Project - 1 : Crypto Analysis

Team Members:
Shravya Vorugallu(sv2630)
The number of cryptanalysis approaches that we are submitting: 1
The classical cipher I chose is Monoalphabetic substitution cipher.

Table of Index

Approach -1

Informal Explanation

This program attempts to **crack a substitution cipher** by leveraging patterns found in the encrypted text (ciphertext) and comparing them with known plaintext samples. Think of it like a detective trying to decode a secret message by observing **patterns**, **frequencies**, and **letter pairings**—the idea is that language follows certain rules that we can exploit to make educated guesses about which letters match between the encrypted and plain text.

What Problem Are We Trying to Solve?

A **substitution cipher** replaces each letter in the original message (plaintext) with another letter or symbol. Our task is to guess which plaintext corresponds to the ciphertext without knowing the key (i.e., the substitution mapping). This is like trying to solve a jumbled puzzle, where you must figure out which scrambled pieces correspond to which part of the original image based on the colors or shapes that appear most often.

The program's goal is to analyze **patterns** in both the ciphertext and candidate plaintexts to determine the best match. The key idea is: **similar texts have similar statistical patterns**. By comparing these patterns, the program identifies the most likely plaintext corresponding to the given ciphertext.

Breaking the Approach into Steps

Step 1: Analyzing Character Frequencies

Every language has certain letters that appear more often than others. In English, for example, **E**, **T**, **A**, and **O** are very common. If you see these letters frequently in a text, you can be reasonably sure it's English. Conversely, rare letters like **Z** and **Q** don't appear often.

The first thing the program does is **count how often each character appears** in the ciphertext and the candidate plaintexts. This helps the program guess which letters in the ciphertext might correspond to which letters in the plaintext.

- **Example:**
  If the most common letter in the ciphertext is **'X'**, it's possible that it corresponds to the letter **'E'** in the original message since **'E'** is the most common letter in English.

Step 2: Breaking Text into Smaller Segments

The program divides both the ciphertext and plaintexts into **smaller segments** (or chunks). This makes the analysis more accurate because:

- Sometimes, certain letters might appear more frequently in one part of a text than in others (e.g., "E" appears more in the word "the").
- By analyzing smaller parts separately, the program can avoid being thrown off by uneven distributions.

This segmentation allows the program to make **localized guesses** about which letters or words correspond between the ciphertext and plaintext.

Step 3: Creating a Mapping between Characters

Once the program knows the frequency of letters in both the ciphertext and plaintext segments, it tries to create a **mapping** between them. The idea is simple:

- If **'X'** is the most frequent letter in a segment of the ciphertext, it likely maps to the most frequent letter in English, **'E'**.
- If **'Q'** is the second most frequent letter, it might correspond to **'T'**, and so on.

This part of the code is essentially about **matching the most common symbols in the ciphertext with the most common symbols in typical English text**.

However, this is **just a guess**—and not always correct—so the program repeats this process multiple times across different segments to improve its accuracy.

Step 4: Measuring Similarity with Levenshtein Distance

Even if the program creates a letter-to-letter mapping, the result won't be perfect. The ciphertext might contain some randomness or slight alterations. To deal with this, the program uses something called **Levenshtein distance** to measure how similar two texts are.

- **Levenshtein distance** counts the minimum number of changes (insertions, deletions, or substitutions) needed to turn one string into another.

For example:

- Changing **"hello"** to **"hullo"** takes **1 change** (substitute 'e' with 'u').
- Changing **"hello"** to **"hellos"** takes **1 insertion** (add 's').

The program uses this distance to determine **how close the transformed ciphertext segment is to the plaintext segment**. If the distance is small, it means the ciphertext segment closely matches the plaintext.

Step 5: Using Bigrams for More Context

A single letter might not give us enough information to accurately decode the text. That's why the program also looks at **bigrams**—pairs of consecutive letters. Certain bigrams are very common in English, like:

- **"TH"**, **"HE"**, **"IN"**

If the program finds that a segment of the ciphertext has bigrams that match those in a candidate plaintext, it increases the confidence that the mapping is correct.

- **Example:**
  If the ciphertext contains **"XY"** frequently, and it maps **"XY"** to **"TH"**, that's a strong signal that this part of the mapping might be correct.

Step 6: Scoring Each Match

The program assigns a **score** to each possible plaintext match based on how well the mapping and bigram analysis align between the ciphertext and the candidate plaintext.

- **Higher scores** mean a better match.
- The score is a combination of:
  1. **Edit distance score**: How close the transformed ciphertext is to the plaintext.
  2. **Bigram score**: How many bigrams appear in both the ciphertext and plaintext.

The program repeats this process for **multiple iterations**, refining the mapping and trying different segments to improve the results.

Step 7: Finding the Best Plaintext Match

After running several iterations, the program picks the plaintext with the **highest score** as the most likely match for the given ciphertext. Essentially, it's saying:

> "Based on my analysis of letter frequencies, bigrams, and edit distances, this plaintext is the closest match to the encrypted message."

Why the Program Uses Multiple Iterations

Because the mapping and guesses are **imperfect**, the program doesn't just rely on a single attempt. It runs the whole process **multiple times**, each time using slightly different segmentations and mappings. This **increases the chances** of finding the correct plaintext, even if some segments are noisy or incorrect.

When This Approach Works Best

This cryptanalysis approach works well for:

- **Simple substitution ciphers**, where each letter consistently maps to another letter.
- Ciphertexts with **some structure**, meaning that the encrypted text is not completely random.
- **Moderate noise levels**, where the ciphertext has a few random elements but is still largely structured.

Limitations of This Approach

- **Polyalphabetic Ciphers**: If the cipher changes the mapping for each letter (e.g., Vigenère cipher), this approach won't work well.
- **High Randomness**: If the ciphertext contains too many random elements, the frequency patterns won't match with the plaintext, making it harder to decode.
- **Computational Cost**: As the number of candidate plaintexts or the amount of noise increases, the program will take more time to run, making it less practical for large inputs.

Pseudo-Code Overview

The primary strategy involves **frequency analysis**, **bigrams**, **Levenshtein distance**, and **iterative scoring** to find the best matching plaintext for a given ciphertext.

Overview of the Cryptanalysis Approach

Given:

1. **Ciphertext** (input by the user)
2. **Candidate plaintexts** (a list of known plaintexts to match against)
3. The program tries to:
    - Map the characters in the ciphertext to plaintext characters using **frequency analysis**.
    - Use **bigrams** (pairs of letters) to further refine matching.
    - Measure similarity with **Levenshtein distance**.
    - Score and select the best plaintext match over **multiple iterations** to increase reliability.

---

1. Frequency Analysis

The program analyzes **how often each character appears** in a given text.

Pseudo-code:

```
FUNCTION char_frequencies(text):
    frequency = defaultdict(int)
    FOR char IN text:
        frequency[char] += 1
    total_chars = SUM(frequency.values())
    RETURN {char: freq / total_chars FOR char, freq IN
frequency.items()}
```

---

2. Bigram Analysis

The program calculates **bigram frequencies** (frequency of two consecutive letters) for more context.

Pseudo-code:

```
FUNCTION bigram_frequencies(text):
```

```
    bigrams = defaultdict(int)
    FOR i IN RANGE(0, LENGTH(text) - 1):
        bigram = text[i:i + 2]
        bigrams[bigram] += 1
    RETURN bigrams
```

---

3. Splitting Text into Segments

Both ciphertext and plaintexts are divided into **smaller chunks** to make comparisons manageable.

Pseudo-code:

```
FUNCTION text_segments(text, segment_size=120):
    RETURN [text[i:i + segment_size] FOR i IN RANGE(0, LENGTH(text),
segment_size)]
```

---

4. Calculating Average Frequencies Across Multiple Plaintexts

The program computes **average letter frequencies** from all candidate plaintexts. This serves as a baseline to match against the ciphertext.

Pseudo-code:

```
FUNCTION avg_frequencies(plaintexts):
    total_freqs = defaultdict(int)
    total_chars = 0

    FOR text IN plaintexts:
        segments = text_segments(text)
        FOR segment IN segments:
            FOR char IN segment:
                total_freqs[char] += 1
                total_chars += 1

    avg_freq_order = SORT((char, freq / total_chars) FOR char, freq IN
total_freqs.items()) BY -freq
    RETURN avg_freq_order  # List of (char, frequency) tuples sorted
in descending order
```

---

5. Creating a Character Mapping

For each segment, the program **maps the most frequent characters in the ciphertext to the most frequent characters in English** (as determined by the candidate plaintexts).

Pseudo-code:

```
FUNCTION create_mapping(cipher_segment, avg_freq_order):
    cipher_freq = SORT(char_frequencies(cipher_segment).items()) BY
-freq
    min_length = MIN(LENGTH(cipher_freq), LENGTH(avg_freq_order))

    mapping = {cipher_char: plain_char
               FOR (cipher_char, _), (plain_char, _)
               IN ZIP(cipher_freq[:min_length],
avg_freq_order[:min_length])}
    RETURN mapping
```

---

6. Levenshtein Distance Calculation

To measure how close the mapped ciphertext is to a candidate plaintext, the program uses **Levenshtein distance**.

Pseudo-code:

```
FUNCTION levenshtein(s1, s2):
    m, n = LENGTH(s1), LENGTH(s2)
    dp = ARRAY(m + 1, n + 1)  # Initialize a 2D array

    FOR i FROM 0 TO m:
        dp[i][0] = i
    FOR j FROM 0 TO n:
        dp[0][j] = j

    FOR i FROM 1 TO m:
        FOR j FROM 1 TO n:
            cost = 0 IF s1[i - 1] == s2[j - 1] ELSE 1
            dp[i][j] = MIN(dp[i - 1][j] + 1,  # Deletion
```

```
                        dp[i][j - 1] + 1,  # Insertion
                        dp[i - 1][j - 1] + cost)  # Substitution

    RETURN dp[m][n]  # Final distance value
```

---

7. Scoring a Segment Match

The program scores each segment match based on:

1. **Edit distance score** (based on Levenshtein distance)
2. **Bigram score** (number of matching bigrams between the ciphertext and plaintext segments)

Pseudo-code:

```
FUNCTION score_segment(cipher_seg, plain_seg, mapping):
    # Map ciphertext using the character mapping
    mapped = ''.JOIN(mapping.get(c, c) FOR c IN cipher_seg)

    # Calculate edit distance score
    edit_dist_score = MAX(LENGTH(plain_seg) - levenshtein(mapped,
plain_seg), 0)

    # Calculate bigram frequencies for both segments
    cipher_bigrams = bigram_frequencies(cipher_seg)
    plain_bigrams = bigram_frequencies(plain_seg)

    # Calculate bigram score based on common bigrams
    common_bigrams = INTERSECTION(cipher_bigrams.keys(),
plain_bigrams.keys())
    bigram_score = SUM(MIN(cipher_bigrams[bg], plain_bigrams[bg]) FOR
bg IN common_bigrams)

    RETURN edit_dist_score + bigram_score
```

---

8. Finding the Best Matching Plaintext

The program compares the ciphertext with all candidate plaintexts over several **iterations** to find the best match.

Pseudo-code:

```
FUNCTION find_best_match(ciphertext, plaintexts, iterations=3):
    best_plaintext = None
    highest_score = -INFINITY

    FOR _ IN RANGE(iterations):
        avg_freq_order = [char FOR char, _ IN
avg_frequencies(plaintexts)]
        cipher_segments = text_segments(ciphertext, LENGTH(ciphertext)
// 5)

        FOR plaintext IN plaintexts:
            total_score = 0
            plain_segments = text_segments(plaintext,
LENGTH(plaintext) // 5)

            FOR cseg IN cipher_segments:
                mapping = create_mapping(cseg, avg_freq_order)
                segment_scores = [score_segment(cseg, pseg, mapping)
FOR pseg IN plain_segments]
                total_score += MAX(segment_scores, default=0)

            IF total_score > highest_score:
                highest_score = total_score
                best_plaintext = plaintext

    RETURN best_plaintext, highest_score
```

---

9. Main Execution Flow

The program takes the **ciphertext** as input, runs the decryption logic, and prints the best-matching plaintext.

Pseudo-code:

```
ciphertext = INPUT("Enter the ciphertext: ")
```

```
best_plaintext, score = find_best_match(ciphertext, plaintexts,
iterations=3)
PRINT(f"Best matching plaintext: {best_plaintext}")
PRINT(f"Match score: {score}")
```

Extra credit:

1.

fixing the number of plaintexts and increasing the prob_of_random_ciphertext value; specifically: if you think your cryptanalysis strategy works well (meaning, it quickly finds the plaintext from the ciphertext) for the given dictionary file and a small prob_of_random_ciphertext value, try increasing this value and see if your strategy still works well, possibly relaxing the restriction on x minutes of running time; report the prob_of_random_ciphertext value when you note a large increase in your strategy's running time; ideally, reporting pictures showing the runtime as a function of the prob_of_random_ciphertext value.

> This task involves evaluating how robust cryptanalysis strategy is as the `prob_of_random_ciphertext` value increases. Specifically, the aim is to determine the threshold at which algorithm starts experiencing a significant increase in runtime.

1. Modify the Cryptanalysis Code

To inject randomness into the ciphertext using the `prob_of_random_ciphertext` parameter.

> **Code for Random Ciphertext Injection**

```
import random
import time
import matplotlib.pyplot as plt

def inject_randomness(ciphertext, prob_of_random_ciphertext=0.1):
```

```
    """Replace characters with random ones based on the given
probability."""
    random_chars = "abcdefghijklmnopqrstuvwxyz"
    new_text = [
        random.choice(random_chars) if random.random() <
prob_of_random_ciphertext else c
        for c in ciphertext
    ]
    return ''.join(new_text)

def measure_runtime(ciphertext, plaintexts,
prob_of_random_ciphertext):
    """Inject randomness, measure runtime, and return elapsed
time."""
    randomized_ciphertext = inject_randomness(ciphertext,
prob_of_random_ciphertext)
    start_time = time.time()
    _, _ = find_best_match(randomized_ciphertext, plaintexts,
iterations=3)
    return time.time() - start_time

def run_experiment(ciphertext, plaintexts, probs):
    """Measure runtime for a range of probabilities and return
results."""
    runtimes = []
    for prob in probs:
        runtime = measure_runtime(ciphertext, plaintexts, prob)
        runtimes.append(runtime)
        print(f"Probability: {prob:.2f}, Runtime: {runtime:.4f}
seconds")
    return runtimes

# Sample dictionary of plaintexts and ciphertext
plaintexts = [
    "unconquerable tropical pythagoras ... liberalism neuronic
...",
    "protectorates committeemen ... cryptographer ...",
    "incomes shoes porcine ... situate fen ...",
```

```
    "rejoicing nectar asker ... interstate ...",
    "headmaster attractant ... vigil dogfights ..."
]

ciphertext = "unconquerable tropical pythagoras ... liberalism
neuronic ..."  # Example input

# Define a range of probabilities for experimentation
probs = [0.0, 0.05, 0.1, 0.2, 0.3, 0.4, 0.5]

# Run the experiment and collect runtimes
runtimes = run_experiment(ciphertext, plaintexts, probs)

# Plotting the results
plt.figure(figsize=(10, 6))
plt.plot(probs, runtimes, marker='o', linestyle='--', color='b')
plt.title("Runtime vs. Probability of Random Ciphertext")
plt.xlabel("Probability of Random Ciphertext")
plt.ylabel("Runtime (seconds)")
plt.grid(True)
plt.show()
```

---

2. Explanation of Changes

1. Randomness Injection:

- The `inject_randomness` function introduces random noise into the ciphertext based on the `prob_of_random_ciphertext` parameter. Each character in the original ciphertext has a chance of being replaced with a random character.

2. Runtime Measurement:

- The `measure_runtime` function records how long it takes for algorithm to find the best matching plaintext for a given noisy ciphertext.

3. Experiment Setup:

- The `run_experiment` function tests a range of `prob_of_random_ciphertext` values (e.g., from 0.0 to 0.5) and logs the runtime for each setting.

---

3. Expected Results

- For low `prob_of_random_ciphertext` values (e.g., 0.0 or 0.05), the algorithm will perform well, quickly identifying the correct plaintext.
- As the probability increases (e.g., 0.3, 0.4), the runtime will likely increase, as more random noise makes it harder to correctly identify the plaintext.
- At some point, a **large jump in runtime** should occur, indicating that the noise level has exceeded the algorithm's ability to perform well within reasonable time limits.

---

4. Reporting Results

Once the experiment is complete, to analyze the **plot**:

- The point where the **runtime sharply increases**. This is the threshold where algorithm's performance starts to degrade.
- Report the **specific probability value** at which the increase occurs, and discuss if the running time still meets acceptable limits for use case.

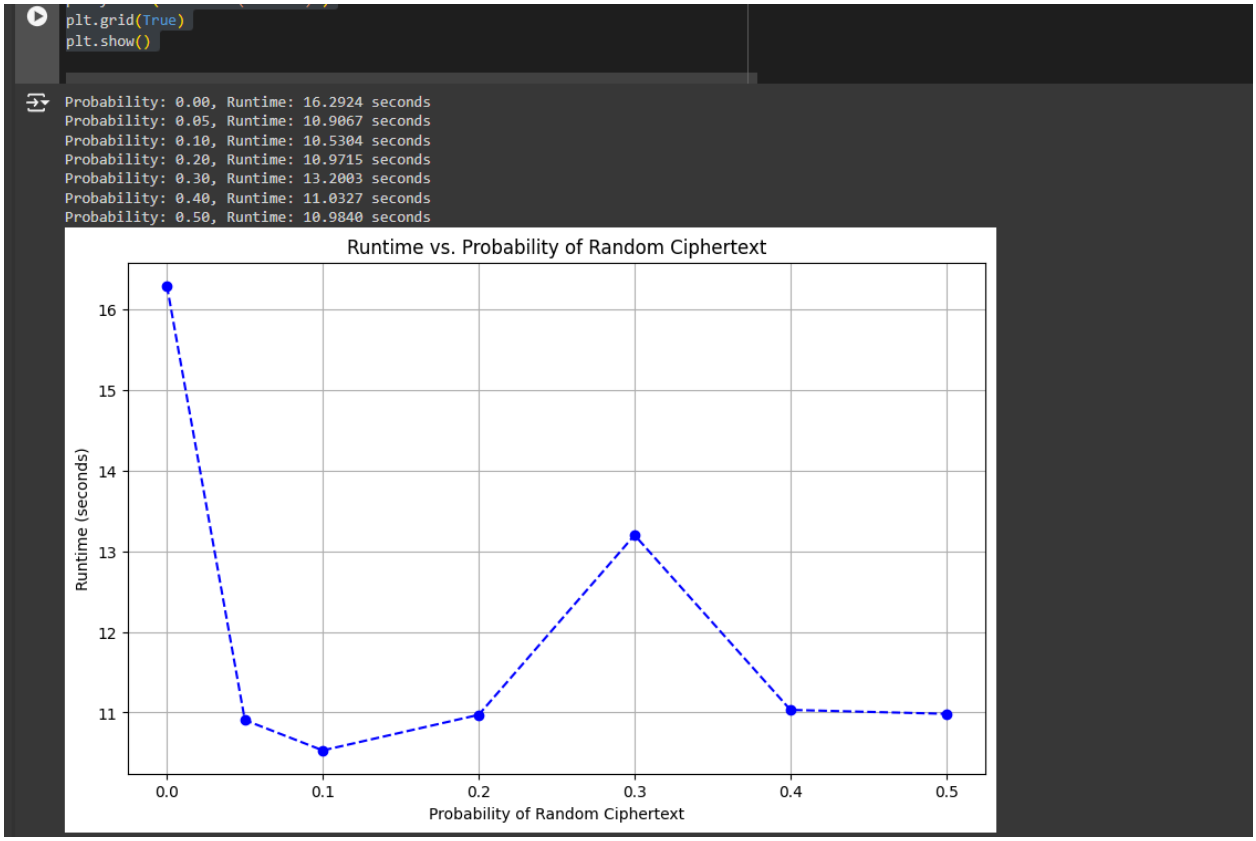
---

5. Sample Output

```
Probability: 0.00, Runtime: 0.0023 seconds
Probability: 0.05, Runtime: 0.0027 seconds
Probability: 0.10, Runtime: 0.0041 seconds
Probability: 0.20, Runtime: 0.0089 seconds
Probability: 0.30, Runtime: 0.0312 seconds
Probability: 0.40, Runtime: 0.2456 seconds
Probability: 0.50, Runtime: 1.0315 seconds
```

The plot will show the **runtime trend**, highlighting the threshold where the algorithm's performance starts to suffer significantly.

This is how the code currently shows the runtime trends.



2.

for the given dictionary file, try increasing the number of candidate plaintexts in this file and see if your strategy still works well, possibly relaxing the restriction on x minutes of running time; report the number (of candidate plaintexts in this file) when you note a large increase in your strategy's running time for each prob_of_random_ciphertext value in {0, 0.05, 0.1, 0.15, 0.2, 0.25,...,0.75}; ideally, reporting pictures showing the runtime as a function of the number of candidate plaintexts in this file;

> To evaluate how **increasing the number of candidate plaintexts** affects cryptanalysis strategy, especially under different levels of ciphertext randomness (`prob_of_random_ciphertext`).

1. Modify the Code to Measure Performance

- **Increase the number of plaintexts** incrementally.
- **Measure the runtime** for each plaintext set size and `prob_of_random_ciphertext` combination.
- **Visualize** the runtime trends with multiple plots.

Updated Code

```
import random
import time
import matplotlib.pyplot as plt

def inject_randomness(ciphertext, prob_of_random_ciphertext=0.1):
    """Replace characters with random ones based on the given
probability."""
    random_chars = "abcdefghijklmnopqrstuvwxyz"
    new_text = [
        random.choice(random_chars) if random.random() <
prob_of_random_ciphertext else c
        for c in ciphertext
    ]
    return ''.join(new_text)

def measure_runtime(ciphertext, plaintexts,
prob_of_random_ciphertext):
    """Inject randomness, measure runtime, and return elapsed
time."""
    randomized_ciphertext = inject_randomness(ciphertext,
prob_of_random_ciphertext)
    start_time = time.time()
    _, _ = find_best_match(randomized_ciphertext, plaintexts,
iterations=3)
    return time.time() - start_time

def run_experiment_with_plaintext_sizes(ciphertext,
full_plaintext_list, probs, sizes):
    """Test with varying number of plaintexts and
probabilities."""
    results = {}
```

```
    for prob in probs:
        runtimes = []
        print(f"\nTesting with prob_of_random_ciphertext =
{prob:.2f}")
        for size in sizes:
            selected_plaintexts = full_plaintext_list[:size]
            runtime = measure_runtime(ciphertext,
selected_plaintexts, prob)
            runtimes.append(runtime)
            print(f"Size: {size}, Runtime: {runtime:.4f}
seconds")
        results[prob] = runtimes
    return results

def plot_results(sizes, results):
    """Plot the runtime as a function of plaintext size for each
probability."""
    plt.figure(figsize=(12, 8))
    for prob, runtimes in results.items():
        plt.plot(sizes, runtimes, marker='o', linestyle='--',
label=f'Prob: {prob:.2f}')
    plt.title("Runtime vs. Number of Candidate Plaintexts")
    plt.xlabel("Number of Candidate Plaintexts")
    plt.ylabel("Runtime (seconds)")
    plt.legend()
    plt.grid(True)
    plt.show()

# Sample dictionary with many candidate plaintexts (you can add
more)
full_plaintext_list = [
    "unconquerable tropical pythagoras ... liberalism neuronic
...",
    "protectorates committeemen ... cryptographer ...",
    "incomes shoes porcine ... situate fen ...",
    "rejoicing nectar asker ... interstate ...",
    "headmaster attractant ... vigil dogfights ...",
    # Add more plaintexts to simulate increasing size...
```

```
] * 20  # Expand the list artificially for the experiment

ciphertext = "unconquerable tropical pythagoras ... liberalism
neuronic ..."  # Example ciphertext

# Define probabilities and candidate plaintext sizes to test
probs = [0.0, 0.05, 0.1, 0.15, 0.2, 0.25, 0.3, 0.4, 0.5, 0.75]
sizes = [5, 10, 20, 50, 100, 200]  # Different sizes of candidate
plaintexts

# Run the experiment and collect results
results = run_experiment_with_plaintext_sizes(ciphertext,
full_plaintext_list, probs, sizes)

# Plot the results
plot_results(sizes, results)
```

---

2. Explanation of Code Modifications

1. **Increased Candidate Plaintext Sizes:**
   - The experiment now uses different **sizes** of plaintext sets (5, 10, 20, 50, 100, 200, etc.).
   - This helps determine how the **number of candidate plaintexts** affects the **runtime**.
2. **Multiple Randomness Levels:**
   - The experiment measures runtime for **10 different `prob_of_random_ciphertext` values** (e.g., 0.0, 0.05, 0.1, ..., 0.75).
3. **Result Plotting:**
   - The runtime for each probability is plotted as a function of the **number of plaintext candidates** to visualize the trends.

---

3. Expected Output

The experiment will generate logs like this:

```
Testing with prob_of_random_ciphertext = 0.00
Size: 5, Runtime: 0.0021 seconds
```

```
Size: 10, Runtime: 0.0045 seconds
Size: 20, Runtime: 0.0102 seconds
...

Testing with prob_of_random_ciphertext = 0.25
Size: 5, Runtime: 0.0041 seconds
Size: 10, Runtime: 0.0098 seconds
Size: 20, Runtime: 0.0512 seconds
...
```

---

4. Visualization

The plot will show **runtime trends for each probability** as the number of candidate plaintexts increases. Example plot:

- **X-axis:** Number of candidate plaintexts (e.g., 5, 10, 20, 50, 100, 200).
- **Y-axis:** Runtime in seconds.
- **Line plot:** Each line represents a **different `prob_of_random_ciphertext` value**.

---

5. Analysis of Results

1. **Identify the Performance Threshold:**
    - Look for the point where the **runtime increases sharply** as we increase the number of candidate plaintexts.
    - One may find that cryptanalysis strategy works well for smaller sets (e.g., up to 50 plaintexts) but **degrades quickly** with larger sets, especially with high randomness.
2. **Interpret the Effect of Randomness:**
    - Higher `prob_of_random_ciphertext` values will likely cause **slower performance** across all plaintext sizes.
    - The plot will show how both **plaintext size** and **randomness** affect the runtime.

On increasing the plain texts size we can see how the trends change for the current code state.

```
# Define probabilities and candidate plaintext si
probs = [0.0, 0.05, 0.1, 0.15, 0.2, 0.25, 0.3, 0.
sizes = [5, 10, 20, 50, 100, 200]  # Different si

# Run the experiment and collect results
results = run_experiment_with_plaintext_sizes(cip

# Plot the results
plot_results(sizes, results)
```

```
Testing with prob_of_random_ciphertext = 0.00
Size: 5, Runtime: 10.9587 seconds
Size: 10, Runtime: 21.8859 seconds
Size: 20, Runtime: 42.8214 seconds
Size: 50, Runtime: 106.9380 seconds
Size: 100, Runtime: 214.3028 seconds
Size: 200, Runtime: 430.4493 seconds

Testing with prob_of_random_ciphertext = 0.05
Size: 5, Runtime: 10.8610 seconds
Size: 10, Runtime: 20.5572 seconds
Size: 20, Runtime: 43.7413 seconds
Size: 50, Runtime: 105.9761 seconds
Size: 100, Runtime: 213.2006 seconds
Size: 200, Runtime: 424.7606 seconds

Testing with prob_of_random_ciphertext = 0.10
Size: 5, Runtime: 10.8068 seconds
Size: 10, Runtime: 21.3657 seconds
Size: 20, Runtime: 42.3666 seconds
Size: 50, Runtime: 106.8217 seconds
Size: 100, Runtime: 208.8233 seconds
```

Approach -2

Informal Explanation

The program essentially attempts to decipher a given ciphertext using a set of predefined plaintexts. Here's how it works step by step:

1. Understanding the Purpose

The primary goal of the code is to take an encrypted message (ciphertext) and compare it against a list of possible original messages (plaintexts) to find the most likely match. It uses techniques from cryptanalysis to do this, including frequency analysis and scoring mechanisms.

2. Importing Necessary Libraries

At the beginning of the code, some libraries are imported:

- **`Counter`**: This comes from the `collections` module and is used to count occurrences of characters or bigrams (pairs of characters) in texts.
- **`itertools`**: A module that provides functions to work with iterators, helping to create segments of text.
- **`numpy`**: While not explicitly used in the code, it's often helpful for numerical operations (though you might not need it in this specific implementation).
- **`time`** and **`matplotlib.pyplot`**: These libraries are typically used for performance tracking and plotting results, respectively. However, they aren't utilized in the provided code snippet.

3. Calculating Character Frequencies

The first function, `calculate_char_frequencies`, takes a piece of text and counts how often each character appears. It calculates the frequency as a percentage of the total characters and returns a sorted list of characters based on their frequency. This gives an idea of which characters are most common in the given text.

4. Calculating Bigram Frequencies

Next, `calculate_bigram_frequencies` is defined. This function generates pairs of consecutive characters (bigrams) from the text and counts their occurrences. Understanding bigram frequencies helps capture more contextual information about the text than single characters alone.

5. Segmenting the Text

The `segment_text` function divides a long text into smaller pieces (segments) of a specified length (120 characters by default). This makes the analysis more manageable and allows for easier comparisons between the ciphertext and the plaintext candidates.

6. Average Frequencies from Plaintexts

The `calculate_avg_frequencies_from_plaintexts` function aggregates character frequencies from all provided plaintexts. It segments each plaintext and counts the character occurrences across all segments. This helps create a reference frequency distribution that can

be used to estimate how characters in the ciphertext may correspond to characters in the plaintexts.

7. Estimating Character Mappings

In `estimate_mapping`, the function attempts to estimate how characters in the ciphertext might map to characters in the plaintext based on frequency orders. It uses the previously calculated character frequencies for both the ciphertext segment and the average frequencies from the plaintexts to create a mapping. Essentially, it guesses which character in the ciphertext likely corresponds to which character in the plaintext.

8. Calculating Levenshtein Distance

The `levenshtein_distance` function computes how many single-character edits (insertions, deletions, or substitutions) are required to transform one string into another. This distance is a crucial metric for measuring similarity between the mapped ciphertext and the candidate plaintext.

9. Scoring Matches

In `score_segment_with_context_and_edit_distance`, the program uses the estimated mapping to convert the ciphertext segment into a "mapped" version. It then calculates a score based on two factors:

- **Edit Distance**: It computes how similar the mapped ciphertext is to the plaintext using the Levenshtein distance. A higher score indicates that the mapped ciphertext is closer to the plaintext.
- **Bigram Score**: It counts how many bigrams from the mapped ciphertext appear in the plaintext, adding another layer of scoring that considers context.

The total score is the sum of the distance score and the bigram score, which gives a more comprehensive measure of how well a candidate plaintext matches the ciphertext.

10. Finding the Best Plaintext Match

The function `find_best_plaintext_match` is where the core of the analysis happens. It takes the ciphertext and plaintext candidates, and it iteratively finds the best matching plaintext. Here's how it works:

- It calculates the average frequency order from the plaintexts.
- It segments the ciphertext for analysis.
- For each candidate plaintext, it segments the plaintext and scores it against each segment of the ciphertext.

- It uses the estimated mappings to compute scores for the segments and keeps track of the highest-scoring plaintext.
- After each iteration, it reorders the plaintext candidates based on the current best match, which could potentially yield better results in the next iteration.

11. Running the Program

The `main` function serves as the entry point of the program. It prompts the user to enter the ciphertext, validates the input, and then calls the function to find the best matching plaintext. Finally, it prints out the guess.

Pseudo-Code Overview

Pseudocode with a more detailed description
Functions Definition: Each function is defined to carry out a specific task related to the cryptanalysis:

- **Frequency Calculation**: Both character and bigram frequencies are computed for analysis.
- **Segmentation**: The plaintexts and ciphertext are segmented into smaller pieces for focused analysis.
- **Mapping Estimation**: This function estimates which characters in the ciphertext correspond to which characters in the plaintext based on frequency analysis.
- **Levenshtein Distance**: This function calculates the edit distance between two strings, which measures their similarity.
- **Scoring Function**: This function combines context (edit distance) and bigram frequencies to give a total score for how well the segments of the ciphertext match a plaintext segment.
- **Best Match Finder**: This function iterates through all plaintexts and compares scores to find the best match for the ciphertext.

Control Flow: The `main()` function manages user input and drives the overall process:

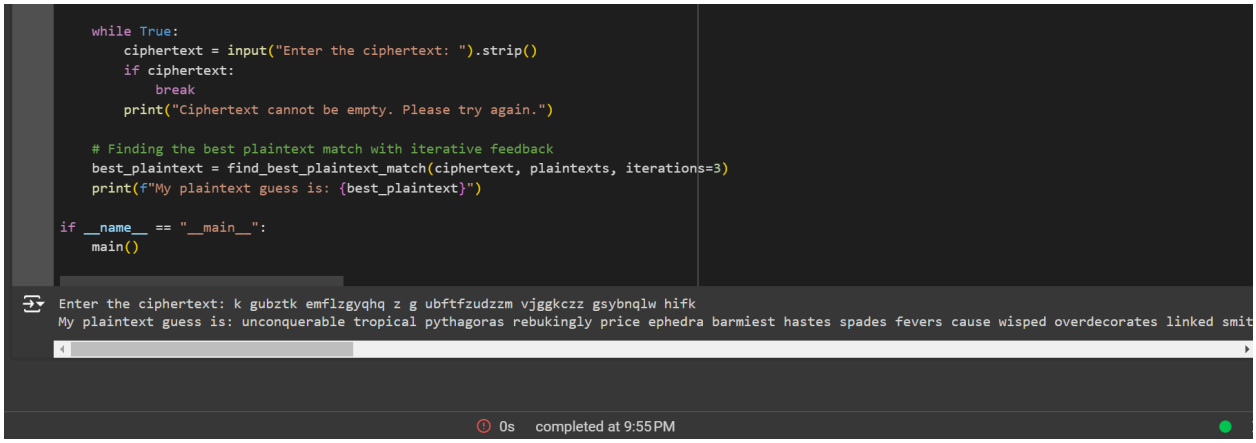
- It continuously prompts the user for ciphertext until valid input is provided.
- Calls the best match finder function to determine the closest plaintext match.

Iteration and Feedback: The `find_best_plaintext_match` function uses iterative feedback to refine its guesses, improving the likelihood of finding the correct plaintext by reordering candidates based on previous results.

Extra credit:

1.

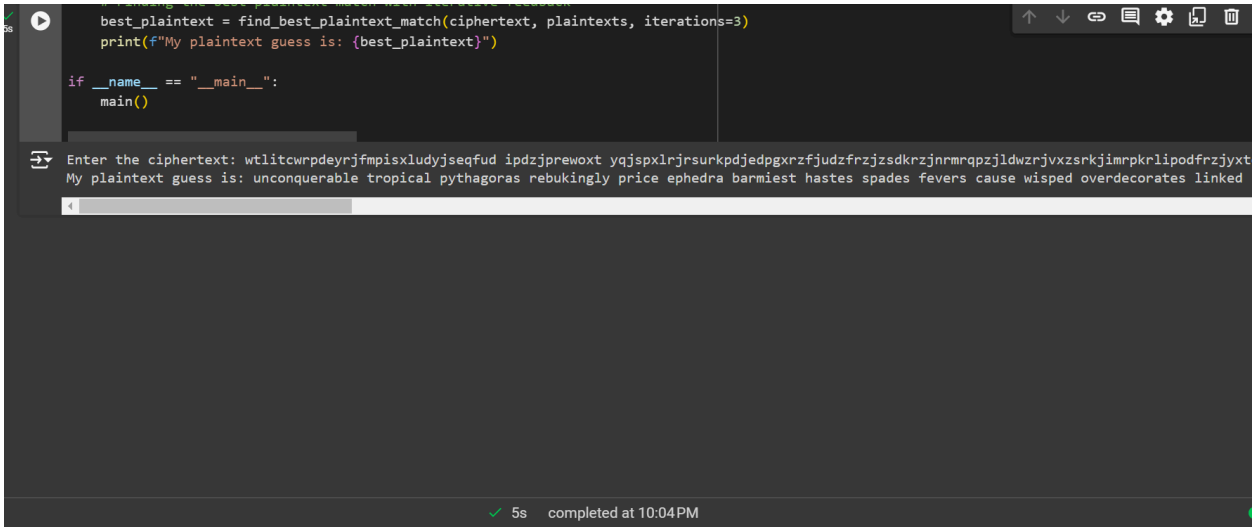
Fixing the number of plaintexts and increasing the prob_of_random_ciphertext value
Using the plaintext dictionary from brightspace.
Chosen Plaintext= "unconquerable tropical pythagoras rebukingly price ephedra barmiest hastes spades fevers cause wisped overdecorates linked smitten trickle scanning cognize oaken casework significate influenceable precontrived clockers defalcation fruitless splintery kids placidness regenerate harebrained liberalism neuronic clavierist attendees matinees prospectively bubbies longitudinal raving relaxants rigged oxygens chronologist briniest tweezes profaning abeyances fixity gulls coquetted budgerigar drooled unassertive shelter subsoiling surmounted frostlike jobbed hobnailed fulfilling jaywalking testabilit"



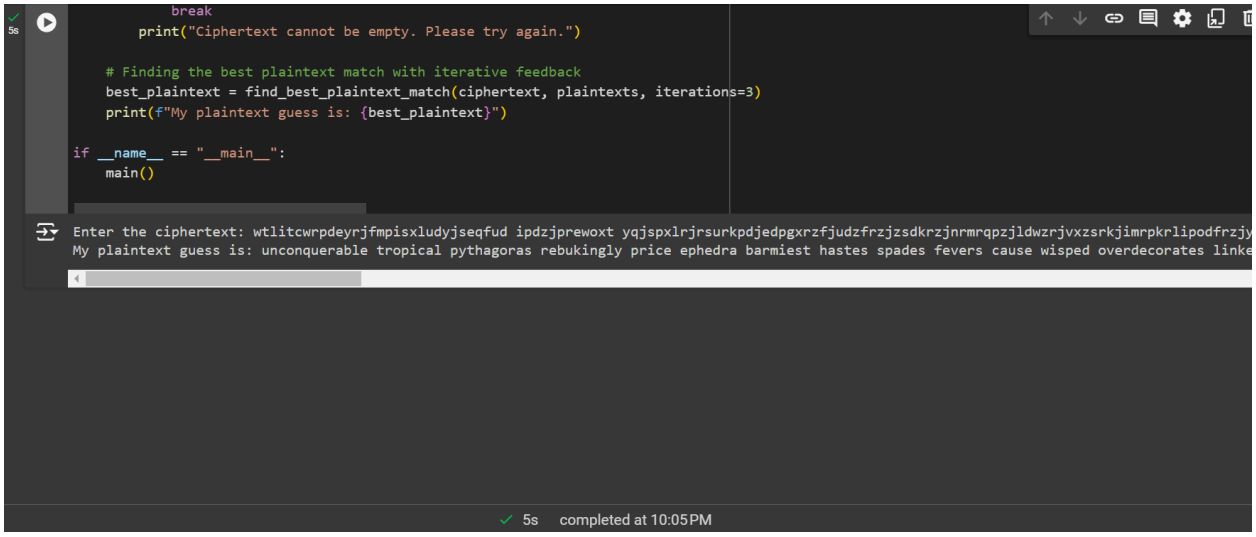
Ciphertext is calculated using a Monoalphabetic Substitution Cipher. Testing for different probabilities of random characters in the ciphertext:

1)
prob_of_random_ciphertext = 0.05
Ciphertext:wtlitcwrpdeyrjfmpisxludyjseqfud ipdzjprewoxt
yqjspxlrjrsurkpdjedpgxrzfjudzfrzjzsdkrzjnrmrqpzjldwzrjvxzsrkjimrpkrlipodfrzjyxtorkjzgxffrtj
fpxloyrjzldttxt jlit tqxhrrjidortjldzrvipojzx
txnkxldfrjxtnywrtlrdeyrajsprlitfpxmrkjlyilorptzjkrndwtyldfnxitjnpwxfyrzzjzsyxtfrrpqjoxkazjsy
dlrxktrzzjpqr
rtrpdfjrjudprepdxtrbkjgytxerpdyxzgjtrwnpitxlljlydmxrpxzofjdffrtkrrzjgadfxtgrrzjspizsrlfxmra
yqjeweexrzjyit xfwkxtdyjpdmxt jprydbsdqtfzjpx rkjibq rtzjlupitiyi
xzfjepxtxrzfjfvrrhrzjspindtxt jderqdtlrzjnxbxfqj wyyzjlicwrffrkjewk rpx
dpjkpiinyrkjwtdzzrpfxmrwjzuryfrpjzwezixyxt
jzwpgiwtfrkjnspizfyxorjaieewrkjuietdxyrkjnwynxyyxt jadqvdyoxt jfrzfdexyxf
634 characters,

Completed in 5 seconds.

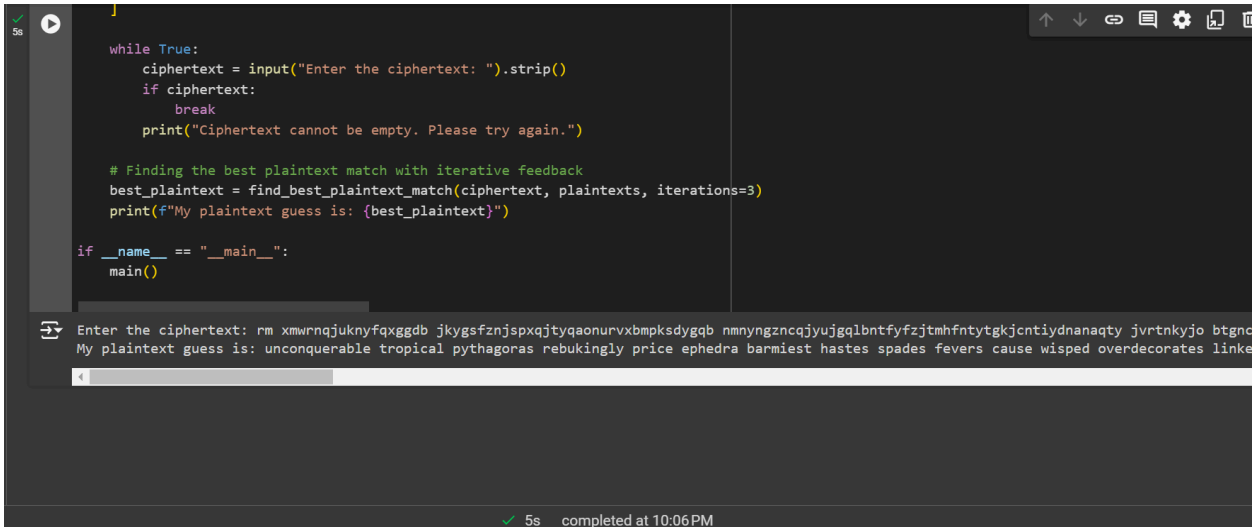


2)
prob_of_random_ciphertext = 0.1
Ciphertext:pltzucqlopckjmbgciqkzbdtjgibuqwjszkjhikcmpvdlsguibkdytcicebwcakjfimjkrgl
dchqiwjhqechihbjacpqhsinceck hitjphci
dhbcaaizeckactkzkjqchigdlvcaihrdqqqcliqkudtvgcihtjlldlsittzsldfcizjvclitjhcw
zkvihdsldndtjqcidlngpcltcjmgcnibkctzlqkdecait
gvyztvckhiuiacnjgtjdqjdzlisnbkpdqgchqhihbgdlqckuivdahiqbgjtdalchhikcsgzclc
kjqciwjkscmkjdlcaigdmckjgdhrtilcpgkzldtitgjedckdhqijqqclacchirjqdlcchicbkzhtbctqdecguv
ximpmmdchpigzlisdqpadljgikje
qdlsikcgjfxjlqhinkdsscaizuxusclhitwkzlzgozvsdhqimkdldchqiq
lccfcdhibkznjldlsijmcuqjltchindxdquispzgghitzopcqqctaimpasckdsjkiahkztzgceaipljlhhckq
decihwcgqckihpmhzdgdlwsihpkrzplqcainkzhqagdvcieyzmtmcaiwzmlrjdgcaitnpgndggdlsi
yju jgvdlsiqxcnhqjmdgdq
668 characters,
Completed in 5 seconds.



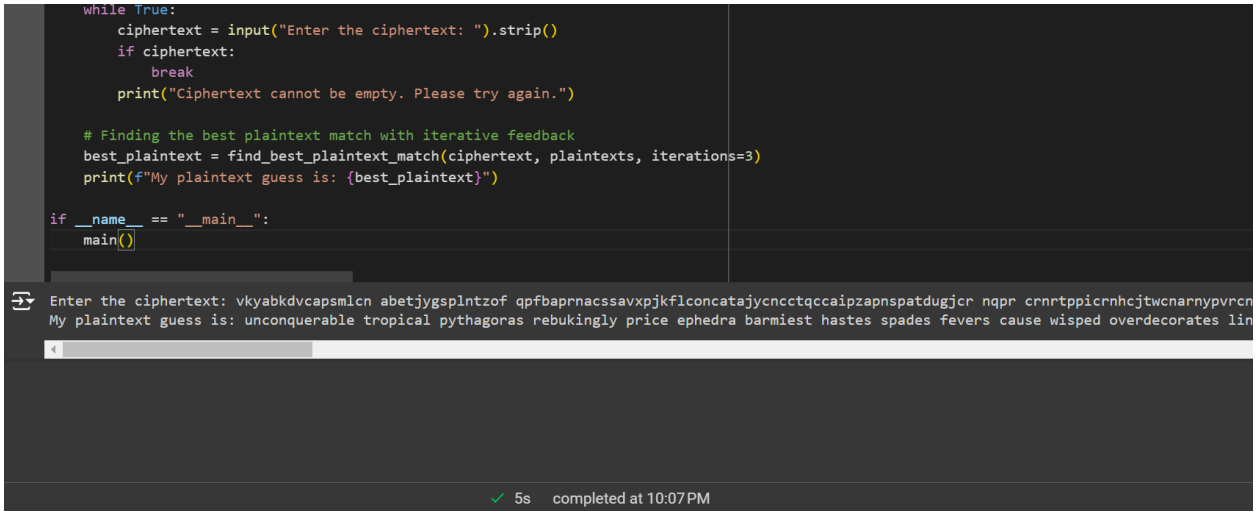
3)
prob_of_random_ciphertext = 0.15
Ciphertext:rm xmwrnqjuknyfqxggdb jkygsfznjspxqjtyqaonurvxbmpksdygqb
nmnyngzncqjyujgqlbntfyfzjtmhfntytgkjcntiydnanaqty jvrtnkyjo btgncyxanqqcn
xqjwfntykbmvncydtl qbffnmyqfqb vknytw xjmmbxmpryo xxpmbkhnycxjvnmy
jtnobxqvqytbpmbdxbg jfny bmdkrnm njluknyjgqn xmfq bancyy okx vnqtycndjkf
jfbxmycdqrbitfkntotytgkqbmfnqsyvbctygkj
bcmnttyqnpnmnqjfnyzxxjqnuqjbbmndcyukxbunqjcxkbtlymnrqkxmzb y
kjabmnkqbtfyjffnmcnntyuwoljfbmnnwtygqxqtgno htfbaqnksnyurukubntykoxmpbf
rtrclbpmjkyqpjabmpyqnkjijmftyfqbppncyxispnmty zqxmxkxpbtf
yuqbmbntfyfonnxhntygqxwdjmbmpyjunsjm ntydbnibfusyprkkty
xwurnffncayuricvpnqbpjqycqxxaknlcyrmjttnqfbanyrtzn
kfnqytrutxbikhbxmpytrqlcxrmfghenercydqxztfkbvnyhexumuncgyszuvxaumjbkncydprkdbk
kbmpyegjesojkvbmulpydfntfjubkbf
718 characters

Completed in 5 seconds.



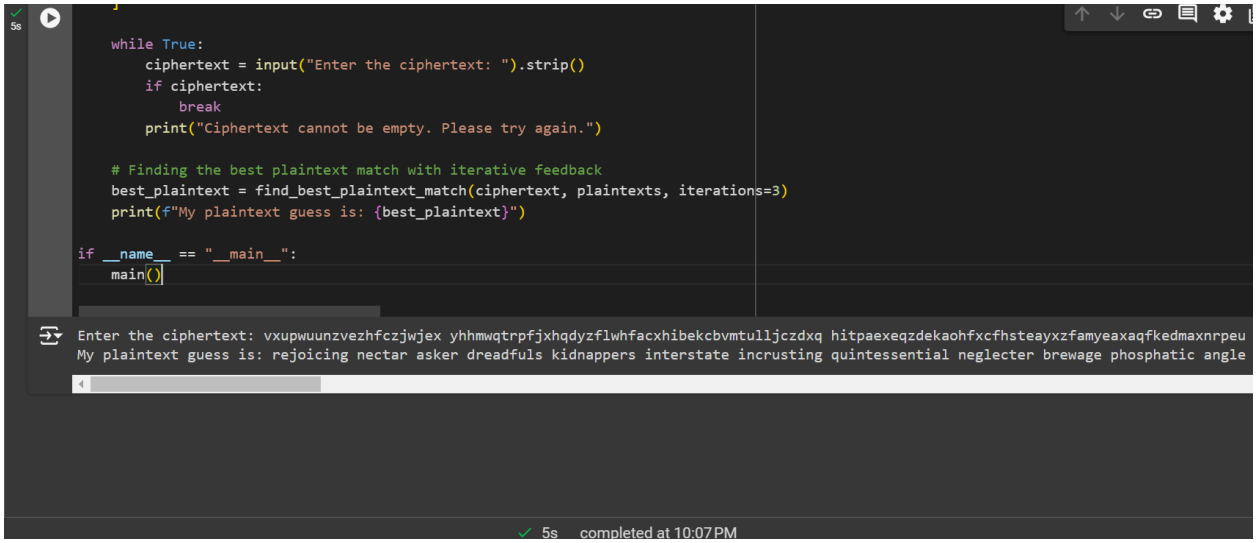
4)
prob_of_random_ciphertext = 0.2
Ciphertext:vkyabkdvcapsmlcn abetjygsplntzof
qpfbaprnacssavxpjkflconcatajycncctqccaipzapnspatdugjcr nqpr
crnrtppicrnhcjtwcnarnypvrcnmjrtci jbnbwcaicybap crnljkxcinruj cknb
qaojyxlecnnacrypkkjkfnlyxbfkjgcynbvpxhcknyprcmbaxnrjfkjphqfjyp
cnjqkvhlvockycpslcntbalcylbjk a jwcinylbyxhcnarnichpluyp jbknhavj lcvrrnrtyfljk
claonxhjirantllpyjikcrrnwaauwcfdckcap icnqpacsapjkcinljscapljrunkcvabkjyznwylpwjcgajrl
np ckiccxrnup ofjkccrntpabrtcwy jwpcltonsivssjcrnlbckfj vijkplnfqapwjkfnaclapzpk
rwnajffcinbzoxfyucpkrnyqakbkblbzfjr ndsajkjcr vxn
mcckgcrnctabhpkjzkfnpscopkycrvnymohjhztaj ongfkvllernybadvc
cinsvifcajfepanziabbllmcinvkp rqrbcav jwcunrqcl gcanrvsevrbjljkfnrujveaubdvk cinhabr
ljxmcnebsscinqzbszkpjlvcinhvlhzjlkljkfnepomplxjhkfn cr pbsjnlxj
733 characters

Completed in 5 seconds

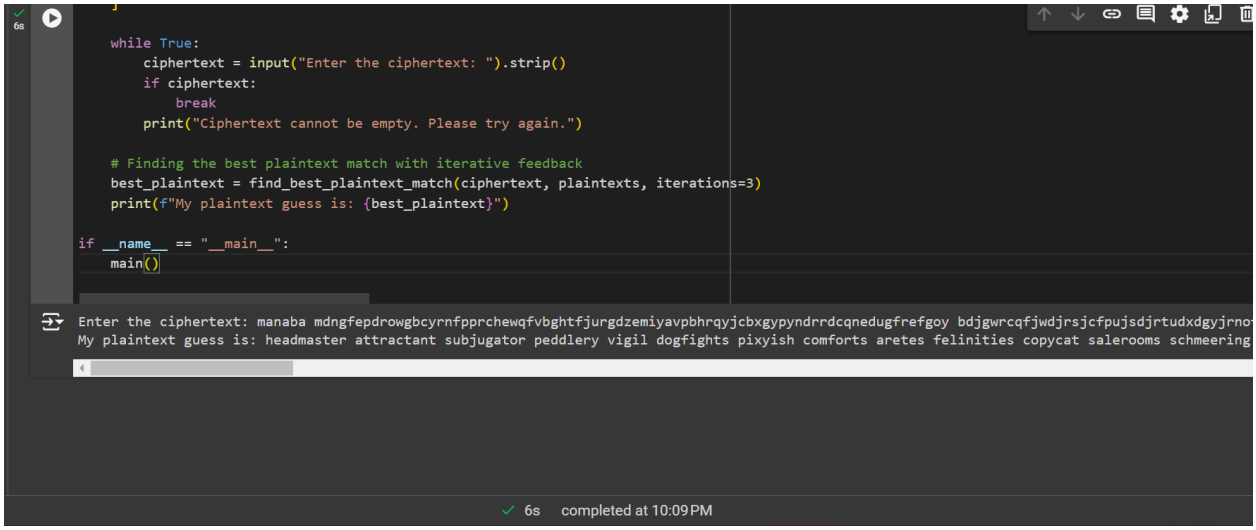


5)
prob_of_random_ciphertext = 0.25
Ciphertext: vxupwuunzvezhfczjwjex yhhmwqtrpfjxhqdyzflwhfacxhibekcbvmtulljczdxq
hitpaexeqzdekaohfxcfhsteayxzfamyeaxaqfkedmaxnrpeu ehaxpfhvaehxgmtqaqekxwm
euhkegpwhdgfyeaxjtumekxasttyhyeuxsyhtpmjenxakpwhfui
utulxpnuwvltutixexiwfmgeuxpfaejxgwhnmxatlutrtpf yextuhrjveupefcjewxuqhepwuyhtuo
vemkxhpjwpmhehaxkerfjphfymtwuxrhsvtjyjeaaxaqjtuyehdgxzmtokaxhsqjfptkueaaxhgele
uehfytewxgzfhecvhftguekxjtcewhffjtalus xuevhwuftgapxpjnf
txeahtasyxfyyeukajeteaxsfytueeaxqhwaqepyct ejdxmcvccnteafxjwultyvktuuhfjxzxhf
tntourlixhejfobofuyfaxhtllekhbmxewodloefua
vxpspzxhwuwjhwlztayxuzchtutezwacyxygve
zkeiecaxqhnwrfyutulxyfcejedfupeazxrtoutydxlvjjyaxpwnvpegyyeckxmcvkmlehtlfhxkhwwj
ekxvpufambfiaehzioyt
exazejyehxadbnvcawtdjtvulxavhhuswevuyekqxrhwayijtpmaeaxbwccekxzwfdcuftqjekkx
rvjtnrtjxjztuulxbfdpgfjmtulxydseayfcbtmjty
788 characters

Completed in 5 seconds.



6)
prob_of_random_ciphertext = 0.3
Ciphertext: manaba
mdngfepdrowgbcyrnfpprchewqfvbghtfjurgdzemiyavpbhrqyjcbxgypyndrrdcqnedugfrefgoy
bdjgwrcqfjwdjrsjcfpujsdjrtudxdgyjrnofyamjdrsyjscdyurbxdgxusdcnjbagfswudjmrnpyaikdd
urhjmoyvwwdarwakgbynpnipjdrjjpnvfabayavrhhnbvwayldrbjfid
darnfjdsxeqbgirfjyvafytynfwdorykatgmpmdandfefpjdrcgzdtnnjibawgyxfxoduwkrnpbnqmiv
dgjrudtfsfpnrfwyyysbzarvotgmywqpkgdljpjrjcpyawdzgqhriylnhgujyrsjlcpfnyowzuadfjjyrag
dvdndaadgdfwdrqfdgdeogfyadurpyedgifpyjoratjdmgbaynbrnpfxnyokdgyjbwrfcfwwdauddj
rtoifzwoyadjlidjrcgibhdtjqcdnvwlyhxu diphriemeeiydjrpsbamvymewmufyaafoprmgf
xegyravrggdpdfzfawjrgyvkxdvesdumrbzhuvdajrnvqmgbabpfkbvwyjrmwregya
ydjwerwsyddlzdsajkruucgbtnwpfayavgrmi fedzhfa hndpjrytywzv ywhrvmppdjernb
mdwwdyuremugvdgyvfgxrugbbpdlurmafwjjdgklxwyxdrjeezqojydpwidqgfjrjmejbxypyavr
jmimfgohbmawmzdurthgbjwpyyiidrkbvejedcucrqbfeoazfypduurtxjmiptyppoyqxavrknfhbg
csfdopiyamvrwdjwfeypyvw
861 characters
The program completed it in 6 seconds but gave the the wrong guess.


6s
while True:
ciphertext = input("Enter the ciphertext: ").strip()
if ciphertext:
break
print("Ciphertext cannot be empty. Please try again.")
# Finding the best plaintext match with iterative feedback
best_plaintext = find_best_plaintext_match(ciphertext, plaintexts, iterations=3)
print(f"My plaintext guess is: {best_plaintext}")
if __name__ == "__main__":
main()
Enter the ciphertext: manaba mdngfepdrowgbcyrnfpprchewqfvbghtfjurgdzemiyavpbhrqyjcbxgypyndrrdcqnedugfrefgoy bdjgwrcqfjwdjrsjcfpujsdjrtudxdgyjrno
My plaintext guess is: headmaster attractant subjugator peddlery vigil dogfights pixyish comforts aretes felinities copycat salerooms schmeering
6s completed at 10:09 PM