CS4520 Security & Cryptography - Assignment 1

Dimitrios Ntatsis - Student number: 6216269

December 6, 2024

Exercise 1

A

The encryption scheme is a Caesar Cipher, with an encryption key k = 5 (plaintext is shifted by 5 characters).
The decrypted plaintext is: To be, or not to be, that is the question: Whether 'tis nobler in the mind to suffer The slings and arrows of outrageous fortune, Or to take arms against a sea of troubles And by opposing end them. To dieto sleep, No more; and by a sleep to say we end The heart-ache and the thousand natural shocks That flesh is heir to: 'tis a consummation Devoutly to be wish'd.

B

The encryption scheme is a Vigenere Cipher, with an encryption key k = "oppenheimer" (first character is shifted by 14 characters, second and third one by 15, etc).
The decrypted plaintext is: Reveal Thy Self; what awful Form art Thou? I worship Thee! Have mercy, God supreme! Thine inner Being I am fain to know; This Thy forth-streaming Life bewilders me. Time am I, laying desolate the world, Made manifest on earth to slay mankind! Not one of all these warriors ranged for strife Escapeth death; thou shalt alone survive. Therefore stand up! win for thyself renown, Conquer thy foes, enjoy the wealth filled realm, By Me they are already overcome, Be thou the outward cause, left-handed one. Drona and Bhshma and Jayadratha, Karna, and all the other warriors here, Are slain by Me. Destroy them fearlessly. Fight! thou shalt crush thy rivals in the field. Sanjaya said: Having heard these words of Keshava, he who weareth a diadem, with joined palms, quaking and prostrating himself, spake again to Krishna, stammering with fear, casting down his face.

C

I wrote two programs in C which decrypt the caesar and vigenere ciphers using kasiski analysis. I have inserted the source code and execution screenshots in the appendix of this document. For execution of the code, you should download the dictionary, hashdict and header files, which can be found at': , alongside with README instructions for compilation. Note: The programs operate correctly when entered ciphertext characters are encoded using UTF-8. If UTF-8 isn't used, you may encounter a bug.
- Caesar: Calculates the statistical distance of each possible shift with the ciphertext, and then decrypts it using the key with the smallest statistical distance. The proposed key is tested by decrypting the first 5 words of the ciphertext which are queried in a small dictionary file. If 3 or more words are not found in the dictionary, then the proposed key is discarded and the shift with the second least statistical distance is checked, otherwise it is printed alongside the decrypted plaintext.

- Vigenere: This program first finds the most common digraphs and trigraphs in the ciphertext, and calculates the distances between them as well as the gcd's of these distances, ignoring very small gcds. Then for each unique gcd (which corresponds to the length of the key) gcd statistical distances are calculated (first one containing characters 0, gcd, 2*gcd ... second one containing characters 1, 1 + gcd, 1 + 2 * gcd,... etc) and a key is proposed. Since the gcds often include values that are not the length of the key as well, each proposed key is tested by decrypting the first 5 words of the ciphertext which are queried in a small dictionary file. If 3 or more words are not found in the dictionary, then the proposed key is discarded, otherwise it is printed alongside the decrypted plaintext. I used an external library for the implementation of hash table, which can be found at: `https://github.com/exebook/hashdict.c`.

For example, in the execution of the program with the ciphertext of (1b), the most common digraph is ee which appears 9 times with distances 256, 71, 99, 11, 71, 1, 103, 34. The GCDs of these distances that are >= 3 are 11,71.
- For length 11 the proposed key is "oppenheimer", with 4 of the first 5 words of the decrypted text existing in the dictionary ("thy" was not found since it contains common words), and the key is therefore considered valid.
- For length 71 the proposed key is "raznaleeedisbqlznaertcppaenxwxzebutwixhpxelkleezstrltcfpwwgjrvnjtwbjfaw", with 0 of the first 5 words of the decrypted text existing in the dictionary, and the key is therefore considered invalid and is discarded.
Note: The amount of words validated as well as the required success percentage to determine validity can be adjusted through the header file.

Exercise 2

A

$$x \equiv 13^{\frac{1}{7}} mod 119 \Rightarrow x^7 \equiv 13 mod 119 \quad (1)$$

which can be re-written as $x^7 = 119 * k + 13 = 7 * (k') + 13 = 7 * (k' + 1) + 6 \quad (2)$, $x^7 = 119 * k + 13 = 17 * k'' + 13 \quad (3)$. Therefore we can rewrite (1) as a system of two linear equations using (2), (3): $x^7 \equiv 6 \bmod 7$ (4), and $x^7 \equiv 13 \bmod 17$ (5)
Here we test the possible values of x, excluding 0 and 1 since they will be congruent to 0 and 1 respectively regardless of their exponent. I will not present all the tested values for brevity, only the solutions.
(4) has a solution of $x \equiv 6 \bmod 7 \Rightarrow x = 7 * k + 6$ (6), since:
- $6^2 \equiv 1 \bmod 7$
- $6^3 \equiv 6 \bmod 7$
- $6^4 \equiv 1 \bmod 7$
- $6^7 \equiv 6 \bmod 7$ (calculated by multiplying the $6^3, 6^4$ congruences).
(5) has a solution of $x \equiv 4 \bmod 17$ (7), since
- $4^2 \equiv 16 \bmod 17$

- $4^3 \equiv 13 \bmod 17$
- $4^4 \equiv 1 \bmod 17$
- $4^7 \equiv 13 \bmod 7$ (calculated by multiplying the $4^3, 4^4$ congruences).

Using (6) and properties of modulo arithmetic we can rewrite (7) as:

$$7 * k + 6 \equiv 4 mod 17 \Rightarrow 7 * k \equiv -2 mod 17 \Rightarrow 7 * k \equiv 15 mod 17$$

This is a simple linear equation with a = 7, b = 15, n = 17. We first calculate gcd(a,n) = gcd(7,17) = 1, from which we conclude that the system has only one solution, which is $k \equiv a^{-1} * b \bmod n$ (8). To find a^{-1} we use the extended euclidian algorithm:

$$17 = 7 * 2 + 3$$

$$7 = 3 * 2 + 1$$

$$1 = 7 - 3 * 2 = 7 - (17 - 7 * 2) * 2 = 7 * 5 - 17 * 2 \Rightarrow$$

$7 * 5 = 17 * 2 + 1 \Rightarrow 7 * 5 \equiv 1 \bmod 17$, therefore $a^{-1} = 5$ (9).
Using (9) we can rewrite (8) as: $k \equiv 5 * 15 \bmod 17 \Rightarrow k \equiv 7 \bmod 17 \Rightarrow k = 17 * l + 7$ (10).
Using (10) x can be rewritten as $x = 7 * (17 * l + 7) + 6 = 119 * l + 49 + 6 = 119 * l + 55$, therefore the solution is $x \equiv 55 \bmod 119$.

B

From the given elliptic curve we can extract a = -2, b = 2, p = 71.
When a point is doubled, $\lambda = \frac{3x_1^2 + a}{2 * y_1} \bmod p$. So for P = (1,1) $\lambda \equiv \frac{3-2}{2} = 2^{-1} \bmod 71$.
Using the extended euclidian algorithm:

$$71 = 2 * 35 + 1$$

$$1 = 1 * 71 + (-35) * 2 \Rightarrow 1 = -1 * 71 + 2 * 71 + (-35) * 2 = -1 * 71 + 36 * 2$$

so we can conclude that $2 * 36 \equiv 1 \bmod 71 \Rightarrow \lambda = 36$.
Using the formulas for point addition:
$2P = (36^2 - 1 - 1 \bmod 71\ , 36(1 - x_3) - 1 \bmod 71) = (1294 \bmod 71\ , 35 - 36 * x_3 \bmod 71) = (16, 35 - 36 * 16 \bmod 71\ ) = (16, 27)$.
When two distinct points are added, $\lambda = \frac{y_2 - y_1}{x_2 - x_1} \bmod p$. So for P = (1,1) and Q = (4,49) $\lambda = \frac{49-1}{4-1} = 16 \bmod 71 = 16$.
Using the formulas for point addition:
$P + Q = (16^2 - 1 - 4 \bmod 71\ , 16 * (1 - x_3) - 1 \bmod 71\ ) = (252 \bmod 71\ , 15 - 16 * x_3 \bmod 71\ ) = (38, 15 - 16 * 38 \bmod 71\ ) = (38, 46)$.
We can ensure that the new points belong in the elliptic curve by verifying through the formula of the curve:
- $27^2 \equiv 19 \bmod 71$
- $16^3 - 2 * 16 + 2 \equiv 19 \bmod 71$

$\Rightarrow 27^2 \equiv 16^3 - 2 * 16 + 2 \mod 71$. so 2P belongs to the curve.
- $46^2 \equiv 57 \mod 71$
- $38^3 - 2 * 38 + 2 \equiv 57 \mod 71$
$\Rightarrow 46^2 \equiv 38^3 - 2 * 38 + 2 \mod 71$. so P + Q belongs to the curve.

Exercise 3

Note: All probabilities were rounded to 4 decimal digits.

A

In order to calculate $P(M = m|C = c)$, for every plaintext m and ciphertext c, we first need to calculate P(C = c) for every c and $P(C = c|M = m)$ for all c,m. First we calculate $P(C = c|M = m)$:

- $P(C = 1|M = a) = P(K = k_4) = \frac{3}{10}$
- $P(C = 2|M = a) = P(K = k_2) = \frac{3}{10}$
- $P(C = 3|M = a) = P(K = k_1) = \frac{1}{5}$
- $P(C = 4|M = a) = P(K = k_3) = \frac{1}{5}$
- $P(C = 1|M = b) = P(K = k_1) = \frac{1}{5}$
- $P(C = 2|M = b) = P(K = k_3) = \frac{1}{5}$
- $P(C = 3|M = b) = P(K = k_4) = \frac{3}{10}$
- $P(C = 4|M = b) = P(K = k_2) = \frac{3}{10}$
- $P(C = 1|M = c) = P(K = k_2) = \frac{3}{10}$
- $P(C = 2|M = c) = P(K = k_4) = \frac{3}{10}$
- $P(C = 3|M = c) = P(K = k_3) = \frac{1}{5}$
- $P(C = 4|M = c) = P(K = k_1) = \frac{1}{5}$
- $P(C = 1|M = d) = P(K = k_3) = \frac{1}{5}$
- $P(C = 2|M = d) = P(K = k_1) = \frac{1}{5}$
- $P(C = 3|M = d) = P(K = k_2) = \frac{3}{10}$
- $P(C = 4|M = d) = P(K = k_4) = \frac{3}{10}$

Then we calculate P(C = c) for all c.

- $P(C = 1) = \sum_{m \in M} P(C = 1|M = m) * P(M = m) = \frac{1}{10} + \frac{4}{75} + \frac{3}{50} + \frac{1}{25} = \frac{19}{75}$
- $P(C = 2) = \sum_{m \in M} P(C = 2|M = m) * P(M = m) = \frac{1}{10} + \frac{4}{75} + \frac{3}{50} + \frac{1}{25} = \frac{19}{75}$
- $P(C = 3) = \sum_{m \in M} P(C = 3|M = m) * P(M = m) = \frac{1}{15} + \frac{6}{75} + \frac{1}{25} + \frac{3}{50} = \frac{18,5}{75}$
- $P(C = 4) = \sum_{m \in M} P(C = 4|M = m) * P(M = m) = \frac{1}{15} + \frac{6}{75} + \frac{1}{25} + \frac{3}{50} = \frac{18,5}{75}$

Finally we can calculate $P(M = m|C = c)$ since we know that $P(M = m|C = c) = \frac{P(M=m)*P(C=c|M=m)}{P(C=c)}$.

- $P(M = a|C = 1) = \frac{1}{3} * \frac{3}{10} * \frac{75}{19} = \frac{15}{38} = 0.3947$
- $P(M = a|C = 2) = \frac{1}{3} * \frac{3}{10} * \frac{75}{19} = \frac{15}{38} = 0.3947$
- $P(M = a|C = 3) = \frac{1}{3} * \frac{1}{5} * \frac{75}{18,5} = \frac{10}{37} = 0.2708$
- $P(M = a|C = 4) = \frac{1}{3} * \frac{1}{5} * \frac{75}{18,5} = \frac{10}{37} = 0.2708$
- $P(M = b|C = 1) = \frac{4}{15} * \frac{1}{5} * \frac{75}{19} = \frac{4}{19} = 0.2105$
- $P(M = b|C = 2) = \frac{4}{15} * \frac{1}{5} * \frac{75}{19} = \frac{4}{19} = 0.2105$

- $P(M = b|C = 3) = \frac{4}{15} * \frac{3}{10} * \frac{75}{18,5} = \frac{12}{37} = 0.3243$
- $P(M = b|C = 4) = \frac{4}{15} * \frac{3}{10} * \frac{75}{18,5} = \frac{12}{37} = 0.3243$
- $P(M = c|C = 1) = \frac{1}{5} * \frac{3}{10} * \frac{75}{19} = \frac{9}{38} = 0.2368$
- $P(M = c|C = 2) = \frac{1}{5} * \frac{3}{10} * \frac{75}{19} = \frac{9}{38} = 0.2368$
- $P(M = c|C = 3) = \frac{1}{5} * \frac{1}{5} * \frac{75}{18,5} = \frac{6}{37} = 0.1622$
- $P(M = c|C = 4) = \frac{1}{5} * \frac{1}{5} * \frac{75}{18,5} = \frac{6}{37} = 0.1622$
- $P(M = d|C = 1) = \frac{1}{5} * \frac{1}{5} * \frac{75}{19} = \frac{3}{19} = 0.1579$
- $P(M = d|C = 2) = \frac{1}{5} * \frac{1}{5} * \frac{75}{19} = \frac{3}{19} = 0.1579$
- $P(M = d|C = 3) = \frac{1}{5} * \frac{3}{10} * \frac{75}{18,5} = \frac{9}{37} = 0.2432$
- $P(M = d|C = 4) = \frac{1}{5} * \frac{3}{10} * \frac{75}{18,5} = \frac{9}{37} = 0.2432$

B

Note: The formulas used in this exercise were found in chapter 9.3.2 of the textbook "Cryptography made simple".
In order to calculate the entropy value for a message given the ciphertext, we first calculate the entropy value for a message given a specific ciphertext $H(M|C = c)$.
Then, $H(M|C) = \sum_{c \in C} P(C = c) * H(M|C = c)$.
From the definition of entropy,
$H(M|C = c) = -\sum_{m \in M} P(M = m|C = c) * log_2 P(M = m|C = c)$.

- $H(M|C = 1) = -(0.3947 * -1.3412 + 0.2105 * -2.248 + 0.2368 * -2.0783 + 0.1579 * -2.663) = -(-1.9152) = 1.9152$
- $H(M|C = 2) = -(0.3947 * -1.3412 + 0.2105 * -2.248 + 0.2368 * -2.0783 + 0.1579 * -2.663) = -(-1.9152) = 1.9152$
- $H(M|C = 3) = -(0.2708 * -1.8847 + 0.3243 * -1.6246 + 0.1622 * -2.624 + 0.2432 * -2.0398) = -(-1.959) = 1.959$
- $H(M|C = 4) = -(0.2708 * -1.8847 + 0.3243 * -1.6246 + 0.1622 * -2.624 + 0.2432 * -2.0398) = -(-1.959) = 1.959$

From the above entropy values we can conclude that if an attacker eavesdrops a cipherkey 1 or 2, they are slightly less uncertain about the latent plaintext than if they eavesdropped 3 or 4.
$H(M|C) = 0.2533 * 1.9152 + 0.2533 * 1.9152 + 0.2467 * 1.959 + 0.2467 * 1.959 = 1.9368$
The above entropy value makes logical sense, since eavesdropping ciphertext causes slightly less uncertainty when compared to the worst case scenario (which would be random guessing with probability of success $\frac{1}{\#M} = \frac{1}{4} \Rightarrow H_{random} = 2$.

C

The scheme is not perfectly secure, since there exists at least one m, c such that $p(M = m|C = c) \neq p(M = m)$. For example for M = a and C = 1, $p(M = a|C = 1) = 0.3947 \neq p(M = a) = 0.333...$

Appendix

Helper function file:

```
#include <stdio.h>
#include <stdbool.h>
#include "helper.h"

const float default_frequency[26] = {8.2, 1.5, 2.8, 4.2, 12.7, 2.2, 2.0, 6.1, 7.0, 0.1, 0.8, 4.0, 2.4, 6.7, 7.5, 1.9, 0.1, 6.0,
      6.3, 9.0, 2.8, 1.0, 2.4, 0.1, 2.0, 0.1};

    // Convert a string to lowercase
    char *
    strlwr(char *str)
{
    unsigned char *p = (unsigned char *)str;

    while (*p)
    {
        *p = tolower((unsigned char)*p); // Convert each character to lowercase
        p++;
    }

    return str;
}

// Extract the first k words from a given text
char **extract_first_k_words(const char *text, int k)
{
    if (k <= 0)
        return NULL; // Return NULL if k is invalid

    // Allocate memory for the word list
    char **word_list = malloc(k * sizeof(char *));
    if (!word_list)
    {
        fprintf(stderr, "Memory allocation failed\n");
        return NULL;
    }

    int word_count = 0;               // Count of words extracted
    size_t word_len = 0;              // Length of the current word
    char word[MAX_WORD_LENGTH + 1] = {0}; // Buffer for the current word

    // Iterate over the text to extract words
    for (const char *p = text; *p != '\0'; p++)
    {
        // Check for delimiters (whitespace, punctuation)
        if (isspace(*p) || *p == ',' || *p == '?' || *p == '!' || *p == '-' || *p == '\'' || *p == ';')
        {
            if (word_len > 0) // If a word is found
            {
                word[word_len] = '\0';            // Null-terminate the word
                word_list[word_count] = strdup(word); // Duplicate the word into the list
                if (!word_list[word_count])
                {
                    fprintf(stderr, "Memory allocation failed for word\n");
                    for (int i = 0; i < word_count; i++)
                        free(word_list[i]); // Free previously allocated words
                    free(word_list);
                    return NULL;
                }
                word_count++;
                if (word_count == k) // Stop if k words are extracted
                    break;
                word_len = 0; // Reset word length for the next word
            }
        }
        else
        {
```

```
            if (word_len < MAX_WORD_LENGTH) // Append character if within word length
            {
                word[word_len++] = tolower(*p);
            }
        }
    }

    // Add the last word if it exists
    if (word_len > 0 && word_count < k)
    {
        word[word_len] = '\0';
        word_list[word_count++] = strdup(word);
    }

    // Fill remaining slots with NULL if fewer than k words are found
    for (int i = word_count; i < k; i++)
    {
        word_list[i] = NULL;
    }

    return word_list;
}

// Function to validate words against a dictionary
int validate_words_with_dictionary(char **words, int k)
{
    int counter = 0; // Count of valid words

    for (int j = 0; j < k; j++)
    {
        if (!words[j])
            continue; // Skip NULL entries

        FILE *cmd;
        char result[100];
        char command[150] = "grep -i -w "; // Grep command to search for words
        strcat(command, words[j]);
        strcat(command, " ./dict.txt"); // Search in the dictionary file

        cmd = popen(command, "r"); // Execute the command
        if (cmd == NULL)
        {
            perror("popen"); // Handle error in opening the process
            exit(EXIT_FAILURE);
        }

        bool found = false;
        // Read results from the command output
        while (fgets(result, sizeof(result), cmd))
        {
            result[strcspn(result, "\n")] = '\0';  // Remove newline character
            if (strcmp(strlwr(result), words[j]) == 0) // Compare words case-insensitively
            {
                found = true;
                break;
            }
        }

        if (found)
        {
            counter++; // Increment valid word count
        }
        pclose(cmd); // Close the process
    }

    return counter; // Return the number of valid words
}
```

Source code for Caesar Cipher decryption:

```
#include "helper.h"

extern float default_frequency[26];

// Structure to hold statistical distance and corresponding shift
struct sd
{
    float statistical_distance; // Distance metric
    int shift;                  // Shift value
} typedef sd;
```

```c
// Function to swap two floats
void swap(float *a, float *b)
{
    float t = *a;
    *a = *b;
    *b = t;
}

// Function to swap two integers
void swap_indx(int *a, int *b)
{
    int t = *a;
    *a = *b;
    *b = t;
}

// Partitions the array for quicksort
float partition(sd arr[], int low, int high)
{
    int pivot = arr[high].statistical_distance; // Pivot value
    int i = (low - 1);                        // Smaller element index

    for (int j = low; j <= high - 1; j++)
    {
        // Check if current element is smaller than pivot
        if (arr[j].statistical_distance < pivot)
        {
            i++; // Increment smaller element index
            swap(&arr[i].statistical_distance, &arr[j].statistical_distance);
            swap_indx(&arr[i].shift, &arr[j].shift);
        }
    }
    // Place pivot in correct position
    swap(&arr[i + 1].statistical_distance, &arr[high].statistical_distance);
    swap_indx(&arr[i + 1].shift, &arr[high].shift);

    return (i + 1); // Return pivot index
}

// QuickSort implementation
void quickSort(sd arr[], int low, int high)
{
    if (low < high)
    {
        // Partition the array
        int pi = partition(arr, low, high);

        // Sort elements before and after partition
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}

int main()
{
    char *arr;
    arr = malloc(MAXC * sizeof(char)); // Allocate memory for input text
    if (arr == NULL)
    {
        printf("Memory error\n");
        return 1;
    }

    // Provide the ciphertext here
    strcpy(arr, "Yt gj, tw sty yt gj, ymfy nx ymj vzjxynts: Bmjymjw 'ynx stgqjw ns ymj rnsi yt xzkkjw Ymj xqnslx fsi fwwtbx tk
        tzywfljtzx ktwyzsj, Tw yt yfpj fwrx flfnsxy f xjf tk ywtzgqjx Fsi gd tuutxnsl jsi ymjr. Yt inj - yt xqjju, St rtwj; fsi
        gd f xqjju yt xfd bj jsi Ymj mjfwy-fhmj fsi ymj ymtzxfsi sfyzwfq xmthpx Ymfy kqjxm nx mjnw yt: 'ynx f htsxzrrfynts
        Ijatzyqd yt gj bnxm'i"); // Example ciphertext

    char *arr_clean;
    arr_clean = malloc(MAXC * sizeof(char)); // Allocate memory for cleaned text
    if (arr_clean == NULL)
    {
        printf("Memory error\n");
        return 1;
    }

    // Clean the text to include only alphabetic characters
    for (int i = 0, j = 0; i < strlen(arr); i++)
    {
        if (arr[i] >= 'A' && arr[i] <= 'Z')
        {
            arr_clean[j++] = tolower(arr[i]); // Convert to lowercase
```

```
    }
    else if (arr[i] >= 'a' && arr[i] <= 'z')
    {
        arr_clean[j++] = arr[i];
    }
    else if (arr[i] == '\0')
    {
        arr_clean[j++] = arr[i]; // Add null terminator
        break;
    }
}
printf("Clean string is: \"%s\"\n", arr_clean);
float cipher_frequency[26]; // Cipher text frequency
sd statistical_distance[26]; // Array to store statistical distances

// Initialize statistical distance and cipher frequency
for (int i = 0; i < 26; i++)
{
    cipher_frequency[i] = 0.0;
    statistical_distance[i].statistical_distance = 0.0;
    statistical_distance[i].shift = i;
}

// Count letter frequencies in the cleaned text
for (int i = 0; i < strlen(arr_clean); i++)
{
    cipher_frequency[arr_clean[i] - 'a'] += 1.0;
}

// Calculate statistical distances for each shift
for (int i = 0; i < 26; i++)
{
    for (int j = 0; j < 26; j++)
    {
        statistical_distance[i].statistical_distance += 0.5 * fabs(default_frequency[j] - cipher_frequency[(j + i) % 26]);
    }
}

// Find minimum statistical distance and corresponding key
float min = statistical_distance[0].statistical_distance;
int key = 0;
for (int i = 0; i < 26; i++)
{
    cipher_frequency[i] /= (float)strlen(arr_clean) / 100;
    if (statistical_distance[i].statistical_distance < min)
    {
        min = statistical_distance[i].statistical_distance;
    }
}

// Sort statistical distances to prioritize smaller distances
quickSort(statistical_distance, 0, 25);

int cnt = 0; // Counter for possible keys
while (cnt < 26)
{
    char *decrypted_plaintext = malloc(MAXC * sizeof(char)); // Allocate memory for plaintext
    key = statistical_distance[cnt].shift;              // Get current key

    // Decrypt the text using the current key
    for (int i = 0; i < strlen(arr); i++)
    {
        if (arr[i] >= 'A' && arr[i] <= 'Z')
        {
            decrypted_plaintext[i] = toupper((tolower(arr[i]) - 'a' + 26 - key) % 26 + 'a');
        }
        else if (arr[i] >= 'a' && arr[i] <= 'z')
        {
            decrypted_plaintext[i] = (arr[i] - 'a' + 26 - key) % 26 + 'a';
        }
        else
        {
            decrypted_plaintext[i] = arr[i];
        }
    }

    // Extract words and validate using a dictionary
    char **words = extract_first_k_words(decrypted_plaintext, CHECKED_WORDS);
    printf("\nTesting Key = %d, with statistical distance %.3f. Extracted Words were: ", key,
           statistical_distance[cnt].statistical_distance);
    for (int i = 0; i < CHECKED_WORDS && words[i] != NULL; i++)
    {
        printf("%d: %s, ", i + 1, words[i]);
```

```
        }

        // Check validity of the decrypted text
        int valid = validate_words_with_dictionary(words, 5);
        float ratio = (float)valid / CHECKED_WORDS;

        // If valid, print plaintext and stop executing
        if (ratio >= (float)PERCENTAGE_REQ / 100)
        {
            printf("\nKey = %d is valid.\nPlain text is: %s\n", key, decrypted_plaintext);
            free(decrypted_plaintext);
            break;
        }
        else
        {
            printf("\nKey = %d produces non-words %.4f, discarded.\n", key, ratio);
            free(decrypted_plaintext);
        }
        cnt++;
    }

    // Free allocated memory
    free(arr);
    free(arr_clean);

    return 0;
}
```

Execution for (1a) ciphertext:

```
dimitris@dimitris-MS-7E12:~/cs4520$ ./caesar
Clean string is: "ytgjtwstyytgjymfynxymjvzjxyntsbmjymjwynxstgqjwnsymjrnsiytxzkkjwymjx
qnslxfsifwwtbxtktzywfljtzxktwyzsjtwytyfpjfwrxflfnsxyfxjftkywtzgqjxfsigdtuutxnsljsiymj
rytinjytxqjjustrtwjfsigdfxqjjuytxfdbjjsiymjmjfwyfhmjfsiymjymtzxfsisfyzwfqxmthpxymfykq
jxmnxmjnwytynxfhtsxzrrfyntsijatzyqdytgjbnxmi"

Testing Key = 5, with statistical distance 90.850. Extracted Words were: 1: to, 2: be
, 3: or, 4: not, 5: to,
Key = 5 is valid.
Plain text is: To be, or not to be, that is the question: Whether 'tis nobler in the
mind to suffer The slings and arrows of outrageous fortune, Or to take arms against a
 sea of troubles And by opposing end them. To die—to sleep, No more; and by a sleep t
o say we end The heart-ache and the thousand natural shocks That flesh is heir to: 't
is a consummation Devoutly to be wish'd
dimitris@dimitris-MS-7E12:~/cs4520$
```

Execution for another ciphertext:

```
dimitris@dimitris-MS-7E12:~/cs4520$ ./caesar
Clean string is: "qlabcbkaqeblmmobppbaxdxfkpqqebfolmmobpplopqlmibxaqebzxrpblcqebtbxhxdxfkpqqebpqolkd
telbumilfqxkazorpeqebjqefpfpqebarqvlcxiiebxoqpqexqexsbklqybbkpmlfibayvbdlfpjxkazloormqflkfqfpplptbbq
qlabslqblkbpbicqllkbpcbiiltpqexqfalklqhklteltqebobzxkybpljxkvrkcloqrkxqbppqfiitfqelrqprmmloqloabcbka
bopxpclojbjvifcbpqxphtfiiybqlebimqelpbtelprccboxkaqlmroprbqeolrdejvxsbkdfkdpmbbzeqelpbtelqxhbmibxpro
bfkqebmxfklclqebopeltexmmvftfiiybfcjvcbbyibbccloqpxobzoltkbatfqeprzzbppxkafcxqqebmofzblcjvabslqflkxk
apxzofcfzbpjvobmrqxqflkfpklqqxokfpebayvqebzofjbplcqeblmmobpplopftfiicfdeq"

Testing Key = 23, with statistical distance 227.850. Extracted Words were: 1: to, 2: defend, 3: the,
 4: oppressed, 5: against,
Key = 23 is valid.
Plain text is: To defend the oppressed against their oppressors, to plead the cause of the weak agai
nst the strong who exploit and crush them, this is the duty of all hearts that have not been spoiled
 by egoism and corruption… It is so sweet to devote oneself to one's fellows that I do not know how
there can be so many unfortunates still without support or defenders. As for me, my life's task will
 be to help those who suffer and to pursue through my avenging speech those who take pleasure in the
 pain of others. How happy I will be if my feeble efforts are crowned with success and if, at the pr
ice of my devotion and sacrifices, my reputation is not tarnished by the crimes of the oppressors I
will fight.
dimitris@dimitris-MS-7E12:~/cs4520$
```

Source code for Vigenere Cipher decryption:

```c
#include "helper.h"
#include "hashdict.h"

extern float default_frequency[26];
int gcd (int a, int b)
{
   int r; // remainder
   while (b > 0)
   {
      r = a % b;
      a = b;
      b = r;
   }

   return a;
}

void clean_string(const char *source, char * dest) {

    for (int i=0, j = 0; i< strlen(source); i++) {
        if(source[i] >= 'A' && source[i] <= 'Z') {
            dest[j++] = tolower(source[i]);
        }
        else if (source[i] >= 'a' && source[i] <= 'z') {
            dest[j++] = source[i];
        }
        else if (source[i] == '\0') {
            dest[j++] = source[i];
            break;
        }
    }
    return;
}

int extract_ngraph_max(struct dictionary *dict, const int n, const char *ciphertext)
{ // Finds the most frequent ngraphs in the cleaned ciphertext, storing their maximum frequencies in the hash dictionary.
    int nmax = 1;
    for (int i = 0; i< strlen(ciphertext) - 1; i++){
        char ngraph[n+1];
        for (int j = 0; j < n; j++){
            ngraph[j] = ciphertext[i+j];
        }
        ngraph[n] = '\0';

        if(dic_add(dict,ngraph,n) == false) {
            *dict->value = 1;
        }
        else {
            *dict->value += 1;
            if (*dict->value > nmax) {
```

```
                nmax = *dict->value;
            }
        }
        if(n == 2 && ngraph[0] == ngraph[1] && ngraph[1] != ciphertext[i+2]){
            i++;
            continue;

        }
    }
    return nmax;
}

void generate_next_permutation(char *string)
{ // Generates the next lexicographical permutation of a given string.
    for (int i = strlen(string) - 1; i >= 0; i--){
        if (string[i] == 'z') {
            string[i] = 'a'; // Wrap around to 'a'
        } else {
            string[i]++; // Increment the character
            return;  // Stop once the increment is successful
        }
    }
}

void extract_ngraph_list_dist (struct dictionary *dict, const int n, const char *ciphertext, const int ngraph_max, char **
      ngraph_list, int ** ngraph_dist, int*nummax, int*maxcnt){
    // Creates a list of the most frequent n-graphs and calculates the distances between their occurrences, storing them for GCD
        calculations.
    int counter;
    int counter2;
    counter = 0;
    char ngraph[n+1];
    char term_ngraph[n+1];
    for(int i = 0; i < n; i++){
        ngraph[i] = 'a';
        term_ngraph[i] = 'z';
    }
    ngraph[n] = '\0';
    term_ngraph[n] = '\0';
    while (strcmp(ngraph,term_ngraph) != 0) {
        if(dic_find(dict,ngraph,n) == true && ngraph_max == *dict->value) {
            strcpy(ngraph_list[(counter)++],ngraph);
        }
        if (counter == MAXIMUM_NGRAPHS)
        {
            break;
        }
        generate_next_permutation(ngraph);
    }
    counter2 = 0; // maximum occurences of one of the most common n_graphs
    for (int i = 0; i < counter; i++){
        printf("Distances between instances of %d-graph %s are: ", n, ngraph_list[i]);
        int ppos = -1;
        int cnt = 0;
        for (int j = 0; j < strlen(ciphertext) - (n-1); j++){

            char ngraph[n+1];
            for (int k = 0; k < n; k++){
                ngraph[k] = ciphertext[j+k];
            }
            ngraph[n] = '\0';
            if(strcmp(ngraph,ngraph_list[i]) == 0){
                if(ppos == -1){
                    ngraph_dist[i][cnt++] = 0;
                }
                else{
                    ngraph_dist[i][cnt++] = j - ppos;
                }
                ppos = j;
            }
        }
        for (int j = 0; j < cnt; j++)
        {
            printf("%d", ngraph_dist[i][j]);
            if (j < cnt - 1)
                printf(", "); // Add a comma if not the last element
        }

        if(cnt > counter2){
            counter2 = cnt;
        }

        printf("\n");
```

```
    }
    *nummax = counter;
    *maxcnt = counter2;

    return;
}

char* decrypt_ciphertext(const char *ciphertext, const char *key, int key_length) {
    // Allocate memory for the decrypted plaintext
    size_t text_length = strlen(ciphertext);
    char *decrypted_plaintext = malloc((text_length + 1) * sizeof(char)); // +1 for null terminator
    if (!decrypted_plaintext) {
        fprintf(stderr, "Memory allocation failed\n");
        return NULL;
    }

    int cnt = 0; // Counter for key cycling

    for (size_t j = 0; j < text_length; j++) {
        int keyc = key[cnt % key_length] - 'a'; // Current key character as shift value

        if (isupper(ciphertext[j])) {
            // Handle uppercase letters
            decrypted_plaintext[j] = toupper((tolower(ciphertext[j]) - 'a' + 26 - keyc) % 26 + 'a');
            cnt++; // Increment key counter for alphabetic characters
        } else if (islower(ciphertext[j])) {
            // Handle lowercase letters
            decrypted_plaintext[j] = (ciphertext[j] - 'a' + 26 - keyc) % 26 + 'a';
            cnt++; // Increment key counter for alphabetic characters
        } else {
            // Copy non-alphabetic characters as-is
            decrypted_plaintext[j] = ciphertext[j];
        }
    }

    // Null-terminate the decrypted plaintext
    decrypted_plaintext[text_length] = '\0';
    return decrypted_plaintext;
}

void calculate_key(const char *ciphertext_clean, int current_gcd, const float *default_frequency, char *key) {
    // Computes the decryption key by matching ciphertext frequency with a standard English letter frequency table through Kasiski
       analysis.

        // Loop through each segment of the ciphertext
        for (int j = 0; j < current_gcd; j++)
    {
        float cipher_frequency[26] = {0.0};
        float statistical_distance[26] = {0.0};
        int cnt = 0;

        // Calculate cipher frequencies for this segment
        for (int k = j; k < strlen(ciphertext_clean); k += current_gcd) {
            cipher_frequency[ciphertext_clean[k] - 'a'] += 1.0;
            cnt++;
        }

        // Normalize the frequencies to percentages
        for (int k = 0; k < 26; k++) {
            cipher_frequency[k] = (cipher_frequency[k] / cnt) * 100.0;
        }

        // Calculate the statistical distance for each possible shift
        for (int k = 0; k < 26; k++) {
            for (int l = 0; l < 26; l++) {
                statistical_distance[k] += 0.5 * fabs(default_frequency[l] - cipher_frequency[(l + k) % 26]);
            }
        }

        // Find the shift with the minimum statistical distance
        int best_shift = 0;
        float min_distance = statistical_distance[0];
        for (int k = 1; k < 26; k++) {
            if (statistical_distance[k] < min_distance) {
                min_distance = statistical_distance[k];
                best_shift = k;
            }
        }

        // Store the key character for this segment
        key[j] = (char)('a' + best_shift);
    }
```

```
    // Null-terminate the key
    key[current_gcd] = '\0';
}

void decrypt_plaintexts (int *number_of_max_digraphs, int **ngraph_max_gcds, const char * ciphertext_original, const char *
      ciphertext_clean) {

    int counter = 0;
    int * checked_gcds = malloc(100*sizeof(int));
    for(int i=0; i < *number_of_max_digraphs ; i++){
        const int number_of_gcds = ngraph_max_gcds[i][0];
        for (int l=1; l < 1 + number_of_gcds; l++){    // check that ngraph_max_gcds[i][l] hasn't been checked so far, if it has,
             skip.
            bool flag = false;
            const int current_gcd = ngraph_max_gcds[i][l];
            for(int j = 0; j < counter; j++){
                if(checked_gcds[j] == current_gcd){
                    flag = true;
                    break;
                }
            }
            if(flag == true){
                continue;
            }
            checked_gcds[counter++] = current_gcd;
            char key[current_gcd + 1];

            calculate_key(ciphertext_clean, current_gcd, default_frequency, key);

            char *decrypted_plaintext = decrypt_ciphertext(ciphertext_original, key,strlen(key));
            const int k = CHECKED_WORDS;
            char **words = extract_first_k_words(decrypted_plaintext, k);
            printf("Checking for key with length = %d. Extracted Words: ",current_gcd);
            for (int i = 0; i < k && words[i] != NULL; i++) {
                printf("%d: %s, ", i + 1, words[i]);
            }
            const int valid = validate_words_with_dictionary(words, k);

            float ratio = (float)valid / k;
            if (ratio < (float)PERCENTAGE_REQ / 100){
                printf(" => Discarded key \"%s\" (produced non-words)\n",key);
                continue;
            }
            printf("=> Key is \"%s\" (w/ success rate %.2f)\n\n",key,ratio);
            printf("Plaintext is %s\n\n",decrypted_plaintext);
        }

    }
    free(checked_gcds);
    printf("\n");
}

int main () {

    const char *ciphertext = "Ftkins Xpk Wvzu; llna eeryc Tdgq nyx Btsl? W ldvfomx Flvs! Wpzr tizoc, Xcs hycyiuq! Xywct mauiz
          Nizbv X ez meqz xf ycda; Goma Flp tdgxu-zxzqedwcv Pvmi jqazzstvf ti. Buqv ob X, pnfmvs hvgdaegl xpq affas, Qnki umrztthx
          bu iidxy hd hpnf qizozbs! Csg vrm aj rza ilrzi emviwdgw ehroqh wcg hxepjm Qwtoetxu kiifl; kvdj wuhpb mpfbt hyecmdq.
          Xysgtjbyi afeer je! avu jwd xymhtps yivaae, Qdcuhlv btc wcth, iaqsg flv ktppgo jqxpvr gteyt, Fg Yi kvtn eel etdirrn
          dzrygwyi, Ss iwsh alm aykkpgh phyaq, pvti-weakil arv. Rgdrn hrl Nljvbp eak Nikeufpiln, Rezze, rbs ppy alm axysg leeymwdw
          ysgt, Eel wtmme pn Bi. Qlwbdsp hwtq slezxijgan. Jvnlb! flfi hweya gzgwy hwn vvcete me hwt jvlpl. Eeexpne fhml: Temwcv
          lrhvl flvgt lsekw wr Ovgwpzn, oi ets nspgigo e lueusb, lmgo nwurvr eppzz, ucmozbv prq wvwexioixrt omueict, heexl eomme
          hd Zvv- zlvm, wkobbiepro imkv utee, jeafmeu sdaa oma rets.";

    struct dictionary *my_dict = dic_new(0);

    char* ciphertext_clean;
    ciphertext_clean = malloc((strlen(ciphertext) + 1) * sizeof(char));
    if (ciphertext_clean == NULL) {
        printf("Memory error\n");
        return 1;
    }
    clean_string(ciphertext,ciphertext_clean); //remove non letters, make everything lowercase

    printf("Clean string is: %s\n\n",ciphertext_clean);
    const int digraph_max = extract_ngraph_max(my_dict,2,ciphertext_clean);

    char **digraph_list;
    digraph_list = malloc(MAXIMUM_NGRAPHS * sizeof(char *));
    for (int i = 0; i < MAXIMUM_NGRAPHS; i++)
    {
        digraph_list[i] = malloc(3*sizeof(char));
    }
```

```
    int **digraph_dist;
    digraph_dist = malloc(MAXIMUM_NGRAPHS * sizeof(int *));
    for (int i = 0; i < MAXIMUM_NGRAPHS; i++)
    {
        digraph_dist[i] = malloc(digraph_max*sizeof(int));
    }
// array that contains distances between each instance of one of the most common digraphs

    int number_of_max_digraphs = 0;
    int maximum_occurences_of_max_digraphs = 0;

    extract_ngraph_list_dist(my_dict,2,ciphertext_clean,digraph_max,digraph_list,digraph_dist,&number_of_max_digraphs,&maximum_occurences_of_max_digraphs);

    int **d_max_gcds;
    d_max_gcds = malloc(number_of_max_digraphs*sizeof(int *));
    for (int i = 0; i < MAXIMUM_NGRAPHS; i++)
    {
        d_max_gcds[i] = malloc(maximum_occurences_of_max_digraphs*sizeof(int));
    }
    for (int i= 0; i< number_of_max_digraphs; i++){ // find all gcd of distances between each common digraph, requiring that
       distance >= 3
        int cnt = 0;
        printf("GCDs of distances between instances of 2-graph %s are:", digraph_list[i]);
        for (int j = 1; j < maximum_occurences_of_max_digraphs; j++)
        {
            for(int k = j + 1; k < maximum_occurences_of_max_digraphs; k++){
                int pgcd = gcd(digraph_dist[i][j],digraph_dist[i][k]);
                bool flag = false;
                for (int l = 1; l < 1+cnt; l++){
                    if(d_max_gcds[i][l] == pgcd){
                        flag = true;
                        break;
                    }
                }
                if(pgcd >= 3 && flag == false) {
                    d_max_gcds[i][1+cnt++] = pgcd;
                }
            }
        }
        d_max_gcds[i][0] = cnt;
        printf(" Total: %d GCD(s), Values: [", cnt);
        for (int j = 1; j < 1 + cnt; j++)
        {
            printf("%d", d_max_gcds[i][j]);
            if (j < cnt)
                printf(", ");
        }
        printf("]\n\n");
    }

    decrypt_plaintexts(&number_of_max_digraphs,d_max_gcds,ciphertext,ciphertext_clean);

    const int trigraph_max = extract_ngraph_max(my_dict,3,ciphertext_clean);
    char **trigraph_list;
    trigraph_list = malloc(MAXIMUM_NGRAPHS * sizeof(char *));
    for (int i = 0; i < MAXIMUM_NGRAPHS; i++)
    {
        trigraph_list[i] = malloc(4*sizeof(char));
    }
    int **trigraph_dist;
    trigraph_dist = malloc(MAXIMUM_NGRAPHS * sizeof(char *));
    for (int i = 0; i < MAXIMUM_NGRAPHS; i++)
    {
        trigraph_dist[i] = malloc(trigraph_max*sizeof(char));
    }

    int number_of_max_trigraphs = 0;
    int maximum_occurences_of_max_trigraphs = 0;
    extract_ngraph_list_dist(my_dict,3,ciphertext_clean,trigraph_max,trigraph_list,trigraph_dist,&number_of_max_trigraphs,&maximum_occurences_of_max_trigraphs);
    int **t_max_gcds;
    t_max_gcds = malloc(number_of_max_trigraphs *sizeof(int *));
    for(int i=0; i<number_of_max_trigraphs; i++){
        t_max_gcds[i] = malloc(maximum_occurences_of_max_trigraphs*sizeof(char));
    }
    for (int i= 0; i< number_of_max_trigraphs; i++){
        int cnt = 0;
        printf("GCDs of distances between instances of 3-graph %s are:", trigraph_list[i]);
        for (int j = 1; j < maximum_occurences_of_max_trigraphs; j++)
        {
            for(int k = j + 1; k < maximum_occurences_of_max_trigraphs; k++){
                int pgcd = gcd(trigraph_dist[i][j],trigraph_dist[i][k]);
                bool flag = false;
                for (int l = 1; l < 1+cnt; l++){
```

```
                if(t_max_gcds[i][1] == pgcd){
                    flag = true;
                    break;
                }
            }
            if(pgcd >= 3 && flag == false) {
                t_max_gcds[i][1+cnt++] = pgcd;
            }
        }
    }
    t_max_gcds[i][0] = cnt;
    printf(" Total: %d GCD(s), Values: [", cnt);
    for (int j = 1; j < 1 + cnt; j++)
    {
        printf("%d", t_max_gcds[i][j]);
        if (j < cnt)
            printf(", ");
    }
    printf("]\n\n");
}
decrypt_plaintexts(&number_of_max_trigraphs,t_max_gcds,ciphertext,ciphertext_clean);
dic_delete(my_dict);
return 0;
}
```

Execution for (1b) ciphertext:

```
dimitris@dimitris-MS-7E12:~/cs4520$ ./vigenere
Clean string is: ftkinsxpkwvzullnaeeryctdgqnyxbtslwldvfomxflvswpzrtizocxcshycyiuqxywctmauiznizbvxezmeqzxfycdagomafl
ptdgxuzxzqedwcvpvmijqazzstvftibuqvobxpnfmvshvgdaeglxpqaffasqnkiumrztthxbuiidxyhdhpnfqizozbscsgvrmajrzailrziemviwdgwehroqhwcghxepjmqwtoetxukiiflkvdjwuhpbmpfbthyecmdqxysgtjbyiafeerjeavujwdxymhtpsyivaaeqdcuhlvbtcwcthiaqsgflvktppgojqxpvrgteytfgyikvtneeletdirrndzrygwyissiwshalmaykkpghphyaqpvtiweakilarvrgdrnhrlnljvbpeaknikeufpilnrezzerbsppyalmaxysgleeymwdwysgteelwtmmepnbiqlwbdsphwtqslezxijganjvnlbflfihweyagzgwyhwnvvcetemehwtjvlpleeexpnefhmltemwcvlrhvlflvgtlsekwwrovgwpznoietsnspgigoeluеusblmgonwurvreppzzucmozbvprqwvwexioixrtomueictheexleommehdzvvzlvmwkobbieproimkvuteejeafmeusdaaomarets

Distances between instances of 2-graph ee are: 0, 256, 71, 99, 11, 71, 1, 103, 34
GCDs of distances between instances of 2-graph ee are:  Total: 2 GCD(s),  Values: [71, 11]

Checking for key with length = 71. Extracted Words: 1: otlvnh, 2: tlg, 3: tnht, 4: vaon, 5: eaafa,  => Discarded key "raznaleeedisbqlznaertcppaenxwxzebutwixhpxelkleezstrltcfpwwgjrvnjtwbjfaw"
 (produced non-words)
Checking for key with length = 11. Extracted Words: 1: reveal, 2: thy, 3: self, 4: what, 5: awful, => Key is "oppenheimer" (w/ success rate 0.80)

Plaintext is Reveal Thy Self; what awful Form art Thou? I worship Thee! Have mercy, God supreme! Thine inner Being I am fain to know; This Thy forth-streaming Life bewilders me. Time am I, laying desolate the world, Made manifest on earth to slay mankind! Not one of all these warriors ranged for strife Escapeth death; thou shalt alone survive. Therefore stand up! win for thyself renown, Conquer thy foes, enjoy the wealth filled realm, By Me they are already overcome, Be thou the outward cause, left-handed one. Drona and Bhishma and Jayadratha, Karna, and all the other warriors here, Are slain by Me. Destroy them fearlessly. Fight! thou shalt crush thy rivals in the field. Sanjaya said: Having heard these words of Keshava, he who weareth a diadem, with joined palms, quaking and prostrating himself, spake again to Kri- shna, stammering with fear, casting down his face.


Distances between instances of 3-graph flv are: 0, 275, 231
Distances between instances of 3-graph ysg are: 0, 176, 11
GCDs of distances between instances of 3-graph flv are:  Total: 1 GCD(s),  Values: [11]

GCDs of distances between instances of 3-graph ysg are:  Total: 1 GCD(s),  Values: [11]

Checking for key with length = 11. Extracted Words: 1: reveal, 2: thy, 3: self, 4: what, 5: awful, => Key is "oppenheimer" (w/ success rate 0.80)

Plaintext is Reveal Thy Self; what awful Form art Thou? I worship Thee! Have mercy, God supreme! Thine inner Being I am fain to know; This Thy forth-streaming Life bewilders me. Time am I, laying desolate the world, Made manifest on earth to slay mankind! Not one of all these warriors ranged for strife Escapeth death; thou shalt alone survive. Therefore stand up! win for thyself renown, Conquer thy foes, enjoy the wealth filled realm, By Me they are already overcome, Be thou the outward cause, left-handed one. Drona and Bhishma and Jayadratha, Karna, and all the other warriors here, Are slain by Me. Destroy them fearlessly. Fight! thou shalt crush thy rivals in the field. Sanjaya said: Having heard these words of Keshava, he who weareth a diadem, with joined palms, quaking and prostrating himself, spake again to Kri- shna, stammering with fear, casting down his face.
```

Execution for another ciphertext:

```
dimitris@dimitris-MS-7E12:~/cs4520$ ./vigenere
Clean string is: jhmmqepkwroekubrpdekvrkrorgmzfaclnvwpnephpbnmamsngainltiyycxqazqlnporrqjueuemmimijjebfaewwekxukrsbyeypvmmagsnjhsolvjqrvetjamnskgraqxwaecnzuvyggeagksyzuognnevqgegtogxvqrbnuppdbueaakqixrpibkrgdybexropzzlrgbipmmeflksskxabiiclbosauqigaixlbognnpgpxvelqtxutsctamedxuartsnphvcpkzmaimbfisddqhprgvzbkwhsizwucptjxxstvqsscqipamypenmpjgiagsezvrcnegwziptdpgpyvlqfuyelkzynnpgckmyewljvvmytfzciziegwcymwhsmtsoveqaxhhtbnshvxprlptrpzlfvmvgxvgpvzmbpsngaitybkpyggbbyqly

Distances between instances of 2-graph ai are: 0, 180, 43, 161
Distances between instances of 2-graph am are: 0, 77, 132, 60
Distances between instances of 2-graph mm are: 0, 74, 27, 103
Distances between instances of 2-graph rg are: 0, 163, 13, 84
Distances between instances of 2-graph sn are: 0, 60, 156, 168
GCDs of distances between instances of 2-graph ai are:  Total: 0 GCD(s),  Values: []

GCDs of distances between instances of 2-graph am are:  Total: 2 GCD(s),  Values: [11, 12]

GCDs of distances between instances of 2-graph mm are:  Total: 0 GCD(s),  Values: []

GCDs of distances between instances of 2-graph rg are:  Total: 0 GCD(s),  Values: []

GCDs of distances between instances of 2-graph sn are:  Total: 1 GCD(s),  Values: [12]

Checking for key with length = 11. Extracted Words: 1: fai, 2: nl, 3: ltyij, 4: oadqcmwh, 5: swnr,  => Discarded key "ehezftwmoia" (produced non-words)
Checking for key with length = 12. Extracted Words: 1: but, 2: if, 3: these, 4: machines, 5: were, => Key is "intelligence" (w/ success rate 0.60)

Plaintext is But if these machines were ingenious, what shall we think of the calculating machine of Mr. Babbage? What shall we think of an engine of wood and metal which can not only compute astronomical and navigation tables to any given extent, but render the exactitude of its operations mathematically certain through its power of correcting its possible errors? What shall we think of a machine which can not only accomplish all this, but actually print off its elaborate results, when obtained, without the slightest intervention of the intellect of man?


Distances between instances of 3-graph gai are: 0, 180, 204
GCDs of distances between instances of 3-graph gai are:  Total: 1 GCD(s),  Values: [12]

Checking for key with length = 12. Extracted Words: 1: but, 2: if, 3: these, 4: machines, 5: were, => Key is "intelligence" (w/ success rate 0.60)

Plaintext is But if these machines were ingenious, what shall we think of the calculating machine of Mr. Babbage? What shall we think of an engine of wood and metal which can not only compute astronomical and navigation tables to any given extent, but render the exactitude of its operations mathematically certain through its power of correcting its possible errors? What shall we think of a machine which can not only accomplish all this, but actually print off its elaborate results, when obtained, without the slightest intervention of the intellect of man?
```