Lab 3

Part 0

For part 0, I first looked into what algorithms to use to check for primes less than n. I choose to use the Sieve of Eratosthenes algorithm. It basically uses an boolean array of size n, that starts with all values false. You check all numbers less than √n. If their array value is true, then you increment the prime counter and mark all multiples of that number as false. Then, you traverse the back end of the array (all the indexes greater than √n), and if their corresponding value in the array is true, increment the primes counter. I chose to use this algorithm because it seemed to be the most efficient algorithm and simple algorithm out there for calculating the prime numbers smaller than a given n.

Design decisions to avoid unsafe rust features

To safely utilize multiple threads in my program, I protected the global counter "primes" and the "sieve" array by using the "Arc" and "Mutex" data types in rust. The Arc data type helps you to do safe reference counting, where multiple pointers can exist to the same reference. Then, the Mutex data type makes sure that only one thread is editing the data in the mutex at each time. Each thread only accessed the "primes" global counter at the end of their lifespan, to add their own local value to it. Each time access to the "sieve" array was not needed, it was "dropped"—meaning that the thread gave up its lock on the variable—so that another thread could use it. Using the recommendation from this Ed post, I used the nth thread to check if the nth number was prime or not. Then the number of "nth" primes less than the number provided to the program would be added to the global counter just before the thread ends. Then all threads are "joined" and the main thread waits for all other threads to finish.

I think the most difficult part of part 0 was figuring out how to manage ownership issues, which I have never had to deal with in C before. The most common error I kept getting was "value moved", when I tried to get a pointer to the mutex in each thread. I then realized that I had to get a reference to the global variable outside of the thread (which takes ownership of all values it uses given the move keyword), and then give that variable to the thread. Because I was using the value "primes" to get its reference, the thread was taking ownership of this value. It took me a bit to get used to these kinds of concerns when writing code.

Source Code for Part 0

For choosing and implementing the algorithm in Rust, I used the Sieve of Eratosthenes implementation of a prime number checker in this source. Since the implementation I modeled mine off of is in C++, the actual lines of code are different, but their meaning/function is the same. I also changed this code to make it more friendly to using multiple threads, by adding threads that performed the algorithm in the previous source on every nth number. I found the code for multi-threading and safe variable sharing in Rust in the Shared Concurrency section of the Rust Programming language book. I used their "Atomic Reference Counting with Arc<T>" example and replaced the code they had in their thread to "increment a counter" with the code from the previous source that checks if a number is prime. From the Accepting Command line arguments section in the Rust Programming book, I got lines 10-12 of my code for getting access to command line arguments like "n" and the number of threads the program should use. I lastly used this stack overflow poset to learn how to convert the string args to numbers for use in the rest of the program.