Assignment Two

Harrison Zheng
harrison.zheng1@marist.edu

October 9, 2021

1 Assignment Description

For this assignment, students were asked to develop a program to read the names of magic items from a text file into an array, and then sort them using different sorting algorithms. The four algorithms that students were required to use were selection sort, insertion sort, merge sort, and quick sort. In addition, they had to shuffle the array before each sort, and then print the number of comparisons performed after each sort.

2 Results of Sorting Algorithms

This table contains my results from using each sorting method on the magic items array. The table shows that for sorting large sets of data, like the magic items array, merge and quick sort significantly outperform selection and insertion sort. For explanations of each algorithm's asymptotic running time, please go to their respective sections below.

Algorithm	**Comparisons**	**Running Time**
Selection Sort	221,445	O(n^2)
Insertion Sort	110,000*	O(n^2)
Merge Sort	6,302	O(n x log n)
Quick Sort	7,000*	O(n x log n)

* denotes approximate value

3 Selection Sort

Selection sort works by keeping sorted elements at the beginning of the array and unsorted ones in the remaining space. In every iteration, the program looks for the element with the lowest lexicographic value in the unsorted portion of the array and moves it to the front of the sorted portion (Listing 3.1). The running time of this and other algorithms can be found by calculating the running time of each step and then taking the largest function of n. In this algorithm, the largest function is $O(n^2)$, which comes from the two nested loops that have a running time of $O(n)$ each. Thus, the running time of selection sort is $O(n^2)$.

3.1 Selection sort source code listings

```java
//Sort magic items using selection sort
public static void selectionSort(String[] arr) {

    for (int j = 0; j < arr.length - 1; j++) {
        int min = j;
        //Look in the rest of the array for the string with the lowest lexicographic value
        for (int k = j + 1; k < arr.length; k++) {
            //compareTo() returns a negative value if the first string
            //is lexicographically less than the string argument
            int compareValue = arr[k].compareTo(arr[min]);
            comparisons++;
            if (compareValue < 0) {
                //Set min to index of string with lower value
                min = k;
            }
        }

        //Swap item at index j with item with the new minimum lexicographic value
        String temp = arr[j];
        arr[j] = arr[min];
        arr[min] = temp;
    }

    //Print the number of comparisons performed
    System.out.println("Number of Comparisons in Selection Sort: ");
    System.out.println(comparisons);
    comparisons = 0;

}
```

4 Insertion Sort

Insertion sort holds a sorted region and an unsorted region in an array, just like the selection sort algorithm. During each iteration, the program compares the current element with its predecessor, and if the current element is smaller than its predecessor, compare it to the elements before. Move the elements with higher lexicographic values one position up to make space for the swapped element (Listing 4.1). Like selection sort, insertion sort also uses two nested loops. Since this algorithm does not contain any steps with an equal or higher running time, its running time is also $O(n^2)$.

4.1 Insertion sort source code listings

```java
//Sort magic items using insertion sort
public static void insertionSort(String[] arr) {

    for (int i = 1; i < arr.length; ++i) {
        String key = arr[i];
        int j = i - 1;

        //Move strings that have higher lexicographic value than key one positions up
        while (j >= 0 && arr[j].compareTo(key) > 0) {
            comparisons++;
            arr[j + 1] = arr[j];
            j = j - 1;
        }
        arr[j + 1] = key;
    }

    //Print the number of comparisons performed
    System.out.println("Number of Comparisons in Insertion Sort: ");
    System.out.println(comparisons);
    comparisons = 0;

}
```

5 Merge Sort

The merge sort algorithm follows the divide-and-conquer-philosophy when sorting data sets. In my code, the mergeSort function takes the original array and divides its elements among two subarrays and then passes each subarray into the function. It repeats this until the subarrays only have 1 element. The function then passes the subarrays into the merge function, which compares and merges the subarrays back into an array with all the elements in sorted order (Listing 5.1). The first part of the algorithm where it divides an array into subarrays has a running time of O(log n) because the size of the arrays are reduced by half in each iteration and looping through them becomes faster. The second part of the algorithm where it sorts and combines the subarrays has a running time of O(n) because the time it takes to compare elements grows linearly as the number of elements increases. When we put these together, we find that merge sort has a running time of O(n x log n).

5.1 Merge sort source code listings

```java
//Sort magic items using merge sort
public static void mergeSort(String[] arr) {

    if (arr.length >= 2) {
        String[] left = new String[arr.length / 2];
        String[] right = new String[arr.length - arr.length / 2];

        //Populate left array with values in first half of magic items array
        for (int i = 0; i < left.length; i++) {
            left[i] = arr[i];
        }

        //Populate right array with values in second half of magic items array
        for (int i = 0; i < right.length; i++) {
            right[i] = arr[i + arr.length / 2];
        }

        //Recursively divide arrays in half until the size of each is 1
        mergeSort(left);
        mergeSort(right);

        //Conquer by merging/sorting
        merge(arr, left, right);
    }

}

//Conquer the problem by merging the subarrays into the final array in sorted order
public static void merge(String[] arr, String[] left, String[] right) {
    int a = 0;
    int b = 0;
    for (int i = 0; i < arr.length; i++) {
        if (b >= right.length || (a < left.length && left[a].compareTo(right[b]) < 0)) {
            arr[i] = left[a];
            a++;
        } else {
            arr[i] = right[b];
            b++;
        }
        comparisons++;
    }

}
```

6 Quick Sort

Like merge sort, quick sort employs the divide and conquer philosophy to sort data. In my code, the algorithm makes the first element in the array the pivot and partitions the rest of the array around it. It then swaps elements at the counters at each end of the array so that a value smaller than the pivot gets moved to the left side and a value greater than the pivot gets moved to the right side. The counters move inward each time a swap is necessary and the process repeats (Listing 6.1). Similar to merge sort, quick sort recursively divides the array in half (running time of O(log n)) and sorts the elements by comparison in each iteration (running time of O(n)). Even though it divides and sorts elements all at once rather than in separate stages like merge sort, the algorithm's running time can still be expressed as O(n x log n).

6.1 Quick sort source code listings

```
//Sort magic items using quick sort
public static void quickSort(String[] arr, int left, int right) {
    //Initialize counters at either end of the array
    int i = left;
    int j = right;

    if (j - i >= 1) {
        //Set the pivot value to the 1st element in the array
        String pivot = arr[left];

        //Conquer problem by sorting the elements around the pivot value
        while (j > i) {

            //Increment the left counter until the current element has
            //a value greater than the pivot, the left counter reaches
            //the right, or the right counter is farther ahead.
            while (arr[i].compareTo(pivot) <= 0 && i < right && j > i) {
                comparisons++;
                i++;
            }

            //Decrement the right counter until the current element has
            //a value less than the pivot, the right counter reaches the left,
            //or the right counter is farther ahead.
            while (arr[j].compareTo(pivot) >= 0 && j > left && j >= i) {
                comparisons++;
                j--;
            }

            //Swap elements at left and right counters' current positions
            if (j > i) {
                String temp = arr[i];
                arr[i] = arr[j];
                arr[j] = temp;
            }
        }

        //Swap left boundary and pivot with last element in
        //left partition once counters have crossed center
        String temp = arr[left];
        arr[left] = arr[j];
        arr[j] = temp;

        //Recursively sort the left and right partitions
        quickSort(arr, left, j - 1);
        quickSort(arr, j + 1, right);
    }
}
```