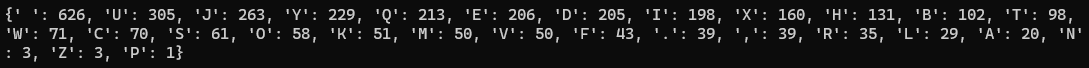
Ng Jia Sing 1006648

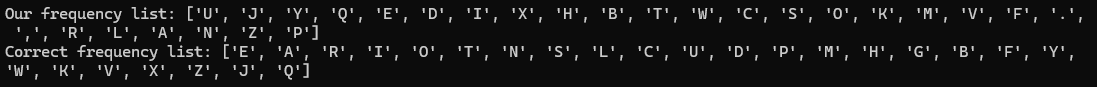
Part I: Substitution Cipher

**How to run:** python3 ex1.py -i story\_cipher.txt

Based on the ciphertext, we first count the occurrence of each character used. We then create a dictionary in python sorted in descending order based on the number of times each character appeared.

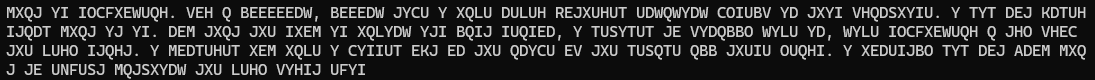


As we know that the spaces do not map to other characters, we can ignore the space character. Barring the whitespace character, we then map each character from our current dictionary to the frequency list of the English alphabet as shown below.



As our frequency list contains 27 characters while the correct frequency list only has 26, we will only map the first 26 characters of our list to the English alphabet. Upon mapping, we will convert the character to a lower case.   
For example: ‘U’ 🡪 ‘e’ ‘J’ 🡪 ‘a’ ‘Y’ 🡪 ‘r’ …  
**Therefore, the character P will remain unmapped as ‘P’ in our ciphertext.** This is because we are mapping 27 wrong characters onto 26. One of the wrong characters (P in this case) remains unmapped.

Before mapping, first 300 character of cipher text:



After mapping, our entire cipher text:

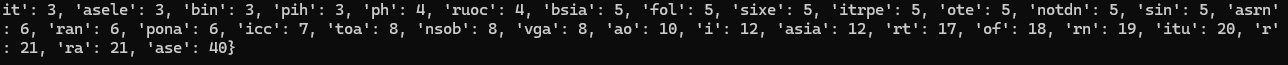
A black screen with white text

Description automatically generated

We observe that the current cipher text still does not make sense and more fine tuning is required. Each time manually we map a letter, we will turn it back to uppercase to show that we’ve “correctly” identified say character. We’ll fine tune our text using the entire cipher text, frequency of unique words and a list of unique words sorted in ascending order of length.

FINE TUNING

We observe that the last character of our cipher text is ‘w’ while we expect it to be ‘ . ’. Furthermore, from the frequency of each unique word found in the text and see that ‘ase’ appears the most. Thus we map ‘ase’ to ‘THE’.



From our new texts, several sentences start with r. we map ‘r’ to ‘I’.  
We then have two single character words “I” and ‘i’, thus we map ‘i’ to ‘A’.  
Also we have two terms “coooooadk” and “coooad”, its highly unlikely that ‘k’ here is an actual alphabet so we map it to ‘ , ’.

From here, we observe our texts to find familiar words, map them, and regenerate our cipher texts. Our words used are:  
“HEARTn” 🡪 “HEARTS”  
“HAxE” 🡪 “HAVE”  
“bHAT” 🡪 “WHAT”   
“vETTER” 🡪 “BETTER”   
“acc” 🡪 “ALL”  
“fol” 🡪 “FOR”  
“TIpE” 🡪 “TIME”  
“LOOOOOtd” 🡪 “LOOOOONG”  
“BOTHEREu” 🡪 “BOTHERED”  
“MhSELF” 🡪 “MYSELF”  
“FRANmHISE” 🡪 “FRANCHISE”  
“THROgGHOgT” 🡪 “THROUGHOUT”  
“zNOW” 🡪 “KNOW”  
“qOURNEY” 🡪 “JOURNEY”  
“POMBIE” 🡪 “ZOMBIE” \*Our initial character P is unmapped\*   
“EjyECT” 🡪 “EXPECT”

Part 2: Compromising OTP Integrity

To modify the ciphertext with no knowledge of the OTP, we simply need access to the following:

1. Original plaintext, m
2. Original ciphertext, c
3. Our modified plaintext, m’

Workings:  
As we are encrypting with XOR, we can make use of the XOR function to modify the ciphertext as follows:

m ^ OTP = c

we XOR both sides of equation with m to get,

m^ OTP ^ m = c^ m 🡪 OTP = c^m

hence we can create a modified plaintext with:

m’ ^ OTP = m’ ^ (c ^ m) = m’ ^ c ^ m = c’ (c’ being the modified ciphertext)

In code we have:

