

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  
4     of the magic items  
5     int comparisons = 0; // Declaring comparisons  
6  
7     // Constructor for insertion sort
```

```

7   public Insertion(String[] magicItems) { // String array
8       this.magicItems = magicItems; // The string array will be
        populated with the magic items
9   }

10
11   // Function for insertion sort
12   public void insertionSort(String[] A) {
13       int size = magicItems.length;
14       for (int i = 1; i < size; i++) {
15           int j = i - 1;
16           String smallPosition = magicItems[i];
17           // Moving elements that are larger than the
        smallPosition to one position in front of where it currently is
18           while (j >= 0 && smallPosition.compareTo(magicItems[j])
        < 0) { // Comparing the small position to the magic items,
        Checks if j is >= 0 prior
19               magicItems[j + 1] = magicItems[j];
20               j--;
21               comparisons++;
22           }
23           // After determining the correct order of values,
        things need to get shifted
24           magicItems[j+1] = smallPosition;
25       }
26   }

27
28   // Function for to string
29   public String toString() {
30       String result = ""; // Setting a string equal to what is
        inside the node
31       /*for (int i = 0; i < magicItems.length; i++) {
32           result += magicItems[i] + "\n"; // += is adding each
        result on to the previous result, \n is so that it gets split
        up by line
33       }*/
34       result += "Insertion sort comparisons: " + comparisons;
35       return result;
36   }
37 }

38
39 /*Original code that didn't work
40
41     for (int i = 0; i < n - 1; i++) {
42         int smallPosition = i;
43         String minValue = magicItems[smallPosition];
44         int minIndex = i; // Setting this to i because it is
        where the loop is beginning
45         for (int j = i; j < n - 1; j++) {
46             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
47                 minValue = magicItems[j];
48                 minIndex = j;
49             }
50         }
51         // After determining the correct order of values,
        things need to get shifted

```

```

52         for (int k = i; k < magicItems.length - 1; k++) { //
53             magicItems[k] = magicItems[k+1];
54         }
55         magicItems[i] = magicItems[minIndex]; // Moving the
smallest value to the front
56     }
57
58 Programming lab help
59 Changed my code above to follow this format
60
61     for (int i = 1; i < n; i++) {
62         String key = magicItems[i];
63         int j = i - 1;
64
65         while (j >= 0 && key.compareTo(magicItems[j]) < 0) {
66             magicItems[j + 1] = magicItems[j];
67             j--;
68         }
69         magicItems[j + 1] = key;
70     }*/

```

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
9         array will be populated with the magic items
10    }
11
12    // Function for selection sort
13    public void selectionSort(String[] A) {
14        int n = magicItems.length;
15        for (int i = 0; i < n - 2; i++) {
16            int smallPosition = i; // Setting small position
17            variable to i
18            for (int j = i + 1; j < n; j++) {
19                comparisons++; // Counting comparisons here
20                if (magicItems[j].compareTo(magicItems[
21                    smallPosition]) < 0) { // If j is less than smallPosition then
22                    small position will be changed
23                    smallPosition = j; // Small position is now the
24                    index of the new smallest value found
25                }
26            }
27            // Swapping the values will require a temporary
28            variable
29            String temp = magicItems[smallPosition]; // Setting the
30            temporary variable to the small position
31            magicItems[smallPosition] = magicItems[i]; // Swapping
```

```

25         magicItems[i] = temp; // Completing the swap, both
        values will now be where they belong
26     }
27 }
28
29 // Function for toString
30 public String toString() {
31     String result = ""; // Initializing result variable as
    empty
32     /*for (int i = 0; i < magicItems.length; i++) { //
    Beginning at 0; go until the end of magic items; increment by 1
33         // Setting result to a string of all of the magic items
34         result += magicItems[i] + "\n"; //+= is adding each
    result on to the previous result, \n is so that it gets split
    up by line
35     }*/
36     result += "Selection sort comparisons: " + comparisons; //
    Labeling the comparisons
37     return result;
38 }
39 }

```

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.

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

```

21     String temp = A[pivot]; // Setting the temporary variable
    to pivot
22     A[pivot] = A[high]; // Swapping
23     A[high] = temp;
24 }
25
26 // Function for swapping
27 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
28     String temp = A[lesserIndex]; // Setting the temporary
    variable to lesser index
29     A[lesserIndex] = A[greaterIndex]; // Swapping
30     A[greaterIndex] = temp;
31 }
32
33 // Function for partition
34 // Splitting the magic items around the pivot into lesser and
    greater arrays
35 public int partition(String[] A, int low, int high) {
36     choosePivot(A, low, high); // Choosing pivot value
37     String pivot = A[high]; // Setting pivot value
38     int i = low - 1;
39     for (int j = low; j < high; j++) {
40         if (A[j].compareTo(pivot) < 0) { // If the second value
            is less than the last item in the array (pivot)
41             i = i + 1;
42             swap(A, i, j); // Swapping the value located at i,
            with a smaller value
43         }
44         comparisons++; // Counting comparisons here
45     }
46     swap(A, i + 1, high); // Swapping so that the pivot value
    is where it belongs
47     return i + 1;
48 }
49
50 // Function for quick sort
51 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
52     if (startIndex < endIndex) {
53         int split = partition(originalA, startIndex, endIndex);
54         // Recursion
55         quick(originalA, startIndex, split - 1); // Quick
            method from 1 to end of the values before the pivot value
56         quick(originalA, split+1, endIndex); // Quick method
            from first value after the pivot to the end of the array
57     }
58 }
59
60 //Function for toString
61 public String toString() {
62     String result = ""; // Initializing result variable as
    empty
63     /*for (int i = 0; i < magicItems.length; i++) { //
    Beginning at 0; go until the end of magic items; increment by 1

```

```

64         // Setting result to a string of all of the magic items
65         result += magicItems[i] + "\n"; //+= is adding each
        result on to the previous result, \n is so that it gets split
        up by line
66     }*/
67     result += "Quick sort comparisons: " + comparisons; //
    Labeling the comparisons
68     return result;
69 }
70 }
71
72 /*
73 Old code for choosing a random value
74
75 int randomOne = (int)(Math.random()*A.length); // Using math
    library, casting the random value as an integer
76     int randomTwo = (int)(Math.random()*A.length);
77     int randomThree = (int)(Math.random()*A.length);
78     int pivot = 0;
79     if (randomTwo < randomThree && randomTwo > randomOne) {
80         pivot = randomTwo;
81     }
82     else if (randomTwo < randomOne && randomTwo > randomThree)
83     {
84         pivot = randomTwo;
85     }
86     else if (randomOne < randomTwo && randomOne > randomThree)
87     {
88         pivot = randomOne;
89     }
90     else if (randomOne < randomThree && randomOne > randomTwo)
91     {
92         pivot = randomOne;
93     }
94     else if (randomThree < randomOne && randomThree > randomTwo
95     ) {
96         pivot = randomThree;
97     }
98     else if (randomThree < randomTwo && randomThree > randomOne
99     ) {
100         pivot = randomThree;
101     }
102     return pivot;
103
104 When I tried to quick sort using a queue and it was a bad idea
105
106 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
107     for (int i = 0; i < magicItems.length; i++) {
108         Node value = new Node(magicItems[i], null); // putting the
        value that is being compared into a node, no pointer is needed
109         if (magicItems[i].compareTo(pivot) < 0) { // seeing if the
        magic items are less than the pivot value

```



```

107         less.enqueue(value); // Putting the value into the less
        queue
    }
108     }
109     else {
110         greater.enqueue(value); // Putting the value into the
        greater queue
111     }
112 }
113 }
114
115 public void quick(){
116     int n = (int)(Math.random()*magicItems.length); // After
        choosing a random value from magicItems, it will become an
        integer (casting)
117     String pivot = magicItems[n]; // Setting the pivot value to the
        actual phrase
118     Queue less = new Queue(); // Queue for storing values less than
        the pivot
119     Queue greater = new Queue(); // Queue for storing values
        greater than the pivot
120     firstPass(less, greater, pivot); // Executing the first pass of
        the sort, populates less and greater arrays
121     less.tail.setNode(greater.head); // Combining the two queues to
        prepare for another round of quick sort
122     magicQueue = less;
123     less = new Queue(); // Re-initializing the queues so that they
        can be re used in the next round of sorting
124     greater = new Queue();
125 }
126
127 public Queue tbd(Queue magicQueue) {
128     while (magicQueue.isEmpty() == false){
129         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.
130         if (value.getName().compareTo(pivot) < 0) { // seeing if
        the magic items are less than the pivot value
131             less.enqueue(value); // Putting the value into the less
        queue
132         }
133         else {
134             greater.enqueue(value); // Putting the value into the
        greater queue
135         }
136     }
137 }
138
139 public String getPivot(Queue magicQueue) {
140     Queue temp = new Queue();
141     temp = (Queue)magicQueue.clone();
142     int n = (int)(Math.random()*magicItems.length);
143     int count = 0;
144     while (count < n) {
145
146     }
147 }*/

```

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
6     // of the magic items
7     int comparisons = 0; // Declaring comparisons
8
9     // Merge sort constructor
10    public Merge(String[] magicItems) { // String array
11        this.magicItems = magicItems; // The string array will be
12        // populated with the magic items
13    }
14
15    // Thought process
16    // First, split array in half until there are "n" many
17    // arrays of size 1
18    // One level at a time, compare values while merging array
19    // Repeat this process until the magic items are fully
20    // sorted and there is one complete array
21
22    // The actual splitting and merging
```

```

19 public void mergeSort(String[] originalA, int startIndex, int
endIndex) {
20     if (startIndex < endIndex) {
21         int middle = startIndex + (endIndex - startIndex)/2; //
Dividing to get middle value
22         // Recursion
mergeSort(originalA, startIndex, middle); // Splitting
23 mergeSort(originalA, middle + 1, endIndex); //
24 Splitting
merge(originalA, startIndex, middle, endIndex); //
Merging
25         comparisons++;
26     }
27 }
28
29
30 // The algorithm for the merging
31 public void merge(String[] A, int leftIndex, int middle, int
rightIndex) {
32     int size1 = middle - leftIndex + 1; // Size of first array
33     int size2 = rightIndex - middle; // Size of second array
34
35     // Temporary arrays for merging purposes
36     String[] left = new String[size1];
37     String[] right = new String[size2];
38
39     // Populating the arrays
40     for (int i = 0; i < size1; i++) {
41         left[i] = A[i + leftIndex]; // This array starts at 0
and goes to the middle
42     }
43     for (int j = 0; j < size2; j++) {
44         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
45     }
46
47     int i = 0; // Index for left array
48     int j = 0; // Index for the right array
49     int k = leftIndex; // Index for the "original array"
50
51     while (i < size1 && j < size2) {
52         if (left[i].compareTo(right[j]) <= 0) {
53             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
54             i++; // Left array gets incremented if this happens
55         }
56         else {
57             A[k] = right[j]; // Right array value will be put
into the original array
58             j++; // Right array gets incremented
59         }
60         k++; // Something is always getting added to the
original array for each pass through of the while loop
61         comparisons++; // Counting comparisons here
62     }
63 }

```

```

64      // Two while loops to take care of if the arrays arent the
        same size, or if we begin with an array with an odd amount of
        values
65      while (i < size1) {
66          A[k] = left[i];
67          i++;
68          k++;
69      }
70      while (j < size2) {
71          A[k] = right[j];
72          j++;
73          k++;
74      }
75  }
76
77  // Function for toString
78  public String toString() {
79      String result = ""; // Initializing result variable as
        empty
80      /*for (int i = 0; i < magicItems.length; i++) { //
        Beginning at 0; go until the end of magic items; increment by 1
81          // Setting result to a string of all of the magic items
82          result += magicItems[i] + "\n"; //+= is adding each
        result on to the previous result, \n is so that it gets split
        up by line
83      }*/
84      result += "Merge sort comparisons: " + comparisons; //
        Labeling the comparisons
85      return result;
86  }
87 }

```

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
```

```

21     public Node getNext() { //Returning a node
22         return next; //Next because we are calling the pointer next
           , returns the node from the variable next
23     }
24
25     //I will now build two setters: setting the name, and setting
           the pointer
26     //Void because it isnt going to return anything
27     public void setName(String n) { //Parameters always go inside
           parenthesis, I am only updating the name value, so we only need
           one parameter
28         name = n;
29     }
30
31     public void setNode(Node m) { //"m" is the node that we are
           going to set next equal to, for the pointer
32         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.
33     }
34
35     //toString so that the program prints what is actually inside
           the node rather than the object identifier
36     public String toString() {
37         String result = name; //setting a string equal to what is
           inside the node
38         return result;
39     }
40 }

```

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        }
```

```

15         while (sc.hasNextLine()) {
16             String item = sc.nextLine();
17             item = item.toUpperCase();
18             item = item.replaceAll("\\s+", "");
19             Node magicItemsNode = new Node(item, null);
20             magicItems[index++] = magicItemsNode.getName();
21         }
22     }
23     catch (FileNotFoundException e) { //If we cant find the file
24         e.printStackTrace();
25     }
26
27     Insertion insertSort = new Insertion(magicItems);
28     Selection selectionSort = new Selection(magicItems);
29     Quick quick = new Quick(magicItems);
30     Merge merge = new Merge(magicItems);
31
32     shuffle(magicItems); // Shuffling
33     selectionSort.selectionSort(magicItems); // Running the
algorithm
34     System.out.println(selectionSort); // Printing the
comparisons
35
36     shuffle(magicItems);
37     insertSort.insertionSort(magicItems);
38     System.out.println(insertSort);
39
40     shuffle(magicItems);
41     quick.quick(magicItems, 0, magicItems.length - 1);
42     System.out.println(quick);
43
44     shuffle(magicItems);
45     merge.mergeSort(magicItems, 0, magicItems.length - 1);
46     System.out.println(merge);
47 }
48
49 // Shuffle function
50 // Will be called prior to each sort being ran
51 public static void shuffle(String[] magicItems){
52     Random random = new Random();
53     for (int i = magicItems.length - 1; i > 0; i--) { //
Shuffling beginning from the last element
54         int randInt = random.nextInt(i+1); // Choosing a random
index for the shuffling
55         String temp = magicItems[i]; // Setting the temporary
variable to magicItems[i] to prepare for swap
56         magicItems[i] = magicItems[randInt]; // Swapping
57         magicItems[randInt] = temp;
58     }
59 }
60 }
61
62 /*
63 Code from when I was testing my sorts
64
65 Quick magicItemsQuick = new Quick(magicItems);
66 magicItemsQuick.quick(magicItems, 0, magicItems.length - 1);

```



```
67 System.out.println(magicItemsQuick);
68
69 Merge magicItemsMerge = new Merge(magicItems);
70 magicItemsMerge.mergeSort(magicItems, 0, magicItems.length - 1);
71 System.out.println(magicItemsMerge);
72
73 Selection magicItemsSelection = new Selection(magicItems);
74 magicItemsSelection.selection();
75 System.out.println(magicItemsSelection);
76
77 Insertion magicItemsInsertion = new Insertion(magicItems);
78 magicItemsInsertion.Insertion();
79 System.out.println(magicItemsInsertion);*/
```

7 L^AT_EX Source Code

```
1 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
2 %
3 % CMPT 435
4 % Lab Zero
5 %
6 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
7
8 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
9 % Short Sectioned Assignment
10 % LaTeX Template
11 % Version 1.0 (5/5/12)
12 %
13 % This template has been downloaded from: http://www.LaTeXTemplates
   .com
14 % Original author: % Frits Wenneker (http://www.howtotex.com)
15 % License: CC BY-NC-SA 3.0 (http://creativecommons.org/licenses/by-
   nc-sa/3.0/)
16 % Modified by Alan G. Labouseur - alan@labouseur.com
17 %
18 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
19
20 %
   -----
21
22 % PACKAGES AND OTHER DOCUMENT CONFIGURATIONS
23 %
   -----
24
25 \documentclass[letterpaper, 10pt]{article}
26
27 \usepackage[english]{babel} % English language/hyphenation
28 \usepackage{graphicx}
29 \usepackage[lined,linesnumbered,commentsnumbered]{algorithm2e}
30 \usepackage{listings}
31 \usepackage{fancyhdr} % Custom headers and footers
32 \pagestyle{fancyplain} % Makes all pages in the document conform to
   the custom headers and footers
33 \usepackage{lastpage}
34 \usepackage{url}
35
36 \fancyhead{} % No page header - if you want one, create it in the
   same way as the footers below
37 \fancyfoot[L]{} % Empty left footer
38 \fancyfoot[C]{page \thepage\ of \pageref{LastPage}} % Page
   numbering for center footer
39 \fancyfoot[R]{}
40
41 \renewcommand{\headrulewidth}{0pt} % Remove header underlines
42 \renewcommand{\footrulewidth}{0pt} % Remove footer underlines
43 \setlength{\headheight}{13.6pt} % Customize the height of the
   header
44
45 \usepackage{xcolor}
```

```

46 \definecolor{codegreen}{rgb}{0,0.6,0}
47 \definecolor{codegray}{rgb}{0.5,0.5,0.5}
48 \definecolor{codepurple}{rgb}{0.58,0,0.82}
49 \definecolor{backcolour}{rgb}{0.95,0.95,0.92}
50
51 \lstdefinestyle{mystyle}{
52     backgroundcolor=\color{backcolour},
53     commentstyle=\color{codegreen},
54     keywordstyle=\color{magenta},
55     numberstyle=\tiny\color{codegray},
56     stringstyle=\color{codepurple},
57     basicstyle=\ttfamily\footnotesize,
58     breakatwhitespace=false,
59     breaklines=true,
60     captionpos=b,
61     keepspaces=true,
62     numbers=left,
63     numbersep=5pt,
64     showspace=false,
65     showstringspaces=false,
66     showtabs=false,
67     tabsize=2
68 }
69
70 \lstset{style=mystyle}
71 %
72 % -----
73 % TITLE SECTION
74 % -----
75 \newcommand{\horrule}[1]{\rule{\linewidth}{#1}} % Create horizontal
76     rule command with 1 argument of height
77
78 \title{
79     \normalfont \normalsize
80     \textsc{CMPT 435 - Fall 2022 - Dr. Labouseur} \\\[10pt] % Header
81     stuff.
82     \horrule{0.5pt} \\\[0.25cm] % Top horizontal rule
83     \huge Assignment Two -- Sorting \\\[0.25cm] % Assignment title
84     \horrule{0.5pt} \\\[0.25cm] % Bottom horizontal rule
85 }
86
87 \author{Genevieve Anderson \\\ \normalsize Genevieve.anderson1
88     @marist.edu}
89
90 \date{\normalsize\today} % Today's date.
91
92 \begin{document}
93 \maketitle % Print the title
94 %
95 % -----

```

```

94 % CONTENT SECTION
95 %
96 -----
97 % - - - - -
98
99 \section{Insertion Sort}
100
101 \noindent
102
103 \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. \\
104 \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. \\
105 \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. \\
106 \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. \\
107 \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. \\
108
109 \begin{lstlisting}[language = Java]
110 public class Insertion {
111
112     String[] magicItems; // Declaring that there is a string array
of the magic items
113     int comparisons = 0; // Declaring comparisons
114
115     // Constructor for insertion sort
116     public Insertion(String[] magicItems) { // String array
117         this.magicItems = magicItems; // The string array will be
populated with the magic items
118     }
119
120     // Function for insertion sort
121     public void insertionSort(String[] A) {
122         int size = magicItems.length;
123         for (int i = 1; i < size; i++) {
124             int j = i - 1;
125             String smallPosition = magicItems[i];
126             // Moving elements that are larger than the
smallPosition to one position in front of where it currently is
127             while (j >= 0 && smallPosition.compareTo(magicItems[j])

```

```

128         < 0) { // Comparing the small position to the magic items,
Checks if j is >= 0 prior
        magicItems[j + 1] = magicItems[j];
129         j--;
130         comparisons ++;
131     }
132     // After determining the correct order of values,
things need to get shifted
133     magicItems[j+1] = smallPosition;
134 }
135 }
136
137 // Function for to string
138 public String toString() {
139     String result = ""; // Setting a string equal to what is
inside the node
140     /*for (int i = 0; i < magicItems.length; i++) {
141         result += magicItems[i] + "\n"; // += is adding each
result on to the previous result, \n is so that it gets split
up by line
142     }*/
143     result += "Insertion sort comparisons: " + comparisons;
144     return result;
145 }
146 }
147
148 /*Original code that didn't work
149
150     for (int i = 0; i < n - 1; i++) {
151         int smallPosition = i;
152         String minValue = magicItems[smallPosition];
153         int minIndex = i; // Setting this to i because it is
where the loop is beginning
154         for (int j = i; j < n - 1; j++) {
155             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
156                 minValue = magicItems[j];
157                 minIndex = j;
158             }
159         }
160         // After determining the correct order of values,
things need to get shifted
161         for (int k = i; k < magicItems.length - 1; k++) { //
162             magicItems[k] = magicItems[k+1];
163         }
164         magicItems[i] = magicItems[minIndex]; // Moving the
smallest value to the front
165     }
166
167 Programming lab help
168 Changed my code above to follow this format
169
170     for (int i = 1; i < n; i++) {
171         String key = magicItems[i];
172         int j = i - 1;
173

```

```

174         while (j >= 0 && key.compareTo(magicItems[j]) < 0) {
175             magicItems[j + 1] = magicItems[j];
176             j--;
177         }
178         magicItems[j + 1] = key;
179     }*/
180 \end{lstlisting}
181
182
183
184 \newpage
185
186 \section{Selection Sort}
187     \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. \\
188     \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. \\
189     \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. \\
190     \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. \\
191
192 \begin{lstlisting}[language = Java]
193 public class Selection {
194
195     String[] magicItems; // Declaring that there is a string array
196     int comparisons = 0; // Declaring the comparisons
197
198     // Selection sort constructor
199     public Selection(String[] magicItems) {
200         this.magicItems = magicItems; // Initializing the string
array will be populated with the magic items
201     }
202
203     // Function for selection sort
204     public void selectionSort(String[] A) {
205         int n = magicItems.length;
206         for (int i = 0; i < n - 2; i++) {
207             int smallPosition = i; // Setting small position
variable to i
208             for (int j = i + 1; j < n; j++) {

```

```

209         comparisons++; // Counting comparisons here
210         if (magicItems[j].compareTo(magicItems[
smallPosition]) < 0) { // If j is less than smallPosition then
small position will be changed
211             smallPosition = j; // Small position is now the
index of the new smallest value found
212         }
213     }
214     // Swapping the values will require a temporary
variable
215     String temp = magicItems[smallPosition]; // Setting the
temporary variable to the small position
216     magicItems[smallPosition] = magicItems[i]; // Swapping
217     magicItems[i] = temp; // Completing the swap, both
values will now be where they belong
218 }
219 }
220
221 // Function for toString
222 public String toString() {
223     String result = ""; // Initializing result variable as
empty
224     /*for (int i = 0; i < magicItems.length; i++) { //
Beginning at 0; go until the end of magic items; increment by 1
225         // Setting result to a string of all of the magic items
226         result += magicItems[i] + "\n"; //+= is adding each
result on to the previous result, \n is so that it gets split
up by line
227     }*/
228     result += "Selection sort comparisons: " + comparisons; //
Labeling the comparisons
229     return result;
230 }
231 }
232 \end{lstlisting}
233 \newpage
234
235 \section{Quick Sort}
236     \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. \\
237     \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. \\
238     \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". \\
239     \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. \\
240 \hspace*{1.5em} Inside of my "partition()" function I also keep
    track of how many comparisons are made [44]. \\
241 \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.
242 \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. \\
243
244 \begin{lstlisting}[language = Java]
245 import java.util.*;
246
247 public class Quick {
248     String[] magicItems; // Declaring that there is a string array
    of magic items
249     int comparisons = 0; // Declaring comparisons
250
    // Quick sort constructor
251     public Quick(String[] magicItems) {
252         this.magicItems = magicItems;
253     }
254
    // Thought process
255     // First, choose random value as pivot
256     // Compare all other values to the pivot and store them in
257     arrays for lesser values, and greater values
258     // Repeat this process until the magic items are fully
259     sorted
260
    // To preserve  $O(n \log n)$  I will attempt to choose a pivot which
    is not the highest or lowest value
261     public void choosePivot(String[] A, int low, int high) { //
262         Random random = new Random();
263         int pivot = random.nextInt(high - low) + low; // Doesn't
264         allow for pivot value of the highest or lowest value, but
        rather a random value inbetween
265         String temp = A[pivot]; // Setting the temporary variable
        to pivot
266         A[pivot] = A[high]; // Swapping
267         A[high] = temp;
268     }
269
    // Function for swapping
270     public void swap(String[] A, int lesserIndex, int greaterIndex)
271     { // Array that will be used for swapping, the index of the
        lesser value, the index of the greater value
272         String temp = A[lesserIndex]; // Setting the temporary
        variable to lesser index
273         A[lesserIndex] = A[greaterIndex]; // Swapping
274         A[greaterIndex] = temp;
275     }
276
    // Function for partition
277     // Splitting the magic items around the pivot into lesser and
278

```



```

greater arrays
279 public int partition(String[] A, int low, int high) {
280     choosePivot(A, low, high); // Choosing pivot value
281     String pivot = A[high]; // Setting pivot value
282     int i = low - 1;
283     for (int j = low; j < high; j++) {
284         if (A[j].compareTo(pivot) < 0) { // If the second value
            is less than the last item in the array (pivot)
285             i = i + 1;
286             swap(A, i, j); // Swapping the value located at i,
with a smaller value
287         }
288         comparisons++; // Counting comparisons here
289     }
290     swap(A, i + 1, high); // Swapping so that the pivot value
is where it belongs
291     return i + 1;
292 }
293
294 // Function for quick sort
295 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
296     if (startIndex < endIndex) {
297         int split = partition(originalA, startIndex, endIndex);
298         // Recursion
299         quick(originalA, startIndex, split - 1); // Quick
method from 1 to end of the values before the pivot value
300         quick(originalA, split+1, endIndex); // Quick method
from first value after the pivot to the end of the array
301     }
302 }
303
304 //Function for toString
305 public String toString() {
306     String result = ""; // Initializing result variable as
empty
307     /*for (int i = 0; i < magicItems.length; i++) { //
Beginning at 0; go until the end of magic items; increment by 1
308         // Setting result to a string of all of the magic items
309         result += magicItems[i] + "\n"; //+= is adding each
result on to the previous result, \n is so that it gets split
up by line
310     }*/
311     result += "Quick sort comparisons: " + comparisons; //
Labeling the comparisons
312     return result;
313 }
314 }
315
316 /*
317 Old code for choosing a random value
318
319 int randomOne = (int)(Math.random()*A.length); // Using math
library, casting the random value as an integer
320     int randomTwo = (int)(Math.random()*A.length);
321     int randomThree = (int)(Math.random()*A.length);

```

```

322     int pivot = 0;
323     if (randomTwo < randomThree && randomTwo > randomOne) {
324         pivot = randomTwo;
325     }
326     else if (randomTwo < randomOne && randomTwo > randomThree)
327     {
328         pivot = randomTwo;
329     }
330     else if (randomOne < randomTwo && randomOne > randomThree)
331     {
332         pivot = randomOne;
333     }
334     else if (randomOne < randomThree && randomOne > randomTwo)
335     {
336         pivot = randomOne;
337     }
338     else if (randomThree < randomOne && randomThree > randomTwo)
339     {
340         pivot = randomThree;
341     }
342     else if (randomThree < randomTwo && randomThree > randomOne)
343     {
344         pivot = randomThree;
345     }
346     return pivot;
347 }
348
349 When I tried to quick sort using a queue and it was a bad idea
350
351 public void firstPass(Queue less, Queue greater, String pivot) { //
352     Will go through one pass of the quick sort, then pass it on
353     the next queue
354     for (int i = 0; i < magicItems.length; i++) {
355         Node value = new Node(magicItems[i], null); // putting the
356         value that is being compared into a node, no pointer is needed
357         if (magicItems[i].compareTo(pivot) < 0) { // seeing if the
358         magic items are less than the pivot value
359             less.enqueue(value); // Putting the value into the less
360             queue
361         }
362         else {
363             greater.enqueue(value); // Putting the value into the
364             greater queue
365         }
366     }
367 }
368
369 public void quick(){
370     int n = (int)(Math.random()*magicItems.length); // After
371     choosing a random value from magicItems, it will become an
372     integer (casting)
373     String pivot = magicItems[n]; // Setting the pivot value to the
374     actual phrase
375     Queue less = new Queue(); // Queue for storing values less than
376     the pivot
377     Queue greater = new Queue(); // Queue for storing values

```

```

greater than the pivot
364 firstPass(less, greater, pivot); // Executing the first pass of
the sort, populates less and greater arrays
365 less.tail.setNode(greater.head); // Combining the two queues to
prepare for another round of quick sort
366 magicQueue = less;
367 less = new Queue(); // Re-initializing the queues so that they
can be re used in the next round of sorting
368 greater = new Queue();
369 }
370
371 public Queue tbd(Queue magicQueue) {
372     while (magicQueue.isEmpty() == false){
373         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.
374         if (value.getName().compareTo(pivot) < 0) { // seeing if
the magic items are less than the pivot value
375             less.enqueue(value); // Putting the value into the less
queue
376         }
377         else {
378             greater.enqueue(value); // Putting the value into the
greater queue
379         }
380     }
381 }
382
383 public String getPivot(Queue magicQueue) {
384     Queue temp = new Queue();
385     temp = (Queue)magicQueue.clone();
386     int n = (int)(Math.random()*magicItems.length);
387     int count = 0;
388     while (count < n) {
389
390     }
391 }*/
392 \end{lstlisting}
393
394 \newpage
395
396 \section{Merge Sort}
397     \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.\\
398     \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. \\
399     \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

```

```

400     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. \\
401     \hspace*{1.5em} Inside of my "merge()" function I also keep
        track of how many comparisons are made [61]. \\
402     \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. \\
403 \begin{lstlisting}[language = Java]
404 import java.util.*;
405
406 public class Merge {
407
408     String[] magicItems; // Declaring that there is a string array
        of the magic items
409     int comparisons = 0; // Declaring comparisons
410
411     // Merge sort constructor
412     public Merge(String[] magicItems) { // String array
413         this.magicItems = magicItems; // The string array will be
        populated with the magic items
414     }
415
416     // Thought process
417         // First, split array in half until there are "n" many
        arrays of size 1
418         // One level at a time, compare values while merging array
419         // Repeat this process until the magic items are fully
        sorted and there is one complete array
420
421     // The actual splitting and merging
422     public void mergeSort(String[] originalA, int startIndex, int
        endIndex) {
423         if (startIndex < endIndex) {
424             int middle = startIndex + (endIndex - startIndex)/2; //
        Dividing to get middle value
425             // Recursion
426             mergeSort(originalA, startIndex, middle); // Splitting
427             mergeSort(originalA, middle + 1, endIndex); //
        Splitting
428             merge(originalA, startIndex, middle, endIndex); //
        Merging
429             comparisons++;
430         }
431     }
432
433     // The algorithm for the merging

```

```

434 public void merge(String[] A, int leftIndex, int middle, int
rightIndex) {
435     int size1 = middle - leftIndex + 1; // Size of first array
436     int size2 = rightIndex - middle; // Size of second array
437
438     // Temporary arrays for merging purposes
439     String[] left = new String[size1];
440     String[] right = new String[size2];
441
442     // Populating the arrays
443     for (int i = 0; i < size1; i++) {
444         left[i] = A[i + leftIndex]; // This array starts at 0
and goes to the middle
445     }
446     for (int j = 0; j < size2; j++) {
447         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
448     }
449
450     int i = 0; // Index for left array
451     int j = 0; // Index for the right array
452     int k = leftIndex; // Index for the "original array"
453
454     while (i < size1 && j < size2) {
455         if (left[i].compareTo(right[j]) <= 0) {
456             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
457             i++; // Left array gets incremented if this happens
458         }
459         else {
460             A[k] = right[j]; // Right array value will be put
into the original array
461             j++; // Right array gets incremented
462         }
463         k++; // Something is always getting added to the
original array for each pass through of the while loop
464         comparisons++; // Counting comparisons here
465     }
466
467     // Two while loops to take care of if the arrays arent the
same size, or if we begin with an array with an odd amount of
values
468     while (i < size1) {
469         A[k] = left[i];
470         i++;
471         k++;
472     }
473     while (j < size2) {
474         A[k] = right[j];
475         j++;
476         k++;
477     }
478 }
479
480 // Function for toString

```

```

481     public String toString() {
482         String result = ""; // Initializing result variable as
empty
483         /*for (int i = 0; i < magicItems.length; i++) { //
Beginning at 0; go until the end of magic items; increment by 1
484             // Setting result to a string of all of the magic items
485             result += magicItems[i] + "\n"; //+= is adding each
result on to the previous result, \n is so that it gets split
up by line
486         }*/
487         result += "Merge sort comparisons: " + comparisons; //
Labeling the comparisons
488         return result;
489     }
490 }
491 \end{lstlisting}
492
493 \newpage
494
495 \section{Node Class}
496     \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. \\
497     \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. \\
498     \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. \\
499     \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. \\
500     \hspace*{1.5em} I was able to recycle this code from Assignment
#1! \\
501
502 \begin{lstlisting}[language = Java]
503 // Format for linked list
504 // Building the framework for each node. the linked list is going
to be made up of several nodes.
505 public class Node { //Creating the node for the linked list
506     String name = ""; //Declaring and initializing the name inside
the node
507     Node next = null; //Declaring and initializing the node pointer
508

```

```

509
510 //Node constructor
511 //Uppercase N is referring to the node class
512 //Lowercase n is referring to the actual node with the
    information in it (could be called anything)
513 public Node(String n, Node node) { //The first parameter is for
    the information the node is holding, the second parameter is
    for the pointer
514     this.name = n; //Initializing the name
515     this.next = null; //Initializing the pointer
516 }
517
518 //I will now build two getters: information for the node,
    information for the pointer
519 public String getName() { //Getting the name variable from the
    node, returns a string
520     return name; //Returns the information from the variable
    name
521 }
522
523 public Node getNext() { //Returning a node
524     return next; //Next because we are calling the pointer next
    , returns the node from the variable next
525 }
526
527 //I will now build two setters: setting the name, and setting
    the pointer
528 //Void because it isnt going to return anything
529 public void setName(String n) { //Parameters always go inside
    parenthesis, I am only updating the name value, so we only need
    one parameter
530     name = n;
531 }
532
533 public void setNode(Node m) { //"m" is the node that we are
    going to set next equal to, for the pointer
534     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.
535 }
536
537 //toString so that the program prints what is actually inside
    the node rather than the object identifier
538 public String toString() {
539     String result = name; //setting a string equal to what is
    inside the node
540     return result;
541 }
542 }
543 \end{lstlisting}
544
545 \newpage
546
547 \section{Main Program}
548     \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].\\
549 \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! \\
550 \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.\\
551 \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.
      \\
552
553 \begin{center}
554 \begin{tabular}{||c c c c||}
555 \hline
556 Selection sort & Insertion sort & Quick sort & Merge sort \\
557 \hline\hline
558 221444 & 108782 & 6401& 6105 \\
559 \hline
560 \end{tabular}
561 \end{center}
562
563 \begin{lstlisting}[language = Java]
564 import java.io.File;
565 import java.io.FileNotFoundException;
566 import java.util.Arrays;
567 import java.util.Scanner;
568 import java.util.*;
569
570 public class Main {
571     public static void main (String[] args) {
572         String[] magicItems = new String[666];
573         try { //Trying to find the file
574             File file = new File("magicitems.txt");
575             Scanner sc = new Scanner(file);

```



```

576         int index = 0;
577
578         while (sc.hasNextLine()) {
579             String item = sc.nextLine();
580             item = item.toUpperCase();
581             item = item.replaceAll("\\s+", "");
582             Node magicItemsNode = new Node(item, null);
583             magicItems[index++] = magicItemsNode.getName();
584         }
585     }
586     catch (FileNotFoundException e) { //If we cant find the file
587         e.printStackTrace();
588     }
589
590     Insertion insertSort = new Insertion(magicItems);
591     Selection selectionSort = new Selection(magicItems);
592     Quick quick = new Quick(magicItems);
593     Merge merge = new Merge(magicItems);
594
595     shuffle(magicItems); // Shuffling
596     selectionSort.selectionSort(magicItems); // Running the
algorithm
597     System.out.println(selectionSort); // Printing the
comparisons
598
599     shuffle(magicItems);
600     insertSort.insertionSort(magicItems);
601     System.out.println(insertSort);
602
603     shuffle(magicItems);
604     quick.quick(magicItems, 0, magicItems.length - 1);
605     System.out.println(quick);
606
607     shuffle(magicItems);
608     merge.mergeSort(magicItems, 0, magicItems.length - 1);
609     System.out.println(merge);
610 }
611
612 // Shuffle function
613 // Will be called prior to each sort being ran
614 public static void shuffle(String[] magicItems){
615     Random random = new Random();
616     for (int i = magicItems.length - 1; i > 0; i--) { //
Shuffling beginning from the last element
617         int randInt = random.nextInt(i+1); // Choosing a random
index for the shuffling
618         String temp = magicItems[i]; // Setting the temporary
variable to magicItems[i] to prepare for swap
619         magicItems[i] = magicItems[randInt]; // Swapping
620         magicItems[randInt] = temp;
621     }
622 }
623 }
624
625 /*
626 Code from when I was testing my sorts
627

```

```

628 Quick magicItemsQuick = new Quick(magicItems);
629 magicItemsQuick.quick(magicItems, 0, magicItems.length - 1);
630 System.out.println(magicItemsQuick);
631
632 Merge magicItemsMerge = new Merge(magicItems);
633 magicItemsMerge.mergeSort(magicItems, 0, magicItems.length - 1);
634 System.out.println(magicItemsMerge);
635
636 Selection magicItemsSelection = new Selection(magicItems);
637 magicItemsSelection.selection();
638 System.out.println(magicItemsSelection);
639
640 Insertion magicItemsInsertion = new Insertion(magicItems);
641 magicItemsInsertion.Insertion();
642 System.out.println(magicItemsInsertion);*/
643 \end{lstlisting}
644 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
645 % The commands below print the source code starting
646 % on a new page. Comment out or delete if you do not want to
647 % include the source code in your document.
648 %
649 \newpage
650 \section{\LaTeX \hspace{.25em} Source Code}
651 \lstset{basicstyle=\footnotesize\ttfamily, breaklines=true,
        language=[LaTeX]{TeX}}
652 \lstinputlisting{main.tex} % <---CHANGE TO CORRECT FILENAME
653 %
654 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
655
656 \end{document}

```