# Encryption Algorithms Report

## 1. Types of Encryption Algorithms

Encryption algorithms can be categorized in two main ways — based on the key usage and on how data is transformed.

### A. Based on Keys

1. Symmetric Key Encryption: The same key is used for both encryption and decryption.
   Examples: Caesar Cipher, XOR Cipher, Vigenère Cipher, AES, DES.
2. Asymmetric Key Encryption: Two different keys are used — a public key for encryption and a private key for decryption.
   Examples: RSA, ECC, Diffie-Hellman.

### B. Based on Technique

1. Substitution Cipher: Each character in the plaintext is replaced with another character or symbol.
   Examples: Caesar Cipher, XOR Cipher, Vigenère Cipher.
2. Transposition Cipher: The positions of the characters are rearranged without changing the characters themselves.
   Example: Rail Fence Cipher.

## 2. Explanation of Selected Algorithms

### 1. Caesar Cipher

**Caesar Cipher is one of the simplest substitution ciphers. Each letter in the plaintext is shifted by a fixed number of positions down the alphabet.**

**Example: If shift = 3, then A → D, B → E, and so on.**

**Encryption:**

It shifts every letter of plaintext by a fixed number of positions in the alphabet.

```
Algorithm CaesarCipher(plaintext, shift)
1. for each character c in plaintext:
2.     if c is a letter:
3.         replace c with (c + shift) mod 26
4.     else:
5.         keep c unchanged
6. return ciphertext
```

**Complexity:**

- **Time:** O(n) — each character processed once

- **Space:** O(n) — ciphertext of same length as plaintext

### Decryption:

Decryption reverses the process by subtracting the shift.

```
Algorithm CaesarDecrypt(ciphertext, shift)
1. for each character c in ciphertext:
2.     if c is a letter:
3.         replace c with (c - shift) mod 26
4.     else:
5.         keep c unchanged
6. return plaintext
```

**Complexity:**

- **Time:** O(n) — each character processed once
- **Space:** O(n) — ciphertext of same length as plaintext

### 2. XOR Cipher

XOR Cipher uses the XOR (exclusive OR) operation on each character of plaintext with a repeating key.
It is simple but effective for small-scale encryption. Decrypting requires the same key and XOR operation again.

### Encryption:

Each character is XORed with the key; XOR is reversible using the same key.

```
Algorithm XORCipher(plaintext, key)
1. for i = 0 to length(plaintext)-1:
2.     ciphertext[i] = plaintext[i] XOR key[i mod key_length]
3. return ciphertext
```

- ✅ **Overall Encryption Complexity:**
  - **Time Complexity: O(n)** → each character is processed once.
  - **Space Complexity:**
    - **O(n)** if you count storing ciphertext (output).
    - **O(1)** auxiliary space (since no big data structure is used).

### Decryption:

XOR decryption is the same as encryption since XOR is self-inverse.

```
Algorithm XORDecrypt(ciphertext, key)
1. for i = 0 to length(ciphertext)-1:
```

```
2.     plaintext[i] = ciphertext[i] XOR key[i mod key_length]
3. return plaintext
```

✅ **Overall Decryption Complexity:**

- **Time Complexity: O(n)**
- **Space Complexity: O(n)** (including output), **O(1)** auxiliary space.

### 3. Vigenère Cipher

Vigenère Cipher is a substitution cipher that uses a keyword. Each letter in the plaintext is shifted based on the corresponding letter in the keyword.
It is stronger than the Caesar Cipher because it uses multiple shifts instead of one.

#### Encryption:

It uses a keyword to shift each letter by an amount determined by the key.

```
Algorithm VigenereCipher(plaintext, key)
1. key_length = length(key)
2. for i = 0 to length(plaintext)-1:
3.     shift = (key[i mod key_length] - 'A')
4.     ciphertext[i] = (plaintext[i] + shift) mod 26
5. return ciphertext
```

#### Decryption:

Decryption subtracts the same shift value to get the original text.

```
Algorithm VigenereDecrypt(ciphertext, key)
1. key_length = length(key)
2. for i = 0 to length(ciphertext)-1:
3.     shift = (key[i mod key_length] - 'A')
4.     plaintext[i] = (ciphertext[i] - shift + 26) mod 26
5. return plaintext
```

#### Complexity Analysis:

Time Complexity: O(n)
Space Complexity: **O(n)** total / **O(1)** auxiliary

### 4. Rail Fence Cipher

Rail Fence Cipher is a transposition cipher. The message is written diagonally across multiple rails (lines) and then read row by row.
It changes the position of characters but not the characters themselves.

### 🔐 Encryption Steps

1. Create an empty matrix with rails rows and len(plaintext) columns.
2. Initialize direction (down or up) and start from the top rail.
3. For each character in plaintext:
    - Place the character in the current row and column.
    - Change direction (down ↔ up) when reaching the top or bottom rail.
4. After filling the matrix in a zigzag manner, read all characters **row by row** to get the ciphertext.
5. Return the ciphertext.

Code:

```
def encrypt_rail_fence(plaintext, rails):

   # Create the matrix for cipher
   rail_matrix = [['\n' for i in range(len(plaintext))]
                  for j in range(rails)]

   # Find the direction
   down_direction = False
   row, col = 0, 0

   for i in range(len(plaintext)):

      # Check the direction of flow
      if (row == 0) or (row == rails - 1):
         down_direction = not down_direction

      # Fill the corresponding alphabet
      rail_matrix[row][col] = plaintext[i]
      col += 1
```

```
        # Find the next row using direction flag
        if down_direction:
            row += 1
        else:
            row -= 1

    # Construct the cipher using the rail matrix
    cipher_text = []
    for i in range(rails):
        for j in range(len(plaintext)):
            if rail_matrix[i][j] != '\n':
                cipher_text.append(rail_matrix[i][j])
    return "".join(cipher_text)
```

🔐 Encryption Complexity

- Time Complexity: O(n)
  → Each character of the plaintext is placed once into the matrix and later read once.
- Space Complexity: O(n)
  → A matrix of size rails × n is used to store the zigzag pattern.

🔓 **Decryption Steps**

1. Create an empty matrix with rails rows and len(cipher) columns.
2. Mark zigzag positions (*) where letters will be placed (based on the rail pattern).
3. Fill these marked positions with letters from the ciphertext (row by row).
4. Read the matrix **in zigzag order** (top to bottom, changing direction at top/bottom) to reconstruct the plaintext.
5. Return the plaintext.

Code:

```
def decrypt_rail_fence(ciphertext, rails):
    rail_matrix = [['\n' for i in range(len(ciphertext))]
                   for j in range(rails)]
```

```
down_direction = None
row, col = 0, 0

# Step 1: Mark the places where characters will be placed
for i in range(len(ciphertext)):
   if row == 0:
      down_direction = True
   if row == rails - 1:
      down_direction = False
   rail_matrix[row][col] = '*'
   col += 1
   if down_direction:
      row += 1
   else:
      row -= 1

# Step 2: Fill the marked positions with ciphertext letters
index = 0
for i in range(rails):
   for j in range(len(ciphertext)):
      if (rail_matrix[i][j] == '*') and (index < len(ciphertext)):
         rail_matrix[i][j] = ciphertext[index]
         index += 1

# Step 3: Read the matrix in zigzag manner to get the plaintext
result = []
row, col = 0, 0
for i in range(len(ciphertext)):
   if row == 0:
      down_direction = True
   if row == rails - 1:
      down_direction = False
   if (rail_matrix[row][col] != '\n'):
      result.append(rail_matrix[row][col])
      col += 1
   if down_direction:
      row += 1
   else:
      row -= 1
return "".join(result)
```

🔓 **Decryption Complexity**

- **Time Complexity: O(n)**
  → Each character is processed a few times (marked, filled, and read), but all are linear in the length of the ciphertext.
- **Space Complexity: O(n)**
  → Same matrix of size rails × n is needed to reconstruct the zigzag pattern.

### 🧩 1. What happens to spaces during encryption?

**In the Rail Fence Cipher, every character in the plaintext — including spaces, punctuation, or digits — is treated just like a normal character.**

**So if your plaintext is:**

**HELLO WORLD**

**It will include the space while filling the zigzag pattern, for example (2 rails):**

**H L O O L**

**E L W R D**

**Ciphertext (reading row by row) = HLOOL ELWRD**

✅ **The space is included in its zigzag position and encrypted normally (not removed or changed).**

---

### 🧩 2. During decryption

**Since encryption didn't remove the spaces —**
**when you reconstruct the zigzag pattern in decryption (steps 1–3 as before),**
**the space automatically returns to its correct position.**

**So decryption output = HELLO WORLD**

✅ **No special handling needed — as long as encryption and decryption both use the same number of rails,**
**the space will return exactly where it was.**

---

### 🧠 Important note:

**If you remove spaces before encryption, then of course decryption can't restore them —**

**because the cipher doesn’t know where they were.**
**So normally, you keep spaces in the plaintext for correct reversible encryption.**