CMPT 435 - Fall 2022 - Dr. Labouseur

Assignment Two – Sorting

Genevieve Anderson
Genevieve.anderson1@marist.edu

October 8, 2022

1 Insertion Sort

In this class I implement the algorithm for insertion sort. First, I declared the String array and the variable for comparisons [3, 4]. Then I built a constructor [7] where I set that the insertion sort will use the string array of the magic items.

Next, I created my "insertionSort()" function [12] which is where the sorting takes place. I used a nested loop. The outer loop is a for loop, and the inner loop is a while loop. The for loop repeats until the end of the magic items list is reached. The while loop is where the swapping actually occurs. While the algorithm is running, if a value that is greater than the "smallPosition" is found, then the value will be placed in its correct spot, then the rest of the array is shifted one place to the right.

Inside of my insertion sort function I also keep track of how many comparisons are made [21]. The comparison count is increased inside of the while loop, since this is where the comparing actually takes place.

Lastly, I have my "toString()" [29] function. This is where I set the result returned equal to the amount of comparisons made during the insertion sort.

I also included some old code at the end [39] that I tried but did not work to show the changes I made.

```
1 public class Insertion {
2 
3     String[] magicItems;  // Declaring that there is a string array of the magic items
4     int comparisons = 0; // Declaring comparisons
5 
6     // Constructor for insertion sort
```

```java
    public Insertion(String[] magicItems) { // String array
        this.magicItems = magicItems; // The string array will be populated with the magic items
    }

    // Function for insertion sort
    public void insertionSort(String[] A) {
        int size = magicItems.length;
        for (int i = 1; i < size; i++) {
            int j = i - 1;
            String smallPosition = magicItems[i];
            // Moving elements that are larger than the smallPosition to one position in front of where it currently is
            while (j >= 0 && smallPosition.compareTo(magicItems[j]) < 0) { // Comparing the small position to the magic items, Checks if j is >= 0 prior
                magicItems[j + 1] = magicItems[j];
                j--;
                comparisons ++;
            }
            // After determining the correct order of values, things need to get shifted
            magicItems[j+1] = smallPosition;
        }
    }

    // Function for to string
    public String toString() {
        String result = ""; // Setting a string equal to what is inside the node
        /*for (int i = 0; i < magicItems.length; i++) {
            result += magicItems[i] + "\n"; // += is adding each result on to the previous result, \n is so that it gets split up by line
        }*/
        result += "Insertion sort comparisons: " + comparisons;
        return result;
    }
}

/*Original code that didn't work

        for (int i = 0; i < n - 1; i++) {
            int smallPosition = i;
            String minValue = magicItems[smallPosition];
            int minIndex = i; // Setting this to i because it is where the loop is beginning
            for (int j = i; j < n - 1; j++) {
                if (magicItems[j].compareTo(minValue) < 0) { // Comparing j to all values, and determining whether or not it is the minimum value in the string of values
                    minValue = magicItems[j];
                    minIndex = j;
                }
            }
            // After determining the correct order of values, things need to get shifted
```

```
            for (int k = i; k < magicItems.length - 1; k++) { //
                magicItems[k] = magicItems[k++];
            }
            magicItems[i] = magicItems[minIndex]; // Moving the smallest value to the front
        }

Programming lab help
Changed my code above to follow this format

        for (int i = 1; i < n; i++) {
            String key = magicItems[i];
            int j = i - 1;

            while (j >= 0 && key.compareTo(magicItems[j]) < 0) {
                magicItems[j + 1] = magicItems[j];
                j--;
            }
            magicItems[j + 1] = key;
        }*/
```

2 Selection Sort

In this class I implement the algorithm for selection sort. First, I declared the String array and the variable for the comparisons [3, 4]. Then I built a constructor [7] where I set that the selection sort will use the string array of the magic items.

Next, I created my ”selectionSort()” function [12] which is where the sorting takes place. I used a nested for loop. The first for loop repeats until the end of the magic items list is reached. I used ”n-2” because in selection sort, we know that the last item will always be in the correct place because of the prior swapping. The inner for loop is where the swapping and comparing takes place. Inside the inner for loop, there is also an if statement. The purpose of the if statement is that if ”j” is less than the ”smallPosition”, then ”j” will take the place of the ”smallPosition”. Then at the end of the function I have set up a temporary string so that the swapping for selection sort can actually take place.

Inside of my insertion sort function I also keep track of how many comparisons are made [17]. The comparison count is increased inside of the for loop, since this is where the comparing actually takes place.

Lastly, I have a toString function. This is where I set the result returned equal to the amount of comparisons made during the insertion sort.

```
1  public class Selection {
2  
3      String[] magicItems;  // Declaring that there is a string array
4      int comparisons = 0; // Declaring the comparisons
5  
6      // Selection sort constructor
7      public Selection(String[] magicItems) {
8          this.magicItems = magicItems; // Initializing the string
       array will be populated with the magic items
9      }
10 
11     // Function for selection sort
12     public void selectionSort(String[] A) {
13         int n = magicItems.length;
14         for (int i = 0; i < n - 2; i++) {
15             int smallPosition = i; // Setting small position
       variable to i
16             for (int j = i + 1; j < n; j++) {
17                 comparisons++; // Counting comparisons here
18                 if (magicItems[j].compareTo(magicItems[
       smallPosition]) < 0) { // If j is less than smallPostion then
       small position will be changed
19                     smallPosition = j; // Small position is now the
        index of the new smallest value found
20                 }
21             }
22             // Swapping the values will require a temporary
       variable
23             String temp = magicItems[smallPosition]; // Setting the
        temporary variable to the small position
24             magicItems[smallPosition] = magicItems[i]; // Swapping
```

```java
            magicItems[i] = temp; // Completing the swap, both values will now be where they belong
        }
    }

    // Function for toString
    public String toString() {
        String result = ""; // Initializing result variable as empty
        /*for (int i = 0; i < magicItems.length; i++) { // Beginning at 0; go until the end of magic items; increment by 1
            // Setting result to a string of all of the magic items
            result += magicItems[i] + "\n"; //+= is adding each result on to the previous result, \n is so that it gets split up by line
        }*/
        result += "Selection sort comparisons: " + comparisons; // Labeling the comparisons
        return result;
    }
}
```

3 Quick Sort

In this class I implement the algorithm for quick sort. First, I declared the String array which will contain magic items and the declared the comparisons variable [4, 5]. Then, I built a constructor [8] where I set that the insertion sort will use the string array of the magic items.

Next, I created my "choosePivot()" function [18] for choosing a pivot value. The function works by choosing a random value located somewhere between the highest value and the lowest value. This is so that the algorithm will not have to pivot around the lowest or highest value, which would take much longer.

Then, I created a simple "swap()" function [27] to perform the swapping needed in the algorithm. This function uses a temporary string array to swap the "lesserIndex" and the "greaterIndex".

Next, I created my "partition()" function [35]. This function splits the values into lesser and greater arrays. Within this function, the "choosePivot" function I created is called, and chooses a pivot value. Inside this function I have a for loop with an if statement inside. This function handles the swapping within the quick sort. I also count comparisons within this function.

Inside of my "partition()" function I also keep track of how many comparisons are made [44].

Then, I have my "quick()" function [51]which is where the algorithm recursively calls upon itself until the list of items is sorted. Lastly, I have my "toString()" function [61]. This is where I set the result returned equal to the amount of comparisons made during the quick sort.

```java
1  import java.util.*;
2
3  public class Quick {
4      String[] magicItems; // Declaring that there is a string array of magic items
5      int comparisons = 0; // Declaring comparisons
6
7      // Quick sort constructor
8      public Quick(String[] magicItems) {
9          this.magicItems = magicItems;
10     }
11
12     // Thought process
13         // First, choose random value as pivot
14         // Compare all other values to the pivot and store them in arrays for lesser values, and greater values
15         // Repeat this process until the magic items are fully sorted
16
17     // To preseve O(nlogn) I will attempt to choose a pivot which is not the highest or lowest value
18     public void choosePivot(String[] A, int low, int high) { //
19         Random random = new Random();
20         int pivot = random.nextInt(high - low) + low; // Doesn't allow for pivot value of the highest or lowest value, but rather a random value inbetween
```

```java
        String temp = A[pivot]; // Setting the temporary variable to pivot
        A[pivot] = A[high]; // Swapping
        A[high] = temp;
    }

    // Function for swapping
    public void swap(String[] A, int lesserIndex, int greaterIndex) { // Array that will be used for swapping, the index of the lesser value, the index of the greater value
        String temp = A[lesserIndex]; // Setting the temporary variable to lesser index
        A[lesserIndex] = A[greaterIndex]; // Swapping
        A[greaterIndex] = temp;
    }

    // Function for partition
    // Splitting the magic items around the pivot into lesser and greater arrays
    public int partition(String[] A, int low, int high) {
        choosePivot(A, low, high); // Choosing pivot value
        String pivot = A[high]; // Setting pivot value
        int i = low - 1;
        for (int j = low; j < high; j++) {
            if (A[j].compareTo(pivot) < 0) { // If the second value is less than the last item in the array (pivot)
                i = i + 1;
                swap(A, i, j); // Swapping the value located at i, with a smaller value
            }
            comparisons++; // Counting comparisons here
        }
        swap(A, i + 1, high); // Swapping so that the pivot value is where it belongs
        return i + 1;
    }

    // Function for quick sort
    public void quick(String[] originalA, int startIndex, int endIndex) { // Parameters start and end index so that when the array is getting split we can reference where to split
        if (startIndex < endIndex) {
            int split = partition(originalA, startIndex, endIndex);
            // Recursion
            quick(originalA, startIndex, split - 1); // Quick method from 1 to end of the values before the pivot value
            quick(originalA, split+1, endIndex); // Quick method from first value after the pivot to the end of the array
        }
    }

    //Function for toString
    public String toString() {
        String result = ""; // Initializing result variable as empty
        /*for (int i = 0; i < magicItems.length; i++) { // Beginning at 0; go until the end of magic items; increment by 1
```

```
            // Setting result to a string of all of the magic items
            result += magicItems[i] + "\n"; //+= is adding each result on to the previous result, \n is so that it gets split up by line
        }*/
        result += "Quick sort comparisons: " + comparisons; // Labeling the comparisons
        return result;
    }
}

/*
Old code for choosing a random value

int randomOne = (int)(Math.random()*A.length); // Using math library, casting the random value as an integer
        int randomTwo = (int)(Math.random()*A.length);
        int randomThree = (int)(Math.random()*A.length);
        int pivot = 0;
        if (randomTwo < randomThree && randomTwo > randomOne) {
            pivot = randomTwo;
        }
        else if (randomTwo < randomOne && randomTwo > randomThree) {
            pivot = randomTwo;
        }
        else if (randomOne < randomTwo && randomOne > randomThree) {
            pivot = randomOne;
        }
        else if (randomOne < randomThree && randomOne > randomTwo) {
            pivot = randomOne;
        }
        else if (randomThree < randomOne && randomThree > randomTwo) {
            pivot = randomThree;
        }
        else if (randomThree < randomTwo && randomThree > randomOne) {
            pivot = randomThree;
        }
        return pivot;



When I tried to quick sort using a queue and it was a bad idea

public void firstPass(Queue less, Queue greater, String pivot) { // Will go through one pass of the quick sort, then pass it on the next queue
    for (int i = 0; i < magicItems.length; i++) {
        Node value = new Node(magicItems[i], null); // putting the value that is being compared into a node, no pointer is needed
        if (magicItems[i].compareTo(pivot) < 0) { // seeing if the magic items are less than the pivot value
```

```
            less.enqueue(value); // Putting the value into the less queue
        }
        else {
            greater.enqueue(value); // Putting the value into the greater queue
        }
    }
}

public void quick(){
    int n = (int)(Math.random()*magicItems.length); // After choosing a random value from magicItems, it will become an integer (casting)
    String pivot = magicItems[n]; // Setting the pivot value to the actual phrase
    Queue less = new Queue(); // Queue for storing values less than the pivot
    Queue greater = new Queue(); // Queue for storing values greater than the pivot
    firstPass(less, greater, pivot); // Executing the first pass of the sort, populates less and greater arrays
    less.tail.setNode(greater.head); // Combining the two queues to prepare for another round of quick sort
    magicQueue = less;
    less = new Queue(); // Re-initializing the queues so that they can be re used in the next round of sorting
    greater = new Queue();
}

public Queue tbd(Queue magicQueue) {
    while (magicQueue.isEmpty() == false){
        Node value = new Node(magicQueue.dequeue().getName(), null); // Making a new node, from the node that was dequeued, and putting the phrase inside the new node. The purpose of this is to clear any pointers that were pointing to the old node.
        if (value.getName().compareTo(pivot) < 0) { // seeing if the magic items are less than the pivot value
            less.enqueue(value); // Putting the value into the less queue
        }
        else {
            greater.enqueue(value); // Putting the value into the greater queue
        }
    }
}

public String getPivot(Queue magicQueue) {
    Queue temp = new Queue();
    temp = (Queue)magicQueue.clone();
    int n = (int)(Math.random()*magicItems.length);
    int count = 0;
    while (count < n) {

    }
}*/
```

4 Merge Sort

In this class I implement the algorithm for merge sort. First, I declared the String array which will contain magic items and the declared the comparisons variable [5, 6]. Then, I built a constructor [9] where I set that the insertion sort will use the string array of the magic items.

Next, I created my "mergeSort()" function [19] which is where the splitting and merging is actually executed. The function recursively calls upon itself until the magic items are sorted.

Then, I created my "merge()" function [31] which is where the algorithm for the merging of the array is. First, I determine the size of the arrays that will be used for the merging [32, 33]. Then, I created temporary arrays [36, 37] for when the algorithm is merging and the arrays are changing each pass. We are only concerned with the original sorted array of all 666 magic items in the end. In my algorithm, I then populate the temporary arrays [40, 43] which prepares them for the comparisons. I created three new integers; an index for the "left" array, an index for the "right" array and an index for the "original array" [46-49]. Then I created a while loop [51], which compares the values of the arrays and merges them together one level at a time, using and changing the indexes I declared, until the array is fully merged again. I have two more small while loops at the end of this function [65, 70] to take care of if the arrays aren't the same size or if there were an odd amount of values.

Inside of my "merge()" function I also keep track of how many comparisons are made [61].

Lastly, I have my "toString()" function [78]. This is where I set the result returned equal to the amount of comparisons made during the merge sort.

```
1 import java.util.*;
2 
3 public class Merge {
4 
5     String[] magicItems;  // Declaring that there is a string array of the magic items
6     int comparisons = 0; // Declaring comparisons
7 
8     // Merge sort constructor
9     public Merge(String[] magicItems) { // String array
10         this.magicItems = magicItems; // The string array will be populated with the magic items
11     }
12 
13     // Thought process
14         // First, split array in half until there are "n" many arrays of size 1
15         // One level at a time, compare values while merging array
16         // Repeat this process until the magic items are fully sorted and there is one complete array
17 
18     // The actual splitting and merging
```

```java
    public void mergeSort(String[] originalA, int startIndex, int endIndex) {
        if (startIndex < endIndex) {
            int middle = startIndex + (endIndex - startIndex)/2; // Dividing to get middle value
            // Recursion
            mergeSort(originalA, startIndex, middle); // Splitting
            mergeSort(originalA, middle + 1, endIndex); // Splitting
            merge(originalA, startIndex, middle, endIndex); // Merging
            comparisons++;
        }
    }

    // The algorithm for the merging
    public void merge(String[] A, int leftIndex, int middle, int rightIndex) {
        int size1 = middle - leftIndex + 1; // Size of first array
        int size2 = rightIndex - middle; // Size of second array

        // Temporary arrays for merging purposes
        String[] left = new String[size1];
        String[] right = new String[size2];

        // Populating the arrays
        for (int i = 0; i < size1; i++) {
            left[i] = A[i + leftIndex]; // This array starts at 0 and goes to the middle
        }
        for (int j = 0; j < size2; j++) {
            right[j] = A[middle + 1 + j]; // j starts at middle + 1, we are going to add values to the right array starting at index 0
        }

        int i = 0; // Index for left array
        int j = 0; // Index for the right array
        int k = leftIndex; // Index for the "original array"

        while (i < size1 && j < size2) {
            if (left[i].compareTo(right[j]) <= 0) {
                A[k] = left[i]; // Sorting, left array value will be put in the original array if the left value is less than the right value
                i++; // Left array gets incremented if this happens
            }
            else {
                A[k] = right[j]; // Right array value will be put into the original array
                j ++; // Right array gets incremented
            }
            k++; // Something is always getting added to the original array for each pass through of the while loop
            comparisons++; // Counting comparisons here
        }

```

```java
        // Two while loops to take care of if the arrrays arent the same size, or if we begin with an array with an odd amount of values
        while (i < size1) {
            A[k] = left[i];
            i++;
            k++;
        }
        while (j < size2) {
            A[k] = right[j];
            j++;
            k++;
        }
    }

    // Function for toString
    public String toString() {
        String result = ""; // Initializing result variable as empty
        /*for (int i = 0; i < magicItems.length; i++) { // Beginning at 0; go until the end of magic items; increment by 1
            // Setting result to a string of all of the magic items
            result += magicItems[i] + "\n"; //+= is adding each result on to the previous result, \n is so that it gets split up by line
        }*/
        result += "Merge sort comparisons: " + comparisons; // Labeling the comparisons
        return result;
    }
}
```

5 Node Class

The node class is where I build the framework for my linked list. The linked list allows for each phrase of the magic items file to be put in their own node. Inside the Node class I am building the framework for each node, the linked list is made up of several nodes.

I declared that inside each node there would be a name, and a next pointer [lines 4, 5] which will be initialized in the constructor. In the constructor [line 11] I initialized the name to "n" and the next pointer to null to begin with. The pointer is what connects the nodes in the linked list.

Within my Node class I have two "getters", and two "setters". The "getters" [lines 17, 21] will provide information for the node, and the "setters" [lines 27, 31] will actually set the name and the pointer in the node. I built "getName()" and "getNext()" which get the name variable, and the pointer for the node. Then I built "setName()" and "setNode()". These are specifically meant for setting the name in the node to a string value, and setting the pointer to the next node.

Lastly, in the Node class I created a "toString()" method [line 36] in order for the program to set the name of the node to the result. This is so that when it comes time to print the magic items phrases from the node it will return the actual phrase rather than the object identifier.

I was able to recycle this code from Assignment 1!

```
1  // Format for linked list
2  // Building the framework for each node. the linked list is going to be made up of several nodes.
3  public class Node { //Creating the node for the linked list
4      String name = ""; //Declaring and initializing the name inside the node
5      Node next = null; //Declaring and initializing the node pointer
6
7
8      //Node constructor
9      //Uppercase N is referring to the node class
10     //Lowercase n is referring to the actual node with the information in it (could be called anything)
11     public Node(String n, Node node) { //The first parameter is for the information the node is holding, the second parameter is for the pointer
12         this.name = n; //Initializing the name
13         this.next = null; //Initializing the pointer
14     }
15
16     //I will now build two getters: information for the node, information for the pointer
17     public String getName() { //Getting the name variable from the node, returns a string
18         return name; //Returns the information from the variable name
19     }
20
```

```
    public Node getNext() { //Returning a node
        return next; //Next because we are calling the pointer next, returns the node from the variable next
    }

    //I will now build two setters: setting the name, and setting the pointer
    //Void because it isnt going to return anything
    public void setName(String n) { //Parameters always go inside parenthesis, I am only updating the name value, so we only need one parameter
        name = n;
    }

    public void setNode(Node m) { //"m" is the node that we are going to set next equal to, for the pointer
        next = m; //I am using m so that the pointer is not null. the pointer will not be null until the end of the linked list.
    }

    //toString so that the program prints what is actually inside the node rather than the object identifier
    public String toString() {
        String result = name; //setting a string equal to what is inside the node
        return result;
    }
}
```

6 Main Program

My main program is where I run my sorts and analyze the outputs! To actually upload the file of magic items, I created a try and catch statement [lines 10-24]. This is also where I changed all of the letters to uppercase and removed all spaces from the magic items. I put all of the phrases into their own node, because a linked list is made up of nodes [line 19].

The following code is where I create and run my sorts [27-46]. Before each sort is run, I shuffle the list of magic items. Then, I run the sort. Lastly, I print what the number of comparisons for the sort. I have created a table below which summarizes the comparisons for each sort!

At the bottom of my main program, I have my "shuffle()" function [51]. This simple algorithm is based on the knuth shuffle. I used the random function to randomly shuffle all elements of the array. Using random also makes it so that the shuffle's output was different each time. Then inside of a for loop I implement the shuffle until the array is fully shuffled. Swapping using a temporary variable is how the shuffle is executed.

I included code at the bottom from when I was testing to see if my sorts worked! Selection sort had the most comparisons and has running time $O(n^2)$. This is because of the nested loop within the algorithm. Insertion sort has the next most comparisons and also has running time $O(n^2)$. Again, this is due to the nested loops within the algorithm. Clearly, these two sorts are not the best option. Next best sort was quick sort. Quick sort has running time $O(n \log(n))$. This is because there are no nested loops. Also, choosing the pivot value is key to having running time of $O(n \log(n))$, which is why we want to avoid having the extreme highest or extreme lowest value. And the best performance was from merge sort. Merge sort also has running time $O(n \log(n))$. This is due to the divide and conquer strategy. This is because we are simply dividing, and then making comparisons in a single while loop.

Selection sort	Insertion sort	Quick sort	Merge sort
221444	108782	6401	6105

```
1 import java.io.File;
2 import java.io.FileNotFoundException;
3 import java.util.Arrays;
4 import java.util.Scanner;
5 import java.util.*;
6
7 public class Main {
8     public static void main (String[] args) {
9         String[] magicItems = new String[666];
10        try { //Trying to find the file
11            File file = new File("magicitems.txt");
12            Scanner sc = new Scanner(file);
13            int index = 0;
14
```

```java
            while (sc.hasNextLine()) {
                String item = sc.nextLine();
                item = item.toUpperCase();
                item = item.replaceAll("\\s+","");
                Node magicItemsNode = new Node(item, null);
                magicItems[index ++] = magicItemsNode.getName();
            }
        }
       catch (FileNotFoundException e) { //If we cant find the file
            e.printStackTrace();
        }

        Insertion insertSort = new Insertion(magicItems);
        Selection selectionSort = new Selection(magicItems);
        Quick quick = new Quick(magicItems);
        Merge merge = new Merge(magicItems);

        shuffle(magicItems); // Shuffling
        selectionSort.selectionSort(magicItems); // Running the algorithm
        System.out.println(selectionSort); // Printing the comparisons

        shuffle(magicItems);
        insertSort.insertionSort(magicItems);
        System.out.println(insertSort);

        shuffle(magicItems);
        quick.quick(magicItems, 0, magicItems.length - 1);
        System.out.println(quick);

        shuffle(magicItems);
        merge.mergeSort(magicItems, 0, magicItems.length - 1);
        System.out.println(merge);
    }

    // Shuffle function
    // Will be called prior to each sort being ran
    public static void shuffle(String[] magicItems){
        Random random = new Random();
        for (int i = magicItems.length - 1; i > 0; i--) { // Shuffling beginning from the last element
            int randInt = random.nextInt(i+1); // Choosing a random index for the shuffling
            String temp = magicItems[i]; // Setting the temporary variable to magicItems[i] to prepare for swap
            magicItems[i] = magicItems[randInt]; // Swapping
            magicItems[randInt] = temp;
        }
    }
}

/*
Code from when I was testing my sorts

Quick magicItemsQuick = new Quick(magicItems);
magicItemsQuick.quick(magicItems, 0, magicItems.length - 1);
```

```
System.out.println(magicItemsQuick);

Merge magicItemsMerge = new Merge(magicItems);
magicItemsMerge.mergeSort(magicItems, 0, magicItems.length - 1);
System.out.println(magicItemsMerge);

Selection magicItemsSelection = new Selection(magicItems);
magicItemsSelection.selection();
System.out.println(magicItemsSelection);

Insertion magicItemsInsertion = new Insertion(magicItems);
magicItemsInsertion.Insertion();
System.out.println(magicItemsInsertion);*/
```

7 LaTeX Source Code

```latex
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% CMPT 435
% Lab Zero
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Short Sectioned Assignment
% LaTeX Template
% Version 1.0 (5/5/12)
%
% This template has been downloaded from: http://www.LaTeXTemplates.com
% Original author: % Frits Wenneker (http://www.howtotex.com)
% License: CC BY-NC-SA 3.0 (http://creativecommons.org/licenses/by-nc-sa/3.0/)
% Modified by Alan G. Labouseur  - alan@labouseur.com
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%----------------------------------------------------------------------------------------
% PACKAGES AND OTHER DOCUMENT CONFIGURATIONS
%----------------------------------------------------------------------------------------

\documentclass[letterpaper, 10pt]{article}

\usepackage[english]{babel} % English language/hyphenation
\usepackage{graphicx}
\usepackage[lined,linesnumbered,commentsnumbered]{algorithm2e}
\usepackage{listings}
\usepackage{fancyhdr} % Custom headers and footers
\pagestyle{fancyplain} % Makes all pages in the document conform to the custom headers and footers
\usepackage{lastpage}
\usepackage{url}

\fancyhead{} % No page header - if you want one, create it in the same way as the footers below
\fancyfoot[L]{} % Empty left footer
\fancyfoot[C]{page \thepage\ of \pageref{LastPage}} % Page numbering for center footer
\fancyfoot[R]{}

\renewcommand{\headrulewidth}{0pt} % Remove header underlines
\renewcommand{\footrulewidth}{0pt} % Remove footer underlines
\setlength{\headheight}{13.6pt} % Customize the height of the header

\usepackage{xcolor}

```

```
\definecolor{codegreen}{rgb}{0,0.6,0}
\definecolor{codegray}{rgb}{0.5,0.5,0.5}
\definecolor{codepurple}{rgb}{0.58,0,0.82}
\definecolor{backcolour}{rgb}{0.95,0.95,0.92}

\lstdefinestyle{mystyle}{
    backgroundcolor=\color{backcolour},
    commentstyle=\color{codegreen},
    keywordstyle=\color{magenta},
    numberstyle=\tiny\color{codegray},
    stringstyle=\color{codepurple},
    basicstyle=\ttfamily\footnotesize,
    breakatwhitespace=false,
    breaklines=true,
    captionpos=b,
    keepspaces=true,
    numbers=left,
    numbersep=5pt,
    showspaces=false,
    showstringspaces=false,
    showtabs=false,
    tabsize=2
}

\lstset{style=mystyle}
% ---------------------------------------------------------------------------------------
% TITLE SECTION
% ---------------------------------------------------------------------------------------

\newcommand{\horrule}[1]{\rule{\linewidth}{#1}} % Create horizontal rule command with 1 argument of height

\title{
   \normalfont \normalsize
   \textsc{CMPT 435 - Fall 2022 - Dr. Labouseur} \\[10pt] % Header stuff.
   \horrule{0.5pt} \\[0.25cm]   % Top horizontal rule
   \huge Assignment Two -- Sorting \\           % Assignment title
   \horrule{0.5pt} \\[0.25cm]   % Bottom horizontal rule
}

\author{Genevieve Anderson \\ \normalsize Genevieve.anderson1@marist.edu}

\date{\normalsize\today} % Today's date.

\begin{document}

\maketitle % Print the title

% ---------------------------------------------------------------------------------------
```

```
%   CONTENT SECTION
%   ----------------------------------------------------------------------------------------

% - -- -  - -- -  - -- -  -

\section{Insertion Sort}

\noindent

    \hspace*{1.5em} In this class I implement the algorithm for insertion sort. First, I declared the String array and the variable for comparisons [3, 4]. Then I built a constructor [7] where I set that the insertion sort will use the string array of the magic items.  \\
    \hspace*{1.5em} Next, I created my "insertionSort()" function [12] which is where the sorting takes place. I used a nested loop. The outer loop is a for loop, and the inner loop is a while loop. The for loop repeats until the end of the magic items list is reached. The while loop is where the swapping actually occurs. While the algorithm is running, if a value that is greater than the "smallPosition" is found, then the value will be placed in its correct spot, then the rest of the array is shifted one place to the right.  \\
    \hspace*{1.5em} Inside of my insertion sort function I also keep track of how many comparisons are made [21]. The comparison count is increased inside of the while loop, since this is where the comparing actually takes place. \\
    \hspace*{1.5em}  Lastly, I have my "toString()" [29] function. This is where I set the result returned equal to the amount of comparisons made during the insertion sort. \\
    \hspace*{1.5em}  I also included some old code at the end [39] that I tried but did not work to show the changes I made. \\

\begin{lstlisting}[language = Java]
public class Insertion {

    String[] magicItems;  // Declaring that there is a string array of the magic items
    int comparisons = 0; // Declaring comparisons

    // Constructor for insertion sort
    public Insertion(String[] magicItems) { // String array
        this.magicItems = magicItems; // The string array will be populated with the magic items
    }

    // Function for insertion sort
    public void insertionSort(String[] A) {
        int size = magicItems.length;
        for (int i = 1; i < size; i++) {
            int j = i - 1;
            String smallPosition = magicItems[i];
            // Moving elements that are larger than the smallPosition to one position in front of where it currently is
            while (j >= 0 && smallPosition.compareTo(magicItems[j])
```

```
 < 0) { // Comparing the small position to the magic items,
Checks if j is >= 0 prior
            magicItems[j + 1] = magicItems[j];
            j--;
            comparisons ++;
        }
        // After determining the correct order of values,
things need to get shifted
        magicItems[j+1] = smallPosition;
    }
}

// Function for to string
public String toString() {
    String result = ""; // Setting a string equal to what is
inside the node
    /*for (int i = 0; i < magicItems.length; i++) {
        result += magicItems[i] + "\n"; // += is adding each
result on to the previous result, \n is so that it gets split
up by line
    }*/
    result += "Insertion sort comparisons: " + comparisons;
    return result;
}
}

/*Original code that didn't work

        for (int i = 0; i < n - 1; i++) {
            int smallPosition = i;
            String minValue = magicItems[smallPosition];
            int minIndex = i; // Setting this to i because it is
where the loop is beginning
            for (int j = i; j < n - 1; j++) {
                if (magicItems[j].compareTo(minValue) < 0) { //
Comparing j to all values, and determining whether or not it is
 the minimum value in the string of values
                    minValue = magicItems[j];
                    minIndex = j;
                }
            }
            // After determining the correct order of values,
things need to get shifted
            for (int k = i; k < magicItems.length - 1; k++) { //
                magicItems[k] = magicItems[k++];
            }
            magicItems[i] = magicItems[minIndex]; // Moving the
smallest value to the front
        }

Programming lab help
Changed my code above to follow this format

        for (int i = 1; i < n; i++) {
            String key = magicItems[i];
            int j = i - 1;

```

```
            while (j >= 0 && key.compareTo(magicItems[j]) < 0) {
                magicItems[j + 1] = magicItems[j];
                j--;
            }
            magicItems[j + 1] = key;
        }*/
\end{lstlisting}



\newpage

\section{Selection Sort}
     \hspace*{1.5em} In this class I implement the algorithm for selection sort. First, I declared the String array and the variable for the comparisons [3, 4]. Then I built a constructor [7] where I set that the selection sort will use the string array of the magic items. \\
     \hspace*{1.5em} Next, I created my "selectionSort()" function [12] which is where the sorting takes place. I used a nested for loop. The first for loop repeats until the end of the magic items list is reached. I used "n-2" because in selection sort, we know that the last item will always be in the correct place because of the prior swapping. The inner for loop is where the swapping and comparing takes place. Inside the inner for loop, there is also an if statement. The purpose of the if statement is that if "j" is less than the "smallPosition", then "j" will take the place of the "smallPosition". Then at the end of the function I have set up a temporary string so that the swapping for selection sort can actually take place. \\
     \hspace*{1.5em} Inside of my insertion sort function I also keep track of how many comparisons are made [17]. The comparison count is increased inside of the for loop, since this is where the comparing actually takes place. \\
     \hspace*{1.5em}  Lastly, I have a toString function. This is where I set the result returned equal to the amount of comparisons made during the insertion sort. \\

\begin{lstlisting}[language = Java]
public class Selection {

    String[] magicItems;  // Declaring that there is a string array
    int comparisons = 0; // Declaring the comparisons

    // Selection sort constructor
    public Selection(String[] magicItems) {
        this.magicItems = magicItems; // Initializing the string array will be populated with the magic items
    }

    // Function for selection sort
    public void selectionSort(String[] A) {
        int n = magicItems.length;
        for (int i = 0; i < n - 2; i++) {
            int smallPosition = i; // Setting small position variable to i
            for (int j = i + 1; j < n; j++) {
```

```
                comparisons ++; // Counting comparisons here
                if (magicItems[j].compareTo(magicItems[smallPosition]) < 0) { // If j is less than smallPostion then small position will be changed
                    smallPosition = j; // Small position is now the index of the new smallest value found
                }
            }
            // Swapping the values will require a temporary variable
            String temp = magicItems[smallPosition]; // Setting the temporary variable to the small position
            magicItems[smallPosition] = magicItems[i]; // Swapping
            magicItems[i] = temp; // Completing the swap, both values will now be where they belong
        }
    }

    // Function for toString
    public String toString() {
        String result = ""; // Initializing result variable as empty
        /*for (int i = 0; i < magicItems.length; i++) { // Beginning at 0; go until the end of magic items; increment by 1
            // Setting result to a string of all of the magic items
            result += magicItems[i] + "\n"; //+= is adding each result on to the previous result, \n is so that it gets split up by line
        }*/
        result += "Selection sort comparisons: " + comparisons; // Labeling the comparisons
        return result;
    }
}
\end{lstlisting}
\newpage

\section{Quick Sort}
    \hspace*{1.5em} In this class I implement the algorithm for quick sort. First, I declared the String array which will contain magic items and the declared the comparisons variable [4, 5]. Then, I built a constructor [8] where I set that the insertion sort will use the string array of the magic items. \\
    \hspace*{1.5em} Next, I created my "choosePivot()" function [18] for choosing a pivot value. The function works by choosing a random value located somewhere between the highest value and the lowest value. This is so that the algorithm will not have to pivot around the lowest or highest value, which would take much longer. \\
    \hspace*{1.5em} Then, I created a simple "swap()" function [27] to perform the swapping needed in the algorithm. This function uses a temporary string array to swap the "lesserIndex" and the "greaterIndex". \\
    \hspace*{1.5em} Next, I created my "partition()" function [35]. This function splits the values into lesser and greater arrays. Within this function, the "choosePivot" function I created is called, and chooses a pivot value. Inside this function I have
```

```
a for loop with an if statement inside. This function handles the swapping within the quick sort. I also count comparisons within this function. \\
    \hspace*{1.5em} Inside of my "partition()" function I also keep track of how many comparisons are made [44]. \\
    \hspace*{1.5em} Then, I have my "quick()" function [51]which is where the algorithm recursively calls upon itself until the list of items is sorted.
    \hspace*{1.5em}  Lastly, I have my "toString()" function [61]. This is where I set the result returned equal to the amount of comparisons made during the quick sort. \\

\begin{lstlisting}[language = Java]
import java.util.*;

public class Quick {
    String[] magicItems; // Declaring that there is a string array of magic items
    int comparisons = 0; // Declaring comparisons

    // Quick sort constructor
    public Quick(String[] magicItems) {
        this.magicItems = magicItems;
    }

    // Thought process
        // First, choose random value as pivot
        // Compare all other values to the pivot and store them in arrays for lesser values, and greater values
        // Repeat this process until the magic items are fully sorted

    // To preseve O(nlogn) I will attempt to choose a pivot which is not the highest or lowest value
    public void choosePivot(String[] A, int low, int high) { //
        Random random = new Random();
        int pivot = random.nextInt(high - low) + low; // Doesn't allow for pivot value of the highest or lowest value, but rather a random value inbetween
        String temp = A[pivot]; // Setting the temporary variable to pivot
        A[pivot] = A[high]; // Swapping
        A[high] = temp;
    }

    // Function for swapping
    public void swap(String[] A, int lesserIndex, int greaterIndex) { // Array that will be used for swapping, the index of the lesser value, the index of the greater value
        String temp = A[lesserIndex]; // Setting the temporary variable to lesser index
        A[lesserIndex] = A[greaterIndex]; // Swapping
        A[greaterIndex] = temp;
    }

    // Function for partition
    // Splitting the magic items around the pivot into lesser and
```

```
    greater arrays
    public int partition(String[] A, int low, int high) {
        choosePivot(A, low, high); // Choosing pivot value
        String pivot = A[high]; // Setting pivot value
        int i = low - 1;
        for (int j = low; j < high; j++) {
            if (A[j].compareTo(pivot) < 0) { // If the second value is less than the last item in the array (pivot)
                i = i + 1;
                swap(A, i, j); // Swapping the value located at i, with a smaller value
            }
            comparisons++; // Counting comparisons here
        }
        swap(A, i + 1, high); // Swapping so that the pivot value is where it belongs
        return i + 1;
    }

    // Function for quick sort
    public void quick(String[] originalA, int startIndex, int endIndex) { // Parameters start and end index so that when the array is getting split we can reference where to split
        if (startIndex < endIndex) {
            int split = partition(originalA, startIndex, endIndex);
            // Recursion
            quick(originalA, startIndex, split - 1); // Quick method from 1 to end of the values before the pivot value
            quick(originalA, split+1, endIndex); // Quick method from first value after the pivot to the end of the array
        }
    }

    //Function for toString
    public String toString() {
        String result = ""; // Initializing result variable as empty
        /*for (int i = 0; i < magicItems.length; i++) { // Beginning at 0; go until the end of magic items; increment by 1
            // Setting result to a string of all of the magic items
            result += magicItems[i] + "\n"; //+= is adding each result on to the previous result, \n is so that it gets split up by line
        }*/
        result += "Quick sort comparisons: " + comparisons; // Labeling the comparisons
        return result;
    }
}

/*
Old code for choosing a random value

int randomOne = (int)(Math.random()*A.length); // Using math library, casting the random value as an integer
        int randomTwo = (int)(Math.random()*A.length);
        int randomThree = (int)(Math.random()*A.length);
```

```java
        int pivot = 0;
        if (randomTwo < randomThree && randomTwo > randomOne) {
            pivot = randomTwo;
        }
        else if (randomTwo < randomOne && randomTwo > randomThree) {
            pivot = randomTwo;
        }
        else if (randomOne < randomTwo && randomOne > randomThree) {
            pivot = randomOne;
        }
        else if (randomOne < randomThree && randomOne > randomTwo) {
            pivot = randomOne;
        }
        else if (randomThree < randomOne && randomThree > randomTwo) {
            pivot = randomThree;
        }
        else if (randomThree < randomTwo && randomThree > randomOne) {
            pivot = randomThree;
        }
        return pivot;



When I tried to quick sort using a queue and it was a bad idea

public void firstPass(Queue less, Queue greater, String pivot) { // Will go through one pass of the quick sort, then pass it on the next queue
    for (int i = 0; i < magicItems.length; i++) {
        Node value = new Node(magicItems[i], null); // putting the value that is being compared into a node, no pointer is needed
        if (magicItems[i].compareTo(pivot) < 0) { // seeing if the magic items are less than the pivot value
            less.enqueue(value); // Putting the value into the less queue
        }
        else {
            greater.enqueue(value); // Putting the value into the greater queue
        }
    }
}

public void quick(){
    int n = (int)(Math.random()*magicItems.length); // After choosing a random value from magicItems, it will become an integer (casting)
    String pivot = magicItems[n]; // Setting the pivot value to the actual phrase
    Queue less = new Queue(); // Queue for storing values less than the pivot
    Queue greater = new Queue(); // Queue for storing values
```

```
    greater than the pivot
    firstPass(less, greater, pivot); // Executing the first pass of the sort, populates less and greater arrays
    less.tail.setNode(greater.head); // Combining the two queues to prepare for another round of quick sort
    magicQueue = less;
    less = new Queue(); // Re-initializing the queues so that they can be re used in the next round of sorting
    greater = new Queue();
}

public Queue tbd(Queue magicQueue) {
    while (magicQueue.isEmpty() == false){
        Node value = new Node(magicQueue.dequeue().getName(), null); // Making a new node, from the node that was dequeued, and putting the phrase inside the new node. The purpose of this is to clear any pointers that were pointing to the old node.
        if (value.getName().compareTo(pivot) < 0) { // seeing if the magic items are less than the pivot value
            less.enqueue(value); // Putting the value into the less queue
        }
        else {
            greater.enqueue(value); // Putting the value into the greater queue
        }
    }
}

public String getPivot(Queue magicQueue) {
    Queue temp = new Queue();
    temp = (Queue)magicQueue.clone();
    int n = (int)(Math.random()*magicItems.length);
    int count = 0;
    while (count < n) {

    }
}*/
\end{lstlisting}

\newpage

\section{Merge Sort}
    \hspace*{1.5em} In this class I implement the algorithm for merge sort. First, I declared the String array which will contain magic items and the declared the comparisons variable [5, 6]. Then, I built a constructor [9] where I set that the insertion sort will use the string array of the magic items.\\
    \hspace*{1.5em} Next, I created my "mergeSort()" function [19] which is where the splitting and merging is actually executed. The function recursively calls upon itself until the magic items are sorted. \\
    \hspace*{1.5em} Then, I created my "merge()" function [31] which is where the algorithm for the merging of the array is. First, I determine the size of the arrays that will be used for the merging [32, 33]. Then, I created temporary arrays [36, 37] for when the algorithm is merging and the arrays are
```

```
    changing each pass. We are only concerned with the original sorted array of all 666 magic items in the end. In my algorithm, I then populate the temporary arrays [40, 43] which prepares them for the comparisons. I created three new integers; an index for the "left" array, an index for the "right" array and an index for the "original array" [46-49]. Then I created a while loop [51], which compares the values of the arrays and merges them together one level at a time, using and changing the indexes I declared, until the array is fully merged again. I have two more small while loops at the end of this function [65, 70] to take care of if the arrays aren't the same size or if there were an odd amount of values. \\
    \hspace*{1.5em} Inside of my "merge()" function I also keep track of how many comparisons are made [61]. \\
    \hspace*{1.5em}  Lastly, I have my "toString()" function [78]. This is where I set the result returned equal to the amount of comparisons made during the merge sort. \\

\begin{lstlisting}[language = Java]
import java.util.*;

public class Merge {

    String[] magicItems;  // Declaring that there is a string array of the magic items
    int comparisons = 0; // Declaring comparisons

    // Merge sort constructor
    public Merge(String[] magicItems) { // String array
        this.magicItems = magicItems; // The string array will be populated with the magic items
    }

    // Thought process
        // First, split array in half until there are "n" many arrays of size 1
        // One level at a time, compare values while merging array
        // Repeat this process until the magic items are fully sorted and there is one complete array

    // The actual splitting and merging
    public void mergeSort(String[] originalA, int startIndex, int endIndex) {
        if (startIndex < endIndex) {
            int middle = startIndex + (endIndex - startIndex)/2; // Dividing to get middle value
            // Recursion
            mergeSort(originalA, startIndex, middle); // Splitting
            mergeSort(originalA, middle + 1, endIndex); // Splitting
            merge(originalA, startIndex, middle, endIndex); // Merging
            comparisons++;
        }
    }

    // The algorithm for the merging
```

```java
    public void merge(String[] A, int leftIndex, int middle, int rightIndex) {
        int size1 = middle - leftIndex + 1; // Size of first array
        int size2 = rightIndex - middle; // Size of second array

        // Temporary arrays for merging purposes
        String[] left = new String[size1];
        String[] right = new String[size2];

        // Populating the arrays
        for (int i = 0; i < size1; i++) {
            left[i] = A[i + leftIndex]; // This array starts at 0 and goes to the middle
        }
        for (int j = 0; j < size2; j++) {
            right[j] = A[middle + 1 + j]; // j starts at middle + 1, we are going to add values to the right array starting at index 0
        }

        int i = 0; // Index for left array
        int j = 0; // Index for the right array
        int k = leftIndex; // Index for the "original array"

        while (i < size1 && j < size2) {
            if (left[i].compareTo(right[j]) <= 0) {
                A[k] = left[i]; // Sorting, left array value will be put in the original array if the left value is less than the right value
                i++; // Left array gets incremented if this happens
            }
            else {
                A[k] = right[j]; // Right array value will be put into the original array
                j ++; // Right array gets incremented
            }
            k++; // Something is always getting added to the original array for each pass through of the while loop
            comparisons++; // Counting comparisons here
        }

        // Two while loops to take care of if the arrrays arent the same size, or if we begin with an array with an odd amount of values
        while (i < size1) {
            A[k] = left[i];
            i++;
            k++;
        }
        while (j < size2) {
            A[k] = right[j];
            j++;
            k++;
        }
    }

    // Function for toString
```

```
    public String toString() {
        String result = ""; // Initializing result variable as empty
        /*for (int i = 0; i < magicItems.length; i++) { // Beginning at 0; go until the end of magic items; increment by 1
            // Setting result to a string of all of the magic items
            result += magicItems[i] + "\n"; //+= is adding each result on to the previous result, \n is so that it gets split up by line
        }*/
        result += "Merge sort comparisons: " + comparisons; // Labeling the comparisons
        return result;
    }
}
\end{lstlisting}

\newpage

\section{Node Class}
    \hspace*{1.5em} The node class is where I build the framework for my linked list. The linked list allows for each phrase of the magic items file to be put in their own node. Inside the Node class I am building the framework for each node, the linked list is made up of several nodes.  \\
    \hspace*{1.5em} I declared that inside each node there would be a name, and a next pointer [lines 4, 5] which will be initialized in the constructor. In the constructor [line 11] I initialized the name to "n" and the next pointer to null to begin with. The pointer is what connects the nodes in the linked list. \\
    \hspace*{1.5em} Within my Node class I have two "getters", and two "setters". The "getters" [lines 17, 21] will provide information for the node, and the "setters" [lines 27, 31] will actually set the name and the pointer in the node. I built "getName()" and "getNext()" which get the name variable, and the pointer for the node. Then I built "setName()" and "setNode()". These are specifically meant for setting the name in the node to a string value, and setting the pointer to the next node. \\
    \hspace*{1.5em} Lastly, in the Node class I created a "toString()" method [line 36] in order for the program to set the name of the node to the result. This is so that when it comes time to print the magic items phrases from the node it will return the actual phrase rather than the object identifier. \\
    \hspace*{1.5em} I was able to recycle this code from Assignment #1! \\

\begin{lstlisting}[language = Java]
// Format for linked list
// Building the framework for each node. the linked list is going to be made up of several nodes.
public class Node { //Creating the node for the linked list
    String name = ""; //Declaring and initializing the name inside the node
    Node next = null; //Declaring and initializing the node pointer

```

```

    //Node constructor
    //Uppercase N is referring to the node class
    //Lowercase n is referring to the actual node with the information in it (could be called anything)
    public Node(String n, Node node) { //The first parameter is for the information the node is holding, the second parameter is for the pointer
        this.name = n; //Initializing the name
        this.next = null; //Initializing the pointer
    }

    //I will now build two getters: information for the node, information for the pointer
    public String getName() { //Getting the name variable from the node, returns a string
        return name; //Returns the information from the variable name
    }

    public Node getNext() { //Returning a node
        return next; //Next because we are calling the pointer next, returns the node from the variable next
    }

    //I will now build two setters: setting the name, and setting the pointer
    //Void because it isnt going to return anything
    public void setName(String n) { //Parameters always go inside parenthesis, I am only updating the name value, so we only need one parameter
        name = n;
    }

    public void setNode(Node m) { //"m" is the node that we are going to set next equal to, for the pointer
        next = m; //I am using m so that the pointer is not null. the pointer will not be null until the end of the linked list.
    }

    //toString so that the program prints what is actually inside the node rather than the object identifier
    public String toString() {
        String result = name; //setting a string equal to what is inside the node
        return result;
    }
}
\end{lstlisting}

\newpage

\section{Main Program}
    \hspace*{1.5em} My main program is where I run my sorts and analyze the outputs! To actually upload the file of magic items, I created a try and catch statement [lines 10-24]. This is also where I changed all of the letters to uppercase and
```

```
removed all spaces from the magic items. I put all of the phrases into their own node, because a linked list is made up of nodes [line 19].\\
    \hspace*{1.5em} The following code is where I create and run my sorts [27-46]. Before each sort is run, I shuffle the list of magic items. Then, I run the sort. Lastly, I print what the number of comparisons for the sort. I have created a table below which summarizes the comparisons for each sort! \\
    \hspace*{1.5em} At the bottom of my main program, I have my "shuffle()" function [51]. This simple algorithm is based on the knuth shuffle. I used the random function to randomly shuffle all elements of the array. Using random also makes it so that the shuffle's output was different each time. Then inside of a for loop I implement the shuffle until the array is fully shuffled. Swapping using a temporary variable is how the shuffle is executed.\\
    \hspace*{1.5em} I included code at the bottom from when I was testing to see if my sorts worked! Selection sort had the most comparisons and has running time $O(n^2)$. This is because of the nested loop within the algorithm. Insertion sort has the next most comparisons and also has running time $O(n^2)$. Again, this is due to the nested loops within the algorithm. Clearly, these two sorts are not the best option. Next best sort was quick sort. Quick sort has running time $O(n \log(n))$. This is because there are no nested loops. Also, choosing the pivot value is key to having running time of $O(n \log(n))$, which is why we want to avoid having the extreme highest or extreme lowest value. And the best performance was from merge sort. Merge sort also has running time $O(n \log(n))$. This is due to the divide and conquer strategy. This is because we are simply dividing, and then making comparisons in a single while loop. \\

\begin{center}
\begin{tabular}{||c c c c||}
\hline
Selection sort & Insertion sort & Quick sort & Merge sort \\ [0.5ex]
\hline\hline
221444   & 108782    &6401&   6105 \\  [1ex]
\hline
\end{tabular}
\end{center}

\begin{lstlisting}[language = Java]
import java.io.File;
import java.io.FileNotFoundException;
import java.util.Arrays;
import java.util.Scanner;
import java.util.*;

public class Main {
    public static void main (String[] args) {
        String[] magicItems = new String[666];
        try { //Trying to find the file
            File file = new File("magicitems.txt");
            Scanner sc = new Scanner(file);
```

```
            int index = 0;

            while (sc.hasNextLine()) {
                String item = sc.nextLine();
                item = item.toUpperCase();
                item = item.replaceAll("\\s+","");
                Node magicItemsNode = new Node(item, null);
                magicItems[index ++] = magicItemsNode.getName();
            }
        }
       catch (FileNotFoundException e) { //If we cant find the file
            e.printStackTrace();
        }

        Insertion insertSort = new Insertion(magicItems);
        Selection selectionSort = new Selection(magicItems);
        Quick quick = new Quick(magicItems);
        Merge merge = new Merge(magicItems);

        shuffle(magicItems); // Shuffling
        selectionSort.selectionSort(magicItems); // Running the algorithm
        System.out.println(selectionSort); // Printing the comparisons

        shuffle(magicItems);
        insertSort.insertionSort(magicItems);
        System.out.println(insertSort);

        shuffle(magicItems);
        quick.quick(magicItems, 0, magicItems.length - 1);
        System.out.println(quick);

        shuffle(magicItems);
        merge.mergeSort(magicItems, 0, magicItems.length - 1);
        System.out.println(merge);
    }

    // Shuffle function
    // Will be called prior to each sort being ran
    public static void shuffle(String[] magicItems){
        Random random = new Random();
        for (int i = magicItems.length - 1; i > 0; i--) { // Shuffling beginning from the last element
            int randInt = random.nextInt(i+1); // Choosing a random index for the shuffling
            String temp = magicItems[i]; // Setting the temporary variable to magicItems[i] to prepare for swap
            magicItems[i] = magicItems[randInt]; // Swapping
            magicItems[randInt] = temp;
        }
    }
}

/*
Code from when I was testing my sorts

```

```
Quick magicItemsQuick = new Quick(magicItems);
magicItemsQuick.quick(magicItems, 0, magicItems.length - 1);
System.out.println(magicItemsQuick);

Merge magicItemsMerge = new Merge(magicItems);
magicItemsMerge.mergeSort(magicItems, 0, magicItems.length - 1);
System.out.println(magicItemsMerge);

Selection magicItemsSelection = new Selection(magicItems);
magicItemsSelection.selection();
System.out.println(magicItemsSelection);

Insertion magicItemsInsertion = new Insertion(magicItems);
magicItemsInsertion.Insertion();
System.out.println(magicItemsInsertion);*/
\end{lstlisting}
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% The commands below print the source code starting
% on a new page. Comment out or delete if you do not want to
% include the source code in your document.
%
\newpage
\section{\LaTeX \hspace{.25em} Source Code}
\lstset{basicstyle=\footnotesize\ttfamily, breaklines=true, language=[LaTeX]{TeX}}
\lstinputlisting{main.tex} % <---CHANGE TO CORRECT FILENAME
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

\end{document}
```