Polybius Square Cipher

Polybius Square Cipher

Keane J. Moraes

1. History

The Polybius Square was invented by the Greeks Cleoxenus and Democleitus but was made famous by the historian and scholar Polybius, earning him a cipher named after him. This cipher was not meant as a stand-alone cipher but rather as an aid to telegraphy.

	1	2	3	4	5
1	A	B	C	D	E
2	F	G	H	I/J	K
3	L	M	N	O	P
4	Q	R	S	T	U
5	V	W	X	Y	Z

2. Modern Usage

The Polybius square cipher is not as popular as some of its variants are. One of its variants, Tap Code (a.k.a Knock Code), was used as a means of communication by US prisoners of war during the Vietnam War. They used it by tapping on the metal bars on the jail cells. In Vietnam, this tap code was very successful. Prisoners not allowed to talk would tap each others thigh.

Polybius, being a monoalphabetic substitution cipher, is vulnerable to Frequency Analysis and thus in the modern era offers no security at all.

3. Encryption and Decryption Algorithms

3.1. Encryption

A standard Polybius Square is a 5×5 table containing 25 characters. The English alphabet has 26 characters and as a consequence, the standard practice is to omit the letter 'J' during encryption. Instead of using a matrix in code (which would use more memory and take more time to run), we utilize some handy tools from modular arithmetic to come up with some formulas to achieve the same task. This way our code is more efficient. The method is as follows.

Given a standard Polybius Square, The trick is in spotting a general formula in generating the rows and columns. Let R_i and C_i denote the row and column of the i^{th} character respectively.

$$R_i = \left\lfloor \frac{i}{5} \right\rfloor + 1$$

$$C_i = i \mod 5 + 1$$

We will perform a letter-by-letter encryption. We must take into account the conversion of the ASCII character values $\big(a(97) - z(122)\big)$ or $\big(A(65) - Z(90)\big)$ into the standard format $(0 - 25)$ else the modular arithmetic will not work with a modulo of 26.
Let E_t be the cipher text.
P_t be the plain text having length 'n'.
S be the shift, where $S \in \mathbb{N}$
$E_t = E_1 E_2 E_3 \dots E_n$ where E_i is the i^{th} character of the cipher text
The i^{th} character of the cipher text is computed -
If the letter is between 65 and 90 (Uppercase)

$$E_i = \Big[R_{P_i - 65} || C_{P_i - 65} \Big]$$

If the letter is between 97 and 122 (Lowercase)

$$E_i = \Big[R_{P_i - 97} || C_{P_i - 97} \Big]$$

where $||$ stands for concatenation.

3.1.1. Code Implementation

The code implementation of the encryption algorithm is a simple conversion from mathematical form to Java code. Note that we will only deal with upper case letters in the code implementation. The reader is free to implement the lower case version.

```
54  String result = "";
55  key = generateCustomKey(key.replaceAll(" ", ""), "ABCDEFGHIKLMNOPQRSTUVWXYZ");
```

Here we intialize the `result` variable and extract the unique key from the inputed 'seed'.

```
57  for (int i = 0; i < plainText.length(); i++) {
58    char character = plainText.charAt(i);
59    if (character < 65 || character > 90)
60      continue;
61    if (character == 'J')
62      character = 'I';
```

We check for any characters that are not in the Uppercase alphabet and ignore them (you can choose to append them by replacing `continue;` with `result += character; continue;`). If `character` is 'J' then it is reassigned to 'I'. First as always we take care of the the special characters. Anything not in the $[65, 90]$ range gets tacked onto the result.

```
64    int letterNumber = key.indexOf(character);
65    int rowNumber = (int) (Math.floor((float) letterNumber / 5) + 1);
66    int columnNumber = letterNumber % 5 + 1;
67    result += rowNumber + "" + columnNumber;
```

Now it is simple to encrypt once we have weeded out the special cases. Here we implement the formulae discussed earlier. Notice that we need to explicitly typecast `letterNumber` else we will be doing integer division. The concatenation of `rowNumber` and `columnNumber` added onto `result`.

3.2. Decryption

We will perform a letter-by-letter decryption. The formulas used in the decryption algorithm are the math formula 'reversed'
Let E_t be the cipher text having length of n
P_t be the plain text
S be the shift, where $S \in \mathbb{N}$
$P_t = P_1P_2P_3 \dots P_n$ where P_i is the i^{th} character of the plain text
The i^{th} character of the plain text is -
If the letter is between 65 and 90 (Uppercase)

$$P_i = \Big[(R_i - 1) \times 5 + C_i\Big] + 65$$

If the letter is between 97 and 122 (Lowercase)

$$P_i = \Big[(R_i - 1) \times 5 + C_i\Big] + 97$$

3.2.1. Code Implementation

The code implementation of decryption is also a translation of formulae into code. The only difference is that instead of the

```
107  String result = "";
108  key = generateCustomKey(key.replaceAll(" ", ""), "ABCDEFGHIKLMNOPQRSTUVWXYZ");
```

Here the `result` variable is initialized and the unique key is generated from the 'seed'.

```
110  for (int i = 0; i < encryptedText.length(); i += 2) {
111    int letterNumber = Integer.parseInt(encryptedText.substring(i, i + 2));
```

The `for`-loop jumps 2 because it has to go from coordinate to coordinate each of which are 2 characters apart. For example, to get from the first coordinate pair to the second coordinate pair, you have to increment by 2.

$$\overset{i}{4531}$$

```
111    int rowNumber = letterNumber / 10;
112    int columnNumber = letterNumber % 10;
113    result += key.charAt(--rowNumber * 5 + --columnNumber);
```

The `rowNumber` and the `columnNumber` variable is initialized to the correct values and then are used to locate the character in the key using the standard decryption formulae.

- Is the row numbers greater than 2? (or)
- If the row number is 2, then is the column number equal to 5?

Why do we need to do this? Because we have to omit mapping J. Every letter after 'J' in the ASCII table will have this correctional measure hence the nature of the `if`-statement. Recall that to map from rows and columns to ASCII we have to multiply the row by 5 and add the column

and then add 65. Take for example K. It has a square coordinate of 25 meaning that it will map to $65 + (5 \times (2-1) + 5) = 75$ which is the result we wanted since 75 is the ASCII for 'K'. If this correctional measure was not in place then we would get $65 + (5 \times (2-1) + (5-1)) = 74$ which is the ASCII value of 'J'.
However, for any element before 'J' this correctional formula doesn't apply. Take for example G. The square coordinate for G is 22. This maps to $65 + (5 \times (2-1) + (2-1)) = 71$ which is correct ASCII table value of G.
The result of this is then attached to `result`.

Alternate Keys

To use a Polybius Sqaure with a custom key, a special method can be written for the same :

```
1  protected static String generateCustomKey(String seed, String customAlphabet) {
2    String generatedKey = "";
3
4    // S1 : ENSURE THAT ONLY UNIQUE CHARACTERS GET ADDED TO generatedKey
5    for (int i = 0; i < seed.length(); i++)
6      if (!generatedKey.contains(seed.charAt(i) + ""))
7        generatedKey += seed.charAt(i);
8
9    // S2 : ATTACH THE REST OF THE LETTERS IN customAlphabet
10   for (int i = 0; i < customAlphabet.length(); i++)
11     if (generatedKey.indexOf(customAlphabet.charAt(i)) == -1)
12       generatedKey = generatedKey.concat(customAlphabet.charAt(i) + "");
13
14   return generatedKey;
15 }// end of String generateCustomKey(String, String)
```

The above method consumes a seed for the key. STEP1 makes sure that only the unique letters of `seed` get appended to `generatedKey`. After doing this we can attach the rest of the `customAlphabet` onto `generatedKey` as long as the letter doesn't already exist in `generatedKey`. For example, the seed "PLAYFAIR" gets mapped as

$$\text{PLAYFAIR} \implies \text{PLAYFIRBCDEGHKMNOQSTUVWXZ}$$

Suppose that you want it to be mapped as

$$\text{PLAYFAIR} \implies \text{PLYFAIRBCDEGHKMNOQSTUVWXZ}$$

then the only change to be made is in line 7. Replace `generatedKey` with `seed.substring(i + 1)`.

```
5      // STEP 1 : USE ONLY THE UNIQUE LETTERS FROM SEED
6      for (int i = 0; i < seed.length(); i++)
7        if (seed.substring(i + 1).indexOf(customAlphabet.charAt(i)) == -1)
8          generatedKey += seed.charAt(i);
9
```

To see these in action refer to Wheatstone-Playfair documentation and code.

4. Variants

There are several variants of the Polybius square cipher, so much so that Polybius Square is a type of encryption scheme. The list of variants is as follows :

- ADFGX Cipher
- Bifid Cipher
- Nihilist Cipher
- Tap Code
- Trifid Cipher
- Wheatstone-Playfair Cipher

We will discuss only the Tap Code, ADFGX and Nihilist here. Bifid, Trifid and Wheatstone-Playfair are complex enough to get their own code and documentations.

4.1. Tap Code

Tap Code also known as Knock Code is a simple modification to the Polybius Square cipher. Instead of numbers to represent the row and column coordinates, dots are used. This was used by Russian prisons of the czar and American prisoners in Vietnam to communicate between each other. They used it to communicate everything from what questions interrogators were asking (in order for everyone to stay consistent with a deceptive story), to who was hurt and needed others to donate meager food rations.

4.2. ADFGX Cipher

The ADFGX cipher is a simple modification of the Polybius cipher. Instead of using numbers (1-5) for rows and columns, it uses the letters A,D,F,G and X. It was invented by the German intelligence officer Fritz Nebel and first used in March 1918. These letters were chosen as such since they are very different from each other in Morse Code thus minimizing the possibility of operator error. [1]

4.3. Nihilist Cipher

The Nihilist cipher was a symmetric key variant of the Polybius square used by the Russian nihilists to plan terrorist attacks against the tsarist regime in imperial Russia. The square coordinates of the key and the plaintext would be added to get the resultant cipher text. So for example if I had the plaintext "IWANTHIMDEAD" with a key "ALCAPONE" then we would get

PT : 24 52 11 33 44 23 24 32 14 15 11 14

KY : 11 31 13 11 35 34 33 15 11 31 13 11

CT : 35 83 24 44 79 57 57 46 25 46 24 25

This essentially makes it a numerical version of the Vignère cipher. There is no fractionation achieved. Hence a modified Kasiski examination will work on cracking this cipher.[2]

5. Further Reading

Fractionating Ciphers

Read more about Fractionating ciphers on Crypto Corner

Read the Bifid and Trifid documentation to learn more about fractionation.

[1] Read more about the ADFGX cipher on its Wikipedia page

[2] To read about an upgraded model of the Nihilist cipher (which implements fractionation), refer to the Bifid cipher.