Project 1: Simple Sorting

Madison Rose Boman

Spring 2017 – Patrick Kelley

Northern Arizona University

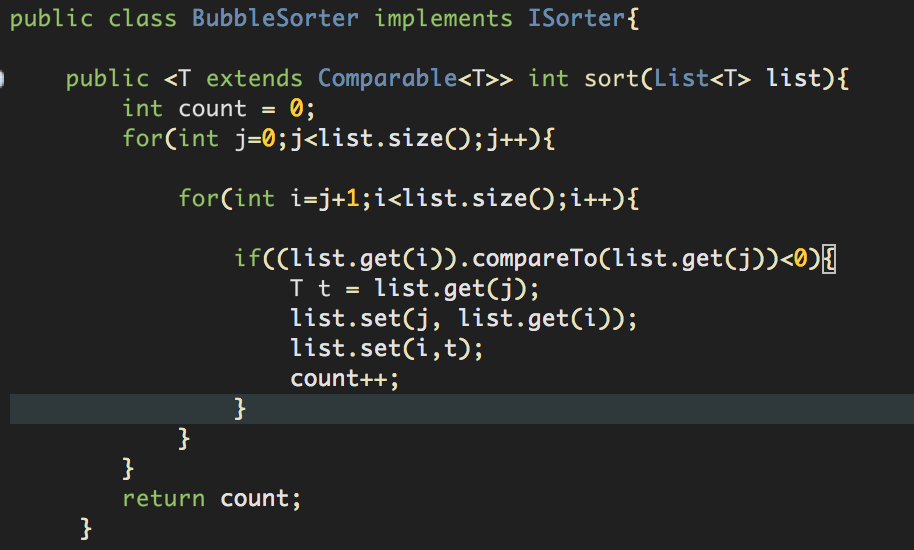
**Overview:**

This project will be consisting of using Generics and Interfaces to create files to perform various sorting tasks. The files created were classes implementing bubble sort, insertion sort, selection sort algorithms, as well as creating comparators for sorting a given library by the artist, what’s hot and new, and the year released. The 3 sort algorithms were also executed in comparison by the size of items that each must cycle through, and displaying the time results. When working on this project, I was in collaboration with Alex Mueller, Claudia Coronel, Brandon (unknown last name), and Tatum (unknown last name).

**Step1:**

My approach for all three of the sorting algorithms was similar, I knew that the Big O Notation for each was O(N2) so, there would consist of 1 type loop inside another type loop for its complexity. The bubble sort algorithm’s pseudocode was easiest out of the three to make. I had created was to for loop through a given ArrayList for variable (index) i equaling 0. Then we do another for loop the same for variable (index) j, but j is equal to one more than i. If index j was bigger than index i, we swap the two values i and j both contain, increase our swap count by 1, and move on to the next positions. If i is less than j, we just move on to the next 2 positions by incrementing i and j and, continue til there are no more values to compare. Next we did the Insertion sort. For every item i in an ArrayList starting at index 1, we created a key(i) for the value at the index i. We then created a variable j, that constitutes as the item previous of i. Looping through while j is greater than or equal to 0, and while j is greater than the key, we will swap the key(i) and j, increase the swap count, and reduce j by 1. Then we moved on to the selection sort, which was the most difficult of the 3. For every item in the ArrayList starting at index 0, we will by default, set the current spot, i, to be the minimum and, create a value at the minimum’s index. Then, for every item that is one ahead of our minimum, j, we will test to see if it is smaller than the minimum. If so the minimum is now j. If the minimum does not equal i, we will swap i, with the minimum and, increase our swap count. Otherwise we will move on to the next index i. The difficulties I had faced was really in creating the selection sort class. I kept swapping the value even when I was equal to the minimum (which was redundant). Because of that, for one of the tests I kept getting 6 swaps when it should have been two swaps. After we ran all our JUnit ComparableSortTests with the algorithms and pass, we can move onto step two.

Big O Notation Bubble Sort:

****

1 (constant)

N for loop

N for loop

1 (constant) conditional

1 (constant)

1 (constant)

1 (constant)

1 (constant)

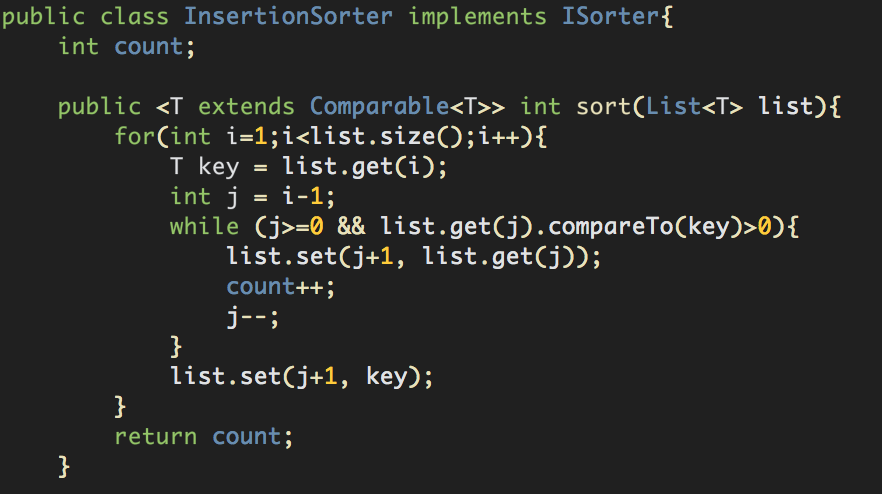
1 (constant) return

N\*N+5 + 2

=N2 +2

= O(N2)

Big O Notation Insertion Sort:

****

N for loop

1 (constant)

1 (constant)

N while loop

1 (constant)

1 (constant)

1 (constant)

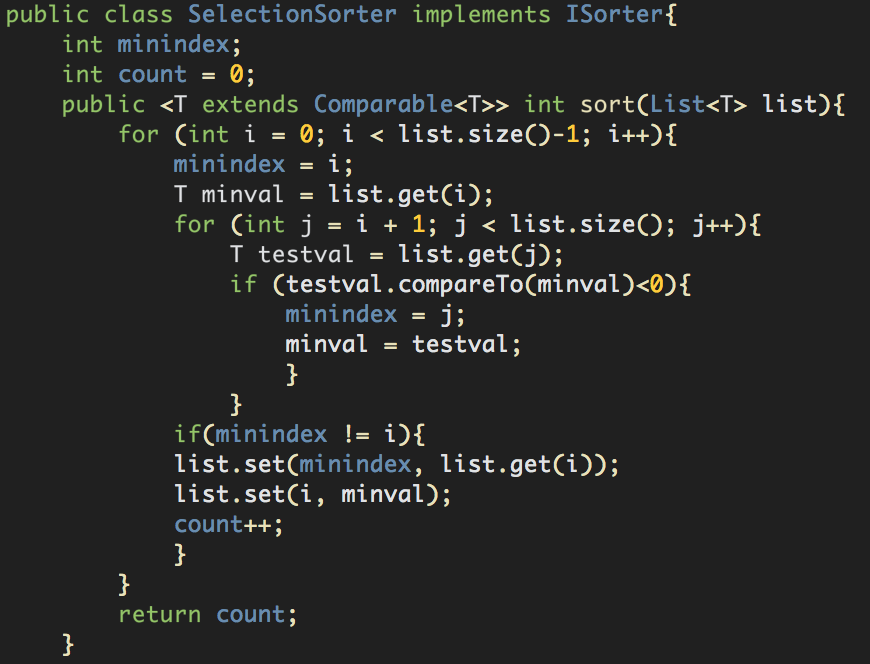
1 (constant)

1 (constant) return

=N\*N+2+3

= O(N2)

Big O Notation Selection Sort:

****

N for loop

1 (constant)

1 (constant)

N for loop

1 (constant)

1 (constant) conditional

1 (constant)

1 (constant)

1 (constant) conditional

1 (constant)

1 (constant)

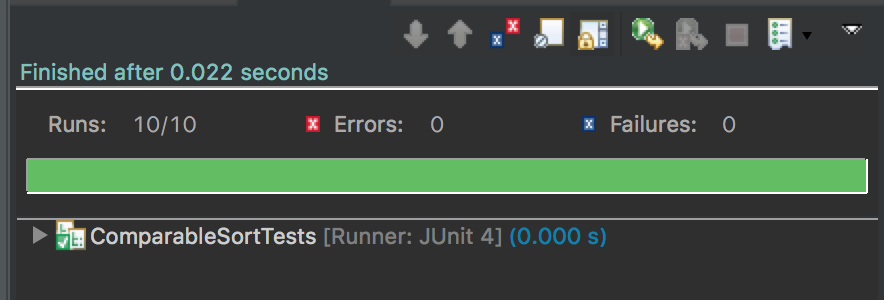
1 (constant)

1 (constant)

=N\*N+6+2

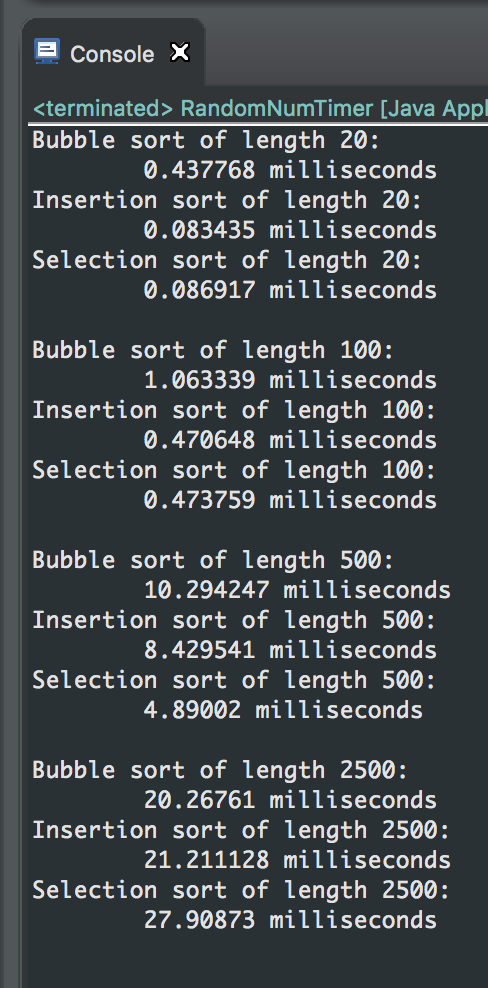
= O(N2)

Proof of ComparableSortTests Passing:

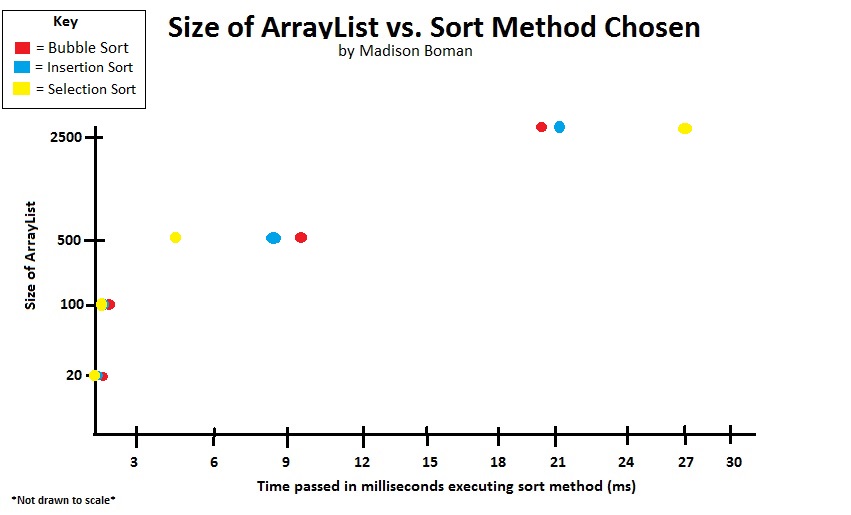
****

This is the screenshot proof of the ComparableSortTests passing using the bubble sort, insertions sort, and selection sort classes.

**Step2:**

 This next step was relatively easy, first we had to class with a main method that called the 3 sorting algorithms. In the main method, we created twelve ArrayList’s, that are of length 20, 100, 500, and 2500. There are three of each length because we will be evaluating the time of how long each method takes to sort the various sizes. In the picture to the right you see the results in milliseconds of the varying lengths in the ArrayLists. After we collected out data, we then took the times and plotted them on a scatterplot below in comparison of ArrayList size versus the sort method that was ran. You can see that for the smaller sized ArrayLists, the Insertion and Selection sort methods were neck and neck, when it came to speed, and not so much the Bubble sort. However, as we had gotten further up in the 2500 ArrayList size, the bubble sort had executed faster than the other two algorithms. Showing that the algorithms vary for data size.

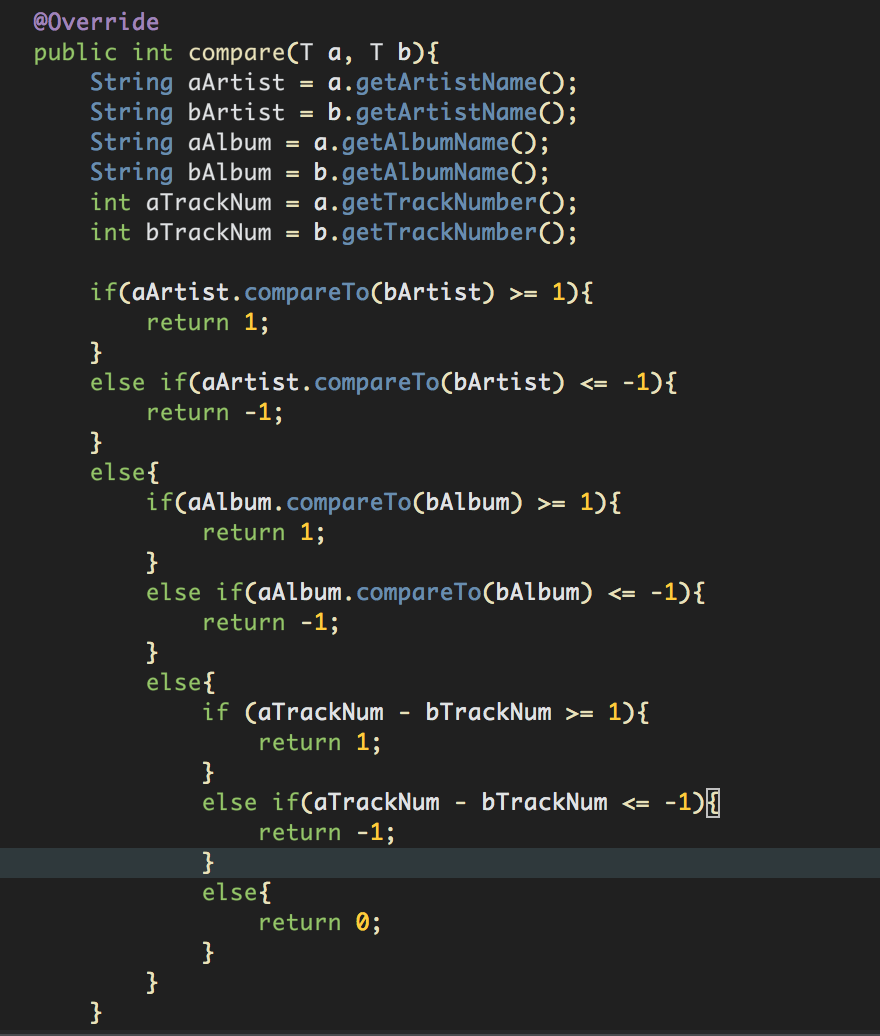
Scatter Plot Analysis of Size of ArrayList Vs. Sort Method



**Step 3:**

In this step, we create three comparator classes that will compare numerous MusicTrack’s given various criteria. In these classes, we will be overriding the compare() method from Java’s built-in Comparator Interface. Our first class is the ArtistComparator class. In this class we first will compare if the given artist A alphabetically comes before artist B. If not, we check if artist B comes before artist A alphabetically. If they are the same we then check the same criteria for their album names. If both album names are the same, we then check the track number from the album, and whatever came chronologically first from that album, we then sort that one as the first. Then we did the ChronologicalComparator class the same way, except we start off comparing the year the track was released in ascending order. If the year is the same we then compare the albums name like in the previous class. If the album is the same, we then compare the track number in ascending order. Lastly we have the HotAndNew Comparator class. In this class we compare the number of rating (one to five stars) in descending order. If tracks have the same rating, we compare the years released in descending order. If they have the same year, we compare the artist name alphabetically. If the artist is the same artist name we compare the album name alphabetically. If both album names are the same, we lastly compare the track number in ascending order and, sort likewise. As a mouthful that all could have been, it was found to be the simpler step out of the total 5 steps.

Big O Notation of ArtistComparator:



1 (constant)

1 (constant)

1 (constant)

1 (constant)

1 (constant)

1 (constant)

1 (constant)

1 (constant)

1 (constant)

1 (constant)

1 (constant)

1 (constant)

1 (constant)

1 (constant)

1 (constant)

1 (constant)

1 (constant)

1 (constant)

1 (constant)

1 (constant)

1 (constant)

1 (constant)

O(0) since all execution time is constant.

Big O Notation of ChronologicalComparator:

****

1 (constant)

1 (constant)

1 (constant)

1 (constant)

1 (constant)

1 (constant)

1 (constant)

1 (constant)

1 (constant)

1 (constant)

1 (constant)

1 (constant)

1 (constant)

1 (constant)

1 (constant)

1 (constant)

1 (constant)

1 (constant)

1 (constant)

1 (constant)

1 (constant)

1 (constant)

O(0) since all execution time is constant.

Big O Notation of HotAndNewComparator:



1 (constant)

1 (constant)

1 (constant)

1 (constant)

1 (constant)

1 (constant)

1 (constant)

1 (constant)

1 (constant)

1 (constant)

1 (constant)

1 (constant)

1 (constant)

1 (constant)

1 (constant)

1 (constant)

1 (constant)

1 (constant)

1 (constant)

1 (constant)

1 (constant)

1 (constant)

1 (constant)

1 (constant)

1 (constant)

1 (constant)

1 (constant)

1 (constant)

1 (constant)

1 (constant)

1 (constant)

1 (constant)

1 (constant)

1 (constant)

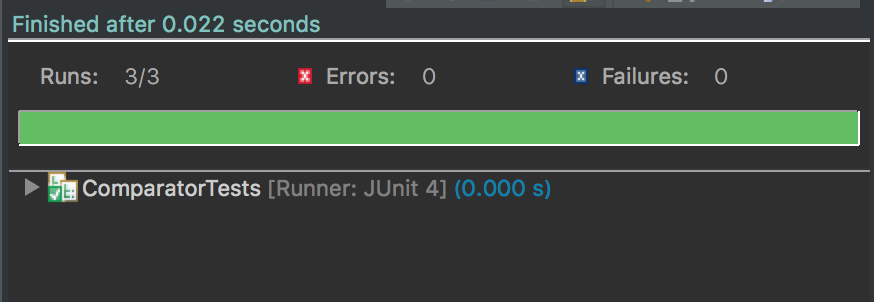
1 (constant)

1 (constant)

O(0) since all execution time is constant.

Proof of ComparatorTests Passing:

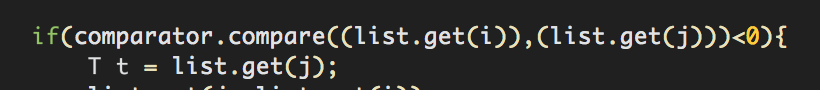
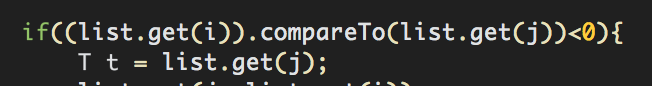
After we had finished writing our code for the 3 comparator classes, we then ran our ComparatorTest class with Junit until our tests all passed.



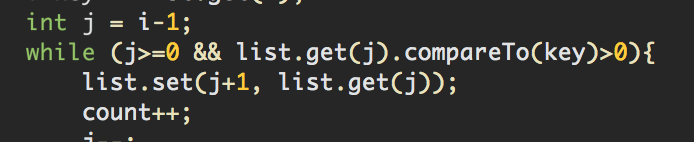
**Step 4:**

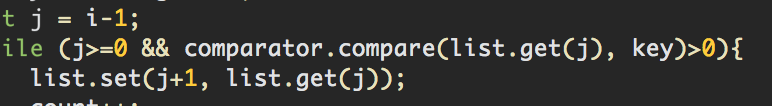
In this step, we go back to our Bubble, Insertion, and Selection sort classes and add another method. The method is going to do the exact same sort as the method above it except, this one we will be passing in a comparator object along with an ArrayList, and we will be changing our .compareTo() line in the original to be comparator.compare()

Bubble sort:

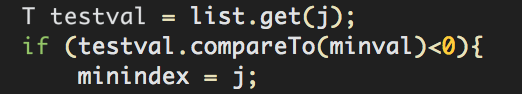


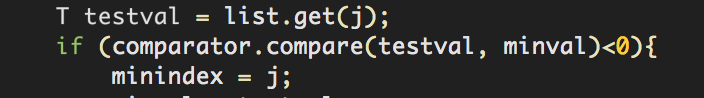
Insertion sort:





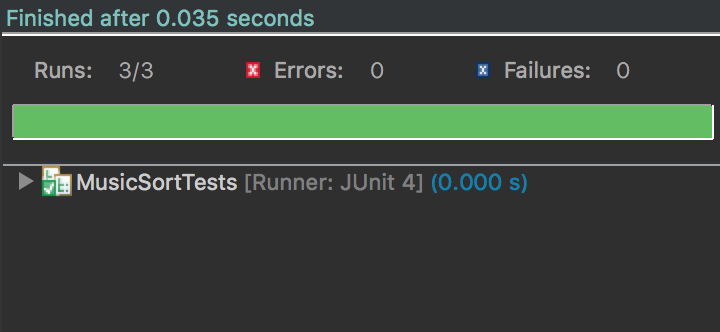
Selection sort:





Proof of Music Sort Tests Passing

After the methods have been created we then run our MusicSortTests on the just previously made methods in our sort classes. We run the JUnit tests until they all pass.

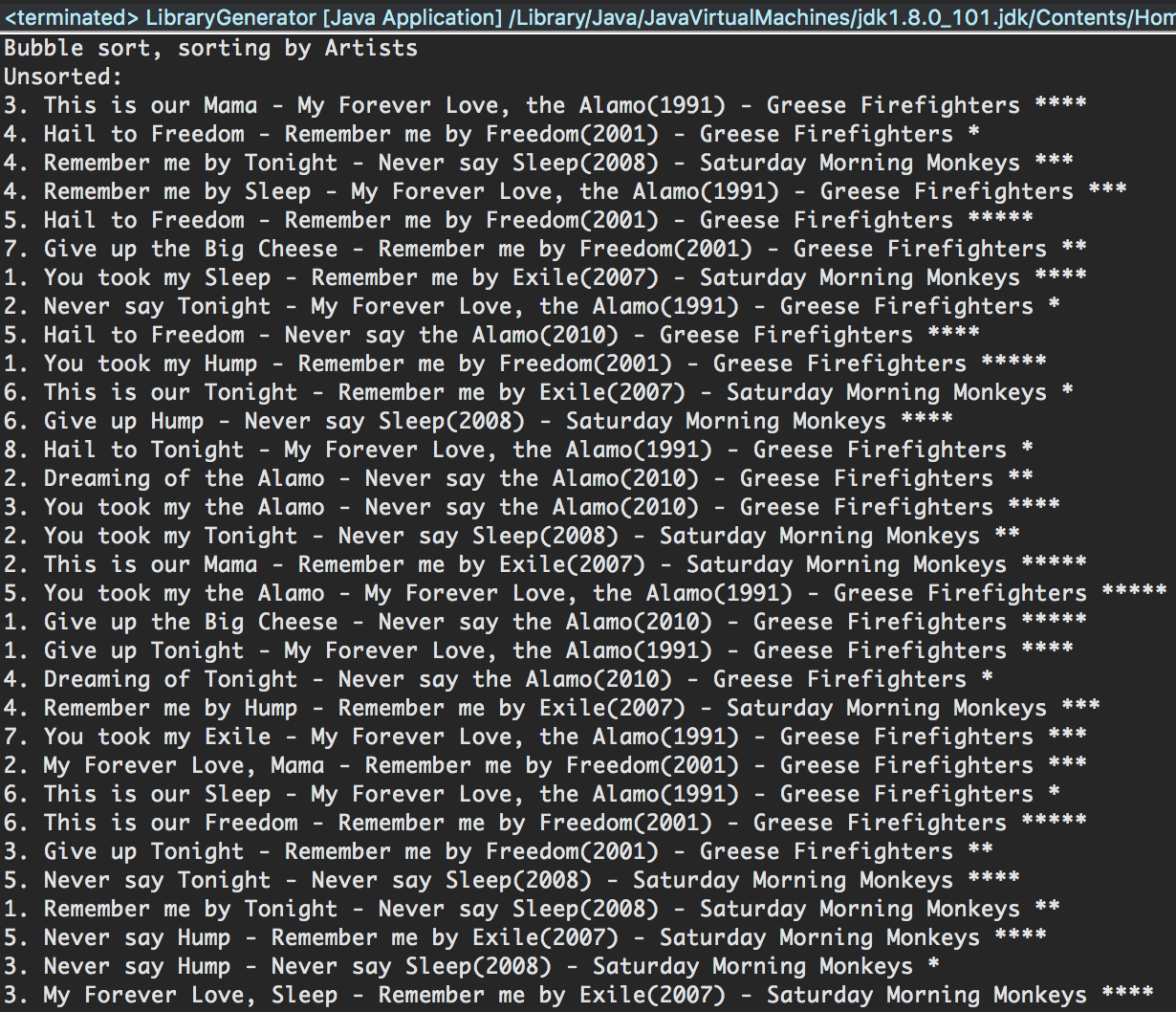


**Step 5:**

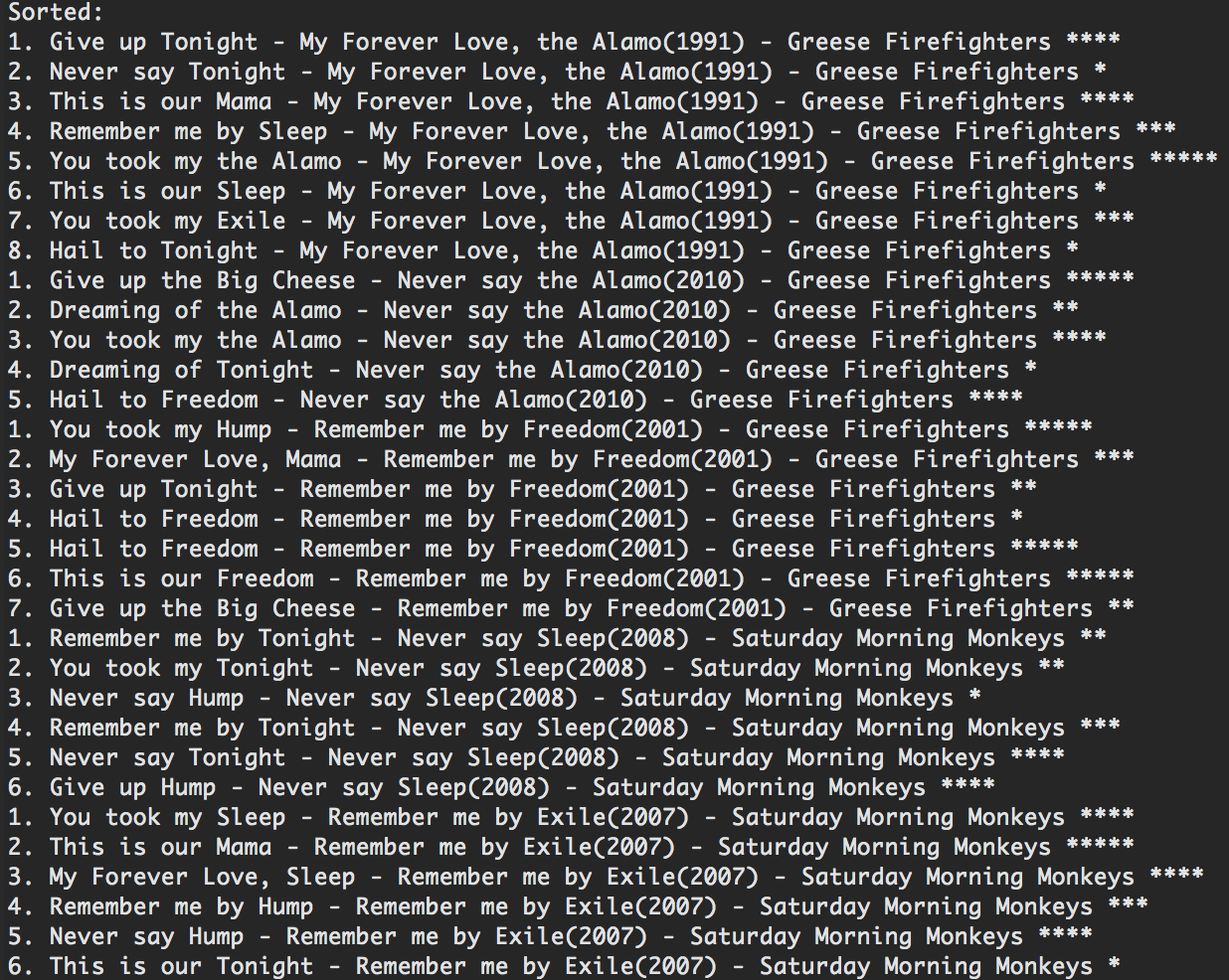
The fifth and last step of the project is to create a main method that generates a random library, then perform one sort algorithm with one comparator. We do this three times, with all three sorts and, camparators. In my file, named LibraryGenerator, I call the method LibraryGenerator passing the randomly generated library we created. In the LibraryGenerator method, I use a for loop to display all contents in the library. Then I sort the library by calling the algorithm sort with the .sort() method passing in our library, and the comparator we are using. In this step we got to pick what algorithm we sort with what comparator. My results are:

Bubble Sort Sorting by Artists

UNSORTED

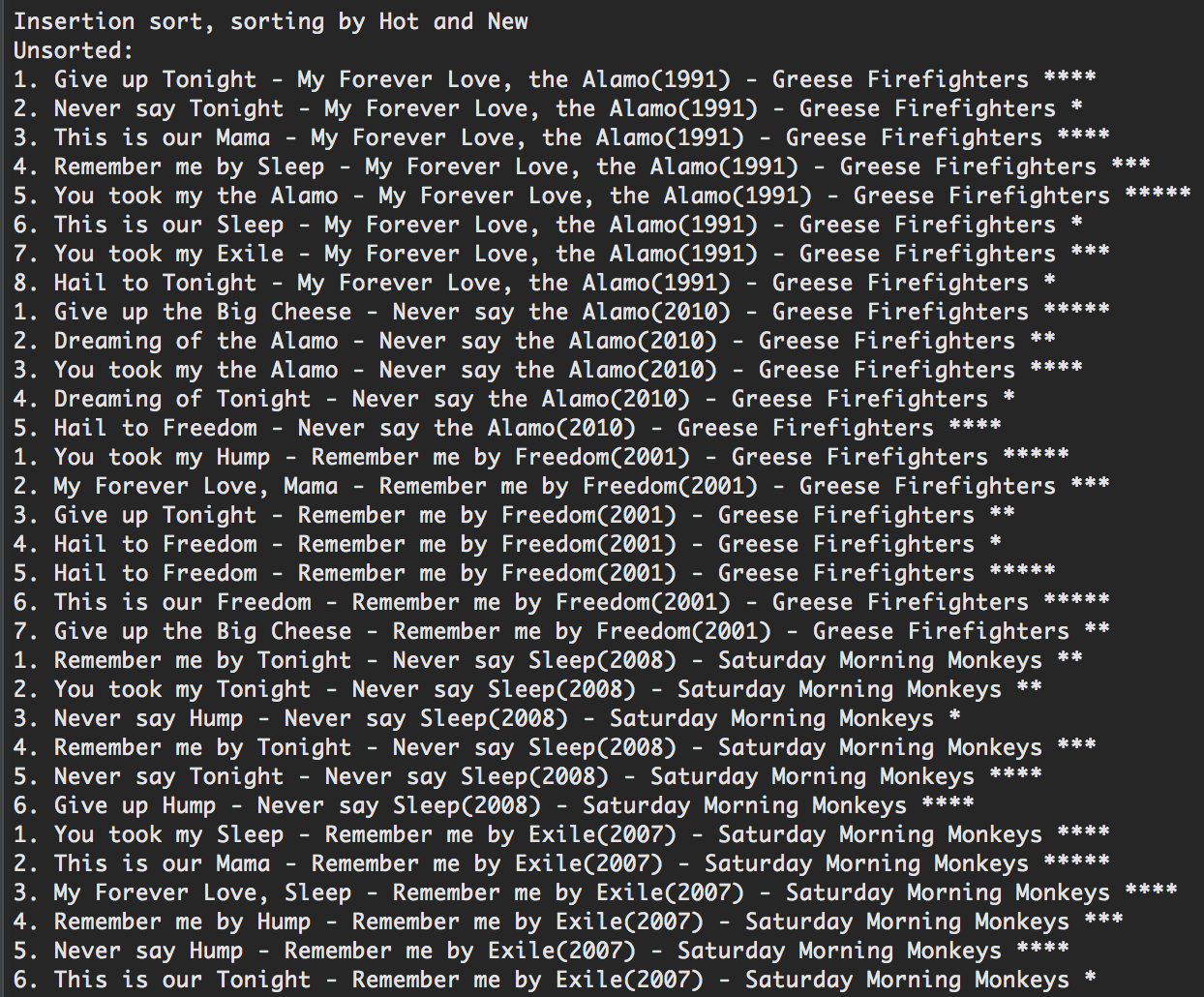
****

SORTED

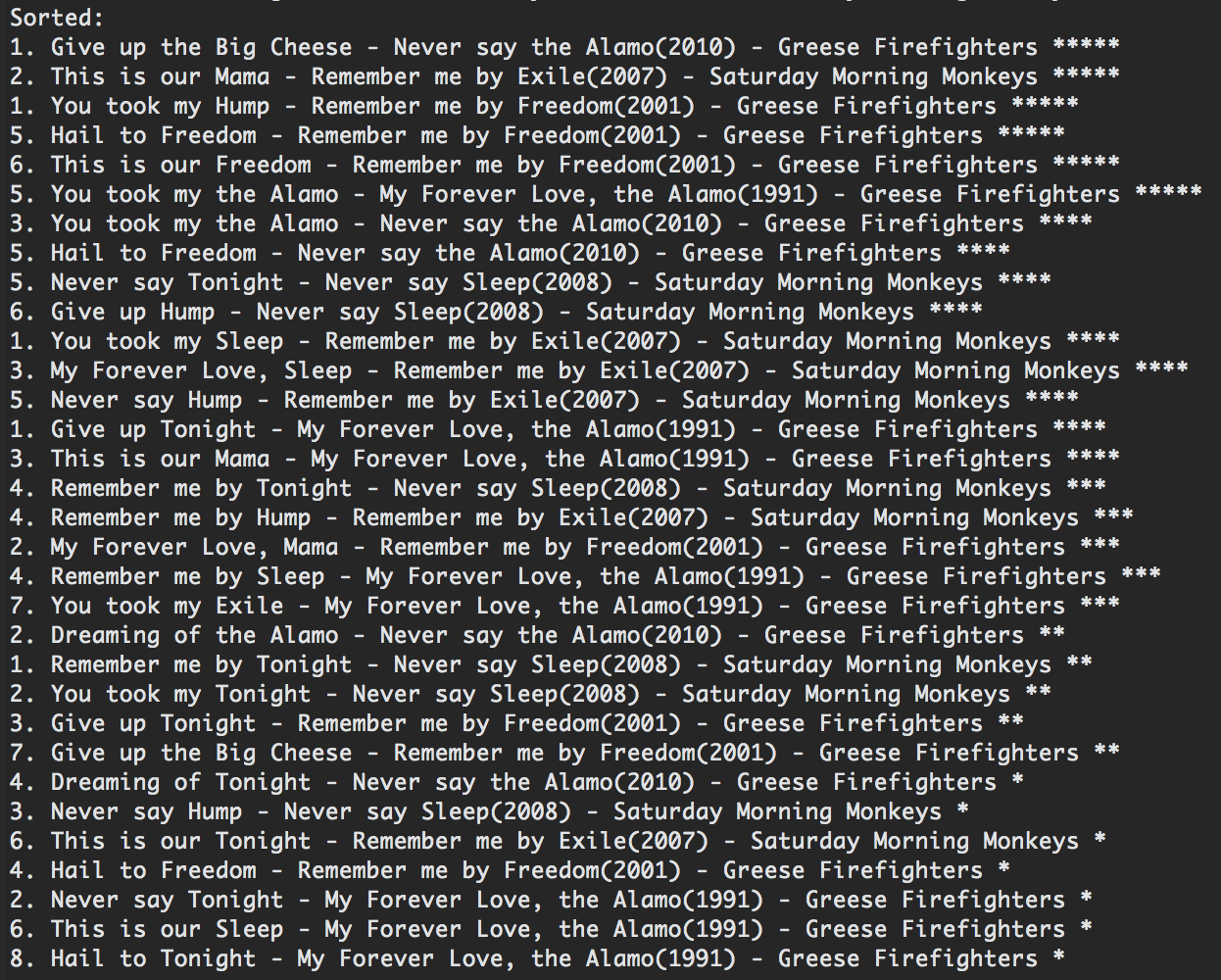
****

Insertion Sort Sorting by Hot and New

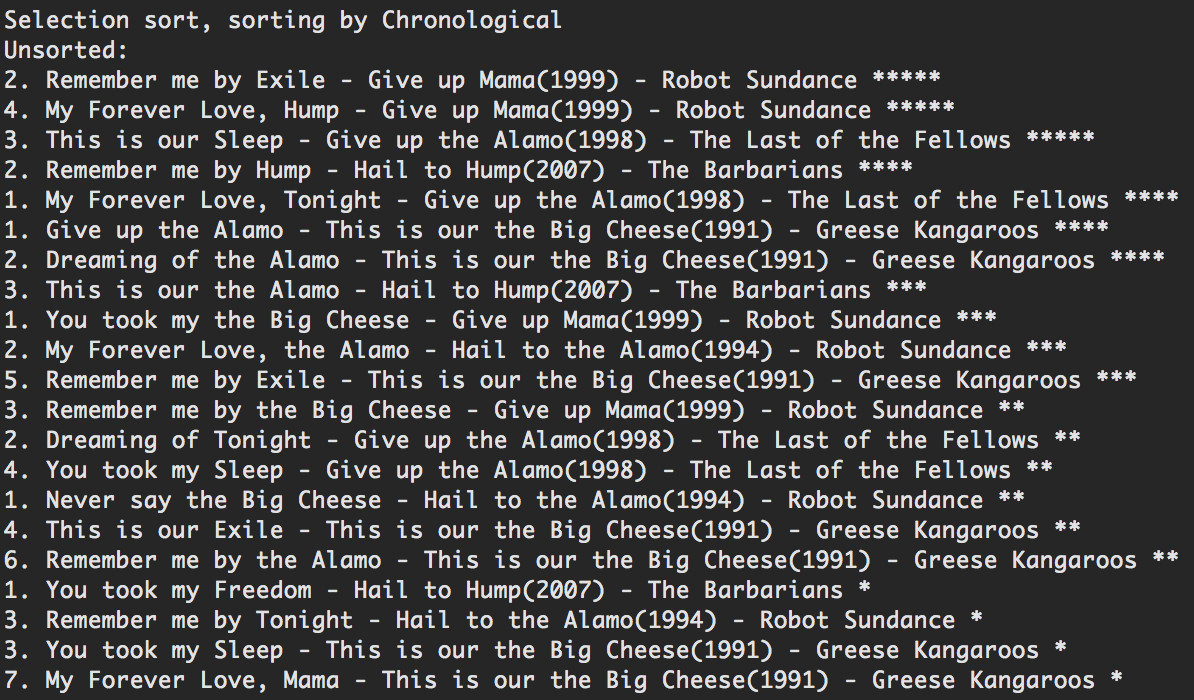
UNSORTED

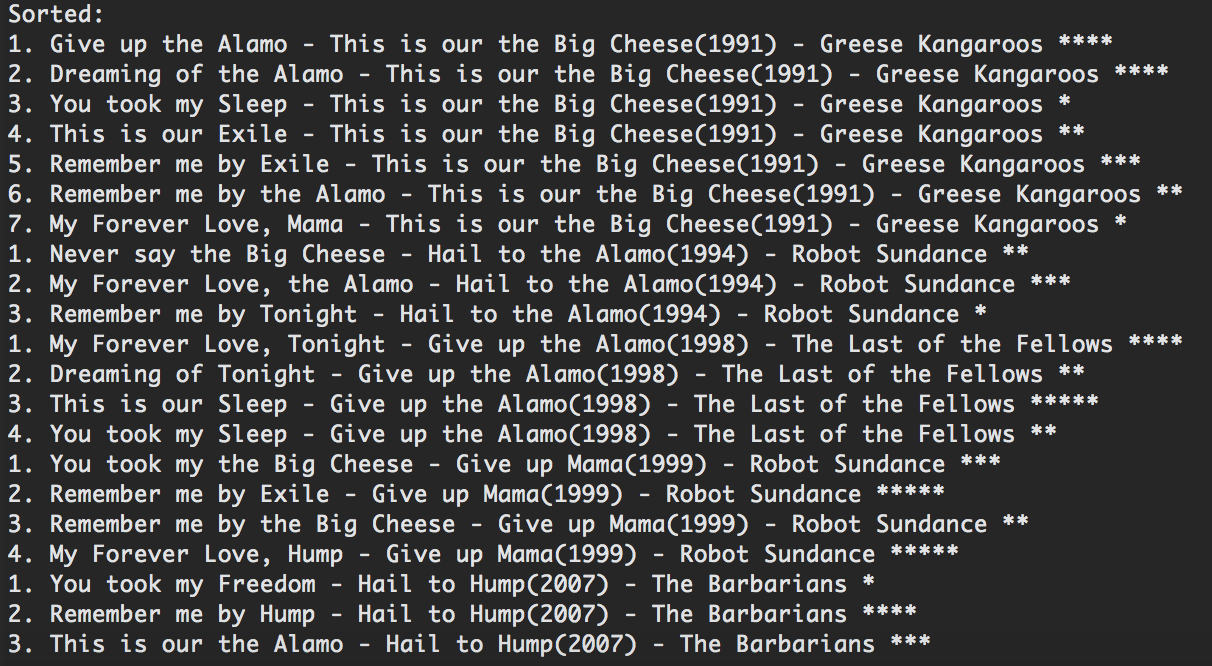


SORTED

****

Selection Sort Sorting by Chronological year

UNSORTED****

SORTED****

**Conclusion:**

Overall this project was much more tedious and really required thinking outside the box. Like the tasks we had to do in step 4. Originally I had started rewriting my bubble sort method for that specific method, when all I had to do was change what I was comparing and, the built in method being used. I really did enjoy this project overall however. It allowed me to see that various sort algorthims do better in certain environments and that could be applied in a large scale of things. It shows that is I’m dealing with a small amount of data, I’m okay to do an insertion sort. However if Im using larger amounts of data, a bubble sort could be better in the long run. To follow that, I also learned a lot about comparator’s as well as really getting the hang of JUnit. In my previous computer science class, CS136, I struggled heavily with JUnit. Using the comparator classes was interesting to see the sorts applied with the comparator. It makes you wonder what the music libraries (AppleMusic, Spotify, Pandora, etc.) in our daily life constitue of behind our phone screens.