Assignment Two

Nicholas Fiore

Nicholas.Fiore@Marist.edu

October 7, 2022

1 Sorts

The main purpose of this lab was to implement various sorting algorithms and determine their efficiency by their total comparisons and time taken. The efficiency of an algorithm is determined largely by its complexity, categorized by its Big-Oh notation.

The sorts are all static methods contained in a class called *Sort*. The sorts implemented are **selection sort**, **insertion sort**, **merge sort**, and **quick sort**.

1.1 Selection Sort

Selection sort is a simple sorting algorithm that will iteratively create a sub-array that is sorted by finding the smallest value each iteration and placing it where it should be. This is done through a set of nested loops. The first loop iterates from element 1 to element $n-2$ using an iterator variable, i. Within the first loop, the position of a variable, $minPos$, is set to the value of i. This i keeps track of the current position within the overall array, ensuring swaps are always with the correct values.

The inner for loop is where the comparisons occur. This loop iterates from element 1 to element $n-1$. The value stored in array[minPos] is compared to every value after it in the array to see if there is an element smaller than the one currently in $array[minPos]$. If a smaller element (or in our case, lexicographically first) is found, $minPos$ is updated with the position of the new value. After the if, the element in $array[i]$ is swapped with the element in $array[minPos]$, even if it is the same location. $compCounter$, a variable used to count how many comparisons the algorithm made, is also incremented by one after the if statement, followed by the inner loop iterating or terminating.

The outer loop will go through each index in the array except for the last index until the array is sorted. The reason that the loop does not go to the last index is because the way the algorithm works, the last element should already by in the correct position after the $n-1$ element is checked. Allowing the outer loop to check the element at n would only allow it to check it against itself, which is wasted efficiency.

Selection sort will always make $\frac{n^2+n}{2}$ comparisons. This means that it has the complexity of the $O(n^2)$ growth function. This growth function holds true for most algorithms that utilize an inner and outer loop.

For selection sort, the outer loop runs $n-1$ times, and the inner loop runs $\frac{n}{2}$ times. Since the inner loop runs $\frac{n}{2}$ times for every 1 time the outer loop runs, it becomes $(n-1)*\frac{n}{2}$, which simplified is $\frac{n^2+n}{2}$.

1.2 Insertion Sort

Insertion sort is another simple sorting algorithm that uses similar concepts to selection sort, though small optimizations lets insertion sort operate faster *in practice* (but not in theory). Insertion sort creates a sub-array for the sorted values, similar to selection sort. Instead of looking specifically for the next sorted value to place into the array, however, insertion sort will take whatever element is next and then place it sorted into the sub-array, wherever that may be.

The outer loop for insertion sort iterates from element 2 to element n, giving it an iteration count of $n-1$. The loop starts at 2 because the first element is assumed to already be sorted in the sub-array, which it technically is. In the outer loop, a variable called *key* is given the value of $array[i]$. This variable will be used in the value comparisons. Variable j is declared and the inner loop is entered

The inner loop of insertion sort iterates backwards from the position of i to the beginning of the array, or until an element that is more than the key is found. This means that the amount of times that the inner loop iterates varies depending on how the array is shuffled. The element at $array[j+1]$ is then replaced with the element at $array[j]$, and the comparison counter is increased. After the inner loop, the value that was moved over and is in $array[j+1]$ is given the value in *key*.

Unlike selection sort, which has a constant run time depending only on the number of elements in the array, insertion sort can have a variable run time. In the best case scenario, it runs at $n-1$, a complexity of $\Omega(n)$. Worst case, it runs the same as selection sort at $\frac{n^2+n}{2}$. On average, however, it runs at $\frac{n^2+n}{4}$. This gives it a complexity of $O(n^2)$. Insertion sort runs best when the list is already partially sorted.

1.3 Merge Sort

Merge sort is a more complex sort that uses recursion. The basic principle that merge sort uses is **Divide and Conquer**. The array is divided recursively into halves until it is completely separated into sub-arrays of length 1. Then all the sub-arrays are merged back together, being sorted as they are merged.

The first method, $mergeSort()$, is the method that is called in the main program. The method has a single if statement that checks to see if the array passed into the method is not of length 1 by comparing the first index to the last index. If it detects that it is not, the if statement is entered, and a new variable *midIndex* is created by finding the average between the $firstIndex$ and *lastIndex*. The method then calls itself twice, passing the original array but with new values for $firstIndex$ and *lastIndex*, being $firstIndex$ and *midIndex* for the first call and $midIndex + 1$ and *lastIndex* for the second call. This will result in the array and its halves being divided in half recursively. This recursion ends once all sub-arrays are of length 1, and the method continues on to begin calling $merge()$.

The *merge* method is where the comparisons occur for the program. First, the sizes of the sub-arrays are stored in variables $leftSize$ and *rightSize*. New array objects are then made to copy these sub arrays into their own spots in memory. After initializing these copies with their values from their sub-arrays, a while loop is entered. This while loop iterates until either iterator (i for the left, j for the right) reaches the end of its sub-array. The loop contains an if-else statement that basically compares two values from each sub-array, and places the lower value in the original array that the sub-arrays came from. The index of the original array (denoted by k) is then incremented, as well as a counter for the comparisons. After this while loop exits, there are two more optional while loops that will place any remaining elements in the sub-arrays into the original array. This begins to collapse the recursion, and will continue to merge until the first call to $mergeSort()$ is reached and the recursion ends.

Merge sort is varied in its run-time and number of comparisons. It does, however, fall into complexity of $O(n * log_2 n)$.

1.4 Quick Sort

Quick sort is another complex sorting algorithm. Like merge sort, it follows the idea of **Divide and Conquer**. Instead of dividing in half, it will choose an element to use as a pivot and partition the sub-arrays around those pivots (lesser values in the left array, greater values in the right array). Once again, this occurs recursively. The biggest difference is that quick sort does the comparing in the dividing step, while merge sort does it in the merging step.

The initial method *quickSort*() is called in the main program, and checks to see if the passed array is not yet length 1. If it isn't it creates an int called pivot, then assigning the value to be whatever the call of *partition*() will be.

The *partition*() method is designed do both the dividing and comparing. The basic idea is that the entire array is taken in. The pivot value is determined as the value in the last index of the array. There is then a for loop that begins at the location of $firstIndex$ and increments until $lastIndex$. Within the loop, it checks each value at $array[i]$ to see if it is less than the pivot value stored in $pivotVal$. If the value is smaller, it is swapped with the value at the current index stored in *pivotLoc*, and then the counter is incremented.

After the loop, the value at $array[pivotLoc + 1]$ is swapped with the value at $arr[lastIndex]$, placing the pivot value in the middle of the two sub-arrays. This index is then returned, assigned to the *pivot* value in *quickSort*(). After this, the other *quickSort*() calls are made, recursively partitioning and collapsing until the array is sorted.

On average, quick sort has a $O(n * log_2 n)$ run-time. However, in the absolute worst case scenario, it can degrade to $O(n^2)$. This is caused when the pivot chosen is either the greatest or least value in the array. One way to prevent this when choosing a pivot is to randomly select 3 numbers and use the median of those numbers, as that would result in there always being at least one value to the left or right of the pivot until all arrays become length 1 (the only exception is in lists with multiples of the same values).

2 Appendix

2.1 MainTwo.java

```java
//Utility imports
import java.io.*;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.util.Scanner;

/*
 * The purpose of this program is TODO
 */

public class MainTwo {
    public static void main(String[] args) {
        //creates a new file object for the magicitems.txt file
        String fileName = "magicitems.txt";
        File file = new File(fileName);
        //long variable to store the total lines of a .txt
        long totLines;
        //array to hold all the items in the list
        String[] itemList;

```

```java

        /* This try-catch block is for reading in a .txt file, putting each line onto an array, and throwing exceptions if there are any. */
        try {
            //Scanner object for scanning the file
            Scanner fileScanner = new Scanner(file);
            Path path = Paths.get(fileName);

            //grabs the amount of lines within the .txt file and prints it
            totLines = Files.lines(path).count();
            System.out.println("Total␣lines:␣" + totLines + "\n");

            //initializes the size of the array with the total lines in the .txt
            itemList = new String[(int)totLines];


            for (int i = 0; i < totLines; i ++) {
                itemList[i] = fileScanner.nextLine();
            }

            //closes the scanner after use to save resources
            fileScanner.close();


            /*
             * Main section of the program. Calls each different sorting algorithm and prints output. The array is shuffled each time before
             * being sorted, using the shuffle() method that follows the Knuth shuffle algorithm.
             */


            //an array counter to be used for the recursive algorithms
            int[] recurCounter = new int[1];

            //Selection Sort
            Sort.shuffle(itemList);
            Sort.selectionSort(itemList);

            //Insertion Sort
            Sort.shuffle(itemList);
            Sort.insertionSort(itemList);


            //Merge Sort
            recurCounter[0] = 0;
            Sort.shuffle(itemList);
            long mergeStart = System.nanoTime();
            Sort.mergeSort(itemList, 0, (itemList.length-1), recurCounter);
            long mergeEnd = System.nanoTime();
            //Print statements including formatting for mergeSort()
            System.out.println("\033[1mMerge␣Sort\033[0m");
            System.out.print("Number␣of␣comparisons:␣");
            System.out.printf("%,5d␣%n", recurCounter[0]);
            System.out.printf("%-21s", "Time␣elapsed");
            System.out.print(":␣");
            System.out.printf("%,5d", ((mergeEnd - mergeStart) / 1000));
            System.out.println("␣ s \n");

            //Quick Sort
            recurCounter[0] = 0;
            Sort.shuffle(itemList);
            long quickStart = System.nanoTime();
            Sort.quickSort(itemList, 0, itemList.length-1, recurCounter);
            long quickEnd = System.nanoTime();
```

```java
            //Print statements including formatting for quickSort()
            System.out.println("\033[1mQuick␣Sort\033[0m");
            System.out.print("Number␣of␣comparisons:␣");
            System.out.printf("%,5d␣%n", recurCounter[0]);
            System.out.printf("%-21s", "Time␣elapsed");
            System.out.print(":␣");
            System.out.printf("%,5d", ((quickEnd - quickStart) / 1000));
            System.out.println("␣ s \n");



        } catch(FileNotFoundException ex) {
                        System.out.println("Failed␣to␣find␣file:␣" + file.getAbsolutePath());
        } catch(Exception ex) {
            System.out.println("Something␣went␣wrong.");
            System.out.println(ex.getMessage());
            ex.printStackTrace();
        }
    }
}
```

2.2 Sort.java

```java
import java.lang.Math;
/*
 * Class that will be used to store sorting methods. While these don't necessarily need to be in their own class, I prefer it to keep things organized (and potentially be
 * able to reuse in the future)
 */

public class Sort {
    //a method for shuffling an array based on the Knuth shuffle
    public static void shuffle(String[] arr) {
        String temp = "";
        int random;
        for (int i = arr.length - 1; i > 0; i--) {
            random = (int)(Math.random() * i);
            temp = arr[i];
            arr[i] = arr[random];
            arr[random] = temp;
        }
    }

    //Selection sort algorithm
    public static void selectionSort(String[] arr) {
        //these two variables are used for determining the amount of time elapsed during the method execution
        long start = System.nanoTime();
        long end;
        //a counter to be used for counting each comparison within the sort.
        int compCounter = 0;
        //temporary sring used for switching element position in an array.
        String temp = "";
        //the sorting algorithm
        for (int i = 0; i < arr.length - 1; i++) {
            int minPos = i;
            for (int j = i + 1; j < arr.length; j++) {
                if ((arr[j].toUpperCase()).compareTo(arr[minPos].toUpperCase()) < 0)
                    minPos = j;
                temp = arr[i];
                arr[i] = arr[minPos];
                arr[minPos] = temp;
```

```java

                compCounter++;
            }
        }
        end = (System.nanoTime());

        //Print statements including formatting
        System.out.println("\033[1mSelection␣Sort\033[0m");
        System.out.print("Number␣of␣comparisons:␣");
        System.out.printf("%,7d␣%n", compCounter);
        System.out.printf("%-21s", "Time␣elapsed");
        System.out.print(":␣");
        System.out.printf("%,7d", ((end - start) / 1000));
        System.out.println("␣ s \n");
    }

    //Insertion sort
    public static void insertionSort(String[] arr) {
        //these two variables are used for determining the amount of time elapsed during the method execution
        long start = System.nanoTime();
        long end;
        //a counter to be used for counting each comparison within the sort.
        int compCounter = 0;
        //temporary sring used for switching element position in an array.
        String temp = "";
        //the sorting algorithm
        for (int i = 1; i < arr.length; i++) {
            String key = arr[i];
            int j;

            for (j = i - 1; j >= 0 && key.toUpperCase().compareTo(arr[j].toUpperCase()) < 0; j--) {
                arr[j + 1] = arr[j];
                compCounter++;
            }
            arr[j + 1] = key;
        }
        end = (System.nanoTime());

        System.out.println("\033[1mInsertion␣Sort\033[0m");
        System.out.print("Number␣of␣comparisons:␣");
        System.out.printf("%,7d␣%n", compCounter);
        System.out.printf("%-21s", "Time␣elapsed");
        System.out.print(":␣");
        System.out.printf("%,7d", ((end - start) / 1000));
        System.out.println("␣ s \n");
    }

    //the initial method called when doing a merge sort. Recursively calls itself until all sub arrays are of size one, and then reverses
    //the calls through merge to create a fully sorted array
    public static void mergeSort(String[] arr, int firstIndex, int lastIndex, int[] counter) {
        if (firstIndex < lastIndex) {
            int midIndex = (firstIndex + lastIndex) / 2;
            //the new subarrays to be sorted recursively
            mergeSort(arr, firstIndex, midIndex, counter);
            mergeSort(arr, midIndex + 1, lastIndex, counter);
            //merges the arrays back together while sorting them
            merge(arr, firstIndex, midIndex, lastIndex, counter);
        }
    }

```

```
    //A seperate algorithm that takes in two subarrays and combines them while sorting them. This method is used recursively in mergeSort()
    //in order to divide and conquer
    private static void merge(String[] arr, int firstIndex, int midIndex, int lastIndex, int[] counter) {
        //variables to store the size of both subarrays
        int leftSize = midIndex - firstIndex + 1;
        int rightSize = lastIndex - midIndex;

        //creates temporary arrays as copies of the sub arrays within the passed array
        String[] arrLeft = new String[leftSize];
        String[] arrRight = new String[rightSize];

        //initializes the copy arrays
        for (int i = 0; i < leftSize; i++) {
            arrLeft[i] = arr[firstIndex + i];
        }
        for (int j = 0; j < rightSize; j++) {
            arrRight[j] = arr[midIndex + j + 1];
        }

        //variables to store the positions in the subarrays
        int i = 0;
        int j = 0;
        int k = firstIndex;

        //Goes through both sub arrays, and places the earlier word/phrase in the original array at that position, until
        //one of subarrays reaches the end
        while (i < leftSize && j < rightSize) {
            if ((arrLeft[i].toUpperCase()).compareTo(arrRight[j].toUpperCase()) <= 0) {
                arr[k] = arrLeft[i];
                i++;
            } else {
                arr[k] = arrRight[j];
                j++;
            }
            k++;
            counter[0]++;
        }

        //puts any remaining elements into the original array
        while (i < leftSize) {
            arr[k] = arrLeft[i];
            i++;
            k++;
        }

        while (j < rightSize) {
            arr[k] = arrRight[j];
            j++;
            k++;
        }
    }
    //initial call for quickSort. Recursively partitions and calls itself until the list is merged
    public static void quickSort(String[] arr, int firstIndex, int lastIndex, int[] counter) {
        if (firstIndex < lastIndex) {
            int pivot = partition(arr, firstIndex, lastIndex, counter);
            quickSort(arr, firstIndex, pivot-1, counter);
            quickSort(arr, pivot+1, lastIndex, counter);
        }
    }

```

```
    //the partition step of the sort. This is where the comparions and sorting occurs.
        Chooses a pivot and puts other elements on either side of it depending
    //on if it is lesser or greater.
    public static int partition(String[] arr, int firstIndex, int lastIndex, int[] counter)
        {
        String pivotVal = arr[lastIndex];
        int pivotLoc = firstIndex - 1;
        String tempStr = "";
        for (int i = firstIndex; i < lastIndex; i++) {
            if (arr[i].toUpperCase().compareTo(pivotVal.toUpperCase()) <= 0) {
                pivotLoc++;
                tempStr = arr[pivotLoc];
                arr[pivotLoc] = arr[i];
                arr[i] = tempStr;
            }
            counter[0]++;
        }
        tempStr = arr[pivotLoc + 1];
        arr[pivotLoc + 1] = arr[lastIndex];
        arr[lastIndex] = tempStr;

        return pivotLoc + 1;
    }
}
```