Historic Crypto

Questions:

1. What are the advantage and disadvantage of these classical cipher?
    a. These ciphers are easy to implement and run very quickly.
2. Will you consider using these classical ciphers for your content protection? Why or why not?
    a. No, since they are easy to implement, they are easy to crack. Most of these are pretty much useless because of how easy it is for modern computers to brute force them.
3. Discuss your experience during the crack implementations.
    a. Cracking the Caesar was easy once we started using letter frequencies. We originally tried it by checking to see if there were English words in the output, but this became an issue once we saw the files with no spaces. Letter frequency analysis worked well though.
    b. Mono-Alphabetic Substitution was the hardest of the three. In the end the program we developed could only approximately get the correct solution and needed some correction at the end.
    c. Vigenère was originally difficult because we used the key length solving algorithms found on Wikipedia. These were not very accurate, so it caused issues that made the rest of the program difficult to solve. Finally, we found the Twist+ algorithm which could very accurately find the key size consistently. After the key size the rest was the same as the Caesar cipher, so not much work.

Caesar:

Description:

The Caesar Cipher works by rotating every character in the plaintext by a specific amount to produce the ciphertext. For example, if the key is 5 then `a` -> `f`, `z` -> `e`, etc.

Encryption Implementation:

To implement this in Python, loop through the entire input and replace each character with the appropriate offset character. This can be done by creating a dictionary with the plaintext characters as keys and the corresponding ciphertext characters as values.

Decryption Implementation:

For each of the 26 possible shifts (including no shift) get the letter frequencies statistics and compare them with the baseline English letter frequencies. Pick whichever one has the closest correlation. Without any adjustment this works for all the provided ciphertexts.

Vigenère:

Description:

Vigenère works by having some key phrase that is used to offset all the characters in the plaintext. If the key phrase is shorter than the plaintext then it can be added to itself until they are both the same length (i.e. key: 'dog' text: 'Hello, world!', long_key: 'dogdo, gdogd!`). The longer key skips over non-letter characters and shifts letters by whatever letter is in the key (i.e. `d` is a shift of three).

Encryption Implementation:

To implement this in Python take the key and make it the same length as the plaintext. Then iterate through both the plaintext and the long key at the same time. Every time you encounter a letter shift it by the current letter from the key.

Decryption Implementation:

The way we implemented the cracking was using the Twist+ algorithm to find the key length then using letter frequency analysis to find each of the letters of the key.

Twist+

Twist+ is really complicated so we've included the link to the research paper about it.

Mono-Alphabetic Substitution:

Description:

Mono-Alphabetic Substitution works by mapping each character from the plaintext to a random character in the ciphertext. Each letter consistently maps to another letter but there is no order.

Encryption Implementation:

The key for a Mono-Alphabetic Substitution Cipher is the shuffled alphabet which maps its plaintext to cipher text. In order to implement this, we can create a dictionary that has plaintext characters as its keys and the ciphertext characters as its values. Then just iterate through the input and replace all the letters.

Decryption Implementation:

This one was tricky to decrypt. Even using letter frequency analysis, it is very difficult to find the right mapping because there are a total of 26! possibilities. The approach we used was a genetic algorithm. The way it works is to generate 50 totally random keys and improve on each of them until we can't get any better. We do this by randomly swapping two characters in the key and checking to see if that makes the resulting ciphertext more "Englishly". In this case we checked by comparing the trigraph frequency between the decoded text and the expected English values. Then we keep swapping characters until we have made 5000 useless swaps in a row. At this point we can conclude that we probably won't get much better, so we move to the

next key. Once we have all 50 keys, we can take the best one of the set and present it for evaluation. At the end the program provides the decoded text according to the key and lets the user swap letters manually until it gets to a better state.