CSC-411 Assignment 1

Anthony Ozog

January 21, 2026

Introduction

Since I am a math major I have explored most programming and software development I know outside of my studies and as a result I have never gotten to implement any of these sorting algorithms before. I really took this as an opportunity to both learn these new algorithms and practice some of the tools I like using (i.e. LaTeX, rust, R, and Test Driven Development). As a bonus I also got to see the capabilities of the rust language, which I am quite fond of, and I would say the language/compiler preformed quite well, at first I was measuring in Microsecond and I was still getting 0 results for smaller n's so I switched to measuring Nanoseconds

1 Sorting Implementations

Over all the implementation was really fun but I found myself a little annoyed with the bounds checking rust imposes to maintain safety, I wish there was some way to turn it off for a whole function or module, but that is probably something that will be addressed in the future.

1.1 Bubble Sort

I use a match statement to handle boolean for the early exit optimization, this reduces necessary rewrite of the boolean. I also sure a rust `unsafe` block to skip the bounds check when I compare elements; however, if I wanted it to be even faster I should have written my own unsafe version of swap to skip the bounds check since rust does not natively provide an unchecked swap.

1.2 Insertion Sort

It my first implementation I used a `.clone` on a `usize` but I later remember that `usize` implements a `copy` trait which is a little faster so I came back and fixed that. Otherwise I was a standard implementation with the same bounds checking optimizations and concessions as mentioned previously.

1.3 Merge Sort

I used an unsafe block to skip the bounds check when I split the vector. Also in my merge function I use iterators to make the merge much faster. I should be using slices instead of vectors so that it can be done without cloning but I did not implement that. Finally the same bounds checking optimizations and concessions apply here as well.

2 Input Generation

For any randomly generated numbers I used this rng from the rand crate in rust.

2.1 Uniform Random

I create an iterator of random numbers, take n of them, and collect it into a `Vec<i64>`.

2.2 Sorted

I call the function to generate a Uniform Random vector of n elements then I sort it using the built in sort function.

2.3 Reverse Sorted

I call the function to generate a Uniform Random vector of n elements then I sort it using the built in sort function and then I reverse it using the built in function.

2.4 Almost Sorted

I call the function of generate a sorted vector of `i64` then make $\frac{n}{4}$ random swaps using the swap function and random from 0 to n.

2.5 Pipe Organ

I call the function to generate 2 sorted vectors of `i64` of length $\frac{n}{2}$ then I append which ever vector has the smaller maximum element to the other. I do this to ensure I am not increasing past the halfway mark.

3 Timing

The only thing to note about the code I used form the timing is that I used this to handle the timing. I also used `clone` more than a optimized program should have but the `clone` does not affect the timing

4 Analysis

To note, this was both my first time implementing these sorting algorithms and my first time comparing them. None of the results surprised me aside from the tests on the sorted data.

4.1 Running the code

To note,

1. make sure you are in the correct directory
2. make sure you have cargo installed
3. the instructions to install it can be found here here
4. make sure you have R installed as well
5. the instructions to install it can be found here
6. make sure you have the R jsonlite library installed too
7. run the following commands in bash if you do note have the library
    - sudo R
    - install.packages("jsonlite")

```
cargo test
cargo build -r
./target/release/CSC-411-Assignment-1
# or however you would run a binary on the operating system of your choice
```

```
# next I make a fresh dir and run the 2 R scripts on the new data
mkdir test
mv optimized_results.json test/
mv deg_space_results.json test/
cp /graphs/graph_building.r test/
cp /graphs/deg_graph_building.r test/
cd test
Rscript graph_building.r
Rscript deg_graph_building.r
```

4.2 Bubble and Insertion Sort Comparison

For random inputs Bubble Sort is much slower than Insertion Sort. I know that they have the same complexity but Bubble Sort preforms more swaps and comparisons so even though the theoretical complexity is the same among the two sorting algorithms Insertion sort is much faster. To this point at $n = 10,000$ Bubble Sort took $63,839,181$ Nanoseconds while Insertion Sort took $8,963,227$ Nanoseconds, $\times 8$ faster.

4.3 Benefits of nearly sorted inputs

The percent faster for each algorithm at $n = 10,000$ as compared to the Uniform Random data is as follows

Bubble Sort %31.94

Insertion Sort %49.79

Merge Sort %12.43

4.4 Input size and Merge Sort

4.5 Early-exit Bubble Sort

4.6 Theory vs. Experiment

5 Degraded Spatial Locality