PROJECT02

COMSC340 Spring 2018

Dhaval Patel, Ian King, Zack Hilton

Abstract

This report details the approach and results of determining the relationship between actual and theoretic function growth. It compares different physical data structures and their impact on efficiency using INSERTION sort. MERGESORT is used to compare theoretic and actual runtimes. It then compares two means of measuring time: time and step counting. It is a team project.

Contents

Abbreviations and Glossary 3
Introduction 4
Requirements 4
Design & Implementation 4
Development Environment 5
Target Environment 5
Testing 5
Summation and Conclusions 5
Acknowledgements 5
References 5
Appendices 6
Requirements 6
Design 6
Tests 6
Traceability Matrix 6
Planned and Actual Schedule 6
Manuals 6

Abbreviations and Glossary

Introduction

Prior to this experiment, we were exposed to linked lists, insertion sort and merge sort in COMSC 111, Data Structures, offered at Roger Williams University. In that course we learned how the sorting algorithms and the linked list data structure worked, what their time complexities were, and the general outline of the code. Using this prior knowledge, we coded each of the algorithms and carefully selected where to increment the step counter and where to time our code, which was the main objective of this experiment. Going into the project, we figured the hardest part would be ensuring our code is giving us the correct results, and implementation of where the steps would be counted. By analyzing the time and steps for each of our implementations, we can see how the input size and order of elements in lists affect the time complexities of Insertion Sort using Arrays, Insertion Sort using Linked Lists and Merge Sort using Arrays.

Requirements

This project focuses on the relation between the theoretical analysis of an algorithm and the analysis of an algorithm running on a real machine, along with showing differences between logical and physical algorithm data structures. There were four main requirements for this project. Firstly, it was required to implement insertion sort using arrays, insertion sort using linked lists, and merge sort using arrays. After implementation, it was required to test given data sets of sequential, reverse, and random order with 5k, 10k, and 100k numbers against the algorithms. After testing the algorithms against the data sets, the first analysis was to analyze the time it took to run each of the data sets through the algorithms. Next, we needed to analyze how the times change while the file grows in input sizes and the order of the data sets change. The growth rates analyzed in the previous parts were then to be used to determine if the theoretical growth rates seem reasonable compared to actual rates. The last requirement for each of the different algorithms was to count the steps (swaps and comparisons) it took to sort the data sets with each algorithm and analyze to results.

Design & Implementation

Insertion Sort Using Arrays -

Knowing the theoretical time complexities for insertion sort is important before implementing our own sorting algorithm and testing it against different data sets. Depending on the state of the data sets we plug into the sorting algorithm, we can make educated guesses on how the algorithm is going to perform. For example, the table below shows the expected results of the different data files that will be run through the algorithm.

Order of Data in File	Expected Time Complexity
Sequential Order	O(n)
Reverse Order	O(n^2)
Random Order	O(n^2)

It is important to explain the reasoning behind the predictions for each data organization. Before we can do that, we need to understand how insertion sort works. Insertion sort works by taking an array which is expected to be unsorted and starts by comparing the first two items in the array. If the two items are not in ascending order, they get swapped. Next, it will check the second item in the array with the third, if they are not in ascending order, they will be swapped. Now, the algorithm checks the sub-list in the array (all numbers to the left of where the swap occurs) to see if the sub-list is still sorted. If the sub-list is not sorted, the algorithm will swap again as needed. The algorithm continues this trend throughout the unsorted array until it is sorted.

Now that we understand how insertion sort works, we can see the reasonings behind the predictions for the time complexities of each of the data files. Firstly, sequential order was predicted an O(n) time complexity. This is a fact because if the list is already sorted then the algorithm will make one pass through the list and do n comparisons. Next, random order was predicted to give a n^2, time complexity. This is because we do not know how close or far from sorted a list of random numbers is so we give it the worst case scenario time complexity to be safe. Thirdly, reverse order was predicted to give an O(n^2) time complexity also. This is true because as the algorithm works through the numbers, it is going to have to do the maximum amount of comparisons and swaps because it is as far from sorted as it can get. The structure and process of this portion of the project was directly designed in relation to this known information. In order to track the various results of this project, a few different design features were implemented to ensure accuracy and correctness of the results. The first goal of this project was to accurately time the runtime of the insertion sort. To do this, the insertion sort code was put into a function, then the function call was put in between two variables that recorded the system time in nanoseconds. Getting a runtime for the sorting function was as simple as taking these two variables and subtracting the starting system time from the ending system time. Coordinating data was easy as there was three groups of files: sequential order, reverse order, and random order. Each group had three input sizes of 5k, 10k, and 100k. Grouping these files regarding their order made it easy to analyze the results because they were related.

Insertion Sort using Linked Lists-

Insertion sort algorithm remains the same as when using arrays. However, the amount of time it takes for sorting may be impacted. Knowing this, the Linked List implementation of insertion sort was going to involve more steps as there are no index values to go off of. The Linked List implementation would store the references to all the necessary nodes and change the references as the List is traversed. The main challenge of this implementation was to figure out a way to keep track of the references and then swapping the nodes as necessary. The implementation was done in Java. The data was coordinated by the order as this has a big impact in the way this algorithm performs. While implementing, the steps were taken in account. The steps in this implementation is a combination of comparisons and swaps. A steps variable was incremented every time there was an if-else block and right after the code for swaps. This implementation was done by creating a new list within the method that is sorted. Comparisons were done with a nested loop. The outer loop traverses the original list while the inner loop is responsible for putting the value in its correct order in the new list by making use of a key variable to compare with. The last part to be implemented was the driver(main) method with the use of a java module called nanotime to track the execution time of the sorting method.

Merge Sort using array-

Merge sort splits an array of size n into smaller arrays of n/2 until each array size is 1. Then the arrays will sort themselves in the reverse order in which they split up. To time the results, nanotime() began timing before putting our array into merge sort, and stopped timing after merge sort was finished. A total of 3 runs were completed prior to recording results- 3 results were recorded and the average of that is plotted on the graphs below. The different problems were run in merge sort by data order and increasing size. We started off with in-order (5, 10, 100), random-order (5, 10, 100), and finally reverse-order (5,10,100). The main challenge in this project was to determine the steps and how to count them. A static variable, steps, was incremented after every comparison and swap. More specifically, after (and inside) every if-else statement and while/for loop. The results were copied over to an excel sheet and transformed into graphs and charts.

Development Environment

Hardware:

Group Member	*Machine Details*
Zack Hilton	*Dell XPS 15 9570, OS: Windows 10, CPU: Intel(R) Core(TM) i9-8950HK @ 2.9GHz, RAM: 32GB*
Ian King	*HP 15 notebook PC , OS: Microsoft 10 64 bit, RAM: 8GB, CPU: AMD A8-6410 APU*
Dhaval Patel	*HP Spectre x360, OS: Windows 10, CPU: Intel i7 8550U, RAM: 16GB*

Software:

Group Member	Programming Language & IDE
Zack Hilton	Language: Java IDE: JGrasp
Ian King	Language: Java IDE: Eclipse
Dhaval Patel	Language: Java IDE: IntelliJ Idea

Target Environment

Hardware:

Group Member	*Machine Details*
Zack Hilton	*Dell XPS 15 9570, OS: Windows 10, CPU: Intel i9-8950HK @ 2.9GHz, RAM: 32GB*
Ian King	*HP 15 notebook PC , OS: Microsoft 10 64 bit, RAM: 8GB, CPU: AMD A8-6410 APU*
Dhaval Patel	*HP Spectre x360, OS: Windows 10, CPU: Intel i7 8550U, RAM: 16GB*

Software:

Group Member	Programming Language & IDE
Zack Hilton	Language: Java IDE: JGrasp
Ian King	Language: Java IDE: Eclipse
Dhaval Patel	Language: Java IDE: IntelliJ Idea

Testing

Insertion Sort Using Arrays -

After the requirements and design and development was done for this project, the testing was next to be done. Before testing and recording results, verification of accuracy was required to ensure correctness. To verify correctness of the insertion sort using arrays code, I began by sorting smaller arrays of numbers. To determine that my code was working properly and insertion sorting, I put small scale arrays of the given data files to test. When testing these small-scale examples, the array contents could easily be modified to mirror the sequential, reverse, and random ordered number sets given for testing. Using scaled down arrays made it easy to display the results and confirm correctness. This proved to be an effective way to confirm that the sorting algorithm code was working properly and would not give incorrect results on the large data sets without our knowledge. After I was confident that the results the program was producing, I began the testing of data files. To test the insertion sort using the provided data files I fed each of the files into my program 10 times a piece in order to get average runtimes for each of the nine files. This also ensured accuracy of results.

Insertion sort using linked lists -

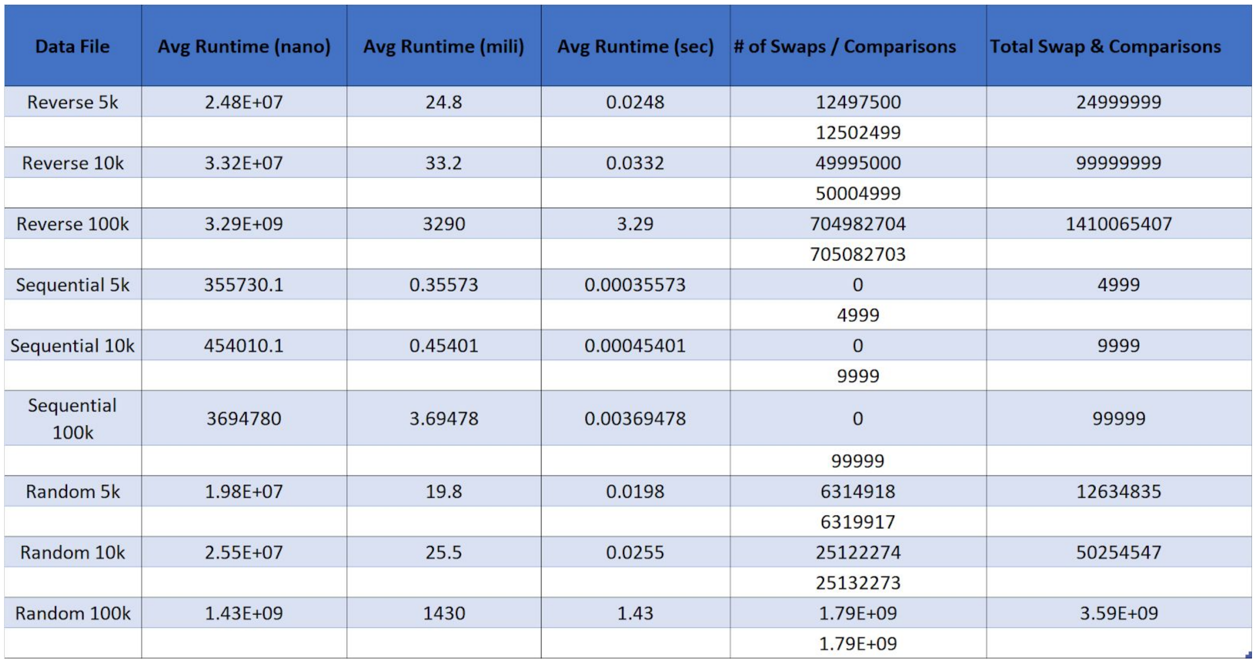
After the tricky implementation, testing was done on a test file of 10 integers in various order (inorder, reverse and random). The implementation successfully sorted the small sample. In addition to this, the amounts of steps were observed and the data showed a rough n(n-1)/2 steps which was expected. There was a strange occurrence of inorder list being the worse-case scenario while reverse order being the best case. This was unexpected but upon further investigation, the implementation seemed like the most likely cause. Specifically, the order in which the comparisons were made seemed to reverse the order in which each element was sorted. However, the algorithm for insertion sort remains the same and the average case(random order) was not impacted. The worst case can be categorized as inorder for this implementation. Thus, the purpose of analyzation of time complexities can be fulfilled without starting the tricky and hard to read implementation over. The decision that the implementation was producing the wanted result for the purpose of this project was made and left the way it was. After this, the tests on the provided data files were done and the times/steps were recorded in Excel. There were 5 trials done to ensure an accurate average time for each data file. The number of steps remained the same throughout the trials.

Merge Sort using Arrays-

To confirm the correctness of code, a small test file was created with a sample array of size 6. This was fed into the merge sort algorithm. Once it was proven to be correct, the various data files were read in and tested. Each of the data files, as mentioned above, were run 3 times through (without recording the results) followed by recording 3 runs and taking their average. Running the program 3 times through in the beginning allows Java to become familiar with the operation, thus limiting the drastic change in results after feeding it in the first few times. This prevented a drastic standard deviation, and made it easier to look at the time complexity of Merge Sort.

Summation and Conclusions

Insertion Sort Using Arrays -



Data File	Avg Runtime (nano)	Avg Runtime (mili)	Avg Runtime (sec)	# of Swaps / Comparisons	Total Swap & Comparisons
Reverse 5k	2.48E+07	24.8	0.0248	12497500	24999999
				12502499	
Reverse 10k	3.32E+07	33.2	0.0332	49995000	99999999
				50004999	
Reverse 100k	3.29E+09	3290	3.29	704982704	1410065407
				705082703	
Sequential 5k	355730.1	0.35573	0.00035573	0	4999
				4999	
Sequential 10k	454010.1	0.45401	0.00045401	0	9999
				9999	
Sequential 100k	3694780	3.69478	0.00369478	0	99999
				99999	
Random 5k	1.98E+07	19.8	0.0198	6314918	12634835
				6319917	
Random 10k	2.55E+07	25.5	0.0255	25122274	50254547
				25132273	
Random 100k	1.43E+09	1430	1.43	1.79E+09	3.59E+09
				1.79E+09	

After testing and accumulating results for the insertion sorting algorithm using arrays, there are clear patterns that can be recognized between file sizes, time growth, and number of swaps and comparisons. The first group of numbers that was run in the testing process was the numbers in reverse order with inputs of 5k, 10k, and 100k. If you look back at the runtime charts, we can see that the average runtimes for the files are 24 milliseconds for 5k, 33.2 milliseconds for 10k, and 3290 milliseconds for 100k. Looking at the averages we can see that the growth for the insertion sort for reverse order is very steep as the input sized rise past the smaller inputs. Before running the tests, it was predicted that the reverse order was going to run with a worst-case time complexity of n^2. We can see now that this is nearly true. It is also informative to look at the number of swaps and comparisons that happen in each of the three input sizes. The swaps total roughly from 25 million to 100 million to 1.4 billion. This also supports the n^2 theoretical complexity. Next, the sequential order was tested. The runtimes for this were predicted to be O(n) and the actual runtimes came out to .35573, .45401, and 3.694, milliseconds. The number of swaps and comparisons for these supports the O(n) theory as they were 4999, 9999, and 99999, which is O(n-1), almost exactly the theoretical O(n). The third and last group to be tested were the random files. The runtimes for this were 19.8, 25.5, and 1430 milliseconds. These runtimes are consistent with the other groups and an exponential growth. The number of swaps and comparisons also match up with an exponential growth with 12 million, 50 million, and 3.5 billion. In conclusion, the actual results of the testing nearly match up with what was expected theoretically. The only thing that was different from theory was the runtimes of the sequential group. Although the runtimes were extremely small, they followed the same growth pattern as the other two groups.

Insertion sort using Linked Lists-

Lists	Trial 1	Trial 2	Trial 3	Trial 4	Trial 5	Average (ms)
In-Order 5k	150.789335	127.391	157.6098	153.7159	155.4287	148.9869482
In-Order 10k	585.067675	595.2532	577.6442	576.1267	600.823	586.9829484
In-Order 100k	72969.78271	72869.26	64044.67	66421.83	65520.5	68365.20873
Random 5k	179.612504	233.4394	195.1509	218.9836	194.8538	204.408038
Random 10k	862.673302	854.5651	910.4066	857.3929	874.7367	871.9549072
Random 100k	159922.6864	156585.5	152620.7	153885.2	155110.7	155624.9449
Reverse 5k	2.658703	2.619119	2.037207	2.562059	2.650477	2.505513
Reverse 10k	3.759298	2.885402	3.520262	3.531571	4.181853	3.5756772
Reverse 100k	17.33297	18.73069	14.29027	14.0903	17.63524	16.4158924

Lists	Steps
In-Order 5k	12497500
In-Order 10k	49995000
In-Order 100k	704982704
Random 5k	12501876
Random 10k	50004553
Random 100k	705094597
Reverse 5k	9998
Reverse 10k	19998
Reverse 100k	199998

After acquiring the data above, there are a few conclusions that can be made. In terms of the steps, the combination of comparisons and swaps increase dramatically as the input size increases but an important thing to notice is the worse-case(in-order) and average-case(random) scenarios in this implementation are very similar. The time-complexities of both also are of interest. As they are similar. The random order is a bit higher in the execution times. This is because of the list elements in the random order list. Although, the input size remains the same, the magnitude of values of the elements involved is higher and this affects the time to sort the lists. On average, the time complexities are expected to be similar for both average case and worst case scenarios so this data is accurately representing the theoretical growth for this algorithm. The steps also give us accurate results as the theoretical amount of steps are n(n-1)/2 for the average and worst case and the actual results reflect this closely. The results were graphed as well to visualize the growth.

Merge Sort using arrays:

inorder	5	10	100
trial 1	4	6	156
trial 2	4	7	158
trial 3	5	7	160
time (ms)	4.33333333	6.66666667	158
trial 1	""	""	""
trial 2	""	""	""
trial 3	1.98223	4.26447	5.267903
steps (in millions)	1.98223	4.26447	5.267903
reverse order	5	10	100
trial 1	4	8	165
trial 2	4	10	157

trial 3	5	6	181
time (ms)	4.33333333	8	167.666667
trial 1	'''	'''	'''
trial 2	'''	"""	53.45663
trial 3	2.02623	4.35247	53.45663
steps (in hundred thousands)	2.02623	4.35247	53.45663
random order	5	10	100
trial 1	6	8	130
trial 2	4	8	129
trial 3	4	7	134
time (ms)	4.66666667	7.66666667	131
trial 1	"""	"""	"""
trial 2	"""	"""	"""
trial 3	0.249153	0.538205	6.711251
steps (in millions)	0.249153	0.538205	6.711251

(Quotation marks represent the same value, since the # of steps will be the same when running the same file)

After looking at the chart above, it is evident that both random order is the worst case, followed by in-order. In most cases these two should be much closer, however in this implementation there is a “steps gap” by about 1 million in a problem size of 100,000. The best case in merge sort is also n-logn, as seen by the reverse order steps. This highlights a versatile quality of merge sort- feed it any array, sorted or unsorted, and you will get a similar number of steps (comparisons & swaps) for each situation. Time complexity of this implementation is modeled by: T(n) = 2T(n/2) + O(n); T(n/2) to split the arrays, and the O(n) would describe the process of merging the arrays back together. The quickest sort was random order (131 milliseconds), followed by in-order (158 milliseconds), and finally reverse-order (167.77 milliseconds). Both the time complexity and step counts are graphed below, which show the similarity between the three orderings.

Acknowledgements

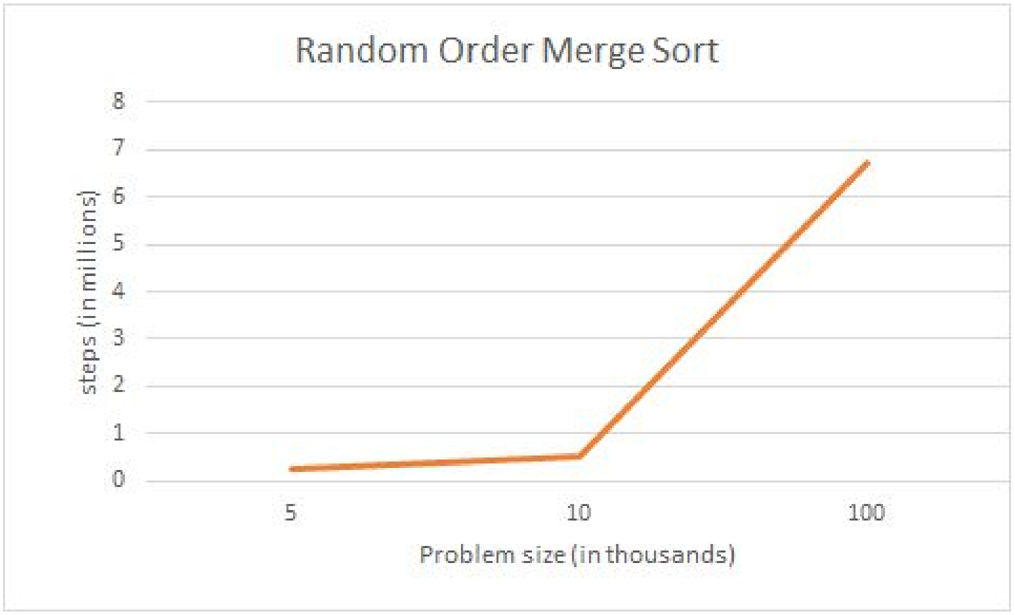
Group members

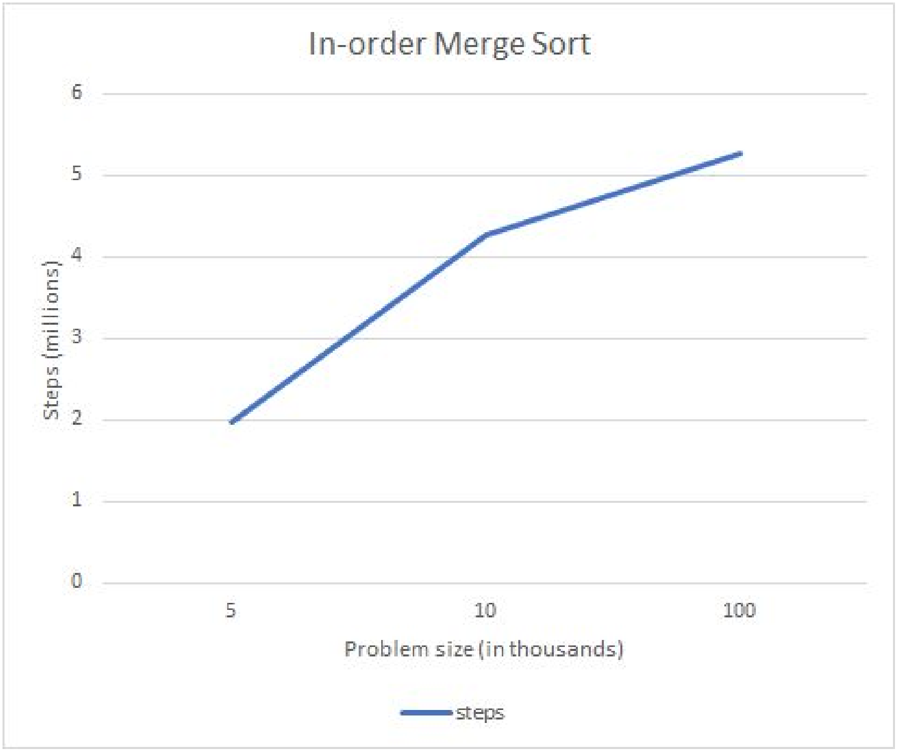
References

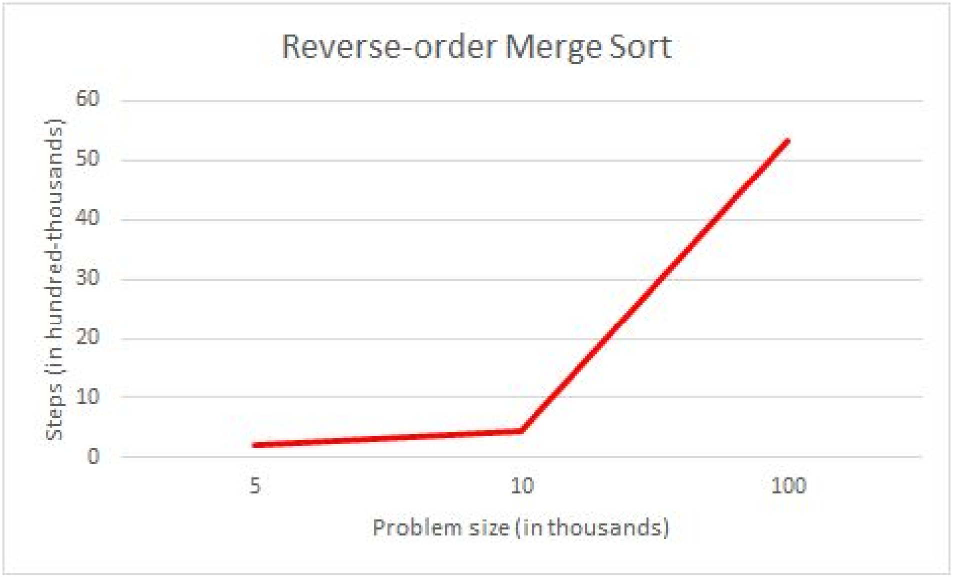
COMSC 111 jsjf package for LinkedList and LinearNode implementation

Appendices

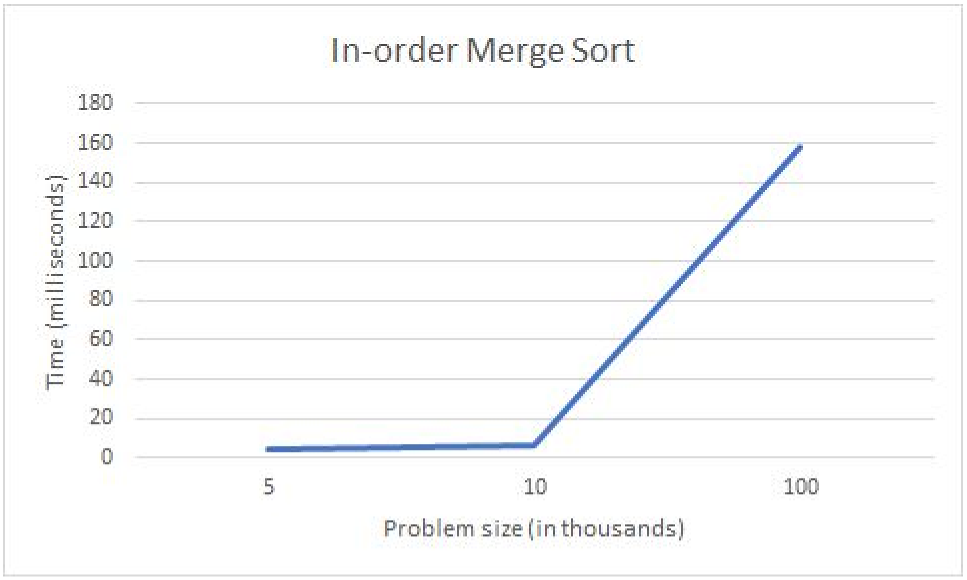
Merge Sort Array implementation: Step graphs

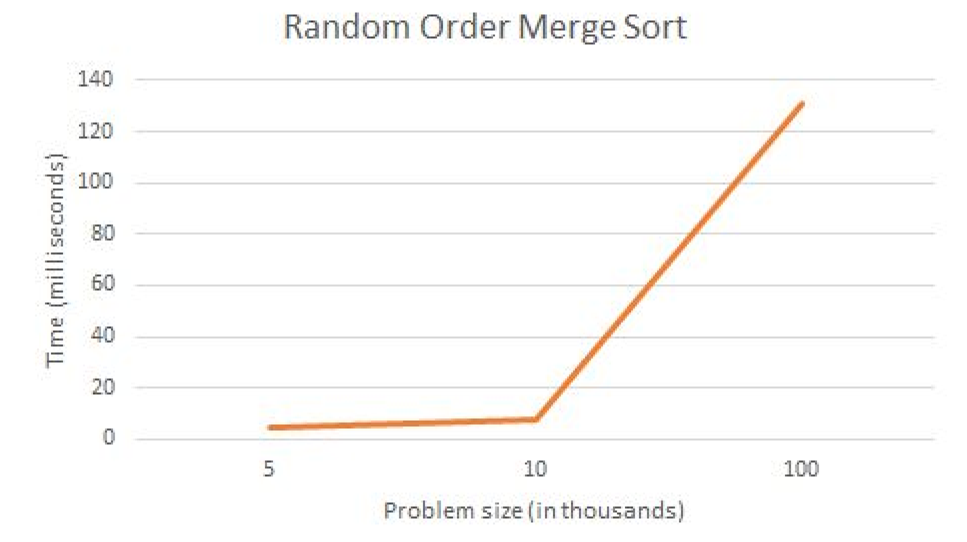


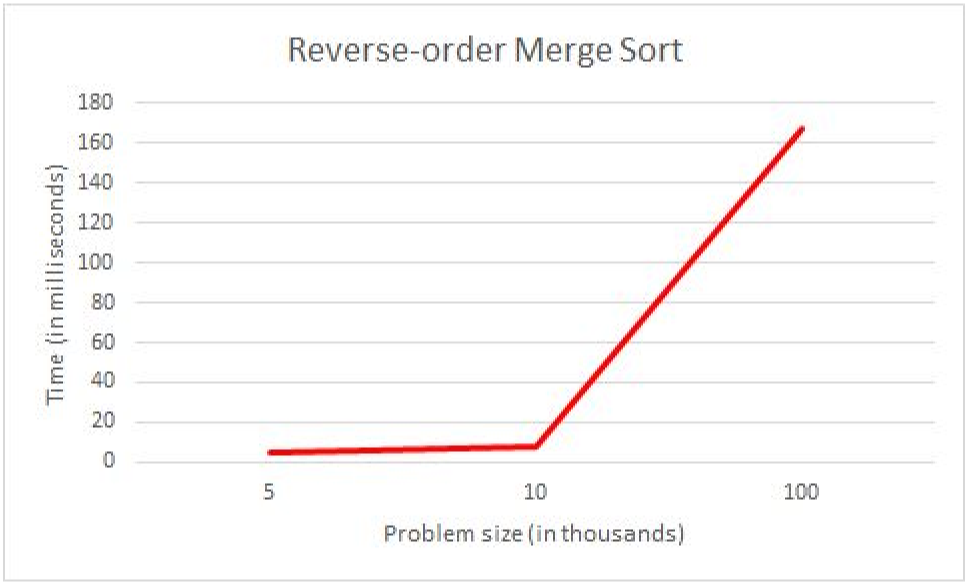

In-order Merge Sort
6
5
4
3
2
1
0
Steps (millions)
5
10
100
Problem size (in thousands)
steps


Reverse-order Merge Sort
60
50
40
30
20
10
0
Steps (in hundred-thousands)
5
10
100
Problem size (in thousands)

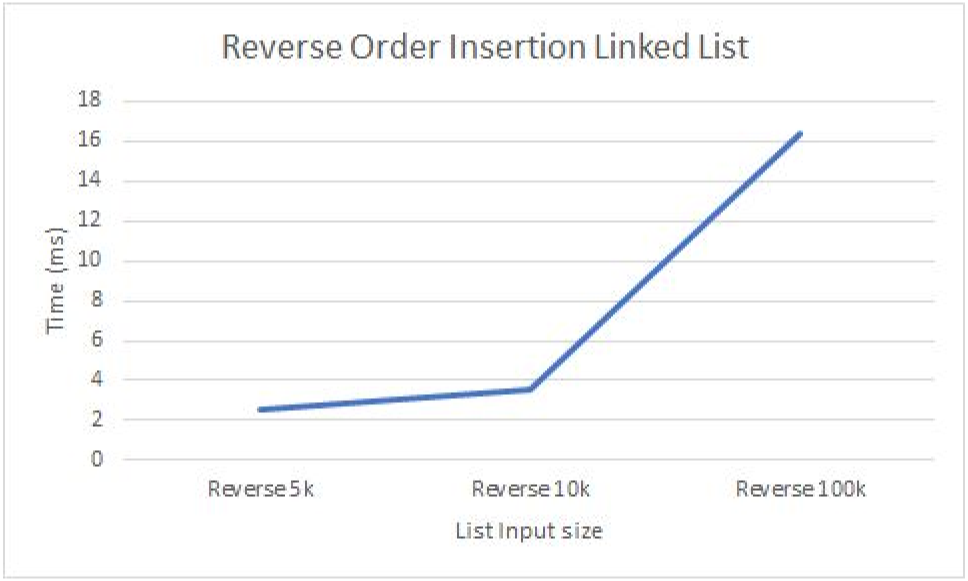
Merge Sort time graphs:

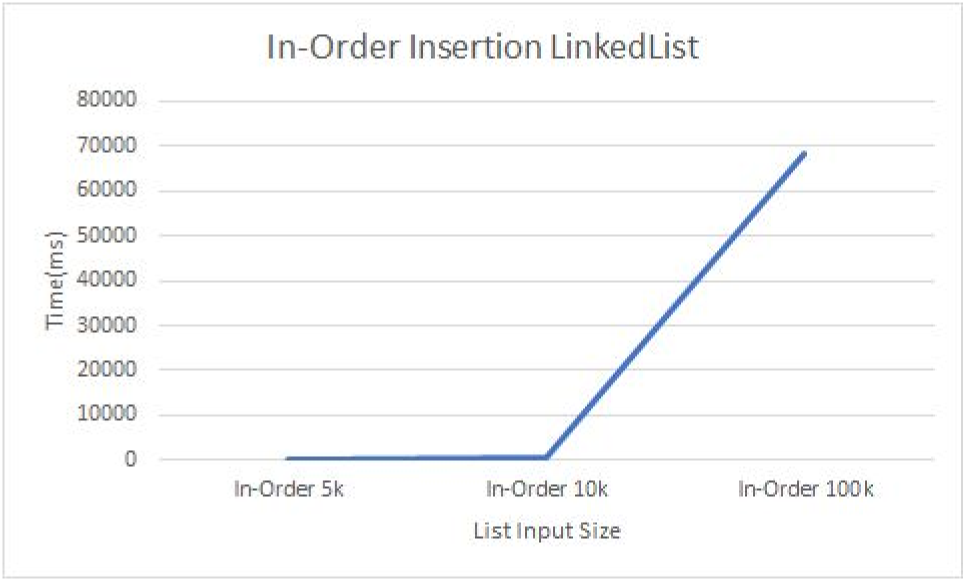


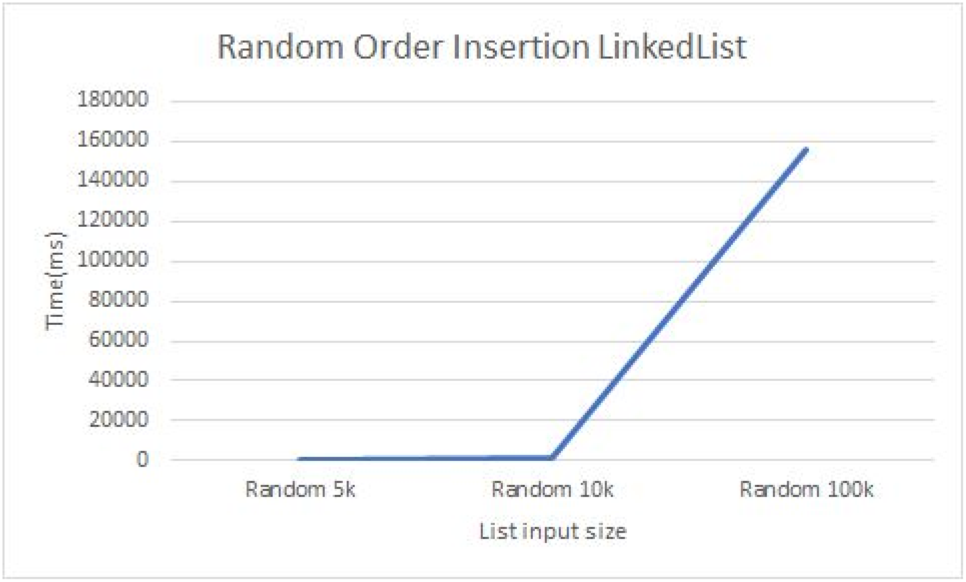




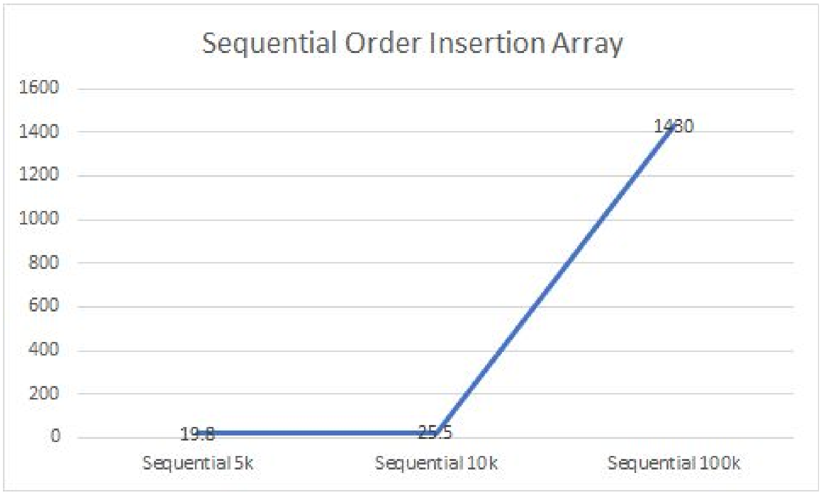
Insertion Sort Linked List Graphs-

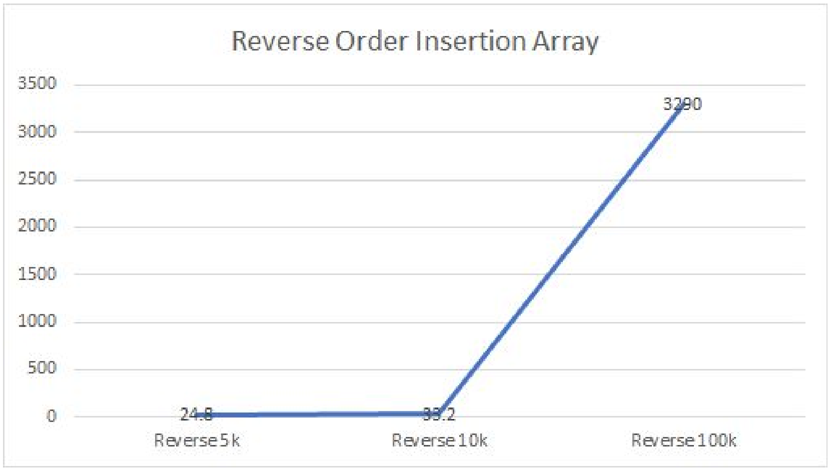


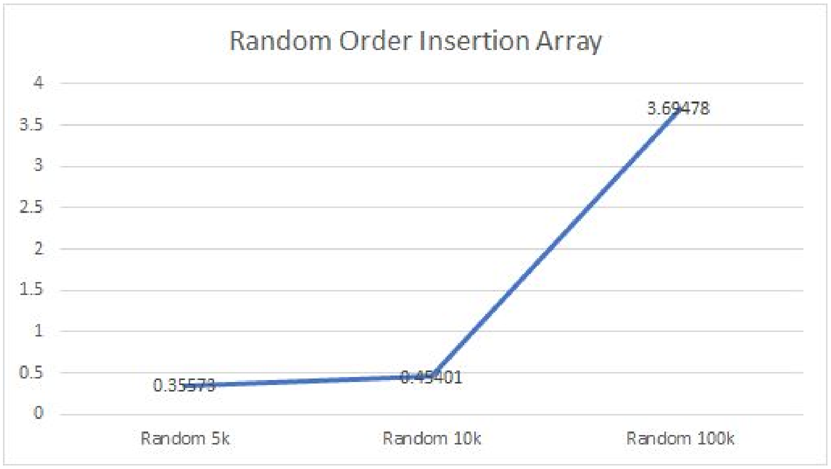

In-Order Insertion LinkedList
80000
70000
60000
50000
40000
30000
20000
10000
0
Time(ms)
In-Order 5k
In-Order 10k
In-Order 100k
List Input Size


Random Order Insertion LinkedList
180000
160000
140000
120000
100000
80000
60000
40000
20000
0
Time(ms)
Random 5k
Random 10k
Random 100k
List input size

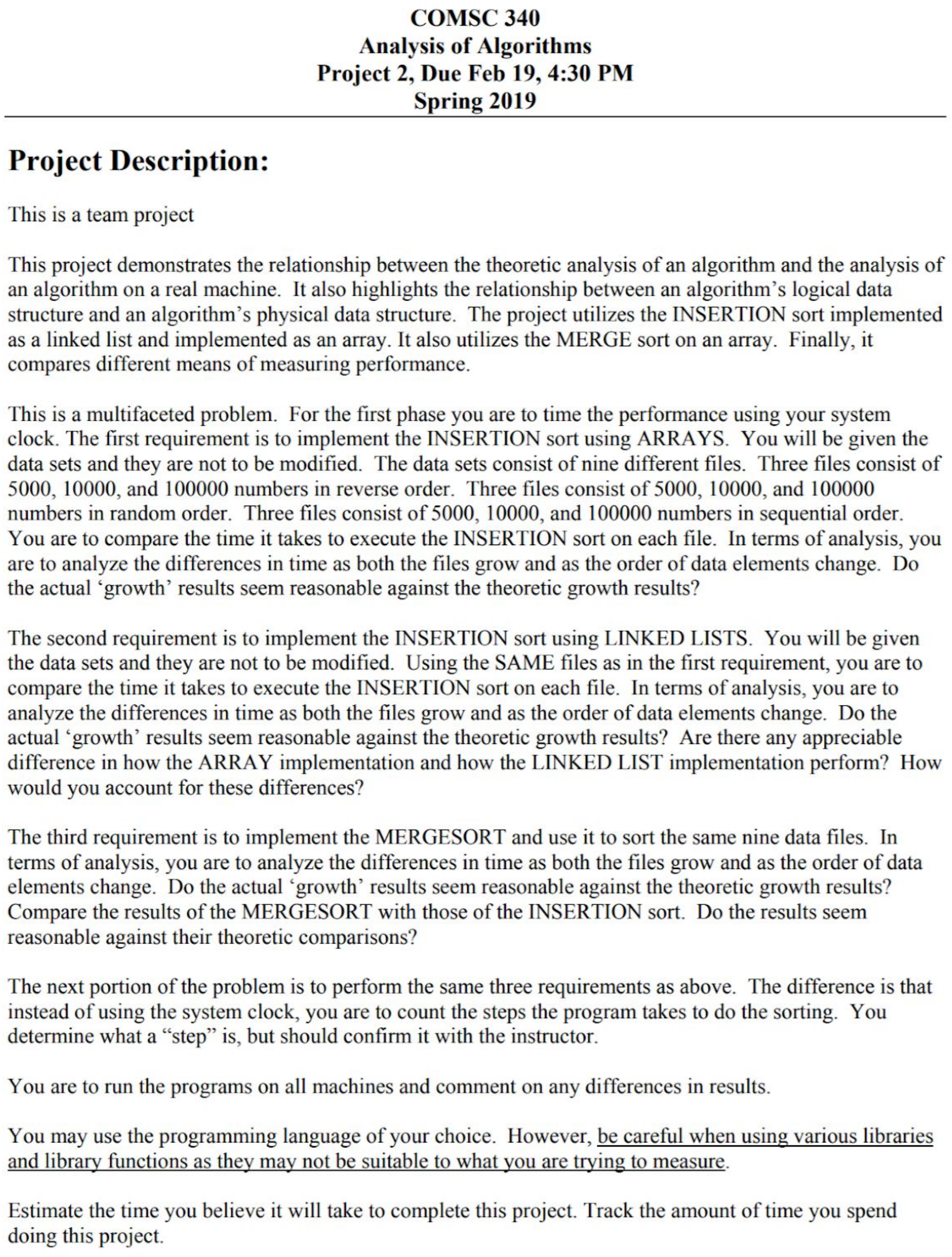
Insertion Sort Array graphs:







Requirements



COMSC 340
Analysis of Algorithms
Project 2, Due Feb 19, 4:30 PM
Spring 2019

Project Description:

This is a team project

This project demonstrates the relationship between the theoretic analysis of an algorithm and the analysis of an algorithm on a real machine. It also highlights the relationship between an algorithm's logical data structure and an algorithm's physical data structure. The project utilizes the INSERTION sort implemented as a linked list and implemented as an array. It also utilizes the MERGE sort on an array. Finally, it compares different means of measuring performance.

This is a multifaceted problem. For the first phase you are to time the performance using your system clock. The first requirement is to implement the INSERTION sort using ARRAYS. You will be given the data sets and they are not to be modified. The data sets consist of nine different files. Three files consist of 5000, 10000, and 100000 numbers in reverse order. Three files consist of 5000, 10000, and 100000 numbers in random order. Three files consist of 5000, 10000, and 100000 numbers in sequential order. You are to compare the time it takes to execute the INSERTION sort on each file. In terms of analysis, you are to analyze the differences in time as both the files grow and as the order of data elements change. Do the actual 'growth' results seem reasonable against the theoretic growth results?

The second requirement is to implement the INSERTION sort using LINKED LISTS. You will be given the data sets and they are not to be modified. Using the SAME files as in the first requirement, you are to compare the time it takes to execute the INSERTION sort on each file. In terms of analysis, you are to analyze the differences in time as both the files grow and as the order of data elements change. Do the actual 'growth' results seem reasonable against the theoretic growth results? Are there any appreciable difference in how the ARRAY implementation and how the LINKED LIST implementation perform? How would you account for these differences?

The third requirement is to implement the MERGESORT and use it to sort the same nine data files. In terms of analysis, you are to analyze the differences in time as both the files grow and as the order of data elements change. Do the actual 'growth' results seem reasonable against the theoretic growth results? Compare the results of the MERGESORT with those of the INSERTION sort. Do the results seem reasonable against their theoretic comparisons?

The next portion of the problem is to perform the same three requirements as above. The difference is that instead of using the system clock, you are to count the steps the program takes to do the sorting. You determine what a "step" is, but should confirm it with the instructor.

You are to run the programs on all machines and comment on any differences in results.

You may use the programming language of your choice. However, be careful when using various libraries and library functions as they may not be suitable to what you are trying to measure.

Estimate the time you believe it will take to complete this project. Track the amount of time you spend doing this project.

Design

The Linked List implementation used for this project is structured from the LinearNode class from COMSC 111. The LinkedList is singly linked. For the purpose of this project the access modifiers were changed for a little bit easier implementation of insertion sort. Specifically, the head and tail variables were made public to allow for easier node access and creation of new lists from nodes.

Tests

Insertion Sort Using Arrays -

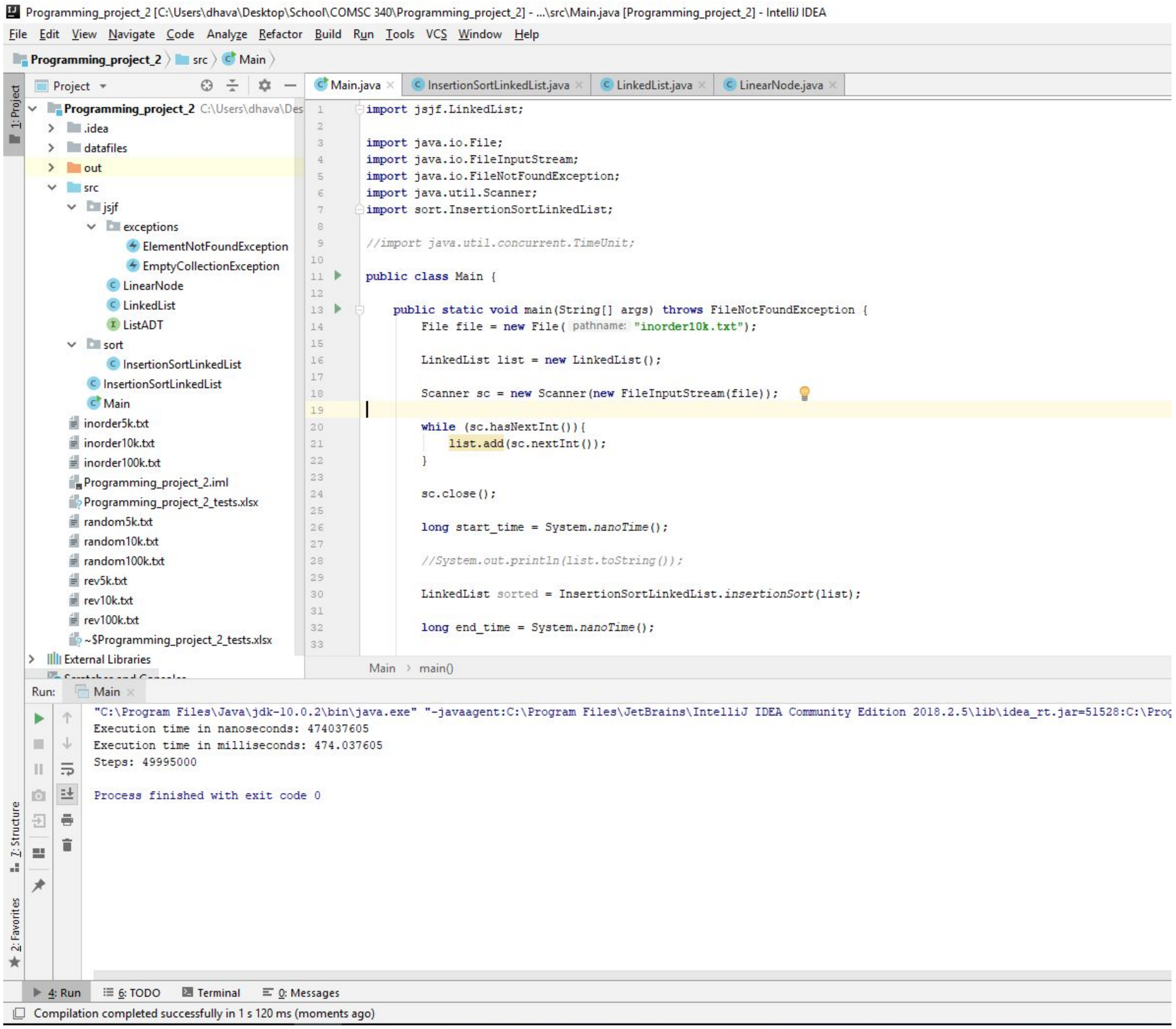
Sequential 5k		Sequential 10k		Sequential 100k	
1	235800	1	429400	1	3239701
2	339900	2	370201	2	4540600
3	438100	3	360100	3	3249001
4	210800	4	554700	4	3798199
5	219600	5	669600	5	3243600
6	473601	6	530600	6	3232200
7	173001	7	351500	7	3680601
8	269399	8	343600	8	3280999
9	231700	9	528000	9	4952700
10	965400	10	402400	10	3730200
Avg (nano)	355730.1	Avg (nano)	454010.1	Avg (nano)	3694780
Avg (mili)	0.35573	Avg (mili)	0.45401	Avg (mili)	3.69478
Swaps	0	Swaps	0	Swaps	0
Comparisons	4999	Comparisons	9999	Comparisons	99999
Total	4999	Total	9999	Total	99999

Reverse 5k		Reverse 10k		Reverse 100k	
1	4.47E+07	1	3.61E+07	1	4.03E+07
2	3.74E+07	2	5.51E+07	2	5.47E+07
3	1.71E+07	3	3.84E+07	3	8.92E+09
4	1.23E+07	4	3.34E+07	4	6.58E+09
5	1.27E+07	5	2.73E+07	5	3.22E+09
6	3.70E+07	6	2.83E+07	6	3.18E+09
7	1.92E+07	7	2.91E+07	7	2.27E+09
8	1.83E+07	8	3.01E+07	8	4.66E+09
9	2.35E+07	9	2.68E+07	9	3.92E+09
10	2.58E+07	10	2.71E+07	10	5.51E+07
Avg (nano)	2.48E+07	Avg (nano)	3.32E+07	Avg (nano)	3.29E+09
Avg (mili)	2.48E+01	Avg (mili)	3.32E+01	Avg (mili)	3.29E+03
Swaps	12497500	Swaps	49995000	Swaps	704982704
Comparisons	12502499	Comparisons	50004999	Comparisons	705082703
Total	24999999	Total	99999999	Total	1410065407

Random 5k		Random 10k		Random 100k	
1	1.77E+07	1	2.59E+07	1	1.45E+09
2	2.22E+07	2	2.29E+07	2	1.54E+09
3	1.59E+07	3	2.96E+07	3	1.89E+09
4	2.47E+07	4	1.84E+07	4	1.29E+09
5	2.34E+07	5	3.87E+07	5	1.49E+09
6	1.69E+07	6	2.06E+07	6	1.48E+09
7	2.34E+07	7	2.96E+07	7	1.20E+09
8	1.96E+07	8	2.80E+07	8	1.35E+09
9	1.72E+07	9	2.08E+07	9	1.28E+09
10	1.66E+07	10	2.01E+07	10	1.32E+09
Avg (nano)	1.98E+07	Avg (nano)	2.55E+07	Avg (nano)	1.43E+09
Avg (mili)	1.98E+01	Avg (mili)	2.55E+01	Avg (mili)	1.43E+03
Swaps	6314918	Swaps	25122274	Swaps	1.79E+09
Comparisons	6319917	Comparisons	25132273	Comparisons	1.79E+09
Total	12634835	Total	50254547	Total	3.59E+09

inorder	5	10	100
	4	6	156
	4	7	158
	5	7	160
time (ms)	4.333333	6.666667	158
	""	""	""
	""	""	""
	1.98223	4.26447	5.267903
steps	1.98223	4.26447	5.267903
reverse or	5	10	100
	4	8	165
	4	10	157
	5	6	181
time (ms)	4.333333	8	167.6667
	"'	"'	"'
	"'	""	53.45663
	2.02623	4.35247	53.45663
steps	2.02623	4.35247	53.45663
random or	5	10	100
	6	8	130
	4	8	129
	4	7	134
time (ms)	4.666667	7.666667	131
	""	""	""
	""	""	""
	0.249153	0.538205	6.711251
steps	0.249153	0.538205	6.711251

Insertion Sort Linked List (Sample run)-



Programming_project_2 [C:\Users\dhava\Desktop\School\COMSC 340\Programming_project_2] - ...\src\Main.java [Programming_project_2] - IntelliJ IDEA

```
import jsjf.LinkedList;

import java.io.File;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.util.Scanner;
import sort.InsertionSortLinkedList;

//import java.util.concurrent.TimeUnit;

public class Main {

    public static void main(String[] args) throws FileNotFoundException {
        File file = new File( pathname: "inorder10k.txt");

        LinkedList list = new LinkedList();

        Scanner sc = new Scanner(new FileInputStream(file));

        while (sc.hasNextInt()){
            list.add(sc.nextInt());
        }

        sc.close();

        long start_time = System.nanoTime();

        //System.out.println(list.toString());

        LinkedList sorted = InsertionSortLinkedList.insertionSort(list);

        long end_time = System.nanoTime();
```

```
"C:\Program Files\Java\jdk-10.0.2\bin\java.exe" "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA Community Edition 2018.2.5\lib\idea_rt.jar=51528:C:\Prog
Execution time in nanoseconds: 474037605
Execution time in milliseconds: 474.037605
Steps: 49995000

Process finished with exit code 0
```

Compilation completed successfully in 1 s 120 ms (moments ago)

Planned and Actual Schedule

Zack Hilton:

Estimated Time: 10 hours

Group Meetings: Thursday, February 14 - 2 hours

Individual Time:

2/14 - 1.5 hours

2/16 - 1.5 hours

2/17 - 45 min

2/18 - 7 hours

Ian King:

Estimated time: 10 hrs

Group meetings: Thursday 14th meetup for 2hrs & group communication via email

Personal work:

2/17: 4 hrs

2/18: 6 hrs

Dhaval Patel:

Estimated time: 12 hrs

Group meetings: Thursday 2/14 - 2 hrs

Personal work time:

2/16 - 3 hrs

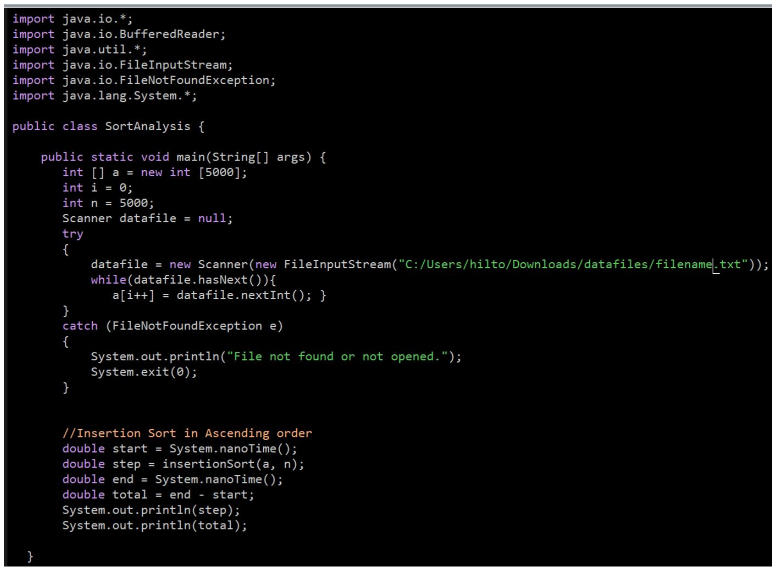
2/17 - 4 hrs 30 min

2/18 - 3 hrs 20 min

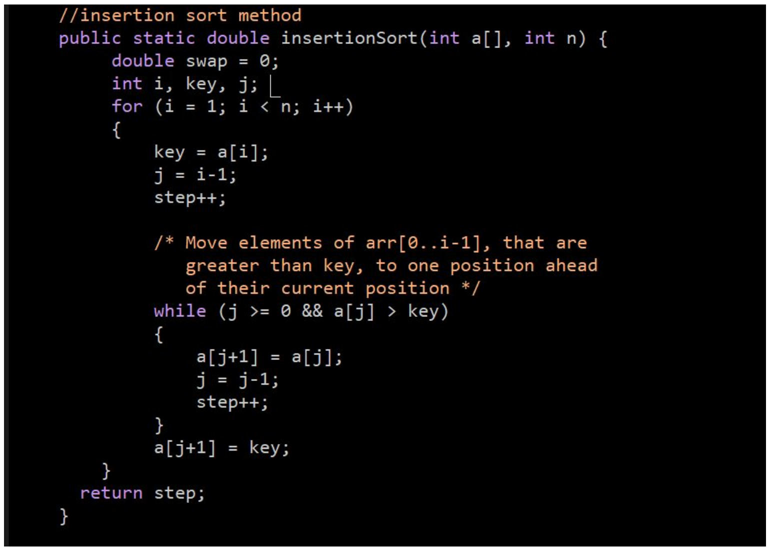
2/19 - 5 hrs

Manuals

Insertion Sort Using Arrays -

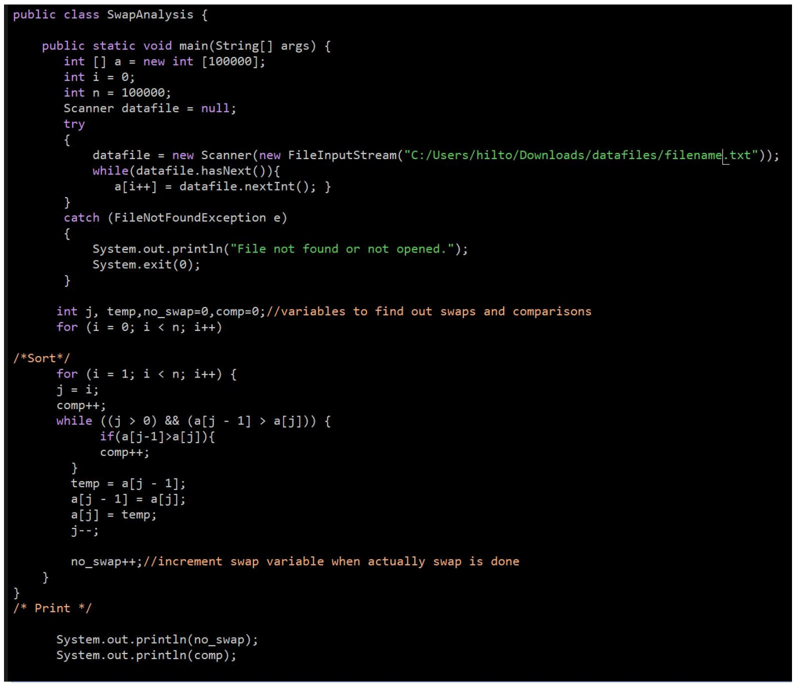


```java
import java.io.*;
import java.io.BufferedReader;
import java.util.*;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.lang.System.*;

public class SortAnalysis {

    public static void main(String[] args) {
       int [] a = new int [5000];
       int i = 0;
       int n = 5000;
       Scanner datafile = null;
       try
       {
           datafile = new Scanner(new FileInputStream("C:/Users/hilto/Downloads/datafiles/filename.txt"));
           while(datafile.hasNext()){
              a[i++] = datafile.nextInt(); }
       }
       catch (FileNotFoundException e)
       {
           System.out.println("File not found or not opened.");
           System.exit(0);
       }


       //Insertion Sort in Ascending order
       double start = System.nanoTime();
       double step = insertionSort(a, n);
       double end = System.nanoTime();
       double total = end - start;
       System.out.println(step);
       System.out.println(total);

  }
```



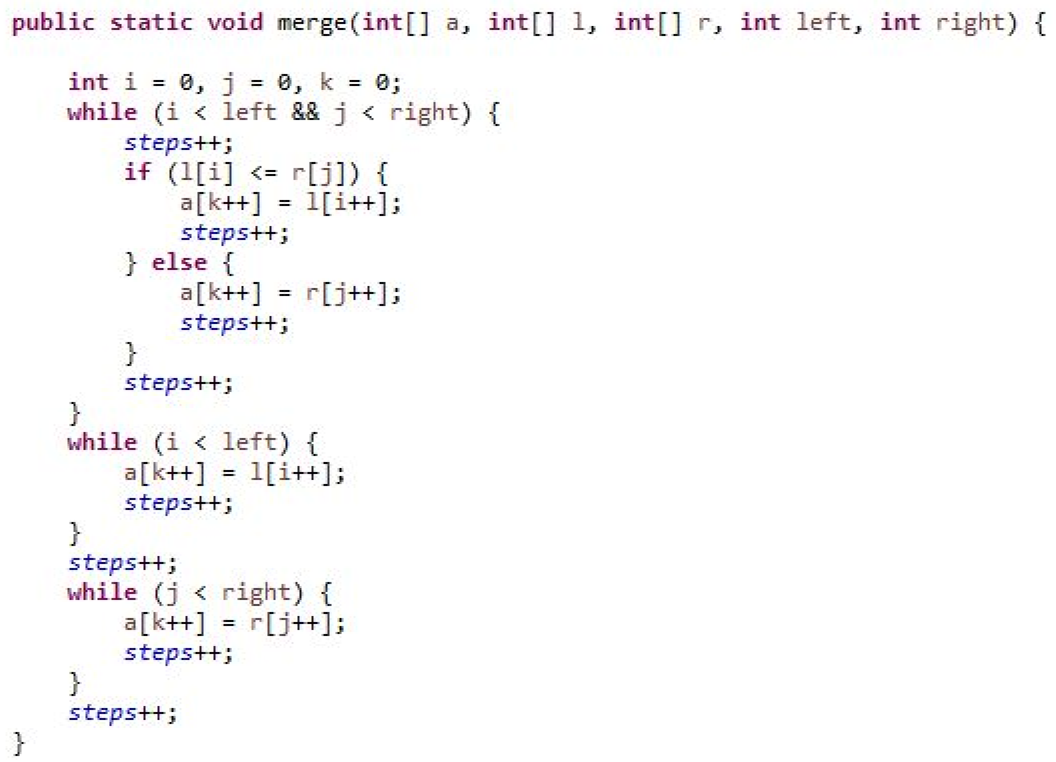
```java
    //insertion sort method
    public static double insertionSort(int a[], int n) {
         double swap = 0;
         int i, key, j;
         for (i = 1; i < n; i++)
         {
             key = a[i];
             j = i-1;
             step++;

             /* Move elements of arr[0..i-1], that are
                greater than key, to one position ahead
                of their current position */
             while (j >= 0 && a[j] > key)
             {
                 a[j+1] = a[j];
                 j = j-1;
                 step++;
             }
             a[j+1] = key;
        }
      return step;
    }
```

Insertion Sort Using Arrays (counting swaps and comparisons) -

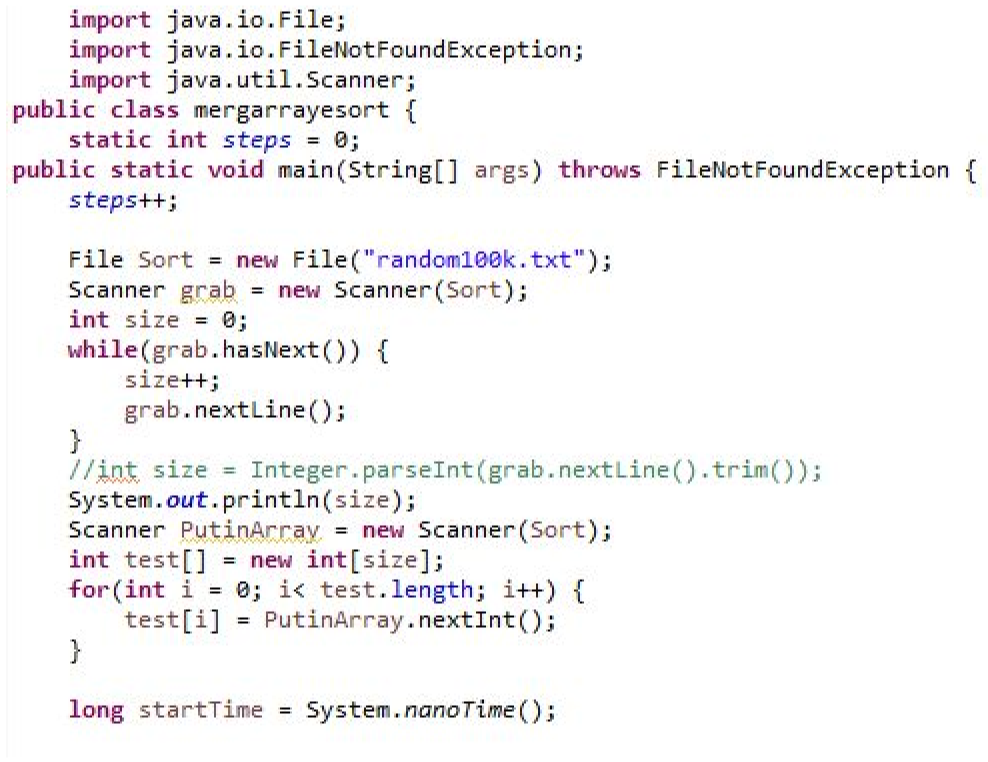


```
public class SwapAnalysis {

    public static void main(String[] args) {
       int [] a = new int [100000];
       int i = 0;
       int n = 100000;
       Scanner datafile = null;
       try
       {
           datafile = new Scanner(new FileInputStream("C:/Users/hilto/Downloads/datafiles/filename.txt"));
           while(datafile.hasNext()){
              a[i++] = datafile.nextInt(); }
       }
       catch (FileNotFoundException e)
       {
           System.out.println("File not found or not opened.");
           System.exit(0);
       }

      int j, temp,no_swap=0,comp=0;//variables to find out swaps and comparisons
      for (i = 0; i < n; i++)

/*Sort*/
      for (i = 1; i < n; i++) {
      j = i;
      comp++;
      while ((j > 0) && (a[j - 1] > a[j])) {
            if(a[j-1]>a[j]){
            comp++;
        }
        temp = a[j - 1];
        a[j - 1] = a[j];
        a[j] = temp;
        j--;

        no_swap++;//increment swap variable when actually swap is done
    }
}
/* Print */

      System.out.println(no_swap);
      System.out.println(comp);
```

Merge Sort using Arrays:

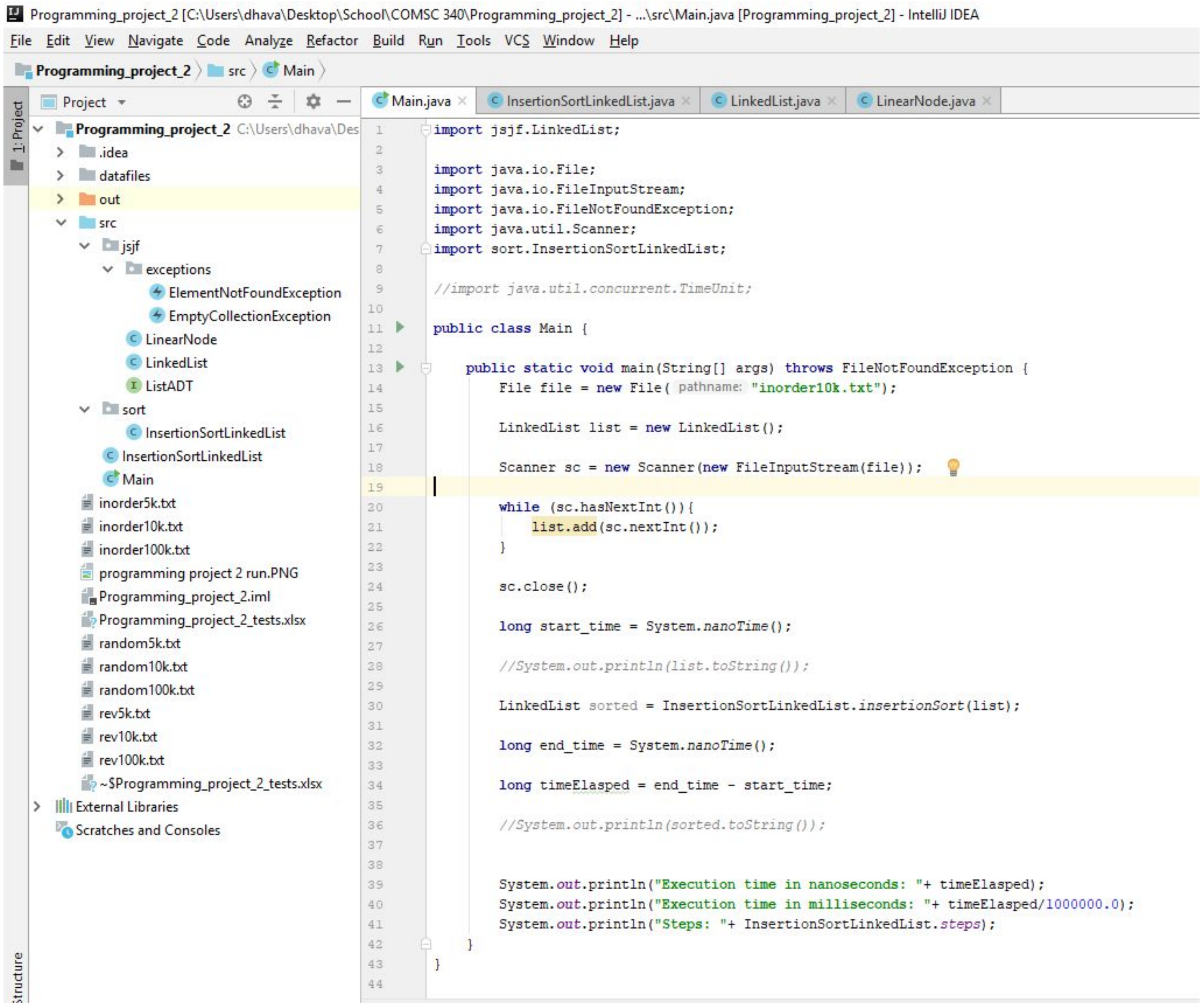


```java
    public static void merge(int[] a, int[] l, int[] r, int left, int right) {

        int i = 0, j = 0, k = 0;
        while (i < left && j < right) {
            steps++;
            if (l[i] <= r[j]) {
                a[k++] = l[i++];
                steps++;
            } else {
                a[k++] = r[j++];
                steps++;
            }
            steps++;
        }
        while (i < left) {
            a[k++] = l[i++];
            steps++;
        }
        steps++;
        while (j < right) {
            a[k++] = r[j++];
            steps++;
        }
        steps++;
    }
```



```java
    import java.io.File;
    import java.io.FileNotFoundException;
    import java.util.Scanner;
public class mergarrayesort {
    static int steps = 0;
public static void main(String[] args) throws FileNotFoundException {
    steps++;

    File Sort = new File("random100k.txt");
    Scanner grab = new Scanner(Sort);
    int size = 0;
    while(grab.hasNext()) {
        size++;
        grab.nextLine();
    }
    //int size = Integer.parseInt(grab.nextLine().trim());
    System.out.println(size);
    Scanner PutinArray = new Scanner(Sort);
    int test[] = new int[size];
    for(int i = 0; i< test.length; i++) {
        test[i] = PutinArray.nextInt();
    }

    long startTime = System.nanoTime();
long startTime = System.nanoTime();

mergeSort(test, test.length);

long endTime = System.nanoTime();

long runTime = endTime - startTime;

System.out.println("This program took " + steps + " steps to run");
System.out.println("The following code runs in " + runTime + " nanosecond(s)");
System.out.println("The following code runs in " + runTime / 1000 + " microsecond(s)");
System.out.println("The following code runs in " + runTime / 1000000 + " millisecond(s)");
System.out.println("The following code runs in " + runTime / 1000000000 + " second(s)");

    public static void mergeSort(int[] a, int n) {
        if (n < 2) {
            steps++;
            return;
        }

        int mid = n / 2;
        int[] l = new int[mid];
        int[] r = new int[n - mid];

        for (int i = 0; i < mid; i++) {
            l[i] = a[i];
            steps++;
        }
        for (int i = mid; i < n; i++) {
            r[i - mid] = a[i];
            steps++;
        }
        mergeSort(l, mid);
        mergeSort(r, n - mid);

        merge(a, l, r, mid, n - mid);
    }
```

Insertion Sort using Linked Lists-



Programming_project_2 [C:\Users\dhava\Desktop\School\COMSC 340\Programming_project_2] - ...\src\Main.java [Programming_project_2] - IntelliJ IDEA

File Edit View Navigate Code Analyze Refactor Build Run Tools VCS Window Help
Programming_project_2 > src > Main
Project
Main.java
InsertionSortLinkedList.java
LinkedList.java
LinearNode.java
1: Project
Structure

```
Programming_project_2  C:\Users\dhava\Des
  .idea
  datafiles
  out
  src
    jsjf
      exceptions
        ElementNotFoundException
        EmptyCollectionException
      LinearNode
      LinkedList
      ListADT
    sort
      InsertionSortLinkedList
    InsertionSortLinkedList
    Main
  inorder5k.txt
  inorder10k.txt
  inorder100k.txt
  programming project 2 run.PNG
  Programming_project_2.iml
  Programming_project_2_tests.xlsx
  random5k.txt
  random10k.txt
  random100k.txt
  rev5k.txt
  rev10k.txt
  rev100k.txt
  ~$Programming_project_2_tests.xlsx
External Libraries
Scratches and Consoles
```

```java
import jsjf.LinkedList;

import java.io.File;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.util.Scanner;
import sort.InsertionSortLinkedList;

//import java.util.concurrent.TimeUnit;

public class Main {

    public static void main(String[] args) throws FileNotFoundException {
        File file = new File( pathname: "inorder10k.txt");

        LinkedList list = new LinkedList();

        Scanner sc = new Scanner(new FileInputStream(file));

        while (sc.hasNextInt()){
            list.add(sc.nextInt());
        }

        sc.close();

        long start_time = System.nanoTime();

        //System.out.println(list.toString());

        LinkedList sorted = InsertionSortLinkedList.insertionSort(list);

        long end_time = System.nanoTime();

        long timeElasped = end_time - start_time;

        //System.out.println(sorted.toString());


        System.out.println("Execution time in nanoseconds: "+ timeElasped);
        System.out.println("Execution time in milliseconds: "+ timeElasped/1000000.0);
        System.out.println("Steps: "+ InsertionSortLinkedList.steps);
    }
}
```

```java
package sort;

import jsjf.LinearNode;
import jsjf.LinkedList;

public class InsertionSortLinkedList {

    public static int steps = 0;

    public static LinkedList insertionSort(LinkedList list) {


        if (list.head == null || list.head.getNext() == null) {
            return list;
        }


        LinkedList newList = new LinkedList();
        LinearNode newListHead = new LinearNode(list.head.getElement());
        LinearNode current = list.head.getNext();

        while (current != null) {

            LinearNode key = newListHead;
            LinearNode next = current.getNext();

            if ((int)current.getElement() <= (int)newListHead.getElement()){
                LinearNode oldHead = newListHead;
                newListHead = current;
```

```java
                newListHead.setNext(oldHead);
                steps++;

            }
            else {
                while (key.getNext() != null){

                    if ((int) current.getElement() > (int) key.getElement() && (int) current
.getElement() <= (int) key.getNext().getElement()){
                        LinearNode oldNext = key.getNext();
                        key.setNext(current);
                        current.setNext(oldNext);
                        steps++;
                    }
                    steps++;


                    key = key.getNext();

                }

                if (key.getNext() == null && (int)current.getElement() > (int)key.getElement
()) {
                    key.setNext(current);
                    current.setNext(null);
                }

            }
            steps++;

            current = next;
        }

        newList.head = newListHead;
        return newList;
    }
}

```