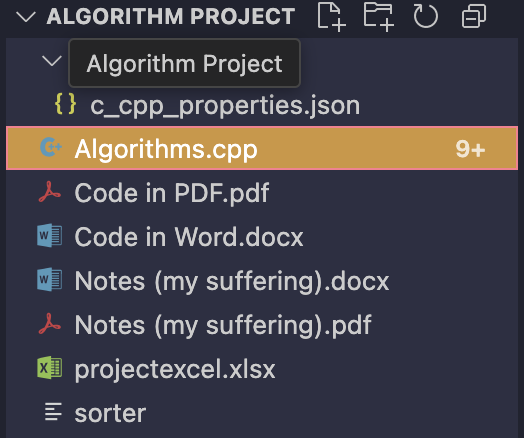
**Explaining Project Code and output**



This program was originally meant to compare sorting algorithms based on the number of comparisons they make and write a code to manually add it onto a chart then generate an excel graph from it. However, I expanded it significantly. I added accurate runtime measurement using high-resolution clocks, displayed both comparisons and time in milliseconds with 4 decimal placesn graphed - because nano was too big, and exported everything to a well-designed Excel report using the libxlsxwriter library. The final product is a tool that not only runs five sorting algorithms but also shows their performance in a clean, professional report with charts and detailed formatting.

**\*\*Note:** ./sorter runs the program on macOS and Linux. It might not work like that on Windows unless you’re using something like Git Bash or WSL. It’s the command used to run the compiled C++ file.

1. **Sorting Algorithms Used (they were based on project instructions)**

The code implements five core sorting algorithms:

* Selection Sort
* Bubble Sort
* Insertion Sort
* Merge Sort (using recursion)
* Quick Sort (also recursive)
* And also added nlogn because why not

Each algorithm is written as a separate function and follows the logic described in algorithm textbooks. Before running any sort, I reset a global variable called comparisons to 0. This variable is increased every time an element comparison is made. After the sorting is done, I store the total number of comparisons and the time it took to complete, so I can display or write them later.

Both Merge Sort and Quick Sort use helper recursive functions to work properly. For example, Merge Sort splits the array into halves, sorts each half, and then merges them while counting comparisons. Quick Sort uses the last element as a pivot, partitions the array around it, and recursively sorts both sides, also counting comparisons.

In this project, I measured two things for each sorting algorithm:

* **Number of comparisons** (how many times elements were checked against each other), and
* **Time taken** (measured in milliseconds using high-resolution timers).

Each algorithm behaves differently, and their performance depends on both the size and arrangement of the input data. Below is a general explanation of how each one works in terms of time complexity and comparison counts.

#### **Selection Sort**

* **Time Complexity:** Always O(n²)
* **Comparisons:** Exactly (n × (n − 1)) / 2 for all inputs
* **Why:** It always searches the rest of the array to find the smallest element, no matter how sorted the array already is.
* **Time Behavior:** Slow, especially for large arrays, because it does a lot of comparisons and swaps.

#### **Bubble Sort**

* **Time Complexity:** O(n²) in the average and worst cases, O(n) if already sorted (best case)
* **Comparisons:** Worst-case is (n × (n − 1)) / 2, similar to Selection Sort
* **Why:** It repeatedly compares and swaps adjacent elements. Even if sorted, it still needs one pass to confirm.
* **Time Behavior:** Very slow for large inputs; often the worst in practice.

#### **Insertion Sort**

* **Time Complexity:** O(n²) in the worst case, O(n) in the best case (when already sorted)
* **Comparisons:** Depends on how unsorted the array is. Worst case is again around (n²)/2.
* **Why:** It compares the current element with everything to its left until it's placed correctly.
* **Time Behavior:** Can be very fast for small or almost-sorted arrays.

#### **Merge Sort**

* **Time Complexity:** Always O(n log n)
* **Comparisons:** Around n log₂(n) comparisons
* **Why:** It divides the array into halves, recursively sorts them, and merges while comparing elements.
* **Time Behavior:** Consistently efficient, good for large inputs.

#### **Quick Sort**

* **Time Complexity:** O(n log n) on average, O(n²) in worst case (bad pivot)
* **Comparisons:** About n log₂(n) comparisons in average case
* **Why:** It partitions the array around a pivot and recursively sorts both halves.
* **Time Behavior:** Usually faster than Merge Sort in practice, unless the pivot is badly chosen.

#### **nlogn (Theoretical Baseline)**

* **Time Complexity:** O(n log n) – theoretical best for comparison-based sorting
* **Comparisons:** Calculated as n × log₂(n)
* **Time:** Set to **0 ms** in the program
* **Why:** This is **not an actual algorithm**, but a theoretical lower bound used for reference. It shows the fewest number of comparisons a “fast” sort could make. Since no actual sorting is performed, the runtime is set to zero to show it's just a formula and not a timed operation.

By showing nlogn with a comparison count but zero time, I created a **reference line** to compare actual algorithms against ideal performance. This helps visually demonstrate which algorithms are close to optimal and which ones are far behind, especially as the input size grows.

This comparison also helps understand that while some algorithms might have similar theoretical complexity (like Quick Sort and Merge Sort), their actual runtime **behavior** may vary due to recursion depth, constant factors, and memory access patterns, something the program captures accurately using real time measurements and visual graphs.

1. **Timing with Chrono (and Runtime Accuracy)**

To measure performance, I used the C++ <chrono> library. It gives access to high-resolution timers that can measure time in nanoseconds. Here's how I did it:

* I used auto start = high\_resolution\_clock::now(); just before the sort runs.
* Then after the sort, I called auto end = high\_resolution\_clock::now();
* I subtracted the two and converted the result from nanoseconds to milliseconds, using duration\_ns / 1'000'000.0, and I stored it as a long double.

To make small runtimes more readable—especially for smaller arrays like n = 100—I displayed time with 4 decimal places using:

cout << fixed << setprecision(4);

This made even tiny times like 0.0034 ms show up correctly instead of rounding to 0, and better than nanoseconds which gave huge numbers.

1. **Excel Export with libxlsxwriter**

One of the most unique parts of my project is how I used the libxlsxwriter library to generate an Excel file from C++ instead of manually inputting the code answer into a table.

Here’s how I did it:

* I created a new workbook with workbook\_new("projectexcel.xlsx").
* I added a worksheet using workbook\_add\_worksheet().
* I defined custom formats for headers, rows, and separator lines:
  + Headers are bold, white text, with a dark blue background.
  + Data rows have a light blue background, centered text, and borders.
  + Separator rows are left blank with a white background, used to visually break blocks of data in Excel.
* I wrote the input size, algorithm name, number of comparisons, and runtime into rows using worksheet\_write\_number() and worksheet\_write\_string().

After writing each input block, I checked if two blocks were written. If yes, I moved to the next Excel column group using current\_col += 5, to avoid the Excel sheet getting too long vertically.

**4. Summary Tables and Charts**

At the bottom of the Excel sheet, I added two summary tables and two charts:

1. **Comparisons Table** – shows all input sizes in rows and each algorithm’s comparisons in columns.
2. **Comparisons Chart** – line graph with input size on the X-axis and number of comparisons on the Y-axis.

Then, under that:

1. **Runtime Table** – shows the runtime (in milliseconds) for each algorithm, again per input size.
2. **Runtime Chart** – another line chart, this time with runtime (ms) on the Y-axis.

These charts were generated using:

lxw\_chart \*chart = workbook\_add\_chart(workbook, LXW\_CHART\_LINE);

chart\_series\_set\_categories(...);

chart\_series\_set\_values(...);

The output is a clean, professional Excel file.

1. **Time Complexity: n log n Baseline**

For reference, I also calculated a theoretical value of n \* log2(n) for each input size. This represents the best-case time complexity of an efficient sorting algorithm.

I displayed it in the table with the label nlogn, and left the runtime cell empty (set to 0). This row is used as a baseline so we can compare actual results against theoretical efficiency.

long long theo\_nlogn = static\_cast<long long>(n \* log2(n));

writeExcelRow(current\_row, current\_col, n, \"nlogn\", theo\_nlogn, 0);

This made it easier to judge which algorithms match nlogn performance and which fall behind.

1. **Compiler Issues and Command-Line Build**

One issue I faced was that my default compiler couldn't link the libxlsxwriter library correctly- or actually any library at all (check my suffering pdf attached). My IDE didn’t support it out of the box, so I had to use the command line to compile and link manually.

I used the following command:

/opt/homebrew/opt/llvm/bin/clang++ -std=c++17 Algorithms.cpp -o sorter \\

-I/opt/homebrew/include \\

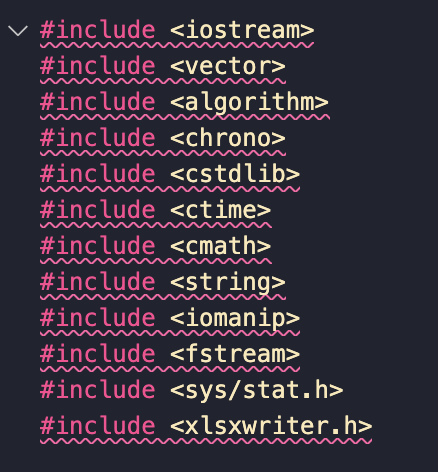
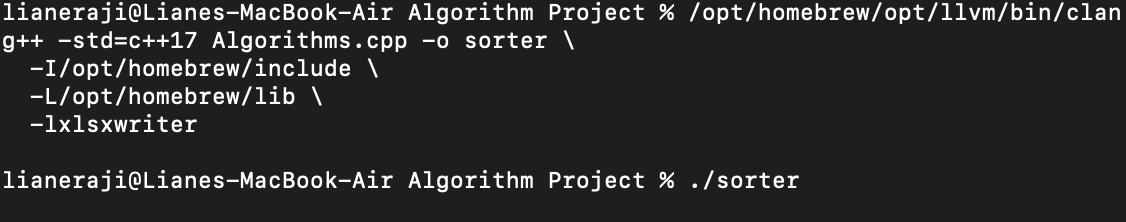
-L/opt/homebrew/lib \\

-lxlsxwriter

Then, I ran the compiled program using:

./sorter

This approach allowed me to link everything correctly on macOS, using the Homebrew-installed version of libxlsxwriter.



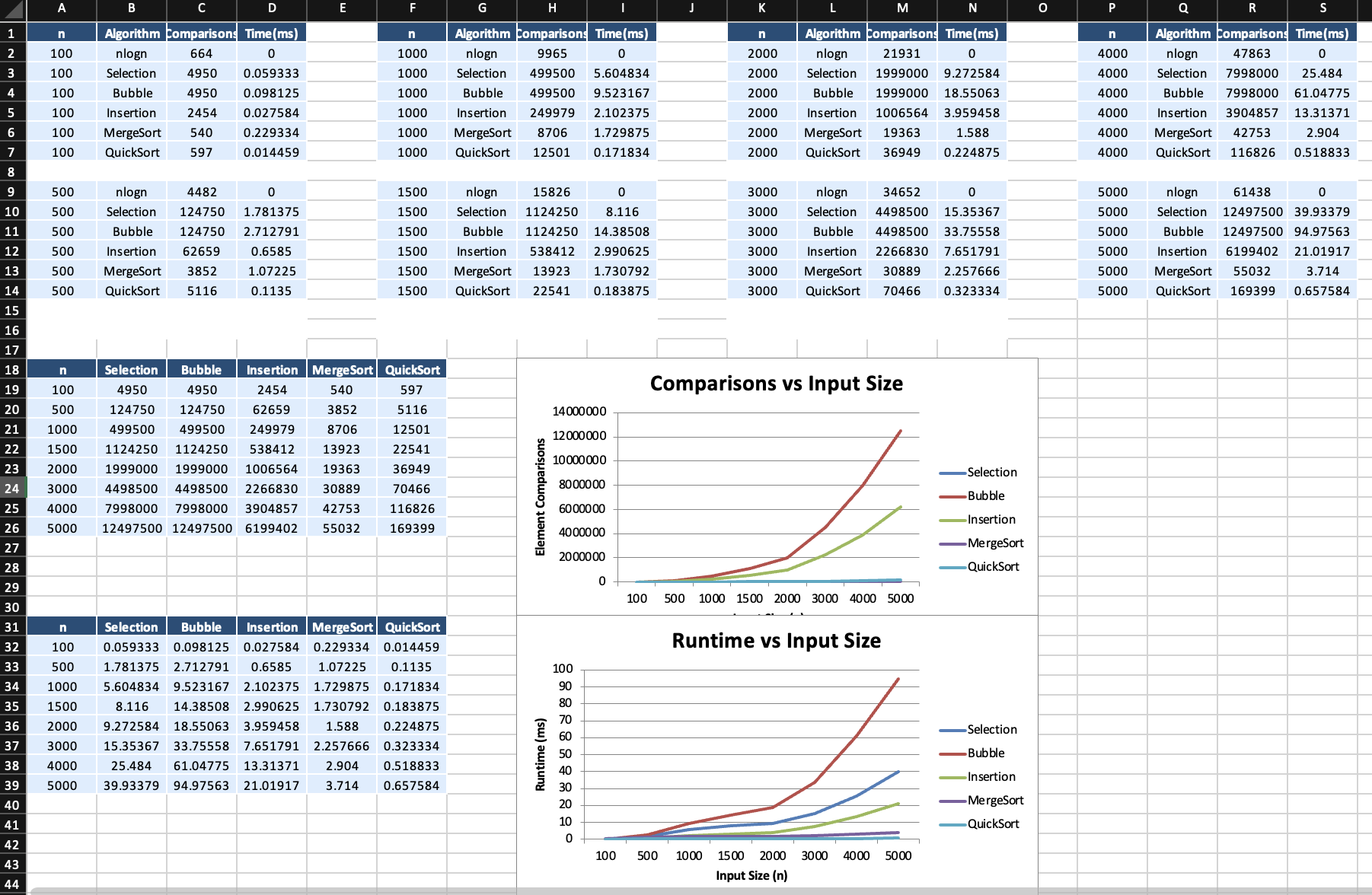
After running all five sorting algorithms across multiple input sizes, the data clearly supports the theoretical time complexities we expect from each algorithm. The number of comparisons and runtime both increase with the input size, but at different rates depending on the algorithm. The Excel summary tables and charts made these differences very clear.

As expected, Selection Sort, Bubble Sort, and Insertion Sort all showed a quadratic growth in both comparisons and runtime. For example, as the input size increased from 100 to 5000, their number of comparisons grew roughly in proportion to n2n^2n2, making them very inefficient for large datasets. In particular, Bubble Sort had the highest runtime overall, even when its comparison count was similar to the others, likely due to the large number of swaps it performs.

In contrast, Merge Sort and Quick Sort performed much better. Their comparison counts closely followed the theoretical nlogn curve, and their runtime increased more slowly than the O(n²) algorithms. Merge Sort was very consistent in its performance, with stable comparison and runtime values across all input sizes. Quick Sort, on the other hand, had slightly more variation depending on how the pivot behaved in each run, but it still showed much better scalability compared to the simpler sorts.

The theoretical nlogn baseline row helped highlight the gap between efficient and inefficient algorithms. While Merge Sort and Quick Sort often stayed close to that line in both comparisons and time, the O(n²) algorithms moved further and further away as the input size increased. This confirmed the real impact of algorithmic complexity on actual performance—not just in theory but also in practice.

Finally, by visualizing the comparison and runtime data side by side in Excel charts, it became easy to understand the trade-offs and patterns. The graphs made it clear which algorithms scale well and which ones are only suitable for small datasets. Overall, the data gathered proves that algorithm choice matters a lot—and that understanding time complexity is essential when working with large-scale problems.



\*Top 14 rows are general summary showing all data. Bottom tables only show data relevant to the graph