**Code Analysis**

Each of the sorting algorithms used are standard Bubble/Insertion/Selection/Merge sort algorithms that are in the Canvas exercises. Implementing the arrays was nearly identical to the Canvas examples however I had to tweak a few things to make the Linked List implementations work. This includes things like slightly adjusting the start/end indexes in FOR loops and creating helper methods in my List class for quicker comparing and returning results.

To add items to my Linked List I used a standard method of taking in a desired index.

Insert(data, index)

Node current = head.Next

FOR(i = 0; i < index; i++)

Current = current.next

Current.next = new Node(current.data, current.next)

Current.data = data

IF(current is at tail) current.next = tail

nodeCount++

This method scrolls from left to right through the list until it reaches the new index. Then it sets the current node’s .Next property to a clone of the current node. Finally it sets the current node’s data to the input data. After making sure it isn’t a new tail, it increments a counter used to keep track of the total length of the linked list. This method requires O(n) time to get to the requested index. After that, it takes O(1) time to insert the item.

I used the standard LinkedList Delete method and tweaked it just like I did for Insert in order to take in an index and delete at the given index. Both methods are derived from the implementation given in Canvas. This method, like insert, takes a total of O(n) time.

I also needed to create a method for my Linked List to swap two item positions in the list.

Swap(index 1, index 2)

Node newPointer = new Node with .Next of NULL

Move currentNode pointer to index1

Move newPointer pointer to index2

Char temp = newPointer.Data

newPointer.Data = currentNode.Data

currentNode.Data = temp

As you can see, this takes a time complexity of O(n) to get to each of the indexes. After that, it is just O(1) to swap the data. This method is used in most of the sorting algorithms and it caused them to have a high average time complexity.

The last important method that I had to create was a replace method for changing the value at a given index.

Replace(data, index)

Move currentPointer to index

currentPointer.Data = data

Again, this takes O(n) total time to run.

In each of these methods, I have to move to a given index. To make each method less confusing, I offloaded this task to a method that takes in an index and moves the pointer to the requested index. The time complexity of this operation is already calculated in each of the descriptions above.

I use two other methods outside of the List class that perform a notably large amount of compares. The first is CharValues which takes the user input name and changes the sorting order so that it fits the instructions for this program.

CharValues()

CREATE string “tempAscii” containing alphabet in order

CREATE char array “chars” of length 26

CREATE char array “nameArr” the same length as the number of characters in input name

FOR(int i < nameArr.Count; i++)

Chars[i] = nameArr[i]

REMOVE char nameArr[i] from tempAscii string

FOR(int i=nameArr.Count; i<26; i++)

Fill chars with remaining characters in tempAscii in order

RETURN chars as string

The last method used is CharCompare which takes in two chars and compares them to the string generated by CharValues to see which is larger with our new sort parameters. It does this by checking if the index value of the character in the generated sorting string is greater, less than, or equal to the index of the second input character.

CharCompare(x, y)

IF(x < y) return -1

ELSE IF(x > y) return 1

ELSE return 0

This method operates at O(n) since it has to search the sorting string for the index of the input.

**Graphical Analysis**

Below are my observed time complexities based on an average of 100 to 1000 runs of each algorithm (dependent on number of elements to sort). Because of this, all time complexities are shown in their average case.

Bubble

Theoretical: Ө (n^2) Empirical: Ө (n^3)

Insertion

Theoretical: Ө (n^2) Empirical: Ө (n^2)

Selection

Theoretical: Ө (n^2) Empirical: Ө (n^3)

Merge

Theoretical: Ө (n log(n)) Empirical: Ө (n^2)

The empirical time complexities for each of the sorts above are almost all effectively one power higher than the theoretical due to the time complexity added by performing operations on the Linked List. I don’t like using a Linked List in this case since you can only move left and right in it to get to a specified index. I don’t think there is any good reason to use a Linked List for these operations and the memory saved by not creating axillary arrays is not worth the added runtime.

Bubble

Theoretical: Ө (n^2) Empirical: Ө (n^2)

Insertion

Theoretical: Ө (n^2) Empirical: Ө (n^2)

Selection

Theoretical: Ө (n^2) Empirical: Ө (n^2)

Merge

Theoretical: Ө(n log(n)) Empirical: Ө (n log(n))

My implementation of these algorithms for sorting arrays used the standard sort algorithms so the average case time complexity matched up to the theoretical complexity well. The only difference are a few constants here and there but those don’t matter. These algorithms were much easier to implement since they didn’t require any list traversals to point to an index in the array. Because of this, I avoided tacking on an extra n in each average case.