Lies, Damn Lies, and Algorithm Analysis

I was recently asked to code a solution for finding overlapping intervals on the integer number line during an interview. While thinking further about the problem after the interview, a number of seeming similar solutions occurred to me, including the one I proposed. Why would I choose one algorithm over the another? There are certainly "style" issues that lead engineers to select one coding approach over another, just as Mathematicians might favor one proof over another. All other things being equal, many of us then resort to selecting an approach based on performance over the range of interest. This got me to thinking . . . how should I quantify "performance"? ***This could be interesting!*** Let’s look at algorithm performance analysis.

Algorithm performance is often studied with simplified mathematical models. One popular performance analysis approach is called "Big O". "Big O" analysis describes asymptotic execution time as a function of input data set size, and can be misleading. The key comparison power of "Big O" notation is expressed in this definition:

*Let O(f(x)) = h(x), and O(g(x)) = h(x); that is, functions f(x) and g(x) are of the same "Big O" order, then:*

*execution\_time(f(x)) / execution\_time(g(x)) <= K*

The ratio of the execution time of two "Big O" equivalent functions has an upper bound, but the execution times are *not necessarily equal! This could lead to trouble*. Additionally, the analysis only examines performance as the input data grows without bound.

Prime Minister Benjamin Disraeli is credited with the quote: "Lies, Damn lies, and Statistics", where he expressed a commonly felt frustration with that branch of Mathematics. Much like Statistics, algorithm analysis makes simplifications about algorithm operation and execution environment to gain insight into performance. Unfortunately, having an understanding of mathematical algorithm performance is not always sufficient to understand actual performance in deployed environments. Let's look at an actual example to see the differences between the theoretical performance and actual performance.

## Algorithm Performance Analysis Example

Let's use our interview problem to explore these concepts. The interview problem was stated as:

*Given a collection of intervals (inclusive), merge all overlapping intervals. Example:*

*Input [[8,10], [1,4], [3,6], [15,18]].*

*Return [[8,10], [1,6], [15,18]].*

Later I discovered that this problem originally came from the educational site Leet Code (see <https://leetcode.com/problems/merge-intervals/description/>). There were two broad solution categories I considered: brute-force search and pre-sorting the intervals to be merged. The brute-force approach would require comparing a candidate interval to a collection of partially merged candidates to see if a merge were possible. An excellent overview of the two broad approaches and a "Big O" analysis of both are found at Leet Code using the above link. We will compare theoretical and actual performance using this problem and its alternative solutions.

My interest was in the faster, and to my mind simpler, sort and merge algorithm. Let us define an interval as a pair of integers [**start**, **end**], with **start** <= **end**. This is the algorithm:

1. Sort the list of intervals into ascending order by *start* and then *end*.
2. Proceeding left to right, pairwise compare two adjacent intervals (with *LHS* followed by *RHS*);
3. if *RHS* is not completely to the right of *LHS* on the number line, then replace *RHS* with the merge of *LHS* and *RHS*, and remove *LHS* from the list.
4. Repeat step 2 until the entire list is merged.

Using the example from above:

* [[8,10], [1,4], [3,6], [15,18]] =(sort)=> [***[1,4], [3,6]***, [8,10], [15,18]] =(merge)=> [**[1,6]**, [8,10], [15,18]]

From a performance perspective, a tricky part of the algorithm is step 3, removing the unneeded and now merged interval from the list. Here is the point where some actual Java code is helpful to review. A GitHub repository with the code associated with this problem is found at <https://github.com/DonaldET/DemoDev/tree/master/dev-topics-codingexams/dev-topics-amazon-rangeconsolidator>. The classic Computer Science solution to the list removal problem is to use a linked list. A linked list removal time is fast (O(1)) when compared to an array (O(n)).

However, I was determined to use the simpler Java *ArrayList* structure. I believed clients would find the merged output more useful as an ArrayList than a *LinkedList. ArrayList* has a problem; the *ArrayList* must copy the remaining elements beyond the removal point into the correct position in that array. A *LinkedList* just manipulates pointers and avoids data movement. To mitigate the removal problem, I first imagined merging from right-to-left instead of left-to-right, thus avoiding moving elements to the left. This approach failed on some inputs (see below for a failure example.) We can now set up some actual tests of the alternative implementations.

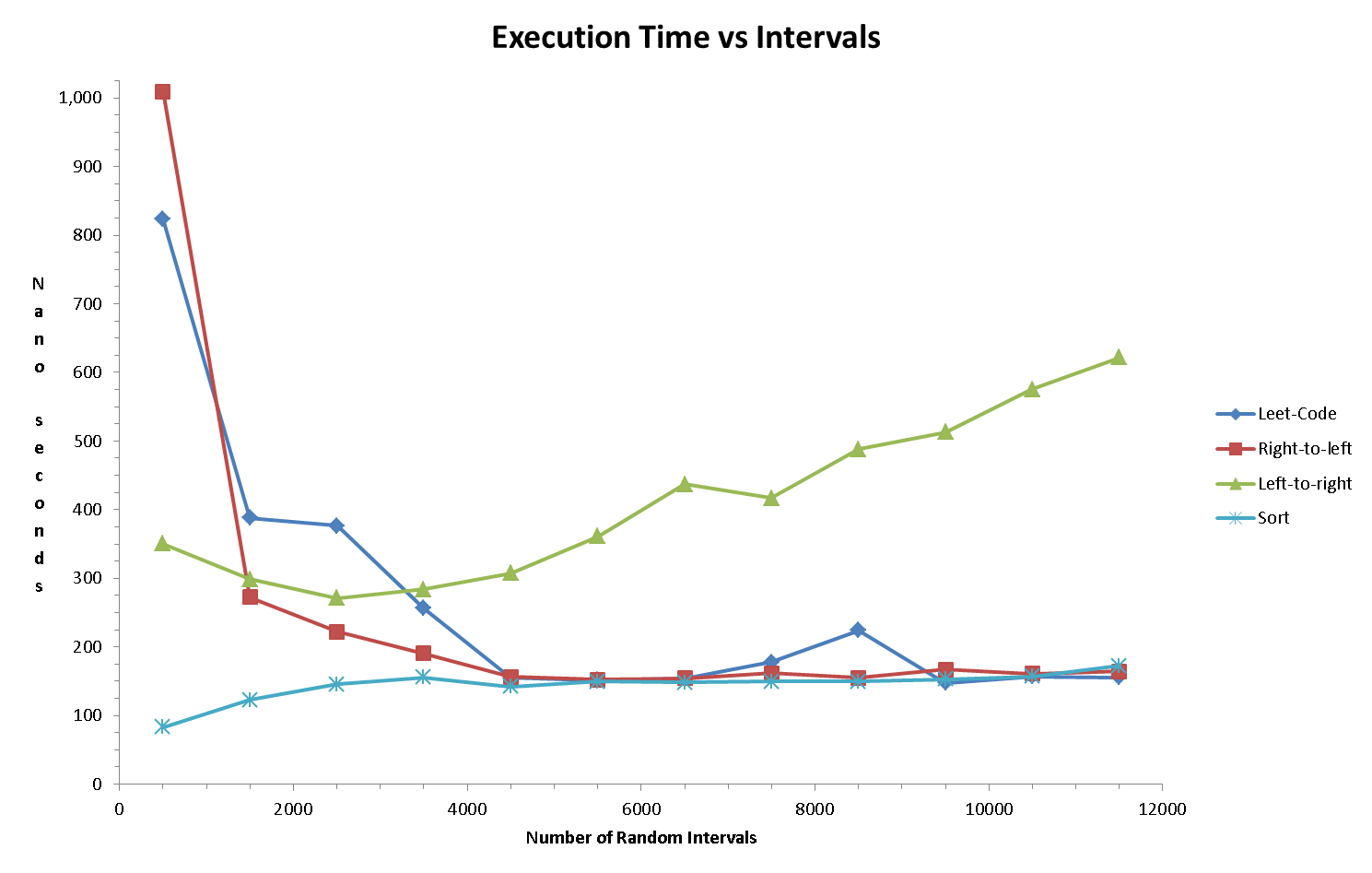
The trick I used to fix this *ArrayList* problem was to replace the merged and to-be-removed entry with a Java **null** value. At the end of the algorithm I copied out the *non-null* values from the merged list. You can view these Java classes from the source repository above:

* Leet-Code: *LinkedList* - OverlapLeetCode.java
* Right-to-left: *ArrayList* with **null** for delete - OverlapR2L.java
* Left-to-right: simple *ArrayList* - OverlapL2R.java

Now the fun part is to compare the actual performance in a meaningful range of interest.

## Algorithm Performance Comparison

Merging of a few thousand random candidate intervals, the range of interest, are performance tested and graphed below. Test machine specifications and testing code are found in the repository link mentioned above. In spite of all three algorithms having O(n\*log(n)) performance, we can see considerable differences in actual performance.

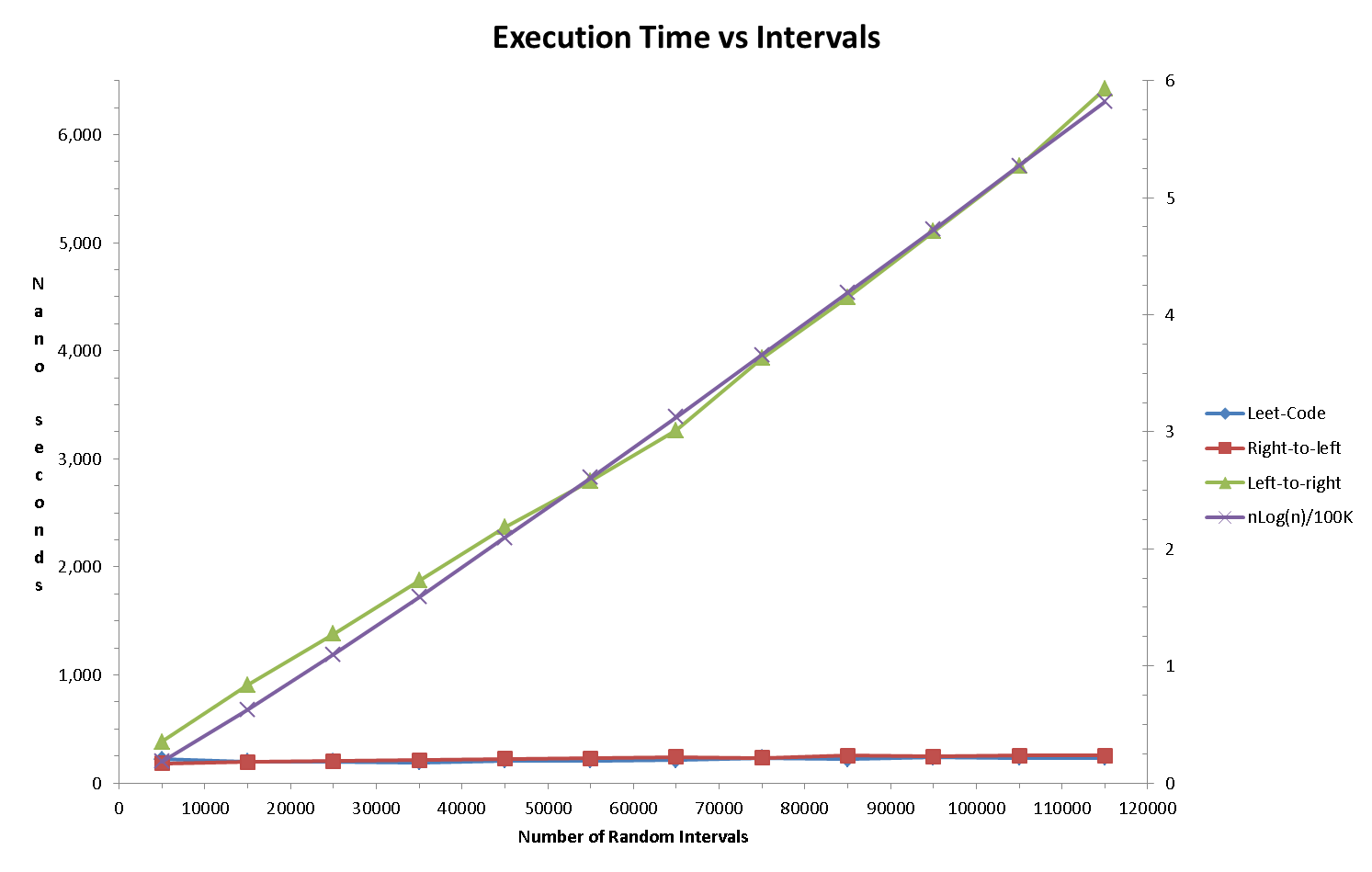


These observations are in order:

* For the Leet Code (*LinkedList*) and Right-to-left (*ArrayList*) implementations, the SORT portion of the algorithm dominates execution time as expected.
* The Right-to-left (*ArrayList*) algorithm is somewhat faster than the Leet Code (*LinkedList*) algorithm in the range of interest.
* The Left-to-right (simple *ArrayList*) is significantly slower than the other two algorithms.

The smaller intervals have more uncertainty in measurement and cause initial extreme values. We can see from the algorithm differences that there are major performance impacts introduced by the run-time system (e.g. memory management), making the formal mathematical analysis inaccurate in some cases. The mathematical analysis is only valid as n grows large.

We next expand the input data size by a factor of 10 to view asymptotic behavior below. We note that the Left-to-right (simple *ArrayList*) still has an *n\*log(n)* performance profile, but it is significantly slower than the other two algorithms, and performance degrades more rapidly.



My friend Hamid Montazeri, an excellent engineer, related the following joke that summarizes the care an engineer should take in constructing software:

*A chemist, a physicist, and a mathematician meet in a bar. The chemist announces that he has found an equation that yields primes: f(x) = −x^3/6 + 3x^2/2 − 7x/3 + 3.*

*Our chemist notes f(1) = 2, f(2) = 3, and f(3) = 5.*

*The physicist examines the equation, adds that f(4) = 7, and states that the equation produces primes for the range of interest.*

*The mathematician observes that f(5) =8, which is not a prime, and therefore the chemist’s conjecture is false.*

The mathematician is perhaps too extreme, and the chemist a bit careless. Let's strive to match the physicist. That is, we must verify performance over the range of interest in addition to mathematical analysis.

## Summary

Performance is not always the primary consideration in choosing an algorithm, as illustrated in the *LinkedList* verses *ArrayList* implementations. When we do care about performance, we see that we must run concrete performance tests in a deployment situation to validate our mathematical performance analysis. The references in the links above are good examples of both mathematical analysis and Java testing.

A similar algorithm performance analysis article, written by this author, is at URL <https://www.linkedin.com/pulse/test-driven-development-tdd-really-works-donald-trummell-1c/>, and includes a passionate case for adapting TDD.

----------------------------------------------------------------------------------------------------------

## Engineering Notes for the Interested

The test environment uses a nano-second timer and repeats executions a number of times so sufficient time is expended making measurement more repeatable. The test code has this parameterized to allow adjustment for different machines. Data printout may be suppressed, and long running tests ignored, making it possible to use a limited version of performance testing in the regular production build. I recommend this kind of "smoke test" because I have had to debug code changes that did not impact functionality but did impact performance.

## Algorithm Operation for Left-to-right and Right-to-left, Input and Sorted:

**Input**: [4217, 9573], [6729, 7161], [3402, 7649], [7743, 9454], [2706, 8358]

**Sorted**: [2706, 8358], [3402, 7649], [4217, 9573], [6729, 7161], [7743, 9454]

**Right-to-left erge:**

[2706, 8358], [3402, 7649], [4217, 9573], [6729, 7161], [7743, 9454]

[2706, 8358], [3402, 7649], [4217, 9573], [7743, 9454]

[2706, 8358], [3402, 9573], [7743, 9454]

[2706, 9573], [7743, 9454]

**Left-to-right merge:**

[2706, 8358], [3402, 7649], [4217, 9573], [6729, 7161], [7743, 9454]

[2706, 8358], [4217, 9573], [6729, 7161], [7743, 9454]

[2706, 9573], [6729, 7161], [7743, 9454]

[2706, 9573], [7743, 9454]

[2706, 9573]

------------------------