HACETTEPE UNIVERSITY

COMPUTER ENGINEERING DEPARTMENT

BBM465 – Information Security Lab

Group ID: 21

Ufuk Ağaya & Ceyhun Arda Gök

Pair ID Numbers: 2210356064 & 2220356168

Experiment Subject : Basic Ciphers

T.A.s : Ali Baran TAŞDEMİR, Sibel KAPAN

Problem Definition

In this project it is expected to implement 3 basic encryption ciphers, their decryption methods and break methods for these encrypted messages.

These methods are Caesar method, Affine method and Monoalphabetic method. We will explain how they work to encrypt the messages and break them.

Approach and Solution of Problem

To address this problem, we will implement three classical encryption methods: the Caesar cipher, the Affine cipher, and the Monoalphabetic Substitution cipher. For each method, we will provide functionality for encryption, decryption and cryptanalysis to break the ciphers. The solution will be divided into three main parts:

1) Caesar Cipher:

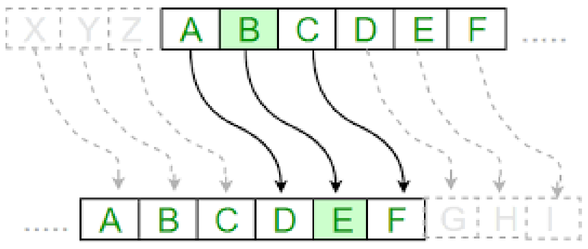
It's a classic technique for encrypting messages that dates back to the Roman Empire. It's main purpose is to encrypt messages by using a new alphabet that we create with the shifting method. For example, if we shift our alphabet by 4, 'e' is replaced by 'a' in this technique. We have implemented 3 methods:

a) `encrypt_caesar(plaintext, shift)`

- **Description**: Shifts each letter in the plaintext by a fixed number, defined by the shift parameter.
- We use following command to encrypt plain.txt using shifting by 13:

```
python ciphers.py caesar plain.txt e -s 13
```

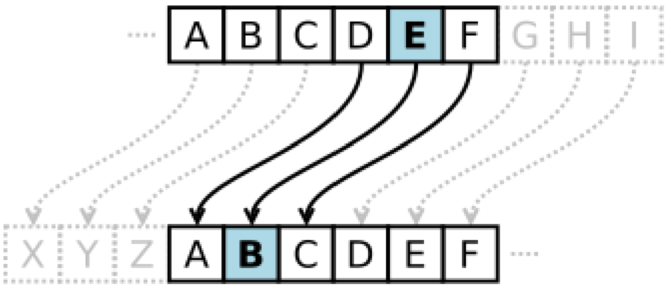
- Also this method write the coded message to a new txt file named coded_caesar.txt, we will use this txt file to break the cipher in the upcoming decrypt_caesar and break_caesar methods.
- Here is a visualisation of encryption process of shifting 3:



- While the original method uses ASCII values to shift letters, we added an optional method which shifts the English alphabet with the given shift amount, we obtain a shifted alphabet, which later we use to replace the letters in the plaintext based on their index.

b) `decrypt_caesar(ciphertext, shift)`

- **Description**: Reverses the shift applied during encryption to retrieve the original plaintext. Uses coded_caesar as a parameter.
- We use the following command to decrypt coded_caesar.txt using the same shift parameter that used in the encryption method:
```
python ciphers.py caesar coded_caesar.txt d -s 13
```
- It will print the decrypted text directly to the console.
- Here is a visualisation of the decryption process of the previous operation:



- Just like the encryption, there is an optional method for decryption, which uses the same method as encryption but shifts the alphabet in opposite direction.

c) `break_caesar(ciphertext)`

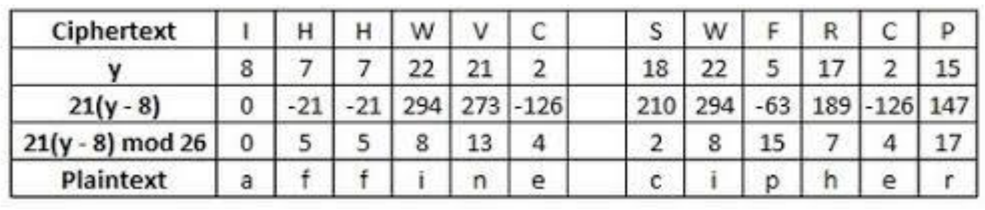
- **Description**: The `break_caesar(ciphertext)` function uses a brute-force approach to decrypt messages encrypted using the Caesar cipher. It systematically tries all possible shifts (1 to 25) of the ciphertext to recover the original plaintext. The fundamental principle behind this method is the shifting nature of the Caesar cipher, which guarantees that the correct plaintext can always be found within a maximum of 25 attempts, regardless of the length of the ciphertext.
- To break the ciphertext using this method, we use the following command:
```
python breakers.py caesar coded_caesar.txt
```
*the parameter dictionary.txt is takes the text file that contains the dictionary that we want to use for frequency analysis in the following methods. So it's not even necessary for this method but not for the format.
- It will create a new text file named break_caesar.txt that contains the cracked code.

2) Affine Cipher:

It is a monoalphabetic substitution cipher that applies a mathematical formula to each letter. We can determine which function was used to encrypt the text. If we know the function used to encrypt the text, we can easily decrypt it using the same function. This method is more secure than the Caesar method because it requires 2 different parameters to get the correct message.

a) `encrypt_affine(plaintext, a, b)`

- **Description**: Each letter is converted to its numeric equivalent an then encrypted using the formula $E(x)=(ax+b)mod\ mE(x) = (ax + b)\ \backslash mod\ m$E(x)=(ax+b)modm.
- We use following command to encrypt the message plain.txt using this method:
  ```
  python ciphers.py affine plain.txt e –a 3 –b 5
  ```
  * As we have 2 parameters, this message is harder to break than the Caesar code
- Also this method writes the ciphertext to a new txt file called coded_affine.txt, we will use this txt file to break it in the upcoming decrypt_affine and break_affine methods
- Here is a visualisation of the affine cipher process:



Ciphertext	I	H	H	W	V	C		S	W	F	R	C	P
y	8	7	7	22	21	2		18	22	5	17	2	15
21(y - 8)	0	-21	-21	294	273	-126		210	294	-63	189	-126	147
21(y - 8) mod 26	0	5	5	8	13	4		2	8	15	7	4	17
Plaintext	a	f	f	i	n	e		c	i	p	h	e	r

Same as the caesar cipher, we added an additional method to encrypt and decrypt. By using the formula, we calculate new indexes for each letter in english alphabet and then replace the characters with the characters in the obtained indexes.

b) `decrypt_affine(ciphertext, a, b)`

- **Description**: Uses the inverse of the encryption formula to retrieve the original plaintext. Uses parameter coded_affine.txt as parameter.
- We use the following command to decrypt coded_affine.txt using the same shift parameter used in the encryption method:
  ```
  python ciphers.py affine coded_affine.txt d -a 3 –b 5
  ```
- This will print the decrypted text directly to the console.

c) `break_affine(ciphertext)`

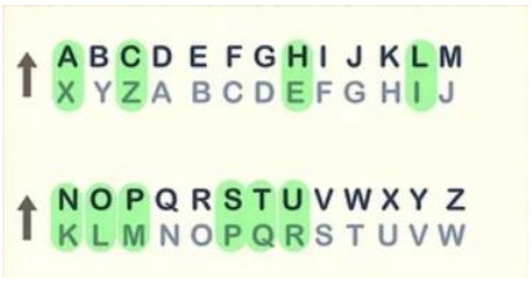
- **Description**: Tries all possible values of a and b, checking the decrypted output against a dictionary. For example if the limit of the function is 5, it will try a maximum of 25 different combinations. And when a fully meaningful text is appears. It will stop and return that as the answer.
- We will use the following command to decrypt coded_affine.txt, using the same shift parameter used in the encryption method:
  ```
  python breakers.py affine coded_affine.txt
  ```
- It will create a new text file named break_affine.txt that contains the cracked code.

3) Mono-alphabetic Cipher:

This method replaces each letter in the plaintext with a corresponding letter from a fixed substitution alphabet. This method provides information security.

a) `encrypt_mono(plaintext, key)`

- **Description**: Maps each letter in the plaintext to a letter in the substitution key. This way we can have better security in the encryption.
- We use the following command to decrypt coded_mono.txt, using the same shift parameter used in the encryption method:
  ```
  python ciphers.py mono plain.txt e -k QWERTYUIOPASDFGHJKLZXCVBNM
  ```
- Again, this method will write the ciphertext message to a new text file called coded_mono.txt, we will use this file to break it in the upcoming decrypt_mono and break_mono methods.
  The encrypt_mono method can be visualised like the picture below as an example:

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

  Similar to the other two, we added an optional method for the mono-alphabetic cipher. The first method finds the index of the characters using ASCII values, while the second method directly checks the letter’s position in the key.

b) `decrypt_mono(ciphertext, key)`

- **Description**: Uses the same mapping in reverse to retrieve the original plaintext. Since we have the same mapping key, we can directly reverse the mapping directly, so we always get the correct plaintext.
- We use the following command to decrypt coded_mono.txt using the same key parameter that used in the encryption method:

```
python ciphers.py mono plain.txt d -k QWERTYUIOPASDFGHJKLZXCVBNM
```

- Since we are using the same key, we will get the correct plaintext.
- The optional method we implemented locates the index using the key, then retrieves the decrypted character from a stored string of the English alphabet. While our main method reverses the key by using ASCII values to find the decrypted character.

c) `break_mono(ciphertext)`

- **Description**: attempts to decrypt a monoalphabetic cipher by analyzing the frequency of letters in the ciphertext and matching common patterns in English. It uses frequency analysis and regular expressions to identify potential letter mappings.
- We use the following command to break coded_mono.txt using frequency analysis and letter prediction techniques which we will mention later:

```
python breakers.py mono coded_mono.txt
```

- The function accesses dictionary.txt with the directory path without taking any parameters.
- It will create a new text file named break_mono.txt that contains the cracked code.

- **The Process Steps**

- **Preparing**:
    - We get a set of all the letters from the file we want to thanks to the `load_dictionary(file_path)` function.
    - Thanks to `analyze_letter_frequency(dictionary) we can` get the frequency order of any dictionary that we want to analyse its frequency characteristic.

- **Analyze 3-letter words:**
    - We have selected all 3-letter words from the ciphertext and sorted them by the number of times they have been used. We care about the first 2 for the technique we are going to use. For this technique we know the most common letters, but we also know the most common words. It's "and" and "the" for 3 letter ones. Because we know that most of the 3 letter words are 2 of them. We also have the knowledge that 'a' is the most common letter around the word 'and', 'e' is the most common letter around the word 'the'. With this information we can assume that if one of the 2 most common words has "e" at the end. We can assume that it's "the" and the other word is "and". Otherwise, we know it's "and" and the other is "the".

- **Pattern matching with Regex:**
    - For each word, create a regex pattern based on known letters in key_guess and uses "." for unknown letters. Matches this pattern against the dictionary to find possible word matches and stores maching words in the `maching_words` list.

- **Update key_guess:**
    - If only one matching word is found, it updates `key_guess` with new letter mappings. If there are multiple matching words, it filters out words with already known letter mappings and updates accordingly.

- **Check for completeness:**
    - If all 26 letters are mapped in `key_guess`, the process stops.

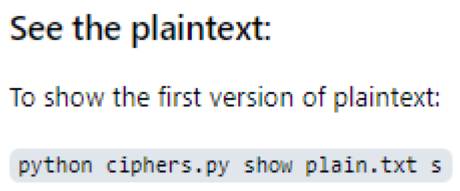
- **Generate the decrypted text:**
    - Uses the final version of `key_guess` to get the decrypted message using the ciphertext in the end.

4) Extra Features:

a) show command:

We have added a command called show to the ciphers.py file. This way, if plaintext or ciphertext is to be displayed, this command can be used to display its contents.

Here is a simple use:



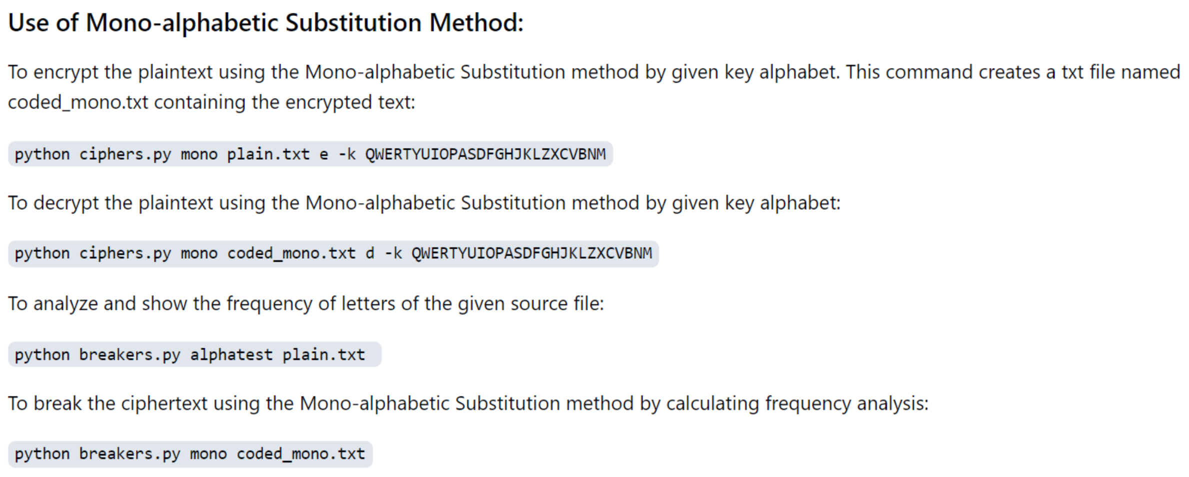
See the plaintext:

To show the first version of plaintext:

```
python ciphers.py show plain.txt s
```

b) `alphatest` command:

We have added a command called show to the ciphers.py file. This way, if plaintext or ciphertext is to be displayed, this command can be used to display its contents. Here is simple use it for Mono-alphabetic Substitution method:



Use of Mono-alphabetic Substitution Method:

To encrypt the plaintext using the Mono-alphabetic Substitution method by given key alphabet. This command creates a txt file named coded_mono.txt containing the encrypted text:

```
python ciphers.py mono plain.txt e -k QWERTYUIOPASDFGHJKLZXCVBNM
```

To decrypt the plaintext using the Mono-alphabetic Substitution method by given key alphabet:

```
python ciphers.py mono coded_mono.txt d -k QWERTYUIOPASDFGHJKLZXCVBNM
```

To analyze and show the frequency of letters of the given source file:

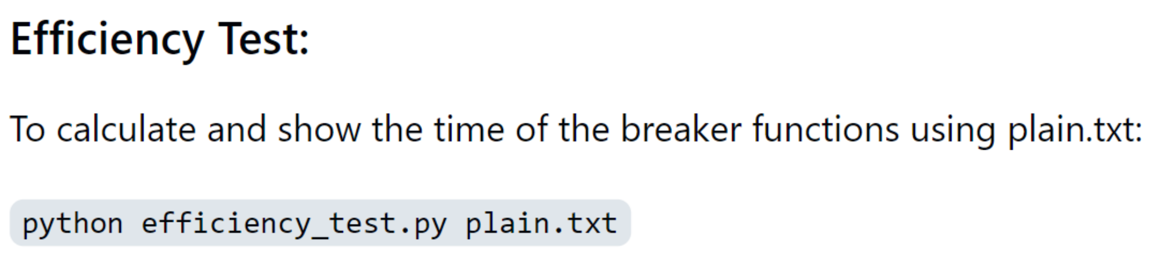
```
python breakers.py alphatest plain.txt
```

To break the ciphertext using the Mono-alphabetic Substitution method by calculating frequency analysis:

```
python breakers.py mono coded_mono.txt
```

c) efficiency_test.py:

Thanks to this script, we have the freedom to measure the speed of our break methods. This helps to make the code faster. Here is a simple use for testing:



Efficiency Test:

To calculate and show the time of the breaker functions using plain.txt:

```
python efficiency_test.py plain.txt
```

Conclusion

This assignment covered the implementation and cryptanalysis of classic ciphers, specifically the Caesar, Affine, and Mono-alphabetic ciphers. Understanding these ciphers provides foundational knowledge for further studies in cryptography and security. Overall, this work provided valuable insights into the fundamentals of cryptography, equipping us with skills that will be beneficial as we advance in securing communications in the digital world.

Resources:

- https://www.geeksforgeeks.org/ord-function-python/
- https://www.w3schools.com/python/python_regex.asp
- https://www.geeksforgeeks.org/caesar-cipher-in-cryptography/
- https://www.geeksforgeeks.org/implementation-affine-cipher/
- https://blog.eduonix.com/2018/11/monoalphabetic-polyalphabetic-cipher-in-python/