

**LAB FILE**

**NAME: MEHAK FATIMA**

**SEAT NUMBER: B21110006057**

**CLASS: BSCS-A (FINAL YEAR)**

**COURSE CODE: 631**

**SUBJECT: NETWORK SECURITY AND CRYPTOGRAPHY**

**TEACHER: SIR BARI**

**LAB 1 (CEASER CIPHER)**

**OBJECTIVE:**

To implement and understand the Caesar Cipher encryption and decryption technique by shifting alphabetic characters by a fixed key value.

**CAESAR CIPHER:**

It is a simple substitution cipher that shifts each letter in the plaintext by a fixed number of positions (3) in the alphabet.

**WORKING OF CAESAR CIPHER:**

The Caesar Cipher works by shifting each letter in the plaintext by a fixed number of positions (shift = 3 in this case). If encrypting, the letters shift forward, and if decrypting, they shift backward. Non-alphabet characters remain unchanged.

**FORMULA:**

The formula used is:

newCode = ((charCode − offset ± shift+26) % 26) + offset

This ensures that the transformation stays within the range of alphabetic characters.

**CODE:**

const [text, setText] = useState("Hello World!");

const [mode, setMode] = useState("encrypt");

const [result, setResult] = useState("");

const shift = 3;

const caesarCipher = (str, shift, encrypt = true) => {

let result = "";

for (let i = 0; i < str.length; i++) {

let char = str[i];

if (str[i].match(/[a-z]/i)) {

let code = str.charCodeAt(i);

let offset = code >= 65 && code <= 90 ? 65 : 97; // check if it is capital or small letter

let newCode = ((code - offset + (encrypt ? shift : -shift) + 26) % 26) + offset;

// (code - offset to bring an index of that value means 'A' is 0)

// (encrypt ? if yes then add 3 value otherwise decrypt means subtract 3 value)

// (+26 to remain the positive value while decrypting)

// (%26 to be in the range of alphabets)

result += String.fromCharCode(newCode);

} else {

result += char;

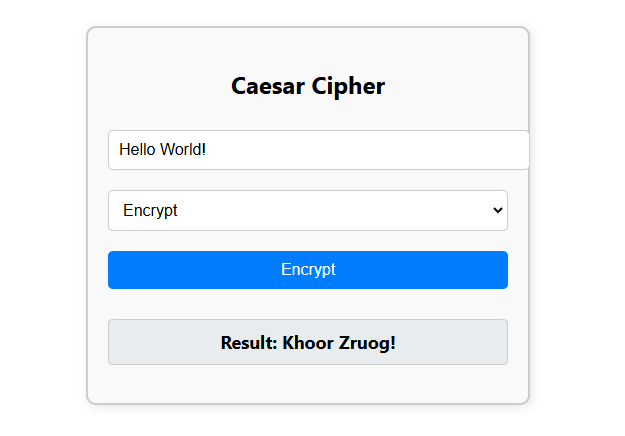
}

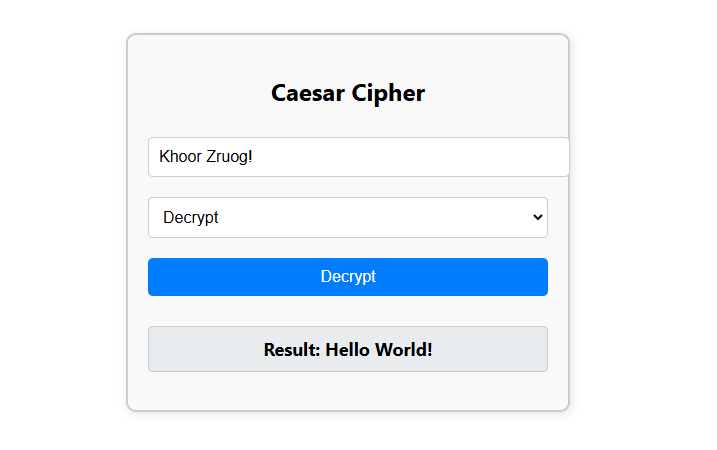
}

return result;

};

**USER INTERFACE:**

**ENCRYTION:**

 **DECRYPTION:**

**LAB 2 (ONE-TIME PAD (OTP) CIPHER)**

**OBJECTIVE:**

To implement and understand the One-Time Pad (OTP) Cipher, a cryptographic technique that uses a random key equal in length to the plaintext for perfect secrecy.

**ONE-TIME PAD (OTP) CIPHER:**The **One-Time Pad (OTP) Cipher** is a symmetric encryption technique that uses a completely random key of the same length as the plaintext. Each character of the plaintext is XORed with the corresponding key character, making the encryption theoretically **unbreakable** if the key is truly random, used only once, and kept secret.

**WORKING:**

The One-Time Pad (OTP) Cipher is a symmetric encryption technique that ensures absolute security when used correctly. It works by generating a completely random key of the same length as the plaintext. Each character of the plaintext is combined with the corresponding character of the key using bitwise XOR. This results in a ciphertext that appears completely random.

For decryption, the same key is used again, and the ciphertext is processed using the reverse operation, which restores the original plaintext. Since the key is truly random and used only once, OTP provides perfect secrecy. However, the major challenge is securely distributing and managing the key.

**CODE:**

const encrypt = () => {

if (!plainText) return;

let generatedKey = key || generateRandomKey(plainText.length);

setKey(generatedKey);

let encryptedText = "";

for (let i = 0; i < plainText.length; i++) {

let pBinary = charToBinary(plainText[i]);

let kBinary = charToBinary(generatedKey[i]);

let cBinary = xorStrings(pBinary, kBinary);

encryptedText += binaryToChar(cBinary);

}

setCipherText(encryptedText); // Save cipher text

setResult(`Encrypted Text: ${encryptedText}`);

};

const decrypt = () => {

if (!cipherText || !key) return;

let decryptedText = "";

for (let i = 0; i < cipherText.length; i++) {

let cBinary = charToBinary(cipherText[i]);

let kBinary = charToBinary(key[i]);

let pBinary = xorStrings(cBinary, kBinary);

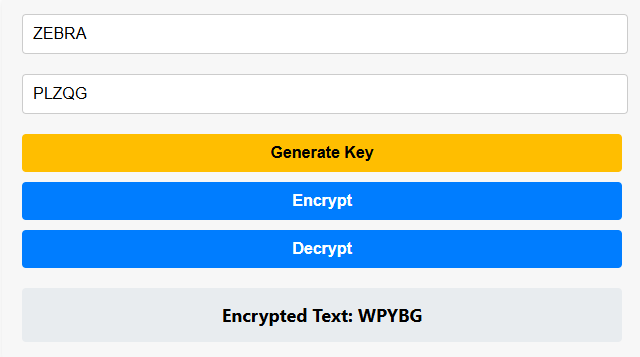
decryptedText += binaryToChar(pBinary);

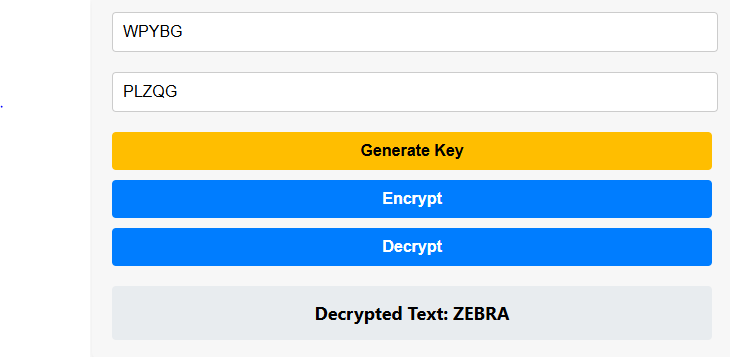
}

setResult(`Decrypted Text: ${decryptedText}`);

};

**UI:**

**ENCRYPTION:**

**DECRYPTION:**

**LAB 3 (RAIL FENCE CIPHER)**

**OBJECTIVE:**

To implement and understand the Rail Fence Cipher encryption and decryption technique by arranging characters in a zigzag pattern across a fixed number of rails and reading them sequentially to transform the plaintext into ciphertext and vice versa.

**RAIL FENCE CIPHER:**

The **Rail Fence Cipher** is a type of transposition cipher where the plaintext is written in a zigzag pattern across multiple rails (or rows) and then read row-wise to create the ciphertext. It rearranges the characters without changing them, making it a simple yet effective encryption method.

**WORKING:**

The Rail Fence Cipher encrypts a message by writing it diagonally in a zigzag pattern across a specified number of rows (rails). Once the entire message is written, the ciphertext is obtained by reading row-wise from top to bottom. For decryption, the rails are reconstructed based on the zigzag pattern, and the message is read in its original sequence to retrieve the plaintext.

**CODE:**

const encryptFenceCipher = (text, depth) => {

if (depth < 2 || !text) return text;

// remove spaces

let filteredText = '';

for (let i = 0; i < text.length; i++) {

if (text[i] !== ' ') {

filteredText += text[i];

}

}

//depth number of arrays

const rails = [];

for (let i = 0; i < depth; i++) {

rails[i] = [];

}

// distribute characters into arrays

let index = 0;

for (let i = 0; i < filteredText.length; i++) {

rails[index].push(filteredText[i]);

index++;

if (index >= depth) index = 0; // reset index to loop through arrays from start again

}

// concatenate arrays manually

let encryptedText = '';

for (let i = 0; i < depth; i++) {

for (let j = 0; j < rails[i].length; j++) {

encryptedText += rails[i][j];

}

}

return encryptedText;

};

const decryptFenceCipher = (cipherText, depth) => {

if (depth < 2 || !cipherText) return cipherText;

// depth number of arrays manually

const rails = [];

for (let i = 0; i < depth; i++) {

rails[i] = [];

}

// Fill the arrays with placeholder elements

let index = 0;

for (let i = 0; i < cipherText.length; i++) {

rails[index].push(null); // Allocate space

index++;

if (index >= depth) index = 0;

}

// Manually fill the arrays with characters from the cipher text

let charIndex = 0;

for (let i = 0; i < depth; i++) {

for (let j = 0; j < rails[i].length; j++) {

rails[i][j] = cipherText[charIndex];

charIndex++;

}

}

// Read characters sequentially from arrays to decrypt

let decryptedText = '';

index = 0;

for (let i = 0; i < cipherText.length; i++) {

decryptedText += rails[index][0];

// Shift manually by creating a new array without the first element

const newArray = [];

for (let j = 1; j < rails[index].length; j++) {

newArray[j - 1] = rails[index][j];

}

rails[index] = newArray;

index++;

if (index >= depth) index = 0;

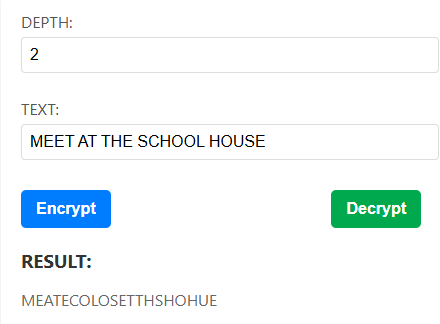
}

return decryptedText;

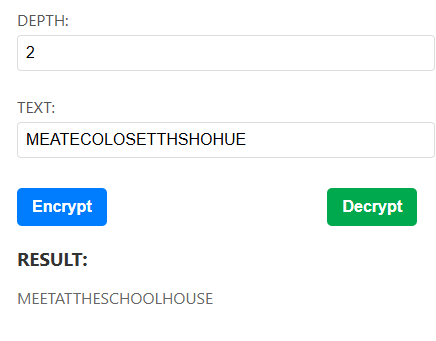
};

**UI:**

**ENCRYPTION:**



**DECRYPTION:**



**LAB 4 (COLUMNAR TRANSPOSITION CIPHER)**

**OBJECTIVE:**

To implement and understand the Columnar Transposition Cipher encryption and decryption technique by rearranging the characters of a message into a grid and transposing columns based on a secret key.

**COLUMNAR TRANSPOSITION CIPHER:**

The **Columnar Transposition Cipher** is a classical encryption method that rearranges the characters of a message based on a predetermined key. Instead of substituting letters, it transposes them by writing the message in a grid and then reading the columns in a specific order determined by the key.

**WORKING:**

The Columnar Transposition Cipher encrypts a message by writing it into a grid row-wise based on the length of a secret key. If needed, padding characters are added to fill empty spaces. The columns are then rearranged according to the alphabetical order of the key. The ciphertext is obtained by reading the grid column-wise in the new order. For decryption, the process is reversed: the ciphertext is placed into a grid column-wise using the key order, then rearranged back to the original sequence, and finally read row-wise to retrieve the plaintext.

**CODE:**

const [inputText, setInputText] = useState("");

const [key, setKey] = useState("");

const [result, setResult] = useState("");

const encryptText = (text, keyVal) => {

if (!text || !keyVal) return "";

let encryptedText = "";

let keyIndex = 0;

const textArray = Array.from(text);

const sortedKey = Array.from(keyVal).sort(); // to mark the columns

const columns = keyVal.length; // to get the number of columns from the key

const rows = Math.ceil(text.length / columns); // calculate number of rows

const paddingNeeded = rows \* columns - text.length;

for (let i = 0; i < paddingNeeded; i++) { // to make the length equal to the total cells

textArray.push("\_");

}

const matrix = [];

for (let i = 0; i < textArray.length; i += columns) {

matrix.push(textArray.slice(i, i + columns));

}

for (let \_ = 0; \_ < columns; \_++) {

const currentIndex = keyVal.indexOf(sortedKey[keyIndex]);

for (const row of matrix) {

encryptedText += row[currentIndex];

}

keyIndex++;

}

return encryptedText;

};

const decryptText = (cipher, keyVal) => {

if (!cipher || !keyVal) return "";

let decryptedText = "";

let keyIndex = 0;

let textIndex = 0;

const columns = keyVal.length; // calculate the length

const rows = Math.ceil(cipher.length / columns); // to calculate the rows

const sortedKey = Array.from(keyVal).sort();

const decryptedMatrix = Array.from({ length: rows }, () =>

Array(columns).fill(null)

);

for (let \_ = 0; \_ < columns; \_++) {

const currentIndex = keyVal.indexOf(sortedKey[keyIndex]);

for (let row = 0; row < rows; row++) {

decryptedMatrix[row][currentIndex] = cipher[textIndex++];

}

keyIndex++;

}

try {

decryptedText = decryptedMatrix.flat().join("");

} catch (error) {

throw new Error("Error processing repeated words.");

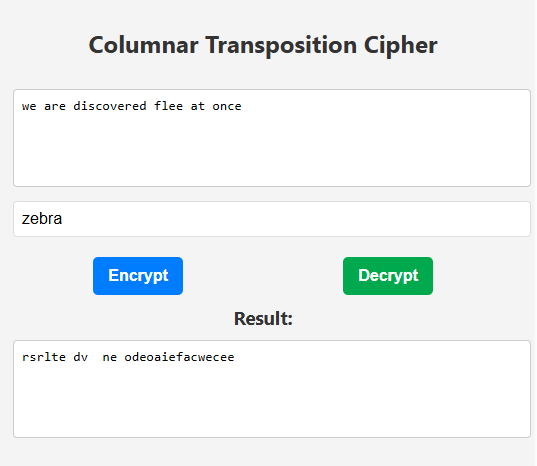
}

return decryptedText.replace(/\_+$/, ""); // Remove padding underscores

};

**UI:**

**ENCRYPTION:**



**DECRYPTION:**

