Vigenere

How to run the program

The program is written in java, so the JVM is needed to run it. It can be installed from https://www.oracle.com/java/technologies/downloads/

Aftwerwards, you need to compile the Main.java file and execute it:

```
javac Main.java
java Main
```

What we found

> Column transposition key : `[1, 5, 6, 3, 0, 4, 2]`
>
> Vigenere key: `oscarwilde`
>
> Deciphered text: `thestudiowasfilledwiththerichodorofrosesandwhenthelightsummer…`

Approach used

The approach we used to crack Vigenere is the frequency analysis of the distributions. The program analyzes all the possible combinations (with the help of Java Threads to reduce the execution time) and select the one which is the most likely to be the correct one.

The core program consists in two parts: in the first one, it focuses to solve the column transposition problem, rearranging the given text; in the second it focuses on the deciphering of the Classic Vigenere cipher.

1. **Launching the execution:**

   ```
   public class Main
     public static void main(String[] args)
       String ciphered="****** text to cipher ******".toLowerCase();
       Utils.createMatrix();
       Vector<Executer> threads = new Vector<Executer>();
       for(int i=6;i<=10;i++) {
         Executer e=new Executer(i,ciphered);
         threads.add(e);
         Thread t=new Thread(e);
         t.start();
   ```

   In the class `Main`, for each key length create an execution thread, which helps in improving the efficiency of the program and reducing the execution time. Each thread tries all possible column permutations for its own key length, and each one tries to crack Vigenere.

2. **Single column transposition:**

   ```
     public void run()
           System.out.println("\n[+]Cracking with permutation: " + lengthToCheck);
           List<List<Integer>> permutations = new ArrayList<>();
           Utils.generatePermutationsForNumber(lengthToCheck, permutations);
           for (List<Integer> permutation : permutations){
               String rearrangedText = Utils.rearrangeText(ciphered, permutation );
   ```

   When `Executer` it's launched, in `run()` the function `Utils.generatePermutationsForNumber(int n, List<List<Integer>> allPermutations)` It's recursively called to generate all the possible permutations of a list of a specific length.

   Iterate through each permutation, calling in the `Utils` class the function `rearrangeText(String cipheredText, List<Integer> permutation)`. It rearranges the ciphered text based on the current permutation. Here's a step-by-step breakdown of its operation:

   1. A copy of the input text is created using a StringBuilder
   2. The function calculates how many extra characters there are (as the text length might not be a multiple of the permutation size) and how many spaces need to be inserted, as well as the size of each column in the final rearranged

text.

3. Two lists, `firstColumn` and `lastColumn`, are created. The first list will contain column indices with an extra character, while the second will contain column indices with less characters.
4. The function then iterates over the permutation, filling `firstColumn` and `lastColumn` with the appropriate indices.
5. It then goes through the text, inserting spaces at the right places to separate columns and adjusting the current position in the text accordingly.
6. A `columnMap` list is created to store the actual columns of characters. For each index in the permutation, the corresponding characters are extracted from the text and added to the `columnMap`.
7. Finally, the rearranged text is reconstructed by concatenating the columns in the right order. The function iterates through each character of each column, adding it to the result string only if it is not a space.

The output of the function is a string where the original text has been rearranged according to the input permutation.

3. **Vigenere classic**

The following part of the code focuses on cracking the classic Vigenere cipher:

```
String foundKey;
String deciphered;
double IC = 0.0;
for (int j = 6; j <= 10; j++)
    foundKey = Cracking.crack(rearrangedText.toCharArray(), j );
    deciphered = Utils.decipher(rearrangedText, foundKey);
    IC = Utils.calculateIC(deciphered);
    if(IC >= 0.06){
        System.out.println("\n[+]KEY FOUND:" +foundKey+", IC:" +IC);
         System.out.println("[+]PLAIN TEXT: \n"+deciphered);
```

Since we know that the key it's in the range of 6 - 10 number of letters, the loop it's set within that range. It contains the core of the cracking process, which consists in the 3 following functions:

1. `foundKey = Cracking.crack(rearrangedText.toCharArray(), j )` : we try to crack the key with the current length j
2. `deciphered = Utils.decipher(rearrangedText, foundKey):` we decipher the text with the found key
3. `IC = Utils.calculateIC(deciphered)` : we calculate the IC of the deciphered text putting two texts side-by-side and counting the number of times that identical letters appear in the same position in both texts. This count, either as a ratio of the total or normalized by dividing by the expected count for a random source model, is known as the **index of coincidence**

At the end the calculated IC is compared with the English IC value (we used the not normalized value, which is 0.0667). If it's > 0.06 , the solution we found has great chances to be the correct one, therefore we print the result.

The cracking process

As mentioned before, the approach we used to find the key is based on frequency analysis. A vector called `baseVector[]` is been used to store the frequencies of the letters in the english vocabulary:

$$\begin{array}{cccccccc} a & b & c & d & e & ... & y & z \\ 0.082 & 0.015 & 0.028 & 0.043 & 0.127 & ... & 0.020 & 0.001 \end{array}$$

`Cracking.crack(rearrangedText.toCharArray(), j )`

1. `makeFrequencyMatrix()` creates the matrix of frequency vectors. each row is a vector shifted right by one: row 0 is baseVector, row 1 is baseVector shifted right by one, row 2 is baseVector shifted right by two, etc. The index of the row represents, therefore, the number of shifts of each letter, and since in Vigenere permutations all the letters are shifted by the same fixed number of positions, this is gonna be useful in few steps.

$$\begin{array}{llcccccc} v_0 & = (& .082, & .015, & .028, & ..., & .020, & .001) \\ v_1 & = (& .001, & .082, & .015, & .028, & ..., & .020) \\ v_2 & = (& .020, & .001, & .082, & .015, & .028, & ...,) \\ \vdots \end{array}$$

2. `nthLetters(ciphered, keyLenght)`

   In Vigenere it's known that the each letter has been shifted by the same amount:

```
C O D E T H E O R Y P R O J E C T -> text
K E Y K E Y K E Y K E Y K E Y K E -> key
```

By creating the following matrix with our `nthLetters` function (as shown below)

```
ciphered = V V H Q W V V R H M
keylength = 3
------|   function output (rows an column are swapped)
V V H |   V Q V M
Q W V |   V W R
V R H |   H V H
M     |
```

we're able to identify the letters encrypted with the same letter of the key.

The idea is to compare the frequency vector of each row with the all possible permutations of the baseVector, in order to find the most similar one.

💡 Finding the two vectors u and v_i that most closely match, means to find the two vectors with the smallest angle between them.

$$u \cdot v = |u||v| \cos \theta$$

Since smaller angles produce larger cosine values and since the magnitude of the denominator is the same for every v_i, as the same 26 numbers are involved each time (just in different orders), we can simply seek the two vectors u and v_i whose dot product is **largest.**

Concretely, the program loops through each row of the matrix computing the following operations:

3. `frequency = makeFrequencyVector(matrix[i])` : it creates a vector of the frequency of each letter in the row (n. of occurrences / 26). It will be like

$$u = (0, 0, 0.104, 0.0149, 0.0149, 0.0298, ...)$$

4. `permutation = checkPermutation(frequency)` : it calculates the vectorial product of the frequency vector of the row and the i row of the matrix, selecting the index of the row with the max value of such a product. If we found an index 3 for example, means that the letters have been shifted of 3 letters to the right and the keyword starts with D.
5. `key=key+(char)('a'+permutation)` : adds the letter of the key that corresponds to the permutation.

The key of length j is found and returned to the caller. The program will try to find a solution with that key.

Sources:

https://mathcenter.oxford.emory.edu/site/math125/vigenereCipher/#:~:text=The Vigenere Cipher is a,-for-letter substitution ciphers