Report Analysis for insertion sort

The analyzed algorithm is a regular insertion sort. It works in a very simple and intuitive way: it takes one element at a time from the unsorted part of the array and places it into the correct position in the sorted part on the left. To do this, it shifts larger elements to the right until it finds where the current element should go.

There is no fancy optimization here - just pure insertion sort logic. This makes it easy to read and easy to analyze both theoretically and experimentally.

The implementation also includes a PerformanceTracker that counts the number of comparisons, swaps (actually element moves), array accesses, and total execution time. That lets us see how the algorithm behaves in different input situations, not just from a complexity standpoint but also in terms of real measured performance.

The benchmark runs the sort on arrays with different initial distributions -> random, sorted, reversed, and nearly sorted -> and stores all results into a CSV file. This setup allows comparing theory with reality in a clear, visual way.

Complexity Analysis

Insertion sort has very predictable behavior depending on how sorted the input already is.

In the best case, when the array is already sorted, every new element only needs to be compared once with the previous one, so no shifting happens. The total time increases linearly with the number of elements. This is O(n).

In the average case, when the array is random, about half of the sorted portion has to be moved on each iteration. That adds up to roughly (n^2)/4 operations, which is still O(n^2).

In the worst case, when the array is reversed, every new element has to be moved through the entire sorted part. The total number of movements becomes 1 + 2 + 3 + ... + (n-1), which equals n*(n-1)/2. That is quadratic growth -> O(n^2).

Memory usage stays constant since everything happens in place. Only a few variables are used, so space complexity is O(1).

To summarize clearly:
best case -> O(n)
average case -> O(n^2)
worst case -> O(n^2)
space -> O(1)

So insertion sort is efficient only when data is already sorted or almost sorted. Otherwise, it becomes slow very quickly.

Code Review

The structure of the code is good - simple, readable, and divided into small classes. The use of PerformanceTracker to measure different types of operations is a nice addition. It gives an educational view of what happens inside the algorithm.

Still, there are a few issues and possible improvements:

-> In the benchmark, before saving results to "docs/performance_results.csv", it would be safer to create the folder first with new File("docs").mkdirs();.

-> The counter called swaps doesn't really measure swaps (like in bubble sort). It actually counts movements when elements are shifted to make space. It would be clearer to rename this metric to moves.

-> The metric counting of array accesses could be made a bit more accurate, since each move includes both a read and a write.

Other than these small points, the code is clean and structured logically. It's exactly what a teaching implementation should look like: easy to follow, easy to instrument, and simple to test.

Empirical Results

The results line up perfectly with what theory says.

For random arrays, the runtime grows roughly as n^2. You can see that the time grows much faster than the input size:
n=100 -> 0.34 ms
n=1,000 -> 15.7 ms
n=10,000 -> 43.3 ms
n=100,000 -> 3550 ms

This means that when the input increases by 10 times, the time increases by more than 200 times, which matches the quadratic pattern.

For already sorted arrays, times are basically flat: less than 0.1 ms even for 100,000 elements. Comparisons grow linearly, swaps stay at zero. That's the O(n) case in action.

For reversed arrays, we hit the true worst case. Time jumps from 0.36 ms at 100 elements to almost 7 seconds at 100,000 elements. The number of moves is around n^2 / 2, exactly as expected.

Nearly sorted arrays sit between these two extremes. For example, 100,000 elements take about 90 ms - slower than perfectly sorted data but far faster than random or reversed data. This shows that insertion sort performs well when the array is almost sorted, which is one of its practical strengths.

Overall, the measurements confirm every theoretical statement:
-> Best case -> linear

-> Average and worst -> quadratic
-> Nearly sorted -> in-between

So both the numbers and the shapes of the performance curves confirm the theoretical expectations.

Conclusion

This implementation of insertion sort behaves exactly how theory predicts. It is very efficient for small or nearly sorted datasets and very inefficient for large or random ones. The performance data clearly demonstrates the n^2 growth pattern for most cases.

The code itself is well-organized and works as a good educational example. The few technical problems are easy to fix.

If someone wanted to make it slightly faster, it could be improved by minimizing repeated array accesses or by skipping the already-sorted check to save one pass. But these changes wouldn't change the fact that insertion sort's growth is quadratic - only its constants would get smaller.

In practice, insertion sort is often used as a helper for more advanced algorithms -> for example, as a subroutine for small partitions in merge sort or quicksort. But as a standalone algorithm, it's best suited for small datasets or situations where data is almost sorted already.

In conclusion, the empirical and theoretical results match perfectly. The algorithm's runtime follows n^2 growth for random and reversed arrays and n growth for sorted data. The implementation is simple, educational, and a good demonstration of how algorithmic theory translates directly into real performance.

Report made by Alexandr Taurulin, SE-2408