CSC-411 Assignment 1

Anthony Ozog

January 20, 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

4 Analysis

5 Degraded Spatial Locality