CMPT 295 Mini-Project: Insertion Sort Performance Analysis

Introduction & Implementation

I evaluated the performance of Insertion Sort, a classic $O(N^2)$ algorithm known for its efficiency on small or nearly-sorted arrays and excellent memory locality. I compared three implementations against `std::sort`:

1. **`insertion_sort_arr` (Baseline):** Standard C++ using raw pointers (`int*`) and a `while` loop.
2. **`insertion_sort_vec` (Vector):** Uses `std::vector<int>&` and indexing to test abstraction overhead.
3. **`insertion_sort_optimized`:** Uses `std::upper_bound` (Binary Search) for $O(\log i)$ lookups and `memmove` for block memory transfers, eliminating the inner loop branch.
4. **`std::sort`:** The C++ Standard Library's Introsort as a reference.

Evaluation Method

Benchmarks were written in C++ (`main.cpp`) and compiled with `g++` 11.4.0 at `-O0`, `-O2`, and `-O3`. Execution time was measured using `std::chrono::high_resolution_clock` on input sizes up to 50,000 integers. Patterns included Sorted, Reverse, Random, and Duplicates.

- **Metrics:** Wall-clock time (avg of 5 trials), `perf stat -d` counters (cycles, instructions, branches), and memory usage.
- **Environment:** All tests ran on the same machine in a single session with background apps minimized.

Results

Optimization Level Comparison (50k elements)

Algorithm	Optimization	Sorted (ms)	Reverse (ms)	Random (ms)
`insertion_sort_arr`	`-O0`	0.09	1710.69	852.03
	`-O2`	0.03	288.51	142.75
	`-O3`	0.03	282.54	140.72
`insertion_sort_optimized`	`-O0`	0.04	58.66	32.47
	`-O2`	0.01	56.33	29.41
	`-O3`	0.01	54.64	28.08

`std::sort`	`-O0`	4.92	3.91	8.77
	`-O2`	0.31	0.22	2.00
	`-O3`	0.27	0.20	1.94

Algorithm Performance Comparison (-O3, 50k elements)

Algorithm	**Sorted (ms)**	**Reverse (ms)**	**Random (ms)**	**Duplicates (ms)**
`insertion_sort_arr`	0.026	282.54	140.72	118.16
`insertion_sort_vec`	0.033	367.34	181.86	152.07
`insertion_sort_optimized`	**0.011**	**54.64**	**28.08**	**22.47**
`std::sort`	0.271	0.198	1.94	0.62

Key Observations: The baseline `insertion_sort_arr` saw a 6x speedup from `-O0` to `-O2` as variables moved from stack to registers. The `insertion_sort_optimized` version was resilient to optimization levels (only ~10% speedup) because `memmove` is always optimized. On reverse data (worst case), the optimized version (54ms) was **5.1x faster** than the baseline (282ms) by replacing the element-by-element shift with `memmove`. The vector version was consistently slower due to bounds-checking overhead.

Analysis (perf & Assembly)

Metric	**`-O0`**	**`-O2`**	**`-O3`**
Instructions	1.33 Trillion	223 Billion	223 Billion
Cycles	334 Billion	38 Billion	37.6 Billion
IPC	3.98	5.84	5.95
Branch Misses	14.7 Million	36.9 Million	12.1 Million

Performance Counters: The 6x instruction reduction from `-O0` to `-O2` explains the speedup. L1 cache misses remained constant (~2B), confirming memory access is the bottleneck. `-O3` achieved the lowest branch misses (12.1M) and a high IPC of ~5.95, indicating aggressive loop optimization and superscalar execution.

The largest input (50k ints ≈ 200 KB) fits entirely in the L3 cache of my machine, which is why the LLC miss rate stayed near zero. Most memory stalls came from L1 churn during the repeated backward shifts in the baseline algorithm. Since insertion sort repeatedly iterates through the same contiguous array region, the hardware prefetcher can keep up reasonably well, but the $O(n^2)$ nature still causes frequent L1 evictions. The optimized version reduces these repeated passes, leading to fewer L1 misses per element.

Assembly Observations: Using Compiler Explorer and files `sorts_o0.S` to `sorts_o3.S`, I observed that the baseline generated a tight loop with a conditional jump (`cmp`, `jge`) that is hard to predict on random data. The optimized version replaced this loop with `memmove` (using SIMD instructions like AVX), removing the branch entirely.

None of the insertion sort loops were vectorized by the compiler, which is expected because each iteration depends on the result of the previous (a[j] > key and j--). This loop-carried dependency prevents automatic SIMD parallelization. The only SIMD usage in this project came from the underlying memmove implementation, which on my machine likely uses 128- or 256-bit wide transfers. This explains why the optimized version benefits from higher memory bandwidth while the baseline versions remain scalar.

Conclusion

This project demonstrated that **Insertion Sort can be optimized to be 5x faster** by changing *how* it moves data. The `memmove` optimization transforms a "branch-heavy" task into a "bandwidth-heavy" task, which modern CPUs handle much better. The optimized version avoids the branch misprediction penalties that hammer the baseline on random/reverse data. While `std::sort` remains fastest for large random sets, the optimized insertion sort is excellent for small or nearly-sorted arrays. Writing hardware-friendly code (predictable branches, linear memory) is just as critical as algorithm choice.