in each of the subclasses
    * classes implementing the particular sorting algorithm.
    */
   protected abstract void auxSort();
}
```

```
/** This is the common super class for the two sorter types that correspond
 * to the algorithms Bubble Sort and Selection Sort. Both are based on
 * successfully finding max values and relocating them in the last position
 * of the portion of the array that remains to be sorted.
 */
public abstract class AbstractMaxValueSorter<E> extends AbstractSorter<E> {
   public AbstractMaxValueSorter(String name) {
      super(name);
   }
   protected void auxSort() {     // From the superclass
      for (int i = arr.length; i>1; i--)
         relocateMaxValueToLastPositionAmongFirst(i);  //
   }
   /** This method needs to be implemented in each of the two classes
    * corresponding to Bubble Sort and Selection Sort sorters.
    */
   protected abstract void relocateMaxValueToLastPositionAmongFirst(int i);
}
```

The classes at the bottom implement the previous three algorithms;these are the classes:

1. `BubbleSortSorter<E> extends AbstractMaxValueSorter<E>`
2. `SelectionSortSorter<E> extends AbstractMaxValueSorter<E>`
3. `InsertionSortSorter<E> extends AbstractSorter<E>`

## 4. Experimentation Part

The code includes an experimentation component to experiment with strategies to sort that are implemented as subclasses of type `Sorter<E>`. The goal is to have a component that facilitates the process of adding a new sorting strategy to experiment with. In those experiments, each strategy is run to sort randomly generated data of integers. The main classes for this component are inside the package `experimentalClasses`. There, you can find two classes:

| Class | Description |
|---|---|
| `StrategiesTimeCollection` | An object of this type is associated with a `Sorter<Integer>` object, which corresponds to a sorting strategy to experiment with. For each such strategy that you want to add to the experimentation, you need to have one object of this type. See the constructor. When applied to an object of this type, the method `runTrial(arr)` executes the associated sorting strategie on the data of array `arr`. |
| `ExperimentController` | An object of this type is the one that runs the experimentation. Study the constructor, which sets the parameters for the experiments. See the method `addStrategy(...)`, which adds to the controller a sorting strategy already wrapped in an object of type `StrategiesTimeCollection`. The controller holds all the strategies to be tested in an internal list (`resultsPerStrategy`). |

To add a sorting strategy to experiment with using this component, that strategy needs to be implemented using a class that implements `Sorter<E>`, as a subclass of `AbstractSorter<E>` class in the hierarchy of classes included. Then, to add the strategy to the experimentations, you need to do the following:

1. Create an instance of the class corresponding to the particular sorting strategy.
2. Wrap that instance inside an object of type `StrategiesTimeCollection`.
3. Add this last object to the controller object.

You can see this in action in lines of code 36-38 inside class `EperimentaionMain`, which is in package `experimentalMainClasses`. Note the three steps listed above are done at once, when added to the controller object (variable `ec` in that code).

## 5. Exercises

Download the partial project included in the zip file and import it to Eclipse. If errors are shown, verify that the Java version you are using is 1.7 or 1.8. The project includes several packages and classes. Inside package sorterClasses you will find several classes implementing the previously discussed strategies.

Also, inside the sorterTesterClasses package, you will see two comparator classes, in addition to the class that tests all the three strategies together. For each possible comparator, the two comparator classes in that package and the default comparator class (implemented as a protected static class inside class `AbstractSorter`), each strategy is tested with randomly generated samples of sizes 1, 6, 11, and 16. The same generated data for each of those sizes is used for each comparator and strategy.

For the sorting algorithms, and how their implementation is done, you should study the strategy used. You should also do that for the experimentation part.

**Exercise 1.** Study how the comparator is used and how one is assigned to a sorter object. Notice that the default comparator works under the premise that the data type of the elements is a subtype of `Comparable<E>`. If that is not the case, then it will eventually throw an `IllegalArgumentException`. You should study where this happens in the implementation. We shall use similar techniques in the implementation of some collections that will be studied later in this course.

To test this, add the following class inside that tester package:

```
public class Entero implements Comparable<Entero> {
   private int value;
   public Entero(int v) { value = v; }
   public int getValue() { return value; }
   public String toString() { return value + ""; }
   public int compareTo(Entero other) {
      if (other == null) throw new IllegalArgumentException("...");
      return this.value - other.value;
   }
}
```

Then write a tester method, which uses one of the sorter objects to sort an array of type `Entero[]` and using the *default comparator*. You should study the code to derive how the sorter object is created and how the default comparator is used when the sort method is invoked on that sorter object. When you run that tester, you should get an exception, for which part of the output trace has the following initial lines:

```
Exception in thread "main" java.lang.IllegalArgumentException: Instantiated data type must be Comparable
      at sorterClasses.AbstractSorter$DefaultComparator.compare(AbstractSorter.java:17)
          ...
```

The rest of the lines will be more specific to which sorter you decide to use.

**Exercise 2.** How the same sorting algorithm can sort in increasing/decreasing order by just changing the comparator.

To see how this work, study the two comparators for integers that have been included and how they are being used in the tester class included. To have a practice with it, do the following:

Write a tester class that sorts the array: {5, 9, 20, 22, 20, 5, 4, 13, 17, 8, 22, 1, 3, 7, 11, 9, 10} in both, decreasing and in increasing order. Your class should only initialize the array of type `Integer[]` with the given values; then create the sorter object (you can choose the one you like, or try the three); then apply the sort method to the sorter object, giving the array and the corresponding comparator as parameters. If the sort is done twice, each time with a different comparator (of the given two), the array should be sorted in increasing and in decreasing order. Don't forget to include the code to display the content of the array: original, plus the two sorted versions. See tester included.

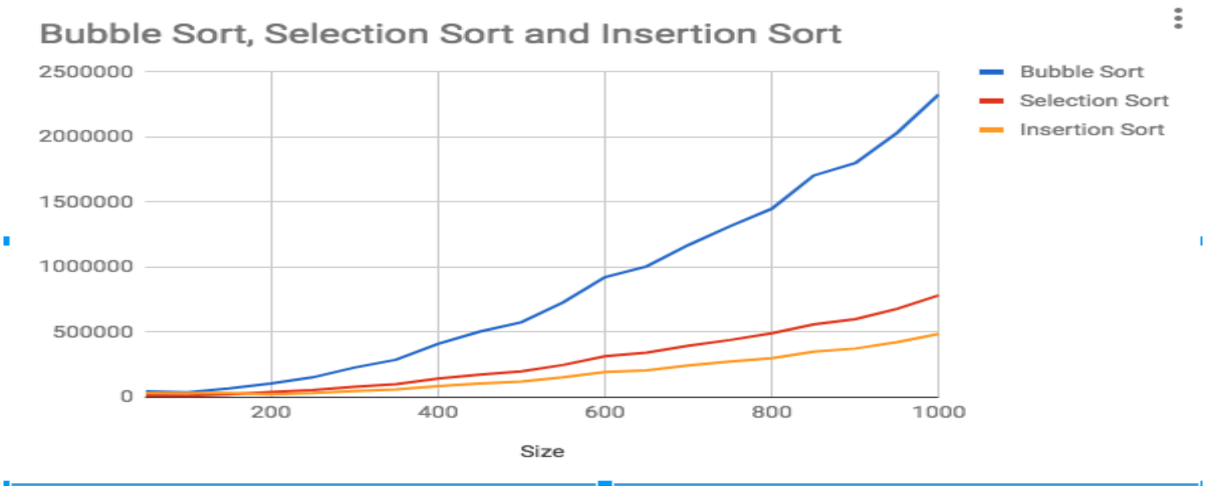
**Exercise 3.** Now run the experimentation code inside the package `experimentalClasse`; in particular, the class `ExperimentationMain`. It should produce average execution times for each of the three sorting algorithms applied to different sizes of the input dataset. The sizes being considered are 50, 100, 150, ..., 1000. For each size, each strategy is run 200 times. These are parameters used when creating object of type ExperimentController, which is in charge of controlling the experimentation process. If everything is correct, the execution will run experimental trials on all the three algorithms to sort. Results are placed inside subdirectory experimentalResults in a file named `allResults.txt`. Note: **The directory experimentalResults needs to be created manually**, since the code is not creating it automatically if it does not exist; or modify code to do it.

Here, you should play executing it from the command prompt. Also, you can change the default value of the experimental parameters when typing the `java` command to execute the class. There are at most four only.

**Exercise 4.** Upload the experimental results in file `allResults.txt` to Excel or Google Sheet. Remember that values of size and experimental time on each line in the corresponding files are separated by TAB characters. Once loaded, you should see something similar to this:

| Size | Bubble Sort | Selection Sort | Insertion Sort |
|---|---|---|---|
| 50 | 45764.19 | 15322.61 | 32642.67 |
| 100 | 37120.67 | 11914.495 | 32720.426 |
| 150 | 66727.53 | 23352.855 | 31078.096 |
| ... | ... | ... | .. |

Once uploaded, use the chart utility to graph the measures of times (y-axis) versus size (x-axis) for each of the strategies. You should get something similar to the following:

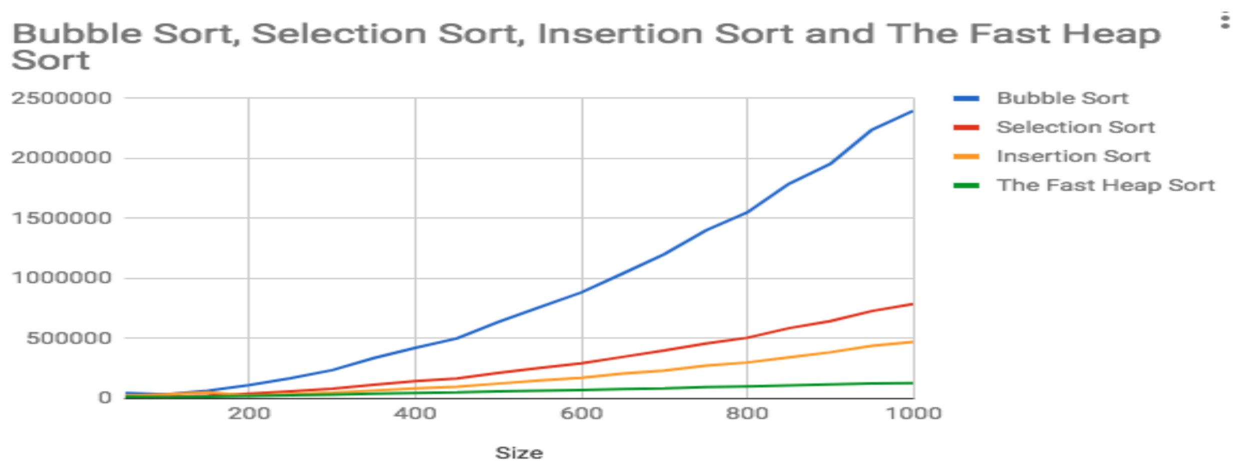


**Exercise 5.** Let's go through the process of adding a new sorting strategy to test with. This is a fast sorting algorithm that we shall study later in the course; for the moment, just use as given. Its implementation as a `Sorter<E>` subclass has been uploaded to the shared folder. See class `HeapSortSorter`[1] at the end of this document. Paste its content as the content of a new class to be named `HeapSortSorter<E>` inside the package `sorterClasses`. **Do not alter the code being pasted.** Once there, you can add to the `IntegerSorterTester` class if you wish. But we are now interested in the experimentation part. To do that, you just need to do the following. Inside the main method in class `ExperimentationMain`, insert the following line after current line 38. Eclipse will ask to add the appropriate import; just click as requested, and done.

```
ec.addStrategy(new StrategiesTimeCollection<Integer>(new HeapSortSorter<Integer>()));
```

After that is done, run class `ExperimentationMain` again, and wait for it to finish. Then upload the generated data to a Google sheet and create graphs. You should get something like the following:

[1] This strategy is based on the **Heapsort algorithm**. For the moment, just use it as given. We don't discuss the details of where this algorithm from, or about its properties, because we first need to study some other data structures (complete binary trees and the heap) in order to understand it; and we shall do that later in the course. But this is a very fast algorithm to sort; and you can see that in the graph included. Its execution time is O(nlog$_2$n).



**HeapSort**, a really fast algorithm. You will learn it later in the course; and also some other faster algorithms.

Show your results to your lab instructor.

```
public class HeapSortSorter<E> extends AbstractSorter<E> {
        public HeapSortSorter() {
                super("The Fast Heap Sort");
        }
        protected void auxSort() {
                // convert to heap
                int n = arr.length;
                for (int r = (n-2)/2; r>=0; r--)
                        downHeap(r, n);
                for (int i = n-1; i>0; i--) {
                        swapArrayElements(i, 0);
                        downHeap(0, i);
                }
        }
        private int left(int r) { return 2*r+1; }
        private int right(int r) { return 2*r+2; }
        private boolean hasLeft(int r, int n) { return left(r) < n; }
        private boolean hasRight(int r, int n) { return right(r) < n; }
        private void downHeap(int r, int n) {
                // r is a root of a subtree in the complete tree formed
                // by the first n elements in dataSet --- 0..n-1
                boolean isHeap = false;
                while (!isHeap && hasLeft(r, n)) {
                        int mci = left(r);
                        if (hasRight(r, n) && -cmp.compare(arr[right(r)], arr[mci]) < 0)
                                mci = right(r);
                        if (-cmp.compare(arr[mci], arr[r]) < 0) {
                                swapArrayElements(r, mci);
                                r = mci;
                        }
                        else
                                isHeap = true;
                }
        }
}
```