Jeffrey Allen

Michael Beaver

CS 355

November 20, 2012

Assignment 11, Group 3

Description of System Design and Implementation

**Structure of Sorting Algorithms**

This system employs a class hierarchy of different sorting algorithms. The base class is an abstract class named AbstractSort, which declares various public virtual methods that derived classes inherit and define as necessary. The public methods can be anything, but a few methods are required: sort, merge, swap, and print. Of course, other methods, such as statistical methods or accessor methods may be included. Because AbstractSort is abstract and cannot be instantiated, it has no real need for private or protected data members. However, any virtual methods that return a value should be defined to return a default value in the case that a derived class does not define one of the virtual methods.

Other sorting algorithms are grouped by their runtimes. For example, Bubble Sort, Selection Sort, and Insertion Sort all have runtimes of O(N2). Hence, they are all grouped under the umbrella class NSquaredSorts. In turn, NSquaredSorts inherits from AbstractSort. Because the classes BubbleSort, SelectionSort, and InsertionSort inherit from NSquaredSorts, all of the methods declared in AbstractSort are inherited. All that remains is to define and implement the virtual methods and other non-virtual methods. Likewise, Quick Sort and Merge Sort have runtimes of O(NlogN). Hence, the classes QuickSort and MergeSort are grouped under and inherit from the umbrella class NLogNSorts, which inherits from AbstractSort. The only remaining sort is Radix Sort, which has a simplified runtime of O(N). Hence, the class RadixSort is grouped under and inherits from the class LinearSorts, which inherits from AbstractSort. For a graphical representation, see the attached class hierarchy illustration.

**System Design Features**

This system design allows a user to implement or use implemented sorting algorithm classes as instantiated objects, which collect statistical data about their respective sorting routines. The collected statistical data may be parsed and analyzed by the user. The system is structured as a class hierarchy. The hierarchy structure allows the user to create their own sorting algorithm classes, import other classes, or remove classes. Essentially, the system allows for “plug-n-play” of sorting algorithms. The included default sorting algorithms include: BubbleSort, SelectionSort, InsertionSort, MergeSort, QuickSort, and RadixSort.

The driver program allows the user to input data via file input or manual input. During manual input, the user may save their original input to a file, which may be read in to the program at a later time. Also, the driver allows the user the option to save sorted data and algorithm statistical data to a file for each sorting algorithm. The statistical data in the saved files is organized so the user may easily peruse the data or use another program to parse the statistical data. Also, the driver allows the user to output statistical data for each sorting algorithm in the form of tables. Also, the user may modify a constant integer value that controls which data sets are printed to the screen. If a data set has size less than the constant’s value, the unsorted data is automatically printed as well as the sorted data from each sorting object.

**Description of the Driver Program**

The driver program prompts the user with a menu of available options: Read data in from a file, input data manually, print tables of collected statistical data, show the main menu, or quit the program. Each menu selection behaves differently. Some menu options have more than one exit path while others have only one path. The menu is displayed after the execution of every menu operation. The only way to quit the program, without manually closing the console, is to select the “quit” menu option. For a graphical representation, see the attached flowchart.

When loading data in from a file, prompts ask the user for information relating to the data file. The user is prompted for the filename, the data set size, the ordering of the data (in order, random, or reverse order), and whether or not to save the sorted data and statistical information to a file. If the user inputs bad information for any prompt, the prompt will reappear until the user inputs valid information. Once a filename is given and the file is determined to exist, the file contents are read into a temporary dynamic array. There are assumptions: (a) the input file has each integer value on an individual line (the program could be modified to allow for custom delimiters); and (b) the data values begin at the beginning of the file. To accommodate the implementation of RadixSort, values should all be the same length (i.e., have the same number of digits). Once the data is read into the temporary array, the data is processed by six different sorting objects, which correspond to the six sorting algorithm classes. If the data set size is less than the constant value MAX\_VALUES, which the user may alter, the unsorted and sorted data for each sorting object are printed to the screen. If the data set size exceeds MAX\_VALUES, neither the unsorted data nor the sorted data are printed to the screen. However, if the user chooses to save the sorted data and statistical information to a file, then the sorted data is indeed saved to a file for easy perusal. After processing the data, the program returns to the main menu.

Loading data directly from user input behaves similarly to loading data from a file. However, no input file is used; the user inputs the data set size, the data ordering, and the actual data. If the user wishes to save their original data, they may choose to do so; a prompt will ask for a filename after all data is entered. Like loading from a file, the user’s input data is processed and statistical information is collected. If the user wants to save the sorted data to a file, that option is also available. After fully processing the data, the program returns to the main menu.

The user may choose to view the collected statistical data for each sorting algorithm in the form of tables. The table values default to zero, but they are updated as the user’s data is processed. Statistical data is stored in matrices, which are then parsed and used to generate tables. The screen is cleared each time the sorting algorithms’ tables are generated. The sorting algorithms follow the order: BubbleSort, SelectionSort, InsertionSort, MergeSort, QuickSort, and RadixSort. Each sorting algorithm has its own series of tables, including: Inner Loop counts, Outer Loop counts, Swap counts, Recursion counts, Divisions counts, and Merges counts. Each table’s data is also divided by ordering (Random, In Order, and Reverse Order) and by the data set sizes (10 items, 100 items, 1000 items, and 10000 items). After RadixSort’s tables are displayed, the program returns to the main menu.

The “help” option merely displays the main menu whenever the user presses the corresponding key. It performs no other tasks—the user’s menu choice is read by the main function. The “quit” option ends the main function and exits the program altogether.

**Routines, Runtimes, and Storage Constraints**

The different components of the system are outlined below. Relevant routines, runtimes, and storage constraints are given under each heading.

*AbstractSort*

**sort():** No runtime because the method is purely virtual.

**sort(int \* a, int arrSize):** No runtime because the method is not defined.

**swap(int val1, int val2):** No runtime because the method is not defined.

**merge(int \* arr1, int \* arr2):** Constant runtime because the method only returns the value 0.

**print():** No runtime because the method is not defined.

**getArr():** Constant runtime because the method only returns 0.

**getNumLoops():** Constant runtime because the method only returns 0.

**getNumInnerLoops():** Constant runtime because the method only returns 0.  
 **getNumOuterLoops():** Constant runtime because the method only returns 0.

**getNumSwaps():** Constant runtime because the method only returns 0.

**getNumMerges():** Constant runtime because the method only returns 0.

**getNumRecursions():** Constant runtime because the method only returns 0.

**getNumDivisions():** Constant runtime because the method only returns 0.

**getSortType():** Constant runtime because the method only returns the CLASS\_NAME data member.

*NSquaredSorts*

**getNumLoops():** No runtime because the method is purely virtual.

**getNumOuterLoops():** Constant runtime because the method only returns the value 0.

**getNumInnerLoops():** Constant runtime because the method only returns the value 0.

**getNumSwaps():** Constant runtime because the method only returns the value 0.

*BubbleSort*

**sort():** Runtime O(N2) if the data is not sorted; otherwise, the runtime is O(N).

**sort(int \* a, int arrSize):** This method calls sort, so it has the same runtime(s).

**print():** This method prints each value, so it has runtime O(N).

**getNumLoops():** Constant runtime because it simply returns a data member value.

**getNumOuterLoops():** Constant runtime because it simply returns a data member value.

**getNumInnerLoops():** Constant runtime because it simply returns a data member value.

**getNumSwaps():** Constant runtime because it simply returns a data member value.

**getArr():** This method returns a pointer to a temporary copy of the data member array. Hence, it has to hit each value in the data set, so the method has runtime O(N).

**swap(int val1, int val2):** Relative to the data set size, this method runs in constant time because it only swaps two values in the data array.

**init(int \* a, int arrSize):** This method has runtime O(N) because it copies the values from the array “a” to the data member array.

The only practical constraint is memory space. The data is stored in a dynamic array; hence, it is possible that a set of data could run out of room during allocation of memory for the data array.

*SelectionSort*

**sort():** Runtime is O(N2) because the algorithm loops through the data array roughly N2times (after eliminating constants).

**sort(int \* a, int arrSize):** Calls sort, so it has the same runtime.

**print():** See BubbleSort::print()

**getNumLoops():** See BubbleSort::getNumLoops()

**getNumOuterLoops():** See BubbleSort::getNumOuterLoops()

**getNumInnerLoops():** See BubbleSort::getNumInnerLoops()

**getNumSwaps():** See BubbleSort::getNumSwaps()

**getArr():** See BubbleSort::getArr()

**swap(int val1, int val2):** See BubbleSort::swap(x, y)

**init(int \* a, int arrSize):** See BubbleSort::init(a, b)

Like BubbleSort, the only practical storage constraint is the amount of memory available. If the data set size exceeds the amount of memory available, then there will obviously be a memory management issue. Possibly the array could be reduced to smaller arrays that are easier to handle and less memory intensive.

*InsertionSort*

**sort():** Runtime of O(N2) because each item in the array is processed possibly more than once. This algorithm is similar to SelectionSort and BubbleSort.

**sort(int \* a, int arrSize):** Calls sort, so it has the same runtime.

**print():** See BubbleSort::print()

**getNumLoops():** See BubbleSort::getNumLoops()

**getNumOuterLoops():** See BubbleSort::getNumOuterLoops()

**getNumInnerLoops():** See BubbleSort::getNumInnerLoops()

**getNumSwaps():** See BubbleSort::getNumSwaps()

**getArr():** See BubbleSort::getArr()

**swap(int val1, int val2):** See BubbleSort::swap(x, y)

**init(int \* a, int arrSize):** See BubbleSort::init(a, b)

This algorithm could be nasty if it inserts values into an array rather than swapping values. In this system’s implementation, values are swapped. Large arrays could prove unwieldy.

*NLogNSorts*

**getNumLoops():** No runtime because the method is purely virtual.

**getNumInnerLoops():** Constant runtime because the method only returns the value 0.

**getNumOuterLoops():**Constant runtime because the method only returns the value 0.

**getNumMerges():** Constant runtime because the method only returns the value 0.

**getNumDivisions():** Constant runtime because the method only returns the value 0.

**getNumRecursions():** Constant runtime because the method only returns the value 0.

*MergeSort*

**sort():** Calls sortHelper(a, t, l, r), which has combined runtime O(NlogN). Hence, the sort method’s runtime is O(NlogN).

**sort(int \* a, int arrSize):** Calls sort, so it has the same runtime.

**getNumLoops():** See BubbleSort::getNumLoops()

**getNumOuterLoops():** See BubbleSort::getNumOuterLoops()

**getNumInnerLoops():** See BubbleSort::getNumInnerLoops()

**getNumSwaps():** See BubbleSort::getNumSwaps()

**getNumMerges():** Constant runtime because the method only returns a data member value.

**getNumDivisions():** Constant runtime because the method only returns a data member value.

**getNumRecursions():** Constant runtime because the method only returns a data member value.

**print():** See BubbleSort::print()

**getArr():** See BubbleSort::getArr()

**merge(int \* a, int \* temp, int left, int middle, int right):** The merge routine merges sub-arrays to make larger arrays. As the sub-arrays grow in size, eventually N items are touched. Hence, the method’s runtime is O(N).

**init(int \* a, int arrSize):** See BubbleSort::init(a, b)

**sortHelper(int \* a, int \* temp, int left, int right):** This method recursively divides arrays into sub-arrays, thereby halving the problem. Hence its initial runtime is O(logN). However, because it calls the merge method, it has a combined runtime of O(NlogN).

Large arrays could pose a problem for MergeSort. Indeed, many recursions could lead to stack overflows. Thus, having a large array means having many sub-arrays, which means many recursions. With more recursions, there is a higher risk for stack overflow.

*QuickSort*

**sort():** Because this method calls sortHelper, which has combined runtime of O(NlogN), this method also has runtime O(NlogN).

**sort(int \* a, int arrSize):** Calls sort, so it has the same runtime.

**getNumLoops():** See BubbleSort::getNumLoops()

**getNumOuterLoops():** See BubbleSort::getNumOuterLoops()

**getNumRecursions():** See MergeSort::getNumRecursions()

**getNumDivisions():** See MergeSort::getNumDivisions()

**getNumSwaps():** See BubbleSort::getNumSwaps()

**swap(int \* a, int val1, int val2):**  Like the other versions of swap(x, y), this version swaps two values in an array, namely “a.” Other versions of swap directly alter the data member array. Because the sortHelper method uses this data member, it is unwise to alter it with swap. Indeed, “a” is actually a sub-array that has been divided from the data member array.

**print():** See BubbleSort::print()

**getArr():** See BubbleSort::getArr()

**sortHelper(int \* a, int left, int right):** Divides the original array into sub- arrays, whose values are then sorted. This implementation uses stacks and is therefore non-recursive.

**init(int \* a, int arrSize):** See BubbleSort::init(a, b)

This implementation of QuickSort is stack-based because recursive implementations experienced stack overflows with large data sets. To counter the problem of stack overflow, an individual stack is used, which does not create new program stack frames but operates independently. Using stacks allows for more “stable” sorting of larger data sets, insofar as the program is less likely to crash.

*LinearSorts*

**getNumLoops():** No runtime because the method is purely virtual.

**getNumInnerLoops():**Constant runtime because the method only returns the value 0.

**getNumOuterLoops():** Constant runtime because the method only returns the value 0.

**getNumSwaps():** Constant runtime because the method only returns the value 0.

*RadixSort*

**sort():** Sort loops D times, where D = the number of digits, and each item is sorted once per round. Hence the runtime is O(D \* N), which simplifies to O(N) because D is constant.

**sort(int \* a, int arrSize):** Calls sort, so this method has the same runtime.

**print():** See BubbleSort::print()

**getNumLoops():** See BubbleSort::getNumLoops()

**getNumInnerLoops():** See BubbleSort::getNumInnerLoops()

**getNumOuterLoops():** See BubbleSort::getNumOuterLoops()

**getNumSwaps():** See BubbleSort::getNumSwaps()

**getArr():** See BubbleSort::getArr()

**init(int \* a, int arrSize):** See BubbleSort::init(a, b)

This implementation uses an array of LinkedLists to simulate BucketSort. This could be memory intensive if multiple values, say 9000 values of 10000 items, map to the same bucket. This could also be costly if all the data values map to the same “bucket.” One convenience is that LinkedLists have an ordering property, so the data is automatically ordered when it is mapped to its “bucket.” To save on memory, the “buckets” are cleared after each outer loop. This means that every node in every LinkedList is returned to memory.

*Driver Program*

**main():** Loops until the user chooses to quit. Runtime varies.

**loadDataFromFile(…):** This function calls readFile(. . .) and has its runtime. This function also calls either processData(int \* data, . . .) or processData(string filename, . . .), so it could potentially have either function’s runtime.

**loadDataFromUser(…):** This function has to read in N items from the user, but it also behaves like loadDataFromFile(. . . ). Hence, this function could have one of two runtimes.

**readFile(…):** This function potentially has to read in every line of a file. In the best case (the file is bad or does not exist), it does not have to do anything. Hence, the runtime is usually O(N), whereas the best case runtime is O(1).

**saveFile(…):** This function has to write N items to a file, in addition to a constant amount of statistical data. Hence, the runtime of this function is O(N).

**processData(int \* data, …):** This function loops through six sorting algorithm objects. It also has the option to print out an unsorted array and sorted arrays. Hence, it has potential runtime O(N). Of course, this function could also have the runtime of each sorting algorithm object’s sort routine.

**processData(string filename, . . .):** See processData(int \* data, . . .)

**makeTables(…):** This function has a terrible runtime because it has to process four 3-D arrays. It has to loop for every sorting algorithm, every statistical data category, every data size category, and every data ordering. Hence this function has a runtime of O(N4).

**printArr(int \* arr, int size):** Prints all items in an array, so this method has runtime O(N).

**PrintMenu(…):** Constant runtime because the method only prints menu items to the screen.