Problem 1

1. $f(x) = x^2 + 1,\ g(x) = x^3 + x^2 + 1$
   (a) $f(x) + g(x) = x^3 + 2x^2 + 2 = x^3$
   (b) $f(x) - g(x) = f(x) + g(x) = x^3$
   (c) $f(x) \times g(x)\ mod\ P(x)$
   $= x^5 + x^4 + x^2 + x^3 + x^2 + 1\ mod\ P(x)$
   $\equiv x(x+1) + (x+1) + x^3 + 2x^2 + 1\ mod\ P(x)$
   $\equiv x^3 + 3x^2 + 2x + 2\ mod\ P(x)$
   $\equiv x^3 + x^2\ mod\ P(x)$

2. $f(x) = x^2 + 1,\ g(x) = x + 1$
   (a) $f(x) + g(x) = x^2 + x + 2 = x^2 + x$
   (b) $f(x) - g(x) = f(x) + g(x) = x^2 + x$
   (c) $f(x) \times g(x)\ mod\ P(x) = x^3 + x^2 + x + 1\ mod\ P(x)$

Problem 2

1. $f(x) = x^7 + x^5 + x^4 + x + 1,\ g(x) = x^3 + x + 1$
   (a) $f(x) + g(x) = x^7 + x^5 + x^4 + x^3 + 2x + 2 = x^7 + x^5 + x^4 + x^3$
   (b) $f(x) - g(x) = f(x) + g(x) = x^7 + x^5 + x^4 + x^3$
   (c) $f(x) \times g(x)\ mod\ m(x)$
   $= x^{10} + 2x^8 + 2x^7 + x^6 + 2x^5 + 2x^4 + x^3 + x^2 + 2x + 1\ mod\ m(x)$
   $\equiv x^{10} + x^6 + x^3 + x^2 + 1\ mod\ m(x)$
   $\equiv x^2(x^4 + x^3 + x + 1) + x^6 + x^3 + x^2 + 1\ mod\ m(x)$
   $\equiv 2x^6 + x^5 + 2x^3 + 2x^2 + 1\ mod\ m(x)$
   $\equiv x^5 + 1\ mod\ m(x)$

2. $f(x) = x^4 + 1 = x^4 + 2x^3 + 2x^2 + 2x + 1 = (x+1)(x^3 + x^2 + x + 1)$, therefore $f(x)$ is reducible.

Problem 3

1. Since $x(x^3 + 1)\ mod\ P(x) \equiv x^4 + x\ mod\ P(x) \equiv 1\ mod\ P(x)$, the inverse of $f(x) = x$ is $x^3 + 1$.

2. Since $(x^2 + x)(x^2 + x + 1)\ mod\ P(x) \equiv x^4 + 2x^3 + 2x^2 + x\ mod\ P(x) \equiv x^4 + x\ mod\ P(x)$
   $\equiv 1\ mod\ P(x)$, the inverse of $g(x) = x^2 + x$ is $x^2 + x + 1$.

Problem 4

1. $(x^3+x^2+x)(x^3+x^2)^{-1}\ mod\ (x^8+x^4+1)$
   $\equiv (x^3+x^2+x)(x^5+x^4+x^3+x^2)\ mod\ (x^8+x^4+1)$
   $\equiv x^8+2x^7+3x^6+3x^5+2x^4+x^3\ mod\ (x^8+x^4+1)$
   $\equiv x^8+x^6+x^5+x^3\ mod\ (x^8+x^4+1)$
   $\equiv (x^4+1)+x^6+x^5+x^3\ mod\ (x^8+x^4+1)$
   $\equiv x^6+x^5+x^4+x^3+1\ mod\ (x^8+x^4+1)$

2. $(x^6+x^3+1)(x^4+x^3+1)\ mod\ (x^8+x^4+1)$
   $\equiv x^{10}+x^9+x^7+2x^6+x^4+2x^3+1\ mod\ (x^8+x^4+1)$
   $\equiv x^{10}+x^9+x^7+x^4+1\ mod\ (x^8+x^4+1)$
   $\equiv (x^2+x)(x^4+1)+x^7+x^4+1\ mod\ (x^8+x^4+1)$
   $\equiv x^7+x^6+x^5+x^4+x^2+x+1\ mod\ (x^8+x^4+1)$

Problem 5

1. The matrix coefficients correspond to specific operations in $\mathbb{GF}(2^8)$:
   - Multiplication by 1 is the identity operation.
   - Multiplication by 2 is a left shift followed by a conditional XOR with 0x1b (if the shift results in a byte larger than 0xFF).
   - Multiplication by 3 is a left shift and then XOR with the original byte, followed by the same conditional XOR with 0x1b.

   Since the multiplication by 2 and 3 are always performed in the same way and involve fixed operations on bytes, the results can be precomputed and stored in a table. The final result for each byte in a column is the XOR of several such multiplications.

2. (a) Byte Substitution
   Uses an S-box to substitute each byte. This substitution is also a transformation that can be precomputed for each possible byte value, effectively acting as a look-up table.

   (b) Shift Rows
   A permutation of bytes within the state matrix that doesn't depend on the byte values but only on their positions. This means that the operation is deterministic and consistent across every round.

   (c) Mix Columns
   As explained, can be implemented using look-up tables for multiplications by 2 and 3, and XOR operations for adding these results.

   Therefore, the entire AES process can be efficiently implemented using lookup tables for byte substitution and precomputed arrays for MixColumns and Inverse MixColumns, alongside simple XOR operations for AddRoundKey.

Problem 6

1. Indifference of order for InvSubBytes and InvShiftRows
   - ShiftRows and SubBytes are non-overlapping operations: ShiftRows permutes byte positions, while SubBytes replaces byte values.
   - InvShiftRows simply undoes the permutation of ShiftRows, and InvSubBytes undoes the byte replacement.

   Since these operations affect different aspects of the state (positions/values), their order can be switched without impacting the outcome.

2. Inverting the order of AddRoundKey and InvMixColumns with adapted round key
   - AddRoundKey involves a simple XOR of the round key with the state.
   - InvMixColumns is a linear operation in the field $\mathbb{GF}(2^8)$.

   Normally, we would compute $InvMixColumns(state \oplus roundKey)$. By adapting the round key first: $adaptedKey = InvMixColumns(roundKey)$, we can instead compute $state \oplus adaptedKey$ and then apply InvMixColumns.

3. Take advantage of this property to simplify the decryption process
   - Software Implementation
     Functions for SubBytes and ShiftRows can be reused as their inverses with minor modifications.
   - Hardware Implementation
     The same circuits can be used for both MixColumns and InvMixColumns, reducing the chip area required.

Problem 7

- When to choose 3DES over AES
  3DES may be the preferable choice in environments where legacy systems require compatibility with older cryptographic standards or where hardware limitations prevent the implementation of AES.
- Advantages of choosing AES over 3DES
  AES offers several advantages over 3DES, including superior performance and enhanced security. With key sizes up to 256 bits, AES provides stronger security and is better suited to defend against modern cryptographic attacks. Its efficiency in processing and lower power consumption make it ideal for use in both high-powered and resource-constrained environments.
- Susceptibility to meet-in-the-middle attack
  While theoretically feasible, a Meet-in-the-Middle attack on 3DES is not practical with current technology due to the vast computational resources required. This is especially true when all three keys $(K_1,\ K_2,\ K_3)$ are independent and strong. The effective security provided by 3DES against such attacks, when using three distinct keys, is considered to be about 112 bits, which is currently beyond the reach of brute-force capabilities for symmetric encryption.

Problem 8

According to NIST Special Publication 800-57 Part 1, the following steps outline a general approach to revoking an old encryption key and deploying a new one:

1. Identify and Isolate Encrypted Data
   Identify all the data that was encrypted with the key held by the former chief technology officer.

2. Generate a New Key
   A new encryption key should be generated securely.

3. Access Control Adjustments
   Update access controls and permissions to ensure that only authorized personnel can access the new encryption key and associated data.

4. Decrypt and Re-encrypt Data
   Decrypt the data using the old key and re-encrypt it using the new key.

5. Deploy the New Key
   Once the data is re-encrypted, deploy the new key across all relevant systems and services that require access to the encrypted data.

6. Revoke the Old Key
   After the data is securely re-encrypted, and the new key is fully deployed, the old key should be revoked.

7. Audit and Compliance
   Conduct a thorough audit of the key change process to ensure compliance with all relevant policies and regulations.

8. Continuous Monitoring and Update
   Implement continuous monitoring mechanisms to detect any unauthorized access attempts using the old key.

Problem 9

1. By only changing e and not n, Alice may encounter a situation where e is not coprime to $\phi(n)$, which makes e lack an inverse modulo $\phi(n)$. Without d, the private key cannot be generated, making the decryption process impossible.

2. In some specific cases such as when $n = pq$ and $n_l = p_l q_l = (p+2)(q+2)$, we can express n_l as $n_l = n + 2(p+q+2)$. This results in $n_l - n = 2(p+q+2)$. Since p and q are odd primes, making $n_l - n$ even. Given that $\phi(n) = (p-1)(q-1)$ is also even, $n_l - n$ is not coprime to $\phi(n)$. Therefore, e and $\phi(n)$ are not coprime, and d cannot be calculated.

Problem 10

1. As mentioned in Problem 5, the following is the finite field arithmetic for $\mathbb{GF}(2^8)$:
   - Addition
     Addition of two elements is performed by the bitwise XOR operation.
   - Multiplication
     If the multiplication of two polynomials results in a degree of 8 or higher, it is then reduced by the irreducible polynomial.

   In the Inverse MixColumns step of AES decryption, the matrix multiplication differs from standard matrix arithmetic because it is performed within the finite field $\mathbb{GF}(2^8)$, where both multiplication and addition of matrix elements are conducted using finite field arithmetic.

2. We can use combinations of simpler operations:
   - Multiplication by 2k
     - $x \cdot 2 = x$ left-shift by 1 bit
     - $x \cdot 4 = (x \cdot 2) \cdot 2$
     - $x \cdot 8 = (x \cdot 4) \cdot 2$
   - Multiplication by 9
     $x \cdot 9 = (x \cdot 8) \oplus x$
   - Multiplication by 11
     $x \cdot 11 = (x \cdot 8) \oplus (x \cdot 2) \oplus x$
   - Multiplication by 13
     $x \cdot 9 = (x \cdot 8) \oplus (x \cdot 4) \oplus x$
   - Multiplication by 14
     $x \cdot 9 = (x \cdot 8) \oplus (x \cdot 4) \oplus (x \cdot 2)$

3. Using lookup tables (LUTs) in AES decryption can significantly speed up the Inverse MixColumns operation by precomputing results of the finite field multiplications and storing them for quick retrieval.
   (a) Advantages
   - Performance: Reduces the computation time significantly as table lookup operations are generally faster than real-time polynomial multiplications in $\mathbb{GF}(2^8)$.
   - Efficiency: Simplifies implementation in hardware and software, leading to faster and potentially more energy-efficient decryption processes.

   (b) Drawbacks
   - Memory Usage: Requires additional memory to store the LUTs, which might be a concern in memory-constrained environments.
   - Scalability and Flexibility: Changes in cryptographic parameters or algorithms would require generating and storing new tables, reducing flexibility.

Problem 11

We can compute m_0 of the procedure following:

1. Exploiting the known information

$$C_0 = IV \oplus Encrypt(m_0)$$
$$C_1 = m_0 \oplus Encrypt(m_1) = m_0 \oplus Encrypt(x)$$
$$C_2 = m_1 \oplus Encrypt(m_2) = x \oplus Encrypt(x)$$

2. Compute $Encrypt(x)$

$$Encrypt(x) = C_2 \oplus x$$

3. Reveal m_0

$$m_0 = C_1 \oplus Encrypt(x) = C_1 \oplus C_2 \oplus x$$

Since the attacker knows $C_1,\ C_2,\ x$, m_0 can be computed by simple XOR operations.

Problem 12

Attack Steps:

1. Create $M3$
   Construct $M3$ such that $M3 = M1||\Delta||M2$, where Δ is a block to adjust the chaining.

2. Choose Δ
   Δ should be choosen such that $\Delta = T_1 \oplus$ desired initial changing value for $M2$.

3. Send $M3$
   The entire message $M3$ processes through the encryption scheme, with $M1$ and Δ setting up the proper conditions so that when $M2$ is processed, it results in $T2$ be the final output, making $(IV_1,\ T_2)$ a valid MAC for $M3$.

This attack method effectively demonstrates an existential forgery against Alice's MAC scheme by exploiting the lack of integrity between concatenated message blocks and the reuse of IV and chaining mechanisms.

Extra Credit 1

1. Security Goals
   (a) Integrity: Integrity guarantees that the source code reflects only authorized modifications.
   (b) Confidentiality: Essential for protecting sensitive data such as source code of private projects.
   (c) Availability: Necessary to ensure that the platform and its services are always accessible to users without interruptions.
2. Roles and Functions
   (a) User Roles
      i. End User
         - Functions: Clone repositories, submit pull requests, and open issues.
         - Policy: Restricted to actions allowed by repository settings based on their role.
      ii. Repository Owner
         - Functions: Manage repository settings, control access permissions for collaborators, merge pull requests, and manage issues.
         - Policy: Responsible for ensuring data integrity, and setting up branch protection rules.
   (b) System Roles
      i. Repository System
         - Functions:Host and manage git repositories, handle git operations, and maintain the version history and integrity of code changes.
         - Policy: Ensure data integrity through cryptographic hash checks, provide backup solutions, and maintain high availability.
      ii. Authentication System
         - Functions: Manage user authentication and session management and enforce multi-factor authentication.
         - Policy: Use strong encryption for data storage and transmission, and regularly update system security measures.
3. Policy
   - Regular security audits and compliance checks.
   - Strong encryption use for all data at rest and in transit.
   - Detailed logging of all user and system actions for audit and recovery purposes.

Extra Credit 2

1. The key to decrypting these ciphertexts lies in their XOR difference, which effectively removes the pad and reveals the XOR of the original plaintexts. We can analyze the plaintexts using Python code:

```python
def hex_to_bytes(hex_string):
    return bytes.fromhex(hex_string)

def xor_bytes(bytes1, bytes2):
    return bytes([b1 ^ b2 for b1, b2 in zip(bytes1, bytes2)])

def string_to_bytes(text):
    return text.encode('utf-8')

def read_words(file_path):
    with open(file_path, 'r') as file:
        return [line.strip() for line in file]

ciphertext1 = "e93ae9c5fc7355d5"
ciphertext2 = "f43afec7e1684adf"

ciphertext1_bytes = hex_to_bytes(ciphertext1)
ciphertext2_bytes = hex_to_bytes(ciphertext2)

xor_result = xor_bytes(ciphertext1_bytes, ciphertext2_bytes)

xor_result_hex = xor_result.hex()
print("XOR Result:", xor_result_hex)

words = read_words("top-10k-word.txt")

for word in words:
    word_bytes = string_to_bytes(word)
    if len(word_bytes) == len(xor_result):
        possible_plaintext = xor_bytes(xor_result, word_bytes)

        decoded_text = possible_plaintext.decode('utf-8')
        if decoded_text in words:
            print(f"Match found: {word} -> {decoded_text}")
```

This Python script reads two hexadecimal-encoded ciphertexts, XORs them together, and then attempts to decode the result by XORing it with each word from a list of the top 10,000 words, checking if the resulting plaintext is also a valid word from the list, and prints any matches found.
Output:

```
XOR Result: 1d0017021d1b1f0a
Match found: security -> networks
Match found: networks -> security
```

The output shows the XOR result and two words **security** and **networks**.

2. To attack the encryption, we first need to identify the valid characters. The messages consist of ”A-Z”, ”a-z”, and punctuation symbols such as ”.,!? ()-”. Then, for each of the ten ciphertexts, we can check whether each character is valid after decryption:

```
Lengths of each sublist: [20, 6, 2, 1, 2, 1, 1, 1, 1, 5, 5, 4, 1, 1, 1, 8, 3,
    2, 2, 2, 1, 1, 1, 3, 3, 1, 1, 1, 1, 5, 4, 1, 1, 2, 1, 2, 2, 9, 1, 4, 2,
   1, 1, 2, 1, 1, 1, 1, 1, 2, 2, 1, 2, 2, 1, 1, 1, 3, 2, 1]
Product of all lengths: 183458856960000
```

Given that the product of all lengths is too large for a brute-force approach, we can tackle the problem by dividing each ciphertext into blocks. As in the previous problem, we have utilized the same list of the top 10,000 words for the word list. Each block is evaluated based on how many words it contains from this word list, with the contribution to the overall score determined by the length of each word. After completing the recursive process, we output the best message and pad. Below is the output from the first attempt:

```
Loaded 9894 unique words from 'top-10k-word.txt'.
Decrypted Messages:
  We stand today on the brink of a revolution in cryptographx.
  Probabilistic encryption is the use of randomness in an enbr
  Secure Sockets Layer (SSL), are cryptographic protocols thbt
  This document will detail a vulnerability in the ssh cryptlg
  MIT developed Kerberos to protect network services providee
  NIST announced a competition to develop a new cryptographib
  Diffie-Hellman establishes a shared secret that can be usec
  Public-key cryptography refers to a cryptographic system rdq
  The keys used to sign the certificates had been stolen frol
  We hope this inspires others to work in this fascinating fhe
Final Pad Values:
  [119, 75, 116, 51, 85, 113, 72, 105, 76, 78, 114, 79, 84, 49, 71, 101, 71,
     88, 116, 78, 113, 102, 113, 87, 84, 65, 51, 55, 99, 56, 107, 69, 116,
     105, 110, 109, 97, 113, 79, 106, 122, 68, 66, 98, 77, 72, 112, 72, 55,
     53, 104, 54, 99, 71, 87, 97, 68, 98, 113, 49]

Execution time: 14.337386846542358 seconds
```

It appears that there are errors in the last block of the decrypted message. Therefore, we have modified the scoring function to use the square of the word length. Below is the revised output:

```
Loaded 9894 unique words from 'top-10k-word.txt'.
Decrypted Messages:
  We stand today on the brink of a revolution in cryptography.
  Probabilistic encryption is the use of randomness in an encr
  Secure Sockets Layer (SSL), are cryptographic protocols that
  This document will detail a vulnerability in the ssh cryptog
  MIT developed Kerberos to protect network services provided
  NIST announced a competition to develop a new cryptographic
  Diffie-Hellman establishes a shared secret that can be used
  Public-key cryptography refers to a cryptographic system req
  The keys used to sign the certificates had been stolen from
  We hope this inspires others to work in this fascinating fie
Final Pad Values:
  [119, 75, 116, 51, 85, 113, 72, 105, 76, 78, 114, 79, 84, 49, 71, 101, 71,
     88, 116, 78, 113, 102, 113, 87, 84, 65, 51, 55, 99, 56, 107, 69, 116,
     105, 110, 109, 97, 113, 79, 106, 122, 68, 66, 98, 77, 72, 112, 72, 55,
     53, 104, 54, 99, 71, 87, 97, 68, 98, 112, 49]

Execution time: 14.37517762184143 seconds
```

Finally, the results display both the message and the pad used. Below is the complete code utilized to attack the ciphertexts:

```
def ben_decrypt(ctext, pad, prev_c = 0):
    assert len(ctext) == len(pad)
    result = []
    for i in range(len(ctext)):
        b = ctext[i] ^ ((pad[i] + prev_c) % 256)
        result.append(b)
        prev_c = ctext[i]
    return result

def text_to_bytes(t):
    return [ord(c) for c in t]

val_char_list = [chr(i) for i in range(65, 91)] + [chr(i) for i in range(97,
   123)] + ['.', ',', '!', '?', ' ', '(', ')', '-']
val_char_bytes = set([ord(c) for c in val_char_list])

tenciphs = [
    [32, 14, 162, 166, 143, 97, 199, 84, 128, 186, 67, 246, 43, 37, 76, 222,
       75, 131, 131, 185, 79, 149, 100, 201, 116, 219, 101, 188, 112, 206,
       25, 63, 147, 142, 153, 112, 190, 67, 231, 37, 246, 85, 249, 123, 161,
        135, 215, 124, 193, 143, 135, 201, 67, 237, 54, 246, 74, 196, 77,
       80],
    [39, 0, 27, 44, 224, 51, 18, 23, 10, 43, 233, 81, 198, 215, 123, 142,
       182, 124, 137, 167, 108, 187, 67, 244, 104, 192, 128, 151, 142, 174,
       124, 225, 32, 250, 13, 90, 212, 35, 82, 206, 41, 3, 33, 236, 84, 242,
        7, 60, 0, 21, 20, 36, 167, 143, 136, 201, 104, 164, 119, 218],
```

```
    [36, 10, 29, 37, 8, 28, 68, 254, 37, 16, 233, 93, 197, 133, 236, 29, 5,
       36, 253, 57, 138, 216, 26, 34, 58, 82, 169, 192, 66, 8, 22, 123, 140,
        135, 140, 137, 158, 96, 200, 64, 219, 111, 217, 82, 252, 100, 164,
       158, 186, 155, 108, 193, 75, 254, 38, 167, 159, 105, 184, 157],
    [35, 6, 19, 53, 170, 127, 168, 114, 203, 116, 131, 188, 100, 181, 139,
       153, 140, 136, 220, 78, 218, 52, 196, 114, 170, 203, 159, 246, 47,
       18, 17, 56, 201, 64, 207, 94, 214, 43, 19, 9, 250, 30, 9, 5, 114,
       206, 86, 251, 18, 52, 239, 77, 144, 180, 121, 163, 151, 141, 146,
       164],
    [58, 204, 20, 103, 216, 44, 2, 14, 54, 235, 45, 25, 9, 26, 42, 234, 67,
       249, 8, 36, 250, 19, 164, 143, 140, 237, 80, 245, 55, 27, 227, 75,
       203, 20, 236, 60, 233, 45, 19, 15, 226, 6, 59, 248, 55, 9, 16, 59,
       23, 63, 135, 205, 66, 230, 75, 197, 109, 170, 126, 143],
    [57, 205, 18, 17, 70, 214, 112, 183, 108, 207, 47, 29, 20, 33, 72, 204,
       51, 232, 51, 236, 45, 246, 19, 3, 35, 13, 47, 8, 75, 247, 13, 114,
       130, 142, 138, 146, 159, 127, 190, 8, 227, 7, 39, 236, 78, 182, 69,
       255, 79, 244, 40, 49, 243, 72, 254, 47, 27, 20, 231, 56],
    [51, 23, 237, 70, 242, 6, 99, 132, 181, 111, 141, 177, 100, 251, 98, 162,
        154, 134, 155, 139, 144, 159, 99, 210, 67, 247, 10, 32, 163, 168,
       123, 161, 103, 181, 71, 148, 134, 146, 130, 158, 125, 181, 215, 77,
       242, 91, 191, 39, 61, 19, 21, 107, 172, 150, 205, 91, 236, 43, 255,
       16],
    [39, 7, 25, 32, 28, 238, 27, 239, 94, 213, 103, 213, 91, 245, 76, 197,
       99, 220, 34, 17, 242, 48, 216, 15, 17, 55, 12, 38, 251, 64, 139, 164,
        119, 192, 79, 156, 158, 125, 181, 111, 157, 142, 183, 107, 217, 81,
       169, 152, 172, 193, 90, 233, 63, 242, 44, 224, 4, 20, 225, 99],
    [35, 6, 31, 114, 172, 120, 185, 81, 189, 126, 131, 183, 111, 128, 179,
       119, 158, 133, 144, 185, 68, 138, 143, 142, 135, 232, 120, 202, 95,
       227, 39, 10, 23, 227, 48, 233, 47, 211, 2, 4, 31, 7, 105, 169, 147,
       190, 64, 168, 172, 149, 146, 164, 98, 199, 62, 249, 79, 222, 35,
       116],
    [32, 14, 162, 189, 125, 158, 131, 204, 108, 210, 45, 15, 67, 29, 10, 28,
       19, 2, 4, 55, 219, 97, 189, 96, 220, 120, 217, 99, 230, 106, 186,
       223, 36, 226, 34, 228, 101, 191, 96, 234, 16, 60, 23, 10, 119, 217,
       40, 3, 89, 231, 33, 54, 237, 93, 218, 92, 128, 132, 157, 171]
]

def cal_pad(ctext, msg, prev_c = 0):
    assert len(ctext) == len(msg)
    pad = []

    for i in range(len(ctext)):
        p = (((ctext[i] ^ msg[i]) - prev_c + 256) % 256)
        pad.append(p)
        prev_c = ctext[i]

    return pad

def get_prev_c(ctext, index):
    if index == 0:
```

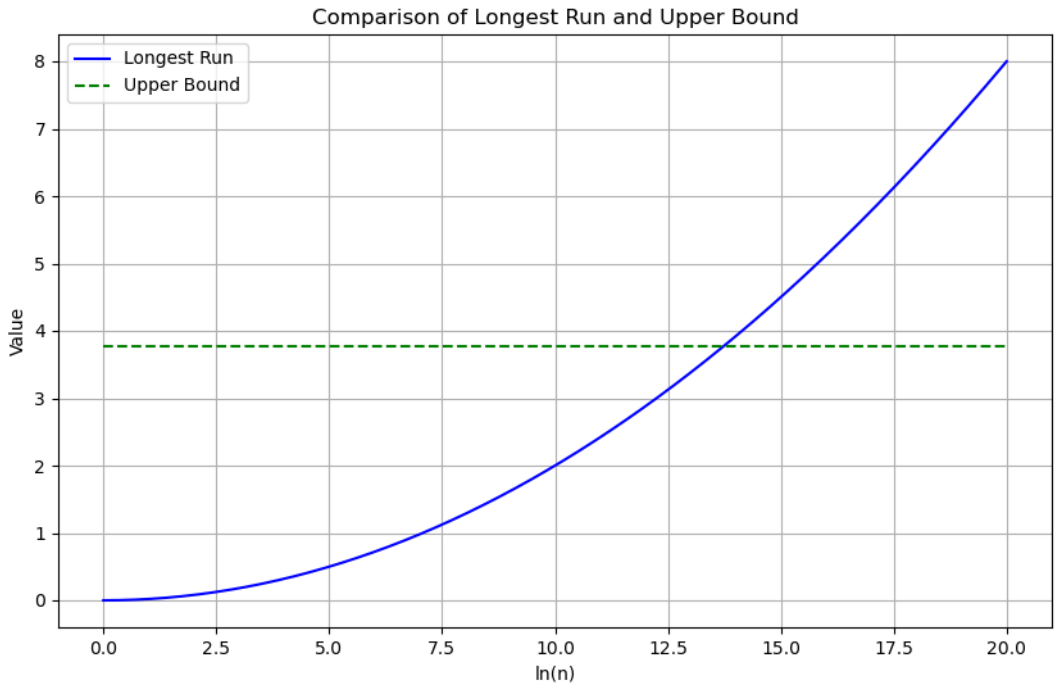
```
42          return 0
43      return ctext[index - 1]
44
45  def get_ctext(ctext, index):
46      return ctext[index:index + 1]
47
48  msglen = 60
49
50  val_pad_bytes = [[] for _ in range(msglen)]
51
52  for index in range(msglen):
53      for c in val_char_bytes:
54          val_pad_byte = cal_pad(get_ctext(tenciphs[0], index), [c], get_prev_c
                (tenciphs[0], index))
55          flg = True
56          for ciph in tenciphs:
57              msg = ben_decrypt(get_ctext(ciph, index), val_pad_byte,
                    get_prev_c(ciph, index))
58              if not set(msg).issubset(val_char_bytes):
59                  flg = False
60                  break
61          if flg:
62              val_pad_bytes[index].append(val_pad_byte)
63
64  def solve(n_index, n_pad, words):
65      if n_index == msglen:
66          msg = [bytes_to_text(ben_decrypt(ciph[:msglen], n_pad)) for ciph in
                tenciphs]
67          proc_msg = "\n".join(msg).lower()
68          scr = sum(len(word) ** 2 for word in words if word in proc_msg)
69          return msg, n_pad, scr
70      else:
71          max_scr = 0
72          bst_msg = []
73          bst_pad = []
74          for p in val_pad_bytes[n_index]:
75              if p:
76                  updated_pad = n_pad + [item for sublist in p for item in (
                        sublist if isinstance(sublist, list) else [sublist])]
77                  msg, pad, scr = solve(n_index + 1, updated_pad, words)
78                  if scr > max_scr:
79                      max_scr = scr
80                      bst_msg = msg
81                      bst_pad = pad
82          return bst_msg, bst_pad, max_scr
83
84
85  def bytes_to_text(b):
86      return ''.join(chr(c) for c in b)
87
```

```python
import time
from functools import reduce
from operator import mul

if __name__ == '__main__':
    start_time = time.time()

    lengths = [len(p) for p in val_pad_bytes]
    print("Lengths of each sublist:", lengths)

    if lengths:
        total_product = reduce(mul, lengths)
    else:
        total_product = 1
    print("Product of all lengths:", total_product)

    with open("top-10k-word.txt", "r") as f:
        words = set(word.strip().lower() for word in f.readlines())
        print(f"\nLoaded {len(words)} unique words from 'top-10k-word.txt'.")

        msg = []
        pad = []

        msglen = 5
        for i in range(10, 65, 5):
            msglen += 5
            msg, pad, _ = solve(i - 10, pad, words)
            if i != 60:
                pad = pad[:-5]

        print("Decrypted Messages:")
        for message in msg:
            print(f"  {message}")
        print("Final Pad Values:")
        print(f"  {pad}")

    end_time = time.time()
    print(f"\nExecution time: {end_time - start_time} seconds")
```

Extra Credit 3

1. We can equate the longest run of matching bits to the longest run of heads in n coin flips. Assuming a pair c_i and c_j is offset by k bits(where$0 < k < n$), there are $2n - 1$ possible offsets. From this, we can derive the upper bound of the longest run of matching bits. Since $n - k < n$, it follows that $R_{0.5}(n - k) < R_{0.5}(n)$. Consequently, $R_{0.5}(n)$ can be expressed $\log_2(n \ln n)$, which simplifies to $\log_2 n + \log_2 \ln n$.

2. The US-ASCII encoding consists of 7-bit sequences. The upper bound is represented by $\log_2(7 \ln(7))$. Below, the graph displays both the longest run and the upper bound:



   The chart indicates that when $\ln n > 13.7$, the average longest run exceeds the upper bound.

3. When the length of two ciphertexts exceeds $e^{13.7}$, we can detect the reuse of the pad:
   (a) Concatenate all ciphertexts into a single string $S = c_1 \&_1 c_{f(n)} \&_{f(n)}$, where $\&_k$ represents the ternimator of the k-th ciphertext. ($O(N)$)
   (b) Construct a suffix tree from S and traverse the tree to find the internal node with the greatest depth d. ($O(N \log N)$)
   (c) When an edge contains $\&_i$, it indicates that this is a suffix of s_i. Using the same process, we can identify another s_j and output them as instances where the pad was reused. ($O(N)$)

   If not, the pad is not reused. The time complexity of this algorithm is $O(N \log N)$.