Lab 1: Data Structures

Nicholas Suchy
Nicholas.Suchy1@Marist.edu

March 10, 2023

1 Introduction

For this lab we were to implement 4 different sorting algorithms (selection sort, insertion sort, merge sort, and quicksort) in order to sort magicitems.txt. We were to print out the number of comparisons and the run time for each individual algorithm.

2 Code Reuse and Changes From Previous Labs

The Linked List and Node classes from Lab 1 were repurposed for this lab. The biggest change with them is that instead of using char values, the Node class now holds Strings and the Linked List was changed to accommodate. This changed allows for lines from magicitems.txt to be stored as Strings.

Additionally, when reading in the magicitems.txt file, the program now uses a relative file path rather than an absolute path. This allows the program to access magicitems.txt without having to travel the whole path from the hard drive. The file magicitems.txt was read into a Linked List named magicItems

3 Structure

When it came to organizing and structuring the sorting algorithms, I made the choice to implement each algorithm as a method in Main, rather than make them into separate classes.

The biggest factor for me deciding this was that it was easier for me to comprehend the assignment when the algorithms were presented as methods rather than classes. In my head, it made more sense to to call a sorting method rather than create a new instance of said algorithm.

Other ideas that pushed me to keep this structure style were that creating sorting classes might take up more memory. When all the sorting algorithms are stored in main, they can use the original Linked List created by reading lines from the magicitems.txt. If they were implemented as classes, each sorting algorithm may need a copy of the original Linked List which could lead to storing redundant data.

4 Keeping Track of Comparisons and Run Time

```
8     static int totalComparisons = 0; // # of comparisons an algorithm makes
9     static long begin; // Holds the time of when a function starts
10    static long end; // Holds the time of when a function finishes
11    static long totalTime = 0; // Holds the Total runtime of a sorting algorithm
```

The above variables were used in every algorithm to help keep track of time and the number of comparisons each algorithm made.

- The totalComparisons variable counts the number of comparisons each algorithm makes. It only keeps track of comparisons within the Linked List and not index-checking comparisons.
- The variable begin keeps track of the start time of the algorithm in nanoseconds. It is initialized at the start of every algorithm.
- The variable end keeps track of the end time of an algorithm in nanoseconds. It is initalized at the end of every algorithm.
- The totalTime variable is initalized after end. It holds the value of end minus the value of begin which represents the total time that the function took in order to run.

I choose this method of keeping track of time and comparisons because it allowed me overcome the problem of only being able to return one value from a function and returning values with recursion. As the variables are initialized out side of the functions, they can be used by all functions and all functions can manipulate them. Additionally, there aren't any adverse effects when using them with recursion. For example, a problem I was running into was that I could only record the run time of the last recursive call for Merge Sort and Quicksort because, when initialized within the functions, totalTime would continually be set to 0. However, by initalizing totalTime outside, I was able keep track of all of the recursive run times and continuously add them to totalTime until the last recursive call has completed.

5 Example Output

Sort	Comparisons	Run Time (μs)	Asymptotic Run Time
Selection Sort	221445	23752	quadratic
Insertion Sort	114415	533981	quadratic
Merge Sort	5420	7522	linearithmic
Quicksort	6271	81	linearithmic

The above table represents output from the program. Because of the shuffle before each algorithm, repeating the exact values would be very difficult. It is evident that the linearithmic algorithms run much faster and make much fewer comparisons compared to the quadratic algorithms.

Quadratic sorting algorithms are denoted by nested loops, meaning that in their worse cases, they check every item to in a list before they finish. However, linearithmic algorithms combine the aspects of linear algorithms with logrithmic algorithms. This means that they work by halving the amount of checks performed and then also use a loop. This results in significantly less checks the a quadratic algorithm resulting in less comparisons and a faster run time.

6 Swap

```
21    String nodeAValue = nodeA.value;
22    nodeA.value = nodeB.value;
```

```
23    nodeB.value = nodeAValue;
```

A major, but rather simple, component of each sorting algorithm is the swap method. When called, it gets passed two Nodes and swaps the values of those Nodes.

7 Knuth Shuffle

```
39   for (int i = unshuffledList.length() - 1; i > 0; i--) {
40       int randomIndex = (int) (Math.random() * (i));
41       swap(unshuffledList.get(i), unshuffledList.get(randomIndex));
42   }
```

As per the instructions of the assignment, we had to shuffle magicItems before every sorting algorithm was called. The above code demonstrates a Knuth shuffle performed in linear time. As we hadn't discussed this shuffle in-depth in class, I used the Fisher-Yates Shuffle Wikipedia page to help with my understanding and implementation of the code.

The shuffle generates a random index, and then swaps the values of the Node at the randomly generated index with the Node at i (which starts as the last index in the Linked List). i then decrements and the process repeats until the whole list has been shuffled.

8 Presentation

```
59    System.out.println("Number of Comparisons : " + comparisons);
60    System.out.println("This took : " + time / 1000 + " microseconds");
61    System.out.println();
62    totalComparisons = 0;
63    totalTime = 0;
```

A simple method that helps streamline and organize the printing of the needed data for the Lab. Additionally, since it is called after each sorting function, it resets the totalComparisons and the totalTime variables so that they can start fresh for the next sorting algorithm.

9 Selection Sort

```
77    for (int i = 0; i < unsortedList.length() - 1; i++) {
78
79        Node minNode = unsortedList.get(i); // Holds minimum Node while navigating list
80        Node tempNode = minNode.nextNode; // Holds comparison to minNode
81
82        for (int j = i + 1; j < unsortedList.length(); j++) {
83            // Check if tempNode comes before minNode alphabetically
84            if (tempNode.value.compareTo(minNode.value) < 0) {
85                minNode = tempNode;
86            }
87            tempNode = tempNode.nextNode;
88            totalComparisons++;
89        }
90
91        // Swap minNode with correct index Node
92        if (minNode != null && unsortedList.get(i) != minNode) {
93            swap(unsortedList.get(i), minNode);
```

```
94        }
95    }
```

The above code shows the main components of the Selection Sort algorithm, namely the time recording variables are left out to save space. The algorithm was built based off of the pseudo code we discussed in class. Notably the algorithm contains nested for loops signifying that it runs in quadratic time.

The method takes a Linked List as a parameter. Then it establishes the Node located at index i as minNode (the alphabetically first Node) and holds the next Node (at index i+1) as a tempNode. The function then iterates through the whole list comparing minNode and tempNode, with tempNode cycling through every item in the list. If at any point during the iteration tempNode comes alphabetically before minNode, tempNode becomes minNode. Once the end of the list has been reached minNode is placed at index i and then i is incremented and the process repeats again until the whole list is sorted.

10 Insertion Sort

```
112    for (int i = 0; i < unsortedList.length(); i++) {
113
114        int j = i;
115
116        while (j > 0 &&
           unsortedList.get(j - 1).value.compareTo(unsortedList.get(j).value) > 0) {
117
118            swap(unsortedList.get(j - 1), unsortedList.get(j));
119            j--;
120            totalComparisons++;
121        }
122    }
```

The above code shows the sorting aspect of the Insertion Sort algorithm. The algorithm was based off the pseudo code we discussed in class. Again, the algorithm contains nested looping signifying that it runs in quadratic time.

The method takes in a Linked List as a parameter which it is to sort. Then it begins iterating through the list. It grabs the item at index j and compares it previous to items in the list until a item that comes before it alphabetically or the start of the list is reached. It then increments i, sets j equal to i and repeats until the list is sorted.

11 Merge Sort

```
198    int length = unsortedList.length();
199        if (length < 2) {
200            return;
201        }
202        int middle = length / 2;
203
204        // Creates and populates a Linked List with values left of the middle
205        LinkedList left = new LinkedList();
206        for (int i = 0; i < middle; i++) {
207            left.addBack(unsortedList.get(i).value);
```

```
208         }
209
210         // Creates and populates a Linked List with values right of the middle
211         LinkedList right = new LinkedList();
212         for (int i = middle; i < unsortedList.length(); i++) {
213             right.addBack(unsortedList.get(i).value);
214         }
215
216         // Recursive statments
217         mergeSort(left);
218         mergeSort(right);
219         totalComparisons += merge(unsortedList, left, right); // Keeps track of comparisons
```

The above method shows the Merge Sort algorithm. This algorithm was also based off of the pseudo code shown and discussed in class. While not evident based off just this method, this algorithm runs in linearithimic time as the method halves the amount of time with each recursive call and merge() uses linear time to iterate through the entirety of sub-arrays.

This method takes a Linked List as a parameter and focuses on splitting it in half into two sub-arrays, Linked Lists in this case. Then it recursively calls itself to further split the halved Linked List, until all sub-arrays are of size one. Once all sub-arrays are of size 1 (meaning that they are technically sorted), the merge() method is called in order to sort and put the original Linked List back together.

11.1 merge()

```
154     while (i < leftLength && j < rightLength) {
155
156         if (left.get(i).value.compareTo(right.get(j).value) < 0) {
157             unsortedList.get(k).value = left.get(i).value;
158             i++;
159         } else {
160             unsortedList.get(k).value = right.get(j).value;
161             j++;
162         }
163         comparisons++;
164         k++;
165     }
166
167     // Both while loops check for any leftover items in left and right
168     // Since left and right are sorted, adds them correctly to unsortedList
169     while (i < leftLength) {
170         unsortedList.get(k).value = left.get(i).value;
171         i++;
172         k++;
173     }
174     while (j < rightLength) {
175         unsortedList.get(k).value = right.get(j).value;
176         j++;
177         k++;
178     }
179
```

```
180     return (comparisons);
```

The above code describes the merge() method within the Merge Sort algorithm. While mergeSort() focuses on splitting the Linked Lists in half, down to lengths of 1, merge() focuses on merging these sub-arrays back into one large array. It takes 3 Linked Lists as parameters, a sub-array of left values, a sub-array of right values, and a Linked List containing the contents of both of the other sub-arrays. The method checks the first value of each sub-array against each other, and then writes the alphabetically first one to the correct spot in the larger Linked List. The counter for the respective sub-array is then incremented and the next values are compared until one entire sub-array has been iterated through. Then since values in both sub-arrays are sorted within the sub-array, the remaining values in the opposing Linked List are filled into their correct spots in the larger Linked List, successfully creating a sorted Linked List comprised of the values of the two sub-arrays.

Additionally, it is worth noting that merge() returns an integer that represents the number of comparisons made while the method was running its course. This is returned and added to totalComparisons to keep track of the total number of comparisons that occur over the whole algorithm.

12 Quicksort

```
238     if (lowerIndex >= upperIndex) {
239         return;
240     }
241
242     // Selects a random index between lowerIndex and upperIndex to become pivot
243     // Places pivot at the end of the list
244     int randomIndex = (int) (Math.random() * (upperIndex - lowerIndex + 1) + lowerIndex);
245     swap(unsortedList.get(randomIndex), unsortedList.get(upperIndex));
246     String pivot = unsortedList.get(upperIndex).value;
247
248     int leftPointer = lowerIndex;
249     int rightPointer = upperIndex;
250
251     while (leftPointer < rightPointer) {
252
253         // Checks if value at leftPointer comes before pivot alphabetically
254         while (unsortedList.get(leftPointer).value.compareTo(pivot) <= 0 &&
          leftPointer < rightPointer) {
255             leftPointer++;
256             totalComparisons++;
257         }
258
259         // Checks if value at rightPointer comes after pivot alphabetically
260         while (unsortedList.get(rightPointer).value.compareTo(pivot) >= 0 &&
          leftPointer < rightPointer) {
261             rightPointer--;
262             totalComparisons++;
263         }
264
265         // Swaps the pointer values so they are on the right "side" of the pivot
266         swap(unsortedList.get(leftPointer), unsortedList.get(rightPointer));
267     }
268
```

```
269     // Put the pivot into the correct spot in the list
270     swap(unsortedList.get(leftPointer), unsortedList.get(upperIndex));
271
272     // Recursive Statements
273     quickSort(unsortedList, lowerIndex, leftPointer - 1); // Everything before pivot
274     quickSort(unsortedList, leftPointer + 1, upperIndex); // Everything after pivot
```

The above code represents the Quicksort algorithm used in my program. It takes a Linked List and two integers representing the upper and lower bounds of a sub-section of the Linked List as parameters. The algorithm functions similarly to merge sort in partitioning the list but also sorts the list as it divides down. The algorithm generates a random index and uses the value at that index to be the pivot value for the current instance of the method. Then starting from the lower index passed to the function, it checks to see if the value of a Node comes alphabetically before the pivot value. If so, it moves onto the next Node. Once it fails however, it begins from the upper index passed to the function and checks to see if the value of the Node if equals of comes alphabetically after the pivot value. Again, if the check is true it moves onto the next Node (going towards the lower index). If the check fails, the the two values at both pointers are swapped and the process repeats until the pointers are on the same Node. Then the pivot value is swapped with the value that both pointers are sharing, and the function recursively calls itself to continue partitioning and sorting.

The method is strongly modeled after the pseudo code we looked at and discussed in class with a few changes. Firstly, instead of always using the last index, the Quicksort algorithm here selects a random index and then assigns it to the last index. On average, this allows the algorithm to perform better. Additionally, one optimization we had talked about that I did not include was skipping the right pointer checking the pivot after it has been moved to the back of the list. While this removes an unneeded comparison from the algorithm, the method has the chance to incorrectly sort a sub-section of size two as both pointers would immediately point to the value that isn't the pivot, skipping a much needed comparison.

While harder to see in this one, Quicksort is also a linearithmic sorting algorithm. While it doesn't always cut the list exactly in half, on average the partitioning steps reduces the size of the array by half. Combined with the linear time property of iterating through the list to sort based on the pivot, Quicksort remains a linearithmic sorting algorithm.