Evan Parry

Professor Daniel Kopta

CS 2420-001

September 13, 2018

Assignment 3 Analysis Document

1. Which aspect of this assignment was the most difficult for you?

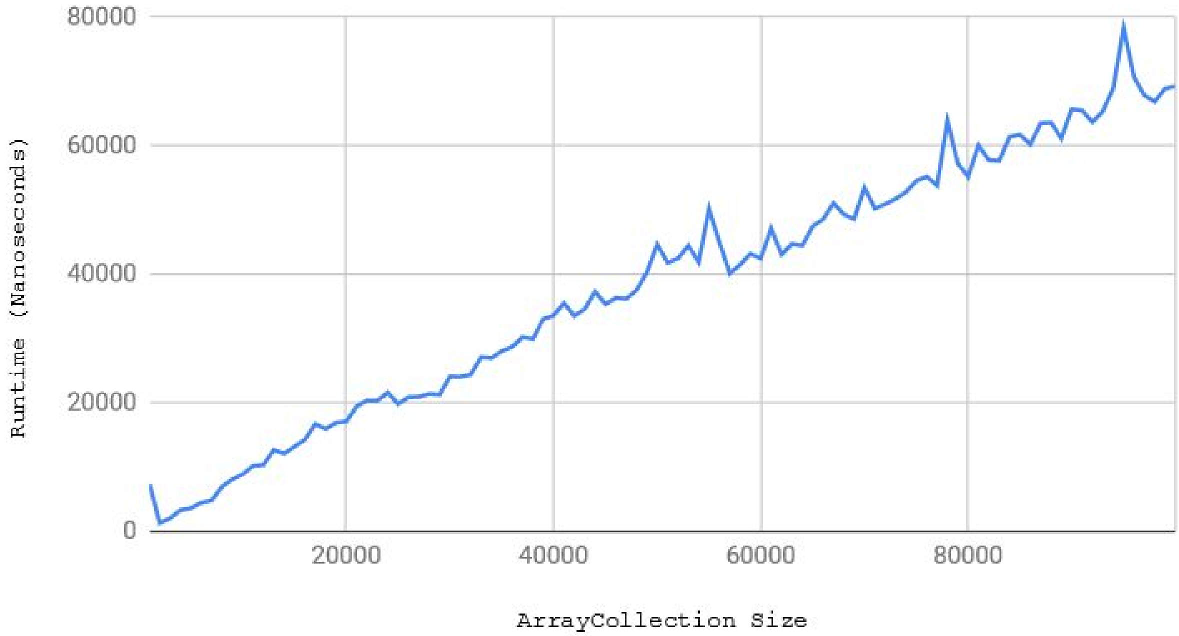
Getting our binary search to work properly was the most difficult, which we later learned was because we'd made it more difficult and complicated than it really needed to be. We did manage to get it fixed in time, however.

It was also difficult to think of some edge cases to test for, though many of them quickly became apparent through experimental testing.

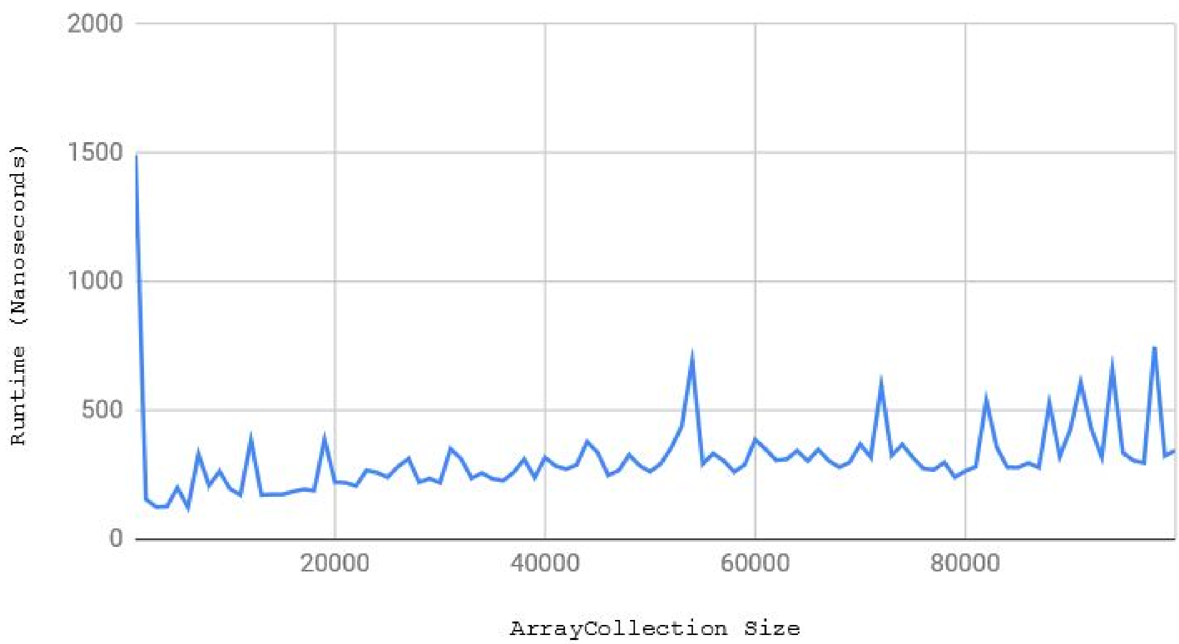
2. Under the Week 3 module, download TimeArrayCollection.java, which is a starting point for your timing code, and contains a method for generating a random Integer. Also download IntegerComparator.java, which implements a comparator on Integers (used for toSortedList and binarySearch). Add both of these files to the assignment3 package. Use the random integer generator to fill up your ArrayCollection with random values for testing. Plot the runtime performance of your toSortedList method for varying problem sizes N (number of items in the collection).

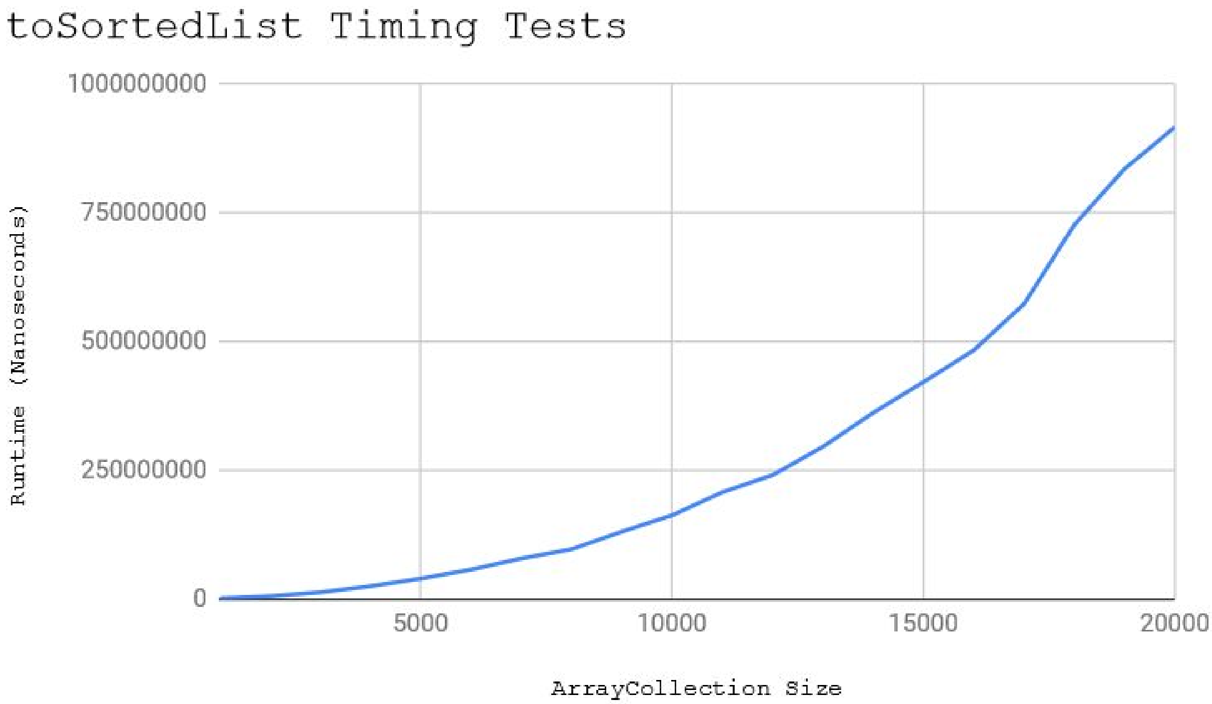
See next page for charts.

contains Timing Tests

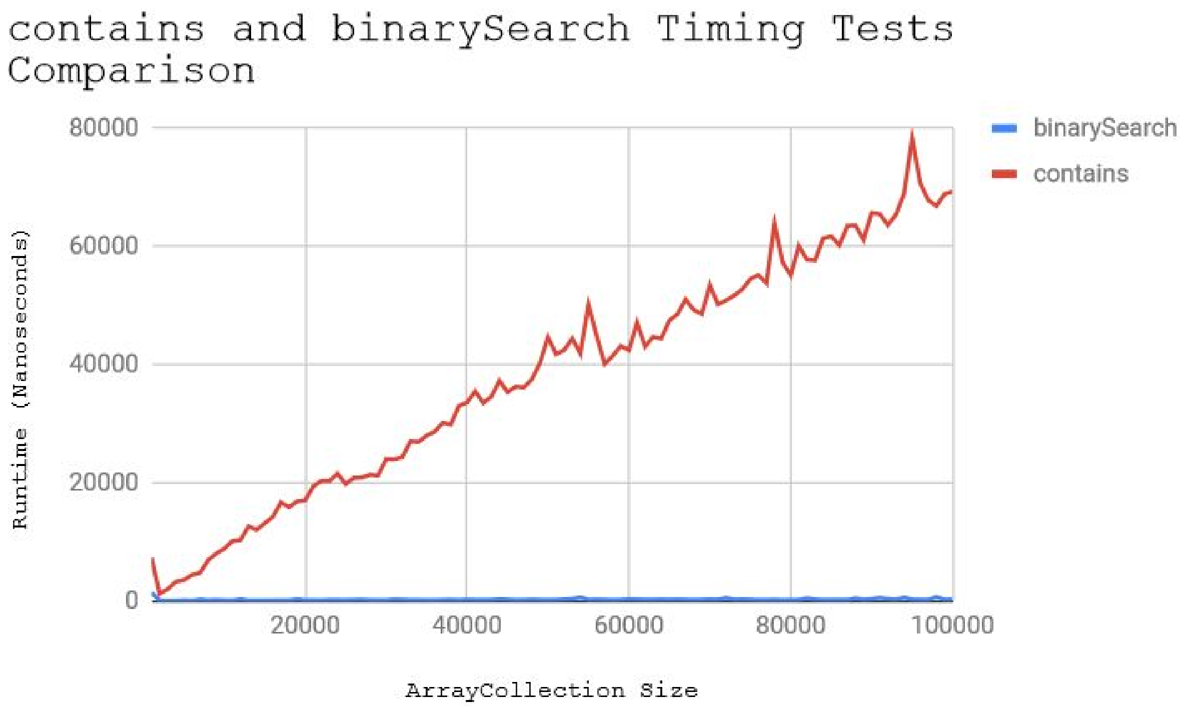


binarySearch Timing Tests




toSortedList Timing Tests
Runtime (Nanoseconds)
1000000000
750000000
500000000
250000000
0
5000
10000
15000
20000
ArrayCollection Size

3. Plot the performance of your contains method vs the performance of your SearchUtil.binarySearch method. Use the same guidelines and input sizes as in part 2. Use a random number as the input item to search for.
    - timesToLoop can (and should) be quite a bit higher for these two methods
    - For each iteration through timesToLoop, search for a different random number. Make sure you generate and store these numbers ahead of time, so that random number generation is not included in your timing results (save them in an array, use a different one on each iteration).



This comparison chart is almost exactly what I expected - contains is very clearly linear, while binarySearch is logarithmic and is barely visible at this scale.

4. Study the code for selection sort (your toSortedList method). What is the expected Big-O complexity? Does your plot for part 2 support your expectations? Why or why not?

The expected Big-O complexity for an average case is O(N^2). The plot does support this expectation, rising exponentially as the size of the ArrayCollection increases.

5. What is the Big-O complexity for your SearchUtil.binarySearch method? Does your plot confirm this?

The expected Big-O complexity of a binarySearch method is O(Log N), which is confirmed by the plot. As shown in the test plot, while the binarySearch runtime did increase as the ArrayCollection's size increased, it did so at a very slow rate.

6. What is the best, average, and worst case performance for contains? Do your timing experiments here have a bias towards the best or worst case? (Hint: Think about the probability of an ArrayCollection of 20,000 integers containing any random integer. There are about 2^31 possible random integer values.)

The best case performance for contains is O(1), which will only happen when the value contains is searching for is at the very first index in the ArrayCollection. The average and worst case scenarios are both N, as contains has to search the entire ArrayCollection before confirming that a given item is not in fact present in the ArrayCollection.

Our experiments have a bias towards the worst case. There's a very high chance that the value contains is searching for won't be in the ArrayCollection at all, forcing it to search through the entire ArrayCollection before it finally returns false. Similarly, if the sought after value is in the array, it has a very low chance of being within the first few indices that contains checks, increasing the average time before contains returns true.