Functional Programming Design Case Study Assignment

This is the report and critical analysis for the COM2108 Functional Programming Design Case Study Assignment. The task was themed around the Enigma machine used in World War II, as well as the Bombe that was used to help decipher it.

The document is split into four parts:

1. Simulation of the Enigma
2. Finding the longest Menu
3. Simulation of the Bombe
4. Critical reflection

Part One: Simulating the Enigma

First, we need to think about the data types of the rotors, the reflectors, the offsets and the stecker.

- Rotor – Should be a pair, the first item is a String (the wirings) and the second item is an Int (knock-on position).
- Reflector – Should be an array of pairs, the pairs will be of type Char for both items.
- Stecker – Same as reflector, array of Char pairs.
- Offsets – Should be a tuple of three Ints that represent the offsets of each rotor left to right.

The design process of the Enigma machine is quite straightforward from a functional programming standpoint, since the Enigma itself can be broken down into multiple stages which will require a function each. We can break down the Enigma into these stages:

1. Change the offsets
2. If using Steckerboard then stecker
3. Encrypt through right rotor
4. Encrypt through middle rotor
5. Encrypt through left rotor
6. Reflect
7. Encrypt through left rotor
8. Encrypt through middle rotor
9. Encrypt through right rotor
10. If using Steckerboard then stecker

This is the process to encrypt a single character, however some of these functions will require others i.e., encrypting through a rotor will require a function that changes the offset of a character before it is put through the wirings. However, the processes that can be written using a single function are reflection and steckering, so they will be the best place to start.

The reflect and stecker functions can be seen as a simple search function on an array of pairs: for a pair (a, b) if a is found then return b and vice-versa, if not then search the rest of the list. Steckering however will differ because if a pair is not found then it should return the character that was input whereas reflecting should return an error if an item is not found as each letter should have a corresponding reflection.

Now that we've done those, we need to write functions to change the offsets before we think about encrypting a character with a rotor. We need a function that simulates a keypress by taking the Offset data type (x, y, z), the left, right, and middle rotor and has three if statements:

- If both the right rotor and the middle rotor will go past their knock-on position, then increase all three offsets
- If just the right rotor is going past its knock-on position, then increase just the middle and right offsets
- If none apply then just increase the right offset, since it increases with every keypress

This function should be used in the final high level "encodeMessage" function, which is where we will be simulating each keypress.

Now we can think about encrypting a single character – when we think of a character going through a rotor, we can think of it like reflection between the alphabet and the wirings of the rotor, if we zip these together then we can encrypt a single character that hasn't been offset. Since our previous search functions only take either type "Reflector" or "Stecker" we will need to make one for our zipped list [(Char, Char)], but other than that the function should be the same as reflect. However, if we map alphabet to wirings that may work going forward – but after reflection the character is encrypted inversely (i.e., check the rotor wirings against the alphabet), so we will need to swap the pairs around.

Before putting it all together we need to account for offset. By looking at the diagram provided in the brief we can see that a rotor in a different position is the equivalent of shifting the input letter by a certain number of positions, like a Caesar cipher, and then searching for that character's pair in the wirings before shifting it back the alphabet by the same number of positions. Figure 1 shows a drawing of the letter A being offset and based through rotor 1, the middle part where the lines are drawn in the second example represents inside the rotor.

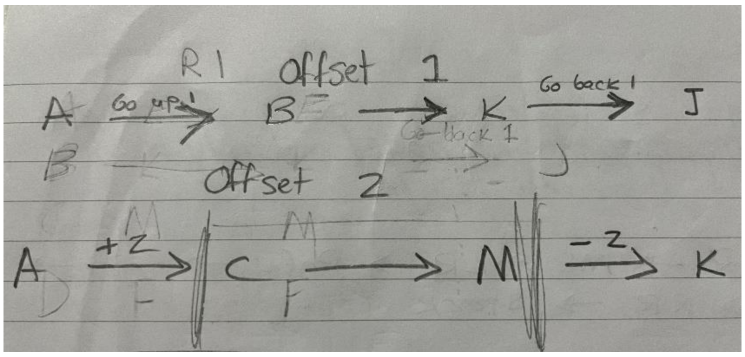


Figure 1: Offset and encryption process

We can easily offset a character by turning it into an Integer, adding the offset, modulo 26, and then finally turning it back into the character we want.

A character will be passed through three rotors both before and after reflection, so we could have a function that passes a character forward through a rotor and a function that passes a character backward through a rotor. Once we have these, we can write a function that does all three rotors pre-reflection using the forward propagation and another one post-reflection using the backward propagation.

Our single forward encryption through one rotor should take the character, rotor and offset for said rotor. It should search for the offset char in the rotor-alphabet pair array and then offset it back by negating the offset that was passed in. We can do the same thing for going backwards but our rotor-alphabet array is flipped.

Now we can chain the forward propagation function three times to simulate a character going through all three rotors before reflection and chain the backwards propagation function three times also to simulate a character going through all three rotors after reflection. These will be two separate functions taking a character, all three rotors as well as their offsets as (x, y, z).

Now we can finally encrypt a single character fully, to encrypt normally we just need to take the character, all three rotors, the offsets and a reflector. For a steckered Enigma we will add another parameter for the steckerboard.

For the standard 3-rotored Enigma the order of function evaluation is:

1. Perform "pre-reflection" encryption on an input character
2. Reflect the output with our reflect function
3. Perform "post-reflection" on the output

With the steckerboard we can write a second function:

1. Stecker the input character with our stecker function
2. Perform the full encryption from before with the output of the stecker
3. Stecker the output of the encryption

Now we look towards the high-level function "encodeMessage": we can encrypt a message by performing our final function on a character and recursing on the rest of the string using pattern matching. We need to make separate cases for steckered Enigma machines.

When we recurse, this is where we can alter the offset of the input Enigma parameter using the keypress for every call of the function, simulating our keypresses.

However, there is one final thing before we finish with "encodeMessage". We must sanitise our input string, so for now let's have our full message encryption under a different name that we can call on a sanitised string for our final function.

A message must be upper case and cannot contain any numbers or special characters besides space, so a simple guard clause will suffice:

- If character is lower case, make it upper case and recurse on the rest of the string
- If it is an upper-case letter or a space, ignore and recurse
- Anything else (i.e., disallowed special characters): join an empty string and recurse

Now we can define our final "encodeMessage": it will cleanse the message first and then perform the recursive encryption.

Testing:

After implementation, we have the following functions (In order of implementation):

- reflect

- stecker
- keypress
- rotorAlphaZip
- searchWirings
- offsetChar
- rotoriseForward
- rotoriseBackward
- preReflection
- postReflection
- encryptWithReflection
- encryptWithStecker
- encryptUncleansedMessage
- cleanse
- encodeMessage

There were also helper functions such as:

- mod26 – so that I only needed one input rather than writing x `mod` 26 every time.
- pos2char – turns an alphabet position into a character, inverse of “alphaPos” given in the code.

<u>reflect</u>

Since a predefined “reflectorB” was given to us in the code, I used this to test the reflect function for a single pairing going both ways:

Input	**Output**
‘A’	‘Y’
‘Y’	‘A’

<u>stecker</u>

Similar for reflect using the “plugboard” defined in the Main file, this time testing for a letter with no match:

Input	**Output**
‘F’	‘T’
‘T’	‘F’
‘B’	‘B’

<u>keypress</u>

For this function we need to test edge cases (e.g. (0,0,25) – make sure it wraps around) and cases where offsets will change. I implemented it as described in the brief (i.e., middle rotor changes when right rotor goes from 17 to 18).

Input	**Output**
rotor3 rotor2 rotor1 (0,0,25)	(0,0,0)
rotor3 rotor2 rotor1 (0,25,25)	(0,25,0)

rotor3 rotor2 rotor1 (25,25,25)	(25,25,0)
rotor3 rotor2 rotor1 (0,0,17)	(0,1,18)
rotor3 rotor2 rotor1 (0,5,17)	(1,6,18)
rotor3 rotor2 rotor1 (1,6,18)	(1,6,19)
rotor3 rotor2 rotor1 (25,5,17)	(0,6,18)
rotor3 rotor2 rotor1 (0,25,17)	(0,0,18)

rotorAlphaZip

For this function there was only really one case to test, just needed to make sure everything was paired.

Input	**Output**
rotor1	[('A','E'),('B','K'),('C','M'),('D','F'),('E','L'),('F','G'),('G','D'),('H','Q'),('I','V'),('J','Z'),('K','N'),('L','T'),('M','O'),('N','W'),('O','Y'),('P','H'),('Q','X'),('R','U'),('S','S'),('T','P'),('U','A'),('V','I'),('W','B'),('X','R'),('Y','C'),('Z','J')]
rotor2	[('A','A'),('B','J'),('C','D'),('D','K'),('E','S'),('F','I'),('G','R'),('H','U'),('I','X'),('J','B'),('K','L'),('L','H'),('M','W'),('N','T'),('O','M'),('P','C'),('Q','Q'),('R','G'),('S','Z'),('T','N'),('U','P'),('V','Y'),('W','F'),('X','V'),('Y','O'),('Z','E')]

searchWirings

Again, this works like reflect and stecker except it goes one-way (i.e., A may output E, but E won't output A), these results were done by zipping rotor1 and the alphabet together.

Input	**Output**
'A' (rotorAlphaZip rotor1)	'E'
'E' (rotorAlphaZip rotor1)	'L'

offsetChar

Input	**Output**
'A' 1	'B'
'A' (-1)	'Z'
'B' 4	'F'
'B' (-17)	'K'

rotoriseForward

The first two tests were based on the diagrams of the rotors in the brief, the third based off the design drawings.

Input	**Output**
'A' rotor1 0	'E'
'A' rotor1 1	'J'
'A' rotor1 2	'K'
'K' rotor2 10	'F'

‘J’ rotor3 5	‘T’

rotoriseBackward

We can prove that this function works as expected by using the output of our tests from rotoriseForward:

Input	**Output**
‘E’ rotor1 0	‘A’
‘J’ rotor1 1	‘A’
‘K’ rotor1 2	‘A’
‘F’ rotor2 10	‘K’
‘T’ rotor3 5	‘J’

preReflection

Input	**Output**
‘A’ rotor1 rotor2 rotor3 (0,0,0)	‘G’
‘A’ rotor1 rotor2 rotor3 (0,1,17)	‘W’
‘A’ rotor3 rotor2 rotor1 (0,1,22)	‘F’
‘W’ rotor2 rotor1 rotor3 (1,1,1)	‘Q’

postReflection

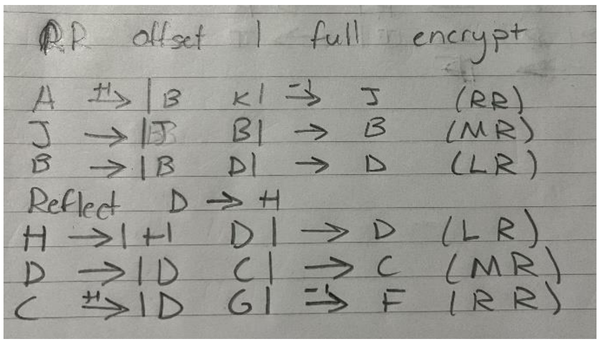
Even without reflecting we can use the output from preReflection to get the input:

Input	**Output**
‘G’ rotor1 rotor2 rotor3 (0,0,0)	‘A’
‘W’ rotor1 rotor2 rotor3 (0,1,17)	‘A’
‘F’ rotor3 rotor2 rotor1 (0,1,22)	‘A’
‘Q’ rotor2 rotor1 rotor3 (1,1,1)	‘W’

encryptWithReflection

Input	**Output**
‘A’ rotor1 rotor2 rotor3 (0,0,0) reflectorB	‘N’
‘A’ rotor1 rotor2 rotor3 (0,0,1) reflectorB	‘F’
‘A’ rotor1 rotor2 rotor3 (1,1,1) reflectorB	‘Y’
‘W’ rotor3 rotor2 rotor1 (1,3,15) reflectorB	‘Y’
‘L’ rotor3 rotor1 rotor2 (2,2,2) reflectorB	‘P’
‘%’ rotor1 rotor2 rotor3 (0,0,0) reflectorB	‘Z’
‘w’ rotor3 rotor2 rotor1 (1,3,15) reflectorB	‘O’

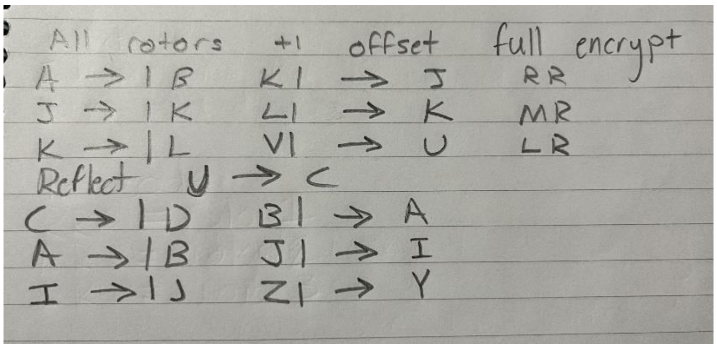
The first test can be seen in the brief, the second two were checked by hand shown by Figure 2 and 3. Once I knew the process was correct, I checked a handful of others. Inputting special characters has a weird effect and ends up being encrypted itself as well as lower case characters being encrypted differently to their uppercase counterparts. However, the string will be cleansed before full encryption, so it is nothing to worry about.



RR offset 1 full encrypt

A $\xrightarrow{+1}$ |B K| $\xrightarrow{-1}$ J (RR)
J -> |J B| -> B (MR)
B -> |B D| -> D (LR)
Reflect D -> H
H -> |H D| -> D (LR)
D -> |D C| -> C (MR)
C $\xrightarrow{+1}$ |D G| $\xrightarrow{-1}$ F (RR)

Figure 2: Test 2



All rotors +1 offset full encrypt

A -> |B K| -> J RR
J -> |K L| -> K MR
K -> |L V| -> U LR
Reflect U -> C
C -> |D B| -> A
A -> |B J| -> I
I -> |J Z| -> Y

Figure 3: Test 3

For clarification: the | | represents the inside of the rotor so for figure 2 A will get offset to B which is wired to K and the offset is taken off again to J and the process continues with no offset afterwards until it gets back to the right rotor.

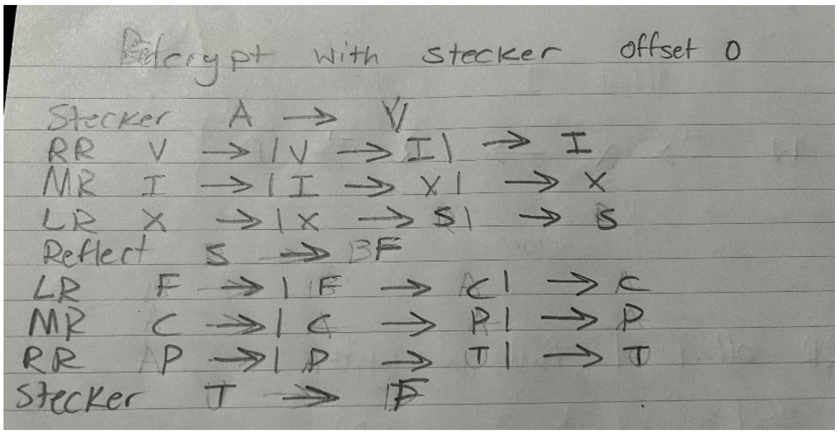
<u>encryptWithStecker</u>

Input	**Output**
‘A’ rotor1 rotor2 rotor3 (0,0,0) reflectorB plugboard	‘F’
‘A’ rotor3 rotor2 rotor1 (0,1,1) reflectorB plugboard	‘J’
‘A’ rotor1 rotor2 rotor3 (25,25,25) reflectorB plugboard	‘I’
‘W’ rotor3 rotor2 rotor1 (1,3,15) reflectorB plugboard	‘R’
‘L’ rotor3 rotor1 rotor2 (2,2,2) reflectorB plugboard	‘P’
‘%’ rotor1 rotor2 rotor3 (0,0,0) reflectorB plugboard	‘H’
‘w’ rotor3 rotor2 rotor1 (1, 3, 15) reflectorB plugboard	‘R’

Some things to note:

- The test using ‘L’ generates the same result as without steckering because there is no pair for L.
- Using a lowercase ‘w’ will generate the same result as uppercase ‘W’ because the stecker search function uses toUpper when searching.

I also handwrote the process for test 1:



Encrypt with stecker offset 0

Stecker A → V
RR V → |V → I| → I
MR I → |I → X| → X
LR X → |X → S| → S
Reflect S → F
LR F → |F → C| → C
MR C → |C → P| → P
RR P → |P → T| → T
Stecker T → F

Figure 4: Test 1

<u>encryptUncleansedMessage</u>

For the Enigma parameter I used both enigma1 and enigma2 defined as follows in Main.hs:

- `enigma1 = (SimpleEnigma rotor1 rotor2 rotor3 reflectorB (0,0,25))`
- `enigma2 = (SteckeredEnigma rotor1 rotor2 rotor3 reflectorB (0,0,25) plugboard)`

Where:

- rotor1 is the **right** rotor
- rotor2 is the **middle** rotor
- rotor3 is the **left** rotor
- plugboard and reflectorB are as defined in Main.hs
- (0, 0, 25) are the initial offsets

Input	**Output**
"THISISATESTWITHOUTSPACES" enigma1	"VMQDXIICKHQTYBLFFVELQXRD"
"THIS IS A TEST WITHOUT SPACES" enigma1	"VMQD XI I CKHQ TYBLFFV ELQXRD" (*)
"THISISATESTWITHOUTSPACES" enigma2	"BFGUXXSNWZOAPISTAIELKIRU"

"THIS IS A TEST WITHOUT SPACES" enigma2	"BFGU XX S NWZO APISTAI ELKIRU"
"thisisatestwithoutspaces" enigma1	"YKLKZWCXEPILNJETJKVXCDJQ"
"this is a test without spaces" enigma1	"YKLK ZW C XEPI LNJETJK VXCDJQ"
“this Is a T3ST witHout sp4CES” enigma1	"YKLK XW C CVHQ LNJLTJK VXDXRD"
“l0ts of %%%% sp4cial char5” enigma2	"XREW FL HDQP WWOCYJS DJIEH"

(*) – because spaces were not used in the actual enigma machine, the implementation skips a keypress if a space is encountered

cleanse

Input	Output
“thisisatestwithoutspaces”	"THISISATESTWITHOUTSPACES"
“this is a test without spaces”	"THIS IS A TEST WITHOUT SPACES"
“this Is a T3ST witHout sp4CES”	"THIS IS A TST WITHOUT SPCES"
“l0ts of %%%% sp4cial char5”	"LTS OF SPCIAL CHAR" (*)
"BIG SPACE"	"BIG SPACE"

The cleanse function is designed to skip over spaces so will not shorten large gaps, however the encryption is designed to not count spaces as a keypress so the message should be the same as if there was only 1 or even no spaces there.

encodeMessage

Input	Output
"thisisatestwithoutspaces" enigma1	"VMQDXIICKHQTYBLFFVELQXRD"
“this is a test without spaces” enigma1	"VMQD XI I CKHQ TYBLFFV ELQXRD"
“this Is a T3ST witHout sp4CES” enigma1	"VMQD XI I CUM NEDRITC GNIXY"
"THIS IS A TST WITHOUT SPCES" enigma1	"VMQD XI I CUM NEDRITC GNIXY"
“l0ts of %%%% sp4cial char5” enigma2	"PVR QD XNFVJG LRLQ"
"LTS OF SPCIAL CHAR" enigma2	"PVR QD XNFVJG LRLQ"
"BIG SPACE" enigma2	"TSI UNQGZ"
“BIG SPACE” enigma2	"TSI UNQGZ"

The additional test string provided in Blackboard (too large to fit in table):

"ALICEWASBEGINNINGTOGETVERYTIREDOFSITTINGBYHERSISTERONTHEBAN
KANDOFHAVINGNOTHINGTODOONCEORTWICESHEHADPEEPEDINTOTHEBOOK
HERSISTERWASREADINGBUTITHADNOPICTURESORCONVERSATIONSINITAND
WHATISTHEUSEOFABOOKTHOUGHTALICEWITHOUTPICTURESORCONVERSATI
ONSSOSHEWASCONSIDERINGINHEROWNMINDASWELLASSHECOULDFORTHEH
OTDAYMADEHERFEELVERYSLEEPYANDSTUPIDWHETHERTHEPLEASUREOFM
AKINGADAISYCHAINWOULDBEWORTHTHETROUBLEOFGETTINGUPANDPICKI
NGTHEDAISIESWHENSUDDENLYAWHITERABBITWITHPINKEYESRANCLOSEBY

HERTHEREWASNOTHINGSOVERYREMARKABLEINTHATNORDIDALICETHINKIT
SOVERYMUCHOUTOFTHEWAYTOHEARTHERABBITSAYTOITSELFOHDEAROHD
EARISHALLBELATEWHENSHETHOUGHTITOVERAFTERWARDSITOCCURREDTO
HERTHATSHEOUGHTTOHAVEWONDEREDATTHISBUTATTHETIMEITALLSEEME
DQUITENATURALBUTWHENTHERABBITACTUALLYTOOKAWATCHOUTOFITSW
AISTCOATPOCKETANDLOOKEDATITANDTHENHURRIEDONALICESTARTEDTOH
ERFEETFORITFLASHEDACROSSHERMINDTHATSHEHADNEVERBEFORESEENAR
ABBITWITHEITHERAWAISTCOATPOCKETORAWATCHTOTAKEOUTOFITANDBU
RNINGWITHCURIOSITYSHERANACROSSTHEFIELDAFTERITANDFORTUNATELY
WASJUSTINTIMETOSEEITPOPDOWNALARGERABBITHOLEUNDERTHEHEDGEIN
ANOTHERMOMENTDOWNWENTALICEAFTERITNEVERONCECONSIDERINGHOW
INTHEWORLDSHEWASTOGETOUTAGAINTHERABBITHOLEWENTSTRAIGHTONL
IKEATUNNELFORSOMEWAYANDTHENDIPPEDSUDDENLYDOWNSOSUDDENLY
THATALICEHADNOTAMOMENTTOTHINKABOUTSTOPPINGHERSELFBEFORESH
EFOUNDHERSELFFALLINGDOWNAVERYDEEPWELLEITHERTHEWELLWASVER
YDEEPORSHEFELLVERYSLOWLYFORSHEHADPLENTYOFTIMEASSHEWENTDO
WNTOLOOKABOUTHERANDTOWONDERWHATWASGOINGTOHAPPENNEXTFIRS
TSHETRIEDTOLOOKDOWNANDMAKEOUTWHATSHEWASCOMINGTOBUTITWAS
TOODARKTOSEEANYTHINGTHENSHELOOKEDATTHESIDESOFTHEWELLANDNO
TICEDTHATTHEYWEREFILLEDWITHCUPBOARDSANDBOOKSHELVESHEREAND
THERESHESAWMAPSANDPICTURESHUNGUPONPEGSSHETOOKDOWNAJARFRO
MONEOFTHESHELVESASSHEPASSEDITWASLABELLEDORANGEMARMALADEB
UTTOHERGREATDISAPPOINTMENTITWASEMPTYSHEDIDNOTLIKETODROPTHEJ
ARFORFEAROFKILLINGSOMEBODYUNDERNEATHSOMANAGEDTOPUTITINTOO
NEOFTHECUPBOARDSASSHEFELLPASTIT"

Part Two: Finding the Longest Menu

Finding the longest menu cannot be as easily broken down into steps as encoding a message with an Enigma machine, as each execution of finding the menu will differ depending on the starting index. The search for the longest menu can be seen as a tree, there could be multiple occurrences of a ciphertext letter in the plaintext and so we would need to try to explore these possible branches in order to find which one is the longest. Before we can think about implementing this, we need to think about how we will represent the Cribs and the Menus.

A Crib is simply an array of pairs of characters generated by using the zip function on two strings, the first item in the pair represents a plaintext character and the second the corresponding ciphertext character. However, we also want to know the index of the pair to make it easier when we create a menu out of a Crib, so we need to define a "FullCrib" as an array of pairs again but the first item in the array is the character pair and the second item is the index – we can achieve this by writing a function that zips a Crib with a list of Ints ranging from 0 to the length of the plaintext, converting it into a "FullCrib". It's important that it is the length of the plaintext, as received ciphertexts may be longer than the plaintext message used for the Crib, so we only want to use the plaintext part.

The Menu type is much simpler, it is just an array of Ints.

After our Crib to FullCrib conversion function, we can now write a function that finds a ciphertext character in the plaintext given a "piece" of the Crib. Using pattern matching we can say that if a ciphertext character c1 matches a plaintext character p2 in the crib *and* their indices are not equal (that would mean they are the same piece) then we return the piece of the Crib containing p2, if not then we recurse on the rest of the list and continue searching. If we cannot find a piece, then we simply return the piece that we were searching for.

When creating a Menu, it is important that the same piece does not appear twice as it could lead to possibly infinite loops. A function that removes a piece from the Crib should be written. This function should essentially reconstruct the Crib but "skip" the piece we are wanting to delete by not concatenating it with the recursive cases.

When creating our branches, we must consider multiple occurrences of a ciphertext character in the plaintext, therefore we need a function that generates a list of all these occurrences. We will once again achieve this through recursion – our base case is if the piece is not found i.e., we have generated all the options or there are none, otherwise we use our "finding in plaintext" function to generate the first occurrence and then recurse using the same piece but removing the found piece from the Crib – forcing it to choose the next occurrence. This should generate a list of pieces that all share the same character in plaintext as the ciphertext and the base case will append an empty list.

Now we have what we need to start our search. We can search through the crib by recursively calling the finding in plaintext function and removing pieces from the crib, and our base case is if the finding function returns the piece it was searching for then we just return a list just containing the piece – otherwise if a piece is found then we return the single list and append the subsequent recursive calls to it.

This function will generate a Menu; however, it will not be the longest. The function only takes the first occurrence of a plaintext character it sees and will not account for other occurrences – in terms of our "tree view" we can say it generates a "left-most tree" i.e., only considering the first option (that would be drawn branching to the left) at any node. Figure 5 shows an example of a tree starting from index 13.

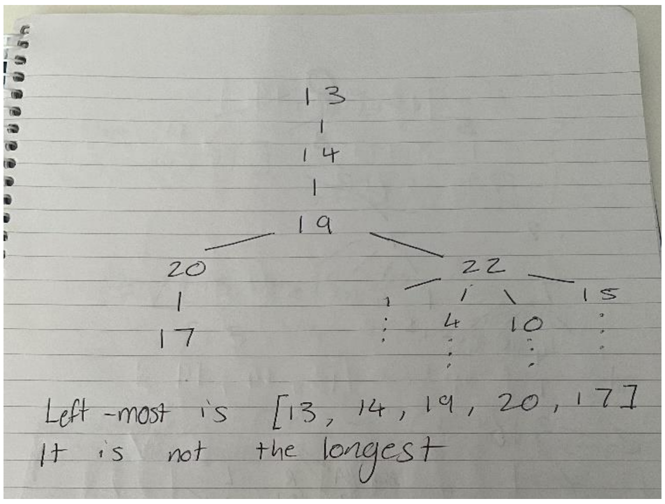


Figure 5: Possible Menu starting from index 13

This is where our "finding all occurrences" function comes in – what we need to do now is go through all the possible nodes at a point generated by said function, create their "left-most tree" and see which one is the longest, the node with the longest sub-tree will be chosen and the search will continue from there. So, we need three new functions:

- A function that, given a list of Crib pieces (the occurrences) and a Crib generates a list of left-most trees starting from each node.
- A function that selects the longest left-most tree and returns it
- A function that returns only the node of the longest tree

Generating all the left-most trees can be done by, once again, using recursion. We take the node at each point using pattern matching, create the tree, then put it as a single item in a list and recurse on the rest of the nodes list. Selecting the longest tree can be done by using foldl and a Lambda function that compares the length of two trees. Getting the node can be done by using the previous function that selects the longest tree and uses head to get just the first Crib piece.

Now we need to write a function that will create the longest chain of pieces given a starting piece and a Crib. We will need three cases:

- If the find in plaintext function returns the piece (i.e., link not found) then put the piece in a list and stop (base case). This is the same base case as the other functions such as the function that creates a left-most tree and our find all occurrences function.
- Call the find all occurrences function and if the length of the list is greater than 1 (more than one occurrence) then recurse using our selecting best node function as the next piece in the call and pass in the Crib with the current piece removed.
- Otherwise simply use the finding in plaintext function as our next piece and remove the current piece from the Crib.

Then we can generate all the chains by going through a given Crib and calling this function each time and append each chain to a list. For this function we will need to have the same Crib as two parameters because one of them must be kept constant in order to have a Crib to search through. Now that we have the chains all we need to do to generate a Menu is take the indices from each piece, add it to a list and recurse.

Finally, we can implement our "longestMenu" top-level function, the order will be as follows:

1. Create all the menus for a Crib
2. Select the longest one
3. Make a list of the indices

Testing

In order of implementation:

- createCrib
- findInPlaintext
- removeCribPiece
- findAllOccurrences
- createLeftmostMenu

- allLeftMenus
- selectLongest
- selectBestNode
- createLongest
- createAllMenus
- takeIndices
- longestMenu

Testing was done using these variables:

- Plaintext message: “WETTERVORHERSAGEBISKAYA”
- Ciphertext message: “RWIVTYRESXBFOGKUHQBAISE”

As well as the variables in the Main.hs file.

<u>createCrib</u>

Input	**Output**
(zip "WETTERVORHERSAGEBISKAYA" "RWIVTYRESXBFOGKUHQBAISE")	[(('W','R'),0),(('E','W'),1),(('T','I'),2),(('T','V'),3),(('E','T'),4),(('R','Y'),5),(('V','R'),6),(('O','E'),7),(('R','S'),8),(('H','X'),9),(('E','B'),10),(('R','F'),11),(('S','O'),12),(('A','G'),13),(('G','K'),14),(('E','U'),15),(('B','H'),16),(('I','Q'),17),(('S','B'),18),(('K','A'),19),(('A','I'),20),(('Y','S'),21),(('A','E'),22)]
(zip "WETTERVORHERSAGEBISKAYA" (encodeMessage "WETTERVORHERSAGEBISKAYA" enigma1))	[(('W','M'),0),(('E','V'),1),(('T','A'),2),(('T','E'),3),(('E','D'),4),(('R','U'),5),(('V','S'),6),(('O','B'),7),(('R','F'),8),(('H','S'),9),(('E','U'),10),(('R','Y'),11),(('S','H'),12),(('A','G'),13),(('G','P'),14),(('E','C'),15),(('B','M'),16),(('I','V'),17),(('S','E'),18),(('K','G'),19),(('A','Q'),20),(('Y','S'),21),(('A','Y'),22)]

N.B. “enigma1” is defined as it was in the first part of the assignment.

<u>findInPlaintext</u>

For ease of testing I defined a variable “crib1” as: [(('W','R'),0),(('E','W'),1),(('T','I'),2),(('T','V'),3),(('E','T'),4),(('R','Y'),5),(('V','R'),6),(('O','E'),7),(('R','S'),8),(('H','X'),9),(('E','B'),10),(('R','F'),11),(('S','O'),12),(('A','G'),13),(('G','K'),14),(('E','U'),15),(('B','H'),16),(('I','Q'),17),(('S','B'),18),(('K','A'),19),(('A','I'),20),(('Y','S'),21),(('A','E'),22)]

Input	**Output**
((‘W’,’R’),0) crib1	((‘R’,’Y’),5)
((‘E’,’W’),1) crib1	((‘W’,’R’),0)
((‘T’,’I’),2) crib1	((‘I’,’Q’),17)
((‘T’,’V’),3) crib1	((‘V’,’R’),6)
((‘I’,’Q’),17)	((‘I’,’Q’),17)
((‘R’, ‘T’), 13)	((‘T’,’I’),2)

Because this function doesn’t check to see whether the input piece is in the crib, inputting a random piece can still result in the function finding the ciphertext character in the plaintext. However, when we create our menus, we will only use pieces from the Crib supplied as we recurse through the list.

removeCribPiece

Input	Output
(('W','R'),0) crib1	[(('E','W'),1),(('T','I'),2),(('T','V'),3),(('E','T'),4),(('R','Y'),5),(('V','R'),6),(('O','E'),7),(('R','S'),8),(('H','X'),9),(('E','B'),10),(('R','F'),11),(('S','O'),12),(('A','G'),13),(('G','K'),14),(('E','U'),15),(('B','H'),16),(('I','Q'),17),(('S','B'),18),(('K','A'),19),(('A','I'),20),(('Y','S'),21),(('A','E'),22)]
((‘G’,’K’),14) crib1	[(('W','R'),0),(('E','W'),1),(('T','I'),2),(('T','V'),3),(('E','T'),4),(('R','Y'),5),(('V','R'),6),(('O','E'),7),(('R','S'),8),(('H','X'),9),(('E','B'),10),(('R','F'),11),(('S','O'),12),(('A','G'),13),(('E','U'),15),(('B','H'),16),(('I','Q'),17),(('S','B'),18),(('K','A'),19),(('A','I'),20),(('Y','S'),21),(('A','E'),22)]
((‘Z’,‘X’),3) crib1	[(('W','R'),0),(('E','W'),1),(('T','I'),2),(('T','V'),3),(('E','T'),4),(('R','Y'),5),(('V','R'),6),(('O','E'),7),(('R','S'),8),(('H','X'),9),(('E','B'),10),(('R','F'),11),(('S','O'),12),(('A','G'),13),(('G','K'),14),(('E','U'),15),(('B','H'),16),(('I','Q'),17),(('S','B'),18),(('K','A'),19),(('A','I'),20),(('Y','S'),21),(('A','E'),22)]

With the way the function is written, any pieces of the Crib that don’t exist within it will have no effect on the Crib.

findAllOccurrences

Input	Output
(('W','R'),0) crib1	[(('R','Y'),5),(('R','S'),8),(('R','F'),11)]
((‘E’,’W’),1) crib1	[(('W','R'),0)]
(('K','A'),19) crib1	[(('A','G'),13),(('A','I'),20),(('A','E'),22)]
(('H','X'),9) crib1	[]

createLeftmostMenu

Input	Output
(('W','R'),0) crib1	[(('W','R'),0),(('R','Y'),5),(('Y','S'),21),(('S','O'),12),(('O','E'),7),(('E','W'),1)]
(('E','W'),1) crib1	[(('E','W'),1),(('W','R'),0),(('R','Y'),5),(('Y','S'),21),(('S','O'),12),(('O','E'),7),(('E','T'),4),(('T','I'),2),(('I','Q'),17)]
(('H','X'),9) crib1	[(('H','X'),9)]
((‘Z’,’F’),23) crib1	[(('Z','F'),23)]
(('E','B'),10) crib1	[(('E','B'),10),(('B','H'),16),(('H','X'),9)]

allLeftMenus

Input	Output
(findAllOccurrences (('W','R'),0) crib1) (removeCribPiece (('W','R'),0) crib1)	[[(('R','Y'),5),(('Y','S'),21),(('S','O'),12),(('O','E'),7),(('E','W'),1)],[(('R','S'),8),(('S','O'),12),(('O','E'),7),(('E','W'),1)],[(('R','F'),11)]]
(findAllOccurrences (('E','W'),1) crib1)	[[(('W','R'),0),(('R','Y'),5),(('Y','S'),21),(('S','O'),12),(('O','E'),7),(('E','T'),4),(('T','I'),2),(('I','Q'),17)]]

(removeCribPiece (('E','W'),1) crib1)	
(findAllOccurrences (('H','X'),9) crib1) (removeCribPiece (('H','X'),9) crib1)	[]
(findAllOccurrences (('L','Y'),22) crib1) (removeCribPiece (('L','Y'),22) crib1)	[[(('Y','S'),21),(('S','O'),12),(('O','E'),7),(('E','W'),1),(('W','R'),0),(('R','Y'),5)]]

We must remove the piece before we search otherwise the piece to start from will appear in the Menus.

This function will also attempt to use pieces that don’t exist in the Crib because it doesn’t consider them. Our longestMenu searches through just the given Crib anyway so there is no problem here.

selectLongest

Input	**Output**
[[(('R','Y'),5),(('Y','S'),21),(('S','O'),12),(('O','E'),7),(('E','W'),1)],[(('R','S'),8),(('S','O'),12),(('O','E'),7),(('E','W'),1)],[(('R','F'),11)]]	[(('R','Y'),5),(('Y','S'),21),(('S','O'),12),(('O','E'),7),(('E','W'),1)]
[]	[]
[[((‘R’,’F’),11)]]	[[((‘R’,’F’),11)]]

selectBestNode

Input	**Output**
[[(('R','Y'),5),(('Y','S'),21),(('S','O'),12),(('O','E'),7),(('E','W'),1)],[(('R','S'),8),(('S','O'),12),(('O','E'),7),(('E','W'),1)],[(('R','F'),11)]]	((‘R’,’Y’),5)
[[((‘R’,’F’),11)]]	((‘R’,’F’),11)

createLongest

Input	**Output**
(('W','R'),0) crib1	[(('W','R'),0),(('R','Y'),5),(('Y','S'),21),(('S','O'),12),(('O','E'),7),(('E','T'),4),(('T','V'),3),(('V','R'),6),(('R','S'),8),(('S','B'),18),(('B','H'),16),(('H','X'),9)]
((‘E’,’W’),1) crib1	[(('E','W'),1),(('W','R'),0),(('R','Y'),5),(('Y','S'),21),(('S','O'),12),(('O','E'),7),(('E','T'),4),(('T','V'),3),(('V','R'),6),(('R','S'),8),(('S','B'),18),(('B','H'),16),(('H','X'),9)]

((‘A’,’G’),13) crib1	[(('A','G'),13),(('G','K'),14),(('K','A'),19),(('A','E'),22),(('E','W'),1),(('W','R'),0),(('R','Y'),5),(('Y','S'),21),(('S','O'),12),(('O','E'),7),(('E','T'),4),(('T','V'),3),(('V','R'),6),(('R','S'),8),(('S','B'),18),(('B','H'),16),(('H','X'),9)]

We know that this function works because the output is different from that of createLeftmostMenu (i.e., the generated Menu is longer).

createAllMenus

N.B. The output for crib1 is too long to fit into a table.

Input: crib1 crib1

Output:

```
[[(('W','R'),0),(('R','Y'),5),(('Y','S'),21),(('S','O'),12),(('O','E'),7),(('E','T'),4),(('T','V'),3),(('V','R')
,6),(('R','S'),8),(('S','B'),18),(('B','H'),16),(('H','X'),9)],[(('E','W'),1),(('W','R'),0),(('R','Y'),5),(('Y
','S'),21),(('S','O'),12),(('O','E'),7),(('E','T'),4),(('T','V'),3),(('V','R'),6),(('R','S'),8),(('S','B'),18),(('
B','H'),16),(('H','X'),9)],[(('T','I'),2),(('I','Q'),17)],[(('T','V'),3),(('V','R'),6),(('R','Y'),5),(('Y','S'),2
1),(('S','O'),12),(('O','E'),7),(('E','W'),1),(('W','R'),0),(('R','S'),8),(('S','B'),18),(('B','H'),16),(('H','
X'),9)],[(('E','T'),4),(('T','V'),3),(('V','R'),6),(('R','Y'),5),(('Y','S'),21),(('S','O'),12),(('O','E'),7),(('
E','W'),1),(('W','R'),0),(('R','S'),8),(('S','B'),18),(('B','H'),16),(('H','X'),9)],[(('R','Y'),5),(('Y','S'),
21),(('S','O'),12),(('O','E'),7),(('E','W'),1),(('W','R'),0),(('R','S'),8),(('S','B'),18),(('B','H'),16),(('H'
,'X'),9)],[(('V','R'),6),(('R','Y'),5),(('Y','S'),21),(('S','O'),12),(('O','E'),7),(('E','W'),1),(('W','R'),0)
,(('R','S'),8),(('S','B'),18),(('B','H'),16),(('H','X'),9)],[(('O','E'),7),(('E','W'),1),(('W','R'),0),(('R','
Y'),5),(('Y','S'),21),(('S','B'),18),(('B','H'),16),(('H','X'),9)],[(('R','S'),8),(('S','O'),12),(('O','E'),7),
(('E','W'),1),(('W','R'),0),(('R','Y'),5),(('Y','S'),21),(('S','B'),18),(('B','H'),16),(('H','X'),9)],[(('H','
X'),9)],[(('E','B'),10),(('B','H'),16),(('H','X'),9)],[(('R','F'),11)],[(('S','O'),12),(('O','E'),7),(('E','W'
),1),(('W','R'),0),(('R','Y'),5),(('Y','S'),21),(('S','B'),18),(('B','H'),16),(('H','X'),9)],[(('A','G'),13),(
('G','K'),14),(('K','A'),19),(('A','E'),22),(('E','W'),1),(('W','R'),0),(('R','Y'),5),(('Y','S'),21),(('S','O
'),12),(('O','E'),7),(('E','T'),4),(('T','V'),3),(('V','R'),6),(('R','S'),8),(('S','B'),18),(('B','H'),16),(('H',
'X'),9)],[(('G','K'),14),(('K','A'),19),(('A','E'),22),(('E','W'),1),(('W','R'),0),(('R','Y'),5),(('Y','S'),2
1),(('S','O'),12),(('O','E'),7),(('E','T'),4),(('T','V'),3),(('V','R'),6),(('R','S'),8),(('S','B'),18),(('B','H')
,16),(('H','X'),9)],[(('E','U'),15)],[(('B','H'),16),(('H','X'),9)],[(('I','Q'),17)],[(('S','B'),18),(('B','H')
,16),(('H','X'),9)],[(('K','A'),19),(('A','E'),22),(('E','W'),1),(('W','R'),0),(('R','Y'),5),(('Y','S'),21),(
('S','O'),12),(('O','E'),7),(('E','T'),4),(('T','V'),3),(('V','R'),6),(('R','S'),8),(('S','B'),18),(('B','H'),16
),(('H','X'),9)],[(('A','I'),20),(('I','Q'),17)],[(('Y','S'),21),(('S','O'),12),(('O','E'),7),(('E','W'),1),(('
W','R'),0),(('R','S'),8),(('S','B'),18),(('B','H'),16),(('H','X'),9)],[(('A','E'),22),(('E','W'),1),(('W','R'
),0),(('R','Y'),5),(('Y','S'),21),(('S','O'),12),(('O','E'),7),(('E','T'),4),(('T','V'),3),(('V','R'),6),(('R','
S'),8),(('S','B'),18),(('B','H'),16),(('H','X'),9)]]
```

takeIndices

Input	**Output**
[(('W','R'),0),(('R','Y'),5),(('Y','S'),21),(('S','O'),12),(('O','E'),7),(('E','T'),4),(('T','V'),3),(('V','R'),6),(('R','S'),8),(('S','B'),18),(('B','H'),16),(('H','X'),9)]	[0,5,21,12,7,4,3,6,8,18,16,9]
[(('A','G'),13),(('G','K'),14),(('K','A'),19),(('A','E'),22),(('E','W'),1),(('W','R'),0),(('R','Y'),5),(('	[13,14,19,22,1,0,5,21,12,7,4,3,6,8,18,16,9]

Y','S'),21),(('S','O'),12),(('O','E'),7),(('E','T'),4),(('T','V'),3),(('V','R'),6),(('R','S'),8),(('S','B'),18),(('B','H'),16),(('H','X'),9)]	
[]	[]

<u>longestMenu</u>

Input	Output
(zip "WETTERVORHERSAGEBISKAYA" "RWIVTYRESXBFOGKUHQBAISE")	[13,14,19,22,1,0,5,21,12,7,4,3,6,8,18,16,9]
(zip "WETTERVORHERSAGEBISKAYA" (encodeMessage "WETTERVORHERSAGEBISKAYA" enigma1))	[2,22,21,12,9,18,1,6]
(zip "WETTERVORHERSAGEBISKAYA" (encodeMessage "WETTERVORHERSAGEBISKAYADASISTMEHRWORTERALSPLAINTEXT" enigma1))	[2,22,21,12,9,18,1,6]
(zip "EINENSCHONENTAGNOCH" (encodeMessage "EINENSCHONENTAGNOCH" enigma1))	[0,7,3,12]

<u>Part Three: Simulating the Bombe</u>

Simulating the Bombe is a long process, as it requires looping through all the offsets and all the initial steckerboard pieces to check for contradictions. For this part we don't have to define any new data types as all we need is the Crib and we return a pair of Offsets and Stecker.

Before we think about looping through stecker pieces and offsets, let's consider what we need to do for 1 set of offsets and 1 initial stecker piece.

1. Take our starting stecker piece as a pair of characters and start from the beginning of the Menu.
2. Take the first element and encode it.
3. Create a new stecker piece using the encoded character in the first position and the current ciphertext character from our Crib in the second position.
4. If this contradicts our current steckerboard (i.e., a character appears more than once) then return nothing.
5. Else, move down the Menu to the next index in the Crib.
6. Try and repeat until we've finished the Menu.

It is important to note that when we encode a character, we use the offsets as if we've made a keypress n times, where n is the index of the Crib + 1. So, if we have initial offsets (0,0,25) and our Crib starts at 0 then the offsets we use to encode are (0,0,0).

Let's start to think about the most important part – checking for a contradiction. We can make a function that takes two pairs of characters as the format (a, b) (c, d) and if any of these characters are the same then we return true to indicate a contradiction has been found. Then we create a function that goes through a list of these pairs and if a contradiction is found at any point, then we return true.

Next, we need a function to get the correct offsets depending on the index of the Menu. There is a handy way to do this using iterate, if we iterate and then use take where the input is the index of the piece in the Menu + 2 (+2 because the Menu starts as index 0, take 0 returns nothing and take 1 would just return what the original offsets were, so we need one more step), then we can get the offsets needed to encode correctly.

For encoding a character and creating a new steckerboard piece, we can combine these into a single function taking a character, a FullCrib piece, offsets, the three rotors and the reflector. We return a pair of characters: the first element is the character encoded with the offsets, rotors and the reflector using our single character encryption function from part one.

Now we have everything we need to attempt to break the Enigma for a single set of offsets and a single initial stecker piece. We take the initial offsets, the initial stecker piece, the steckerboard (passed in recursively – should be initialised as []), the Menu in FullCrib form so we can use both the letters and the indices, the 3 rotors (LR, MR, RR) and the reflector. Our base case will be if the Menu is empty, meaning we have broken the Enigma and we can return the offsets and the full steckerboard, else our function goes as follows:

- If there is a contradiction in our current steckerboard, return Nothing.
- Otherwise, generate our new offsets and generate our new stecker piece based on these offsets, then recurse the function keeping everything the same except pass in the new piece and append the old piece to the stecker list.

What if we can't break the Enigma for these options? Use another initial stecker piece. We can create all possible initial stecker pieces by taking the plaintext character from the first Crib piece in the Menu and then offsetting this character each time we recurse. Our base case is when our offset is 26 i.e., we have gone through the whole alphabet. If our Menu starts with 'A' then the possible initial stecker pieces should be ('A','A'), ('A', 'B'), ('A','C'), …, ('A','Z').

Now we just write a function that loops through all the stecker pieces, runs our break function and if it goes through the whole stecker list without a possible steckerboard then it returns Nothing, else we recurse on the rest of the list.

However, there is another layer to this: if we've gone through all stecker pieces then we need to change the offsets. So, we need a function to generate all the offsets and a function that loops through all the offsets and runs our function that loops through all the stecker pieces.

We can generate all possible offsets through list comprehension, if we have (x, y, z) and for each number take from [0...25] then Haskell will generate all possible combinations. For our final "break" function we just do as we did before and loop through all offsets, run the function that loops through the stecker pieces, and if we get through the whole list of offsets without a possible steckerboard then we return Nothing.

Our final breakEnigma function will use this break function, and all our inputs will be either constant (i.e., rotors, reflector, empty list for initial steckerboard, list of offsets) or generated from the Menu (list of stecker pieces, Menu itself).

Testing

Unfortunately, I was unable to muster a properly working breakEnigma function in time. Despite this, I have these functions:

- longestFullCrib
- createSteckerPiece
- isContradiction
- checkContradiction
- changeOffsetByCrib
- tryBreak
- createAllInitialSteckers
- tryAllInitialSteckers
- createAllConfigs
- tryAllOffsetConfigs
- breakEnigma

longestFullCrib

This is just a helper function I wrote that does the exact same as longestMenu but does not take the indices and leaves the Menu in its "FullCrib" form.

Input	**Output**
zip "WETTERVORHERSAGEBISKAYA" "RWIVTYRESXBFOGKUHQBAISE"	[(('A','G'),13),(('G','K'),14),(('K','A'),19),(('A','E'),22),(('E','W'),1),(('W','R'),0),(('R','Y'),5),(('Y','S'),21),(('S','O'),12),(('O','E'),7),(('E','T'),4),(('T','V'),3),(('V','R'),6),(('R','S'),8),(('S','B'),18),(('B','H'),16),(('H','X'),9)]

createSteckerPiece

This function creates a stecker piece using a character, the current crib piece, 3 rotors, offsets and a reflector. It will encrypt the character and pair it with the ciphertext character in the piece.

Input	**Output**
'A' (('A','G'),13) rotor1 rotor2 rotor3 (0,0,13) reflectorB	('B','G')
'B' (('G','K'),14) rotor1 rotor2 rotor3 (0,0,14) reflectorB	('E','K')
'X' (('J','L'),42) rotor1 rotor3 rotor2 (13,22,0) reflectorB	('V','L')

isContradiction

This function checks two stecker pieces (pairs of characters) and checks if any of them violate the rules for building a steckerboard.

Input	**Output**
(‘A’,’A’) (‘A’,’A’)	False
(‘A’,’B’) (‘B’,’A’)	False
(‘A’,’B’) (‘A’,’B’)	False
(‘A’,’A’) (‘C’,’B’)	False
(‘A’,’A’) (‘C’,’A’)	False
(‘A’,’C’) (‘A’,’D’)	True
(‘A’,’C’) (‘D’,’A’)	True

<u>checkContradiction</u>

This function calls isContradiction recursively over a steckerboard.

Input	**Output**
(‘A’,’A’) [(‘A’,’A’), (‘C’,’D’), (‘E’,’F’)]	False
(‘A’, ‘B’) [(‘A’,’A’), (‘C’,’D’), (‘E’,’F’)]	False
(‘A’,’C’) [(‘A’,’A’), (‘C’,’D’), (‘E’,’F’)]	True
(‘C’,’A’) [(‘A’,’A’), (‘C’,’D’), (‘E’,’F’)]	True
(‘D’, ‘C’) [(‘A’,’A’), (‘C’,’D’), (‘E’,’F’)]	False
(‘D’, ‘E’) [(‘A’,’A’), (‘C’,’D’), (‘E’,’F’)]	True

<u>changeOffsetByCrib</u>

Given a crib piece, take its index and simulate keypress.

Input	**Output**
(0,0,25) rotor1 rotor2 rotor3 ((‘E’,’W’),1)	(0,0,1)
(13,14,2) rotor3 rotor1 rotor2 ((‘A’,’G’),13)	(13,15,16)
(25,25,25) rotor3, rotor2, rotor1 ((‘H’,’L’),42)	(25,1,16)

<u>tryBreak</u>

Will try to break the Enigma for 1 offset and 1 initial stecker piece. This function returns “Nothing” a lot so I made a tryBreak’ function which prints the offsets and the stecker when it finds the contradiction.

N.B. “menu” is a variable defined for testing based off one in the brief.

```
menu = [(('E','W'),1::Int), (('W','R'),0::Int), (('R', 'Y'),
5::Int), (('Y','S'),21::Int), (('S','O'),12::Int),
(('O','E'),7::Int), (('E', 'T'),4::Int), (('T','V'), 3::Int),
(('V', 'R'),6::Int), (('R','S'),8::Int), (('S', 'B'),18::Int),
(('B','H'),16::Int), (('H','X'),9::Int)]
```

Input	**Output**

(0,0,25) ('E','E') [] (longestFullCrib (zip "WETTERVORHERSAGEBISKAYA" "RWIVTYRESXBFOGKUHQBAISE")) rotor1 rot or2 rotor3 reflectorB	((0,0,25),[('E','E'),('L','G'),('Y','K'),('N','A'),('I','E'),('H','W'),('Z','R'),('Y','Y'),('E','S')])
(0,0,1) (‘A’, ‘G’) [] menu rotor1 rotor2 rotor3 reflectorB	((0,0,1),[('A','G'),('Z','W'),('T','R'),('H','Y'),('O','S'),('C','O')])
(25,25,25) (‘Z’,’X’) [] menu rotor2 rotor3 rotor1 reflectorB	((25,25,25),[('Z','X'),('P','W'),('Q','R'),('R','Y')])

<u>createAllInitialSteckers</u>

This is a function that generates all 26 possible starting stecker pieces from a crib piece.

Input	**Output**
((‘A’,’G’),13) 0 []	[('A','A'),('A','B'),('A','C'),('A','D'),('A','E'),('A','F'),('A','G'),('A','H'),('A','I'),('A','J'),('A','K'),('A','L'),('A','M'),('A','N'),('A','O'),('A','P'),('A','Q'),('A','R'),('A','S'),('A','T'),('A','U'),('A','V'),('A','W'),('A','X'),('A','Y'),('A','Z')]
(('X','Z'),12) 0 []	[('X','X'),('X','Y'),('X','Z'),('X','A'),('X','B'),('X','C'),('X','D'),('X','E'),('X','F'),('X','G'),('X','H'),('X','I'),('X','J'),('X','K'),('X','L'),('X','M'),('X','N'),('X','O'),('X','P'),('X','Q'),('X','R'),('X','S'),('X','T'),('X','U'),('X','V'),('X','W')]

<u>tryAllInitialSteckers</u>

Runs tryBreak recursively with all initial stecker pieces. The outputs for these are too big to fit in a table. Here is one case which has the inputs (0,0,25) (createAllInitialSteckers (('A','G'),13) 0 []) [] menu rotor1 rotor2 rotor3 reflectorB.

((0,0,25),[('A','A'),('B','W'),('E','R'),('R','Y')])

((0,0,25),[('A','B'),('B','W')])

((0,0,25),[('A','C'),('B','W'),('E','R'),('R','Y')])

((0,0,25),[('A','D'),('B','W'),('E','R'),('R','Y')])

((0,0,25),[('A','E'),('B','W'),('E','R')])

((0,0,25),[('A','F'),('B','W'),('E','R'),('R','Y')])

((0,0,25),[('A','G'),('B','W'),('E','R'),('R','Y')])

((0,0,25),[('A','H'),('B','W'),('E','R'),('R','Y')])

((0,0,25),[('A','I'),('B','W'),('E','R'),('R','Y')])

((0,0,25),[('A','J'),('B','W'),('E','R'),('R','Y')])

((0,0,25),[('A','K'),('B','W'),('E','R'),('R','Y')])

```
((0,0,25),[('A','L'),('B','W'),('E','R'),('R','Y')])
((0,0,25),[('A','M'),('B','W'),('E','R'),('R','Y')])
((0,0,25),[('A','N'),('B','W'),('E','R'),('R','Y')])
((0,0,25),[('A','O'),('B','W'),('E','R'),('R','Y')])
((0,0,25),[('A','P'),('B','W'),('E','R'),('R','Y')])
((0,0,25),[('A','Q'),('B','W'),('E','R'),('R','Y')])
((0,0,25),[('A','R'),('B','W'),('E','R')])
((0,0,25),[('A','S'),('B','W'),('E','R'),('R','Y')])
((0,0,25),[('A','T'),('B','W'),('E','R'),('R','Y')])
((0,0,25),[('A','U'),('B','W'),('E','R'),('R','Y')])
((0,0,25),[('A','V'),('B','W'),('E','R'),('R','Y')])
((0,0,25),[('A','W'),('B','W')])
((0,0,25),[('A','X'),('B','W'),('E','R'),('R','Y')])
((0,0,25),[('A','Y'),('B','W'),('E','R'),('R','Y')])
((0,0,25),[('A','Z'),('B','W'),('E','R'),('R','Y')])
```

tryAllOffsetConfigs and breakEnigma

These functions run for a very long time, and it would not be feasible to put the output here. Unfortunately, I could not get a working solution for these functions so it will return "Nothing" almost every time.

Part Four: Critical Reflection

Admittedly I went into this assignment struggling to understand the concepts behind the Enigma machine, but by the end I feel like I have understood the encryption algorithm and the concept of Cribs and Menus. Obviously, I would've liked to have understood the breaking of the Enigma part a little bit better so that I could've had a working solution, but I did my best to try give one.

I believe that this assignment has really helped to challenge and further my proficiency in Haskell as I had only done Haskell a bit before during my A-Level and so the most advanced thing we covered was recursion. Whereas now I've covered so many more concepts such as foldr/foldl, IO, lambda functions, where…, let…in, etc.

One key thing that I think I could've done better in my code was the design of the functions. In part one especially, I was using functions within functions a lot and in most cases a function would purely exist so that it didn't take up much space in the code editor when I could've used a different approach. In particular, the postReflection and preReflection functions which use rotoriseForward and rotoriseBackward three times in a chain, I feel like this could've been replaced with a foldr/foldl function and if I had more time I would seek to implement that.

Overall, this assignment has been challenging but also incredibly rewarding. I hope that in the future of my career I can put my Haskell knowledge to good use.