Sorting Algorithms – Organizing Data

SUMMARY:

During the start of my senior year in Programming and Web Development, I implemented multiple sorting algorithms to help organize data. First, I gathered information on nine different types of sorting algorithms in Java online. Next, I placed each example on one Java file while ensuring there were no errors. Afterwards, I created a Launcher Java file that would be able to run the file with the algorithms separately. Lastly, I altered my code to include each code's execution time for multiple array sizes. With the knowledge of these sorts, I can efficiently sort any amount of information I may possess.

BACKGROUND:

In order to successfully fulfill this project, I needed to recollect the coding skills that I learned through my sophomore and junior year. I also needed to be familiar with the Java IDE that I used for the project. The programming topics associated with the project are the use of multiple classes and working with methods. It is assumed in this project that the reader has some knowledge of these topics.

SOFTWARE:

- Java IDE (Eclipse Neon is used in this example)
- Any browser with access to the internet

PROCEDURE:

To begin the project, you need to research the different types of sorting algorithms. You will only be working with nine sorts for this, although there are much more to be found on the internet. To begin the research, open your preferred internet browser. Then, use various search engines to find information about nine sorting algorithms in Java. In this case, we researched

bubble, bucket, insertion, merge, heap, quick, selection, counting, and shell sorts. Once you get a general idea about what each sort is, take notes on the space complexity and big O notation for them. This will not be directly used within the project, but it will be helpful to know for the conclusion and objective of the project. Afterwards, find an example of source code for each type of sort. I copied them into notepad and saved them as text files just to have them ready for the next section of the task.

For the next segment of this assignment, the sorting algorithms are to be put into one Java file. To complete this project, you first need to open up your Java IDE and create a Class file either within an existing Java project or in a new one. Make sure this class file has no main method. Once the file is created, copy and paste in all of the Java methods for each sorting algorithm. If you copied your snippets of code correctly, you should not have any errors within your code. But in most cases, there will be variable or method names that conflict with each other. Parse through the code that you manufactured and search for any errors in your code. After this, look through the methods that delete / change any methods that you do not need to implement (aka duplicate methods or methods that serve no purpose). Lastly, make sure you keep note of which methods start each sort and that a method to display the sorts are present. You will need them for the next section.

By this point, you may realize that there is no way to actually test the functionality of this code, since there is no main method to run the program. We will be creating a second class in order to run the logic presented in the class filled with the nine sorting algorithms. Create a second class within the same package as the first one you forged. This one will have a main method in it. In the main class, create an array with random numbers inside it (it does not matter if the values are inputted manually or randomly). In the main method of this class, create a new array object that will hold the sorted values of the original array. Then, call the methods from the other class that initiates each sort. After initiating each sort, display the array to show that each sort works. Once you know each sort is working properly, this part of the section is complete.

With the sorts working to their potential, you can now alter this code to see which sort is the most efficient. To find out the times, you can implement System.nanoTime to record the time it takes for each array sort. First off, create a new array (I called it “sizes”) filled with different array sizes. I used 10; 100; 1,000; 10,000; and 100,000. Next, create a method in the SortingAlgorithm class, create a method to randomly populate the array with values based on the

size. Next, in the main class, delete all displayArray instances. If you use large arrays such as 100,000 displaying all of the values could potentially crash the console. Then, put all of the sorting algorithms into a loop. Each iteration of the loop should populate and sort an array with one of the sizes specified in the size array you created. To measure the time, make a long variable on the line before each sort called startTime. Make it equal to System.nanoTime(). Then, create another long variable called endTime equal to System.nanoTime as well. Afterwards, print the difference between the endTime and startTime to show the execution time for said sorting algorithm. Repeat this for each sorting algorithm and the project should be complete.

RESULTS:

Once the final version of this project is run, the execution times for each sort should be displayed within the console of the IDE. The sort with the most efficient execution time for most cases happened to be the shell sort.

```
//////////Shell Sort//////////
        /**
         * This method sorts the data in ascending order via shell sorting
         *
         * @param  sequence   the unsorted array
         *
         * @return int []         the array sorted properly
         */
    public static int [] shellSort(int sequence[])
    {
        int size = sequence.length;

        // Start with big gap, then reduce the gap
        for (int gap = size/2; gap > 0; gap /= 2)
        {
            // Do a gapped insertion sort for this gap size.
                sequence = insertionSort(sequence);
        }
        return sequence;
    }//end of shellSort
```

1: An example of a sort used in the project.

CONCLUSION:

By the end of the project, you should have a few sorting algorithms that show their execution times in the console. This activity was meant to teach how to implement methods from other classes, how to use certain sorting methods, and how to find out which sorting algorithm to use for each size. With this knowledge, sorting through small and large amounts of data will be much easier.

```
int [] sizes = {10, 100, 1000, 10000, 100000};

long [] times = new long[9];

for(int i = 0; i < sizes.length; i++){

int [] unsortedarray = ArrayStuff.populateArray(sizes[i]);//Original array

int [] sortedarray;//Sorted array

int maxVal = SortingAlgorithm.maxValue(unsortedarray);//For bucket sort only

//Bucket Sort
System.out.println("Bucket Sort at size "+sizes[i]);
long startTime = System.nanoTime();
sortedarray = SortingAlgorithm.bucketSort(unsortedarray, maxVal);
long endTime = System.nanoTime();
times[0] = endTime - startTime;
System.out.println("Total execution time: " + (endTime - startTime)+ " nanoseconds" );
System.out.println("");

//Bubble Sort
System.out.println("Bubble Sort at size "+sizes[i]);
startTime = System.nanoTime();
sortedarray = SortingAlgorithm.bubbleSort(unsortedarray);
endTime = System.nanoTime();
times[1] = endTime - startTime;
System.out.println("Total execution time: " + (endTime - startTime)+ " nanoseconds" );
System.out.println("");

//Insertion Sort
System.out.println("Insertion Sort at size "+sizes[i]);
startTime = System.nanoTime();
sortedarray = SortingAlgorithm.insertionSort(unsortedarray);
endTime = System.nanoTime();
times[2] = endTime - startTime;
System.out.println("Total execution time: " + (endTime - startTime)+ " nanoseconds" );
System.out.println("");

//Selection Sort
System.out.println("Selection Sort at size "+sizes[i]);
startTime = System.nanoTime();
sortedarray = SortingAlgorithm.selectionSort(unsortedarray);
endTime = System.nanoTime();
times[3] = endTime - startTime;
System.out.println("Total execution time: " + (endTime - startTime)+ " nanoseconds" );
System.out.println("");

//Heap Sort
System.out.println("Heap Sort at size "+sizes[i]);
startTime = System.nanoTime();
sortedarray = SortingAlgorithm.heapSort(unsortedarray);
endTime = System.nanoTime();
times[4] = endTime - startTime;
System.out.println("Total execution time: " + (endTime - startTime)+ " nanoseconds" );
System.out.println("");

//Merge Sort
System.out.println("Merge Sort at size "+sizes[i]);
startTime = System.nanoTime();
sortedarray = SortingAlgorithm.mergeSort(unsortedarray);
endTime = System.nanoTime();
times[5] = endTime - startTime;
System.out.println("Total execution time: " + (endTime - startTime)+ " nanoseconds" );
System.out.println("");

//Quick Sort
System.out.println("Quick Sort at size "+sizes[i]);
startTime = System.nanoTime();
sortedarray = SortingAlgorithm.quickSort(unsortedarray);
endTime = System.nanoTime();
times[6] = endTime - startTime;
System.out.println("Total execution time: " + (endTime - startTime)+ " nanoseconds" );
System.out.println("");

//Counting Sort
System.out.println("Counting Sort at size "+sizes[i]);
startTime = System.nanoTime();
sortedarray = SortingAlgorithm.countingSort(unsortedarray);
endTime = System.nanoTime();
times[7] = endTime - startTime;
System.out.println("Total execution time: " + (endTime - startTime)+ " nanoseconds" );
System.out.println("");

//Shell Sort
System.out.println("Shell Sort at size "+sizes[i]);
startTime = System.nanoTime();
```

2: A snippet of code in the MainClass