University of Puerto Rico – Mayagüez
Department of Computer Science and Engineering

CIIC 4020 / ICOM 4035 – Data Structures
Prof. Juan O. López Gerena
Spring 2019-2020
Laboratory #5 – Sorting and Experimentation

In this lab activity, we'll work with different 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: *Selection Sort*, *Bubble Sort*, *Insertion Sort*, *Merge Sort*, *Quicksort*, and *Heap Sort*. The first three are probably the most common simple algorithms for sorting, the fourth and fifth are based on divide and conquer, and the last one is based on the heap; note that these are all *comparison-based* sorting algorithms. Assuming *n* values are to be sorted, the time complexities of these algorithms are as follows:

Algorithm	Average Running Time	Worst-case Running Time
Selection Sort	$O(n^2)$	$O(n^2)$
Bubble Sort	$O(n^2)$	$O(n^2)$
Insertion Sort	$O(n^2)$	$O(n^2)$
Merge Sort	$O(n\cdot\log n)$	$O(n\cdot\log n)$
Quicksort	$O(n\cdot\log n)$	$O(n^2)$
Heap Sort	$O(n\cdot\log n)$	$O(n\cdot\log n)$

We'll work on finishing the implementation for most of these algorithms, using pseudocode provided in this document and/or in class. To try to achieve as much code reusability as possible, not only will we be implementing generic versions of these algorithms (to be able to sort different data types), but we'll also use **comparator** objects to perform the comparisons needed. Once the algorithms have been implemented *and tested*, we'll analyze their running time by performing sorting experiments using an experimentation framework that has been provided to you.

O(n^2) sorting algorithms

First we'll work on the simpler algorithms, Selection Sort, Bubble Sort, and Insertion Sort, all of which an average running time of O(n^2).

Selection Sort

In each iteration of Selection Sort, we *select* the smallest value of the unsorted portion of the array and then swap it into the first index of the unsorted portion.

```
for i = 0 to n – 2
    # i represents beginning of unsorted portion
    sm ← index of smallest value from A[i] to A[n−1]
    swap A[sm] with A[i]
```

Bubble Sort

Bubble Sort iterates through the array, up to the last position of the unsorted portion, comparing adjacent values and swapping them if they are out of order. Once the end has been reached, if any swaps were performed in that iteration, then perform another iteration.

```
lastUnsorted ← n – 1
do
    swapped = false
    for i = 1 to lastUnsorted
          if A[i – 1] > A[i]
                swap A[i – 1] with A[i]
                swapped = true
    lastUnsorted ← lastUnsorted – 1
while swapped
```

Insertion Sort

The idea behind Insertion Sort is that we have a left-most portion of the array that is sorted (within itself) and the rest of the array is unsorted. We insert the first value in the unsorted portion of the array into its rightful position within the sorted portion, by shifting elements to make room for this new value.

```
for i = 1 to n – 1
    # i represents beginning of unsorted portion
    valToInsert ← A[i]
    j ← i – 1
    while j ≥ 0 and A[j] > valToInsert
          # Keep moving values to the right, making room for valToInsert
          A[j + 1] ← A[j]
          j ← j – 1
    A[j + 1] ← valToInsert
```

O(n·log n) sorting algorithms

Divide and conquer

Two fast algorithms to sort are the *Merge Sort* and the *Quicksort* algorithms. They are both based on the *divide-and-conquer strategy*. Such strategy can be applied to several types of problems to derive algorithms, not just sorting. It is based on partitioning the input into different parts, finding a solution for each one of those parts, and combining the partial solutions to construct a solution to the problem over the whole input. More formally:

```
DCSolution(Input S):
    if (size of S > BASE_SIZE)
        a. partition S into $S_1$, $S_2$, ..., $S_n$
        b. recursively apply DCSolution to $S_1$, $S_2$, ..., $S_n$ to obtain a solution for each of them:
           DCSolution ($S_1$), DCSolution ($S_2$), ..., DCSolution ($S_n$)
        c. properly combine the partial solutions to construct a solution for S
    else
        solve the problem for the base case
```

In the case of sorting, the Quicksort and Merge Sort strategies divide the input in two portions whenever the size of the input is greater than 1 (the BASE_SIZE). The solution for the base case (size ≤ 1) is to return the input as given.

In the case of Merge Sort, the input would be partitioned into two portions. Technically, Quicksort partitions into three portions, but the middle portion always consists of only one element that doesn't need to be moved because it's already in its final position, so we can consider Quicksort to also have two portions in its partition. We can then adapt the divide-and-conquer pattern to be more specific for these sorting algorithms:

```
DCSort(List S):
    if (size of S > 1)
        d. partition S into $S_1$ and $S_2$
        e. recursively apply DCSort to $S_1$ and $S_2$ to sort each of them: DCSort ($S_1$), DCSort ($S_2$)
        f. properly combine $S_1$ and $S_2$ to reconstruct S completely sorted
    else
        nothing needs to be done since, if size ≤ 1, then it is already sorted
```

Key assumptions

In the discussion that follows, we assume that:

- The list to sort is given as an `ArrayList` object, representing a collection of objects of a generic type `E`. Note that this could easily be converted to the case of an array or any other similar list structure.
- Every partition will consist of a contiguous portion of the list, defined by the indexes of its first and last position, which we'll refer to as *`first`* and *`last`*, respectively.
- The order relation upon which the final order of elements in the list will be based on is given by a comparator object of type `Comparator<E>`, which is represented by the variable *`cmp`* in the partial code that is shown.

NOTE: Algorithms or pseudo-code presented here might differ slightly from those presented in lectures. However, they are based on the same ideas.

Merge Sort

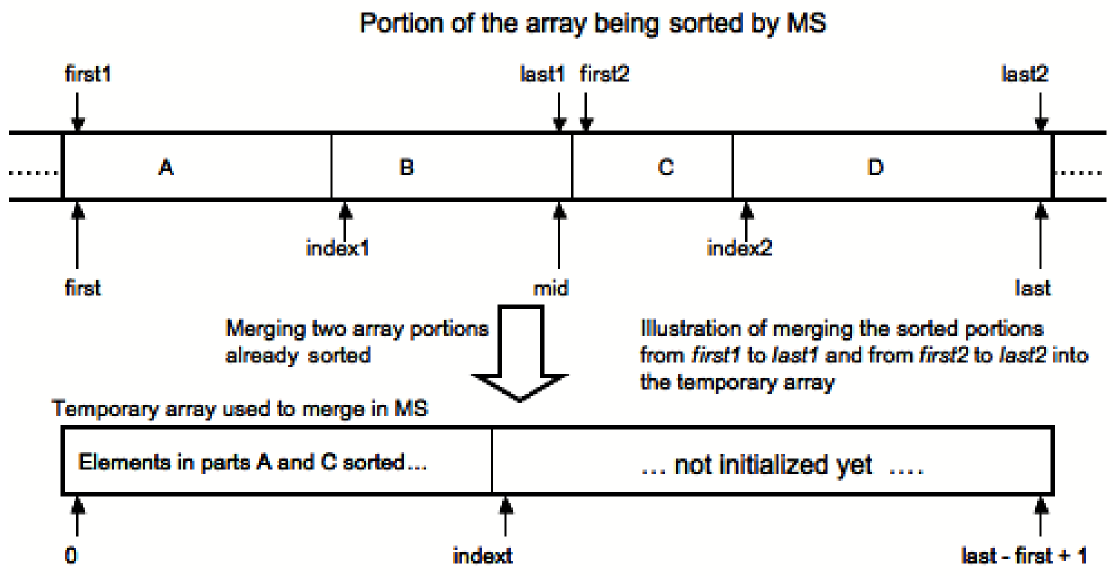
If the portion of the list being considered consists of at least two elements (two consecutive positions in the ArrayList object), then the algorithm:

1. Partitions the portion being sorted in two halves, or as close as possible.
2. Sorts each half recursively.
3. Merges the two sorted halves into the final sorted list.

The partitioning process is simple, just determine the index corresponding to the middle of the list portion using the following calculation: `middle = (first + last) / 2`. The first element of the partition is then the list portion from *`first`* up to (and including) *`middle`*. The second element of the partition is the list portion from *`middle`* `+ 1` up to (and including) *`last`*.

The next figure summarizes the idea behind the merging operation of two halves of the list, which are assumed to be already sorted. Note that these two halves are currently stored in the same list that must later contain the final sorted values, so the two halves are merged into a temporary array, which is then copied back to the list. The figure shows the state of the list at some point during the merging process. The first half of the list consists of the union of parts A and B in the figure, and is assumed to be sorted. The same is assumed for the second half of the list, consisting of the union of parts C and D. The algorithm creates a new array where the merged elements are temporarily placed. The size of that temporary array must be at least equal to the size of parts A, B, C, and D combined; this is precisely the value of (`last-first+1`) since those parts correspond to the portion of the list from position *`first`* to position *`last`*. A specific algorithm is shown in the method **`merge`** that follows the figure. The figure shows a snapshot of the first while-loop of the algorithm at a point in the merging process in which portions A and C have already been processed (parts B and D are yet to be processed). As they were processed, the elements were copied to the temporary array, one by one, guided by the if-statement inside the loop body. At the moment shown, elements in A and C have been properly merged, and placed in order in the first *`indext`* positions of the temporary array (note that `indext = index1 + index2`). The loop continues until one of the portions being merged is completely processed. At that moment, the first while-loop ends and the algorithm just needs to verify which half still has elements remaining to be processed. That remaining portion is “appended”, as it is, to the temporary array.

Once the merging into the temporary array is completed, its content is copied back to the portion of the list being sorted. Notice that this implies that, after the method is executed, and assuming that the preconditions are met, that particular portion of the list (from *`first`* to *`last`*) will be in sorted order.


Portion of the array being sorted by MS
first1
last1
first2
last2
A
B
C
D
index1
index2
first
mid
last
Merging two array portions already sorted
Illustration of merging the sorted portions from first1 to last1 and from first2 to last2 into the temporary array
Temporary array used to merge in MS
Elements in parts A and C sorted...
... not initialized yet ....
0
indext
last - first + 1

Method **merge** works on the following two halves of the list:

- First half: from position **first** to position **mid**
- Second half: from position **mid+1** to position **last**

It is assumed that both halves are already sorted.

```
private void merge(ArrayList<E> list, int first, int mid, int last) {
     E[] tempList = (E[]) new Object[last-first+1];
     int index1 = first, last1 = mid;  // Start and end of first  sorted half
     int index2 = mid+1, last2 = last; // Start and end of second sorted half
     int indexTL = 0; // index of tempList
     while (index1 <= last1 && index2 <= last2)
         if (cmp.compare(list.get(index1), list.get(index2)) <=0)
             tempList[indexTL++] = list.get(index1++);
         else
             tempList[indexTL++] = list.get(index2++);

     /* move the remaining data to tempList -- notice that only one
        of the following loops will iterate at least once */
     while (index1 <= last1)
         tempList[indexTL++] = list.get(index1++);
     while (index2 <= last2)
         tempList[indexTL++] = list.get(index2++);

     // put sorted data back to the list portion....
     for (int i=0; i<tempList.length; i++)
        list.set(first+i, tempList[i]);
}
```

As you can see, partitioning of the list is a simple task; the hard work in Merge Sort is done by the **merge** method.

Quicksort

This algorithm puts all its effort into the partitioning operation. To partition a contiguous portion of the list, from positions `first` to `last`, it partially reorganizes the list, and finally returns a value, say *p*, in the range from `first` to `last`. Assuming that the portion being considered has at least two elements (i.e. `first` **<** `last`), the partial reorganization carried out, and the final value of *p*, are guaranteed to comply with the following:

- `list.get(i) ≤ list.get(p)` for *i* = `first` … *p*-1
- `list.get(i) ≥ list.get(p)` for *i* = *p*+1 … `last`

Hence, the partition separates the list into at most three lists portions: a first portion (possibly empty) from position `first` up to position *p*-1, another with just one element (the **pivot**) at position *p*, and a third portion (possibly empty) consisting of positions *p*+1 up to position `last`.

The properties discussed above guarantee the following three important results:

a. The element finally left in position *p* of the list, does not need to be moved from there later on. It already is in the position it has to be once the list is sorted.
b. To complete the sorting process, elements in the first portion (if not empty) do not need to be moved outside that portion of the list where they now are. The same thing happens with the elements in the third portion (if not empty).

The following method implements one possible partitioning strategy.

```
private int partition(ArrayList<E> list, int first, int last) {
   int left = first, right = last-1;
   E pivot = list.get(last);   // using the last element as the pivot

   while (left <= right)  {
      while (left <= right && cmp.compare(list.get(left), pivot) <= 0)
            left++;

      while (left <= right && cmp.compare(list.get(right), pivot) >= 0)
            right--;

      if (left < right) {
         //swap list positions: left and right
         swapListElements(list, left++, right--);
     }
   }

   // swap list positions: left and last
   // the pivot is placed permanently at position left, its rightful position
   swapListElements(list, left, last);

   return left; // Must return the position of the pivot
}
```

The next figure illustrates the behavior of the outer while-loop in the algorithm. When both inner while loops are completed on any particular iteration of the outer loop (right before the if-statement) then the following is satisfied: *y* > *x* and *z* < *x*. The body of the if-statement that follows swaps these two values. The idea is to keep in the left part of the list (from position **`first`**) only values that are less than or equal to *x*, while keeping on the right part only values that are greater than or equal to *x*. As the outer loop iterates, the values of variables **`left`** (initialized to **`first`**) and **`right`** (initialized to **`last-1`**) move towards each other, iterating until **`right`** < **`left`**. When this happens, then the process is completed, and at that moment the algorithm just needs to swap the values in positions **`left`** and **`last`** (which is the position of the **pivot**). Finally, the value that is to be returned by the method (referred to as *p* in our earlier discussion) is determined as the value of **`left`** at that moment.

Portion of the array being sorted by QS

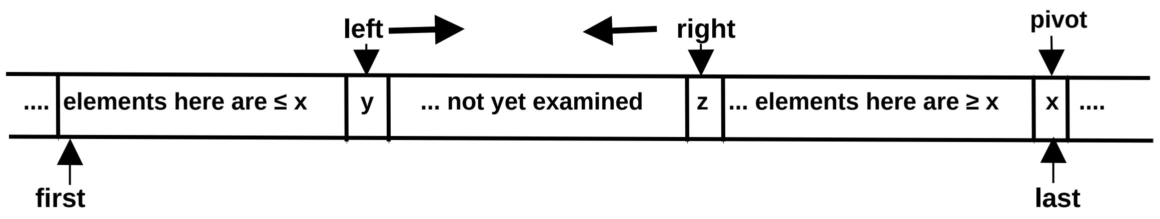


Illustration of partition process in list portion from first to last.

Heap Sort

The Heap Sort algorithm is based on the heap data structure, which will be covered in future lectures. At this point it’s not important for you to know anything about the heap data structure, or the Heap Sort algorithm, except that its running time is always O(n·log n), so it’s a good sort to include in our experiments so that we may compare its results against Merge Sort and Quicksort.

Exercises

Download the partial project included in the zip file and import it into Eclipse. Inside the package `strategiesClasses` you’ll find several classes implementing (or attempting to implement) the aforementioned sorting algorithms. You’ll notice that the `AbstractSortingStrategy` class is **not** the root class, since it extends the `AbstractStrategyToTest` class. At the end of this document we briefly present how we can create a generic experimentation framework by abstracting the strategies that are to be tested. However, that is not really part of this activity, it is included just for you to see that the same experimental code (classes inside package `experimentClasses`) can be used to experiment with different strategies for which appropriate classes may be implemented. *You are not expected to make changes within the `experimentClasses` package*. However, feel free to study them, so that you can see how they are structured and implemented; it would be a good learning activity.

Also, inside the `testerClasses` package, you will see tester classes for each one of the strategies. You should use them after finishing the corresponding implementation.

Notice that all of these sorting strategies are subclasses of the `AbstractSortingStrategy` class. The constructor for this type of object receives, as an explicit parameter, the comparator object to be used in the sorting process. Each such class implements its corresponding sorting algorithm inside the method that has the following signature:

`public void sortList(ArrayList<E> dataSet)` - Initiates the execution of the particular sorting algorithm on the elements of list **`dataSet`**. The different comparisons performed by the algorithm are based on the comparator that is assigned during the creation of the particular sorting object.

Exercise 1

Finish the implementation of the **Selection Sort** algorithm. Test your implementation by executing the `SelectionSortTester` program and analyzing the results.

Exercise 2

Finish the implementation of the **Bubble Sort** algorithm. Test your implementation by executing the `BubbleSortTester` program and analyzing the results.

Exercise 3

Finish the implementation of the **Insertion Sort** algorithm. Test your implementation by executing the `InsertionSortTester` program and analyzing the results.

Exercise 4

If you run the `MergeSortTester` program, you will notice that the data is left in the same order as it was generated; it is not sorted. That's because the strategy is *partially* implemented, but there's a part that is missing. You must fully implement the **`ms`** method of the `MergeSort` class, based on the previous discussion about the Merge Sort algorithm and the other methods that are already implemented in this class. Once you've finished your implementation, run the tester again. If correct, you should see that the data is sorted in non-decreasing order.

Exercise 5

Repeat the above exercise, but this time for the `QuickSort` class. Add the missing parts to the **`qs`** method based on the previous discussion about the Quicksort algorithm and the other methods that are already implemented in this class. Once you've finished your implementation, run the tester again. If correct, you should see that the data is sorted in non-decreasing order.

Exercise 6

In the above exercises all of the sorting objects were created using `IntegerComparator1`. Open the `IntegerComparator1` and `IntegerComparator2` classes; can you notice the difference between them? Change the five testers and run them again, but this time use `IntegerComparator2` instead. Are the results as expected?

Exercise 7

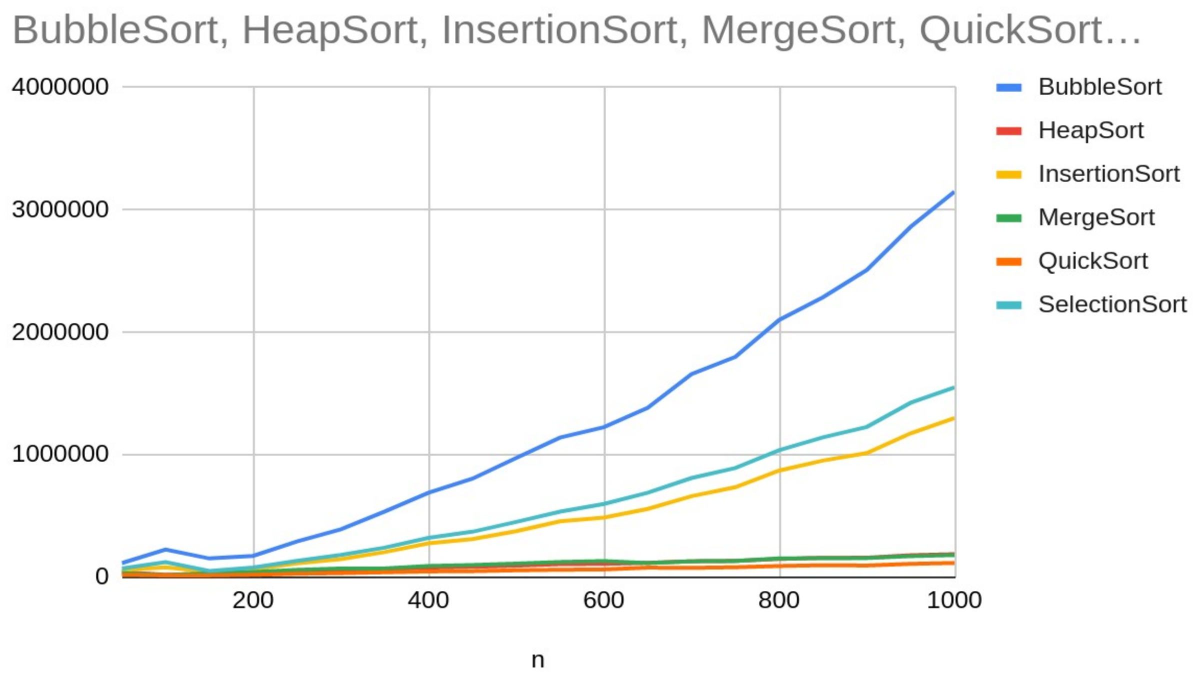
Now run the experimentation code in class `ExperimentalTrials` as it is. It should produce average execution times for each of the six sorting algorithms applied to different sizes of the input *`dataset`*. The sizes being considered are 50, 100, 150, …, 1000. For each size, each strategy is run 200 times (these are parameters used when creating object `ExperimentController`, which is in charge of controlling the experimentation process. See section **Experimentation Part** at the end of this document). If everything is correct, the execution will run experimental trials on all of the six algorithms to sort. Results are placed inside the subdirectory `experimentalResults` in separate files, each having the name of the particular strategy it corresponds to as part of the filename.

Exercise 8

Upload the experimental results for each sorting algorithm to Excel or Google Sheets. Remember that the values of size and experimental time on each line in the corresponding files are separated by TAB characters. Arrange the results in columns in the following order:

n	SelectionSort	BubbleSort	InsertionSort	HeapSort	QuickSort	MergeSort
50	71605.512	73581.586	50017.344	37101.105	10588.635	16852.46
100	32456.879	34738.4	31673.46	10884.155	20051.79	12398.385
…..		…	…	…	…	…

Then, use the Chart utility to graph the size versus times for each of the strategies on a single plot. You should get something similar to the following (consistent with the theoretical results displayed at the beginning of this document):



The chart you generate must be included with your lab submission!

<u>This is the end of the laboratory activity; the section below is completely optional to read.</u>

A framework for experimentation and measuring execution time

The zip file contains a partial implementation of a general framework aimed to facilitate experimentation with different solutions to particular problems. It has been implemented with the goal of achieving code reusability. For example, it can be used to test solutions not only for the problem of sorting, for also for the problem of counting frequencies (**F**requency **D**istribution), as suggested by the hierarchy diagram below:

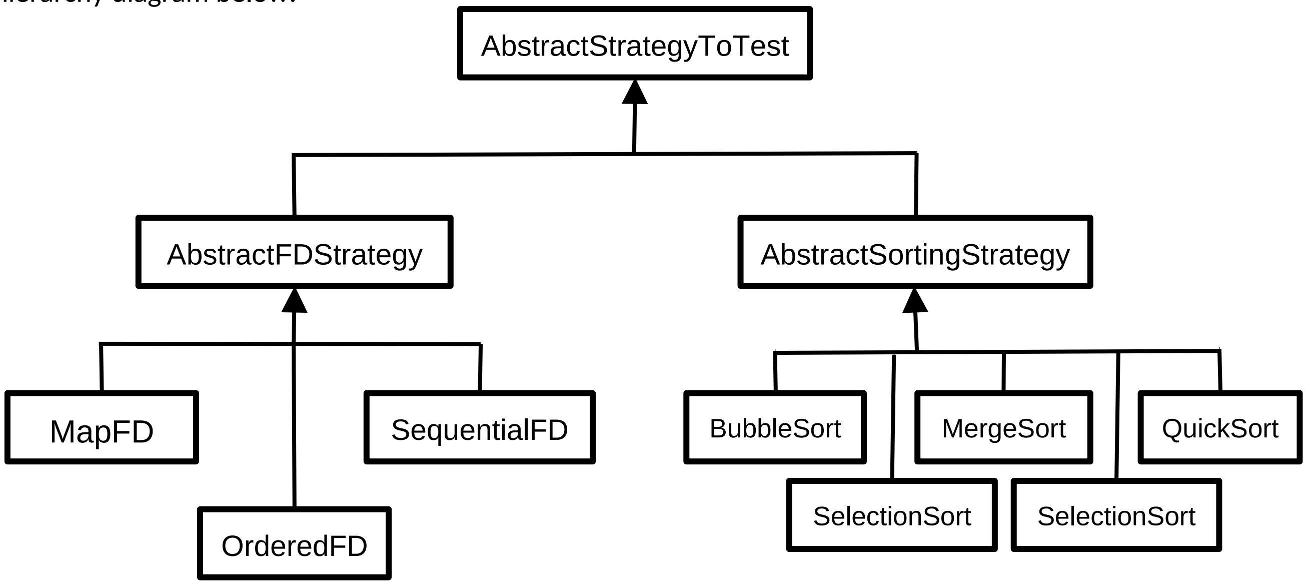


Figure 1. Hierarchy of classes that allow using the same experimentation code to experiment with different strategies for different problems; in particular, for frequency distribution and sorting.

Figure 1 shows a hierarchy of classes extending a general class named `AbstractStrategyToTest`. The experimental framework implemented here (as part of the partial project in the zip file)[1] requires that any strategy to be tested must be implemented as a subclass of the class on top of the diagram, perhaps with some other intermediate classes. In particular, the hierarchy depicted includes classes that implement strategies to solve two different problems: finding the frequency of objects in a data set and sorting a data set. If you want to test strategies for other problems, then the appropriate subclasses need to be implemented for those. Notice that the final classes (the leaves of the hierarchy) corresponding to strategies for frequency distribution need to extend the intermediate subclass `AbstractFDStrategy`. Similarly, those for the sorting problem need to extend the intermediate subclass `AbstractSortingStrategy`. In the package `strategiesClasses` you can see that these intermediate classes are used to normalize how the particular underlined strategy will be executed. Each

1 We're not claiming that this is the best approach for this generic experimentation framework, but it is good enough for our purposes, and allows code reuse for the two problems: frequency counting and sorting. Hence, it might be useful for you to use on other instances in which you might need similar experimentations. One immediate deficiency of the approach being used is the need to implement each strategy as a subclass in the shown hierarchy. This dependency is not good in general (it forces you to implement a strategy thinking in experimentation), but for the purpose of the current lab activity we will use it as it is. You may want to think of better alternatives to deal with the mentioned deficiency. One alternative would be that instead of inheritance, we use composition… But as stated before, we will work with what we have.

must implement the abstract method **`experimentallyExecuteStrategy`** that is part of the abstract class at the top of the previous hierarchy (`AbstractStrategyToTest`).

In the case of the *sorting problem*, in class `AbstractSortingStrategy`, that normalization is achieved as follows:

```
public void experimentallyExecuteStrategy(ArrayList<E> dataSet) {
      sortList(dataSet);
}
```

Then, any subclass corresponding to a strategy to solve the sorting problem would need to implement its algorithm as part of method named **`sortList`**.

For the case of the *frequency distribution problem*, we would have a class `AbstractFDStrategy` that would achieve that normalization as follows:

```
public void experimentallyExecuteStrategy(ArrayList<E> dataSet) {
      computeFDList(dataSet);
}
```

Then, any subclass corresponding to a strategy to solve the frequency distribution problem would need to implement its algorithm as part of a method named **`computeFDList`**.
Note: The `AbstractFDStrategy` class and the corresponding strategies were removed from the zip file to avoid confusion, but hopefully their purpose has been made clear.

Experimentation Part

The experimentation code is executed by an object of type `ExperimentController`, which is one of the classes included inside package `experimentClasses`. This class defines a data type corresponding to an object that can execute and measure execution times of different algorithms to solve particular problems in which the input can be represented by a list (we use `ArrayList` in this case). An object of this type can hold several strategies at the same time, as long as those strategies are all implemented as subclasses of the `AbstractStrategyToTest` class and embedded as part of an object of type `StrategiesTimeCollection`. The approach of the experimentation is the following (see **`run()`** in class `ExperimentController`)

```
for each size n to be considered do
    a. repeat the following m times (where m is the number of trials for each size;
       repetitionsPerSize)
        i.   generate a data set of n random integers: ds
        ii.  for each strategy s to test do
               1. make copy of ds: dsc
               2. execute s.experimentallyExecuteStrategy(dsc) and
                  measure its execution time
               3. accumulate execution time for strategy s
    b. for each strategy s compute the average execution time obtained for size n (the
       accumulated execution times divided by m)
```