Martin You
Sep 19, 2024

CSE 360
Amir Rahmati

Enigma Machine

Introduction:

This program simulates the enigma machine, using settings for rotors, reflectors and a plugboard, it takes an input and encodes it. It works exactly as the one used in WWII by Nazi Germany. This program takes three lines and outputs the encoded text, it works backwards, so if you enter the encoded message with the same settings, it gives the decoded message.

This program assumes all the input is valid, so there is no error checking. The input taken is as follows:

1. The first line is a six char long string stored in the **'settings'** variable.
    a. The first letter is the reflector (A,B,C). Stored in **'reflectorSetting'**.
    b. The three next characters are the rotors, in order Left, Middle, Right (1,2,3,4,5), these should not be repeated. Stored in **'rotorOrder'**.
    c. The last three characters are each rotor's starting position (A-Z). Stored in **'rotorStart'**.
2. The second line is the **'plugboard'** settings. It connects two letters so that each maps to the other.
    a. Letters can only be used once, and can be empty.
    b. (AB,CD) <- A maps to B and vice versa, C maps to D.
3. The third line is the text to be encrypted, stored in the **'text'** variable. Only alphabetical upper-case characters are accepted.

How the program works:

First, it takes in the input (three strings). Then, it enters them into **'enigma_machine()'**. This function takes in three strings, which are then split into substrings representing the settings. It sets up the reflector to the **'reflector'** variable using **'setReflector()'**, rotors **'setRotor()'**, and plugs **'setPlugs()'**. For each character, it transforms it into an int, with A being 0 and Z being 25. It rotates the rotor using **'stepRotor()'**, the character is changed with the plug by using it as an index and changing it to the number, then it goes through the three rotors, first the rightmost rotor, the middle rotor, and finally the leftmost rotor. The character is put into the reflector as an index and set to the resulting int. It comes back and goes through the rotors in reverse, note that the encryption through the rotors is now inverted, and goes back again through the plugs. At the end of this, the character is printed, and so on until we run out of characters.

Settings

How the Plugboard is Implemented

The plugboard is implemented with an array called **'plugMap'** that has 26 spaces, and every index has its index position (**plugMap[0]** = 0, **plugMap[1]** = 1). It is set up by splitting the input by commas, and depending on the plugs inputted, maps to a certain number that represents a number. Ex. when you input "AZ" into **setPlug()**, **plugMap[0]** = 25, **plugMap[25]** = 0, and when inputting the char, it will output the other number.

Martin You
Sep 19, 2024

CSE 360
Amir Rahmati

Rotor

This rotor actually takes a position and outputs another position, so depending on the rotation of the rotor, it gives a different result even if the same number is used. The rotor uses an array **'rotorArr[3][3][x]'** that stores the three rotors, two doubly ended queues, and an int. The original rotor settings are in **'rotor[5][3]'** which has two arrays and an int, the arrays are converted to doubly ended queues when transferred to **'rotorArr[]'**.

1. The first array **'Rotor[x][0]'** stores the amount of spots that it travels (Ex. **'rotor[1][0][0]'** = 4, so if 0 is put in, it moves up 4 spots, and outputs 5, which means when A (0) is inputted, E (4) is output), this array helps with the initial rotor encryption.
2. The second array **'rotorArr[x][1]'** is the letter (number code) that the spot is mapped into, this helps locate how the letter was output (Ex. if when you input 5 you get 25, if you reverse it, you have 25 and entering it you get 5).
3. The third one **'rotorArr[x][3]'** stores the position of the notch, if the position is 5, it is 5 spins away, when the rotor spins and the value is 0, the rotor to the left is rotated.

The data type used is a doubly ended queue. When using **'setRotor()'**, it makes a doubly ended queue using the appropriate rotors' arrays and number. It rotates the rotors to their position indicated. When rotating, **'stepRotor()'** pops the first element and appends it to the end for both queues, simulating the rotation. The values in the second doubly ended queue are subtracted by one for every turn by **'cycleRotor()'**, as the position of the letter goes down one (If B is in 1, when rotated, it goes to 0), if it reaches -1, it goes to the top (position 25).

The rotors can have different starting positions, the notches are fixed (Ex. it will always rotate the left rotor when it goes over Q). Whenever a letter is typed, the rightmost rotor takes a step, and if it was previously where the notch is positioned, the rotor in the left moves down. The rotor maps one letter to another, and is invertible (Ex. if Y goes in and X goes out, when going backwards, if X goes in Y goes out), think of the rotors as connections to certain positions more than certain letters.

When first going through the rotors, the character is put into the rotor in a certain position let's say A is put in position 0, B in 1…, and that maps to another position which is obtained by adding the amount in **'rotorArr[2][0][0]'**. Imagine the int in that position is 5, so 0+5 = 5, now the position will be 5, and it is put in again **'rotorArr[1][0][5]'**, and imagine we get 21, so 5+21 = 26, which is over the index, we subtract 25 so it is mod 26, we get 1. Finally it is mapped to **'rotorArr[0][0][1]'**.

Going backwards, we will use the second doubly ended queue. What this does is take an int, x, and finds what index it is in **'rotorArr[0][1][x]'**, and turns it into that int. Basically finds where the number came from, and it is done with **'rotorArr[1][1][x]'** and **'rotorArr[2][1][x]'**.

Rotors

Rotor #	ABCDEFGHIJKLMNOPQRSTUVWXYZ	Notch (If rotor in _, when rotating, left rotor rotates too)

1	EKMFLGDQVZNTOWYHXUSPAIBRCJ	Q
2	AJDKSIRUXBLHWTMCQGZNPYFVOE	E
3	BDFHJLCPRTXVZNYEIWGAKMUSQO	V
4	ESOVPZJAYQUIRHXLNFTGKDCMWB	J
5	VZBRGITYUPSDNHLXAWMJQOFECK	Z

* 1: A to E, 2: A to A, 3: A to B, 4: A to E, 5: A to V

Reflector

There are three available reflectors stored in **'refArr()'**. This machine is not invertible, so if you plug in Y and get X, if you plug in X you are not guaranteed to output Y. When setting the reflector, it returns the array of the reflector used (A-0,B-1,C-2) to **'reflector[]'**. This is just a substitution encryption, takes the result from the leftmost rotor, maps the int and outputs to the leftmost rotor.

Reflectors

Reflector	ABCDEFGHIJKLMNOPQRSTUVWXYZ
A	EJMZALYXVBWFCRQUONTSPIKHGD
B	YRUHQSLDPXNGOKMIEBFZCWVJAT
C	FVPJIAOYEDRZXWGCTKUQSBNMHL

* If Ref. A: A goes to E, B: A to Y, C: A to F

Example:

Input:	Input:
B123AAA AB,CD,EF SECURITY	B421ZBD GI SECURITY
Output:	Output:
JKJRFLHF	ORBVTPIK