

CS3310

Third Assignment (a2)

Report

Brandon Rodriguez

10-27-17

# Preface

First off, a working, compiled jar of the assignment can be found in the root directory, called “a2.jar”. To run this, open up a terminal, navigate to the directory, and type “java -jar a2.jar”. The rest should be self-explanatory.

To find the source code of the assignment, navigate from the assignment’s root folder to “root/a2/src/edu/wmich/cs3310/a2”. A majority of the code is located within the controller.java class and the “DataStructures” subfolder.

Used libraries are included in the “root/a2/lib/” folder.

JavaDocs can be found at “root/Documents/JavaDocs/index.html”, as well as within the project dist folder.

Runtime output logs can be located at “root/Documents/RuntimeOutput/”.

Program was created using JetBrains’ “IntelliJ” IDE.

If not familiar with IntelliJ:

Should the jar file not execute, you should be able to load it into IntelliJ and get the same results. When first opening the project, use “Import Project” through IntelliJ’s splash screen. Then select “Create project from existing sources”.

If IntelliJ doesn’t automatically do it for you, make sure to set the “src” directory as the “Sources Root”, and the “tests” directory as the “Test Sources Root”. You may also need to load in junit, which should be provided within the lib folder.

You may also need to “Edit Configurations” and set Main as an application to launch from. But after that, it should work. Just run the program with the green arrow button and it should launch.

# Problem Statement

This assignment focus on the comparison of various basic sorts against eachother, both in linked list and array form. Given sorts are “Bubble”, “Selection”, “Merge”, and “Binary Insertion”. For both completeness and ease of programming the Binary Insertion, I have done a standard Insertion sort too.

---

## Program Description

Due to no restrictions on linked list implementation, this uses a doubly linked list with a node that can store both chars and ints. The chars are always used and the ints are only used if the user opts to show stability of sorting.

At startup, all user prompts are displayed, including numbner of random characters to generate and given user “name” to use for sorting.

After user input is given, the randomly generated character is created and then used as-is for all sort types.

In regards to sorting itself, it is done alphabetically with one modification- The given user’s ‘name’ input is parsed, and all characters found within get priority over the standard alphabet ordering. Any duplicate characters within the name are ignored.

Once all sorting methods complete, the given output and time information is displayed.

# Algorithm Descriptions

## Bubble Sort

Standard bubble sort, modified to have a boolean that keeps track of if it has been sorted during the previous loop. If the boolean stays false through a full loop, then all remaining values are in the proper order and the sort can end. Otherwise, it acts as a standard bubblesort and compares all remaining values.

This allows the loop to have a theoretical best case of  $O(n)$  if sorting data that's already properly ordered. It will travel through all  $n$  values, the boolean will stay false, and then it will exit.

```
Initialize sortedBool to true
Initialize maxIndex to number of characters to sort
Initialize index to 0
while sortedBool is true
    set sortedBool to false
    while index is less than maxIndex
        if index and (index + 1) are out of order
            swap values
            set sortedBool to true
        endif
        increment index
    set index back to 0
    decrement maxIndex
endwhile
```

For linked lists, it simply help a “maxNode” instead maxIndex and compared current node to current.next.

---

## Selection Sort

Standard selection sort. Loop through all unsorted values and find highest one. Then set the last “unsorted” index to the found value. Repeat, slowly filling the end of data structure with sorted values until all are found.

```
Initialize selectedIndex to 0
Initialize lastValueSorted to (characters - 1)
while lastValueSorted is greater than 0
    loop from 0th index to lastValueSorted index
        compare currentIndex to selectedIndex
        if currentIndex has higher value
```

```

        set selectedIndex to currentIndex
    endif
endloop
set lastValueSorted to value at selectedIndex
set selectedIndex back to 0
decrement lastValueSorted
endwhile

```

For linked lists, it simply held “highestNode” instead of a “lastValueSorted” index, and then compared current node to a “selectedNode”.

---

## Merge Sort

Standard merge sort. Recursively divide data until down to base case of one item. Then merge back together, in order.

recursiveDivide:

```

    If low index is same as high index
        Back out of recursion
    endif

    Call recursiveDivide on left half
    Call recursiveDivide on right half
    Call mergeHalves

```

mergeHalves:

```

    Duplicate values of array into secondary array

    loop from lowerIndex until upperIndex
        if leftHolder > middleIndex
            set origArray[index] to secondArray[leftHolder]
            increment leftHolder
        else if rightHolder > upperIndex
            set origArray[index] to secondArray[rightHolder]
            increment rightHolder
        else if secondArray[leftHolder] is smaller than secondArray[rightHolder]
            set origArray[index] to secondArray[leftHolder]
            increment leftHolder
        else
            set origArray[index] to secondArray[rightHolder]
            increment rightHolder
        endif
    endloop

```

endloop

For linked lists, had to hold the appropriate “index” values separate from the list. Then, when the index of a given node needed to be used, would have to step through the list equal to the index number.

In all honesty, I’m surprised this had as good of time as it did, given how it would have to frequently step through. It could also probably be improved by having a “total list size” tracker. Then, if the desired index was greater than half the list size, it would step through starting from end of list, not start. Obviously, the amount to step would need to be adjusted as well.

---

## Insertion Sort

Mostly-standard insertion sort. All values at start of data are sorted. For each loop, grab the “next” value and compare. Progressively bump this new value towards the start of list until it’s in the proper location.

```
loop from 0 to last index
  save index as maxIndex
  while index is greater than (index + 1)
    swap index and index + 1 value
    decrement index
  endwhile
  set index to back to maxIndex
endloop
```

The linked list was handled essentially the same. Just instead of indexes, it used nodes.

---

## Binary Insertion Sort

This one essentially was the same as above until it found an unsorted value ( $\text{index} < \text{index} + 1$ ). Then it would turn into a standard binary search to place the value at the correct spot. As with merge sort, the linked list implementation had to keep track of indexes outside the list, and then step through to find the appropriate node location.

For psuedocode, see above and also reference any generic binary search psudeo code.

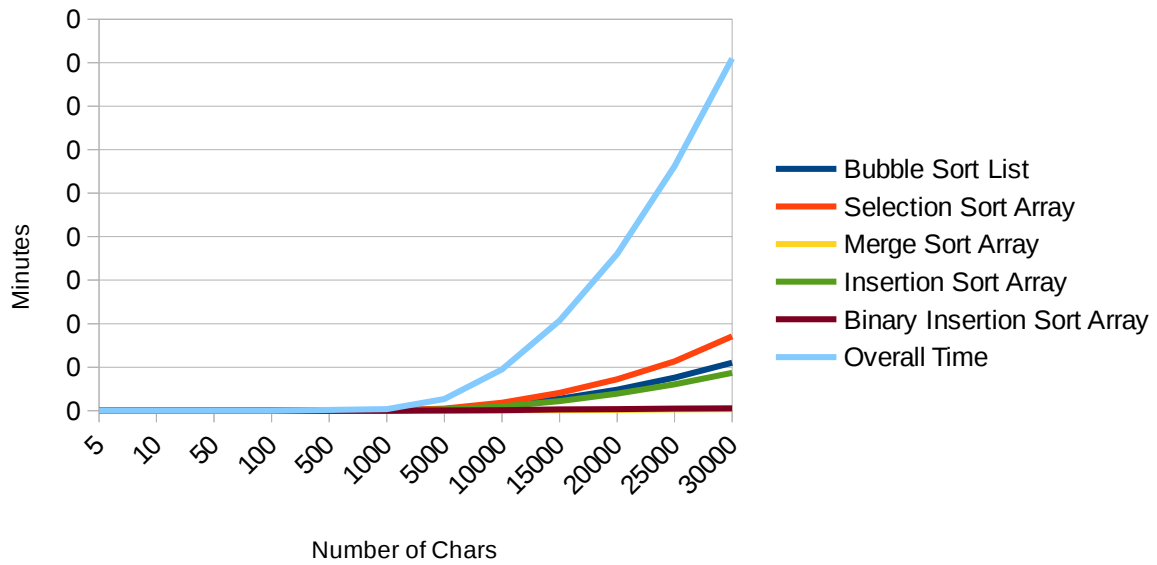
# Tables of Observed Time Complexities

Note: Format is of ( Minute : Second : Millisecond )

If no minute values are present, then minute column is omitted.

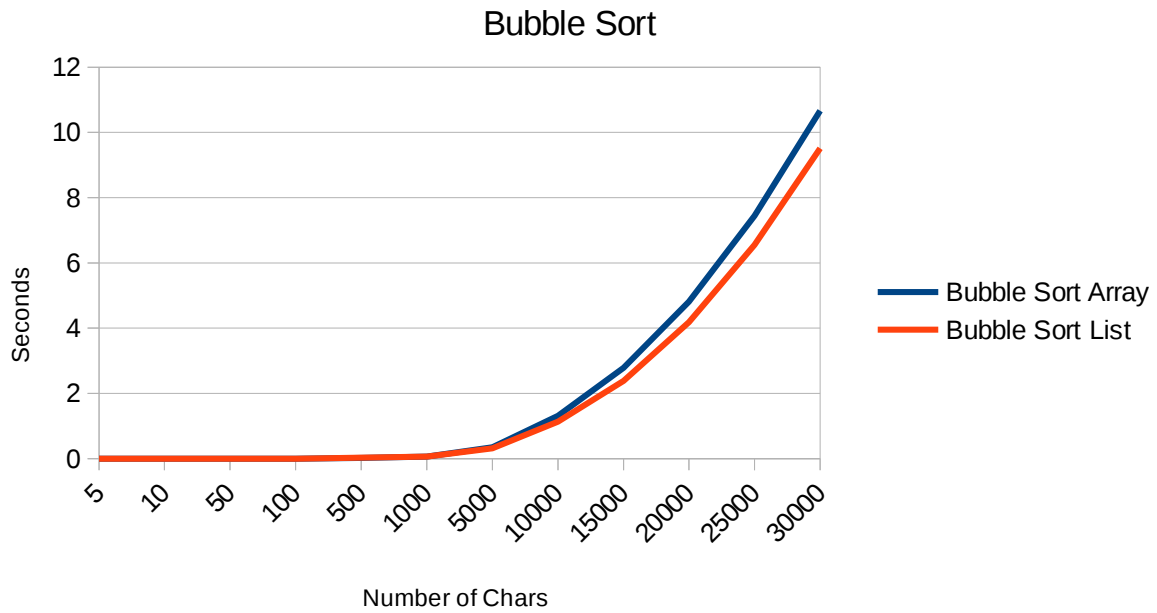
Char #	Bubble Sort Array	Bubble Sort List	Selection Sort Array	Selection Sort List	Merge Sort Array	Merge Sort List	Insertion Sort Array	Insertion Sort List	Binary Insertion Sort Array	Binary Insertion Sort List
5	00:000	00:000	00:000	00:000	00:000	00:000	00:000	00:000	00:000	00:000
10	00:000	00:001	00:000	00:000	00:000	00:000	00:000	00:001	00:000	00:000
50	00:002	00:001	00:001	00:000	00:001	00:001	00:000	00:001	00:000	00:001
100	00:004	00:002	00:002	00:003	00:000	00:002	00:001	00:002	00:001	00:002
500	00:025	00:031	00:018	00:021	00:002	00:008	00:011	00:036	00:002	00:005
1000	00:064	00:064	00:031	00:049	00:002	00:007	00:018	00:044	00:003	00:009
5000	00:350	00:315	00:412	00:507	00:014	00:057	00:243	00:289	00:034	00:133
10000	01:313	01:134	01:564	01:746	00:054	00:275	00:841	00:866	00:126	00:300
15000	02:785	02:388	03:551	03:881	00:102	00:577	01:887	01:876	00:281	00:570
20000	04:810	04:186	06:268	06:923	00:160	00:873	03:365	03:349	00:342	00:881
25000	07:446	06:555	09:803	10:863	00:256	01:383	05:245	05:218	00:407	01:380
30000	10:667	09:512	14:770	14:693	00:312	02:108	07:526	07:562	00:456	02:210

Best of Each Sort Vs Overall

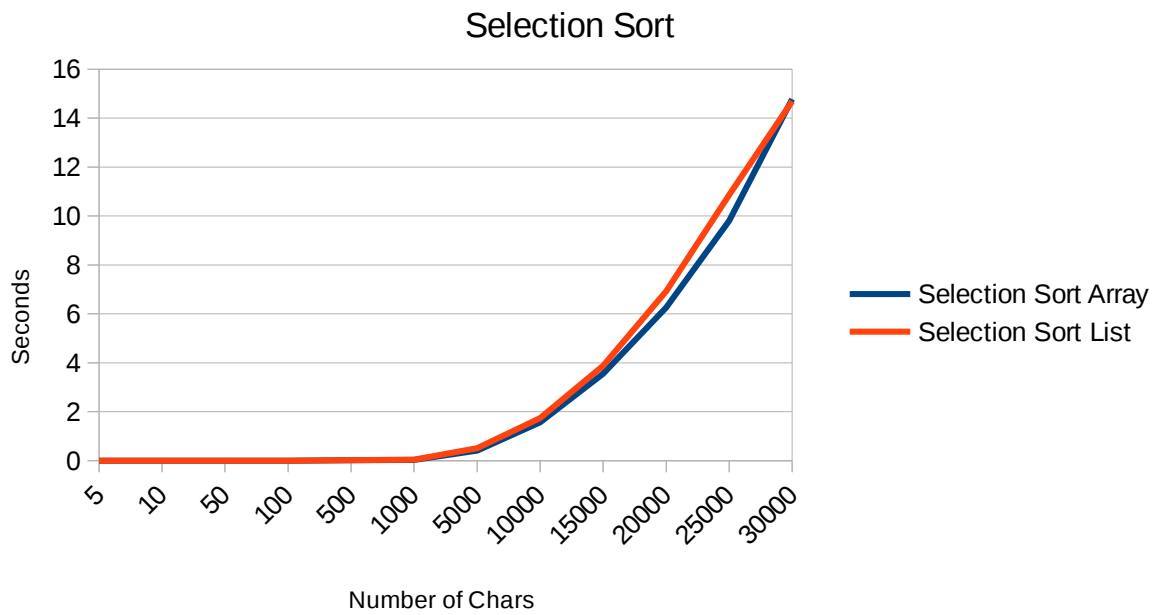


Char #	Bubble Sort List	Selection Sort Array	Merge Sort Array	Insertion Sort Array	Binary Insertion Sort Array	Overall Time
5	00:00.000	00:00.000	00:00.000	00:00.000	00:00.000	00:00.000
10	00:00.001	00:00.000	00:00.000	00:00.000	00:00.000	00:00.002
50	00:00.001	00:00.001	00:00.001	00:00.000	00:00.000	00:00.008
100	00:00.002	00:00.002	00:00.000	00:00.001	00:00.001	00:00.018
500	00:00.031	00:00.018	00:00.002	00:00.011	00:00.002	00:00.159
1000	00:00.064	00:00.031	00:00.002	00:00.018	00:00.003	00:00.291
5000	00:00.315	00:00.412	00:00.014	00:00.243	00:00.034	00:02.354
10000	00:01.134	00:01.564	00:00.054	00:00.841	00:00.126	00:08.219
15000	00:02.388	00:03.551	00:00.102	00:01.887	00:00.281	00:17.898
20000	00:04.186	00:06.268	00:00.160	00:03.365	00:00.342	00:31.157
25000	00:06.555	00:09.803	00:00.256	00:05.245	00:00.407	00:48.556
30000	00:09.512	00:14.770	00:00.312	00:07.526	00:00.456	01:09.934

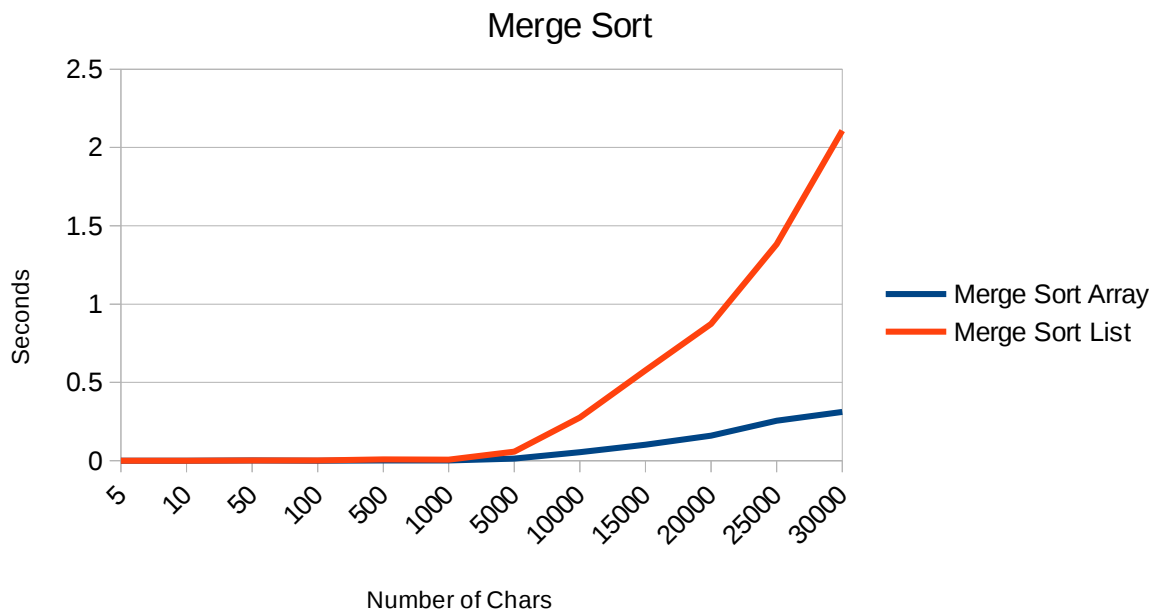




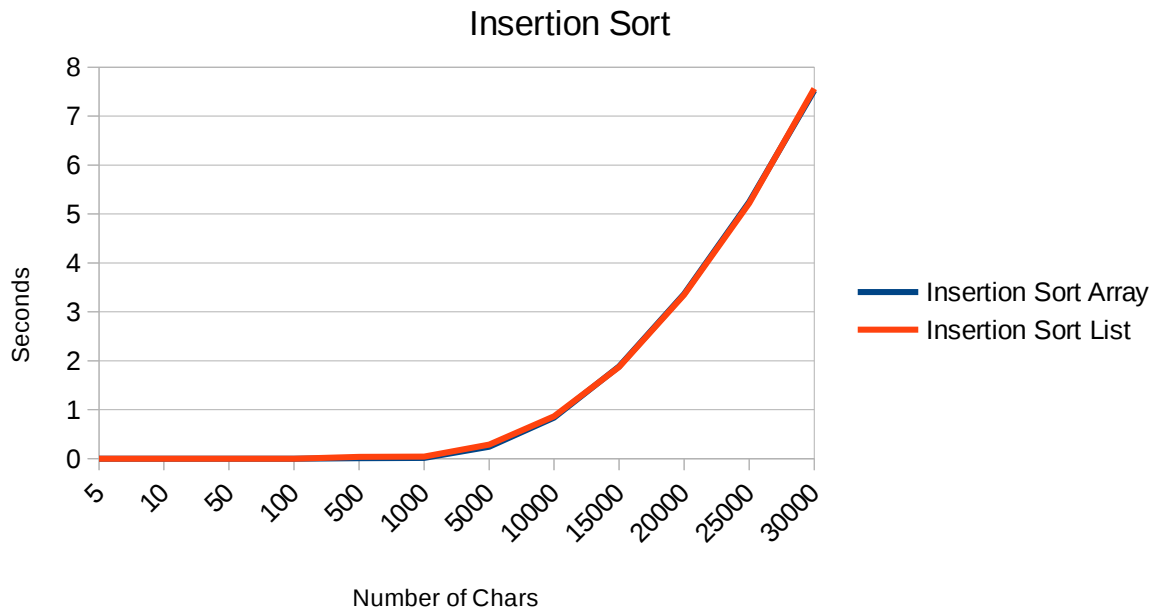
Char #	Bubble Sort Array	Bubble Sort List
5	00.000	00.000
10	00.000	00.001
50	00.002	00.001
100	00.004	00.002
500	00.025	00.031
1000	00.064	00.064
5000	00.350	00.315
10000	01.313	01.134
15000	02.785	02.388
20000	04.810	04.186
25000	07.446	06.555
30000	10.667	09.512



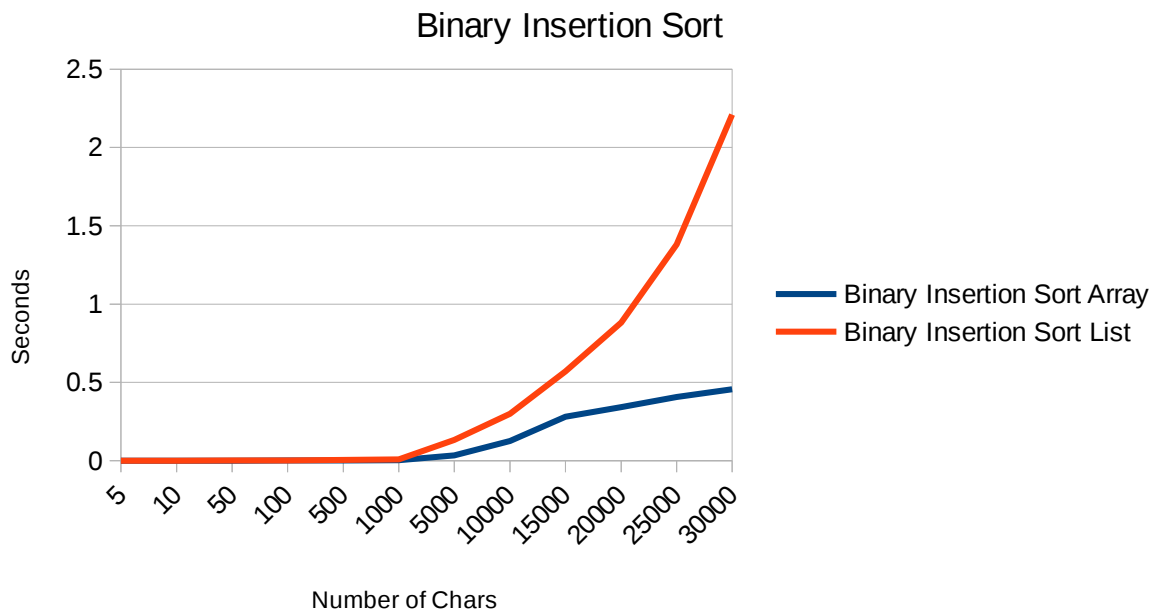
Char #	Selection Sort Array	Selection Sort List
5	00.000	00.000
10	00.000	00.000
50	00.001	00.000
100	00.002	00.003
500	00.018	00.021
1000	00.031	00.049
5000	00.412	00.507
10000	01.564	01.746
15000	03.551	03.881
20000	06.268	06.923
25000	09.803	10.863
30000	14.770	14.693



Char #	Merge Sort Array	Merge Sort List
5	00.000	00.000
10	00.000	00.000
50	00.001	00.001
100	00.000	00.002
500	00.002	00.008
1000	00.002	00.007
5000	00.014	00.057
10000	00.054	00.275
15000	00.102	00.577
20000	00.160	00.873
25000	00.256	01.383
30000	00.312	02.108



Char #	Insertion Sort Array	Insertion Sort List
5	00.000	00.000
10	00.000	00.001
50	00.000	00.001
100	00.001	00.002
500	00.011	00.036
1000	00.018	00.044
5000	00.243	00.289
10000	00.841	00.866
15000	01.887	01.876
20000	03.365	03.349
25000	05.245	05.218
30000	07.526	07.562



Char #	Binary Insertion Sort Array	Binary Insertion Sort List
5	00.000	00.000
10	00.000	00.000
50	00.000	00.001
100	00.001	00.002
500	00.002	00.005
1000	00.003	00.009
5000	00.034	00.133
10000	00.126	00.300
15000	00.281	00.570
20000	00.342	00.881
25000	00.407	01.380
30000	00.456	02.210

# Concluding Analysis

For the most part, the data matches what I expected.

---

## Time Complexities

Due to my modification, bubble sort was not the worst sort, and actually should have a best-case time of  $O(n)$ . As such, selection sort ended up being the worst sort overall. Insertion sort ended up being just a tad better than bubble sort.

Due to my modification of bubble sort, I feel like it and insertion should be the same time, as one just pushes elements forward and the other pushes backward. But they both should take  $O(n)$  time on a fully sorted list of elements and  $O(n^2)$  on a list of reverse-order elements. I suppose it has a small enough margin of error that the difference may be due to outside factors.

As predicted, merge sort was the best out of all of them, at a constant time of  $O(n \log n)$ .

Surprisingly though, binary insertion sort was a close second to merge. I thought that adding a binary search to insertion sort wouldn't change the time complexity. For arrays, the binary search would stop it from having to iterate through  $n$  values to find the correct spot, but then it would still have to move all elements over to free up the desired location, adding back a  $O(n)$  time.

For lists, the binary search would still have to repeatedly step through the list in order to actually find the node location, which again, should result in  $O(n)$  time being added back. So in all honesty, I'm not sure how the binary insertion sort was actually an improvement over the standard insertion sort, but it definitely shows an improvement none the less.

For the two faster sorts (merge and binary insertion), linked list was noticeably slower, which makes sense due to linked lists being unable to take advantage of indexes. For all other sorts, the linked list was either similar or very slightly faster than the array. This is due to not having to iterate through the list for a given node value and simultaneously being able to take advantage of the  $O(1)$  time to insert or delete an already found node.

---

## Space Complexities

All used sorts should have time of  $O(n)$  except for merge sort, as they all, at most, add a single temporary variable to hold a value while doing swaps.

Merge sort is the exception. For arrays, it duplicates the given array on merge, ultimately resulting in a max of  $O(2n)$  space. This obviously simplifies to  $O(n)$ .

For linked lists, it copies each half of the given list, as this was the easiest way to handle splitting the list without having to use even more index pointers. When possible, I put priority on avoiding index pointers, even if it was at the cost of (mostly negligible) additional space.

---

## Summary

Overall, this program really doesn't do a whole lot, other than check that we can implement various sorts, and then give data on how well they performed. Obviously, expanding into larger amounts of characters will give increasingly accurate representation of how well each sort performs.

However, even with the current data, I think it's safe to say that bubble, selection, and insertion sorts tended closer to  $O(n^2)$  time. Meanwhile, binary insertion and merge sort both appeared to be near  $O(n \log n)$  time.