Project 1: Enigma

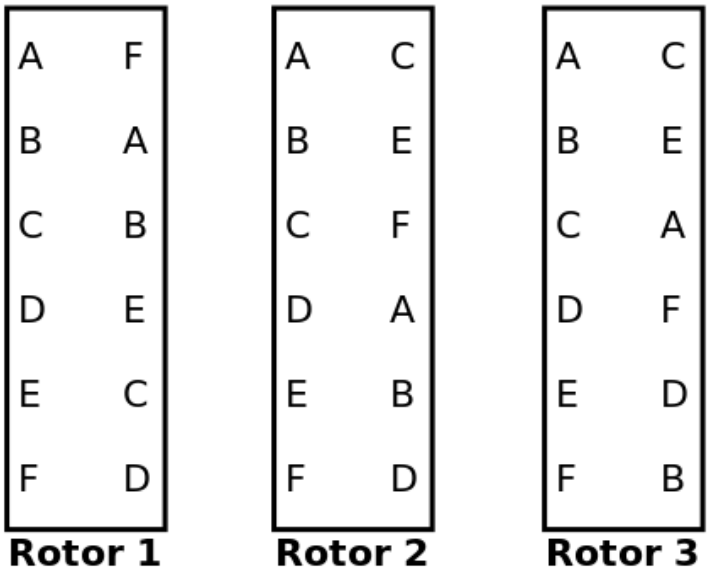
- **Due before 23:59 TRT on Sunday, December 6**
- Submit via blackboard.
- You are allowed to use any programming language.

1 History

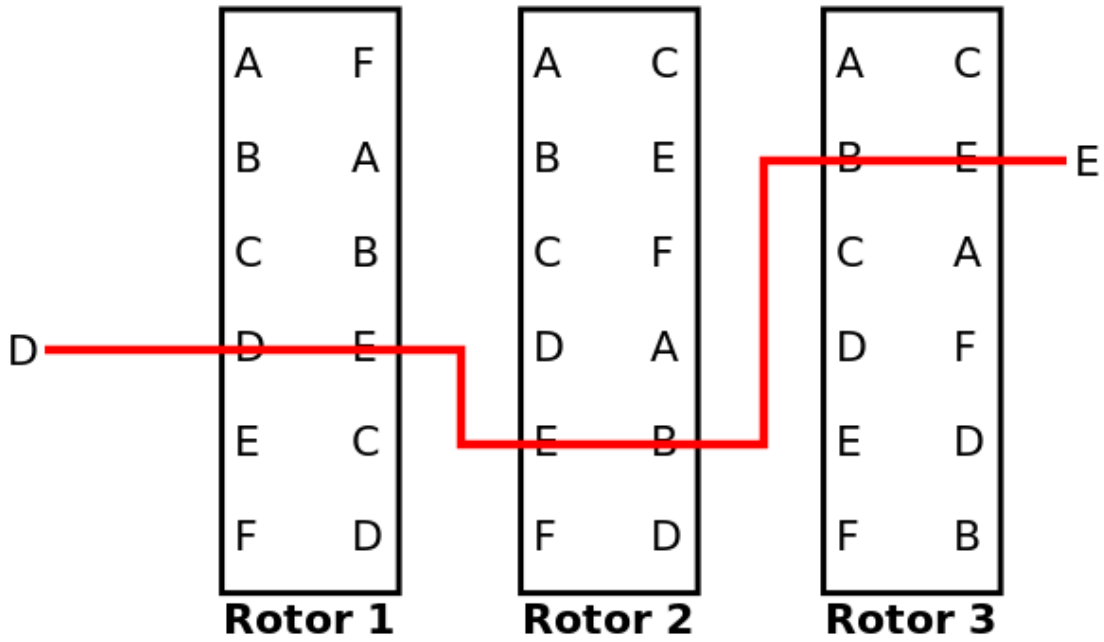
Substitution ciphers that encode a message by substituting one character for another go back at least as far as Julius Caesar, who used a rotating character scheme to encode military orders. As we know from COMP443/543, this simple type of encryption is extremely vulnerable to statistical attacks, however. In World War II, the German military employed an encryption scheme that addressed this weakness of simple substitution ciphers. This scheme, implemented by typewriter-sized devices known as Enigma machines, was believed by the Germans to be unbreakable.

2 Simple Model of the Enigma

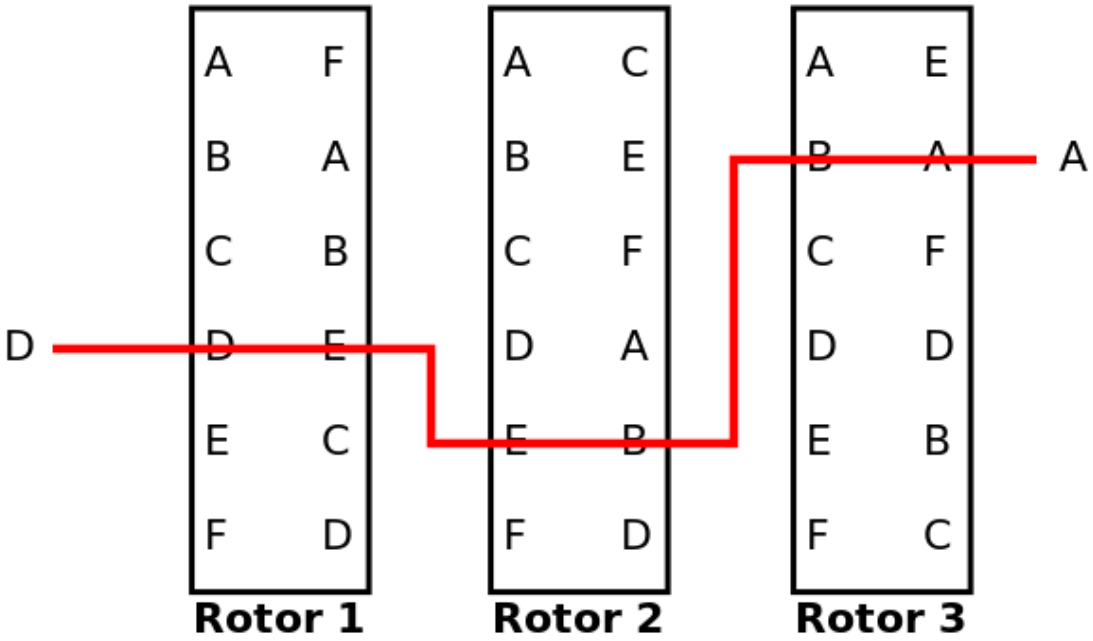
Enigma machines used interchangeable rotors that could be placed in different orientations to obtain different substitution patterns. More significantly, the rotors rotated after each character was encoded, changing the substitution pattern and making the code very difficult to break. Let's pretend there are only six letters in the alphabet, and we have the three rotors below.



In the first rotor, if an 'E' is entered, a 'C' comes out. If an 'A' comes in, an 'F' comes out. Whatever comes out of the first rotor, goes into the second rotor. Whatever comes out of the second rotor, goes into the third rotor. For our simple model, whatever comes out of the third rotor is the encrypted character. For example, to encrypt a 'D'...



At this point, it may seem that this is just a substitution cypher, just an unnecessarily complex one. Well, after encrypting this 'D', Rotor 3 is rotated one slot (in the below picture, notice how in the third rotor, 'A' no longer maps to 'C' - instead, it now maps to 'E', which is the letter 'B' used to map to. In other words, each letter in the right column has moved up one spot). So, when we encrypt 'D' again, we get a different encrypted character.



OK, fine, you say, so it's not a substitution cypher. But, because the rotor must eventually rotate all the way around again, it is just a Vigenere cypher with a key length of 6, and those are still vulnerable to a statistical attack. So here's the second twist... when that third rotor is rotating around to return to its original position (after the encryption of the 6th character), the second rotor rotates once as well, again changing the encryption. When the second rotor rotates all the way around, the first rotor clicks once, as well. So, the rotors don't return to their original position until you have typed 6^3 characters (26^3 if we're using a full alphabet). Essentially, this message has the same statistical properties as a one-time pad.

But how would this be decrypted? Well, each day the Germans had a secret key, consisting of how to put the rotors in the machine. If you had them in the machine in the proper places, you merely had to run the encryption backwards, allowing the rotors to click after each letter as if encrypting, and you would get the message back out.

3 The Assignment

Part One: 40 points

In this part, you will build an Enigma machine capable of encrypting and decrypting messages. You'll have to build two classes, Rotor and Enigma.

Rotor

A Rotor will have a list of characters as a field, which encodes the order of the letters in the rotor. The first character in the list will be whatever character is at the top of the Rotor (for example, in the first rotor in the images above, the list would be **['F','A','B','E','C','D']**. In addition, a Rotor will have a constructor, and three other methods.

The constructor should take a single String as an argument which contains the encoding of the Rotor. For example, to build the first rotor above, you would call **inner=Rotor("FABECD")** (for real messages, these strings will consist of 26 characters, not just the first six). The constructor should take the characters of the String, and put them into the list.

The method **encryptLetter(self, letter)** should return that rotor's encoding of the character **letter**. So, for the first Rotor in the image, **r.encryptLetter('D')** should return 'E'.

The method **decryptLetter(self, letter)** should do the opposite. So, in the first Rotor, **r.decryptLetter('E')** would return '**D**'.

The method **click(self)** should turn the rotor one click. After calling click on the first rotor, the first character should be 'A'. If this click completes one entire time around (meaning the new first element was the first element when the rotor was first initialized), it should return True. Otherwise, it should return False. You may find it helpful to add another field to your class to make this work more easily.

Enigma

Your Enigma class should have as fields three Rotors. I'm not going to give as much guidance on what an Enigma should be able to do, but at the very least, it should have methods called "encrypt" and "decrypt" that accept a String as an argument, and return another String.

Additionally, the Enigma should have a constructor, which accepts as arguments three strings describing the three rotors' initial settings (ie, for the above example, **e=Enigma('FABECD','CEFABD','CEAFDB')**).

For testing, first test just your decrypt function by decoding the following messages using the following settings:

- First rotor: **SHBMFWEIQRODTAVXCPYZUJKGNL**
- Second rotor: **GYRFNUCZLQDWMKHSJOEPBVITXA**
- Third rotor: **MSEWGQHDPRFNXATOIBUJLCZVYK**

OQMTANMGPABQSDAKAUFXXGJBSPHBZXHLXMBNOHTNQZQGDBMIQNZJ
MXNMFKPDVWDPIMPYACVYZUGUWGPVGAOXCDZDGYTLATOIBUJLCZVYNXATOIBUJLCZ
VYKMSEWGQHDPRFIBUJLCZVYKMSEWGQHDPRFNXATOWGQHDPRFNXATOIBUJLCZVYK

MSEZVYKMSEWGQHDPRFNXATOIBUJLCXATOIBUJLCZVYKMSEWGQHDPRFNFNXATOIBUJLCZVYKMSEWGQHDPRDPRFNXATOIBUJLCZVYKMSEWGQHUJLCZVYKMSEWGQHDPRFNXATOIBRFNXATOIBUJLCZVYKMSEWGQHDPTOIBUJLCZVYKMSEWGQHDPRFNXAGQHDPRFNXATOIBUJLCZVYKMSEWOIBUJLCZVYKMSEWGQHDPRFNXATSEWGQHDPRFNXATOIBUJLCZVYKMCZVYKMSEWGQHDPRFNXATOIBUJLPRFNXATOIBUJLCZVYKMSEWGQHDJLCZVYKMSEWGQHDPRFNXATOIBUVYKMSEWGQHDPRFNXATOIBUJLCZMSEWGQHDPRFNXATOIBUJLCZVYKHDPRFNXATOIBUJLCZVYKMSEWGQYKMSEWGQHDPRFNXATOIBUJLCZVBUJLCZVYKMSEWGQHDPRFNXATOIQHDPRFNXATOIBUJLCZVYKMSEWGATOIBUJLCZVYKMSEWGQHDPRFNXLCZVYKMSEWGQHDPRFNXATOIBUJZ

You can then test your encrypt by encrypting, and then decrypting, and making sure you end up with the same thing you started with. I'll be testing your code with my own examples, so be sure to be thorough.

4 Part Two: 30 points

The reason why the Enigma cryptosystem was considered secure by the Germans is not because it was impossible to break, but because the system was so complex, that by the time a message was broken, it wouldn't matter any more (modern cryptosystems actually work the same way, we've just made things even more complex). After all, to find the correct rotor settings for that day, somebody would have to try every possible combination of rotor settings, perform the decryption, and see if the output looked like German. With three rotors in any order, and 26 possible settings for each rotor, the number of possible messages is \(26^3 \cdot 3!\), which equals 105,456 possible decryptions to check - impossible by hand, a challenge for 1940s computing, and very fast with any modern computer like your laptop! (I'll note that our Enigma thus far is a simplified version; the true number of possible keys was 10,586,916,764,424,000.)

Researchers at Bletchley Park, England (led by Alan Turing), heavily used the work of **Polish mathematicians**, and built machines known as bombe (from the Polish "bomba kryptologiczna," or "cryptologic bomb") to eliminate many of these initial settings as impossibilities, and then run the remainder through a machine (arguably a computer) allowing for timely decryption of Nazi messages. This is hailed as one of the turning points of the war, and the first great success of computer science.

In order for this to work, they needed to know what part of the message was, so they could figure out which Enigma settings could produce that message. They eased their effort with non-computational approaches. For example, once the English learned that Enigma operators had been told to spell out their numbers, Turing discovered 90% of all messages contained the word "ONE." Therefore, a message containing the string "eins" was much more likely to be the correct decryption. Additionally, if it was particularly important to find that day's rotor settings, they would lay mines in sight of German lookouts, knowing they could anticipate a message containing the word "mine." This made it much easier to scan all the candidate plaintext messages for the word "mine" in order to find the correct one. Still another approach was that some operators liked to have a personal "sign-off" phrase; a message sent by that operator could be assumed to have that sign-off phrase included.

For the second part of this project, using your own Enigma machine that you made for part 1, you are going to build a Bombe. Below is some long cryptotext; your job is to decrypt this cryptotext WITHOUT knowing the rotor settings. Know that aircraft just **MINE**d the coast, and that odds are, the message contains the word **ONE**. The rotors are the same as in part one, though they obviously haven't started in the same settings, and may not be in the same order.

To do this, you'll have to try every single possible setting for the three rotors given in part 1, in every possible order, and see which ones might make sense. Make a class called **Bombe**. Your Bombe class can have whatever else you want in it, but it *must* have a constructor that takes three arguemnts for the rotor strings (though they might be in the wrong order), and it must also have a method **crack(self, ciphertext)** that tries all possible Enigma settings using the given rotors, and returns a list of possible decryptions that contain both the word "**MINE**" and the word "**ONE**".

For your own testing, here is a message for you to try that should have exactly one possible decryption, using the same rotor settings as in the previous part:

NPWCDPBRIVDZGARYLECHBTOCKJCMJVDRFZEYFWJTRZLPDEVDHIJXYHRBRJTVVQCFD QUWHRQKYPYFAJJKSDEJVOVZNWYFYINBPBSNHZAGDACJRYRLLJAWCJKHTEVATAAZW VUHSBTCKBVHTNSGFDPHGIZDSZXMBSIKWLMMISUQNWCRPSHSNFAALBQNMKESIHCPGV RTRFTPRYTRIRMNYMVSLEKAPRISAUSTRXQFVCLYWXZLLXHHKHJJUTPKHBTFENHMFRL FLUHYQJSCMNEBB

5 Part Three: 30 points

This part is related to communication among programs and interception of the messages. You need to code a simple chat interface (running as two separate programs, Alice and Bob) that are communicating with each other via enigma encryption and decryption (i.e., the enigma class). A simple communication can be implemented via writing and reading common files Alice.txt and Bob.txt (for messages Alice and Bob are sending respectively) the **encrypted version** of the entered messages. Note that each party's messages need to be printed on the other party's user interface after being sent. Also, after writing to the common file, simple pinging (i.e., independent of the even a bit of the message sent) to the other program is allowed. Instead of pinging, screening of the file in a loop is also allowed.

An important notice: Due to the Enigma security, the rotors after each sent message continue from the current position. In particular, they **do not** rewind back into the configuration of the first message.

You also need to code an enigma breaker interface program that takes a file containg enigma cyphertext as input and using the class bombe, breaks the code and outputs the possible decryptions on the screen.