Enigma Breaker

Miguel Guthridge, Tutorial group H18B, z5312085, COMP6841

What I produced

A computer program that is capable of emulating an enigma machine, and of breaking the enigma cipher using brute-force methods. The code for this program is available at https://github.com/MiguelGuthridge/Enigma-Breaker.

Results

I have produced a piece of software that is capable of deciphering enigma machine enciphered messages, given only a small amount of known information. The software is performant, and can break ciphers in a matter of seconds, provided that they don't use the machine's plug board (as it causes the number of potential combinations to increase exponentially).

Reflector	Encoders	Plug board	Message constraints	Time to brute-force (s)
Known	3 unknown	No plugs	Message start given	1.1
Unknown	**3 unknown**	No plugs	Message start given	3.2
Unknown	3 unknown	No plugs	**Message end given**	3.5
Unknown	3 unknown	No plugs	**Message contains string**	69.4
Known	3 known	**3 plugs**	Message start given	6.6
Known	3 known	**4 plugs**	Message start given	329.4

What I did

Initially, my plans for this project were vastly different. I intended to create a programming language that I could use to implement basic ciphers. Unfortunately, after designing the language, I came to realise that my project was far too complex to implement in one term. My work on this project was primarily documentation of the language syntax, but I did get some work done for the tokenizer. All of my work for this project can be found at https://github.com/MiguelGuthridge/equator-lang. Overall, I spent about 5 hours working on this.

After realising the complexity, I went about looking for a new project. One of the course exercises I found extremely enjoyable was the exercise discussing enigma ciphers. As such, I decided to write a computer program to attempt to take advantage of various flaws in the design of enigma machines in order to brute-force decrypt enigma machines.

First, I implemented an enigma machine in Rust. This took about 10 hours, primarily due to the complexity of the rotor stepping - it was very challenging to get the stepping to work correctly, and I had to spend a lot of time debugging my test cases before I was confident that my implementation was correct.

From there, I started writing the code for brute-forcing through all possible combinations. Doing this was quite difficult, as I wanted to allow the user to specify any amount of known information, and have the program only iterate over combinations that fit the known information, which was a challenge to implement safely in Rust's strict type system. Getting an initial working implementation took about 10 hours.

Finally, I implemented some benchmarks to measure the performance of my software, and attempted to use these to optimise the software. I spent about 5 hours learning the `criterion` benchmarking library, implementing the benchmarks and attempting to improve my software's performance. One of the major performance improvements I got was from using the `rayon` library to parallelise the iteration.

How I was challenged

Although I was relatively familiar with the Rust programming language before starting this project, I hadn't created any fully-functional projects with it. As such, designing the implementation to be type-safe was quite challenging, and I learnt a lot about Rust's type system.

Additionally, before attempting this project, I did not have very much familiarity with the Rust ecosystem. As such, during this project, I learnt the basics of many popular libraries in Rust, including `itertools` (advanced iteration), `rayon` (parallelisation), `strum` (enum iteration), `serde` (data serialisation), `clap` (command-line argument parsing), and `criterion` (benchmarking).

One challenge that was particularly difficult to overcome was the discovery that contrary to my initial research, the Enigma machine is cryptographically secure, requiring 67 bits of work to brute-force. To address this, I changed my program's design so that it was more flexible, accepting constraints on its search to reduce the scope of the work. Ideally, this would be used by cryptographers to specify known data, thereby greatly increasing the speed of the search.