1. Team Information

- **Team Name:** N/A (Solo)
- **Team Members:**
    - Joshua Alonzo

---

2. Project Information and Details

Problem Statement →

In the digital age, the security of information during transmission is important. Simple substitution ciphers, such as the Caesar Cipher, are easily broken by frequency analysis because they map each letter of the plaintext to a single fixed letter in the ciphertext. The problem we are addressing is the need for a stronger, lightweight method of encryption that obscures letter frequency patterns, making the message more difficult for unauthorized parties to decipher without the correct key. We aim to demonstrate how polyalphabetic substitution can be implemented programmatically to solve this vulnerability in basic text transmission.

Proposed Solution →

To address this problem, I have implemented the Vigenère Cipher using the C++ programming language. Unlike monoalphabetic ciphers, the Vigenère Cipher uses a keyword to shift letters variably throughout the message. This essentially applies a series of different Caesar Ciphers based on the letters of a keyword. We solved the problem of implementation by developing an algorithm that:

1. Ingests a plaintext message and a keyword from the user.
2. Normalizes the input (removing non-alphabetic characters) to ensure mathematical consistency.
3. Generates a "running key" by repeating the keyword until it matches the length of the plaintext.
4. Apply modular arithmetic to shift the plaintext characters into ciphertext and reversible decryption logic to recover the original message.

Calculations and Algorithm Implementation →

The core of our implementation relies on mapping the English alphabet to a set of integers.

- **Encryption**
  The the plaintext(P) and key(K) are added modulo 26.
  $E_i = (P_i + K_i) \bmod 26$

  Decryption
  $D_i = (E_i - K_i + 26) \bmod 26$

Program Objectives and User Interaction →

The primary objective of this program is to provide a user-friendly tool for symmetric key encryption and decryption. It serves as an educational demonstration of how classical cryptography works mathematically.

- **User Interaction:** Upon launching the console application, the user is prompted to enter two strings: a **Keyword** (the secret key) and a **Message** (the text to be encrypted).
- **Purpose:** The program processes these inputs to demonstrate the full cryptographic lifecycle. It first cleans the data, generates the full cyclic key, encrypts the message to show the ciphertext, and immediately decrypts it back to show the integrity of the algorithm. This immediate feedback loop allows the user to verify that the math is accurate and the message was not corrupted during the transformation.

Implementation of Discrete Structures →

Discrete structures are the mathematical foundation of this C++ program. Specifically, we utilize Modular Arithmetic and Congruence Relations.

1. **Sets and Mapping:** We define a set *S* containing the 26 characters of the English alphabet. We implement a bijective function (one-to-one correspondence) that maps elements from Set *S* to the set of integers modulo 26 (Z_26). This discrete mapping allows us to perform arithmetic operations on "letters."
2. **Congruence Modulo :** The cipher operates in a cyclic group. The modulo operator % in C++ enforces the rule that *26 is equiv to 0.* This is a direct application of congruence theory, ensuring that if a shift operation goes past 'Z', it wraps around to 'A' seamlessly.
3. **Functions:** The encryption and decryption processes are implemented as mathematical functions where the domain and codomain are both Z_26. The generation of the cyclic key can be viewed as a periodic function that repeats the sequence of the keyword *K* for the length of *P*

Program Limitations –.

While the program successfully implements the algebraic logic of the Vigenère Cipher, it has specific limitations:

1. **Character Set Restriction:** The program currently only supports the 26 letters of the English alphabet. It strictly removes numbers, punctuation, and spaces during the "cleaning" phase. A message like "Meeting at 5!" becomes "MEETINGAT".

2. **Loss of Formatting:** Because spaces are removed, the sentence structure is lost in the decryption process. The output is a continuous block of text (e.g., "HELLOWORLD"), which can be difficult to read for long messages.
3. **Case Insensitivity:** The program converts all input to uppercase to simplify the integer mapping. The distinction between "Apple" and "apple" is lost.

Recommendations for Improvement →

To address the limitations above, the following improvements are recommended for future iterations:

1. **Expand the Modulo Base:** Instead of using Modulo 26, we could implement Modulo 95 or Modulo 128 (ASCII base). This would allow the encryption of numbers, symbols, and punctuation without stripping them, as we would map the entire printable ASCII range rather than just A-Z.
2. **Preserve Non-Alphabetic Characters:** We could modify the algorithm to check if a character is a letter. If it is a letter, apply the shift; if it is a space or punctuation, leave it unchanged in the ciphertext. This would preserve readability (e.g., "Hello World" -> "Rijvs Yvjn").
3. **Case Preservation:** We could check the case of the input character (using isupper() or islower()) before processing, apply the math relative to 'A' or 'a' respectively, and cast the result back to the correct case

==========PSUDO-CODE==========

```
START PROGRAM

   // 1. Helper Function to remove spaces/symbols and uppercase text
   FUNCTION CleanString(inputString)
      Create empty string result
      FOR each character C in inputString
         IF C is a letter
            Convert C to Uppercase
            Append C to result
         END IF
      END FOR
      RETURN result
   END FUNCTION

   // 2. Helper Function to stretch the keyword to match message length
   FUNCTION GenerateKey(text, keyword)
      Create empty string newKey
      Set k_index = 0

      FOR i from 0 to length of text
         Append keyword[k_index] to newKey
         Increment k_index

         // Loop back to start of keyword if we reach the end
         IF k_index equals length of keyword
            Set k_index = 0
         END IF
      END FOR
      RETURN newKey
   END FUNCTION

   // 3. Encryption Logic
```

```
FUNCTION Encrypt(plaintext, key)
    Create empty string cipherText
    FOR i from 0 to length of plaintext
        P_val = value of plaintext[i] (A=0 ... Z=25)
        K_val = value of key[i] (A=0 ... Z=25)

        // Formula: (P + K) % 26
        EncryptedVal = (P_val + K_val) MOD 26

        Convert EncryptedVal back to Character
        Append Character to cipherText
    END FOR
    RETURN cipherText
END FUNCTION

// 4. Decryption Logic
FUNCTION Decrypt(cipherText, key)
    Create empty string originalText
    FOR i from 0 to length of cipherText
        E_val = value of cipherText[i] (A=0 ... Z=25)
        K_val = value of key[i] (A=0 ... Z=25)

        // Formula: (E - K + 26) % 26
        // We add 26 to handle negative results from subtraction
        DecryptedVal = (E_val - K_val + 26) MOD 26

        Convert DecryptedVal back to Character
        Append Character to originalText
    END FOR
    RETURN originalText
END FUNCTION

// --- MAIN EXECUTION FLOW ---
MAIN
    OUTPUT "Enter Keyword: "
    INPUT rawKeyword

    OUTPUT "Enter Message: "
    INPUT rawMessage

    // Step A: Prepare the data
    cleanMsg = CleanString(rawMessage)
    cleanKey = CleanString(rawKeyword)
```

```
        // Step B: Create the cyclic key
        fullKey = GenerateKey(cleanMsg, cleanKey)

        // Step C: Perform Operations
        encryptedMsg = Encrypt(cleanMsg, fullKey)
        decryptedMsg = Decrypt(encryptedMsg, fullKey)

        // Step D: Show results
        OUTPUT "Cleaned Text: " + cleanMsg
        OUTPUT "Generated Key: " + fullKey
        OUTPUT "Encrypted: " + encryptedMsg
        OUTPUT "Decrypted: " + decryptedMsg
    END MAIN

END PROGRAM
```