Assignment Two – Magic Items Sorter

C. Marcus DiMarco
C.DiMarco1@Marist.edu

October 8, 2021

1 Objectives

- Create a program that sorts all Strings within a given text file using selection sort, insertion sort, merge sort and Quicksort.
- Print the number of comparisons for each sort.

2 Conditions

- The relation on the set of Strings must be 'less than', such that for any element a which appears before any other element b, a is less than b. For Strings, 'less than' is defined as preceding alphabetically.
- The array must be shuffled before running the sorts, and the sorts must act on the same shuffle of the array.
- The program must only use custom algorithms for the sorts. Any of these sorts which already exist in the language may not be used.

3 Overview

In order to enhance readability, each sorting algorithm is a method contained in the class MarcusSort.

3.1 selectionSort()

The most straightforward sorting algorithm in this program is selection sort. Selection sort starts at the first index, finds the minimum value in the array and places it at the current index, repeating until the array is sorted.

```java
// Selection sort will search for minimum unsorted value and
// place it at first unsorted index.
public void selectionSort(String[] strings) {

  // Start with counter at 0
  this.resetCounter();

  // First unsorted index through each pass is i
  for (int i = 0; i < strings.length; i++) {
    counter++;     // Comparison in for loop
    int minIndex = i;
    // Find index of smallest value in remainder of array
    for (int j = i + 1; j < strings.length; j++) {
      counter += 2;  // Comparisons in for loop and if statement
      if (strings[minIndex].compareToIgnoreCase(strings[j]) > 0) {
        minIndex = j;
      }
    }
    counter++;     // Comparison for inner loop termination

    // Swap indices minIndex and i
    String temp = strings[i];
    strings[i] = strings[minIndex];
    strings[minIndex] = temp;
  }
  counter++;       // Comparison for outer loop termination

  // Print completion message with number of comparisons
  this.printCompletionMessage();
}
```

3.2 insertionSort()

Improving on selection sort is insertion sort. Insertion sort relies on taking a key value and placing it in sorted order among all of the previous elements. We start at index 1 and compare it to the value at index 0, sorting those two as needed. Note that after each iteration, indices 0-i are sorted.

```java
  // Insertion sort will sort the array from index 0 through i,
  // 'inserting' each value into its correct position.
  public void insertionSort(String[] strings) {

    // Start with counter at 0
    this.resetCounter();

    // Start comparisons at index 1
    for (int i = 1; i < strings.length; i++) {
      counter++;   // Comparison in outer loop
      String keyString = strings[i];
      for (int j = i - 1; j >= 0; j--) {
        counter += 2;  // Inner loop and if statement
        // Move all items larger than key forward 1 index
        if (keyString.compareToIgnoreCase(strings[j]) < 0) {
          strings[j + 1] = strings[j];
          counter++; // Second comparison in if statement
          // If index 0 reached, assign it currentString
          if (j == 0) {
            strings[j] = keyString;
          }
        } else {
          // Insert key at index ahead of first smaller element
          strings[j + 1] = keyString;
          break;
        }
      }
      counter++;   // Comparison for inner loop termination
    }
    counter++;     // Comparison for outer loop termination

    // Print completion message with number of comparisons
    this.printCompletionMessage();
  }
```

3.3 mergeSort()

In stark contrast to the other sorting algorithms seen so far, merge sort drastically reduces complexity by implementing a "divide and conquer" tactic, which relies on a simple principle: arrays of size 1 are sorted. Therefore, if we divide the target array into two subarrays, and continue to divide those until we reach arrays of size 1, we can recombine the subarrays in sorted order and greatly reduce the amount of work needed. It uses two operations, sortThenMerge() and merge().

```java
  /**
   * mergeSort() is an abstraction for cleaner user interaction. It will reset
   * the counter, call the recursive merge function, and then print the
   * completion message with the number of comparisons.
   * @param strings
   */
  public void mergeSort(String[] strings) {

    // Start with counter at 0
    this.resetCounter();

    // Nest recursive function inside for readability and proper
    // counter/print calls
    this.sortThenMerge(strings, 0, strings.length - 1);

    // Print completion message with number of comparisons
    this.printCompletionMessage();
  }

  // Merge sort will divide the array recursively until subarrays are
  // of size 1, then merge the subarrays together in sorted order.
  private void sortThenMerge(String[] strings, int leftIndex, int rightIndex) {

    counter++;   // Increment the if comparison
    if (leftIndex < rightIndex) {
      int midpoint = (leftIndex + rightIndex) / 2;
      sortThenMerge(strings, leftIndex, midpoint);
      sortThenMerge(strings, midpoint + 1, rightIndex);
      merge(strings, leftIndex, midpoint, rightIndex);
    }
  }

  // Merge will take an array, create two sorted subarrays, and
  // sort the elements in place.
  private void merge(String[] strings, int leftIndex, int midpoint,
          int rightIndex) {

    // Create two (sorted, due to recursion) subarrays
    String[] stringsL = new String[midpoint - leftIndex + 1];
    String[] stringsR = new String[rightIndex - midpoint];

    // Populate the subarrays, incrementing for each comparison
    for (int i = 0; i < stringsL.length; i++) {
      counter++;
      stringsL[i] = strings[leftIndex + i];
    }
```

```
    counter++;   // Increment the loop exit comparison

    for (int j = 0; j < stringsR.length; j++) {
      counter++;
      stringsR[j] = strings[midpoint + j + 1];
    }
    counter++;   // Increment the loop exit comparison

    int i = 0;
    int j = 0;
    int k = leftIndex;
    // For the selected range, sort until one subarray is exhausted
    while (i < stringsL.length && j < stringsR.length) {
      counter += 2;  // Increment for loop and if statements
      if (stringsL[i].compareToIgnoreCase(stringsR[j]) < 0) {
        strings[k++] = stringsL[i++];
      } else {
        strings[k++] = stringsR[j++];
      }
    }
    // Append the remaining (sorted) subarray
    if (i == stringsL.length) {
      for ( ; k <= rightIndex; k++) {
        counter++;
        strings[k] = stringsR[j++];
      }
      counter += 2;  // Increment for loop exit and if
    } else {
      for ( ; k <= rightIndex; k++) {
        counter++;
        strings[k] = stringsL[i++];
      }
      counter += 2;  // Increment for loop exit and if
    }
  }
```

3.4 quickSort()

Building off of the "divide and conquer" principle in merge sort, Quicksort looks to make things even more efficient by beginning the conquering during the dividing instead of waiting until after. Quicksort selects a pivot value and divides the elements in the array around it, with all smaller elements to its left and all larger elements to its right. This means that after each call, any pivot value is in its correct index.

```java
  /**
   * quickSort() is an abstraction for cleaner user interaction. It will reset
   * the counter, call the recursive quicksort function, and print the
   * completion message with the number of comparisons.
   * @param strings
   */
  public void quickSort(String[] strings) {

    // Set counter to 0
    this.resetCounter();

    // Nest recursive function inside for readability and proper
    // counter/print calls
    this.quickSort(strings, 0, strings.length - 1);

    // Print completion message with number of comparisons
    this.printCompletionMessage();
  }

  // Quicksort will take an array and recursively divide it around
  // a pivot value, sorting the elements around the pivots
  private void quickSort(String[] strings, int leftIndex, int rightIndex) {

    counter++;   // Increment for if comparison
    if (leftIndex < rightIndex) {
      // Quicksort the elements to either side of the partition
      int partition = partition(strings, leftIndex, rightIndex);
      quickSort(strings, leftIndex, partition - 1);
      quickSort(strings, partition + 1, rightIndex);
    }
  }

  // Partitions an array around a pivot value and places all values
  // smaller than the pivot to its left, then places pivot at its sorted index
  private int partition(String[] strings, int leftIndex, int rightIndex) {

    String pivotString = strings[rightIndex];

    int sortedIndex = leftIndex;

    for (int i = leftIndex; i <= rightIndex; i++) {
      counter += 2;    // Increment for loop and if statement
      if (strings[i].compareToIgnoreCase(pivotString) < 0) {
        // If element at i is smaller than pivot, place in the
        // left half of the array
        String tempString = strings[sortedIndex];
```

```java
        strings[sortedIndex++] = strings[i];
        strings[i] = tempString;
      }
    }
    counter++;   // Increment for loop exit comparison

    // Swap pivot string and value at the pivot's sorted index
    strings[rightIndex] = strings[sortedIndex];
    strings[sortedIndex] = pivotString;
    return sortedIndex;
  }
```

3.5 Other helper functions

Below are the remaining functions present in MarcusSort().

```java
  // Shuffle routine based on the Knuth or Fisher-Yates, but not
  // Rosanna, shuffle
  public void notRosannaShuffle(String[] strings) {

    for (int i = strings.length - 1; i >= 0; i--) {
      // Get random index in the remaining length
      int randomIndex = (int) Math.floor(Math.random() * i);
      if (randomIndex == i) {
        continue;
      } else {
        // If the random index is not i, swap their values
        String tempString = strings[i];
        strings[i] = strings[randomIndex];
        strings[randomIndex] = tempString;
      }
    }
  }

  // Getter for counter
  public int getCounter() {
    return counter;
  }

  // Controlling setter functionality to only reset to 0
  public void resetCounter() {
    counter = 0;
  }

```

```java
  // Message to print upon completing sort which includes counter
  public void printCompletionMessage() {
    System.out.println("Sort complete! Number of comparisons: "
              + counter);
  }
```

3.6 Assignment2

With the methods we have just defined, we are ready to create our sorter.
We'll need some imported libraries: namely, java.io.File to import a file, java.io.FileNotFoundException to account for errors finding the input file, and java.util.Scanner to read the file.

```java
/**
 * A program which reads a constant number of Strings
 * from a file and sorts them using various methods,
 * comparing the efficiency of the sorts by
 * printing the number of comparisons for each.
 */
import java.io.File;
import java.io.FileNotFoundException;
import java.util.Scanner;

public class Assignment2 {
  public static void main(String[] args) {
    final int NUM_OF_ITEMS = 666;          // Length of file as constant
    String[] magicItems = new String[NUM_OF_ITEMS]; // Array of file strings
    MarcusSort sorter = new MarcusSort();      // Instance of MarcusSort

    // Try/catch block for file import and reading
    try {
      File file = new File("magicitems.txt");
      Scanner read = new Scanner(file);
      for (int i = 0; i < NUM_OF_ITEMS; i++) {
        magicItems[i] = read.nextLine();
      }
      read.close();
    } catch (FileNotFoundException e) {
      System.out.println("Whoops! Couldn't find magicitems.txt");
      e.printStackTrace();
    }

    // Shuffle array before beginning sorts
    sorter.notRosannaShuffle(magicItems);

```

```
    // Create second array to store the shuffle
    String[] magicItemsShuffled = new String[NUM_OF_ITEMS];
    System.arraycopy(magicItems, 0, magicItemsShuffled, 0, NUM_OF_ITEMS);

    // Sort using selection sort, print comparisons, reset shuffle
    sorter.selectionSort(magicItems);
    System.arraycopy(magicItemsShuffled, 0, magicItems, 0, NUM_OF_ITEMS);

    // Sort using insertion sort, print comparisons, reset shuffle
    sorter.insertionSort(magicItems);
    System.arraycopy(magicItemsShuffled, 0, magicItems, 0, NUM_OF_ITEMS);

    // Sort using merge sort, print comparisons, reset shuffle
    sorter.mergeSort(magicItems);
    System.arraycopy(magicItemsShuffled, 0, magicItems, 0, NUM_OF_ITEMS);

    // Sort using quicksort, print comparisons
    sorter.quickSort(magicItems);
  }
}
```

4 Results

Due to the pseudorandom nature of notRosannaShuffle(), exact results will vary from execution to execution. The below table shows the results obtained during 5 trials, along with the average number of comparisons and the expected order of growth for each sort.

Sort Performance (in number of comparisons)				
	selectionSort()	insertionSort()	mergeSort()	quickSort()
Trial 1	444,223	334,807	22,004	15,454
Trial 2	444,223	344,659	22,011	15,834
Trial 3	444,223	338,621	22,025	17,330
Trial 4	444,223	336,871	22,051	15,246
Trial 5	444,223	337,681	22,033	14,963
Average	444,223	338,527.8	22,024.8	15,765.4
$O(g(n))$	$O(n^2)$	$O(n^2)$	$O(nlog(n))$	$O(nlog(n))$

The first two algorithms belong to $O(n^2)$ due to their nested loops; ignoring constant factors and lower order terms, they iterate over the array during each iteration of the sort, and the sort iterates for each term in the array. The second pair of algorithms expect to run in $O(nlog(n))$ time due to their trademark "linearithmic" approach, "divide and conquer". The division is logarithmic, and the conquering/merging is linear.