****

**Institute of Engineering & Management,**

**University of Engineering & Management**

**Kolkata**

**Department of Computer Science & Engineering**

**Laboratory: - Introduction to Cyber Security Lab**

**Name: - Gourab Mondal**

**Section: - B**

**Roll Number: - 26**

**Enrollment Number: - 12022002001116**

**Year: - 3rd**

****

**INDEX PAGE**

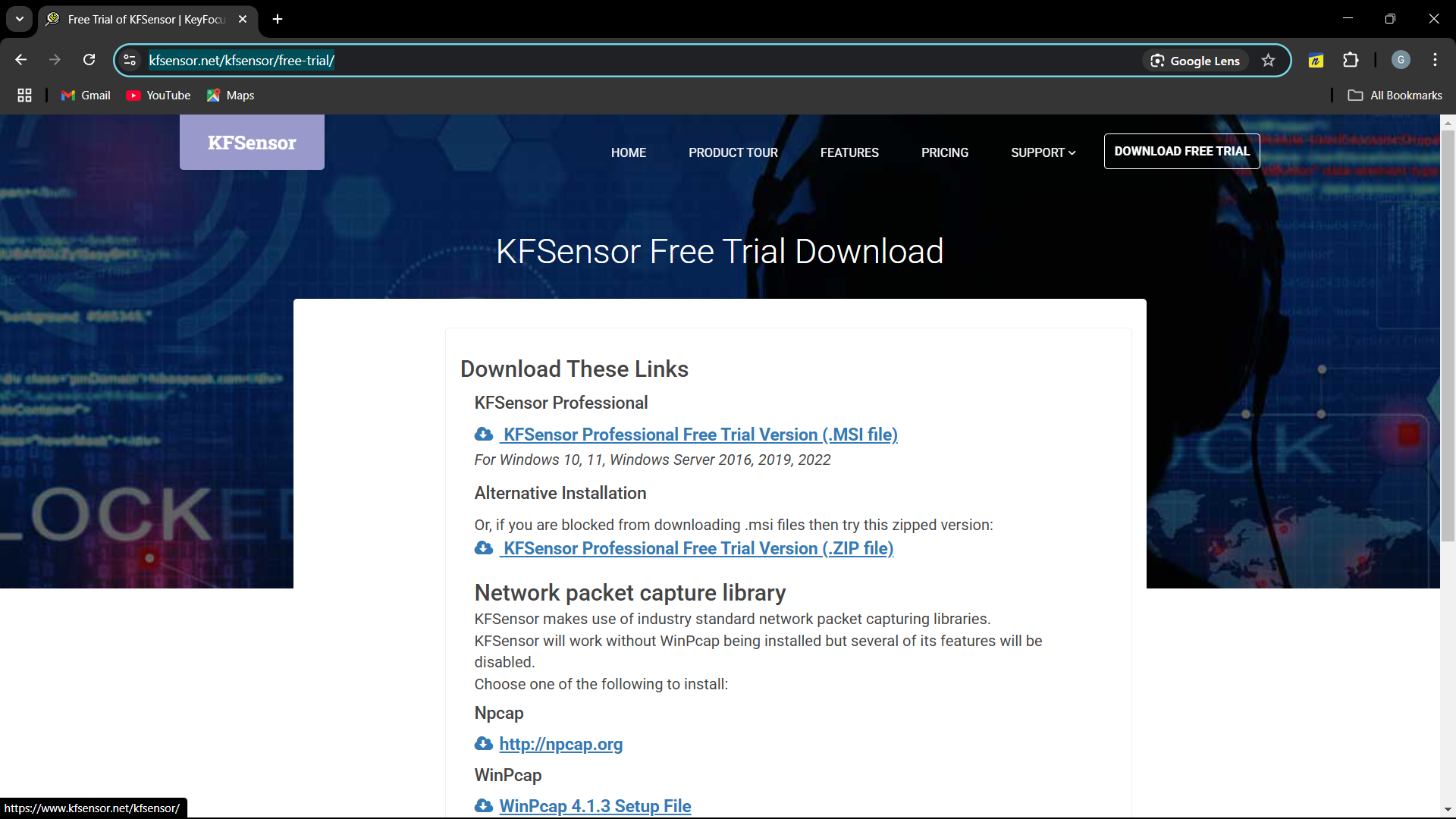
|  |  |  |
| --- | --- | --- |
| **SL NO.** | **ASSIGNMENT** | **PAGE NUMBER** |
| **1** | **Week 1: Week 1: Implementation of Caesar cypher and Playfair cypher using java** | **03 - 11** |
| **2** | **Week 2: Installation of KFSensor and Npcap, Setting up honeypot, java implementation of Hill Cipher** | **12 - 23** |
| **3** |  |  |
| **4** |  |  |
| **5** |  |  |
| **6** |  |  |
| **7** |  |  |
| **8** |  |  |
| **9** |  |  |
| **10** |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

**Week 2**

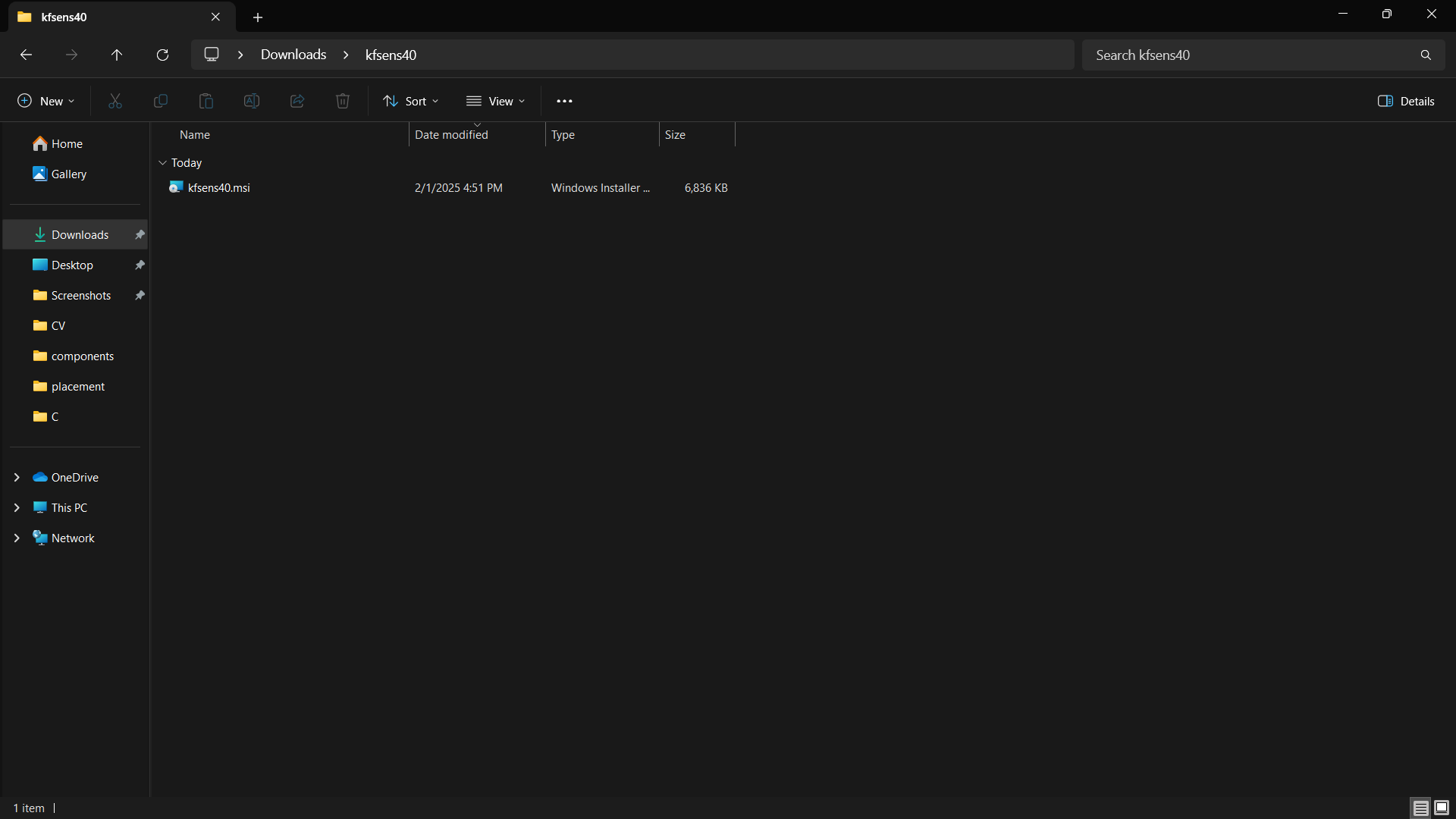
**1a.**

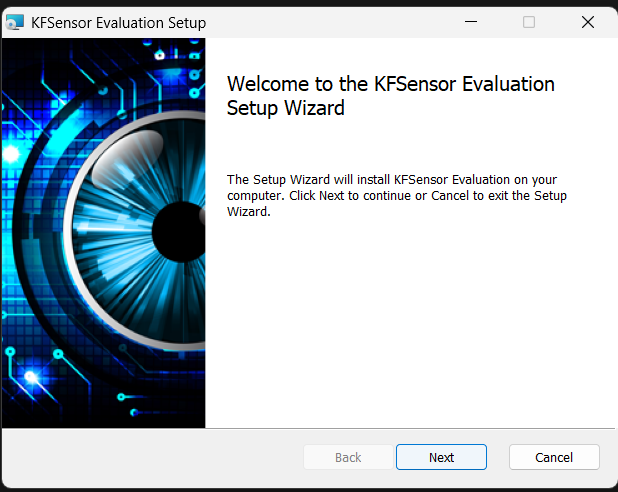
**Title:** Installation of **Npcap** and **KFSensor** and setting up **honey pot.**

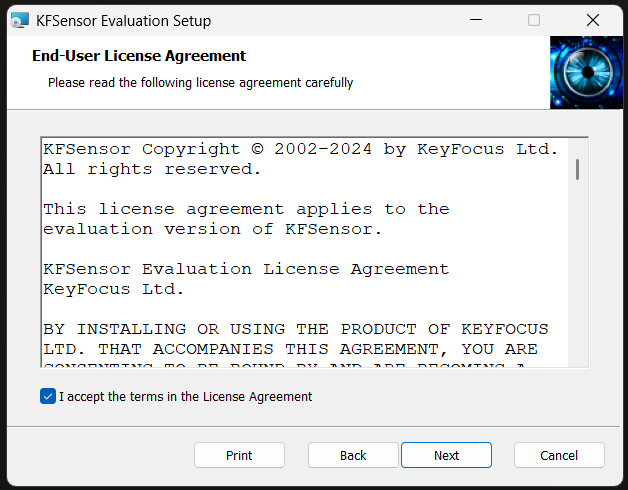
**Step 1:** Visit the link <https://www.kfsensor.net/kfsensor/free-trial/>

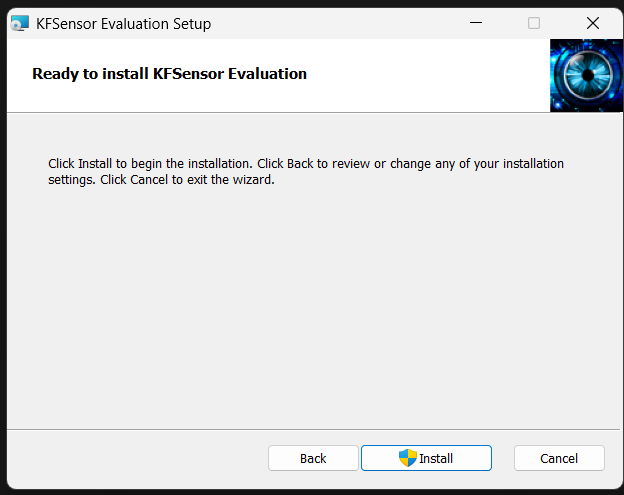
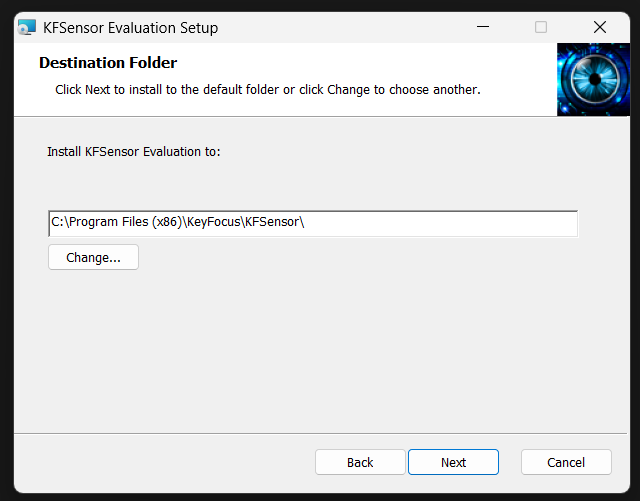
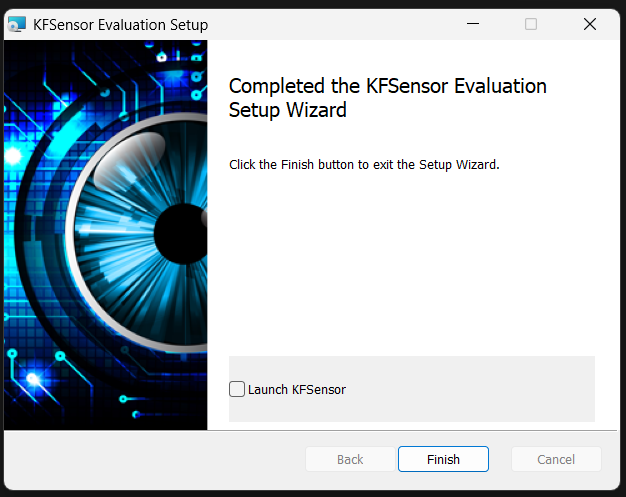


**Step 2:** Install the free trial version of KFSensor in ZIP format and extract in explorer

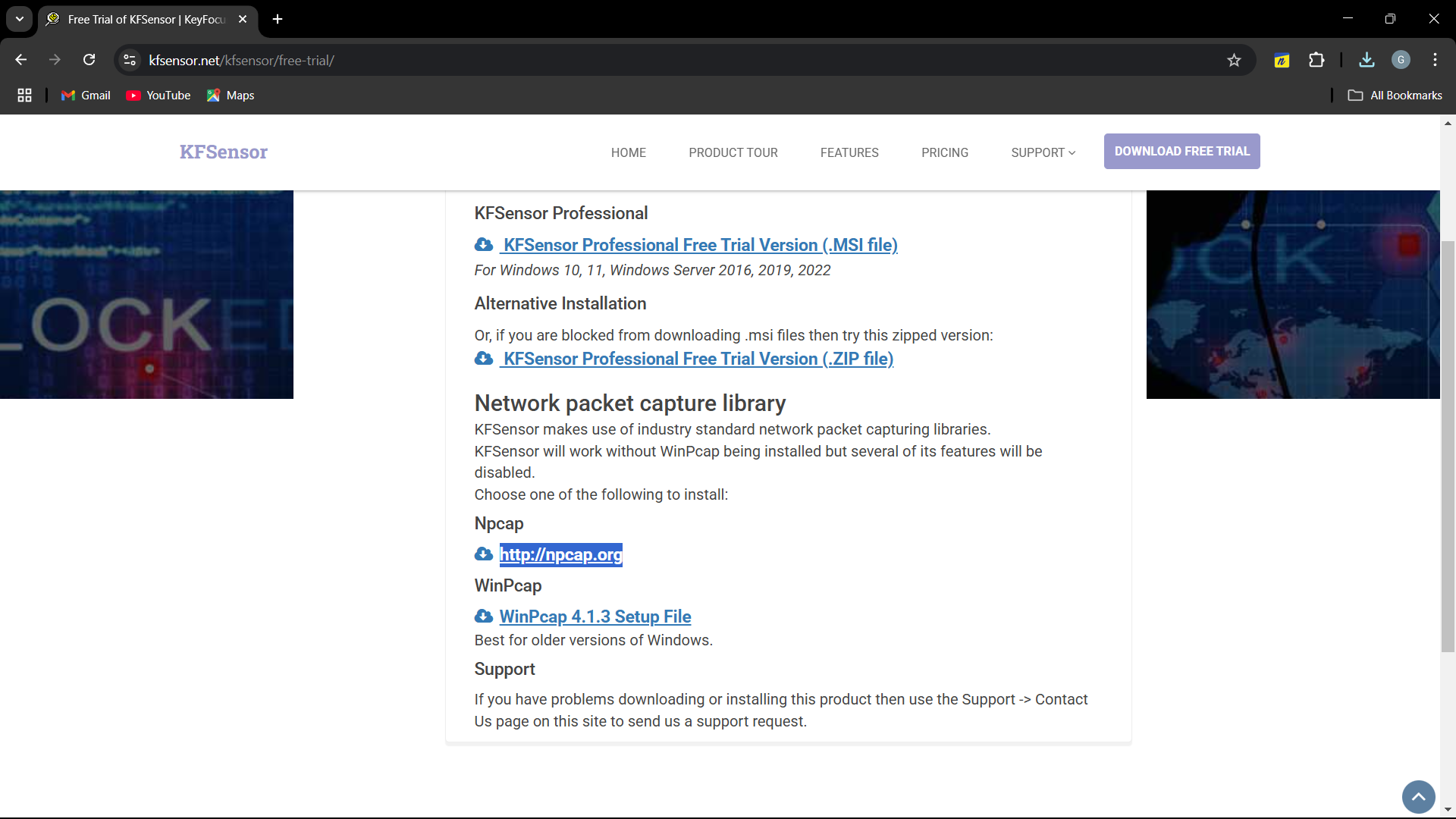


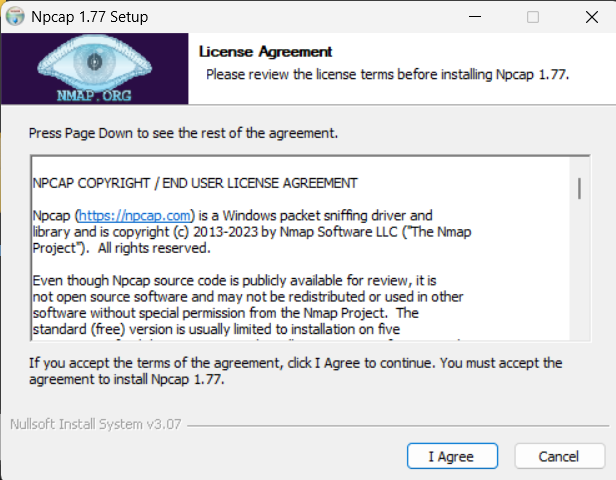
**Step 3:** Run the msi executable and follow below steps

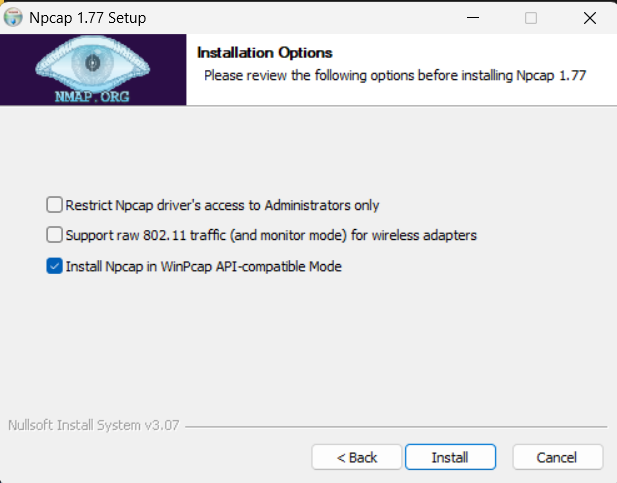


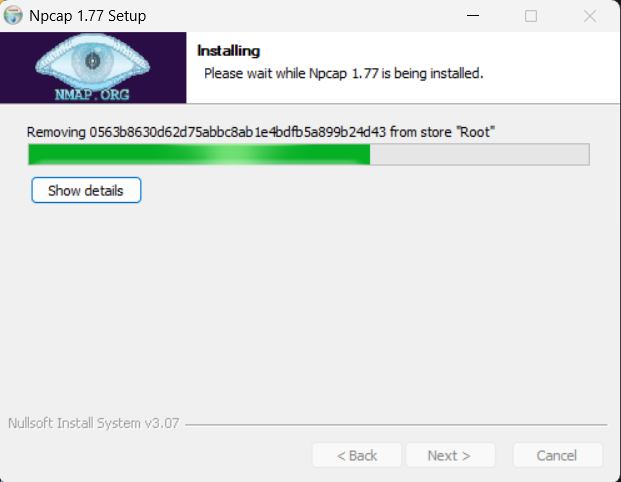
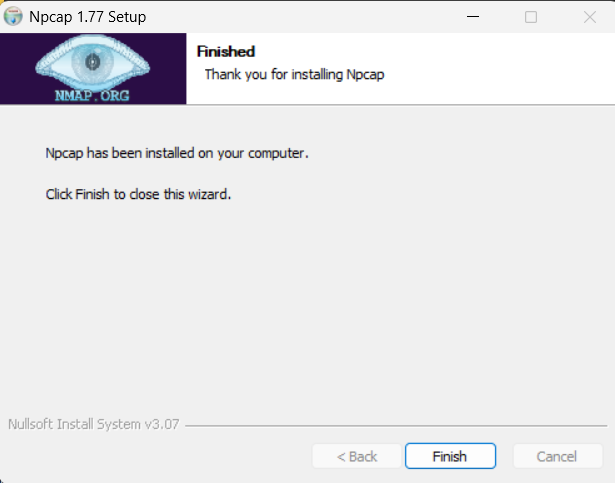


**Step 4:** From <https://www.kfsensor.net/kfsensor/free-trial/> also visit the link for Npcap

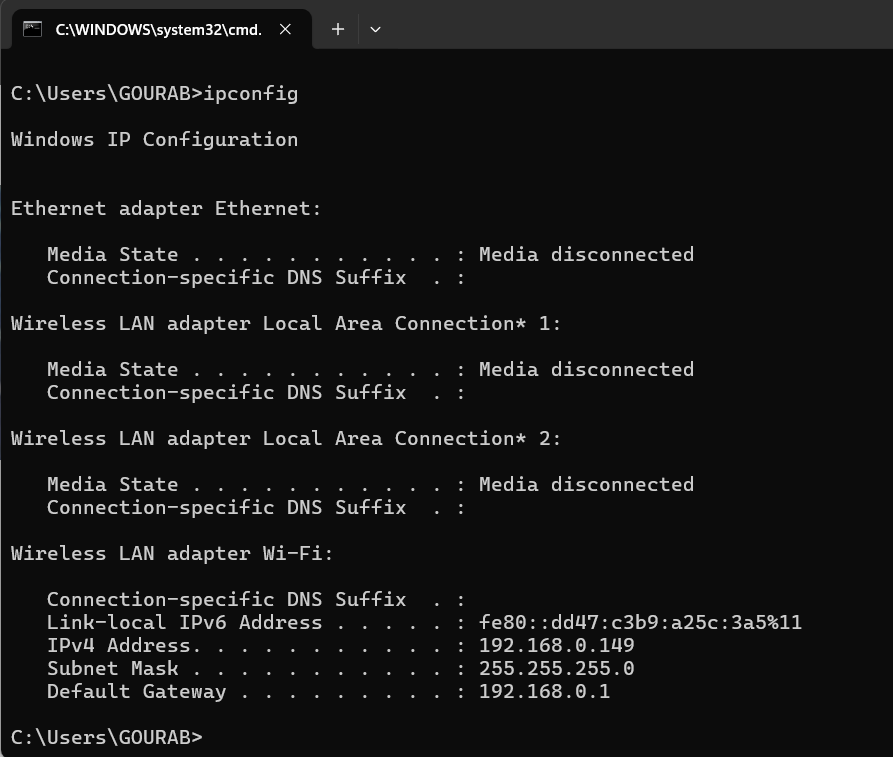


**Step 5:** Run the recently installed Npcap setup exe file and follow below steps

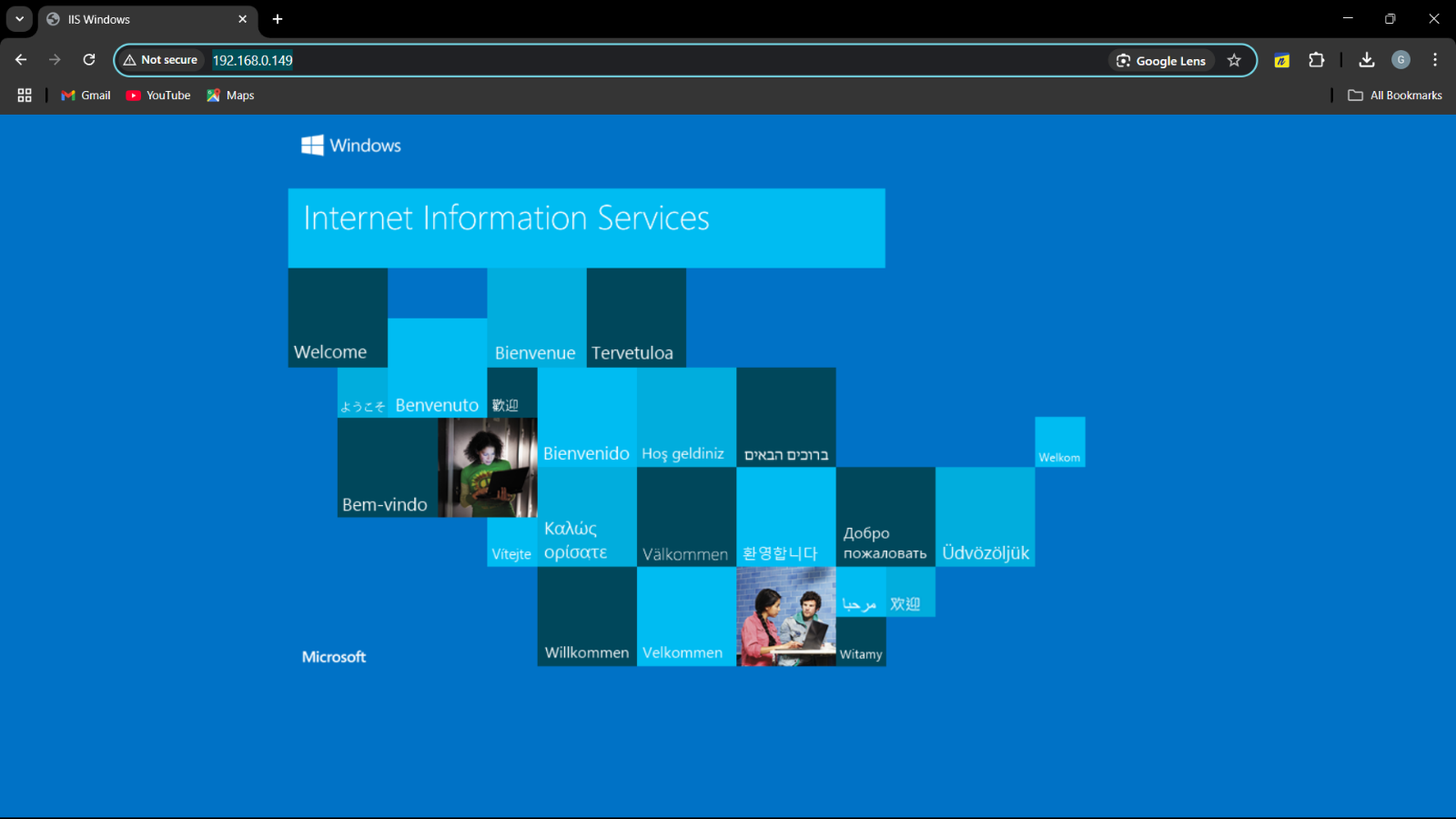




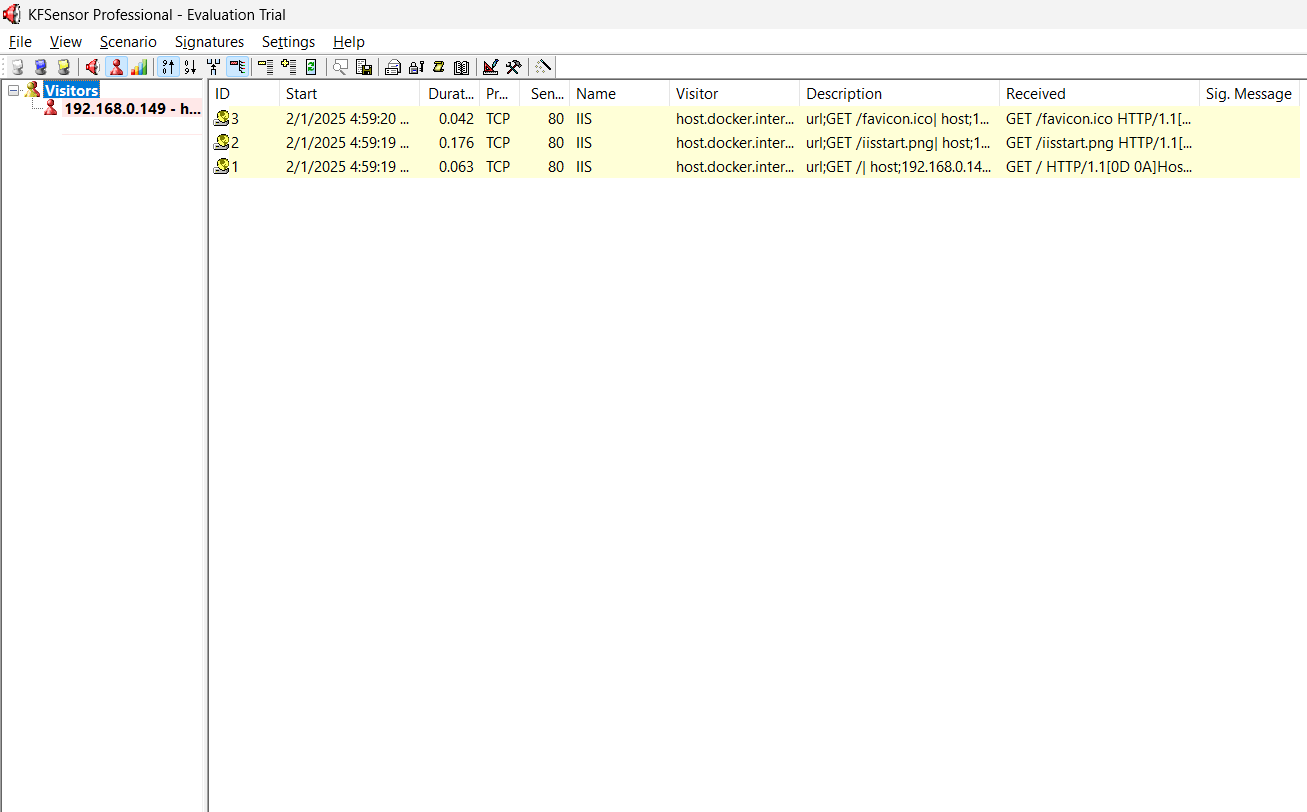
**Step 6:** Run CMD and execute command “ipconfig” copy the ipv4 address



**Step 7:** Launch kfsensor and paste the ipv4 address in browser



**Step 8:** Check the logs in kfsensor for recent http request in visitors tab



**1b.**

**Title:** Implementation of Hill Cipher using java

**Algorithm:**

 **Key Preparation:**

* Choose a key of length n×n (3x3 for this implementation).
* Ensure the key has no non-alphabetic characters and is of the required length.
* Convert the key to lowercase.

 **Matrix Creation (for encryption):**

* Convert the key into a matrix of numbers (0-25 for letters 'a' to 'z').

 **Plaintext Preparation:**

* Remove any non-alphabetic characters and convert to lowercase.
* If the plaintext length is not a multiple of n, add padding (commonly 'x').

 **Encryption:**

* Break the plaintext into digraphs (pairs of n characters).
* For each digraph, convert each character to its numerical equivalent.
* Multiply the key matrix with the digraph matrix (mod 26).
* Convert the result back to characters to form the ciphertext.

 **Matrix Inversion (for decryption):**

* Calculate the determinant of the key matrix (mod 26).
* Find the modular inverse of the determinant.
* Compute the adjugate matrix of the key matrix.
* Multiply the adjugate matrix by the modular inverse of the determinant (mod 26) to obtain the inverse matrix.

 **Decryption:**

* For each ciphertext digraph, convert each character to its numerical equivalent.
* Multiply the inverse key matrix with the ciphertext matrix (mod 26).
* Convert the result back to characters to recover the plaintext.

**Java Program:**

// basic implementation of hill cipher for only 3\*3 key matrix

class HillCipher {

    private static final int KEYSIZE = 3;

    private int[][] keyMat = new int[KEYSIZE][KEYSIZE];

    private String key;

    private static final Map<Character, Integer> CHAR\_TO\_NUM = new HashMap<>();

    private static final Map<Integer, Character> NUM\_TO\_CHAR = new HashMap<>();

    static {

        for (char i = 'a'; i <= 'z'; i++) {

            CHAR\_TO\_NUM.put(i, i - 'a');

        }

        for (char i = 'a'; i <= 'z'; i++) {

            NUM\_TO\_CHAR.put(i - 'a', i);

        }

    }

    HillCipher(String key) {

        this.key = key;

        this.validateAndFormatKey();

        this.createKeyMat();

    }

    private void validateAndFormatKey() {

        this.key = this.key.replaceAll("[^a-zA-Z]", "");

        if (this.key.length() != KEYSIZE \* KEYSIZE) {

            throw new IllegalArgumentException("Key must contain exactly 9 alphabetic characters for a 3x3 matrix.");

        }

        this.key = this.key.toLowerCase();

    }

    private int getNumVal(char c) {

        return CHAR\_TO\_NUM.getOrDefault(c, -1);

    }

    public char getCharVal(int n) {

        return NUM\_TO\_CHAR.getOrDefault(n, '\_');

    }

    private void createKeyMat() {

        int k = 0;

        for (int i = 0; i < KEYSIZE; i++) {

            for (int j = 0; j < KEYSIZE; j++) {

                this.keyMat[i][j] = this.getNumVal(this.key.charAt(k++));

            }

        }

    }

    private String addPadding(String text) {

        if (text.length() % KEYSIZE == 0) {

            return text;

        }

        StringBuilder sb = new StringBuilder(text);

        while (sb.length() % KEYSIZE != 0) {

            sb.append('x');

        }

        return sb.toString();

    }

    private String formatText(String text) {

        if (!text.matches("[a-zA-Z]+")) {

            throw new IllegalArgumentException("text must contain only alphabetic characters.");

        }

        text = text.toLowerCase();

        text = this.addPadding(text);

        return text;

    }

    private List<int[]> createDigraphs(String plainText) {

        List<int[]> digraphs = new ArrayList<>();

        for (int j = 0; j < plainText.length(); j += KEYSIZE) {

            int[] digraph = new int[KEYSIZE];

            for (int k = 0; k < KEYSIZE; k++) {

                digraph[k] = this.getNumVal(plainText.charAt(j + k));

            }

            digraphs.add(digraph);

        }

        return digraphs;

    }

    private int[] multiplyMatrices(int[][] keyMat, int[] digraph) {

        int[] result = new int[KEYSIZE];

        for (int i = 0; i < KEYSIZE; i++) {

            result[i] = 0;

            for (int j = 0; j < KEYSIZE; j++) {

                result[i] += keyMat[i][j] \* digraph[j];

            }

            result[i] = result[i] % 26;

        }

        return result;

    }

    private String getCipher(int[] digraph) {

        int[] cipher = this.multiplyMatrices(this.keyMat, digraph);

        String cipherText = "";

        for (int i = 0; i < cipher.length; i++) {

            cipherText = cipherText.concat(String.valueOf(this.getCharVal(cipher[i])));

        }

        return cipherText;

    }

    public String encrypt(String plainText) {

        String cipherText = "";

        plainText = this.formatText(plainText);

        List<int[]> digraphs = this.createDigraphs(plainText);

        for (int[] digraph : digraphs) {

            cipherText = cipherText + this.getCipher(digraph);

        }

        return cipherText;

    }

    private int determinant(int[][] matrix) {

        return (matrix[0][0] \* (matrix[1][1] \* matrix[2][2] - matrix[1][2] \* matrix[2][1]) -

                matrix[0][1] \* (matrix[1][0] \* matrix[2][2] - matrix[1][2] \* matrix[2][0]) +

                matrix[0][2] \* (matrix[1][0] \* matrix[2][1] - matrix[1][1] \* matrix[2][0])) % 26;

    }

    private int modInverse(int det, int mod) {

        det = (det % mod + mod) % mod;

        for (int i = 1; i < mod; i++) {

            if ((det \* i) % mod == 1)

                return i;

        }

        throw new ArithmeticException("No modular inverse exists!");

    }

    private int[][] invertMatrix(int[][] matrix) {

        int det = determinant(matrix);

        int detInverse = modInverse(det, 26);

        int[][] adj = new int[3][3];

        adj[0][0] = (matrix[1][1] \* matrix[2][2] - matrix[1][2] \* matrix[2][1]) % 26;

        adj[0][1] = (matrix[0][2] \* matrix[2][1] - matrix[0][1] \* matrix[2][2]) % 26;

        adj[0][2] = (matrix[0][1] \* matrix[1][2] - matrix[0][2] \* matrix[1][1]) % 26;

        adj[1][0] = (matrix[1][2] \* matrix[2][0] - matrix[1][0] \* matrix[2][2]) % 26;

        adj[1][1] = (matrix[0][0] \* matrix[2][2] - matrix[0][2] \* matrix[2][0]) % 26;

        adj[1][2] = (matrix[0][2] \* matrix[1][0] - matrix[0][0] \* matrix[1][2]) % 26;

        adj[2][0] = (matrix[1][0] \* matrix[2][1] - matrix[1][1] \* matrix[2][0]) % 26;

        adj[2][1] = (matrix[0][1] \* matrix[2][0] - matrix[0][0] \* matrix[2][1]) % 26;

        adj[2][2] = (matrix[0][0] \* matrix[1][1] - matrix[0][1] \* matrix[1][0]) % 26;

        for (int i = 0; i < 3; i++) {

            for (int j = 0; j < 3; j++) {

                adj[i][j] = (adj[i][j] \* detInverse) % 26;

                if (adj[i][j] < 0)

                    adj[i][j] += 26;

            }

        }

        return adj;

    }

    private String getDecipher(int[] digraph) {

        int[][] inverseKeyMat = this.invertMatrix(this.keyMat);

        int[] deCipher = this.multiplyMatrices(inverseKeyMat, digraph);

        StringBuilder deCipherText = new StringBuilder();

        for (int num : deCipher) {

            deCipherText.append(this.getCharVal(num));

        }

        return deCipherText.toString();

    }

    public String decrypt(String cipherText) {

        StringBuilder plainText = new StringBuilder();

        cipherText = this.formatText(cipherText);

        List<int[]> digraphs = this.createDigraphs(cipherText);

        for (int[] digraph : digraphs) {

            plainText.append(this.getDecipher(digraph));

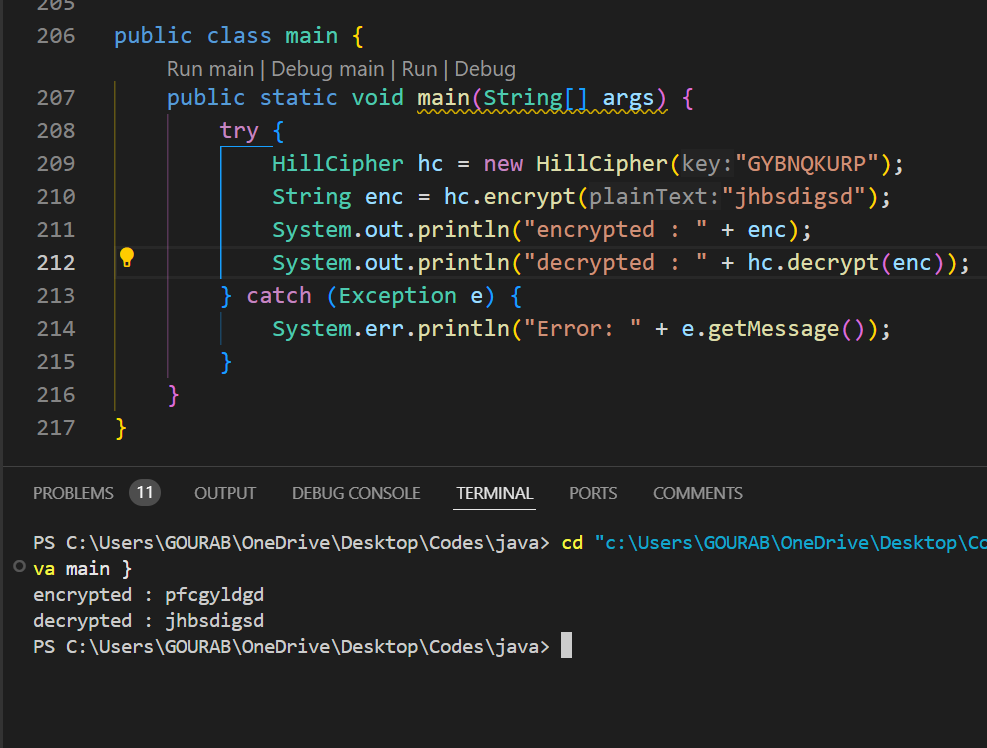
        }

        return plainText.toString();

    }

}

**Output:**

****