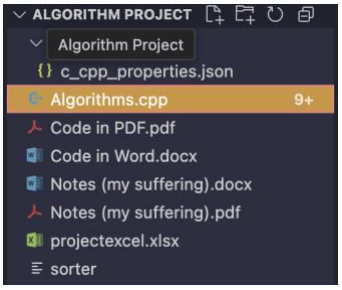
Explaining Project Code and output



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 $O(n^2)$
- Comparisons: Exactly $(n \times (n - 1)) / 2$ 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: $O(n^2)$ in the average and worst cases, $O(n)$ if already sorted (best case)
- Comparisons: Worst-case is $(n \times (n - 1)) / 2$, 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: $O(n^2)$ 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 $(n^2)/2$.
- 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 $n \log_2(n)$ 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, $O(n^2)$ in worst case (bad pivot)
- **Comparisons:** About $n \log_2(n)$ 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 $n \times \log_2(n)$
- **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.

2. 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.

3. 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:

3. Runtime Table – shows the runtime (in milliseconds) for each algorithm, again per input size.
4. 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.

5. 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.

6. 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.

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <chrono>
#include <cstdlib>
#include <ctime>
#include <cmath>
#include <string>
#include <iomanip>
#include <fstream>
#include <sys/stat.h>
#include <xlsxwriter.h>
```

```
lianeraji@Lianes-MacBook-Air Algorithm Project % /opt/homebrew/opt/llvm/bin/clan
g++ -std=c++17 Algorithms.cpp -o sorter \
  -I/opt/homebrew/include \
  -L/opt/homebrew/lib \
  -lxlsxwriter

lianeraji@Lianes-MacBook-Air Algorithm Project % ./sorter
```

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.

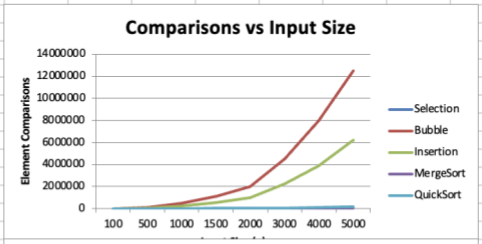
In contrast, Merge Sort and Quick Sort performed much better. Their comparison counts closely followed the theoretical nlogn curve, and their runtime increased more slowly than the $O(n^2)$ algorithms. Merge Sort was very consistent in its performance, with stable comparison and runtime values across 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 $O(n^2)$ 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.

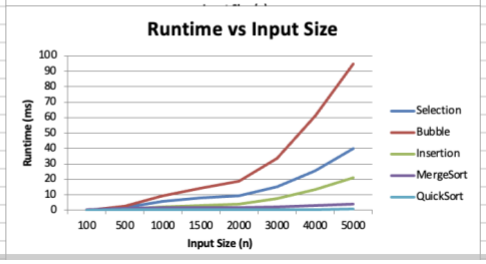
Finally, by visualizing the comparison and runtime data side by side in Excel charts, it became easy to understand the trade-offs and patterns. The graphs made it clear which algorithms scale well and which ones are only suitable for small datasets. Overall, the data gathered proves that algorithm choice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

n	Algorithm	Comparisons	Time(ms)	n	Algorithm	Comparisons	Time(ms)	n	Algorithm	Comparisons	Time(ms)	n	Algorithm	Comparisons	Time(ms)
100	nlogn	664	0	1000	nlogn	9965	0	2000	nlogn	21931	0	4000	nlogn	47863	0
100	Selection	4950	0.059333	1000	Selection	499500	5.604834	2000	Selection	1999000	9.272584	4000	Selection	7998000	25.484
100	Bubble	4950	0.098125	1000	Bubble	499500	9.523167	2000	Bubble	1999000	18.55063	4000	Bubble	7998000	61.04775
100	Insertion	2454	0.027584	1000	Insertion	249979	2.102375	2000	Insertion	1006564	3.959458	4000	Insertion	3904857	13.31371
100	MergeSort	540	0.229334	1000	MergeSort	8706	1.729875	2000	MergeSort	19363	1.588	4000	MergeSort	42753	2.904
100	QuickSort	597	0.014459	1000	QuickSort	12501	0.171834	2000	QuickSort	36949	0.224875	4000	QuickSort	116826	0.518833
500	nlogn	4482	0	1500	nlogn	15826	0	3000	nlogn	34652	0	5000	nlogn	61438	0
500	Selection	124750	1.781375	1500	Selection	1124250	8.116	3000	Selection	4498500	15.35367	5000	Selection	12497500	39.93379
500	Bubble	124750	2.712791	1500	Bubble	1124250	14.38508	3000	Bubble	4498500	33.75558	5000	Bubble	12497500	94.97563
500	Insertion	62659	0.6585	1500	Insertion	538412	2.990625	3000	Insertion	2266830	7.651791	5000	Insertion	6199402	21.01917
500	MergeSort	3852	1.07225	1500	MergeSort	13923	1.730792	3000	MergeSort	30889	2.257666	5000	MergeSort	55032	3.714
500	QuickSort	5116	0.1135	1500	QuickSort	22541	0.183875	3000	QuickSort	70466	0.323334	5000	QuickSort	169399	0.657584

n	Selection	Bubble	Insertion	MergeSort	QuickSort
100	4950	4950	2454	540	597
500	124750	124750	62659	3852	5116
1000	499500	499500	249979	8706	12501
1500	1124250	1124250	538412	13923	22541
2000	1999000	1999000	1006564	19363	36949
3000	4498500	4498500	2266830	30889	70466
4000	7998000	7998000	3904857	42753	116826
5000	12497500	12497500	6199402	55032	169399



n	Selection	Bubble	Insertion	MergeSort	QuickSort
100	0.059333	0.098125	0.027584	0.229334	0.014459
500	1.781375	2.712791	0.6585	1.07225	0.1135
1000	5.604834	9.523167	2.102375	1.729875	0.171834
1500	8.116	14.38508	2.990625	1.730792	0.183875
2000	9.272584	18.55063	3.959458	1.588	0.224875
3000	15.35367	33.75558	7.651791	2.257666	0.323334
4000	25.484	61.04775	13.31371	2.904	0.518833
5000	39.93379	94.97563	21.01917	3.714	0.657584



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