Assignment 2 Report

Chang Yan (Charlotte)

April 30 2019

1 Extra Credit

In `arrayListUnsorted` class, I implement linear search called `myLinearSearch(E key)` and binary search search called `mybinarySearch(E key)` for the array list, where `key` represents the value to be searched. If we find this value, these two search methods returns the index of the value in the array list. Otherwise, we return -1. I have two runner classes named `runnerSearchingMethodArrayList` and `runnerSearchingMethodLinkedList` to test the searching time for linear search and binary search for array list and linked list. I cover the best case, average case and worst case in these two runner classes. The best case searches the first element in the list. The average case searches the middle element in the list, and the worst case is when we have to go through every element in the list. Figure 1 shows the worst searching time for linear search and binary search for array list and linked list. One thing we can observe is that binary search is slower than linear search in linked list. This is because I implement a while loop to find the middle node in binary search, which I should find a better way to do this. Overall, we can see that binary search is faster than linear search. This is because binary search keeps comparing the target value to the middle element of the array, which means that the algorithm of binary search takes logarithmic time $O(\log(n))$, whereas the running for linear search is $O(n)$.

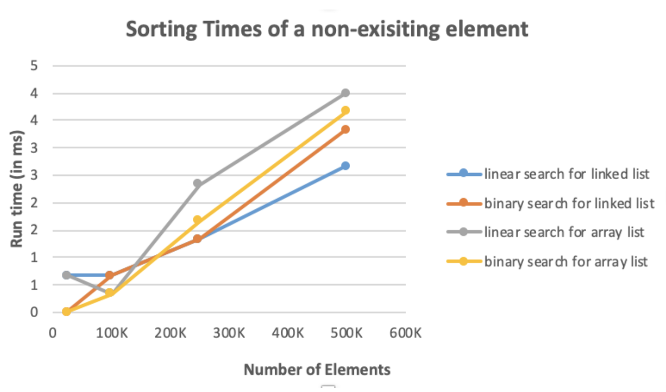
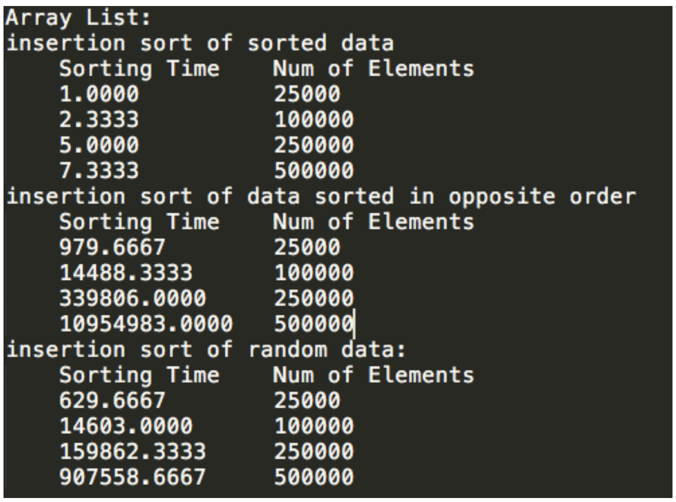


Figure 1: Linear and Binary Search for Array List and Linked List (worst case scenario)

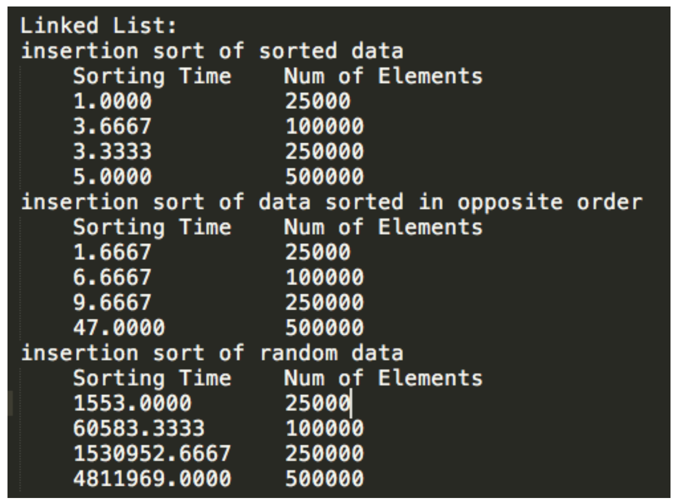
2 Sorting Algorithms

2.1 Insertion Sort.

Insertion sort is a sorting algorithm that builds a final sorted list one element at a time. Figure 2 shows the average sorting time for the array list and linked list. We can see that it only takes a few milliseconds to sort a sorted data for both array list and linked list. The best case is when we have our input as an array that is already sorted. In this case insertion sort has a linear runtime $O(n)$. During each iteration, the first remaining element of the input is only compared with the right-most element of the sorted sub-array. The worst case is when we have our input as an array sorted in opposite order. This is the worst case because every iteration of the inner loop has to scan and shift the entire sorted sub-array before inserting the next element. This gives insertion sort a quadratic runtime $O(n^2)$. This explains the reason why the sorting time for the array list in opposite order takes the longest time. However, I notice that the sorting time for the linked list in opposite order is pretty fast. I think this is because when I am inserting a node for a linked list, I simply break the old link and re-connect the new link in order for the node to be sorted. Unlike in array list, we have to shift the entire sorted sub-array before inserting an element.


```
Array List:
insertion sort of sorted data
    Sorting Time    Num of Elements
    1.0000          25000
    2.3333          100000
    5.0000          250000
    7.3333          500000
insertion sort of data sorted in opposite order
    Sorting Time    Num of Elements
    979.6667        25000
    14488.3333      100000
    339806.0000     250000
    10954983.0000   500000
insertion sort of random data:
    Sorting Time    Num of Elements
    629.6667        25000
    14603.0000      100000
    159862.3333     250000
    907558.6667     500000
```

(a) Insertion Sort for Array List.


```
Linked List:
insertion sort of sorted data
    Sorting Time    Num of Elements
    1.0000          25000
    3.6667          100000
    3.3333          250000
    5.0000          500000
insertion sort of data sorted in opposite order
    Sorting Time    Num of Elements
    1.6667          25000
    6.6667          100000
    9.6667          250000
    47.0000         500000
insertion sort of random data
    Sorting Time    Num of Elements
    1553.0000       25000
    60583.3333      100000
    1530952.6667    250000
    4811969.0000    500000
```

(b) Insertion Sort for Linked List.

Figure 2: Sorting Time for Insertion Sort.

2.2 Odd-even Sort.

Odd-even sort is a comparison sort related to bubble sort. In the algorithm for the odd-even sort, we begin by comparing all odd or even indexed pairs of adjacent elements in the list. If a pair is in the wrong order, we use bubble sort to switch these two elements. This process repeats for all even or odd indexed pairs until the list is sorted. Similar to insertion sort, the best case is the list is already sorted. In this case odd-even sort has a linear runtime $O(n)$. The worst case is when the list is opposite order, which gives us a quadratic runtime $O(n^2)$.

```
Array List:
odd-even sort of sorted data
    Sorting Time    Num of Elements
    0.0000          25000
    1.3333          100000
    1.3333          250000
    1.6667          500000
odd-even sort of data sorted in opposite order
    Sorting Time    Num of Elements
    1597.0000       25000
    27580.3333      100000
    546928.3333     250000
    2207111.6667    500000
odd-even sort of random data
    Sorting Time    Num of Elements
    2462.0000       25000
    45096.3333      100000
    493510.6667     250000
    2591524.3333    500000
```

(a) Odd-even Sort for Array List.

```
Linked List:
odd-even sort of sorted data
    Sorting Time    Num of Elements
    0.3333          25000
    2.0000          100000
    2.0000          250000
    11.6667         500000
odd-even sort of data sorted in opposite order
    Sorting Time    Num of Elements
    1852.3333       25000
    32651.6667      100000
    680360.3333     250000
    1306556.0000    500000
odd-even sort of random data
    Sorting Time    Num of Elements
    2988.0000       25000
    64657.6667      100000
    601238.3333     250000
    3403148.3333    500000
```

(b) Odd-even Sort for Linked List.

Figure 3: Sorting Time for Odd-even Sort.

2.3 Counting Sort.

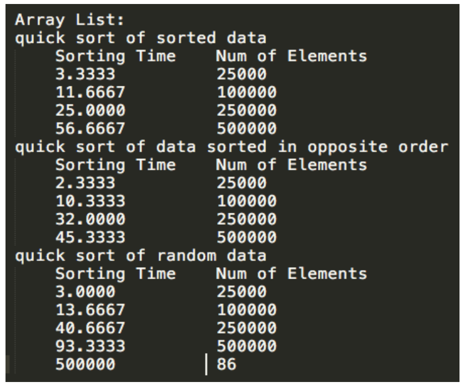
Counting sort us an algorithm for sorting a list of objects according to keys that are integers, that is, it is an integer count-based sorting. Figure 4 shows the runtime for counting sort. We notice that the sorting time for counting sort is shorter comparing to the sorting time for insertion sort and odd-even sort. The runtime for counting sort varies due to outliers in the original array. First, we need m for initializing the count array. Then, we need n for going over the original array to put numbers into count array. Finally we need n to put numbers from count into the original array. Hence, we the runtime for counting sort is $O(m + n)$.

```
Array List:
count sort of sorted data
    Sorting Time    Num of Elements
    1.0000          25000
    2.3333          100000
    11.3333         250000
    14.6667         500000
count sort of data sorted in opposite order
    Sorting Time    Num of Elements
    0.3333          25000
    1.6667          100000
    3.3333          250000
    6.0000          500000
count sort of random data
    Sorting Time    Num of Elements
    0.6667          25000
    1.3333          100000
    4.0000          250000
    7.6667          500000
```

Figure 4: Counting Sort for Array List.

2.4 Quick Sort.

Quicksort is a comparison sort. Suppose we pick the middle element as our pivot. Quicksort algorithm starts by sorting low, middle, high elements. Then we set up helper markers i and j to keep track of positions of elements that are less than or greater than the pivot. We repeat this process until i crosses j. It is important that we pick a good pivot. A good pivot choice separates the whole list into equivalent number of elements of partitions, which gives us a runtime $O(n \log(n))$. A bad pivot choice gives us a runtime $O(n^2)$.

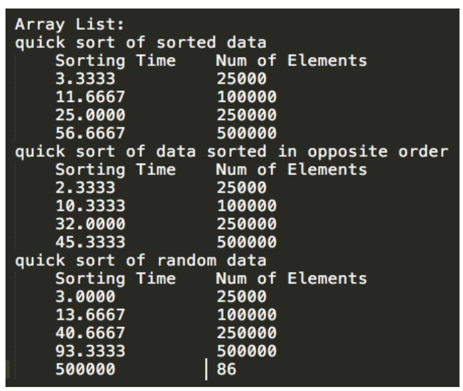


```
Array List:
quick sort of sorted data
    Sorting Time    Num of Elements
    3.3333          25000
    11.6667         100000
    25.0000         250000
    56.6667         500000
quick sort of data sorted in opposite order
    Sorting Time    Num of Elements
    2.3333          25000
    10.3333         100000
    32.0000         250000
    45.3333         500000
quick sort of random data
    Sorting Time    Num of Elements
    3.0000          25000
    13.6667         100000
    40.6667         250000
    93.3333         500000
    500000         | 86
```

Figure 5: Quick Sort for Array List.

2.5 Merge Sort.

A merge sort algorithm first divides the unsorted list into n sublists, each containing one element (a list of one element is considered sorted). Then the algorithm repeatedly merges sublists to produce new sorted sublists until there is only one sublist remaining. This will be the sorted list we desire. Therefore, the runtime for merge sort is $O(n\log(n))$.

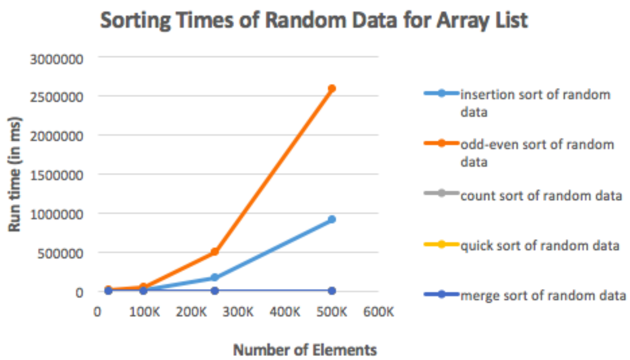


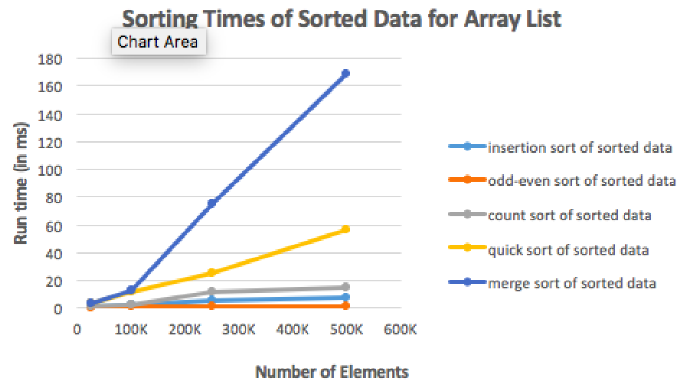
```
Array List:
quick sort of sorted data
    Sorting Time    Num of Elements
    3.3333          25000
    11.6667         100000
    25.0000         250000
    56.6667         500000
quick sort of data sorted in opposite order
    Sorting Time    Num of Elements
    2.3333          25000
    10.3333         100000
    32.0000         250000
    45.3333         500000
quick sort of random data
    Sorting Time    Num of Elements
    3.0000          25000
    13.6667         100000
    40.6667         250000
    93.3333         500000
    500000         | 86
```

Figure 5: Merge Sort for Array List.

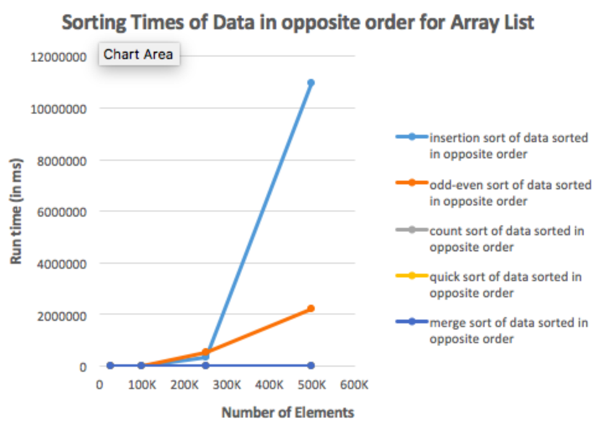
2.6 Comparison of Sorting Times and Conclusion

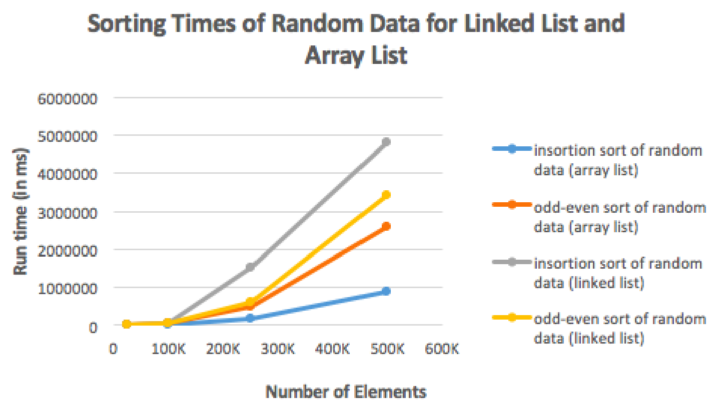
Table 1 shows the average runtime for insertion sort, odd-even sort, counting sort, quick sort and merge sort. From figure (a), we can see that insertion sort and odd-even are very time-consuming to sort a list of random data due to runtime of $O(n^2)$. However, in figrue (b), merge sort takes the longest time sort a sorted list of integers, whereas insertion sort, odd-even sort, count sort and quick sort did a good job. This is because merge sort does not check if the list is sorted or not. Merge sort algorithm takes an extra step to divide the list and then merge the all sublists. Other sorting algorithms keeps comparing data in the array list. Figure (c) also shows the insertion sort and odd-even sort are two time-consuming algorithms to sort data in an opposite order due to runtime $O(n^2)$. Figure (d) shows the insertion sort and odd-even sort in linked list behave slower comparing to array list. This is because in linked list we have to break and re-connect the link between nodes when we are sorting linked list. Overall, quick sort, merge sort and counting sort are three algorithms that are fast in sorting a list. However, counting sort can only be used to a list of integers and it is not good at dealing outliers in the list. Merge sort has a bigger space complexity, because it needs space for





(a) Sorting Times of Random Data for Array List. (b) Sorting Times of Sorted Data for Array List.





(c) Sorting Times of Data in Opposite Order for Array List. (d) Sorting Times of Random Data for Array List and Linked List.

all the sublists. Quick sort is fast when we have a good pivot. As a conclusion, there is no perfect sorting algorithm, but we can find a relatively ideal algorithm depending on different scenarios.

Algorithm	Runtime
Insertion Sort	$O(n^2)$
Odd-even Sort	$O(n^2)$
Counting Sort	$O(m+n)$
Quick Sort	$O(n\log(n))$
Merge Sort	$O(n\log(n))$

Table 1: Runtime for Different Sorting Algorithms