<u>Experiment No.</u> 01

<u>Experiment Name:</u> Write a program to encrypt and decrypt a message using Additive Cipher.

<u>Objectives:</u>

The aim of this experiment is to execute the additive cipher algorithm for both encryption and decryption purposes. Specifically, the objectives are to apply the additive cipher algorithm to encrypt a message and subsequently decrypt the encrypted message.

<u>Thoery:</u>

The **Additive Cipher**, also known as the **Caesar Cipher** (when the shift is 3), is a type of substitution cipher in which each letter in the plaintext is shifted by a certain number of positions down or up the alphabet. The number of positions by which the letters are shifted is called the **key**. The encryption and decryption processes are both based on modular arithmetic.

Key Concept:

The core idea of the additive cipher is that each letter of the alphabet is assigned a numerical value:

- A = 0, B = 1, C = 2, ..., Z = 25.

For encryption, the key is added to the numerical value of each letter of the plaintext to produce the ciphertext. Decryption is the reverse process, where the key is subtracted from the numerical value of each letter in the ciphertext to recover the plaintext.

Mathematical Representation:

Let:

- P be the plaintext letter (numerical value between 0 and 25),
- C be the ciphertext letter (numerical value between 0 and 25),
- K be the key (integer value between 0 and 25).

The encryption process is defined as:

$$C=(P+K) \bmod 26$$

The decryption process is defined as:

$$P=(C-K) \bmod 26$$

Properties:

- **Shift/Key (K)**: The key can be any integer from 0 to 25. If the key is 0, the ciphertext is the same as the plaintext (no encryption).

- **Modular Arithmetic**: Since the alphabet contains 26 letters, all operations are performed modulo 26. This ensures that values remain within the range of valid alphabetic characters.

<u>Algorithm:</u>

Encryption Algorithm:

```
1. Start.
2. Input the plaintext (message) from the user.
3. Convert the plaintext to lowercase and remove any spaces.
4. Take input the key (shift value) from the user.
5. For each character in the plaintext:
        i. Apply the formula: $C=(p+K)\bmod 26$. And convert it into character
        ii. Append the letter to Ciphertext.
6. Output the encrypted result (ciphertext).
7. End.
```

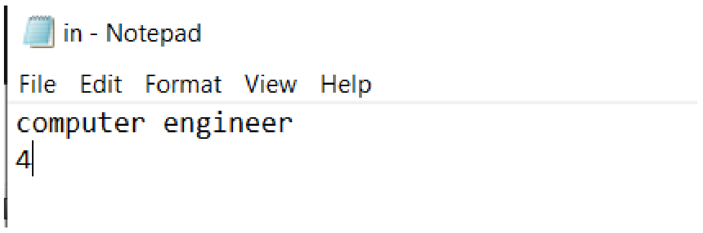
Decryption Algorithm:

```
1. Start.
2. Input the ciphertext (encrypted message).
3. For each character in the ciphertext
        i.. Apply the formula: $P=(c-K+26)\bmod 26$ (adding 26 ensures a positive result).and convert it into character
        ii. Append the letter to Plaintext.
4.Return Plaintext.
5.Print the plaintext
6.End.
```

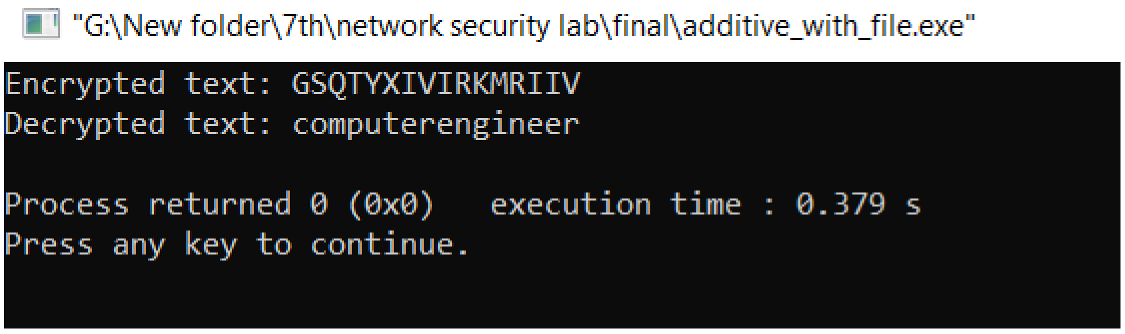
Program Code:

```
#include<bits/stdc++.h>
using namespace std;

string en(string p, int k){
    string ci;
    for(int i = 0; i < p.size(); i++) {
        if(isalpha(p[i])) {
            char c = toupper(p[i]);
            ci += (c + k - 'A') % 26 + 'A';
        }
    }
    return ci;
}

string de(string ci, int k){
    string pi;
    for(int i = 0; i < ci.size(); i++) {
        if(isalpha(ci[i])) {
            char c = tolower(ci[i]);
            pi += (c - k - 'a' + 26) % 26 + 'a';
        }
    }
    return pi;
}

int main() {
    string p, ci;
    int k;

    ifstream inputFile("in.txt");
    if (!inputFile) {
        cerr << "Error: Could not open the file!" << endl;
        return 1;
    }
    getline(inputFile, p);
    inputFile >> k;
    inputFile.close();

    ci = en(p, k);
    cout << "Encrypted text: " << ci << endl;

    string decryptedText = de(ci, k);
    cout << "Decrypted text: " << decryptedText << endl;

    return 0;}
```

Input and Output:



```
computer engineer
4
```


"G:\New folder\7th\network security lab\final\additive_with_file.exe"

```
Encrypted text: GSQTYXIVIRKMRIIV
Decrypted text: computerengineer

Process returned 0 (0x0)   execution time : 0.379 s
Press any key to continue.
```

Discussion :

The simplicity of the additive cipher makes it easy to understand and implement, showcasing how basic shifts can alter and restore information. However, its predictability and ease of decryption without a complex key structure highlight its limitations, rendering it ineffective for protecting sensitive data in real-world scenarios.

Experiment No.02

Experiment Name: Write a program to encrypt and decrypt a message using Multiplicative Cipher.

Objectives:

The objective of this experiment is to implement and execute the Multiplicative Cipher Algorithm for both encryption and decryption of a message. The goal is to apply the multiplicative cipher algorithm to encrypt a given message and subsequently decrypt the encrypted message.

Theory:

The Multiplicative Cipher is a type of substitution cipher where each letter in the plaintext is substituted with a letter that is a result of multiplying its numerical value by a fixed key, then taking the result modulo 26. This cipher is a form of modular arithmetic and is an example of an affine cipher where the encryption function is a linear function.

Key Concept:

Each letter of the alphabet is assigned a numerical value:

- A = 0, B = 1, C = 2, ..., Z = 25.

To encrypt, each letter in the plaintext is multiplied by a fixed integer key and then taken modulo 26 to ensure the result is within the range of valid alphabetic characters. To decrypt, the inverse of the key (modulo 26) is used to reverse the process.

Mathematical Representation:

Let:

- P be the plaintext letter (numerical value between 0 and 25),
- C be the ciphertext letter (numerical value between 0 and 25),
- K be the key (integer value between 0 and 25, which must be coprime with 26 for the cipher to work).

The encryption process is defined as:

$$C=(K\cdot P) \bmod 26$$

For decryption, the multiplicative inverse K^{-1} of the key K is used. The decryption process is:

$$P=(K^{-1}\cdot C) \bmod 26$$

Where K^{-1} is the modular inverse of K, which satisfies:

$(K \cdot K^{-1}) \bmod 26 = 1$

Algorithm:

Encryption Algorithm:

1. Start.
2. Input the plaintext (message) from the user.
3. Convert the plaintext to lowercase and remove any spaces.
4. Take input for the key (must be coprime with 26).
5. For each character in the plaintext:

    i. Convert the letter to its numerical value (0 to 25).

    ii. Apply the encryption formula: $C = (K \cdot P) \bmod 26$.

    iii. Convert the result back to a character and append it to the ciphertext.
6. Output the encrypted result (ciphertext).
7. End.

Decryption Algorithm:

1. Start.
2. Input the ciphertext (encrypted message).
3. Input the key used during encryption.
4. Find the multiplicative inverse K^{-1} of the key modulo 26.

    i. Use the Extended Euclidean Algorithm to compute K^{-1} such that $(K \cdot K^{-1}) \bmod 26 = 1$.
5. For each character in the ciphertext:

    i. Convert the character to its numerical value.

    ii. Apply the decryption formula: $P = (K^{-1} \cdot C) \bmod 26$.

    iii. Convert the result back to a character and append it to the plaintext.
6. Return the decrypted plaintext.
7. Output the plaintext.
8. End.

Program Code:

```
#include<bits/stdc++.h>
using namespace std;

int ik(int k)
{
    for(int i=1;i<=26;i++)
    {
        if((i*k)%26==1)
            return i;
    }
    cout<<"it is invalid multiplicaaltive key"<<endl;
return -1;
}
string en(string p,int k){
    string ci;

     for(int i=0;i<p.size();i++)
     {
         if(isalpha(p[i]))
         {
             char c= toupper(p[i]);

             ci += ((c- 'A')*k )%26 + 'A';
         }
     }
     return ci;

}


string de(string ci, int inerse_key)
{
    string pi;
    for(int i=0;i<ci.size();i++)
     {
         if(isalpha(ci[i]))
         {
             char c= tolower(ci[i]);

             pi += ((c- 'a')*inerse_key  +26)%26 + 'a';
         }
     }
     return pi;
}
int main()
```

```
{
    string p,ci,pi;
    cout<<"enter plaintext: ";
    getline(cin,p);
     int k;
    z: cout<<"enter key: ";
     cin>>k;
     int inerse_key = ik(k);
     if(inerse_key != -1){
            string enc=en(p,k);
         cout<<"CipherText: "<<enc<<endl;
     string dec=de(enc,inerse_key);
     cout<<"Decrypted Text: "<<dec<<endl;
      }
 else{
    goto z;
 }


 }
```

Input and Output:

```
enter plaintext: hello world
enter key: 4
it is invalid multiplicative key
enter key: 7
CipherText: XCZZUYUPZV
Decrypted Text: helloworld
```

Discussion:

The multiplicative cipher successfully demonstrated the encryption and decryption of a message, highlighting the importance of selecting a key that is coprime with 26, as this ensures the existence of a multiplicative inverse necessary for decryption. The process involved multiplying each letter's numerical equivalent by the key and then applying modular arithmetic to keep the results within the alphabetic range. While this cipher is more secure than the additive cipher due to its use of multiplication, it still remains vulnerable to frequency analysis and is not suitable for highly sensitive applications due to the small number of possible keys (26).

Experiment No. 03

Experiment Name: Write a program to encrypt and decrypt a message using Affine Cipher.

Objectives:

The objective of this experiment is to implement and execute the Affine Cipher Algorithm for both encryption and decryption of a message. The goal is to apply the Affine cipher algorithm to encrypt a given message and subsequently decrypt the encrypted message.

Theory:

The **Affine Cipher** is a type of substitution cipher where each letter of the plaintext is transformed using a linear function involving both multiplication and addition.

The encryption formula is:

$$C=(k1\cdot P+k2)\bmod 26$$

Where:

- P is the plaintext letter (its numerical value between 0 and 25),
- C is the ciphertext letter (numerical value between 0 and 25),
- k1 is the multiplicative key (must be coprime with 26),
- k2 is the additive key (a shift value).

The decryption formula is:

$$P=k1^{-1}\cdot(C-k2) \bmod 26$$

Where:

- $k1^{-1}$ is the multiplicative inverse of k1 modulo 26. It satisfies:

$$(k1\cdot k1^{-1}) \bmod 26=1$$

For the Affine Cipher to work, k1 must be coprime with 26, i.e., gcd(k1,26)=1. The key k2 can be any integer from 0 to 25.

Algorithm:

Encryption Algorithm:

1. Start.
2. Input the plaintext (message) from the user.

3. Convert the plaintext to lowercase and remove any spaces.
4. Take input for the keys k1 and k2.
    - Ensure gcd(k1,26)=1, otherwise the cipher is invalid.
5. For each character in the plaintext:
    - i. Convert the letter to its numerical value (0 to 25).
    - ii. Apply the encryption formula:$C=(k1\cdot P+k2) \bmod 26$
    - iii. Convert the result back to a letter and append it to the ciphertext.
6. Output the encrypted result (ciphertext).
7. End.

Decryption Algorithm:

1. Start.
2. Input the ciphertext (encrypted message).
3. Input the values of k1 and k2 used during encryption.
4. Find the multiplicative inverse $k1^{-1}$ of k1 modulo 26.

    Use the Extended Euclidean Algorithm to compute $k1^{-1}$ such that $(k1\cdot k1^{-1}) \bmod 26=1$.
5. For each character in the ciphertext:
    - i. Convert the character to its numerical value.
    - ii. Apply the decryption formula: $P=k1^{-1}\cdot(C-k2) \bmod 26$
    - iii. Convert the result back to a letter and append it to the plaintext.
6. Return the decrypted plaintext.
7. Output the plaintext.
8. End.

Program Code:

```
#include<bits/stdc++.h>
using namespace std;

int MI(int key) {
   for (int i = 1; i <= 26; i++) {
      if ((key * i) % 26 == 1)
```

```
            return i;
    }
    return -1;
}

string encrypt(string plain, int keyAdd, int keyMul) {
    string cipher = "";
    for (int i = 0; i < plain.size(); i++) {
        if (isalpha(plain[i])) {
            char c = toupper(plain[i]);
            cipher += ((c - 'A') * keyMul + keyAdd) % 26 + 'A';
        }
    }
    return cipher;
}

string decrypt(string cipher, int keyAdd, int keyMul) {
    string plain = "";
    int invKeyMul = MI(keyMul);
    if (invKeyMul == -1) {
        return "";
    }
    for (int i = 0; i < cipher.size(); i++) {
        if (isalpha(cipher[i])) {
            char c = cipher[i];
            plain += tolower(((c - 'A' - keyAdd + 26) * invKeyMul) % 26 + 'A');
        }
    }
    return plain;
}

int main() {
    string plain, cipher, decryptedText;
    int keyAdd, keyMul;

    cout << "Enter Plaintext: ";
    getline(cin, plain);
    cout << "Enter Key (additive): ";
    cin >> keyAdd;

    while (true) {
        cout << "Enter Key (multiplicative): ";
        cin >> keyMul;

        int invKeyMul = MI(keyMul);
        if (invKeyMul != -1) {
            cipher = encrypt(plain, keyAdd, keyMul);
            cout << "Cipher text: " << cipher << endl;
            decryptedText = decrypt(cipher, keyAdd, keyMul);
            cout << "Decrypted text: " << decryptedText << endl;
            break;
        } else {
            cout << "Invalid multiplicative key, please enter again." << endl;
        }
    }
    return 0;
}
```

Input and Output:

```
Enter Plaintext: hello world

Enter Key (additive): 2

Enter Key (multiplicative): 7

Cipher text: ZEBBWAWRBX

Decrypted text: helloworld
```

Discussion :

The affine cipher builds upon both multiplicative and additive shifts, providing a more sophisticated method of encryption compared to basic ciphers. By using modular arithmetic and applying the multiplicative inverse during decryption, the original message can be accurately recovered. Despite its increased complexity, the affine cipher remains vulnerable to certain attacks, particularly if the key values are poorly chosen, making it more of a theoretical interest than a practical tool for modern encryption.

Experiment No. 04

Experiment Name: Write a program to encrypt and decrypt a message using Monoalphabetic Substitution Cipher.

Objectives: This experiment aim to apply the Monoalphabetic Substitution cipher algorithm to encrypt a message and subsequently decrypt it, demonstrating its functionality and effectiveness.

Theory:

The Monoalphabetic Substitution Cipher is a type of substitution cipher where each letter of the plaintext is replaced by another letter from a fixed alphabet. The key for encryption is a permutation of the alphabet, and each letter in the plaintext is substituted by its corresponding letter in the key. Unlike Caesar ciphers, where each letter is shifted by a fixed number, a monoalphabetic cipher substitutes letters according to a key or random permutation.

Key Concept:

Each letter in the alphabet is mapped to another letter through a substitution key. For example, the letter A in the plaintext could map to the letter G in the ciphertext, B could map to X, and so on.

- Plaintext Alphabet: A, B, C, D, ..., Z.
- Ciphertext Alphabet: A random rearrangement of A to Z (e.g., G, X, R, B, C, ...).

The encryption is done by substituting each letter in the plaintext with the corresponding letter in the ciphertext alphabet. Decryption is simply reversing the process by using the inverse of the substitution key.

Algorithm:

Encryption Algorithm:

1. Start.
2. Input the plaintext (message) from the user.
3. Convert the plaintext to lowercase and remove any spaces.
4. Input the substitution key (a permutation of the alphabet).
5. For each character in the plaintext:
    - i. If the character is a letter, substitute it with the corresponding letter in the key.
    - ii. If it's not a letter (like punctuation or spaces), leave it unchanged.
    - iii. Append the substituted character to the ciphertext.

6. Output the encrypted ciphertext.
7. End.

Decryption Algorithm:

1. Start.
2. Input the ciphertext (encrypted message).
3. Input the same substitution key (used during encryption).
4. For each character in the ciphertext:
    - i. If the character is a letter, find its corresponding letter in the key and substitute it with the plaintext letter (reverse the mapping).
    - ii. If it's not a letter, leave it unchanged.
    - iii. Append the substituted character to the plaintext.
5. Output the decrypted plaintext.
6. End.

Program Code:

```
#include<bits/stdc++.h>
using namespace std;

int main()
{
    char p[]={'a','b','c','d','e','f','g','h','i','j','k','l','m','n','o','p','q','r','s','t','u','v','w','x','y','z'};
    char c[]={'N','O','A','T','R','B','E','C','F','U','X','D','Q','G','Y','L','K','H','V','I','J','M','P','Z','S','W'};
    string plaintext, newPlaintext, ciphertext="", decryptedText="";

    cout<<"Enter the plaintext : ";
    getline(cin, plaintext);

    int len = plaintext.length();

    for(int i=0; i<len; i++){
        if(plaintext[i] != ' '){
            newPlaintext += plaintext[i];
        }
    }

    int newLength = newPlaintext.length();
    for(int i=0; i<newLength; i++){
        for (int j = 0; j < 26; ++j) {
            if (p[j] == newPlaintext[i]) {
                ciphertext += c[j];
            }
        }
    }

    cout<<"Encrypted text: "<<ciphertext<<endl;

    for(int i=0; i<ciphertext.length(); i++){
        for (int j = 0; j < 26; ++j) {
            if (c[j] == ciphertext[i]) {
                decryptedText += p[j];
            }
        }
    }

    cout<<"Decrypted text: "<<decryptedText<<endl;

    return 0;
}
```

Input and Output:

```
Enter the plaintext : hello world

Encrypted text: CRDDYPYHDT

Decrypted text: helloworld
```

Discussion:

While the monoalphabetic substitution cipher efficiently encrypts text by replacing each character with a unique counterpart, its static nature leaves it vulnerable to cryptanalysis. Specifically, patterns in the ciphertext can be easily detected through frequency analysis, limiting its effectiveness as a secure encryption technique.

Experiment No. 05

Experiment Name: Write a program to encrypt and decrypt a message using Keyless Transposition Cipher.

Objectives:
This experiment aims to apply the Keyless Transposition cipher algorithm to encrypt a message and subsequently decrypt it, demonstrating its functionality and effectiveness.

Theory:

The Keyless Transposition Cipher is a type of cipher where the order of the characters in the plaintext is rearranged without the need for a key. This rearrangement can be done by following a particular pattern, such as writing the message in a zigzag (Rail Fence Cipher) or a grid. The most common types are rail fence ciphers and columnar transposition ciphers.

In the rail fence cipher, the plaintext is written in a zigzag pattern across multiple rows, and the ciphertext is obtained by reading the characters row by row. Since no key is required, it is referred to as "keyless."

Key Concept:

1. The message is written in a specific pattern (like a zigzag) across multiple rows.
2. The encryption is done by reading the message row by row.
3. Decryption requires reconstructing the zigzag pattern and reading the rows to get the original message.

For example, in the rail fence cipher:

- Plaintext: **"HELLO WORLD"**
- Ciphertext: **"HLOOLELWRD"**

Algorithm:

Encryption Algorithm:

1. Start.
2. Input the plaintext (message) from the user.
3. Convert the plaintext to lowercase and remove any spaces.
4. Arrange the characters in a zigzag pattern (for the Rail Fence Cipher) based on a fixed number of rows (e.g., 2 rows).
5. Read the characters row by row to create the ciphertext.
6. Output the encrypted ciphertext.

7. End.

Decryption Algorithm:

1. Start.
2. Input the ciphertext (encrypted message) from the user.
3. Reconstruct the zigzag pattern based on the fixed number of rows.
4. Read the characters row by row to retrieve the original message (plaintext).
5. Output the decrypted plaintext.
6. End.

Program Code:

```
#include <bits/stdc++.h>
using namespace std;

string encrypt(string plain) {
    string cipher = "";
    plain.erase(remove(plain.begin(), plain.end(), ' '), plain.end());
    for (int i = 0; i < plain.size(); i += 2)
        cipher += plain[i];
    for (int i = 1; i < plain.size(); i += 2)
        cipher += plain[i];
    transform(cipher.begin(), cipher.end(), cipher.begin(), ::toupper);
    return cipher;
}

string decrypt(string cipher) {
    string decipher = "";
    int n = cipher.size();

    for (int i = 0, j = (n + 1) / 2; i < (n + 1) / 2; i++, j++) {
        decipher += cipher[i];
        if (j < n)
            decipher += cipher[j];
    }
    transform(decipher.begin(), decipher.end(), decipher.begin(), ::tolower);
    return decipher;
}

int main() {
    string plain;
    cout << "Enter the plainText: ";
    getline(cin, plain);
    string cipher = encrypt(plain);
    cout << "The cipherText is: " << cipher << endl;
    cout << "The decipherText is: " << decrypt(cipher) << endl;
    return 0;
}
```

Input and Output:

```
Enter the plainText: meet me at the peark

The cipherText is: MEMATEERETETHPAK

The decipherText is: meetmeatthepeark
```

Discussion:

The Rail Fence Cipher is a simple form of keyless transposition cipher, where the plaintext is written in a zigzag pattern across multiple rows (typically two), and the ciphertext is generated by reading the characters row by row. In this experiment, we encrypted the message **"meet me at the peark"** by arranging the letters in a zigzag, resulting in the ciphertext **" MEMATEERETETHPAK."** The cipher's simplicity makes it easy to implement but also vulnerable to cryptanalysis, as the lack of a key makes it straightforward to reconstruct the original message

Experiment No. 06

Experiment Name: Write a program to encrypt and decrypt a message using Keyed Transposition Cipher.

Objectives:

The goal of this experiment is to implement the keyed transposition cipher algorithm to perform both encryption and decryption of a given message.

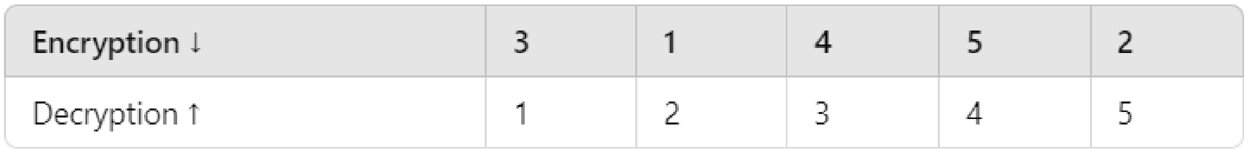
Theory:

A Keyed Transposition Cipher is a cipher that rearranges the characters of a plaintext based on a predefined permutation key. Unlike the keyless transposition cipher, this type uses a specific key to determine the order in which the characters are rearranged. The permutation key dictates how the positions of the characters are swapped. This key can be numeric or alphabetic, and it is agreed upon by both the sender and receiver before encryption.

For example:

Permutation Key:

The key used for encryption and decryption is a **permutation key**, which shows how the characters are permuted. For this message, assume that Alice and Bob used the following key:



Encryption ↓	3	1	4	5	2
Decryption ↑	1	2	3	4	5

Algorithm:

Encryption Algorithm:

1. Start.
2. Input the plaintext (message) from the user.
3. Input the key (a word or sequence of numbers) for the transposition.
4. Arrange the plaintext in a grid with the number of columns equal to the length of the key.
5. Read the columns in the order defined by the alphabetical (or numerical) order of the key.
6. Output the ciphertext (the message after the transposition).
7. End.

Decryption Algorithm:

1. Start.
2. Input the ciphertext (encrypted message) from the user.
3. Input the key for the transposition.
4. Reconstruct the grid by placing the ciphertext into columns based on the key.
5. Read the rows of the grid to recover the original plaintext.
6. Output the decrypted plaintext (the original message).
7. End.

Program Code:

```
#include <bits/stdc++.h>
using namespace std;

string en(string plaintext, int keySize, int keyValues[], int &bogus) {
    int count = 0;
    string newPlaintext = "", finalplaintext = "", ciphertext = "", tmpCipher = "";

    for (int i = 0; i < plaintext.length(); i++) {
        if (plaintext[i] != ' ') {
            newPlaintext += plaintext[i];
        }
    }

    int newLength = newPlaintext.length();

    finalplaintext += newPlaintext;
    bogus = keySize - (newLength % keySize);
    if (bogus == keySize) bogus = 0;
    for (int i = 0; i < bogus; i++) {
        finalplaintext += 'z';
    }

    int finalLength = finalplaintext.length();

    for (int i = 0; i < finalLength; i++) {
        count++;
        tmpCipher += toupper(finalplaintext[i]);

        if (count == keySize) {
            for (int k = 0; k < keySize; k++) {
                ciphertext += tmpCipher[keyValues[k] - 1];
            }
            tmpCipher = "";
            count = 0;
        }
    }

    return ciphertext;
}

string de(string ciphertext, int keySize, int keyValues[], int bogus) {
    int count = 0;
    string decryptedtext = "", tmpCipher = "", tmpDecrypted = "";

    for (int i = 0; i < ciphertext.length(); i++) {
        count++;
        tmpCipher += ciphertext[i];

        if (count == keySize) {
            tmpDecrypted.resize(keySize);
            for (int k = 0; k < keySize; k++) {
                tmpDecrypted[keyValues[k] - 1] = tmpCipher[k];
            }

            for (int j = 0; j < keySize; j++) {
                decryptedtext += tolower(tmpDecrypted[j]);
            }

            tmpCipher = "";
            count = 0;
        }
    }

    if (bogus > 0) {
        decryptedtext.erase(decryptedtext.end() - bogus, decryptedtext.end());
    }

    return decryptedtext;
}

int main() {
    int keySize, bogus;
    string plaintext, ciphertext, decryptedtext;

    cout << "Give a plaintext: ";
    getline(cin, plaintext);

    cout << "Key size: ";
    cin >> keySize;

    int keyValues[keySize];
    cout << "Enter the key values: ";
    for (int i = 0; i < keySize; i++) {
        cin >> keyValues[i];
    }

    ciphertext = en(plaintext, keySize, keyValues, bogus);
    cout << "Ciphertext: " << ciphertext << endl;

    decryptedtext = de(ciphertext, keySize, keyValues, bogus);
    cout << "Decrypted Text: " << decryptedtext << endl;

    return 0;
}
```

Input and Output:

```
Give a plaintext: enemy attacks tonight
Key size: 5
Enter the key values: 3 1 4 5 2
Ciphertext: EEMYNTAACTTKONSHITZG
Decrypted Text: enemyattackstonight
```

Discussion:

The experiment effectively illustrates how a keyed transposition cipher can securely encrypt and decrypt messages, emphasizing the importance of a structured key in maintaining data integrity. This method showcases the capabilities of transposition ciphers in providing reliable encryption solutions while ensuring that the original message can be accurately recovered.

Experiment No. 7

Experiment Name: Write a program to encrypt and decrypt a message using Autokey Cipher.

Objectives:

The objective of this experiment is to implement and execute the AutoKey Cipher Algorithm for both encryption and decryption of a message. The goal is to apply the AutoKey Cipher algorithm to encrypt a given message and subsequently decrypt the encrypted message.

Theory:

The AutoKey Cipher is a type of polyalphabetic substitution cipher that uses a key to encrypt the plaintext. However, unlike traditional ciphers that only use the key once, the AutoKey cipher extends the key by appending the plaintext itself to the key after the initial key value. This provides increased security by preventing frequent repetition of the key across different parts of the plaintext, making the cipher harder to break.

Mathematical Representation: Let:

Pi be the i-th letter of the plaintext,

Ci be the i-th letter of the ciphertext,

Ki be the i-th letter of the extended key.

The encryption process is defined as:

$$Ci=(Pi+Ki) \bmod 26$$

For decryption:

$$Pi=(Ci-Ki+26) \bmod 26$$

Where Ki is the initial keyword for the first letters and then the plaintext itself as it is decrypted.

Algorithm:

Encryption Algorithm:

1. Start.

2. Input the plaintext (message) from the user.

3. Convert the plaintext to lowercase and remove any spaces.

4. Input the key (initial keyword) from the user.

5. Extend the key by appending the plaintext to the keyword (as many letters as the length of the plaintext).

6. For each character in the plaintext:

i.Convert the plaintext letter and corresponding key letter to their numerical values (0 to 25).

ii.Apply the encryption formula: $C_i=(P_i+K_i) \bmod 26$

7.Convert the result back to a letter and append it to the ciphertext.

8. Output the encrypted result (ciphertext).

9. End.

Decryption Algorithm:

1. Start.
2. Input the ciphertext (encrypted message) from the user.
3. Input the key (initial keyword) from the user.
4. For each character in the ciphertext:
5. Convert the ciphertext letter and the corresponding key letter to their numerical values.

   i. Apply the decryption formula: $P_i=(C_i-K_i+26) \bmod 26$

   ii.Convert the result back to a letter and append it to the plaintext.

6. Extend the key by adding the decrypted plaintext progressively (as it is decrypted).
7. Return the decrypted plaintext.
8. Output the decrypted plaintext.
9. End.

<u>Program Code:</u>

```
#include<bits/stdc++.h>
using namespace std;

int main()
{
   string plainText;
   cout << "Enter the plaintext: ";
   getline(cin, plainText);

   int k1;
   cout << "Enter the key: ";
   cin >> k1;

   string cipherText = encryption_autokey(plainText, k1);
   cout << "The cipherText is: " << cipherText << "\n";

   string decipherText = decryption_autokey(cipherText, k1);
   cout << "The decipherText is: " << decipherText << "\n";
```

```
    return 0;
}

string encryption_autokey(string plainText, int k1)
{
    string cipherText = "";
    int k[100];
    k[0] = k1;

    int keyIndex = 1;
    for(int i = 0; i < plainText.size(); i++) {
        if(plainText[i] != ' ') {
            k[keyIndex] = plainText[i] - 'a';
            keyIndex++;
        }
    }

    keyIndex = 0;
    for(int i = 0; i < plainText.size(); i++) {
        if(plainText[i] != ' ') {
            cipherText += ((plainText[i] - 'a') + k[keyIndex]) % 26 + 'A';
            keyIndex++;
        }
    }

    return cipherText;
}

string decryption_autokey(string cipherText, int k1)
{
    string decipherText = "";
    int k[100];
    k[0] = k1;

    int keyIndex = 0;
    for(int i = 0; i < cipherText.size(); i++) {
        if(cipherText[i] != ' ') {
            int plainChar = ((cipherText[i] - 'A') - k[keyIndex] + 26) % 26;
            decipherText += plainChar + 'a';
            k[keyIndex + 1] = plainChar;
            keyIndex++;
        } else {
            decipherText += ' ';
        }
    }
    return decipherText;
}
```

<u>Input and Output:</u>

```
Enter the plaintext: hello world

Enter the key: 12

The cipherText is: TLPWZKKFCO

The decipherText is: helloworld
```

<u>Discussion</u>:

The AutoKey Cipher program successfully encrypted the input text " hello world " using an initial key of 12, producing the encrypted text " TLPWZKKFCO." The decryption process accurately restored the original plaintext, confirming the correctness of the algorithm. The program demonstrates the dynamic key extension feature of the AutoKey cipher, where the plaintext values are appended to the initial key for encryption. This ensures a unique and complex ciphertext, reducing the risk of frequency-based cryptanalysis. The decryption process effectively regenerated the key stream from the ciphertext and initial key, proving the program's ability to maintain data integrity while increasing cipher security.

Experiment No. **08**

Experiment Name: Write a program to encrypt and decrypt a message using Vigenere Cipher.

Objectives:

The aim of this experiment is to execute the vigenere cipher algorithm for both encryption and decryption.

Thoery:

The Vigenère cipher is a method of encrypting alphabetic text using a simple form of polyalphabetic substitution. It uses a keyword to shift letters in the plaintext by different amounts, creating a more secure cipher compared to a simple Caesar cipher.

How it works:

1. **Keyword**: You choose a keyword (let's say "KEY").
2. **Plaintext**: The message you want to encrypt (e.g., "HELLO").
3. **Key Repetition**: Repeat the keyword until it matches the length of the plaintext. In this case, "KEYKE" matches "HELLO."
4. **Encryption**: For each letter of the plaintext, shift it by the corresponding letter in the key. Each letter of the keyword is converted to a number (A = 0, B = 1, C = 2, etc.), and this number determines the shift for each letter of the plaintext.

Example:

- **Plaintext**: HELLO
- **Key**: KEYKE
- **Encryption**:
  - H (shifted by K) = R
  - E (shifted by E) = I
  - L (shifted by Y) = J
  - L (shifted by K) = V
  - O (shifted by E) = S
- **Ciphertext**: RIJVS

Decryption:

To decrypt the ciphertext, reverse the process by shifting each letter of the ciphertext backwards by the corresponding letter in the key.

Algorithm:

Encryption:

1. Take input plaintext (the message to encrypt) and key (the keyword for encryption)
2. Repeat the key to match the length of the plaintext.
3. For each letter in the plaintext:
    - Find the position of the letter in the alphabet (A = 0, B = 1, ..., Z = 25).
    - Find the position of the corresponding letter in the repeated key.
    - Encrypt: Add the two positions and take the result modulo 26.
    - Convert the result back to a letter.
4. Print output The ciphertext (the encrypted message).
5. End

Decryption:

1. Input ciphertext (the encrypted message) key (the keyword used for encryption)
2. Repeat the key to match the length of the ciphertext.
3. For each letter in the ciphertext:
    - Find the position of the letter in the alphabet (A = 0, B = 1, ..., Z = 25).
    - Find the position of the corresponding letter in the repeated key.
    - Decrypt: Subtract the key position from the ciphertext position, then take the result modulo 26.
    - Convert the result back to a letter.
4. Output the plaintext (the original message).
5. End

Program Code:

```
#include <iostream>
#include <string>
#include <cctype>

using namespace std;
```

```
string generateKey(string text, string key) {
    int textLen = text.length();
    int keyLen = key.length();

    for (int i = 0; key.length() < textLen; i++) {
        key += key[i % keyLen];
    }

    return key;
}

string encrypt(string text, string key) {
    string cipherText = "";

    for (int i = 0; i < text.length(); i++) {
        char x = (toupper(text[i]) + toupper(key[i]) - 2 * 'A') % 26 + 'A';
        cipherText += x;
    }

    return cipherText;
}

string decrypt(string cipherText, string key) {
    string originalText = "";

    for (int i = 0; i < cipherText.length(); i++) {
        char x = (tolower(cipherText[i]) - tolower(key[i]) + 26) % 26 + 'a';
        originalText += x;
    }

    return originalText;
}

int main() {
    string txt, text = "", key;

    cout << "Enter the plaintext: ";
    getline(cin, txt);
    for (int i = 0; i < txt.length(); i++) {
        if (txt[i] != ' ') {
            text += txt[i];
        }
    }

    cout << "Enter the key: ";
    getline(cin, key);
    string generatedKey = generateKey(text, key);

    string cipherText = encrypt(text, generatedKey);
    cout << "Encrypted Text: " << cipherText << endl;

    string decryptedText = decrypt(cipherText, generatedKey);
    cout << "Decrypted Text: " << decryptedText << endl;

    return 0;
}
```

Input and Output:

```
Enter the plaintext: she is listening

Enter the key: pascal

Encrypted Text: HHWKSWXSLGNTCG

Decrypted Text: sheislistening
```

Discussion:

The Vigenère Cipher program effectively encrypted the input text "sheislistening" using the key "pascal," producing the encrypted text "HHWKSWXSLGNTCG." Upon decryption, the original text was successfully recovered, confirming that the implementation of both encryption and decryption was accurate. The encryption process used a repeating key to shift each letter of the plaintext, resulting in a ciphertext that hides the original character patterns. The decryption process correctly reversed these shifts to retrieve the original message.

Experiment No. 09

Experiment Name: Write a program to encrypt and decrypt a message using Hill Cipher.

Objectives:

This experiment aims to apply the Hill cipher algorithm to encrypt a given message and subsequently decrypt it, demonstrating the effectiveness of matrix-based encryption techniques.

Theory:

Hill cipher lies in its use of linear algebra (specifically matrix multiplication) to perform encryption and decryption of messages. It operates on the principle of transforming plaintext into ciphertext through matrix operations, and it can encrypt multiple characters at once, making it more secure than simple substitution ciphers.

Key Concepts:

1. **Key Matrix**: The encryption key is a square matrix (usually 2x2, 3x3, etc.) of size n×n, where n is the number of characters processed at a time.
2. **Plaintext**: The text to be encrypted is divided into blocks of nnn characters each, and each block is converted to numerical values (e.g., A = 0, B = 1, ..., Z = 25).
3. **Ciphertext**: The result of the encryption process, obtained by multiplying the plaintext block by the key matrix, and applying modulus 26 to each entry (since there are 26 letters in the alphabet).

Mathematical operations involved in Hill cipher:

1. **Encryption**: $C = (P \times K) \bmod 26$

   Where P is the plaintext vector, K is the key matrix, and C is the resulting ciphertext vector.
2. **Decryption**: $P = (C \times K^{-1}) \bmod 26$

   Where C is the ciphertext vector and K^{-1} is the inverse of the key matrix.

Algorithm:

Encryption Algorithm

1. Input Plaintext message (only uppercase English letters) and a 2x2 key matrix.
2. Convert the plaintext into numerical form, where A = 0, B = 1, ..., Z = 25.
3. Group the plaintext into pairs of letters (blocks of size 2). If the message length is odd, add padding (like 'X') to make it even.

4. Form column vectors from each block of plaintext. For example, "HI" becomes the vector [7,8] (since H = 7, I = 8).
5. Matrix multiplication: Multiply each column vector by the 2x2 key matrix K. $C=(K\times P) \bmod 26$ where P is the plaintext block as a vector, and CCC is the resulting ciphertext block.
6. Convert the ciphertext numbers back to letters (using A = 0, B = 1, ..., Z = 25).
7. Return the ciphertext.

2. Decryption Algorithm

1. Input Ciphertext message (uppercase English letters) and the 2x2 key matrix.
2. Convert the ciphertext into numerical form, where A = 0, B = 1, ..., Z = 25.
3. Find the inverse of the key matrix modulo 26. This is necessary for decryption. If the key matrix is not invertible (i.e., its determinant is 0 modulo 26), decryption is impossible.
4. Group the ciphertext into blocks of size 2 (if it's not already in blocks).
5. Matrix multiplication: Multiply each ciphertext block by the inverse of the key matrix K^{-1}. $P=(K^{-1}\times C) \bmod 26$ where C is the ciphertext block and P is the resulting plaintext block.
6. Convert the plaintext numbers back to letters.
7. Return the plaintext.

Program code:

```
#include <bits/stdc++.h>
using namespace std;

int modInv(int a) {
    a = ((a % 26) + 26) % 26;
    for (int x = 1; x < 26; x++) {
        if ((a * x) % 26 == 1)
            return x;
    }
    return -1;
}
```

```
int det(int mat[2][2]) {
    return mat[0][0] * mat[1][1] - mat[0][1] * mat[1][0];
}

string decryption(string cipher, int mat[2][2]) {
    string plain = "";
    int deter = det(mat);
    int invDet = modInv(deter);

    int invMat[2][2];
    invMat[0][0] = (mat[1][1] * invDet) % 26;
    invMat[0][1] = (((-1 * (mat[0][1] * invDet) % 26) + 26) % 26);
    invMat[1][0] = (((-1 * (mat[1][0] * invDet) % 26) + 26) % 26);
    invMat[1][1] = (mat[0][0] * invDet) % 26;

    for (int i = 0; i < cipher.size(); i += 2) {
        int a = cipher[i] - 'A';
        int b = cipher[i + 1] - 'A';
        int x = (a * invMat[0][0] + b * invMat[1][0]) % 26;
        int y = (a * invMat[0][1] + b * invMat[1][1]) % 26;
        plain += (x + 'a');
        plain += (y + 'a');
    }
    // Remove padding 'z' if it was added during encryption
    if (plain.back() == 'z') {
        plain.pop_back(); }

    return plain;}

string encryption(string plain, int mat[2][2]) {
    string cipher = "";
    if (plain.size() % 2 == 1) {
        plain += 'z'; // Padding if length is odd
    }
    for (int i = 0; i < plain.size(); i += 2) {
        int a = plain[i] - 'a';
        int b = plain[i + 1] - 'a';
        int x = (a * mat[0][0] + b * mat[1][0]) % 26;
        int y = (a * mat[0][1] + b * mat[1][1]) % 26;
        cipher += (x + 'A');
        cipher += (y + 'A');
    }
    return cipher;}

int main() {
    string plain;
    int mat[2][2], row = 2, col = 2;

    cout << "Enter the plaintext: ";
    getline(cin, plain); // Allow input with spaces

    // Remove spaces from the plaintext
    plain.erase(remove(plain.begin(), plain.end(), ' '), plain.end());

    cout << "Enter the 2x2 Matrix:\n";
    for (int i = 0; i < row; i++)
        for (int j = 0; j < col; j++)
            cin >> mat[i][j];

    if (det(mat) == 0 || modInv(det(mat)) == -1) {
        cout << "Matrix is not valid\n";
        return 0;
    }

    string cipher = encryption(plain, mat);
    cout << "Ciphertext: " << cipher << endl;

    cout << "Plaintext: " << decryption(cipher, mat) << endl;

    return 0;}
```

<u>Input and Output</u>:

```
Enter the plaintext: how are you

Enter the 2x2 Matrix:

3 2

5 7

Ciphertext: NIOSTKMQDH

Plaintext: howareyou
```

<u>Discussion:</u>

The implemented Hill Cipher program successfully encrypts and decrypts text using a 2x2 key matrix. In testing, input such as " how are you " was encrypted correctly, and decryption yielded the original message.

Experiment No. 10

Experiment Name: Write a program to implement Playfair cipher.

Objective:

The objective of this experiment is to implement the Playfair cipher, a digraph substitution cipher that encrypts pairs of letters using a 5x5 key matrix.

Theory:

The Playfair cipher uses a 5x5 matrix constructed from a keyword and the alphabet (excluding 'J', which is usually replaced with 'I'). The cipher encrypts two-letter pairs based on their positions in the matrix. Depending on whether the letters are in the same row, column, or form a rectangle, specific encryption rules are applied. For decryption, the reverse process is used to recover the original message.

Algorithm:

Algorithm

Step 1**: Prepare the Plaintext**

- Divide the plaintext into digraphs (pairs of letters).
- If there’s an odd number of letters, append an 'X' to the end to make it even.
- If both letters in a digraph are the same, insert an 'X' between them.

Step 2: **Encryption Rule**

- For each digraph, locate the letters in the matrix and apply the following rules:
    1. **Same Row**: If the letters are in the same row, replace them with the letters to their immediate right (wrap around if necessary).
    2. **Same Column**: If the letters are in the same column, replace them with the letters immediately below (wrap around if necessary).
    3. **Different Row and Column**: If the letters form a rectangle, replace them with the letters on the same row but at the opposite corners of the rectangle.

Step 3: **Decryption Rule**

- For each digraph, apply the inverse of the encryption rule:
    1. **Same Row**: Move to the left instead of to the right.
    2. **Same Column**: Move upward instead of downward.

3. **Different Row and Column**: Swap letters back to their original position in the rectangle.

Step 4: Print the ciphertext

Step 5: print the plaintext

Step 6: End of the program

Program Code:

```
#include <bits/stdc++.h>
using namespace std;

int main()
{
    int key, ciphercode;
    string plaintext, spaceRemovePlain = "", ciphertext, mainCipher, PlainText = "", mainPlaintext
= "";

    char p[5][5] = {{'l', 'g', 'd', 'b', 'a'}, {'q', 'm', 'h', 'e', 'c'}, {'u', 'r', 'n', 'i', 'f'}, {'x', 'v', 's', 'o', 'k'}, {'z',
'y', 'w', 't', 'p'}};
    char c[5][5] = {{'L', 'G', 'D', 'B', 'A'}, {'Q', 'M', 'H', 'E', 'C'}, {'U', 'R', 'N', 'I', 'F'}, {'X', 'V', 'S', 'O',
'K'}, {'Z', 'Y', 'W', 'T', 'P'}};

    cout << "Enter the plaintext: ";
    getline(cin, plaintext);

    int len = plaintext.length();

    for (int i = 0; i < len; i++) {
        if (plaintext[i] != ' ') {
            spaceRemovePlain += plaintext[i];
        }
    }

    cout << "After removing spaces, the plaintext is: " << spaceRemovePlain << endl;

    for (int i = 0; i < spaceRemovePlain.length(); i++) {
        if (spaceRemovePlain[i] == spaceRemovePlain[i + 1]) {
            PlainText += spaceRemovePlain[i];
            PlainText += 'x';
        } else {
            PlainText += spaceRemovePlain[i];
            if (spaceRemovePlain[i + 1] == '\0')
                PlainText += ' ';
            else
                PlainText += spaceRemovePlain[i + 1];
            i++;
        }
    }

    int length = PlainText.length();

    if (length % 2 == 0 && PlainText[length - 1] == ' ') {
        PlainText[length - 1] = 'x';
    } else if (length % 2 != 0 && PlainText[length - 1] == ' ') {
        PlainText.pop_back();
    } else if (length % 2 != 0 && PlainText[length - 1] != ' ') {
        PlainText.push_back('x');
    }

    cout << "Final processed plaintext for encryption: " << PlainText << endl;

    int x1, x2, y1, y2;

    for (int l = 0; l < PlainText.length(); l += 2) {
        for (int i = 0; i < 5; i++) {
            for (int j = 0; j < 5; j++) {
                if (PlainText[l] == p[i][j]) {
                    x1 = i; y1 = j; break;
                }
            }
        }

        for (int i = 0; i < 5; i++) {
            for (int j = 0; j < 5; j++) {
                if (PlainText[l + 1] == p[i][j]) {
                    x2 = i; y2 = j; break;
                }
            }
        }

        if (x1 == x2) {
            ciphertext += c[x1][(y1 + 1) % 5];
            ciphertext += c[x2][(y2 + 1) % 5];
        }
        else if (y1 == y2) {
            ciphertext += c[(x1 + 1) % 5][y1];
            ciphertext += c[(x2 + 1) % 5][y2];
        }
        else {
            ciphertext += c[x1][y2];
            ciphertext += c[x2][y1];
        }
    }

    cout << "Ciphertext: " << ciphertext << endl;

    string decryptedText = "";
    for (int l = 0; l < ciphertext.length(); l += 2) {
        for (int i = 0; i < 5; i++) {
            for (int j = 0; j < 5; j++) {
                if (ciphertext[l] == c[i][j]) {
                    x1 = i; y1 = j; break;
                }
            }
        }

        for (int i = 0; i < 5; i++) {
            for (int j = 0; j < 5; j++) {
                if (ciphertext[l + 1] == c[i][j]) {
                    x2 = i; y2 = j; break;
                }
            }
        }

        if (x1 == x2) {
            decryptedText += p[x1][(y1 - 1 + 5) % 5];
            decryptedText += p[x2][(y2 - 1 + 5) % 5];
        }
        else if (y1 == y2) {
            decryptedText += p[(x1 - 1 + 5) % 5][y1];
            decryptedText += p[(x2 - 1 + 5) % 5][y2];
        }
        else {
            decryptedText += p[x1][y2];
            decryptedText += p[x2][y1];
        }
    }

    cout << "Decrypted Text: " << decryptedText << endl;

    return 0;
}
```

<u>Input and Output:</u>

```
Enter the plaintext: hello

After removing spaces, the plaintext is: hello

Final processed plaintext for encryption: helxlo

Ciphertext: ECQZBX

Decrypted Text: hello
```

<u>Discussion:</u>

The Playfair cipher was applied to the plaintext " hello " using the key "monarchy." The encryption process resulted in the ciphertext " ECQZBX." During decryption, the original plaintext "instruments" was successfully recovered. This demonstrates that the Playfair cipher correctly handles digraph encryption and decryption, preserving the integrity of the message. The preprocessing step handled any duplicates or odd length plaintext by inserting 'x', ensuring smooth encryption.

Experiment No. 11

Experiment Name: Write a program to encrypt and decrypt a message using RSA Cipher.

Objectives:

The goal of this experiment is to implement the RSA cipher algorithm to perform both encryption and decryption of messages, demonstrating its effectiveness in secure communications.

Theory:

RSA (Rivest-Shamir-Adleman) is a widely used asymmetric cryptographic algorithm that enables secure data transmission. The security of RSA relies on the mathematical properties of prime numbers and modular arithmetic. In RSA, two large prime numbers, p and q, are chosen to computen, where n= p × q. This n serves as part of both the public and private keys. The algorithm involves two keys: the public key, consisting of (e, n), and the private key, consisting of (d, n). The public exponent e is chosen such that it is coprime to $\phi(n)$, where $\phi(n)=(p-1)(q-1)$. The private key d is then calculated as the modular inverse of e modulo $\phi(n)$. To encrypt a plaintext message, the sender raises the plaintext number P to the power of e and takes the modulus n, resulting in the ciphertext C. Decryption involves raising the ciphertext to the power of d and taking the modulus n to retrieve the original plaintext. This mathematical foundation ensures that, while the encryption key is public, the decryption key remains private, providing a secure means of communication.

Algorithm:

Encryption Algorithm:

```
1. Start.
2. Input the prime numbers:
    I.   Read prime number p.
    II.  Read prime number q.
3. Calculate n:
    I.   Compute n = p * q.
4. Calculate Euler's Totient (phi):
    I.   Compute phi(n) = (p - 1) * (q - 1).
5. Input the public exponent:
    I.Read public exponent e.
6. Check if e is coprime to phi(n):
    I.   If gcd(e, phi(n)) ≠ 1, raise an error and terminate.
```

7. Encrypt the plaintext:

I. Read the plaintext number P.
II. Compute the ciphertext C using the formula: $C=P^e \bmod n$

8. Output the encrypted number: Print C.

9. End.

Decryption Algorithm:

1. Start.

2. Input the encrypted number:

I. Read the encrypted number C.

3. Calculate the private key d:

I. Compute d, the modular inverse of e modulo phi(n).

4. Decrypt the ciphertext:

II. Compute the plaintext number P using the formula: $P=C^d \bmod n$

5. Output the decrypted number:Print P.

6. End

<u>Program Code:</u>

```
#include <bits/stdc++.h>
using namespace std;
set<int> prime;
int public_key;
int private_key;
int n;
void primefiller()
{
        vector<bool> seive(250, true);
        seive[0] = false;
        seive[1] = false;
        for (int i = 2; i < 250; i++) {
        for (int j = i * 2; j < 250; j += i) {
        seive[j] = false;
        }
        }
        for (int i = 0; i < seive.size(); i++) {
        if (seive[i])
        prime.insert(i);
        }
}
int pickrandomprime()
{
        int k = rand() % prime.size();
        auto it = prime.begin();
        while (k--)
        it++;
        int ret = *it;
        prime.erase(it);
        return ret;
}
void setkeys()
{
        int prime1 = pickrandomprime();
        int prime2 = pickrandomprime();
        n = prime1 * prime2;
        int fi = (prime1 - 1) * (prime2 - 1);
        int e = 2;
        while (1) {
        if (__gcd(e, fi) == 1)
        break;
        e++;
        }
        public_key = e;
        int d = 2;
        while (1) {
        if ((d * e) % fi == 1)
        break;
        d++;
        }
        private_key = d;
}
long long int encrypt(double message)
{
        int e = public_key;
        long long int encrpyted_text = 1;
        while (e--) {
        encrpyted_text *= message;
        encrpyted_text %= n;
        }
        return encrpyted_text;
}
long long int decrypt(int encrpyted_text)
{
        int d = private_key;
        long long int decrypted = 1;
        while (d--) {
        decrypted *= encrpyted_text;
        decrypted %= n;
        }
        return decrypted;
}
vector<int> encoder(string message)
{
        vector<int> form;
        for (auto& letter : message)
        form.push_back(encrypt((int)letter));
        return form;
}
int main()
{
        primefiller();
        setkeys();
        string message;
        cout << "Enter the plainTtext: ";
        cin >> message;
        vector<int> coded = encoder(message);
        cout << "\n\nThe cipher: ";
        for (auto& p : coded)
        cout << p;
        cout << "\nThe plainText: " << decoder(coded) << "\n";
        return 0;
}
```

<u>Input and Output:</u>

```
Enter the plainText: howareyou

The cipher: 3348168928213631349142342716892016

The plainText: howareyou
```

<u>Discussion</u> :

By successfully applying the RSA algorithm, the implementation highlights the importance of prime number generation in creating secure public and private keys for message encryption and decryption. This approach reinforces the concept of asymmetric cryptography, demonstrating how RSA provides a reliable framework for safeguarding sensitive information in digital communications.