Complexity in action

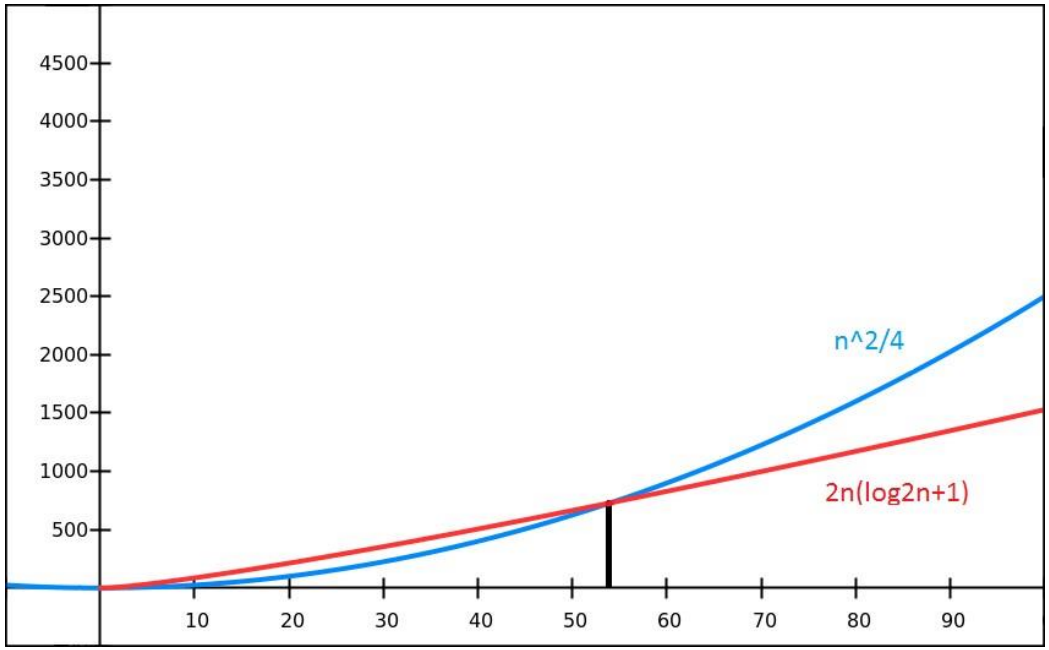
Design

I considered dividing the code in classes, but then the program would have been slower as I would need either to create a new instance of a sort object in every test or have a single dummy class at the beginning. I also thought about moving the sorts to other classes and leave the methods static, but I decided that the code isn't so complicated and put everything in a single class.

Mathematical analysis

On average, there would be around n/4 swaps when sorting a single element with insertion sort. Roughly half-way along the list and we'll need to go roughly half way back. This applies to every element, so the runtime of the algorithm is n^2/4.

Merge sort use a recursion, where the partitioning takes n time, because we need to copy all the elements of the array to two lists. Then the merging also takes n time as we go through all the elements to merge them in a single list. This result in 2n work and we do this work $\log_2 n+1$ times, the depth of the tree structure that is unfolded with the recursion. So, the runtime of the algorithm is 2n(log2n+1).

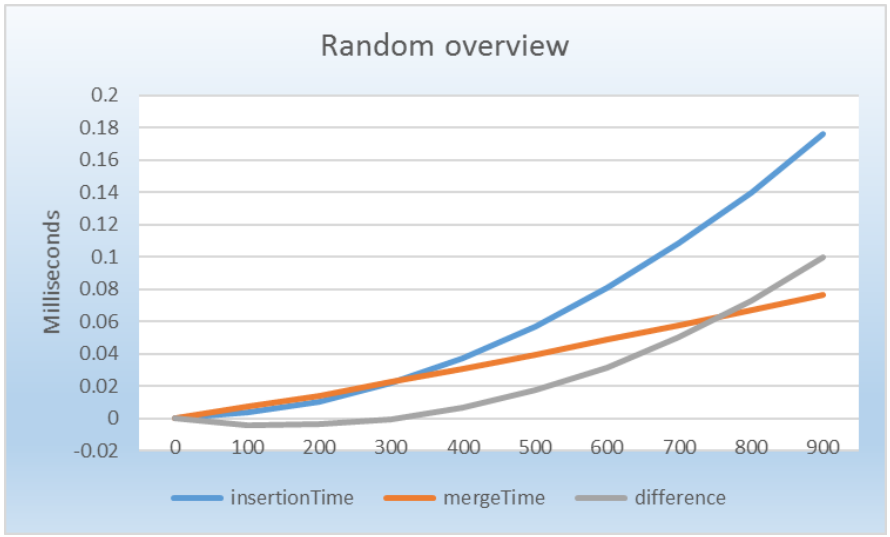


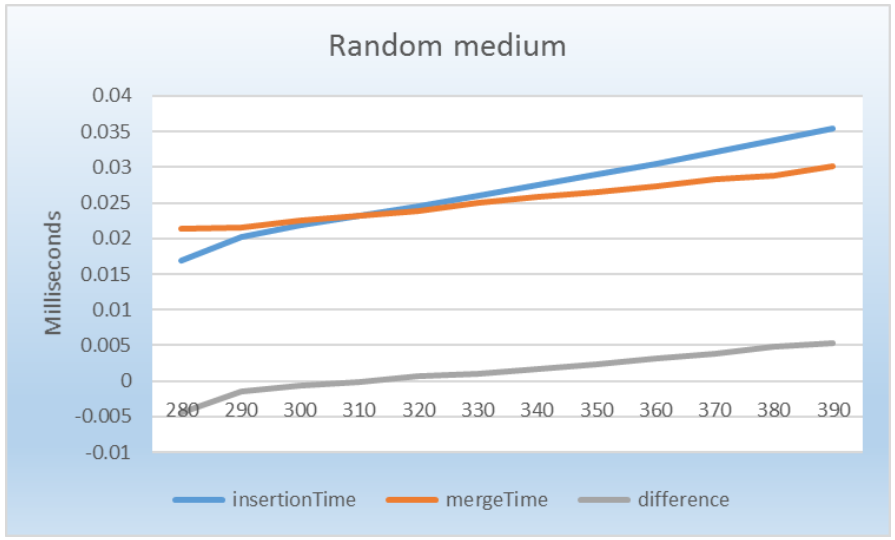
I plotted the results in the right graph. This turned out to be far from the results I produced with doing tests run. I suppose that the difference comes from the way instructions are executed in the language.

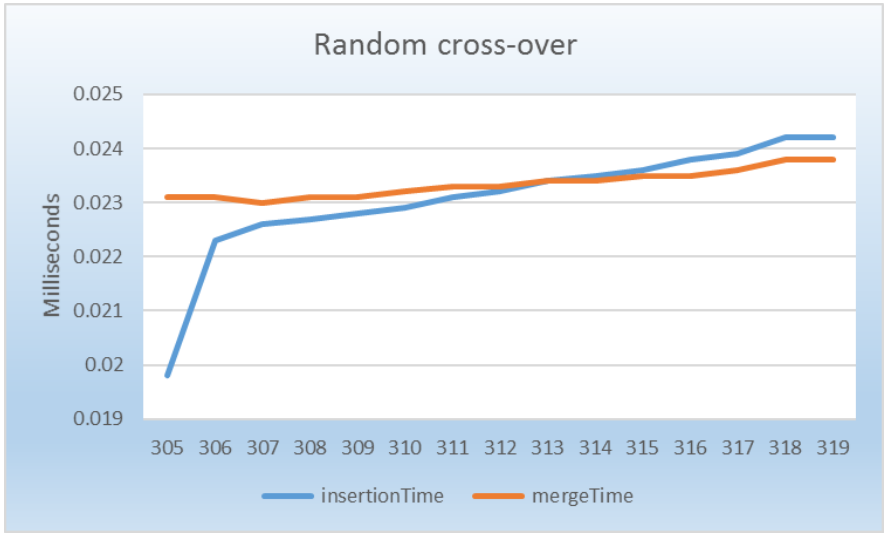
Charts

The following charts are created from data exported by my program in a csv files. Because the content of the data also influences the runtime of the sorts I test three different types of data: random, sorted and reverse. This illustrates better the difference between the way the algorithms work and their runtime.

Random

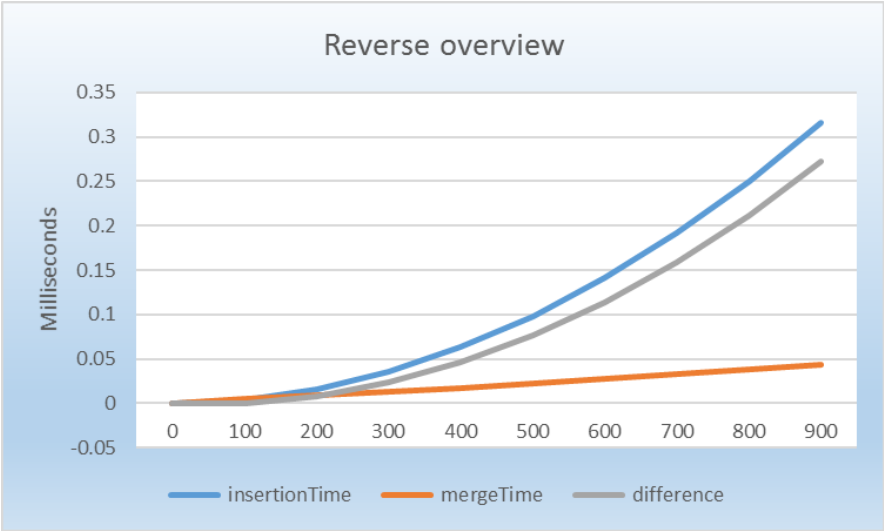


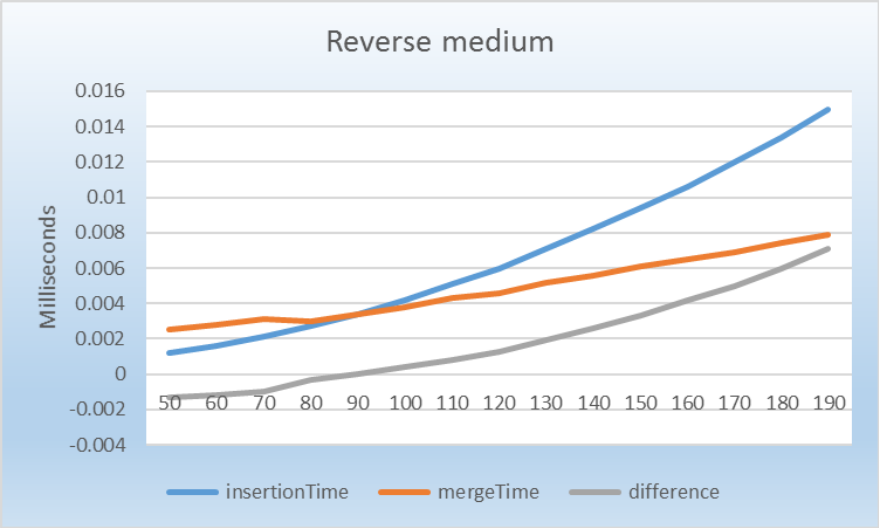


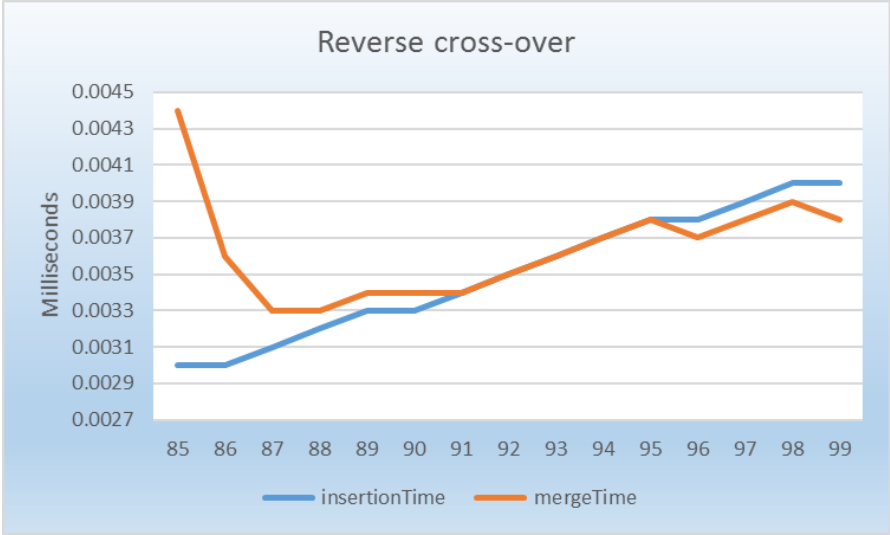


The expected- or average-case asymptotic time complexities for insertion sort is O(n2) and for merge is O(n logn). The surrounding graphs support this statement. The overview can be easily compared to the mathematical plotting on the previous page. The curve of insertion sort looks quadratic and the merge one logarithmic. To simulate an average case, I generated an array with random values.

Reverse

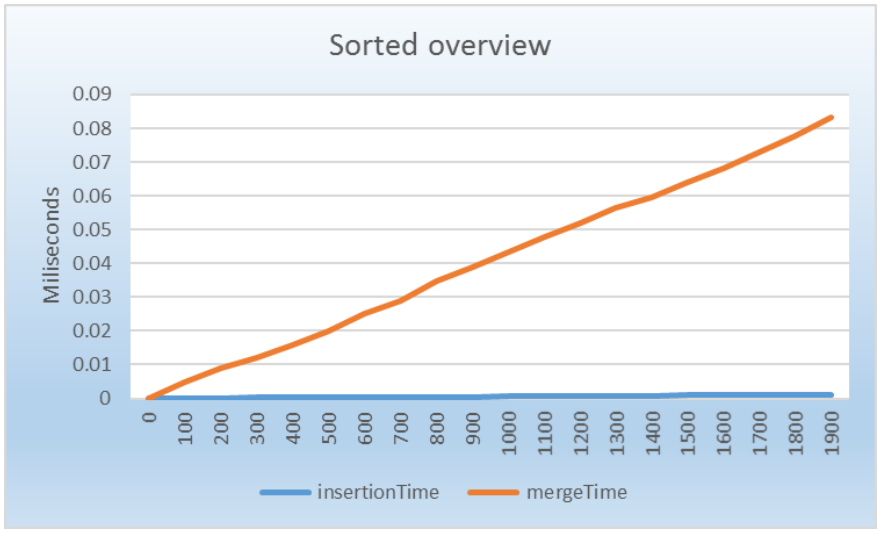


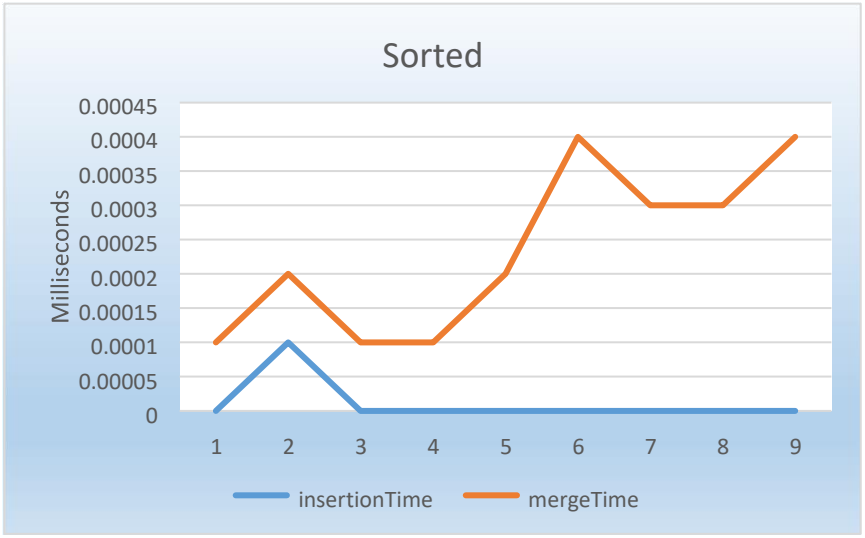




In contrast to insertion sort, merge doesn't have worst and best cases. Here I am testing the sorts with a reverse array to see how the algorithms time complexity would change. It turns out that the cross point has moved to the origin of the graph. The reason is that insertion sort is performing at its worst, when merge does almost the same amount of work.

Sorted





The best case of insertion sort is O(n), when the array it process is sorted. In this case the two algorithms doesn't have a cross-over except from the origin, because merge would always do the partitioning and copping of the elements. The upper graphs show exactly that.

Code

My implementation of insertion sort works in place, in contrast to merge. This makes it possible to pass the same array with data to both sorts, but the order is critical as it should prevent the second sort to use the sorted array from the first one. The practical specification said to use the milliseconds from the system, but I found that I needed nanoseconds, because the algorithms are fast and this way smaller diversions can be taken into account. To ensure that the algorithms are sorting the data correctly I run a check on the result of each one in the tests.

Tests

Java isn't a practical computer language for comparing algorithms, because it is an intermediate representation as java bytecode. This introduces another step when running the program so most of the following graphs has peeks at the origin. Other processes can also influence the performance of the algorithms. This is the reason I choose to test each n with 100000 tests, a bit excessive. The next few graphics show how the number of tests flatten the curves move the cross-over point to the right. If I had more time I would do the tests with a median rather than average.

Tests charts