Introduction

The Monoalphabetic Substitution Cipher is a type of cryptoanalysis that replaces each letter of the plaintext using a fixed substitution (A becomes X, B becomes Y). Also, we need a fixed key that maps each letter in the alphabet to a unique replacement letter. The key space for a monoalphabetic cipher with 26 letters is 26. Actually, in the Monoalphabetic Cipher, we map our plain alphabet (using a key) to the cipher alphabet and convert it again for the Decryption process. Moreover, one of the common attacks is frequency analysis (counting letters) is the study of the frequency of letters or groups of letters in a ciphertext.

Overview

This report presents the implementation of the Monoalphabetic Substitution in Python, and uses frequency analysis to break the ciphertext, covering 2 phases:

- Monoalphabetic Substitution: Initialization, setting a unique alphabet, generating a random key, running the encryption and decryption functions, and displaying output.

- Frequency Analysis: Attempting to break the cipher by counting and testing all words and groups of letters, and displaying output.

1. Monoalphabetic Substitution Process

1.1 Encryption

In this phase, each letter of the plaintext is replaced with another letter from the alphabet in lowercase format we set in our code. So, we are prepared to generate the key randomly with the alphabet we add, and we also have the text (from the path). Finally, we are ready to make the transition (mapping method) to substitute letters with another one.

1.1.1 Initialization

Defining Paths and reading text from the location we set (/opt/mono-cipher/plan.txt)

```python
if __name__ == "__main__":

    print("This is Task2-mono.py")
    print("Kimia Sadat Karbasi - Student ID Number ='60393958'")

    #Plain_text path
    file_path = "/opt/mono-cipher/plain.txt"
    cipher_text_path = "/opt/mono-cipher/cipher.txt"
    decrypted_text_path = "/opt/mono-cipher/decrypted.txt"

    #Reading plain.txt
    try:
        with open(file_path, "r", encoding="utf-8") as f:
         text = f.read().lower()
    except Exception as e:
        print(f"The file was not found")
        exit()
```

Figure 1 – Defining Paths and Reading Plain Text

1.1.2 Importing Library

Using random (generating a key randomly) and String (converting output to string format).

```python
#KimiaSadatKarbasi-SID60393958
#Import library
import random
import string
```

Figure 2 - Importing libraries

1.1.3 Adding characters

Using string.asci_lowercase to get a sorted alphabet (from a to z).

```python
#Adding Charachters
alphabet = string.ascii_lowercase
```

Figure 3 – Adding Characters

1.1.4 Generating a Random Key

We use the length of the alphabet to generate a key, which is a non-repeating unique.

```python
#Generating Key
key_list = random.sample(alphabet, len(alphabet))
#Convert key to a string
key = ''.join(key_list)
print( f"Generated Key: {key}")
```

Figure 4 – Generating a Random key

1.1.5 Encryption Function

It takes three parameters: text (input plain text), alphabet (the original alphabet string), and a random key we generate. We are able to produce a mapping table to translate each letter in the alphabet to a corresponding letter in the key. Then it will apply our substitution from the translation table we used.

For example: It means that when the letter ‘C’ becomes ‘A’, every ‘C throughout the Text will be translated to A.

```
#Reading plain.txt
try:
    with open(file_path, "r", encoding="utf-8") as f:
     text = f.read().lower()
except Exception as e:
    print(f"The file was not found")
    exit()

print("Plain text:", text)
cipher_text = mono_encrypt(text, alphabet, key)
cipher_text_lower = cipher_text.lower()
```

Figure 5 - Reading Plain Text

```
def mono_encrypt(text, alphabet, key):
    # maketrans creates a mapping between characters from alphabet to key
    translation_table = str.maketrans(alphabet, key)
    print("Translation_table", translation_table)
    return text.translate(translation_table)
```

Figure 6 – Mono-Encryption Function

1.2 Decryption Function

After adding libraries and the file path to read the cipher.txt, encrypted by the mono-encrypt function, we are going through the decryption function to return the original text.
It would define a function that takes three inputs: text, alphabet, and key.

- The text we want to decrypt.
- The original alphabet we set in the first part of the code.
- The substitution and the unique key that was used to encrypt the text.
- The goal of this function is to reverse the encryption process and is going to return the original plain text.
- In the encryption, we map (alphabet --> key), but in the decryption, we map (key --> alphabet), actually the reverse one.

```
decrypted_text = mono_decrypt(cipher_text, alphabet, key)
```

Figure 7 – decrypted Text

```
def mono_decrypt(text, alphabet, key):
    #maketrans creates a mapping between characters from key to alphabet
    reverse_translation_table = str.maketrans(key, alphabet)
    print("Translation Table:", reverse_translation_table)
    return text.translate(reverse_translation_table)
```

Figure 8 – Mono-Decryption Function

1.3 Display Output

Here is the result of my encryption and decryption monoalphabetic code on the Linux server machine. I implemented a main section, reading my file, and all those functions to be executed respectively.

1.3.1 Key Generation

```
root@Docker-PAM:/opt/mono-cipher# vim Task2-mono.py
root@Docker-PAM:/opt/mono-cipher# rm -rf Task2-mono.py
root@Docker-PAM:/opt/mono-cipher# vim Task2-mono.py
root@Docker-PAM:/opt/mono-cipher# python3 Task2-mono.py
This is Task2-mono.py
Kimia Sadat Karbasi - Student ID Number ='60393958'
Generated Key: vsljyundhkwxrofgbtzapqeimc
```

Figure 9 - Key Generation

1.3.2 Plain Text Output

```
Plain text: "the clockmaker of rothenburg"
in the heart of bavaria, nestled between rolling green hills and
cobblestone streets, sat the town of rothenburg ob der tauber.
known for its perfectly preserved medieval walls and timber-framed
houses, the town looked like a postcard from the past. but what truly
gave rothenburg its charm was the little clock shop on
schmiedgasse street, run by the old clockmaker, herr baumann.
herr baumann had lived in rothenburg all his life. his small shop,
zeit & seele—time and soul—was filled with all kinds of clocks:
cuckoo clocks with dancing figurines, grandfathers with deep chimes,
and delicate pocket watches that glimmered like stardust. people
came from all over germany—and even farther—to see his creations.
but what they didn't know was that herr baumann's clocks held more
than gears and springs. they held memories.
it was said that when herr baumann built a clock, he carved into it a
tiny piece of someone's story. a couple in love, a soldier coming
home, a child's laughter—he somehow captured these moments, and
his clocks ticked with life, echoing not just time but feeling.
one chilly april morning, a young boy named lukas wandered into
the shop. he was ten, wearing a red scarf too big for his neck, and
had curious eyes that never stopped moving.
"guten tag," said herr baumann with a smile. "looking for
something?"
lukas shook his head. "just looking. papa says you make magic
clocks."
herr baumann chuckled, kneeling down so his eyes met lukas's.
"magic? well, maybe a little."
the boy's smile faded. "do you make clocks that can go backward?"
the clockmaker blinked. "backward?"
```

Figure 10 - Plain Text Output

1.3.3 Encrypted Text

```
Translation_table {97: 111, 98: 108, 99: 113, 100: 120, 101: 104, 102: 105, 103: 100, 104: 121
, 105: 106, 106: 109, 107: 98, 108: 119, 109: 102, 110: 117, 111: 114, 112: 122, 113: 110, 114
: 112, 115: 115, 116: 118, 117: 99, 118: 97, 119: 116, 120: 107, 121: 103, 122: 101}
Translation Table: {111: 97, 108: 98, 113: 99, 120: 100, 104: 101, 105: 102, 100: 103, 121: 10
4, 106: 105, 109: 106, 98: 107, 119: 108, 102: 109, 117: 110, 114: 111, 122: 112, 110: 113, 11
2: 114, 115: 115, 118: 116, 99: 117, 97: 118, 116: 119, 107: 120, 103: 121, 101: 122}
Encrypted text saved to: /opt/mono-cipher/cipher.txt
```

Figure 11 – Translation Table

```
Encrypted Text: "vyh qwrqbfobhp ri prvyhulcpd"
ju vyh yhopv ri loaopjo, uhsvwhx lhvthhu prwwjud dphhu yjwws oux
qrllwhsvruh svphhvs, sov vyh vrtu ri prvyhulcpd rl xhp voclhp.
burtu irp jvs zhpihqvwg zphshpahx fhxjhaow towws oux vjflhp-ipofhx
yrcshs, vyh vrtu wrrbhx wjbh o zrsvqopx iprf vyh zosv. lcv tyov vpcwg
doah prvyhulcpd jvs qyopf tos vyh wjvvwh qwrqb syrz ru
sqyfjhxdossh svphhv, pcu lg vyh rwx qwrqbfobhp, yhpp locfouu.
yhpp locfouu yox wjahx ju prvyhulcpd oww yjs wjih. yjs sfoww syrz,
ehjv & shhwh-vjfh oux srcw-tos ijwwhx tjvy oww bjuxs ri qwrqbs:
qcqbrr qwrqbs tjvy xouqjud ijdcpjuhs, dpouxiovyhps tjvy xhhz qyjfhs,
oux xhwjqovh zrqbhv tovqyhs vyov dwjffhphx wjbh svopxcsv. zhrzwh
qofh iprf oww rahp dhpfoug-oux hahu iopvyhp-vr shh yjs qphovjrus.
lcv tyov vyhg xjxu'v burt tos vyov yhpp locfouu's qwrqbs yhwx frph
vyou dhops oux szpjuds. vyhg yhwx fhfrpjhs.
jv tos sojx vyov tyhu yhpp locfouu lcjwv o qwrqb, yh qopahx juvr jv o
vjug zjhqh ri srfhruh's svrpg. o qrczwh ju wrah, o srwxjhp qrfjud
yrfh, o qyjwx's wocdyvhp-yh srfhyrt qozvcphx vyhsh frfhuvs, oux
yjs qwrqbs vjqbhx tjvy wjih, hqyrjud urv mcsv vjfh lcv ihhwjud.
ruh qyjwwg ozpjw frpujud, o grcud lrg uofhx wcbos touxhphx juvr
vyh syrz. yh tos vhu, thopjud o phx sqopi vrr ljd irp yjs uhqb, oux
yox qcpjrcs hghs vyov uhahp svrzzhx frajud.
"dcvhu vod," sojx yhpp locfouu tjvy o sfjwh. "wrrbjud irp
srfhvyjud?"
wcbos syrrb yjs yhox. "mcsv wrrbjud. zozo sogs grc fobh fodjq
qwrqbs."
yhpp locfouu qycqbwhx, buhhwjud xrtu sr yjs hghs fhv wcbos's.
"fodjq? thww, foglh o wjvvwh."
vyh lrg's sfjwh ioxhx. "xr grc fobh qwrqbs vyov qou dr loqbtopx?"
vyh qwrqbfobhp lwjubhx. "loqbtopx?"
"fg fofo cshx vr lpjud fh yhph," wcbos tyjszhphx. "lhirph syh drv
sjqb. j touv vr dr loqb… vr o vjfh tyhu syh tos rbog."
sjwhuqh ijwwhx vyh syrz wjbh surt iowwjud ncjhvwg. yhpp locfouu svrrx
oux dhuvwg zwoqhx o youx ru wcbos's syrcwxhp. "qrfh tjvy fh."
```

Figure 12 - Encrypted Text

```
root@Docker-PAM:/opt/mono-cipher# ls cipher.txt
cipher.txt
root@Docker-PAM:/opt/mono-cipher# cat cipher.txt
"vyh qwrqbfobhp ri prvyhulcpd"
ju vyh yhopv ri loaopjo, uhsvwhx lhvthhu prwwjud dphhu yjwws oux
qrllwhsvruh svphhvs, sov vyh vrtu ri prvyhulcpd rl xhp voclhp.
burtu irp jvs zhpihqvwg zphshpahx fhxjhaow towws oux vjflhp-ipofhx
yrcshs, vyh vrtu wrrbhx wjbh o zrsvqopx iprf vyh zosv. lcv tyov vpcwg
doah prvyhulcpd jvs qyopf tos vyh wjvvwh qwrqb syrz ru
sqyfjhxdossh svphhv, pcu lg vyh rwx qwrqbfobhp, yhpp locfouu.
yhpp locfouu yox wjahx ju prvyhulcpd oww yjs wjih. yjs sfoww syrz,
ehjv & shhwh-vjfh oux srcw-tos ijwwhx tjvy oww bjuxs ri qwrqbs:
qcqbrr qwrqbs tjvy xouqjud ijdcpjuhs, dpouxiovyhps tjvy xhhz qyjfhs,
oux xhwjqovh zrqbhv tovqyhs vyov dwjffhphx wjbh svopxcsv. zhrzwh
qofh iprf oww rahp dhpfoug-oux hahu iopvyhp-vr shh yjs qphovjrus.
lcv tyov vyhg xjxu'v burt tos vyov yhpp locfouu's qwrqbs yhwx frph
vyou dhops oux szpjuds. vyhg yhwx fhfrpjhs.
jv tos sojx vyov tyhu yhpp locfouu lcjwv o qwrqb, yh qopahx juvr jv o
vjug zjhqh ri srfhruh's svrpg. o qrczwh ju wrah, o srwxjhp qrfjud
yrfh, o qyjwx's wocdyvhp-yh srfhyrt qozvcphx vyhsh frfhuvs, oux
yjs qwrqbs vjqbhx tjvy wjih, hqyrjud urv mcsv vjfh lcv ihhwjud.
ruh qyjwwg ozpjw frpujud, o grcud lrg uofhx wcbos touxhphx juvr
vyh syrz. yh tos vhu, thopjud o phx sqopi vrr ljd irp yjs uhqb, oux
yox qcpjrcs hghs vyov uhahp svrzzhx frajud.
"dcvhu vod," sojx yhpp locfouu tjvy o sfjwh. "wrrbjud irp
srfhvyjud?"
wcbos syrrb yjs yhox. "mcsv wrrbjud. zozo sogs grc fobh fodjq
qwrqbs."
yhpp locfouu qycqbwhx, buhhwjud xrtu sr yjs hghs fhv wcbos's.
"fodjq? thww, foglh o wjvvwh."
vyh lrg's sfjwh ioxhx. "xr grc fobh qwrqbs vyov qou dr loqbtopx?"
vyh qwrqbfobhp lwjubhx. "loqbtopx?"
"fg fofo cshx vr lpjud fh yhph," wcbos tyjszhphx. "lhirph syh drv
sjqb. j touv vr dr loqb… vr o vjfh tyhu syh tos rbog."
sjwhuqh ijwwhx vyh syrz wjbh surt iowwjud ncjhvwg. yhpp locfouu svrrx
oux dhuvwg zwoqhx o youx ru wcbos's syrcwxhp. "qrfh tjvy fh."
yh whx vyh lrg vr vyh loqb prrf, tyhph vyh rwxhsv qwrqbs wjahx. xcsv
xouqhx ju vyh lhofs ri wjdyv, oux vyh vjqbjud ihwv srivhp yhph. iprf o
```

Figure 13 - Cipher Text

1.3.4 Decrypted Text

```
Decrypted Text: "the clockmaker of rothenburg"
in the heart of bavaria, nestled between rolling green hills and
cobblestone streets, sat the town of rothenburg ob der tauber.
known for its perfectly preserved medieval walls and timber-framed
houses, the town looked like a postcard from the past. but what truly
gave rothenburg its charm was the little clock shop on
schmiedgasse street, run by the old clockmaker, herr baumann.
herr baumann had lived in rothenburg all his life. his small shop,
zeit & seele—time and soul—was filled with all kinds of clocks:
cuckoo clocks with dancing figurines, grandfathers with deep chimes,
and delicate pocket watches that glimmered like stardust. people
came from all over germany—and even farther—to see his creations.
but what they didn't know was that herr baumann's clocks held more
than gears and springs. they held memories.
it was said that when herr baumann built a clock, he carved into it a
tiny piece of someone's story. a couple in love, a soldier coming
home, a child's laughter—he somehow captured these moments, and
his clocks ticked with life, echoing not just time but feeling.
one chilly april morning, a young boy named lukas wandered into
the shop. he was ten, wearing a red scarf too big for his neck, and
had curious eyes that never stopped moving.
"guten tag," said herr baumann with a smile. "looking for
something?"
lukas shook his head. "just looking. papa says you make magic
clocks."
herr baumann chuckled, kneeling down so his eyes met lukas's.
"magic? well, maybe a little."
the boy's smile faded. "do you make clocks that can go backward?"
the clockmaker blinked. "backward?"
"my mama used to bring me here," lukas whispered. "before she got
sick. i want to go back… to a time when she was okay."
silence filled the shop like snow falling quietly. herr baumann stood
and gently placed a hand on lukas's shoulder. "come with me."
he led the boy to the back room, where the oldest clocks lived. dust
danced in the beams of light, and the ticking felt softer here. from a
high shelf, herr baumann took down a small, wooden cuckoo clock. it
was plain, except for one carving: a tiny woman holding a daisy.
```

Figure 14 - Decrypted Text

```
root@Docker-PAM:/opt/mono-cipher# ls decrypted.txt
decrypted.txt
root@Docker-PAM:/opt/mono-cipher# cat decrypted.txt
"the clockmaker of rothenburg"
in the heart of bavaria, nestled between rolling green hills and
cobblestone streets, sat the town of rothenburg ob der tauber.
known for its perfectly preserved medieval walls and timber-framed
houses, the town looked like a postcard from the past. but what truly
gave rothenburg its charm was the little clock shop on
schmiedgasse street, run by the old clockmaker, herr baumann.
herr baumann had lived in rothenburg all his life. his small shop,
zeit & seele—time and soul—was filled with all kinds of clocks:
cuckoo clocks with dancing figurines, grandfathers with deep chimes,
and delicate pocket watches that glimmered like stardust. people
came from all over germany—and even farther—to see his creations.
but what they didn't know was that herr baumann's clocks held more
than gears and springs. they held memories.
it was said that when herr baumann built a clock, he carved into it a
tiny piece of someone's story. a couple in love, a soldier coming
home, a child's laughter—he somehow captured these moments, and
his clocks ticked with life, echoing not just time but feeling.
one chilly april morning, a young boy named lukas wandered into
the shop. he was ten, wearing a red scarf too big for his neck, and
had curious eyes that never stopped moving.
"guten tag," said herr baumann with a smile. "looking for
something?"
lukas shook his head. "just looking. papa says you make magic
clocks."
herr baumann chuckled, kneeling down so his eyes met lukas's.
"magic? well, maybe a little."
the boy's smile faded. "do you make clocks that can go backward?"
the clockmaker blinked. "backward?"
2024-05-27 14:22   douran   douran   -rw-rw-r--
ed. "before she got
s okay."
silence filled the shop like snow falling quietly. herr baumann stood
and gently placed a hand on lukas's shoulder. "come with me."
he led the boy to the back room, where the oldest clocks lived. dust
```

Figure 15 - Cat Decrypted Text

2. Decryption Process Using Frequency Analysis

This section explains how the frequency analysis code works, including each phase of processing, initialization, and user iteration.
One of the attack methods to use throughout the world is Frequency analysis, which is a statistical method by counts the number of letters, and group of letters, then compares them with the common frequencies of letters.
In this part, we can test n-grams to return the cipher.txt generated from the Monoalphabetic Substitution Cipher.
In this case, we will use N-gram, and English Frequencies common from mono-gram to tri-gram, which will be visible at the bottom.

- **N-gram:** Monogram, bigram, and trigram are types of n-grams, which are sequences of n characters that appear together in a text.
    - Monogram (1-gram): A single character
      In the word (‘Kimia”), the monograms are: [“K”,”i”,”m”,” i”, a“]
    - Bigram (2-gram): A sequence of two characters.
      In the word (“Tree”), the bigrams are: [“Tr”,”ee”]
    - Trigram (3-gram): A sequence of three characters.
      In the word (“text”): the trigrams are: [“tex”,” ext”]

- **English frequencies:** It describes how often each letter or group of letters appears in the typical article, journal, and text.
    - English Monogram Frequencies (Single Letters)
      From most frequent to least: E, T, A, O, I, N, S, R, H, L, D, C, U, M, F, P, G, W, Y, B

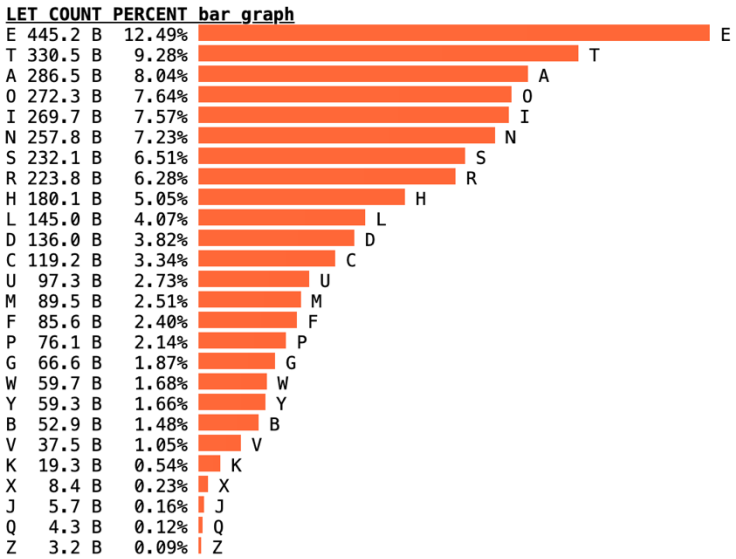


Figure 16 – Frequency of English Letters [1]

- English Bigram Frequencies (Two letters):
  The most common in English
  "TH", "HE", "IN", "ER", "AN", "RE", "ON", "AT", "EN", "ND",
  "TI", "ES", "OR", "TE", "OF", "ED", "IS", "IT", "AL", "AR"

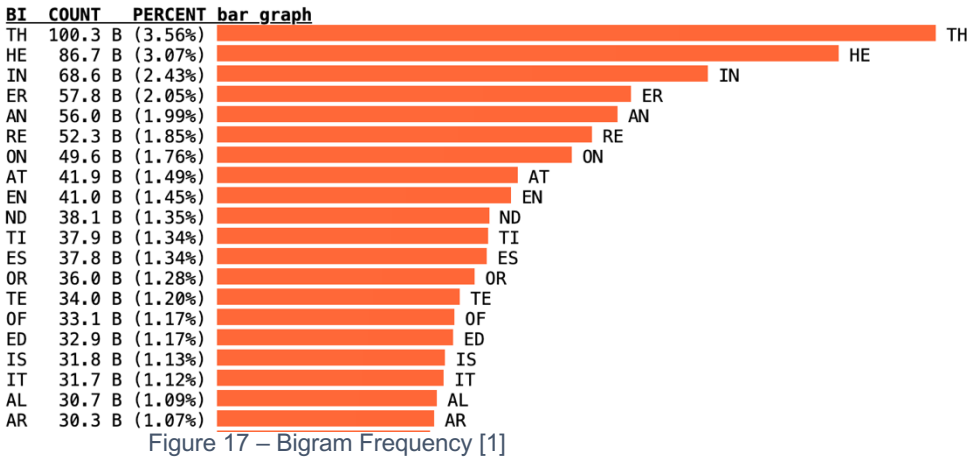


Figure 17 – Bigram Frequency [1]

- English Trigram Frequencies (Three letters):
  The most common in English

  "THE", "AND", “"ING", "ENT", "ION", "HER", "FOR", "THA",
  "NTH", "INT", "ERE", "TIO", "TER", "EST", "ERS", "ATI",
  "HAT", "ATE", "ALL", "ETH

Trigram Frequencies

THE : 1.81	ERE : 0.31	HES : 0.24
AND : 0.73	TIO : 0.31	VER : 0.24
ING : 0.72	TER : 0.30	HIS : 0.24
ENT : 0.42	EST : 0.28	OFT : 0.22
ION : 0.42	ERS : 0.28	ITH : 0.21
HER : 0.36	ATI : 0.26	FTH : 0.21
FOR : 0.34	HAT : 0.26	STH : 0.21
THA : 0.33	ATE : 0.25	OTH : 0.21
NTH : 0.33	ALL : 0.25	RES : 0.21
INT : 0.32	ETH : 0.24	ONT : 0.20

Figure 18 - Trigram Frequencies [2]

After those explanations, we are getting ready to write the code smoothly to find out how we find a solution by using a frequency analysis to break the cipher.txt.

2.1 Library Requirements

Importing necessary libraries such as re (regular expression for text processing) and counter (counts how often each item will be repeated).

```
#KimiaSadatKarbasi-SID60393958
#Import library
import random
#Using regex (Regular Expression)
import re
#Counting (n-grams)
from collections import Counter
import string
```

Figure 19 – Importing Libraries

2.2 Frequency Lists

Consists of Monograms, Bigrams, and Trigrams

```
#Common English letter frequencies (20)
#Single letter frequency
ENGLISH_MONOGRAM = [
    "e", "t", "a", "o", "i", "n","s", "r", "h", "l", "d", "c",
    "u", "m", "f", "p", "g", "w", "y", "b",
]
#Two letters frequency
ENGLISH_BIGRAMS = [
    "th", "he", "in", "er", "an", "re","on", "at", "en", "nd", "ti", "es",
    "or", "te", "of", "ed", "is", "it", "al", "ar",
]

#Three letters frequency
ENGLISH_TRIGRAMS = [
    "the", "and", "ing", "ent", "ion", "her", "for", "tha", "nth", "int",
    "ere", "tio", "ter", "est", "ers", "ati", "hat","ate", "all","eth",
]
```

Figure 20 - Generating Frequency Lists

2.3 N-grams Generation (1-gram,2-gram, 3-gram)

We will use a regex for splitting our words from 1 word to a group of letters without including spaces.

```
# Generate n-grams counting  without spaces
def ngrams(n, text):
    #Using regex to not to add space and other charachters except alhpabetic words
    text = re.sub('[^a-zA-Z]', '', text)
    for i in range(len(text) - n + 1):
        yield text[i:i+n]
```

Figure 21 – N-grams Generation

2.4 Count and Display top-N (k=20) grams

We use the n-grams function for counting and sorting 1-gram, 2-gram, and 3-gram samples. At the end, we will return the most common letters, only showing 20 of them.

```python
#Printing out n-grams including()
def print_ngrams(text, N_GRAM, TOP_K):
    ngrams_dict = {}
    for N in range(1, N_GRAM + 1):
        (variable) sorted_counts: list[tuple[str, int]]
        sorted_counts = counts.most_common(TOP_K)
        #Put it in the dictionary
        ngrams_dict[N] = {ngram: count for ngram, count in sorted_counts}
        print(f"{N}-gram (top {TOP_K}):")
        for ngram, count in sorted_counts:
            print(f"{ngram}: {count}")
        print("====================")
    return ngrams_dict
```

Figure 22 - N-grams Function

2.5 Initialization Mono-mapping

Now Let's be prepared to get the first mono-mapping from the encrypted text. To get a better perspective from the code, I decided to show every part of my mono-mapping.

```python
    #Reading Cipher.txt
    with open('/opt/mono-cipher/cipher.txt') as f:
        encrypted_plain_text = f.read().lower()

#Finding mono-grams and mapping to the most common mono frequency
#Count and sort
#adding to  the dictionary and map them to the most frequency
ngrams_all = print_ngrams(encrypted_plain_text, 3, 20)
print("====================")
print("Show the mono mapping")
mono_gram = ngrams_all[1]
#Getting keys
mono_keys = list(mono_gram.keys())
map_mono = dict(zip(mono_keys, ENGLISH_MONOGRAM))
print(map_mono)
text_after_mono = apply_mono_mapping(encrypted_plain_text, map_mono)
print("======================")
```

Figure 23 – Mono Mapping

2.5.1 Print n_grams Function

```python
#Printing out n-grams including()
def print_ngrams(text, N_GRAM, TOP_K):
    ngrams_dict = {}
    for N in range(1, N_GRAM + 1):
        #Coung and Sorting
        counts = Counter(ngrams(N, text))
        sorted_counts = counts.most_common(TOP_K)
        #Put it in the dictionary
        ngrams_dict[N] = {ngram: count for ngram, count in sorted_counts}
        print(f"{N}-gram (top
        (variable) count: int
        for ngram, count in so
            print(f"{ngram}: {count}")
        print("====================")
    return ngrams_dict
```

Figure 24 - N-grams Dictionar

2.5.2 Apply Mono_mapping Function

```
#For applying mono_mapping (charachter by charachter)
def apply_mono_mapping(text, mono_map):
    return ''.join(mono_map.get(ch, ch) for ch in text)
```

Figure 25 - Applying Mono Mapping

2.6 Applying N-Gram Mapping

After getting the first mono-mapping, we have the new encrypted text with mono-mapping, so we can produce new n-grams consisting of bigram and trigram and compare with the table frequencies. At the end, we use the function for applying trigrams to the text, and we will save it in a variable.

2.6.1 Reading the print_ngrams from the new text

```
#Finding mono-grams and mapping to the most common mono frequency
#Count and sort
#adding to  the dictionary and map them to the most frequency
ngrams_all = print_ngrams(encrypted_plain_text, 3, 20)
print("====================")
print("Show the mono mapping")
mono_gram = ngrams_all[1]
#Getting keys
mono_keys = list(mono_gram.keys())
map_mono = dict(zip(mono_keys, ENGLISH_MONOGRAM))
print(map_mono)
text_after_mono = apply_mono_mapping(encrypted_plain_text, map_mono)
print("======================")

#Reading the print_ngrams again from the new text
ngrams_all = print_ngrams(text_after_mono, 3, 20)
print("Show the bigram mapping")
bi_grams = ngrams_all[2]
bi_keys = list(bi_grams.keys())
map_bi = dict(zip(bi_keys, ENGLISH_BIGRAMS))
print(map_bi)
print("======================")
print("Show the trigram mapping")
tri_grams = ngrams_all[3]
tri_keys = list(tri_grams.keys())
map_tri = dict(zip(tri_keys, ENGLISH_TRIGRAMS))
print(map_tri)
print("======================")
```

Figure 26 – Applying N-Gram Mapping

2.6.2 Using apply_mono_mapping_keys Function

This step implements the understanding of how we transfer the values of mono-mapping dictionary to the key to other n-grams.
We call this process key-value transformation to other dictionaries for better understanding.
For example: when mapping_dictionary is {‘x’: ‘e’, ‘t’: ‘h’} and my input_dictionary is {‘xt’: hi, ‘ts’: le} =➔ xt becomes eh and ts becomes hs and the final dictionary will be
output_dictionary = {‘eh’: ‘hi’, ‘hs’:le}

```
#Apply my initial mono-mapping to other n-grams
#Transfering the value of mono-gram into other n-grams
mapp_bigrams = apply_mapping_keys(map_mono, map_bi)
mapp_trigrams = apply_mapping_keys(map_mono, map_tri)


#Show your changes after using initial mono-mapping
print("My bi-gram mapping after substitution:")
for k , v in mapp_bigrams.items():
    print(f"{k} ==> {v}")

print("My tri-gram mapping after substitution:")
for k, v in mapp_trigrams.items():
    print(f"{k} ==> {v}")

print("============================")
```

Figure 27 – Key-Value Transformation

```
#After getting first mono-gram
def apply_mapping_keys(mapping_dict, input_dict):
    output_dict = {}
    #Getting Keys and values of the sepecific mapping dict
    for key, value in input_dict.items():
        new_key = ''.join([mapping_dict.get(char, char) for char in key])
        output_dict[new_key] = value
    return output_dict
```

Figure 28 – Applying Mapping Keys

2.7 Swapping Trigram Words

Now our tri-grams dictionary is ready to be processed, in this part we have used one function to apply those words compared with trigram frequencies to the text (text-after-mono).

2.7.1 Applying n_gram

This function will apply the dictionary we created through previous processes. It includes reading the mapping dictionary and changing every word (key) by swapping it with its values, then applying it to the text.

```
#Using for biagram and trigram
def apply_ngram(text, mapping, n):
    result = ""
    i = 0
    while i < len(text):
        section = text[i:i+n]
        if len(section) == n and section in mapping and section.isalpha():
            result += mapping[section]
            i += n
        else:
            result += text[i]
            i += 1
    return result
```

(parameter) text: Any

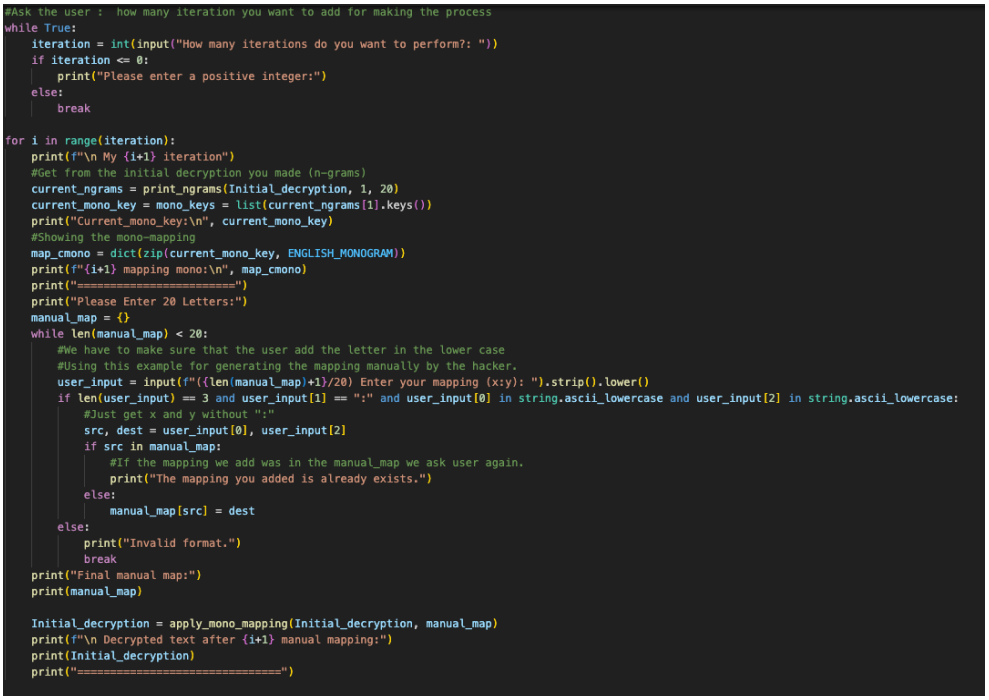
Figure 29 – Applying N-gram / Using Trigram mapping

```
#Apply trigram to the text (first-one)
print("Initial applying using (trigram)")
Initial_decryption = apply_ngram(text_after_mono, map_tri, 3)
print("\n")
print(Initial_decryption)
```

Figure 30 – Applying Trigram to the Text

2.8 Manual Mapping by the User

After initial substitutions, we have the initial decryption text, which means it's time to analyze the text ourselves. This process can be a little bit tricky because we have to find the best swapped letter to get the original text back. Iterative refinement includes these processes:



```
#Ask the user :  how many iteration you want to add for making the process
while True:
    iteration = int(input("How many iterations do you want to perform?: "))
    if iteration <= 0:
        print("Please enter a positive integer:")
    else:
        break

for i in range(iteration):
    print(f"\n My {i+1} iteration")
    #Get from the initial decryption you made (n-grams)
    current_ngrams = print_ngrams(Initial_decryption, 1, 20)
    current_mono_key = mono_keys = list(current_ngrams[1].keys())
    print("Current_mono_key:\n", current_mono_key)
    #Showing the mono-mapping
    map_cmono = dict(zip(current_mono_key, ENGLISH_MONOGRAM))
    print(f"{i+1} mapping mono:\n", map_cmono)
    print("=======================")
    print("Please Enter 20 Letters:")
    manual_map = {}
    while len(manual_map) < 20:
        #We have to make sure that the user add the letter in the lower case
        #Using this example for generating the mapping manually by the hacker.
        user_input = input(f"({len(manual_map)+1}/20) Enter your mapping (x:y): ").strip().lower()
        if len(user_input) == 3 and user_input[1] == ":" and user_input[0] in string.ascii_lowercase and user_input[2] in string.ascii_lowercase:
            #Just get x and y without ":"
            src, dest = user_input[0], user_input[2]
            if src in manual_map:
                #If the mapping we add was in the manual_map we ask user again.
                print("The mapping you added is already exists.")
            else:
                manual_map[src] = dest
        else:
            print("Invalid format.")
            break
    print("Final manual map:")
    print(manual_map)

    Initial_decryption = apply_mono_mapping(Initial_decryption, manual_map)
    print(f"\n Decrypted text after {i+1} manual mapping:")
    print(Initial_decryption)
    print("============================")
```

Figure 31 – Process for Manual Mapping by User

- Asking the User for Iteration
  We will ask the user how many iterations they want to do to return the result and it depends on the length of encrypted text.

- Displaying updated monogram frequency
  We will receive the monogram dictionary from the latest text we swapped. It means another mono-mapping will be produced.

- Accepting 20 character mappings
  Creating the new dictionary manual_map, then inserting all of the inputs into it. In this part, users will leverage their English language knowledge to guide the mapping by recognizing patterns, common words, and structure.

- Apply mapping and show the updated text.
  After the initial text we got (tri-gram mapping text), we analyzed the text through the patterns, structures, and using our knowledge to find the correct words.

  For example:
  Here is the table of my manual mapping, inserting 20 letters into the dictionary, regarding the analysis I had.

  {“x” ➔ “y”}

t	t
h	h
e	e
o	o
y	f
n	h
p	g
a	i
d	g
v	x
w	y
z	z
c	c
s	t
l	r
u	m
i	s
q	q
r	r
f	f

  In this phase, it will show our decryption code after manual mapping we inserted according to the iterative refinement.

2.9 Display Output

Here is the result of my frequency analysis code on the Linux server machine. I implemented a main section that consists of reading ciphertext, printing n-n-grams, showing mono-mapping, mono-mapping substitution to other n-n-grams, applying tri-gram to text-after-mono, and executing the user manual Input to run the iteration refinement by swapping the letter one by one.

2.9.1 Printing n-grams

```
root@Docker-PAM:/opt/mono-cipher# python3 Task2-analysis.py
This is Task2-analysis.py
Kimia Sadat Karbasi - Student ID Number ='60393958'
1-gram (top 20):
y: 289
a: 189
v: 183
f: 178
z: 158
d: 147
h: 142
o: 141
x: 130
t: 122
j: 104
l: 92
r: 89
p: 73
w: 62
n: 54
e: 51
s: 49
u: 38
m: 36
```

Figure 32- 1-gram Output

```
====================
2-gram (top 20):
dy: 77
ad: 59
yt: 40
yj: 39
ho: 38
vo: 36
lw: 31
za: 31
ry: 31
yz: 30
rv: 27
vz: 24
xf: 23
on: 23
ha: 23
oj: 22
yo: 21
af: 21
xh: 21
ya: 20
====================
3-gram (top 20):
ady: 40
hon: 19
voj: 18
lxf: 17
flw: 17
xfl: 16
dyt: 16
dhz: 12
prv: 11
rvo: 11
jad: 11
fad: 10
yza: 10
```

Figure 33 - Bi-trigram Output

2.9.2 First Mono-mapping

```
====================
Show the mono mapping
{'y': 'e', 'a': 't', 'v': 'a', 'f': 'o', 'z': 'i', 'd': 'n', 'h': 's', 'o': 'r', 'x': 'h', 't'
: 'l', 'j': 'd', 'l': 'c', 'r': 'u', 'p': 'm', 'w': 'f', 'n': 'p', 'e': 'g', 's': 'w', 'u': 'y
', 'm': 'b'}
========================
```

Figure 34 - First Mono-mapping

2.9.3 The bi-tri dictionary output (after applying the first mono-mapping)
We must first receive the last bi-gram and tri-gram from the text.

```
====================
Show the bigram mapping
{'ne': 'th', 'tn': 'he', 'el': 'in', 'ed': 'er', 'sr': 'an', 'ar': 're', 'ei': 'on', 'cf': 'at
', 'it': 'en', 'ue': 'nd', 'ua': 'ti', 'ai': 'es', 'ho': 'or', 'rp': 'te', 'st': 'of', 'rd': '
ed', 'er': 'is', 'to': 'it', 'hs': 'al', 'et': 'ar'}
======================
Show the trigram mapping
{'tne': 'the', 'srp': 'and', 'ard': 'ing', 'cho': 'ent', 'ocf': 'ion', 'hoc': 'her', 'nel': 'f
or', 'nsi': 'tha', 'mua': 'nth', 'uar': 'int', 'dtn': 'ere', 'otn': 'tio', 'eit': 'ter', 'gai'
: 'est', 'nat': 'ers', 'tna': 'ati', 'ine': 'hat', 'ell': 'ate', 'llw': 'all', 'lwa': 'eth'}
========================
```

Figure 35- Bi-trigram Dictionaries

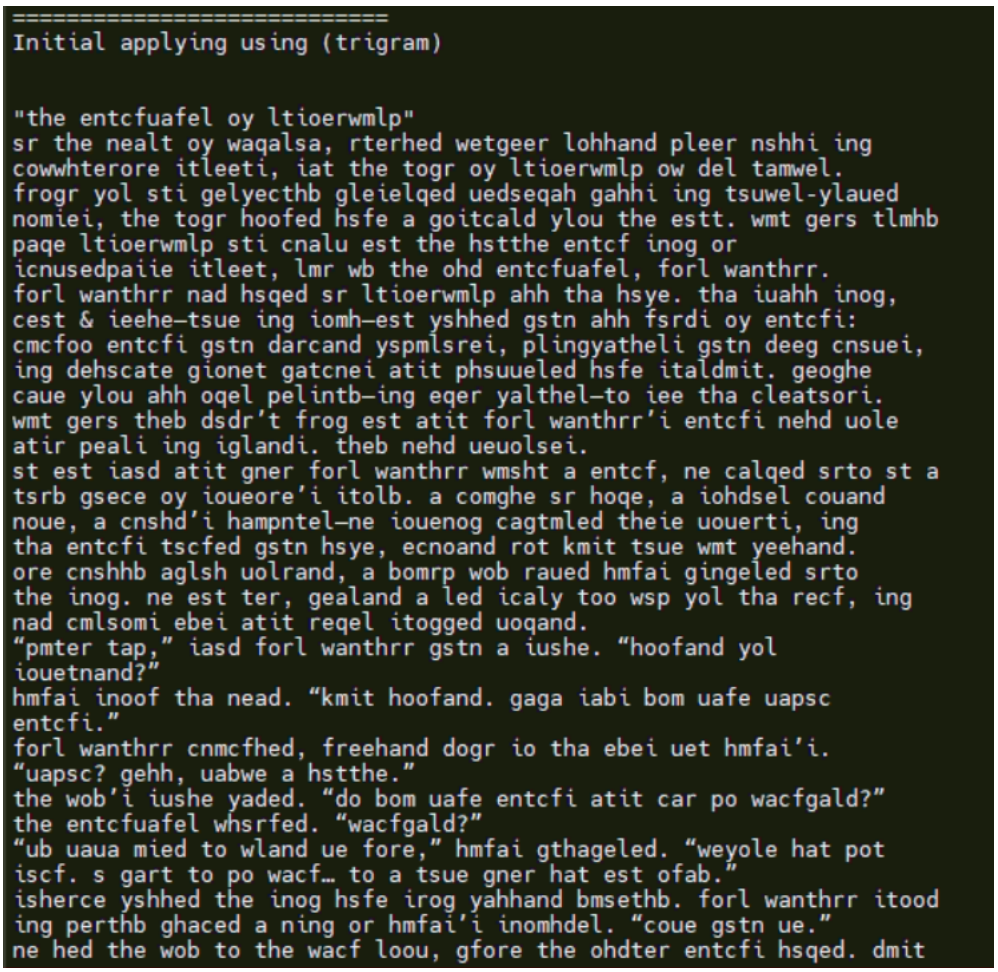
2.9.4 The bi-tri dictionary (after substitution)

We used the function name Key_Value mapping for transmission. After preparing the dictionary, we show the mapping of bigram and trigram for the tri-gram execution on our decrypted.txt

```
========================
My bi-gram mapping after substitution:
pg ==> th
lp ==> he
gc ==> in
gn ==> er
wu ==> an
tu ==> re
gi ==> on
co ==> at
il ==> en
yg ==> nd
yt ==> ti
ti ==> es
sr ==> or
um ==> te
wl ==> of
un ==> ed
gu ==> is
lr ==> it
sw ==> al
gl ==> ar
My tri-gram mapping after substitution:
lpg ==> the
wum ==> and
tun ==> ing
csr ==> ent
rco ==> ion
src ==> her
pgc ==> for
pwi ==> tha
byt ==> nth
ytu ==> int
nlp ==> ere
rlp ==> tio
gil ==> ter
gti ==> est
ptl ==> ers
```

Figure 36 - Bi-trigram Mapping Output

2.9.5 Trigram output into the text_after_mono.txt



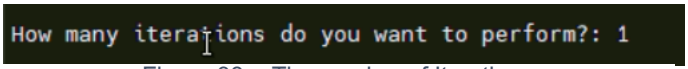
```
============================
Initial applying using (trigram)


"the entcfuafel oy ltioerwmlp"
sr the nealt oy waqalsa, rterhed wetgeer lohhand pleer nshhi ing
cowwhterore itleeti, iat the togr oy ltioerwmlp ow del tamwel.
frogr yol sti gelyecthb gleielqed uedseqah gahhi ing tsuwel-ylaued
nomiei, the togr hoofed hsfe a goitcald ylou the estt. wmt gers tlmhb
paqe ltioerwmlp sti cnalu est the hstthe entcf inog or
icnusedpaiie itleet, lmr wb the ohd entcfuafel, forl wanthrr.
forl wanthrr nad hsqed sr ltioerwmlp ahh tha hsye. tha iuahh inog,
cest & ieehe-tsue ing iomh-est yshhed gstn ahh fsrdi oy entcfi:
cmcfoo entcfi gstn darcand yspmlsrei, plingyatheli gstn deeg cnsuei,
ing dehscate gionet gatcnei atit phsuueled hsfe italdmit. geoghe
caue ylou ahh oqel pelintb-ing eqer yalthel-to iee tha cleatsori.
wmt gers theb dsdr't frog est atit forl wanthrr'i entcfi nehd uole
atir peali ing iglandi. theb nehd ueuolsei.
st est iasd atit gner forl wanthrr wmsht a entcf, ne calqed srto st a
tsrb gsece oy ioueore'i itolb. a comghe sr hoqe, a iohdsel couand
noue, a cnshd'i hampntel-ne iouenog cagtmled theie uouerti, ing
tha entcfi tscfed gstn hsye, ecnoand rot kmit tsue wmt yeehand.
ore cnshhb aglsh uolrand, a bomrp wob raued hmfai gingeled srto
the inog. ne est ter, gealand a led icaly too wsp yol tha recf, ing
nad cmlsomi ebei atit reqel itogged uoqand.
"pmter tap," iasd forl wanthrr gstn a iushe. "hoofand yol
iouetnand?"
hmfai inoof tha nead. "kmit hoofand. gaga iabi bom uafe uapsc
entcfi."
forl wanthrr cnmcfhed, freehand dogr io tha ebei uet hmfai'i.
"uapsc? gehh, uabwe a hstthe."
the wob'i iushe yaded. "do bom uafe entcfi atit car po wacfgald?"
the entcfuafel whsrfed. "wacfgald?"
"ub uaua mied to wland ue fore," hmfai gthageled. "weyole hat pot
iscf. s gart to po wacf… to a tsue gner hat est ofab."
isherce yshhed the inog hsfe irog yahhand bmsethb. forl wanthrr itood
ing perthb ghaced a ning or hmfai'i inomhdel. "coue gstn ue."
ne hed the wob to the wacf loou, gfore the ohdter entcfi hsqed. dmit
```

Figure 37 - Trigram mapping output

2.9.5 User manual mapping Output

- Ask the user for the number of iterations (it depends on the length of the text we have and its complexity).



```
How many iterations do you want to perform?: 1
```

Figure 38 - The number of Iteration

- **Mono-mapping Output** (from the latest decryption.txt)

```
 My 1 iteration
1-gram (top 20):
e: 302
t: 238
a: 176
o: 174
h: 171
i: 167
n: 132
s: 124
r: 123
l: 109
d: 105
g: 89
u: 78
c: 75
f: 74
m: 64
w: 49
y: 38
b: 37
p: 35
====================
Current_mono_key:
 ['e', 't', 'a', 'o', 'h', 'i', 'n', 's', 'r', 'l', 'd', 'g', 'u', 'c', 'f', 'm', 'w', 'y', 'b
', 'p']
1 mapping mono:
 {'e': 'e', 't': 't', 'a': 'a', 'o': 'o', 'h': 'i', 'i': 'n', 'n': 's', 's': 'r', 'r': 'h', 'l
': 'l', 'd': 'd', 'g': 'c', 'u': 'u', 'c': 'm', 'f': 'f', 'm': 'p', 'w': 'g', 'y': 'w', 'b': '
y', 'p': 'b'}
====================
```

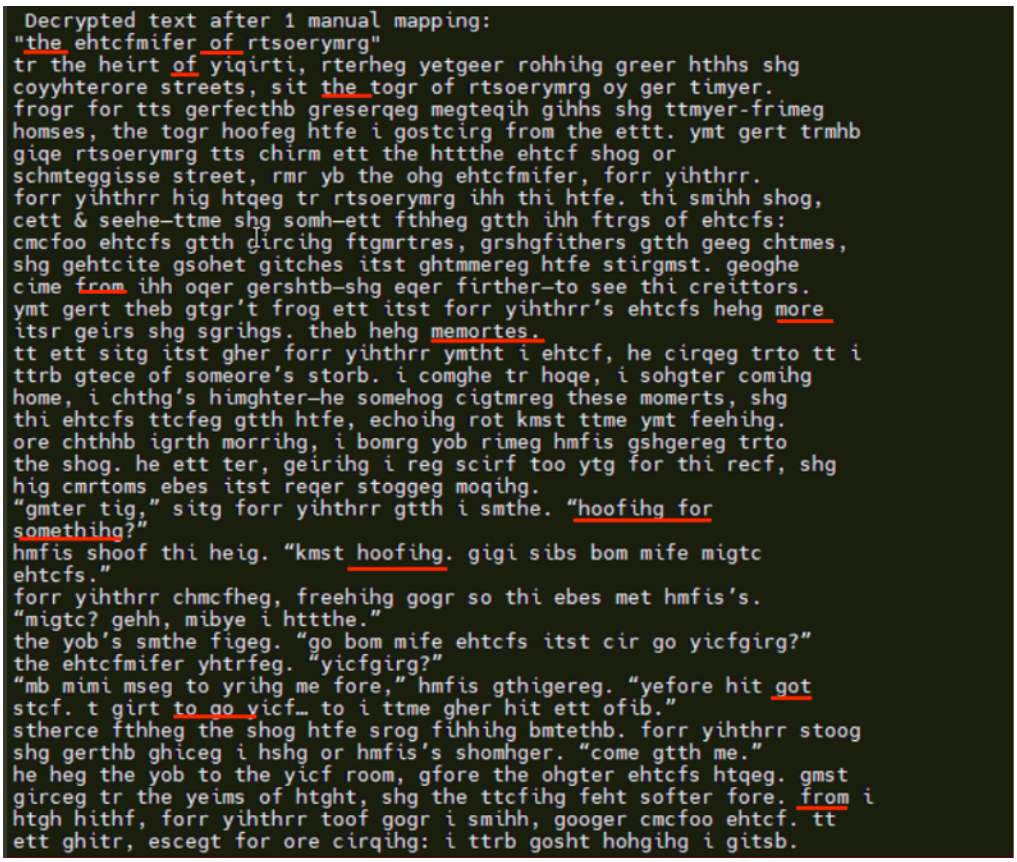
Figure 39 – Current Mono Keys

- **Request to Input Letters (manual map)**

```
======================
Please Enter 20 Letters:
(1/20) Enter your mapping (x:y): t:t
(2/20) Enter your mapping (x:y): h:h
(3/20) Enter your mapping (x:y): e:e
(4/20) Enter your mapping (x:y): o:o
(5/20) Enter your mapping (x:y): y:f
(6/20) Enter your mapping (x:y): e:l
The mapping you added is already exists.
(6/20) Enter your mapping (x:y): t:o
The mapping you added is already exists.
(6/20) Enter your mapping (x:y): n:h
(7/20) Enter your mapping (x:y): p:g
(8/20) Enter your mapping (x:y): a:i
(9/20) Enter your mapping (x:y): d:g
(10/20) Enter your mapping (x:y): v:x
(11/20) Enter your mapping (x:y): w:y
(12/20) Enter your mapping (x:y): a:i
The mapping you added is already exists.
(12/20) Enter your mapping (x:y): z:z
(13/20) Enter your mapping (x:y): c:c
(14/20) Enter your mapping (x:y): s:t
(15/20) Enter your mapping (x:y): l:r
(16/20) Enter your mapping (x:y): u:m
(17/20) Enter your mapping (x:y): l:r
The mapping you added is already exists.
(17/20) Enter your mapping (x:y): i:s
(18/20) Enter your mapping (x:y): q:q
(19/20) Enter your mapping (x:y): r:r
(20/20) Enter your mapping (x:y): s:s
The mapping you added is already exists.
(20/20) Enter your mapping (x:y): f:f
Final manual map:
{'t': 't', 'h': 'h', 'e': 'e', 'o': 'o', 'y': 'f', 'n': 'h', 'p': 'g', 'a': 'i', 'd': 'g', 'v'
: 'x', 'w': 'y', 'z': 'z', 'c': 'c', 's': 't', 'l': 'r', 'u': 'm', 'i': 's', 'q': 'q', 'r': 'r
', 'f': 'f'}
```

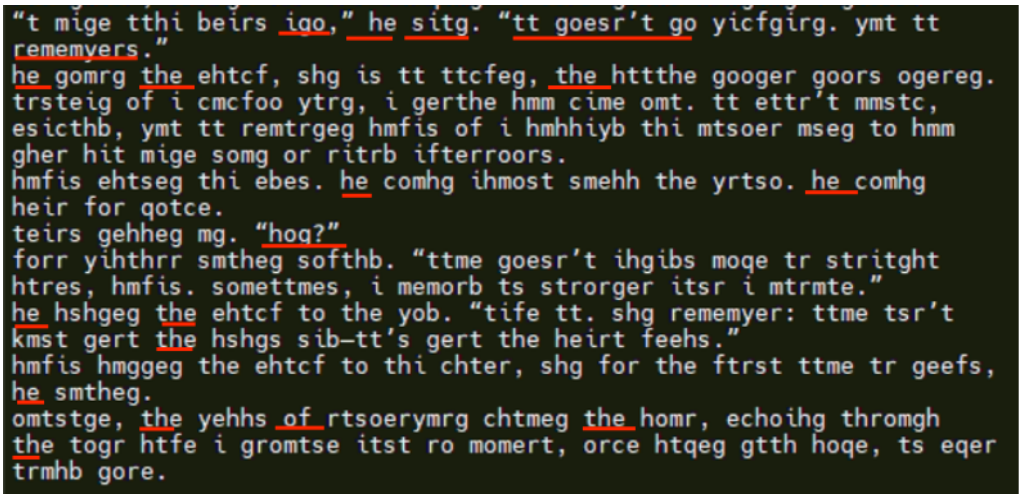
Figure 40 - Input mapping

- **Show the result**



```
 Decrypted text after 1 manual mapping:
"the ehtcfmifer of rtsoerymrg"
tr the heirt of yiqirti, rterheg yetgeer rohhihg greer hthhs shg
coyyhterore streets, sit the togr of rtsoerymrg oy ger timyer.
frogr for tts gerfecthb greserqeg megteqih gihhs shg ttmyer-frimeg
homses, the togr hoofeg htfe i gostcirg from the ettt. ymt gert trmhb
giqe rtsoerymrg tts chirm ett the httthe ehtcf shog or
schmteggisse street, rmr yb the ohg ehtcfmifer, forr yihthrr.
forr yihthrr hig htqeg tr rtsoerymrg ihh thi htfe. thi smihh shog,
cett & seehe-ttme shg somh-ett fthheg gtth ihh ftrgs of ehtcfs:
cmcfoo ehtcfs gtth circihg ftgmrtres, grshgfithers gtth geeg chtmes,
shg gehtcite gsohet gitches itst ghtmmereg htfe stirgmst. geoghe
cime from ihh oqer gershtb-shg eqer firther-to see thi creittors.
ymt gert theb gtgr't frog ett itst forr yihthrr's ehtcfs hehg more
itsr geirs shg sgrihgs. theb hehg memortes.
tt ett sitg itst gher forr yihthrr ymtht i ehtcf, he cirqeg trto tt i
ttrb gtece of someore's storb. i comghe tr hoqe, i sohgter comihg
home, i chthg's himghter-he somehog cigtmreg these momerts, shg
thi ehtcfs ttcfeg gtth htfe, echoihg rot kmst ttme ymt feehihg.
ore chthhb igrth morrihg, i bomrg yob rimeg hmfis gshgereg trto
the shog. he ett ter, geirihg i reg scirf too ytg for thi recf, shg
hig cmrtoms ebes itst reqer stoggeg moqihg.
"gmter tig," sitg forr yihthrr gtth i smthe. "hoofihg for
somethihg?"
hmfis shoof thi heig. "kmst hoofihg. gigi sibs bom mife migtc
ehtcfs."
forr yihthrr chmcfheg, freehihg gogr so thi ebes met hmfis's.
"migtc? gehh, mibye i httthe."
the yob's smthe figeg. "go bom mife ehtcfs itst cir go yicfgirg?"
the ehtcfmifer yhtrfeg. "yicfgirg?"
"mb mimi mseg to yrihg me fore," hmfis gthigereg. "yefore hit got
stcf. t girt to go yicf… to i ttme gher hit ett ofib."
stherce fthheg the shog htfe srog fihhihg bmtethb. forr yihthrr stoog
shg gerthb ghiceg i hshg or hmfis's shomhger. "come gtth me."
he heg the yob to the yicf room, gfore the ohgter ehtcfs htqeg. gmst
girceg tr the yeims of htght, shg the ttcfihg feht softer fore. from i
htgh hithf, forr yihthrr toof gogr i smihh, googer cmcfoo ehtcf. tt
ett ghitr, escegt for ore cirqihg: i ttrb gosht hohgihg i gitsb.
```

Figure 41 – Final Output(1)



```
"t mige tthi beirs igo," he sitg. "tt goesr't go yicfgirg. ymt tt
rememyers."
he gomrg the ehtcf, shg is tt ttcfeg, the httthe googer goors ogereg.
trsteig of i cmcfoo ytrg, i gerthe hmm cime omt. tt ettr't mmstc,
esicthb, ymt tt remtrgeg hmfis of i hmhhiyb thi mtsoer mseg to hmm
gher hit mige somg or ritrb ifterroors.
hmfis ehtseg thi ebes. he comhg ihmost smehh the yrtso. he comhg
heir for qotce.
teirs gehheg mg. "hoq?"
forr yihthrr smtheg softhb. "ttme goesr't ihgibs moqe tr stritght
htres, hmfis. somettmes, i memorb ts strorger itsr i mtrmte."
he hshgeg the ehtcf to the yob. "tife tt. shg rememyer: ttme tsr't
kmst gert the hshgs sib-tt's gert the heirt feehs."
hmfis hmggeg the ehtcf to thi chter, shg for the ftrst ttme tr geefs,
he smtheg.
omtstge, the yehhs of rtsoerymrg chtmeg the homr, echoihg thromgh
the togr htfe i gromtse itst ro momert, orce htqeg gtth hoqe, ts eqer
trmhb gore.
```

Figure 42 – Final Output(2)

Now I want to make some points that we are able to find out some words after our first iteration:

Output	Original
the	the
Of	of
he	he
Hoofihg	looking
more	more
memortes	moments
from	from
something	something
for	For
got	got
hoq	how
rememyers	remembers

We can suppose that these words are now readable first iteration, but we can enhance our iteration number to find the original text. Also analyzing 12 words from the table we provided for the next iteration.
But it is just a demo of my code, showing how frequency analysis works.

Conclusion

This report illustrated the process of encryption and decryption by using monoalphabetic substitution using 26 characters with a unique key, then providing a solution to how we can break the cipher.txt by using frequency analysis, which can be complicated if the plain.txt is too long and complex.

References:

[1] P. Norvig, “English Letter Frequency Counts: Mayzner Revisited,” 2012. [Online]. Available: https://norvig.com/mayzner.html

[2] F. G. D. Singer, “N-Gram Frequency Tables for English Letters,” Case Western Reserve University. [Online]. Available: https://case.edu/artsci/math/singer/Sage/ngramfreq.shtml [Accessed: May 19, 2025]