Enigma Machine Report

Jack Marchant

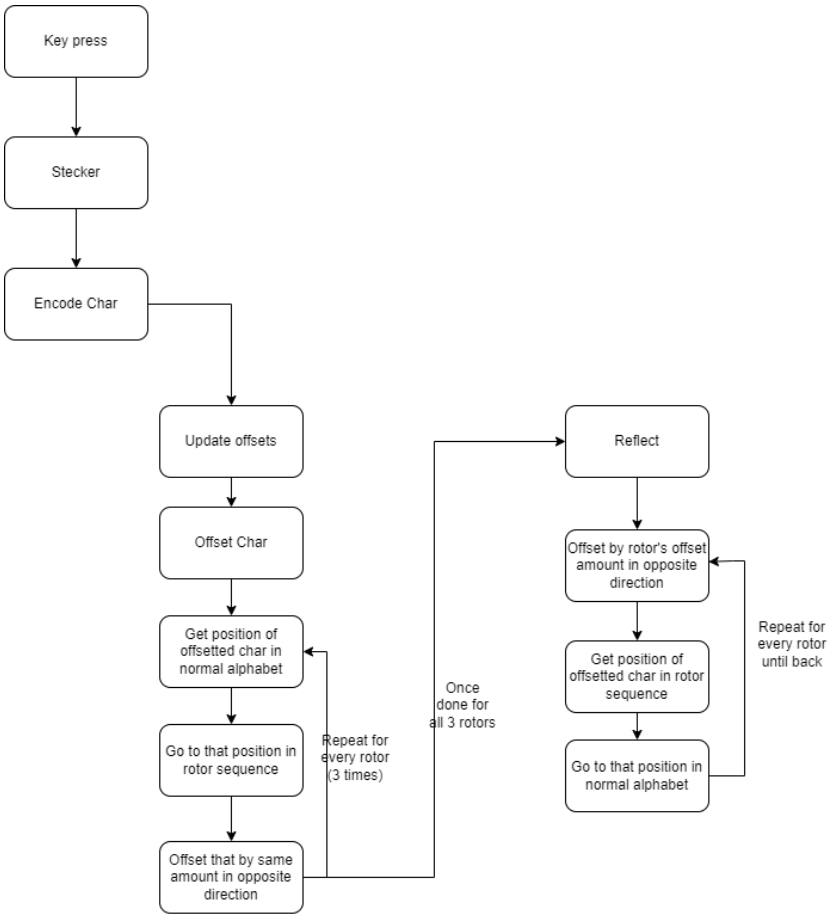
December 7, 2022

1 Enigma

1.1 Design

1.1.1 Explanation of Approach

After reading the background section of the brief and making rough notes on the general idea of how the Enigma machine works, I started to think of how that would translate into code. Naturally, this meant breaking down the main ideas into functions. I started with creating general functions covering the big ideas, such as encodeMessage, and worked through that to see what needed to be done to characters at different steps in order for the function to return the correctly encoded message. A diagram showing an initial basic idea of what needed to be done to each character is shown below, each one initially being planned to be its own function when translated to code. I did this process iteratively, writing down more and more functions that would be needed, until down at basic level functions that could run from basic types, regardless of any input manipulation. Each function is explained in detail in section 1.1.2.



1.1.2 Functions

Explanation of each function and their implementations in order of high level to low level

- encodeMessage
  - Arguments: Raw message (String), Enigma
  - Returns: Encoded message (String)
  - Functions called directly: encodeChar, reflectChar, encodeCharReverse, updateOffsets, encodeMessage (recursion)
  - What it does: Recurses through every letter in a string, fully encoding each one through all 3 rotors both ways (reflected in the middle) with given offsets. Encoded letters are all added onto one string to be returned. Offsets are updated before encoding the characters. Given a steckered enigma, it will also apply the given stecker to the characters both before and after encoding them.
- encodeChar
  - Arguments: Raw character (Char), Rotors and given offsets ([(Rotor, Int)])
  - Returns: Encoded character (Char)
  - Functions called directly: offsetChar, alphaPos, getMappedLetter, encodeChar (recursion)
  - What it does: Takes a character and an array of tuples giving rotors and their given offsets; recurses through rotors encoding the character through each one in turn from right to left.
- encodeCharReverse
  - Arguments: Raw character (Char), Rotors and given offsets ([(Rotor, Int)])
  - Returns: Encoded character (Char)
  - Functions called directly: offsetChar, getMappedLetter, getLetterFromPos, int2let, encodeCharReverse (recursion)
  - What it does: Takes a character and an array of tuples giving rotors and their given offsets, recurses through rotors encoding the character through each one in turn from left to right.
- reflectChar
  - Arguments: Character to be reflected (Char), Reflector to use (Reflector)
  - Returns: Reflected character
  - Functions called directly: reflectChar (Char)
  - What it does: Recurses through list of (Char, Char) tuples that make up the reflector and returns the character in the tuple with the character put in as an argument.
- offsetChar
  - Arguments: Character to offset (Char), Amount to offset it by (Int)
  - Returns: Character offset by given amount
  - Functions called directly: alphaPos, int2let
  - What it does: Converts the character to its position in the alphabet before adding offset, modding by 26 (so 'Z+1' goes to 'A'), and converting back to a character to be returned.
- updateOffsets
  - Arguments: Current offsets (Offsets), Middle rotor (Rotor), Right rotor (Rotor)
  - Returns: Updated offsets (Offsets)
  - Functions called directly: *NONE*
  - What it does: Uses guard statements to test if any of the rotors are at their knock on positions and updates offsets accordingly.

- getMappedLetter
  - Arguments: Rotor, Position to look for in rotor (Int)
  - Returns: Letter at given position on rotor (Char)
  - Functions called directly: *NONE*
  - What it does: Uses the Prelude '!!' in order to find the letter at the given position in the given rotors sequence.
- getLetterFromPos
  - Arguments: Rotor sequence (String), Character to search for (Char)
  - Returns: Index at which the letter is found on the rotor (Char)
  - Functions called directly: *NONE*
  - What it does: Uses the Prelude 'elemIndex' function to return either the position of the letter (if it is present in the sequence) or -1 (if it is not).
- int2let
  - Arguments: Integer to convert to letter (Int)
  - Returns: Letter found at index in alphabet of input
  - Functions called directly: *NONE*
  - What it does: Uses prelude 'ord' to add a given number to 'A' and convert to char.

1.1.3 Data Structure Definitions

- Rotor = (String, Int). Tuple with rotor sequence and knock on position.
- Reflector = [(Char, Char)]. List of tuples mapping every character to another character.
- Offsets = (Int, Int, Int). Three part tuple with offsets for each of the rotors.
- Stecker = [(Char, Char)]. List of tuples mapping certain characters to other characters.

1.2 Testing

1.2.1 General Testing

As with all functional programs, once the skeleton of the design was done and I had an idea of what functions I would need to implement and what inputs each of those functions would take, I started to implement the functions from the bottom of the tree upwards. For each function, I looked at what it was intended to do in the greater scheme of the entire program, and from there could decide on what sensible test cases would look like, where, upon passing, I could be confident that the function would always work in the intended way for any input that it may be given. Once I was happy the function had passed, I could then move on to higher level functions.

1.2.2 Function Specific Testing

- int2let
  - Rationale behind testing: It was fairly straightforward to work out by hand what this function should be returning if working correctly. Given a number, it should return the capital letter at that index in the alphabet. I could do this by hand and compare what the function was returning with what it should be returning to test if it was correct. Because of the way the function is called in the greater program, the input will always be restricted between 0-25, and therefore edge cases of different inputs outside of this restriction did not need to be tested.

Input	Expected Output	Actual Output
0	The first capital letter of the alphabet ('A')	'A'
1	The second capital letter of the alphabet ('B')	'B'
25	The last capital letter of the alphabet ('Z')	'Z'

- getMappedLetter
  - Rationale behind testing: Testing this function just included choosing any position (0-25) on a given rotor and testing that the function returned the number at the correct index. Similarly to int2let, this function would always be passed a number in the restrictions (0-25) based on how it is called in the higher level functions, so as long as it returned the correct Char for the rotor based on the input it was correct.
Input	Expected Output	Actual Output
0, rotor1	The first character on rotor1 ('E')	'E'
25, rotor1	The last character on rotor1 ('J')	'J'
10, rotor3	The 11th character on rotor3 ('X')	'X'
16, rotor5	The 17th character on rotor5 ('A')	'A'

- getLetterFromPos
  - Rationale behind testing: Testing this function was similar to testing getMappedLetter. Given a rotor sequence and a character, it should return the position of that character in that rotor sequence. As I could count any letter's position in a given rotor sequence, I could work out by hand what the output should be and compare that to what the function was returning.
Input	Expected Output	Actual Output
'E', rotor1	The index at which 'E' is on rotor1 (0)	0
'J', rotor1	The index at which 'J' is on rotor1 (25)	25
'X', rotor3	The index at which 'X' is on rotor3 (10)	10
'A', rotor5	The index at which 'A' is on rotor5 (16)	16

- updateOffsets
  - Rationale behind testing: Upon giving initial offsets to the function, it should update them and return offsets based on the right rotor advancing by one, and the others in turn based off knock on points being hit. To make sure there was no issues with the nesting of the conditional statements, it was important to test an edge case where the right and middle rotors are both at their knock ons, and therefore all rotors should be updated at the same time. Furthermore, no offsets should go beyond 25, so testing that all of the rotors reset to 0 after being knocked on from 25 was important.
Input	Expected Output (when Middle=rotor2 and Right=rotor1)	Actual Output
(0,0,0)	RR knocked on by 1, all others unchanged	(0,0,1)
(0,0,17)	RR and MR knocked on by 1, LR unchanged	(0,1,18)
(0,4,17)	All three rotors knocked on by 1	(1, 5, 18)
(0,0,25)	RR back to 0, all others unchanged	(0,0,0)
(0,25,17)	RR knocked on by 1, MR back to 0, LR unchanged	(0,0,18)
(25,4,17)	RR and MR knocked on by 1, LR back to	(0,5,18)

- offsetChar
  - Rationale behind testing: If implemented correctly, offsetChar should be able to offset a capital letter in the alphabet forwards by a given amount, or backwards if the given amount is negative. Edge cases for this including testing if the function looped round if given a large offset and/or a letter late in the alphabet, returning a capital letter rather than another un-expected character; as well as being able to offset with negative numbers, going back in the alphabet rather than forwards (and looping in a similar way).

–

Input	Expected Output	Actual Output
'A', 1	Capital letter 1 AFTER 'A' in alphabet	'B'
'M', -3	Capital letter 3 BEFORE 'M' in alphabet	'J'
'Z', 1	The capital letter at the START of the alphabet (loop)	'A'
'A', -1	The capital letter at the END of the alphabet (reverse loop)	'Z'

- reflectChar
  - Rationale behind testing: Simply, giving a character as an input to the function should return its partner in the given reflector. During testing, I gave both partners as input for multiple pairs, and expected to get returned each ones partner in the reflector.
  -

Input	Expected Output (using reflectorB)	Actual Output
'A'	'A's partner in the reflector ('Y')	'Y'
'Y'	'A' ('A' reflected to 'Y')	'A'
'K'	'K's partner in the reflector ('N')	'N'
'N'	'K' ('K' reflected to 'N')	'K'

- encodeChar
  - Rationale behind testing: I worked through encoding characters by hand, and following a character the entire way through three rotors, choosing a different trio of rotors every time, to see what it should look like after being passed through each one, initially without any offset, and then with differing offsets on each rotors. With this, I got a number of input letters, and the letters they should get mapped to after getting passed through all three rotors. Also, to check the offset was being applied, I inputted the same letters and same rotors but with different offsets to see if they returned different results.
  -

Input	Expected Output	Actual Output
'A' (without offset)	'G'	'G'
'A' (with offset)	'D'	'D'
'Z' (without offset)	'D'	'D'
'Z' (with offset)	'X'	'X'

- encodeCharReverse
  - Rationale behind testing: Similarly to encodeChar, I worked through encoding the characters by hand, with the only difference being this time I was encoding the other way (from left to right). I also got a list of characters and what they should output after going through all three rotors in the reverse direction with and without offsets. With the function returning matching outputs, I could be satisfied that the function was working.
  -

Input	Expected Output	Actual Output
'A' (without offset)	'D'	'D'
'A' (with offset)	'K'	'K'
'Z' (without offset)	'A'	'A'
'Z' (with offset)	'N'	'N'

- encodeMessage
  - Rationale behind testing: There were a few different checkpoints that would suggest this function worked as intended with all of it's functions. Firstly, the plain string and the encoded string should have the same amount of letters, this would suggest that every character is being encoded one by one as intended. Secondly, to check that the offsets were getting updated after each key press, inputting multiple of the same letter should not encode them all to the same letter, instead each one should be different from the last. Thirdly, with the enigma being symmetric and reversible, taking an output from the function, and putting it back into the function with the same enigma setup (rotors, initial offsets, reflector, and stecker) should return the original message. Finally, to check that all offsets were definitely being updated correctly, I input a long input string- if the string was still being encoded correctly at the end, it would confirm that all offsets on all rotors are updating/applying correctly.

–

Input	Expected Output	Actual Output
"HELLOWORLD"	Same amount of letters	"MFNCZBBFZM"
"AAAAAA"	Not all same letter	"FTZMGI"
"MFNCZBBFZM"	Initial message encoded with same enigma setup	"HELLOWORLD"
Alice in Wonderland scene	End of string correct	"...FELLPASTIT"

2 Longest Menu

2.1 Design

2.1.1 Explanation of Approach

Similarly to how I approached the design of the enigma machine, I initially sketched out what the flow would look like for working out the longest menu. From this point, I was able to break it into functions and begin to implement them. Each function is explained in detail in section 2.1.2.

2.1.2 Functions

Explanation of each function and their implementations in order of high level to low level

- longestMenu
  - Arguments: Crib
  - Returns: Longest menu possible from crib (Menu)
  - Functions called directly: longetsMenuFromPoint, longest
  - What it does: Calls longestMenuFromPoint with the inital starting point from the the end of the crib. With recursion, this returns a list of the longest menus from each point, which I then call 'longest' on in order to get the longest menus from them, and head to get just one meny of the longest length.
- longestMenuFromPoint
  - Arguments: Crib, Starting point (Int)
  - Returns: A list of menus, 1 of the longest from each starting point in the crib
  - Functions called directly: zipWithIndexes, getBranches, convertToMenuList, longest, longestMenuFromPoint (recursion)
  - What it does: Firstly, zips the crib to give each char pair an index making them more managable to work with. Then, starting from the point given, get (one of) the longest menu's possible from that point by calling getBranches. Once done, append to a list and call again, from the starting point one before the last. Once it is starting from 0, append that longest menu and return the list of longest menus from each point.
- getBranches
  - Arguments: The crib to search through (IndexedCribList), the element in the list at which to search from (IndexedCrib)
  - Returns: On its own- the possible branches that the element to search from can go to in the given list of indexed cribs (IndexedCribList). When recursed with exploreSubBranches however- a list of a list of all indexed crib items possible to go down, branching when necessary ([IndexedCribList]).
  - Functions called directly: exploreSubBranches (recursion)
  - What it does: Gets all possible branches the inputted IndexedCrib can go to based off of its cipher character, and the other elements plain characters. It then explores all of those possible branches by passing them through as an input to the exploreSubBranhces function with a filtered version of the IndexedCribList it was given to filter out itself in order to ensure it cannot go back to itself again. Finally, it maps the current IndexedCrib to the head of the list of all sub branches, meaning that for every branch that is possible to reach, all sub branches will be explored until there is no more elements left to add to the menu.

- exploreSubBranches
  - Arguments: The crib to search through (IndexedCribList), the list of elements in the first list given as input that it needs to explore (IndexedCribList)
  - Returns: A list of IndexedCribLists, where the possible branches have been concatenated with an exploration of the sub branches ([IndexedCribList]).
  - Functions called directly: getBranches (recursion), exploreSubBranches (recursion)
  - What it does: Calls get branches for every possible sub branch that is given to it through the getBranches method. Through recursion of these two methods calling eachother whenever new branches are found and need to be explored, the concatenation of the results in the method, and the mapping to the heads of the lists in getBranches; the two methods should work together in order to return a list every possible menu, branching at appropriate times and continuing until there is nowhere left to go.
- convertToMenuList
  - Arguments: The list of elements returned from the getBranches method ([IndexedCribList]).
  - Returns: A list of Menus from the list of IndexedCribLists ([Menu]).
  - Functions called directly: convertToMenu (simple function)
  - What it does: Recurses through the list of IndexedCribLists and converts each one to a Menu using a simple function called convertToMenu. Convert to menu simply just discards everything in the IndexedCrib that isn't wanted a returns a Menu using just the index part of the IndexedCrib.

2.1.3 Data Structure Definitions

- Menu = [Int]. List of numbers, each one being a position in the original message/cipher pair that can be reached from its neighbour before in the list.
- Crib = [(Char, Char)]. A zipped array of character tuples, with each first one being the plain character, and the second its matching cipher character.
- IndexedCrib = (Int, (Char, Char)). An index added on to the orignal character tuple as the first element in a new surrounding tuple. Makes working with the crib more manageable, especially when removing elements of the list and still wanting to know the index of the Char tuple within the grander scheme of the original crib.
- IndexedCribList = [IndexedCrib]. A list of IndexedCribs. Result of zipping up the crib with numbers starting from 0.

2.2 Testing

2.2.1 General Testing

My general approach to testing was very similar to how I approached the testing of the enigma portion of the project. I implemented my functions from the bottom of the tree upwards (low level to high level), and only moved on to the function above once I was happy the current one I was working on would work for any case it could possibly be given. Similarly, I created a number of tests for each one that would satisfy me of this.

2.2.2 Function Specific Testing

- longest
  - Rationale behind testing: Testing this function simply included giving it an input of lists, and seeing if it would return the lists with the longest elements. The cases I needed to check were would it just return the one if there was only one list in the list longer than the rest. Also, would it return more than one if there was more than one list in the lists of the shared longest length.

–

Input	Expected Output	Actual Output
[[1,2,3],[1,2],[1,2,3,4]]	Just one longest list	[[1,2,3,4]]
[[1,2],[3,4,5,6],[7,8,9,10]]	A list of two longest lists	[[3,4,5,6],[7,8,9,10]]

- convertToMenuList
  - Rationale behind testing: Given a List of IndexedCribLists, it should return a list of the same length, and inside those lists, instead of each element being an IndexedCrib, it should just be the index part of the IndexedCrib.
  -

Input	Expected Output	Actual Output
[zipWithIndexes crib]	List of a list with numbers 0...crib length	[[0...22]]
[[zipWithIndexes crib], [zipWithIndexes crib]]	Two lists with numbers 0..crib length	[[0...22], [0...22]]

- getBranches
  - Rationale behind testing: I first tested this function without the recursive call to exploreSubBranches to see if it was working as I intended it to. Without the recursive call, its function is to get a list of the possible branches given a starting point and points it can go to.
  -

Input	Expected Output	Actual Output
zipWithIndexes crib, first element without recursion	List of all crib items that first element can go to	[(5,('R','Y')), (8,('R','S')), (11,('R','F'))]
zipWithIndexes crib, first element with recursion	All branched/explored lists	[[(0,('W','R'))..]..[..(11,('R','F'))]]

- exploreSubBranches
  - Rationale behind testing: As the getBranches function was already written, the testing of this function was all about putting it together with the getBranches function. Together, they should recurse on eachother in order to get all the possible lists. To test they were getting everything, I followed the cribs through by hand, and made sure that all branches were being explored from each point.
  -

Input	Expected Output	Actual Output
zipWithIndexes crib, first element with recursion	List of IndexedCribLists starting with a list with just the starting element and then the rest of the list	[[(0, ('W','R'))], ... [..(11,('R','F'))]]

- longestMenuFromPoint
  - Rationale behind testing: In essence, as long as getBranches (and thus exploreSubBranches) are working as intended, the only real testing this function needs is checking that it is applying the basic functions to the results correctly, and recursing to the end. With this in mind, I ran tests with known crib inputs and known longest menus for each one.
  -

Input	Expected Output	Actual Output
zipped version of (crib1, message1), 0	Just the longest menu from the first element	[0,5,21,...,16,9]
zipped version of (crib1, message1), length	A list of each longest menu from every start point	[[22,1,...9]...[0,5,...12]]

- longestMenu
  - Rationale behind testing: With longestMenuFromPoint returning a list of the longest menu from every point, this function just needed to get the longest one from that list.
  -

Input	Expected Output	Actual Output
zipped version of (crib1, message1)	The longest menu from the crib/message pair	[13,14,19...,16,9]

3 Bombe

3.1 Design

3.1.1 Explanation of Approach

The bombe had a more challenging design than the previous two sections of the project. There was a lot more conditional moving from function to function based on what the previous had returned. For this reason, I felt a figure (included below) was necessary as an aide to view the flow all in one place, and be to able to follow it through. Simply put, the design firstly involves assuming an enigma configuration, and initial offsets for each of the three rotors. Once these are in place, you also assume an initial plugboard. With these, the algorithm is now prepped to do the hard work, which is following through the longest menu, and adding the steckers that would be necessary on the plugboard in order for the configuration to be correct. In code, this part will be one recursive function that passes through to itself the updated list of steckers that have been discovered already, and the menu that still needs to be followed. If there is any clashes with the stecker pairs added to the list, meaning one letter steckers to more than one other letter, the solution is incorrect. Upon this happening, the function would return nothing, prompting the parent function to increment the plugboard by one, and once more call the recursive plugboard testing function. If this is done 25 times, and the plugboard is back at its initial configuration, it should go back one more level, and increment the offets by one before starting all over again. If any working configurations are found (no clashes), the configuration will be returned. However, if the offsets end up incremented all the way back around to the initial chosen offsets, it means every configuration has been tried, and therefore 'Nothing' will be returned.