**Sorter Lab**

In this lab you will modify several of the sorting routines that you created, allowing you to measure the performance of these sorting algorithms.

To begin, copy the folder, “Sorting starter code” from the shared drive to your working directory. Open the BlueJ project file. The project contains three classes. The first is an abstract class, Sorter. The Sorter class has a number of methods that will help you measure the performance of your sorting methods. These methods replace certain primitive operations that we want to measure. By calling the method instead of the primitive operation, your program will count the number of primitive operations performed during the sort operation.

*public boolean less(int leftInt, int rightInt)*

returns true if leftInt < rightInt; false otherwise. Increments the comparison counter.

*public boolean lessEqual(int leftInt, int rightInt)*

returns true if leftInt <= rightInt; false otherwise. Increments the comparison counter.

*public void swap(int[] a, int index1, int index2)*

Exchanges the elements in the array at the specified indices. Adds 3 to move counter.

*public void move(int[] a, int destIndex, int sourceIndex)*

Copies the element in the array at index, sourceIndex, to destIndex. Increments move counter.

*public void moveValue(int[] a, int destIndex, int value)*

Assigns value to the a[destIndex]. Increments move counter.

*public int[] allocateTempArray(int length)*

Returns a new integer array of size, length. Increments extra space counter.

*public int[] CopyArray(int[] a)*

Returns a new integer array that is a copy of a. Increments extra space counter.

*public abstract void sort(int[] a)*

Subclasses of Sorter must implement this method. The body contains a particular sort algorithm.

The second class in your project, QuickSorter, is a subclass of Sorter. It is included as an example of how to create a proper sub-class of Sorter. If you open it you will see that it implements the sort method.

The third class in your project, SortRunner, calls the sort method of QuickSorter several times with different input and prints out performance data on each call to sort().

**Activity 1 – Run Sorter application as delivered**

Compile your application and run it by calling the main method of SortRunner. You should see some performance statistics in the output window!

These statistics summarize the performance of the QuickSort algorithm. We discussed this algorithm briefly in class but never implemented it. Nevertheless, by looking at the performance output, make an educated guess about the performance classes for the QuickSort algorithm in the best, average and worst cases. Record your answers in the worksheet provided.

**Activity 2 – Implement the selection sort algorithm in a subclass of Sorter**

1. Create a new class, SelectionSorter.
2. Modify the class declaration so that SelectionSorter extends Sorter
3. Implement the sort method of SelectionSorter.

You already created one implementation of the selection sort algorithm in a previous lab. Simply copy to body of that method into the sort method. Modify the copied body so that you call the less method instead of any uses of the “<” operator. Do the same thing for uses of “<=”, swaps, moves and allocations of temporary space.

1. In the main method of SortRunner you will find 2 lines of code that are commented out:

// SelectionSorter selectionSorter = new SelectionSorter();

// selectionSorterArray[lengthIndex] = runStandardSorts(selectionSorter, length, "Selection sort");

Uncomment these two lines so that your SelectionSorter class is called.

Compile your application and run it by calling the main method of SortRunner. You should more performance statistics in the output window! Make an educated guess about the performance classes for the selection sort algorithm in the best, average and worst cases. Record your answers in the worksheet provided.

**Bonus Activities**

Repeat activity 2 for insertion sort and merge sort.

**Appendix A – Performance worksheet**

Name: Emma Chiu

Quick Sort Algorithm

|  |  |  |  |
| --- | --- | --- | --- |
| *Given Formula* | *Comparisons* | *Moves* | *Extra space* |
| Best case | O(length\*log(length)) | O(length)? | O(1) |
| Average case | O(length\*log(length)) | O(length\*log(length)) | O(1) |
| Worst case | O(length\*log(length)) | O(length)? | O(1) |
| *My Data* | *Comparisons* | *Moves* | *Extra space* |
| Best case | O(length\*5 + length\*0.5 – 1) | O(0) | O(0) |
| Average case | No formula; appears exponential | No formula; appears exponential | O(0) |
| Worst case | O(length\*5 – 1) | O(length\*1.5) | O(0) |

Selection Sort Algorithm

|  |  |  |  |
| --- | --- | --- | --- |
| *Given Formula* | *Comparisons* | *Moves* | *Extra space* |
| Best case | O(length2) | O(length) | O(1) |
| Average case | O(length2) | O(length) | O(1) |
| Worst case | O(length2) | O(length) | O(1) |
| *My Data* | *Comparisons* | *Moves* | *Extra space* |
| Best case | O(length^2 + length - 1) | O(3\*length – 3) | O(0) |
| Average case | O(length^2 + length - 1) | No formula; appears exponential | O(0) |
| Worst case | O(length^2 + length - 1) | No formula; appears exponential | O(0) |

Insertion Sort Algorithm

|  |  |  |  |
| --- | --- | --- | --- |
| *Given Formula* | *Comparisons* | *Moves* | *Extra space* |
| Best case | O(length) | O(1) | O(1) |
| Average case | O(length2) | O(length2) | O(1) |
| Worst case | O(length2) | O(length2) | O(1) |
| *My Data* | *Comparisons* | *Moves* | *Extra space* |
| Best case | O(length^2 + length - 1) | O(0) | O(0) |
| Average case | O(length^2 + length - 1) | No formula; appears exponential | O(0) |
| Worst case | O(length^2 + length - 1) | No formula; appears exponential | O(0) |

Merge Sort

|  |  |  |  |
| --- | --- | --- | --- |
| *My Data* | *Comparisons* | *Moves* | *Extra space* |
| Best case |  |  |  |
| Average case |  |  |  |
| Worst case |  |  |  |

Heap Sort

|  |  |  |  |
| --- | --- | --- | --- |
| *My Data* | *Comparisons* | *Moves* | *Extra space* |
| Best case |  |  |  |
| Average case |  |  |  |
| Worst case |  |  |  |