COM2108: Functional Programming -

The Enigma and the Bombe

8th December 2022

By: Ian Vexler Galarza

Table of Content:

1. Design:

1.1. Enigma Machine:

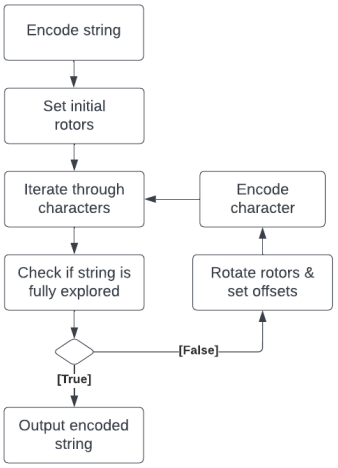

Figure 1.1.1.: General design diagram to simulate the Enigma Machine

In order to simulate the enigma machine a general plan was designed at first containing different stages of the process. The general design of the structure of the program can be seen in Figure 1.1.1. This diagram will allow us to understand the process as a whole to then be able to look and plan in careful detail what has to be done to achieve the different stages.

To achieve the simulation, a recursive process is followed to iterate through every letter of the string, stopping once the string is empty (meaning fully explored). Before the process of encoding can begin, the rotors have to be set according to the initial offsets. Once that is set the program will recursively apply the encoding on every character but taking the updated enigma resulting from the previous iteration.

The rotors and offsets are set before encoding each character to then go to the actual process of encoding. In order to rotate the rotors and set the offsets many checks have to be performed. Offsets have to be checked twice: to check for an overflow or to check for the knock-on positions. When the offset surpasses 25 it has to be reset to 0 as it has done a full rotation. It also has to be checked to see if the knock-on positions have been passed, and if so the subsequent rotor has to be rotated.

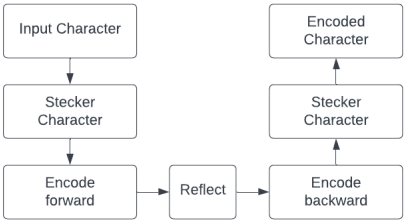

Figure 1.1.2.: Diagram of the encoding procedure

Encoding a character follows the procedure seen in Figure 1.1.2., the current character in the string is swapped with its steckerboad pair (in the case of having a plugboard), encoded following the rotors, reflected, encoded following the rotors in reverse and finally ‘steckered’ again. Encoding is achieved by finding the equivalent of the position of letter in the alphabet in the rotor or viceversa. This process is repeated for each character in the string but with incrementally rotated rotors and alphabets as it goes. Once the string has been fully encoded, the final encoded message is returned.

Lower level functions to be implemented first include all of those who act as helpers. For example, rotating the rotors, finding the equivalent of the position on a rotor or alphabet, reflecting letters and swapping a plugboard pair. Once those functions are set, implementing encoding on a single rotor is possible and therefore the next step. This enables repeating this process through multiple rotors and once repeating this process through every other character, it can fully simulate the enigma.

1.2. Longest Menu:

To find the longest menu every possible chain from every starting position has to be figured out. The program will iterate through all positions in the crib taking them as the starting point and adding them to a list (menu) to then find the longest menu. Figure 1.2.1. shows the general design diagram to achieve this.

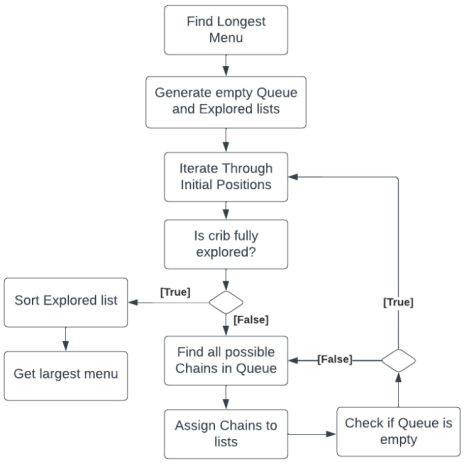

Figure 1.2.1.: General design diagram to find the Longest Menu

To be able to generate multiple menus a complete search of all menus has to be done. To do this, two list types named as queue and explored are defined. The first stores all the menus that are yet to be fully explored, whilst the second stores all fully explored menus. A fully explored menu consists of a menu that doesn't have any more chains for the last character.

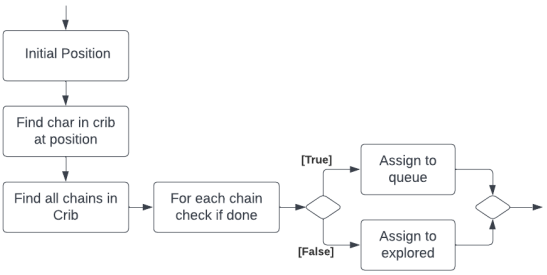

Figure 1.2.2.: In detail diagram of finding and assigning chains

The process of finding a chain starts by taking an initial position and finding the equivalent pair of characters in the crib. With this given initial position the program recursively looks for the next possible chains until all possibilities have been explored. In each iteration the chains are assigned to its corresponding list depending on the outcome of it. If chains are found they are added to the list Queue to be later explored, if no more chains are found it is assigned to Explored. Once all menus in the queue have been explored and therefore the queue list is empty, it will move on to the next starting position in the crib repeating the process.

Figure 1.2.3.: In details diagram of finding all chains in crib

Finding chains is achieved by looking if the crib contains any equivalent characters in the plain text. If it does, the positions of these chains are added to the possible menus and then these menus are added to a queue list. Before moving to the next iteration, the old queue is deleted to get rid of the incomplete menus. It is important to note that the items added to the menus cannot be already present in them, therefore a check has to be done before adding a new chain.

Once all possible menus have been explored and the queue list is empty the Explored list is sorted. From this it is possible to find the largest obtained menu, returning that as the final answer.

1.3. Breaking the enigma:

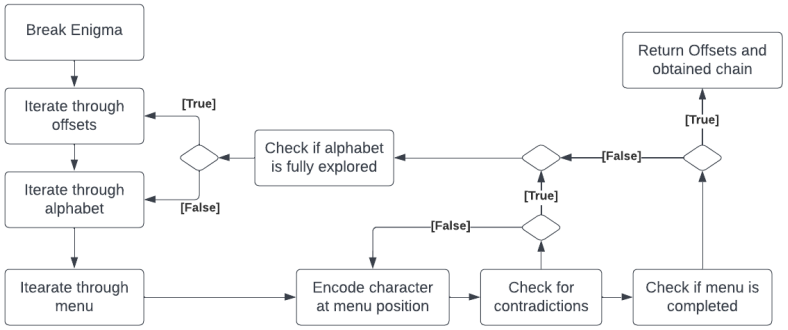

Figure 1.3.1.: General design diagram to Break the Enigma

Breaking the enigma requires a recursive program that iterates through all possible letters for all possible offsets. The general idea is that it iterates through a pair of starting letters and a starting offset to encode characters in a crib according to its menu position.

For every starting pair and offset, letters in a crib are encoded following the order of the menu. In order to take in consideration that different positions at a menu mean different rotor rotations, the enigmas have to be set accordingly before encoding. The starting enigma is rotated n times according to the position in the menu that is being encoded. The result of this encoding is then paired with the text in plain at the same position, to then be checked for contradictions.

Checking for contradictions is checking that all pairs follow a pattern. A character can only be paired to another one character, so if any encoding result shows different the program moves on to the next iteration. When checking, it has to be taken into consideration that a pair could be inverted (e.g. (‘A’,’B’) and (‘B’,’A’)) therefore it checks all possibilties. If no

contradictions are found, it is assumed that the pairs equal the plugboard configuration. If no chains can be successfully paired with a plugboard, then Nothing is returned.

2. System and Testing

Thorough testing of each function was made to make sure that they were all giving the correct output. This was made by testing each function as they were developed, from low-level to top-level.

2.1. Enigma Machine:

Before trying to simulate the Enigma, all helper functions were made. These include all functions to set rotors, pluboard and reflectors.

The first step of the development of the enigma machine was preparing the function that would rotate the rotors each time you encode a letter. These worked by moving the head of a list to the back Tests were done with each rotor like in Figure2.1.1. In a very similar approach the function offsetAlphabet was done to rotate the alphabet depending on the current offset.

```
ghci> rotateRotor rotor1
("KMFLGDQVZNTOWYHXUSPAIBRCJE",17)
```

```
ghci> offsetAlphabet 5 ['A'..'Z']
"FGHIJKLMNOPQRSTUVWXYZABCDE"
```

Figure 2.1.1.: Tests for rotating rotors and the alphabet

Other low level functions were in charge of preparing the plugboard and reflector. These had a similar approach in which they searched in tuples of characters if any matched with the character and returned the opposite in the tuple. Reflecting was done by the same approach but if no match was found it would return the same character.

```
ghci> steckerChar plugboard 'A'
'V'
```

```
ghci> steckerChar plugboard 'E'
'E'
```

```
ghci> reflect reflectorB 'A'
'Y'
```

```
ghci> reflect reflectorB 'Y'
'A'
```

Figure 2.1.2.: Tests for reflector and plugboard

In order to encode the characters equivalent values of a character in either the rotor or alphabet. This was achieved by finding the index of the character in the rotated rotor or the offseted alphabet and finding its equivalent in the opposite at the same index. As the enigma machine should be backwards compatible, testing as seen in Figure 2.1.3., was done with the same letters and rotors on both scenarios.

```
ghci> rotorEq 0 rotor1 'A'
'E'
```
```
ghci> alphabetEq 0 rotor1 'E'
'A'
```

Figure 2.1.3.: Tests for finding characters in rotor and alphabet positions

As we finally reach the encoding stage three functions were responsible for this, encodeChar, encodeFwd and encodeBack. These three combined would encode a character through all three rotors, reflect it and encode it backwards. Tests were done as shown in Figure 2.1.4., and as the previous examples these were with the same set of letters to make sure it was backwards compatible.

```
ghci> encodeChar 'A' (SimpleEnigma rotor1 rotor2 rotor3 reflectorB (0,0,0))
'N'
ghci> encodeChar 'N' (SimpleEnigma rotor1 rotor2 rotor3 reflectorB (0,0,0))
'A'
```

Figure 2.1.4.: Tests to encode a single character

The action previous to encoding a character is rotating the rotors so before being able to encode a full string, this had to be dealt with. checkOffset would rotate the rotors and add to the offsets whilst also checking and making sure that the knock-on positions are reached to rotate the middle and/or left rotors. In Figure 2.1.5.the initial offset (0,5,17) and after checking and adding to the offsets the output is (1,6,18) as both knock-on positions were reached.

```
ghci> checkOffset (SimpleEnigma rotor1 rotor2 rotor3 reflectorB (0,5,17))
SimpleEnigma ("KMFLGDQVZNTOWYHXUSPAIBRCJE",17) ("JDKSIRUXBLHWTMCQGZNPYFVOEA",5) ("DFHJLCPRTXVZNYEIWGAKMUSQOB",22)
 [('A','Y'),('B','R'),('C','U'),('D','H'),('E','Q'),('F','S'),('G','L'),('I','P'),('J','X'),('K','N'),('M','O'),(
'T','Z'),('V','W')] (1,6,18)
```

Figure 2.1.5.: Tests for increasing the offset of an enigma

It is important to note that in this test the rotor strings are also rotated, but they are not representative of the initial offsets. This was handled by setRotor which would recursively rotate a rotor according to the offsets.

To finally encode a full message the function encodeChar had to be called recursively. The function encodes the head of the string while recursively calling itself to encode the remaining tail. On each iteration the function checkOffset was also called in order to rotate the offsets and update the new offsets. Before actually encoding the rotors were prepared by calling setRotor on each. To test that the encoding was working properly traceShow was used to check that the offsets were being updated in each iteration. Figure 2.1.6. shows the rotation of each offset as each character in the string "Here is a test input string." is successfully encoded.

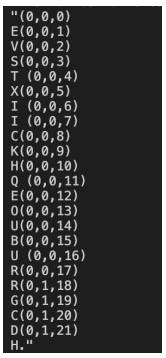

Figure 2.1.6.: Test for simulating the enigma machine

To test the correctness, the resulting string of the encryption was encoded as it should print the previously input string: "Here is a test input string.". A successful example of the testing can be seen in Figure 2.1.7. This shows that the enigma machine was successfully simulated.

```
ghci> encodeMessage "EVSTXIICKHQEOUBURRGCDH" enigma1
"HEREISATESTINPUTSTRING"
```

Figure 2.1.7.: Test of the backward compatibility of enigma encryption

2.2: Longest Menu:

As a first step, the first thing that was develoed an tested was the function cipherEq, which is responsible for finding the cipher equivalent of a character at a certain position. Tests were done with positions 0 and 5 which according to Figure 2.2 should correspond to ‘R’ and ‘Y’ respectively.

```
ghci> cipherEq 0 (zip crib1 message1)
'R'
```

```
ghci> cipherEq 5 (zip crib1 message1)
'Y'
```

Figure 2.2.1.: Tests for finding cipher characters in crib

Once that was done and tested, the implementation of the actual generation of menus could be done. In order to generate a menu the possible chains had to be found in the crib, which was done by the function findsChain. In case possible chains existed, it would return a list with all the indexes of the chain characters and if not it would return Nothing. This was tested as with characters ‘R’ and ‘Z’ which represent both scenarios.

```
ghci> findsChain 'R' (zip crib1 message1)
Just [5,8,11]
```

```
ghci> findsChain 'Z' (zip crib1 message1)
Nothing
```

Figure 2.2.2.: Tests for finding chains in crib

When all possible chains were able to be obtained the next step was to add these to the menus. In the case of having possible chains, the menus were attatched to the value False (representing being not fully explored), in the case of Nothing they were attatched to True. It is important to note that the function also makes sure that the chain added to the menu is not already present in it. Therefore we could have any of the three following scenarios.

```
ghci> findMenu 'R' [0] (zip crib1 message1)
[([0,5],False),([0,8],False),([0,11],False)]
```

```
ghci> findMenu 'X' [9] (zip crib1 message1)
[([9],True)]
```

Figure 2.2.3.: Tests for finding and adding all chains to search menus

The next stage of the programm involved generating all possible menus for a character. Function generateMenus was tested by calling it with the caracter at position 0 which had multiple possible chains.

```
ghci> generateMenu ([[0]],[[]]) (zip crib1 message1)
([],[[],[0,11],[0,8,12,7,15],[0,8,18,16,9],[0,5,21,12,7,15],[0,5,21,18,16,9],[0,8,12,7,4,2,17],[0,8,
12,7,10,16,9],[0,5,21,12,7,4,2,17],[0,5,21,12,7,10,16,9],[0,8,12,7,4,3,6,11],[0,5,21,12,7,4,3,6,11],
[0,5,21,12,7,4,3,6,8,18,16,9],[0,8,12,7,4,3,6,5,21,18,16,9]])
```

Figure 2.2.4.: Test for generating all menus for a given character

The final test for this section was a matter of iterating generateMenus through all initial positions and finding all possible chains in order to find the longest one. With the help of traceShow we can be able to visualize all fully explored menus for each character in the following figure.

Figure 2.2.5.: Test that shows all possible menus in given crib

2.3. Breaking the Enigma:

To break the enigma, characters would've had to be encoded at different positions without necessarily following an incremental order. Therefore a function setOffsets was made to set the enigma offsets and rotors according to the character's position in the menu. This would

use previously implemented functions to take care of the overflows and knock-on positions as seen in Figure 2.3.1.

```
ghci> setOffsets (SimpleEnigma rotor1 rotor2 rotor3 reflectorB (0,0,0)) 50
SimpleEnigma ("CJEKMFLGDQVZNTOWYHXUSPAIBR",17) ("DKSIRUXBLHWTMCQGZNPYFVOEAJ",5) ("BDFHJLCPRTXVZNYEIWGAKMUSQO",22) [('A','Y'),('B','R')
,('C','U'),('D','H'),('E','Q'),('F','S'),('G','L'),('I','P'),('J','X'),('K','N'),('M','O'),('T','Z'),('V','W')] (0,2,24)
```

Figure 2.3.1.: Test of seting the offsets according to a given menu position

The function checkChain takes care of catching chain contradictions. If the list of tuples (possible plugboard) contains one of hte already given guesses, the function would return True indicating that there is indeed a contradiction. This also checks if the possible contradiction is a previous plugboard pair in which case it wouldnot be flagged as a contradiction returning False. An example of this last case can be seen in the third test in Figure 2.3.2.

```
ghci> checkChain [('A','B'),('E','P'),('R','W')]
False
ghci> checkChain [('A','B'),('E','P'),('R','A')]
True
ghci> checkChain [('A','B'),('E','P'),('B','A')]
False
```

Figure 2.3.2.: Tests for finding contradictions in plugboard guesses

To encode a character following a menu, the enigma used has to be set encoded and has to be encoded given a position in the menu. This enigma is set using setOffsets to match the offset it should be at a given menu position. A failed test can be seen in Figure 2.3.3. as the result was expected to be the encoding of ‘A’ to ‘N’ with the enigma configuration seen.

```
ghci> encodeMenuPos 'A' (SimpleEnigma rotor1 rotor2 rotor3 reflectorB (0,0,0)) 0
'F'
```

Figure 2.3.3.: Test on encoding at different menu positions.

In order to find possible plugboards, iterating through all offsets and starting positions had to be made. Examples of it can be seen in Figures 2.3.4 and 2.3.5 where tests checking that every combination is being explored are shown.

```
(0,0,25)
(0,0,0)
(0,0,1)
(0,0,2)
(0,0,3)
(0,0,4)
(0,0,5)
(0,0,6)
(0,0,7)
(0,0,8)
(0,0,9)
(0,0,10)
(0,0,11)
(0,0,12)
(0,0,13)
(0,0,14)
(0,0,15)
(0,0,16)
(0,0,17)
(0,1,18)
(0,1,19)
(0,1,20)
```

```
('N','B')
('N','C')
('N','D')
('N','E')
('N','F')
('N','G')
('N','H')
('N','I')
('N','J')
('N','K')
('N','L')
('N','M')
('N','N')
('N','O')
('N','P')
('N','Q')
('N','R')
('N','S')
('N','T')
('N','U')
('N','V')
('N','W')
('N','X')
```

Figure 2.3.4.: Tests that show the successful iteration of every offset and starting pair

For each of the combinations, a plugboard is tested. Due to the recursiveness and complexity of this section testing these higher-level functions was made very difficult. An attempt was made to implement breaking the enigma, but tests to find the suitable plugboard failed. Final tests to attempt and break the engima failed after evaluating three different crib scenarios.

3. **Critical Reflection:**

Learning Haskell has been a valuable experience. Working with Haskell has allowed me to think on how to solve coding problems in different ways. Having to think harder in order to solve problems has allowed me to improve my creativity in programming. For example, techniques like recursion were not something that I would use very often but after this experience it's something that I found myself using very often.

The introduction to pure functions has also allowed me to use the concept in other programming languages. This allows for easier debugging as I deal with less side effects on the inputs. Combining this with lazy evaluation allows me to produce more efficient and fancier code.

Haskell has also helped me produce more concise programs. Having to deal with multiple functions led me to realize that in order to make more organised code I have to separate them so that they only handle one thing at the time. For example, separating lower level funcitons from the rest makes debugging easier and allows one to test the correctness of each function with different functionalities being separated from each other.

In conclusion, learning Haskell has allowed me to gain valuable experience in functional programming and in new techniques to apply in my day to day programming.