1. **Introduction**

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: *HeapSort*, *MergeSort*, *QuickSort*, *BubbleSort*, and *SelectionSort*. The first is based on the heap, the second and third are based on divide and conquer, and the last two are two 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** |  |  |
| **Selection Sort** |  |  |
| **Heap Sort** |  |  |
| **Merge Sort** |  |  |
| **Quick Sort** |  |  |

1. **Divide and Conquer Strategy Applied to Sorting**

Two fast algorithms to sort are the *MergeSort* 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 to sorting. It is based on partitioning the input into different parts, finding a solution on 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)   1. if (size of S > BASE\_SIZE)    1. partition S into S1, S2, …, Sn[[1]](#footnote-0)    2. recursively apply *DCSolution* to S1, S2, …, Sn to obtain a solution for each of them: *DCSolution* (S1), *DCSolution* (S2), …, *DCSolution* (Sn)    3. properly combine those partial solutions to construct a solution for S   else  solve the problem for the base case   1. end |

In the case of sorting, the quicksort and mergesort strategies divide the input in two parts 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.

**More Details about MergeSort and QuickSort**  We can adapt the divide and conquer pattern shown to the problem of sorting a list of values as follows:

|  |
| --- |
| *DCSort*(List S)   1. if (size of S > BASE\_SIZE)    1. partition S into S1, S2, S3    2. recursively apply *DCSort* to S1, S2, S3 to sort each of them    3. properly combine the S1, S2, and S3, to reconstruct S completely sorted   else  nothing needs to be done since, if size <= 1, then it is already sorted   1. end |

The partition has been generalized to three sublists, but, in the case of the Mergesort, one of them is always empty (say S3), whereas in the case of the Quicksort (as will be implemented here), one of them has only one element. Moreover, in the Quicksort, the one containing just one element does not need to be combined, because it is already moved to a position in S where it can stay when the list is finally sorted: it does not need to be moved anymore. Hence the previous pattern can be further adapted as follows:

|  |
| --- |
| *DCSort*(List S)   1. if (size of S > BASE\_SIZE) 2. partition S into S1, S2 3. recursively apply *DCSort* to S1 and S2 to sort each of them in place 4. properly combine the S1 and S2, to reconstruct S completely sorted   else  nothing needs to be done since, if size <= 1, then it is already sorted   1. end |

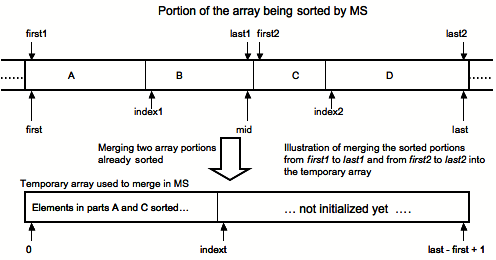
We will apply this technique to sort the elements of a List object in place: reorganize its content so that it becomes sorted in “increasing or decreasing order” based on a particular relation order for which a proper comparator is provided to be used when comparing elements in the list. At any moment, the list being considered is a contiguous portion of the original list, defined by the indexes of its first and its last position, which we refer to as *first* and *last*, respectively.

In the discussion that follows, we assume that the list to sort is given as an ArrayList[[2]](#footnote-1) object, representing a collection of objects of generic type E. We also assume that the order relation upon which the final order of elements in the list will be based 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 presented here may differ in details from those presented in lectures; however, they are based on the same ideas. We may also use the words “array” or “list” or “ArrayList” to refer to the list being sorted; but in the final implementation we use ArrayList.

**MergeSort**

If the portion of the list being considered consists of at least two elements (two consecutive positions in the ArrayList object), then the algorithm partitions it in two halves, or as close as possible. Sort each half in place, merge the two halves into a temporary array, and then copy back to the original portial of the ArrayList object. The partition process is simple, just determine the index corresponding to the middle of the list portion. The first part is then the list from the index value corresponding to the *first* position in the portion being under consideration up to the position in the *middle* (the one having index equal to the middle index value). The second list is the portion of the array having index values going from that *middle index value, plus 1*, up to the *last* position in the list portion being considered. Notice that the middle value is the value of the expression: *(first + last)/2*.

The next figure summarizes the idea followed when performing the *merging operation* of two contiguous portions of the list, which are assumed to be already sorted. A specific algorithm is shown in method **merge** that follows the figure. The figure shows the state of the arrays at some point during the merging process. That portion of the list (or array) consisting of the union of parts A and B in the figure is assumed to be sorted. The same is assumed for the section 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 they correspond to the portion of the list from position **first** to position **last**. The figure shows an *instance of the while-loop* in the merging process in which portions A and C have already been 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 first **indext** positions of the temporary array. Parts B and D are yet to be processed. The loop continues until one of the portions being merged is completely processed. At that moment, the while-loop ends and the algorithm just needs to verify which one is the sublist that 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 array being sorted. Notice that this implies that, after the method is executed, and assuming that the preconditions are met, that particular portion of **list** (the array being sorted) will be sorted.

Method **merge** works on the following two portions of the array:

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

It is assumed that both portions are already sorted.

|  |
| --- |
| **private void merge(int first, int mid, int last) {**  **E[] tempList = (E[]) new Object[last-first+1];**  **int index1 = first, index2 = mid+1;**  **int last1 = mid, last2 = last;**  **int indexTL = 0;**  **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 temp -- 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]);**    **}** |

The hard work in the merge sort strategy is done by method merge. The operation to partition the list is a simple task.

**QuickSort**

This algorithm puts all its effort in the partitioning operation. To partition a contiguous portion of the array (**list** in this case -- an ArrayList), from position **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 (**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 all *i* in the range from first to *p-1*, and *list.get(i) >= list.get(p)* for all *i* in the range from *p+1* up to **last**. Hence the partition separates the list into at most three lists portions: a first list (possibly empty) from positions **first** up to position *p-1*, another with just one element, the one at position *p*, and a third list (possibly empty) consisting of positions *p+1* up to position *last*.

The properties discussed above guarantee the following three important results:

1. Element finally left in position *p* of the list, does not need to be moved from there. It already is in the position it has to be once the list is sorted.
2. To complete the sorting process, elements in the first list (if not empty) do not need to be moved outside that portion of the list where they now are. The same will happen to elements in the third list (if not empty).

The following method implements one algorithm to determine the value of *p* that has been described.

|  |
| --- |
| **private int partition(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) { // [st1]**  **//swap list positions: left and right**  **swapListElements(left, right);**  **left++;**  **right--;**  **}**  **}**  **// swap list positions: left and last; the pivot value is finally**  **// placed at location left**  **swapListElements(left, last);**    **return left;**  **}** |

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 statement marked as [st1]) then the following is satisfied: *y >x* and *z < x*. The statement that follows switches these two values. The idea is to keep in the left part of the array portion (from position **first**) those values that are less than *x*, while keeping on the right part those that are greater or equal to *x*. As the outer loop iterates, the values of variables **left** (initialized to **first**) and **right** (initialized to **last-1**) move one toward the other, iterating until **right** *<* **left**. When this happens, then the process is completed; and, at that moment, the algorithm just needs to switch the values in positions **left** and **last** (which is the position of **pivot**). Finally, the value of *p* is determined as the value of **left**at that moment.



1. **HeapSort Algorithm**

As we studied in class, the HeapSort is another efficient algorithm to sort data. It is based on the *heap*. You should study what we saw in class about this algorithm, or what is presented in the textbook or in PowerPoint slides. But remember that the general idea of the algorithm to sort a list is as follows. Assume that the list is given as an *ArrayList* object representing a *complete binary* tree as was studied in the class, and as in the textbook.

|  |
| --- |
| Algorithm HeapSort:  INPUT: list - the list of elements to sort  OUTPUT: content of list sorted in decreasing order (remember, the order is relative to the comparator used)   1. Convert list into a heap. 2. Let n = list.size() 3. repeat the following for i=n-1, n-2, ..., 1    1. swap elements in position i and 0 in the list    2. apply downHeap algorithm from position 0 on the complete binary tree formed by the first i elements of the list (this is done by invoking downHeap(0, i) in class HeapSort) |

In the above algorithm, it is assumed that downHeap(r, n) does the downHeap operation in the subtree that is part of the complete binary tree formed by the first n positions in the list (or array) and whose root is the element of the list at position r; as discussed in lectures.

To convert the list into a heap, we use the *bottom-up strategy*: starting at level right before the last, and going backwards all the way to level 0 (the root of the tree), apply the downHeap operation at every internal node in that level. In particular, the strategy does the following:

|  |
| --- |
| 1. Let n be the size of the list (the size of the complete binary tree that it represents) 2. Let i be the index of the position in the list that represents the the last internal node in the complete binary tree represented by the list. 3. Repeat for r=i, i-1, ..., 0    1. apply downHeap algorithm from position r on the subtree rooted at r and which is part of the complete binary tree formed by the first n elements of the list (this is done by invoking downHeap(r, n) in class HeapSort) |

1. **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 classes, some of which are not really relevant for this lab activity. Inside package strategiesClasses you will find several classes implementing (or attempting to implement) different algorithms for solving two problems: the problem of counting frequency of objects in a dataset (these include the different strategies to count the number of repetitions (frequency) of the different elements in a dataset), and the problem of sorting elements in a dataset. *We are not interested on those strategies for counting frequencies here, they are 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 are implemented.* So, you are not expected to make changes on these classes. However, it is good that you study them, so that you can see how they are implemented; it is a good learning activity.

Also, inside the testerClasses package, you will see tester classes for each one of the strategies. You will only use some of them in the following exercises; in particular those dealing with the sorting problem.

In these exercises, we will practice with the following five algorithms to sort:

1. SelectionSort
2. BubbleSort
3. HeapSort
4. QuickSort
5. MergeSort

There are complete implementations for each of the first three. These are classes: SelectionSort, BubbleSort, and HeapSort. Also, we have included partial implementations of classes QuickSort and MergeSort. For each of those, there is also a tester class. That tester class generates random data and sorts it using the particular strategy. Study the corresponding testing classes inside the package named testerClasses.

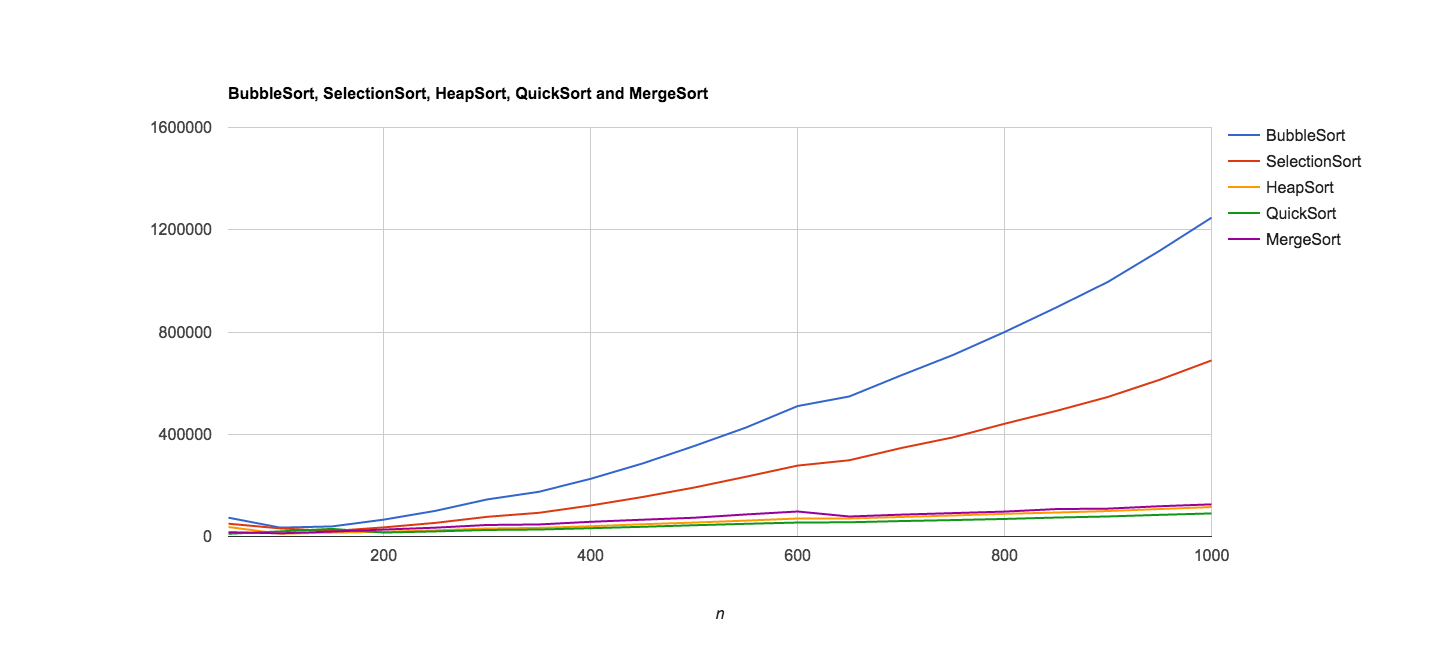
Notice that all these sorting strategies are subclasses of class AbstractSortingStrategy. The constructor or this type of object receives, as an explicit parameter, the comparator object to be used in the sorting process. Each such class implements its corresponding sorting algorithm inside the method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

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

* 1. Run the following tester classes: BubbleSortTester1, SelectionSortTester1, and HeapSortTester1. You should see as output a list of 500 randomly generated integer values as they were generated, and also as they are left after the corresponding sorting strategy is run.
  2. In the above cases, all the sorting objects are created using IntegerComparator1. Change it to IntegerComparator2 in those three cases and run again. Do your notice any difference in the final sorted output? Explain why.
  3. If you run the MergeSortTester1 class, you will notice that the data is left in the same order as it was generated; it is not sorted. That is because the strategy is not fully implemented. Look inside the code in class MergeSort and add the part that is missing. In particular, you need to complete method ms: determine how to correctly implement this method based on the previous discussion about the MergeSort algorithm and using the other methods that are already implemented in this class. Then, run the tester again. If correct, you should see that the data is sorted in increasing order.
  4. Repeat the above exercise, but this time for the QuickSort algorithm. Add the missing parts and test again. You need to complete method qs based on the previous discussion about QuickSort algorithm.
  5. Repeat exercise 2, this time for merge sort and for quick sort.
  6. Now run the experimentation code in class ExperimentalTrials as it is. It should produce average execution times for each of the five 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 parameter used when creating object ExperimentController, which is in charge of controlling the experimentation process. (See section at the end of this document.) If everything is correct, the execution will run experimental trials on all the five algorithms to sort. Results are placed inside subdirectory experimentalResults in separate files, each having, as part of its name the name of the particular strategy it corresponds to.
  7. Upload the experimental results for each sorting algorithm 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. Arrange in columns as suggested next, each containing the results that have been captured.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| n | BubbleSort | SelectionSort | HeapSort | QuickSort | MergeSort |
| 50 | 73581.586 | 50017.344 | 37101.105 | 10588.635 | 16852.46 |
| 100 | 34738.4 | 31673.46 | 10884.155 | 20051.79 | 12398.385 |
| ..... | ... | ... | ... | ... | ... |

Then, 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 (Consistent with theoretical results displayed at the beginning of this document.):



* 1. END

|  |
| --- |
| YOU DON’T NEED TO READ THIS SECTION TO WORK ON THE PREVIOUS LAB EXERCISES; IT IS HERE FOR THE PURPOSE OF EXPLAINING THE APPROACH FOLLOWED IN THE EXPERIMENTATION PART WITH THE GOAL TO MAXIMIZE CODE REUSABILITY. |

1. **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. Just to show its reusability, part of it includes solutions to another problem in addition to sorting; that is the problem of counting frequencies - number of occurrences of distinct elements in a dataset.



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)[[3]](#footnote-2) requires that any strategy to be tested is 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 dataset. 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. See the package strategiesClasses. See that these intermediate classes are used to normalize how the particular underlined strategy will be executed. Each must implement the abstract method (experimentallyExecuteStrategy) that is part of the abstract class at the top of the previous hierarchy (AbstractStrategyToTest).

For the case of the *frequency distribution problem*, in class AbstractFDStrategy, that normalization is achieved as follows:

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

Then, any subclass corresponding to a strategy to solve the frequency distribution problem needs to implement its algorithm as part of method named computeFDList.

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 needs to implement its algorithm as part of method named sortList.

**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 of implemented as subclasses of class AbstractStrategyToTest and embedded as part of an object of type StrategiesTimeCollection. The approach of the experimentation is the following (see run() in class ExperimentController)

|  |
| --- |
| 1. for each size n to be considered do    1. repeat the following m times (where m is the number of trials for each size or repetitionsPerSize)       1. generate a dataset of n random integers: ds       2. 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    2. for each strategy s compute the average execution time obtained for size n (the accumulated execution times divided by m) |

1. END

1. We say then that the elements of the partition are S1, S2,…, Sn. [↑](#footnote-ref-0)
2. This can easily converted to the case of an array or other similar structures. [↑](#footnote-ref-1)
3. I am not claiming that this is the best approach for this generic implementation framework, but it is good enough for the purpose here, and allows code reusability 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 on better alternatives to deal with this deficiency. One alternative would be that instead of inheritance, use composition.... But as said before, we will work with what we have. [↑](#footnote-ref-2)