# Assignment Two

Tyler Hernandez
Tyler.Hernandez1@marist.edu

October 7, 2022

This page left blank intentionally.

# 1 Main Method

```
public static void main(String[] args) throws Exception {
        Reader reader = new Reader("./magicitems.txt");
        // I know you said to just put "magicitems.txt", but that was not working
        // regardless of whatever directory I inserted the file into.

        // Characters we will ignore when reading from the file.
        char[] ignoreList = { ' ', ',', '.', '\'', '-', '+' };

        // Holds each char of a line
        char[] line = reader.getNextLineOfChars(ignoreList, true);

        // Holds each line as a string
        String[] fullText = new String[0]; // Start the array at no length in case given an empty list.

        // Until we've reached the end of the file, keep looping.
        // Even empty lines in the file will have at least a '\n' character.
        while (line.length > 0) {
            // Though, we must ignore '\n' characters manually in order to keep looping
            // based on if there is a character in the next line.
            if (line[line.length - 1] == '\n') {
                // it seems that '\n' characters only show up at the end of the array (and not
                // in the last line).
                line = Utils.removeLastElementOfArray(line);
            }

            // Expand array by one everytime we add a new value to it.
            fullText = Utils.expandArrayByOne(fullText);
            // Takes line of characters and puts them into fullText as a concatted string.
            fullText[fullText.length - 1] = String.valueOf(line);

            // Grab the next line
            line = reader.getNextLineOfChars(ignoreList, true);

        } // ends while

        String[] ORIGINAL_TEXT = fullText;

        // These sorts will return the amount of comparisons they performed.

        // Insertion Sort!
        insertionCount += insertionSort(fullText);
        fullText = ORIGINAL_TEXT;

        // Selection Sort!
        selectionCount += selectionSort(fullText);
        fullText = ORIGINAL_TEXT;

        // Recursive functions in java are trickier when it comes to returning values,
        // soooo... global variables.

        // Merge Sort!
        MergeSort msort = new MergeSort();
        msort.sort(fullText, 0, fullText.length - 1);
        fullText = ORIGINAL_TEXT;

        // Quick Sort!
        QuickSort qsort = new QuickSort();
        qsort.sort(fullText, 0, fullText.length - 1);
        Utils.printArray(fullText);
        fullText = ORIGINAL_TEXT;

        System.out.println("Insertion sort: " + insertionCount);
        System.out.println("Selection sort: " + selectionCount);
```

```
        System.out.println("Merge sort: " + msort.mergeCount);
        System.out.println("Quick sort: " + qsort.quickCount);

    }
```

### 1.1 Main Method

These past few weeks I have learned about different sorting algorithms and the benefits to each. While they all retrieve the same sorted result, they differ in time complexity heavily under many different circumstances. These circumstances include given an array that is nearly sorted, how well does the algorithm perform? How about an array that is all the same number?

In this assignment you will see four sorting algorithms working towards sorting the same lists. Each will track how many comparisons are made, for us to get a better understanding on why some algorithms work way faster than others.

Here are the results for each sorting algorithm.

Insertion Sort: 114,314 comparisons.

Selection Sort: 221,445 comparisons.

Merge Sort: 3,978 comparisons.

Quick Sort: 17,852 comparisons. (what?? that can't be! – oh yes. with a somewhat randomized pivot, yes.)

### 1.2 Insertion Sort

```
    public static int insertionSort(String[] line) {
        int recordedComparisons = 0;
        for (int ptr1 = 1; ptr1 < line.length; ptr1++) { // we do not need to compare the first index.
            String key = line[ptr1]; // record our character to copy over.
            int ptr2 = ptr1 - 1;

            // Moves over every character greater than our key by one.
            while (ptr2 >= 0 && line[ptr2].compareTo(key) >= 0) { // this counts as two comparisons.
                recordedComparisons++;
                line[ptr2 + 1] = line[ptr2];
                ptr2--;
            }
            line[ptr2 + 1] = key; // paste over (insert) our recorded character.
        }
        return recordedComparisons;
    }
```

Insertion Sort works by iterating over a list and comparing the next element with the previous ones. If that next element is less than our previous ones, it will be inserted into a place that is less than its n+1 element, but still greater or equal to its n-1 element. This algorithm performs at $O(n^2)$

### 1.3 Selection Sort

```
    public static int selectionSort(String[] line) {
        int recordedComparisons = 0;

        // loop over entire array with ptr1.
        for (int ptr1 = 0; ptr1 < line.length - 1; ptr1++) {

            // Retrieve index of the earliest alphabetical character in the subarray.
```

```
            int minimum = ptr1;
            for (int i = ptr1 + 1; i < line.length; i++) {
                if (line[i].compareTo(line[minimum]) < 0) {
                    minimum = i;
                }
                recordedComparisons++;
            }

            // Swap the found minimum element with line[ptr1].
            String temp = line[minimum]; // record before writing over.
            line[minimum] = line[ptr1];
            line[ptr1] = temp;
        }
        return recordedComparisons;
    }
```

Selection Sort works by constantly finding the next minimum element. This differs from Insertion sort, which rather finds the next element and puts it where it should be. They essentially do the same thing, just in opposite order. This, however, makes a huge difference in time complexity depending on the data given. With magicitems, for example, Insertion sort did almost half the comparisons Selection sort did!

## 1.4 Merge Sort

```
class MergeSort {
    // Merges two subarrays of arr[].
    // First subarray is arr[left..mid]
    // Second subarray is arr[mid+1..right]
    int mergeCount = 0;

    // I was having trouble retaining mergeCount in the main class with these recursive functions. I specul
    // it's static type. Regardless, dragging it into this class allowed it to update properly.

    void merge(String arr[], int left, int m, int right) {
        // Find sizes of two subarrays to be merged
        int n1 = m - left + 1;
        int n2 = right - m;

        /* Create temp arrays */
        String LeftArray[] = new String[n1];
        String RightArray[] = new String[n2];

        /* Copy data to temp arrays */
        for (int i = 0; i < n1; ++i)
            LeftArray[i] = arr[left + i];
        for (int j = 0; j < n2; ++j)
            RightArray[j] = arr[m + 1 + j];

        /* Merge the temp arrays */

        // Initial indexes of first and second subarrays
        int i = 0, j = 0;

        // Initial index of merged subarray array
        int k = left;
        while (i < n1 && j < n2) {
            if (LeftArray[i].compareTo(RightArray[j]) <= 0) {
                arr[k] = LeftArray[i];
                i++;
            } else {
                arr[k] = RightArray[j];
                j++;
            }
```

```
            k++;
            mergeCount++;
        }

        /* Copy remaining elements of LeftArray[] if any */
        while (i < n1) {
            arr[k] = LeftArray[i];
            i++;
            k++;
        }

        /* Copy remaining elements of RightArray[] if any */
        while (j < n2) {
            arr[k] = RightArray[j];
            j++;
            k++;
        }
    }

    // Recursive merge sort: Also, divide and conquer!
    void sort(String arr[], int left, int right) {
        if (left < right) {
            mergeCount++;
            // Find the middle point
            int mid = left + (right - left) / 2;

            // Sort each half
            sort(arr, left, mid); // divide in to left array
            sort(arr, mid + 1, right); // divide into right array

            merge(arr, left, mid, right); // stitch arrays back together.
        }
    }

}
```

Merge sort works through the magic of recursion! Merge sort is a divide and conquer style approach to sorting, meaning it will constantly break up the problem into smaller and smaller and so small of a problem that it only has to compare two numbers. Then, it stitches these arrays back up in place. This allows us to perform sorting at O(n log2(n))

## 1.5 Quick Sort

```
import java.util.Random;

class QuickSort {

    int quickCount;

    QuickSort(){
        this.quickCount = 0;
    }

    // A utility function to swap two elements
    void swap(String[] arr, int i, int j) {
        String temp = arr[i];
        arr[i] = arr[j];
        arr[j] = temp;
    }

    /*
     * This function takes last element as pivot, places
```

```java
     * the pivot element at its correct position in sorted
     * array, and places all smaller (smaller than pivot)
     * to left of pivot and all greater elements to right
     * of pivot
     */
    int partition(String[] arr, int low, int high) {

        // pivot
        String pivot = arr[getPivot(arr)];

        // Index of smaller element and
        // indicates the right position
        // of pivot found so far
        int i = (low - 1);

        for (int j = low; j <= high - 1; j++) {

            // If current element is smaller
            // than the pivot
            if (arr[j].compareTo(pivot) <= 0) {
                // Increment index of
                // smaller element
                i++;
                swap(arr, i, j);
            }
            quickCount++;
        }
        swap(arr, i + 1, high);
        return (i + 1);
    }

    /*
     * The main function that implements QuickSort
     * arr[] --> Array to be sorted,
     * low --> Starting index,
     * high --> Ending index
     */
    void sort(String[] arr, int low, int high) {
        if (low < high) {

            // pi is partitioning index, arr[p]
            // is now at right place
            int pi = partition(arr, low, high);

            // Separately sort elements before
            // partition and after partition
            sort(arr, low, pi - 1);
            sort(arr, pi + 1, high);
        }
    }

    // Select small amt of random indexes in list and get median.
    int getPivot(String[] line) {
        int n = line.length;

        if (n <= 0) {
            quickCount++;
            return -1;
        } else if (n <= 3) { // n is between 1 and 3.
            quickCount++;
            // grab median of all 3.

            // grab the latest alphabetical string
            String max = line[0];
            int maxIndex = 0;
```

```

            if (line[1].compareTo(max) > 0) {
                quickCount++;
                max = line[1];
                maxIndex = 1;
            }
            if (line[2].compareTo(max) > 0) {
                quickCount++;
                max = line[2];
                maxIndex = 2;
            }

            // grab the smallest number.

            String min = line[0];
            int minIndex = 0;
            // 0, 3, 1

            if (line[1].compareTo(min) < 0) {
                quickCount++;
                min = line[1];
                minIndex = 1;
            }
            if (line[2].compareTo(min) < 0) {
                quickCount++;
                min = line[2];
                minIndex = 2;
            }

            // if largest is the smallest, return 1. all numbers are already sorted.
            if (minIndex == maxIndex) {
                quickCount++;
                return 1;
            } else {
                quickCount++;
                // deduce number that hasn't been grabbed, that is the median.
                if ((minIndex != 0) && (minIndex != 0))
                    return 0;
                if ((minIndex != 1) && (minIndex != 1))
                    return 1;
                if ((minIndex != 2) && (minIndex != 2))
                    return 2;
            }
        } else { // n > 3. Grab 3 and return the median's index in line[].

            int first = 0;
            int last = n - 1;
            Random rand = new Random();
            int random = rand.nextInt(1, n - 2);

            // Results will return 1, 2, or 3. Based on the first, second, and third
            // parameter given.
            int results = medianOfThree(line[first], line[last], line[random]);
            if (results == 1) {
                return first;
            } else if (results == 2) {
                return last;
            } else {
                return random;
            }
        }
        return 1;
    } // the fact that this function is constant time is hilarious... hopefully
      // itll
      // be used for big arrays!
```

```java

    // Returns indexes of parameters one, two, and three rather than the number
    // themselves.
    public static int medianOfThree(String one, String two, String three) {
        // 6 permutations with three numbers.

        if (one.compareTo(two) > 0) {
            if (two.compareTo(three) >= 0) {
                return 2;
            } else if (one.compareTo(three) >= 0) {
                return 3;
            } else {
                return 1;
            }
        } else {
            if (one.compareTo(three) >= 0) {
                return 1;
            } else if (two.compareTo(three) >= 0) {
                return 3;
            } else {
                return 2;
            }
        }
    }

}
```

Lastly, Quick Sort also uses the magic of recursion! This divide and conquer style approach sorting algorithm is very popular, however not always consistent. It is expected to run in O(n log2(n)) time, however with a bad pivot, this is not always the case. As a matter of fact, if you give Quick Sort the worst possible pivot (a minimum or maximum), it will take $O(n^2)$!

To demonstrate this inconsistency, I've randomized Quick Sort's pivot! :)

### 1.6 Overall

Regardless of the fact that merge sort is my favorite sorting algorithm, each of these has their purpose and has trade offs! While Merge sort was victorious in this round, (and is one of the quickest sorting algorithms) it may fall to Quick Sort when given a proper pivot!