```c
#include <stdio.h>
#include <math.h>

long long int power_mod(long long int base, long long int exp, long long int mod) {
    long long int result = 1;
    while (exp > 0) {
        if (exp % 2 == 1) {
            result = (result * base) % mod;
        }
        base = (base * base) % mod;
        exp /= 2;
    }
    return result;
}

int main() {
    long long int p, g, a_private, b_private, a_public, b_public, shared_secret_a, shared_secret_b;

    p = 23;
    g = 5;

    printf("Enter private key for A: ");
    scanf("%lld", &a_private);

    printf("Enter private key for B: ");
    scanf("%lld", &b_private);

    a_public = power_mod(g, a_private, p);
    b_public = power_mod(g, b_private, p);

    shared_secret_a = power_mod(b_public, a_private, p);
    shared_secret_b = power_mod(a_public, b_private, p);

    printf("Public key of A: %lld\n", a_public);
    printf("Public key of B: %lld\n", b_public);
    printf("Shared secret computed by A: %lld\n", shared_secret_a);
    printf("Shared secret computed by B: %lld\n", shared_secret_b);

    return 0;
}
```

```c
#include <stdio.h>

long long int euclidean_algorithm(long long int a, long long int b) {
    while (b != 0) {
        long long int temp = b;
        b = a % b;
        a = temp;
    }
    return a;
}

int main() {
    long long int a, b;

    printf("Enter two numbers: ");
    scanf("%lld %lld", &a, &b);

    long long int gcd = euclidean_algorithm(a, b);

    printf("GCD of %lld and %lld is: %lld\n", a, b, gcd);

    return 0;
}
```

```c
#include <stdio.h>

long long int extended_euclidean_algorithm
(
long long int a,
long long int b,
long long int *x,
long long int *y
) {
    if (b == 0) {
        *x = 1;
        *y = 0;
        return a;
    }

    long long int x1, y1;
    long long int gcd = extended_euclidean_algorithm(b, a % b, &x1, &y1);

    *x = y1;
    *y = x1 - (a / b) * y1;

    return gcd;
}

int main() {
    long long int a, b, x, y;

    printf("Enter two numbers: ");
    scanf("%lld %lld", &a, &b);

    long long int gcd = extended_euclidean_algorithm(a, b, &x, &y);

    printf("GCD of %lld and %lld is: %lld\n", a, b, gcd);
    printf("Coefficients x and y are: x = %lld, y = %lld\n", x, y);

    return 0;
}
```

```
#include <stdio.h>
#include <ctype.h>

void caesar_encrypt(char *text, int shift) {
    for (int i = 0; text[i] != '\0'; ++i) {
        if (isalpha(text[i])) {
            char base = islower(text[i]) ? 'a' : 'A';
            text[i] = (text[i] - base + shift) % 26 + base;
        }
    }
}

void caesar_decrypt(char *text, int shift) {
    for (int i = 0; text[i] != '\0'; ++i) {
        if (isalpha(text[i])) {
            char base = islower(text[i]) ? 'a' : 'A';
            text[i] = (text[i] - base - shift + 26) % 26 + base;
        }
    }
}

int main() {
    char text[1000];
    int shift;

    printf("Enter text: ");
    scanf("%s", text);

    printf("Enter shift: ");
    scanf("%d", &shift);

    caesar_encrypt(text, shift);
    printf("Encrypted: %s\n", text);

    caesar_decrypt(text, shift);
    printf("Decrypted: %s\n", text);

    return 0;
}
```

```
#include <stdio.h>

void shift_encrypt(char *text, int shift) {
    for (int i = 0; text[i] != '\0'; i++) {
        if (text[i] >= 'a' && text[i] <= 'z') {
            text[i] = (text[i] - 'a' + shift) % 26 + 'a';
        } else if (text[i] >= 'A' && text[i] <= 'Z') {
            text[i] = (text[i] - 'A' + shift) % 26 + 'A';
        }
    }
}

void shift_decrypt(char *text, int shift) {
    for (int i = 0; text[i] != '\0'; i++) {
        if (text[i] >= 'a' && text[i] <= 'z') {
            text[i] = (text[i] - 'a' - shift + 26) % 26 + 'a';
        } else if (text[i] >= 'A' && text[i] <= 'Z') {
            text[i] = (text[i] - 'A' - shift + 26) % 26 + 'A';
        }
    }
}

int main() {
    char text[100];
    int shift;

    printf("Enter the text: ");
    scanf("%s", text);

    printf("Enter shift value: ");
    scanf("%d", &shift);

    shift_encrypt(text, shift);
    printf("Encrypted text: %s\n", text);

    shift_decrypt(text, shift);
    printf("Decrypted text: %s\n", text);

    return 0;
}
```

```
#include <stdio.h>
#include <string.h>

void rail_fence_cipher_encrypt(char *text, int key) {
    int len = strlen(text);
    char rail[key][len];


    for (int i = 0; i < key; i++) {
        for (int j = 0; j < len; j++) {
            rail[i][j] = '\n';
        }
    }

    int row = 0, col = 0;
    int dir_down = 0;


    for (int i = 0; i < len; i++) {
        rail[row][col++] = text[i];


        if (row == 0 || row == key - 1)
            dir_down = !dir_down;

        row += dir_down ? 1 : -1;
    }


    printf("Cipher Text: ");
    for (int i = 0; i < key; i++) {
        for (int j = 0; j < len; j++) {
            if (rail[i][j] != '\n')
                printf("%c", rail[i][j]);
        }
    }
    printf("\n");
}


void rail_fence_cipher_decrypt(char *cipher, int key) {
    int len = strlen(cipher);
    char rail[key][len];


    for (int i = 0; i < key; i++) {
        for (int j = 0; j < len; j++) {
            rail[i][j] = '\n';
        }
    }


    int row = 0, col = 0;
    int dir_down = 0;
```

```
    for (int i = 0; i < len; i++) {
        rail[row][col++] = '*';

        if (row == 0 || row == key - 1)
            dir_down = !dir_down;

        row += dir_down ? 1 : -1;
    }


    int index = 0;
    for (int i = 0; i < key; i++) {
        for (int j = 0; j < len; j++) {
            if (rail[i][j] == '*' && index < len) {
                rail[i][j] = cipher[index++];
            }
        }
    }


    row = 0, col = 0;
    dir_down = 0;
    char decrypted[len + 1];
    int decrypt_index = 0;

    for (int i = 0; i < len; i++) {
        decrypted[decrypt_index++] = rail[row][col++];

        if (row == 0 || row == key - 1)
            dir_down = !dir_down;

        row += dir_down ? 1 : -1;
    }

    decrypted[decrypt_index] = '\0';
    printf("Decrypted Text: %s\n", decrypted);
}

int main() {
    char text[100];
    int key;

    printf("Enter text to encrypt: ");
    fgets(text, sizeof(text), stdin);
    text[strcspn(text, "\n")] = 0;

    printf("Enter key (number of rails): ");
    scanf("%d", &key);

    rail_fence_cipher_encrypt(text, key);

    char cipher[100];
    printf("Enter the cipher text to decrypt: ");
    scanf(" %[^\n]", cipher);
```

```
    rail_fence_cipher_decrypt(cipher, key);

    return 0;
}
```

```
#include <stdio.h>
#include <string.h>

#define MOD 26


int mod(int a, int m) {
    int res = a % m;
    return res < 0 ? res + m : res;
}


int mod_inverse(int a, int m) {
    a = mod(a, m);
    for (int x = 1; x < m; x++) {
        if ((a * x) % m == 1) return x;
    }
    return -1;
}


void encrypt(char *message, int key_matrix[2][2], char *encrypted) {
    int vector[2];
    for (int i = 0; i < 2; i++) {
        vector[i] = message[i] - 'A';
    }

    for (int i = 0; i < 2; i++) {
        encrypted[i] = mod(key_matrix[i][0] * vector[0] + key_matrix[i][1] * vector[1], MOD) + 'A';
    }
    encrypted[2] = '\0';
}


void decrypt(char *cipher, int key_matrix[2][2], char *decrypted) {
    int det = mod(key_matrix[0][0]*key_matrix[1][1] - key_matrix[0][1]*key_matrix[1][0], MOD);
    int det_inv = mod_inverse(det, MOD);

    if (det_inv == -1) {
        printf("Key matrix not invertible modulo 26.\n");
        return;
    }

    int inv_matrix[2][2];
    inv_matrix[0][0] = mod( key_matrix[1][1] * det_inv, MOD);
    inv_matrix[0][1] = mod(-key_matrix[0][1] * det_inv, MOD);
    inv_matrix[1][0] = mod(-key_matrix[1][0] * det_inv, MOD);
    inv_matrix[1][1] = mod( key_matrix[0][0] * det_inv, MOD);

    int vector[2];
    for (int i = 0; i < 2; i++) {
        vector[i] = cipher[i] - 'A';
    }

    for (int i = 0; i < 2; i++) {
```

```
        decrypted[i] = mod(inv_matrix[i][0] * vector[0] + inv_matrix[i][1] * vector[1], MOD) + 'A';
    }
    decrypted[2] = '\0';
}

int main() {
    char message[3], encrypted[3], decrypted[3];
    int key_matrix[2][2];

    printf("Enter 2-letter message: ");
    scanf("%2s", message);

    printf("Enter 2x2 key matrix:\n");
    for (int i = 0; i < 2; i++) {
        for (int j = 0; j < 2; j++) {
            scanf("%d", &key_matrix[i][j]);
        }
    }

    encrypt(message, key_matrix, encrypted);
    printf("Encrypted text: %s\n", encrypted);

    decrypt(encrypted, key_matrix, decrypted);
    printf("Decrypted text: %s\n", decrypted);

    return 0;
}
```

```
#include <stdio.h>
#include <string.h>


void vernam(const char *input, const char *key, char *output) {
    int len = strlen(input);
    for (int i = 0; i < len; i++) {
        output[i] = input[i] ^ key[i];
    }
    output[len] = '\0';
}

int main() {
    char plaintext[100];
    char key[100];
    char ciphertext[100];
    char deciphered[100];


    printf("Enter the plaintext: ");
    scanf("%99s", plaintext);


    printf("Enter the key (same length as plaintext): ");
    scanf("%99s", key);

    if (strlen(plaintext) != strlen(key)) {
        printf("Error: The key must be the same length as the plaintext.\n");
        return 1;
    }


    vernam(plaintext, key, ciphertext);


    printf("Ciphertext:");
    for (int i = 0; i < strlen(plaintext); i++) {
        printf("%02x ", (unsigned char)ciphertext[i]);
    }
    printf("\n");


    vernam(ciphertext, key, deciphered);
    printf("Deciphered text: %s\n", deciphered);

    return 0;
}
```

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#include <time.h>

void otp_cipher(char *plaintext, char *key, char *ciphertext) {
    int plaintext_len = strlen(plaintext);
    int key_len = strlen(key);
    int i;

    for (i = 0; i < plaintext_len; i++) {
        if (isalpha(plaintext[i])) {
            int base = isupper(plaintext[i]) ? 'A' : 'a';
            ciphertext[i] = ((plaintext[i] - base + (key[i] - base)) % 26) + base;
        } else {
            ciphertext[i] = plaintext[i];
        }
    }
    ciphertext[i] = '\0';
}

void otp_decipher(char *ciphertext, char *key, char *plaintext) {
    int ciphertext_len = strlen(ciphertext);
    int i;

    for (i = 0; i < ciphertext_len; i++) {
        if (isalpha(ciphertext[i])) {
            int base = isupper(ciphertext[i]) ? 'A' : 'a';
            plaintext[i] = ((ciphertext[i] - base - (key[i] - base) + 26) % 26) + base;
        } else {
            plaintext[i] = ciphertext[i];
        }
    }
    plaintext[i] = '\0';
}

int main() {
    char plaintext[100];
    char key[100];
    char ciphertext[100];
    char decryptedtext[100];

    printf("Enter the plaintext: ");
    scanf("%99s", plaintext);

    srand(time(NULL));
    for (int i = 0; i < strlen(plaintext); i++) {
        if (isalpha(plaintext[i])) {
            int base = isupper(plaintext[i]) ? 'A' : 'a';
            key[i] = (rand() % 26) + base;
        } else {
            key[i] = plaintext[i];
        }
    }
```

```
    key[strlen(plaintext)] = '\0';

    otp_cipher(plaintext, key, ciphertext);
    printf("Ciphertext: %s\n", ciphertext);
    printf("Key: %s\n", key);

    otp_decipher(ciphertext, key, decryptedtext);
    printf("Decrypted text: %s\n", decryptedtext);

    return 0;
}
```

```
#include <stdio.h>
#include <string.h>
#include <ctype.h>

void create_playfair_matrix(char *key, char matrix[5][5]) {
    int i, k;
    char temp[26] = {0};

    for (i = 0, k = 0; i < strlen(key); i++) {
        if (key[i] != 'J') {
            if (temp[toupper(key[i]) - 'A'] == 0) {
                temp[toupper(key[i]) - 'A'] = 1;
                matrix[k / 5][k % 5] = toupper(key[i]);
                k++;
            }
        }
    }

    for (i = 0; i < 26; i++) {
        if (temp[i] == 0) {
            if (i == 'J' - 'A') {
                continue;
            }
            matrix[k / 5][k % 5] = 'A' + i;
            k++;
        }
    }
}

void playfair_cipher(char *plaintext, char *key, char *ciphertext) {
    char matrix[5][5];
    int i, k, row1, col1, row2, col2;
    int plaintext_len = strlen(plaintext);

    create_playfair_matrix(key, matrix);

    for (i = 0, k = 0; i < plaintext_len; i += 2) {
        for (row1 = 0; row1 < 5; row1++) {
            for (col1 = 0; col1 < 5; col1++) {
                if (matrix[row1][col1] == toupper(plaintext[i])) {
                    break;
                }
            }
            if (col1 < 5) {
                break;
            }
        }
        for (row2 = 0; row2 < 5; row2++) {
            for (col2 = 0; col2 < 5; col2++) {
                if (matrix[row2][col2] == toupper(plaintext[i + 1])) {
                    break;
                }
            }
            if (col2 < 5) {
                break;
```

```
            }
        }

        if (row1 == row2) {
            ciphertext[k++] = matrix[row1][(col1 + 1) % 5];
            ciphertext[k++] = matrix[row2][(col2 + 1) % 5];
        } else if (col1 == col2) {
            ciphertext[k++] = matrix[(row1 + 1) % 5][col1];
            ciphertext[k++] = matrix[(row2 + 1) % 5][col2];
        } else {
            ciphertext[k++] = matrix[row1][col2];
            ciphertext[k++] = matrix[row2][col1];
        }
    }
    ciphertext[k] = '\0';
}

void playfair_decipher(char *ciphertext, char *key, char *plaintext) {
    char matrix[5][5];
    int i, k, row1, col1, row2, col2;
    int ciphertext_len = strlen(ciphertext);

    create_playfair_matrix(key, matrix);

    for (i = 0, k = 0; i < ciphertext_len; i += 2) {
        for (row1 = 0; row1 < 5; row1++) {
            for (col1 = 0; col1 < 5; col1++) {
                if (matrix[row1][col1] == toupper(ciphertext[i])) {
                    break;
                }
            }
            if (col1 < 5) {
                break;
            }
        }
        for (row2 = 0; row2 < 5; row2++) {
            for (col2 = 0; col2 < 5; col2++) {
                if (matrix[row2][col2] == toupper(ciphertext[i + 1])) {
                    break;
                }
            }
            if (col2 < 5) {
                break;
            }
        }

        if (row1 == row2) {
            plaintext[k++] = matrix[row1][(col1 + 4) % 5];
            plaintext[k++] = matrix[row2][(col2 + 4) % 5];
        } else if (col1 == col2) {
            plaintext[k++] = matrix[(row1 + 4) % 5][col1];
            plaintext[k++] = matrix[(row2 + 4) % 5][col2];
        } else {
            plaintext[k++] = matrix[row1][col2];
            plaintext[k++] = matrix[row2][col1];
```

```
        }
    }
    plaintext[k] = '\0';
}

int main() {
    char plaintext[100];
    char key[100];
    char ciphertext[sizeof(plaintext) * 2];
    char decryptedtext[sizeof(plaintext) * 2];

    printf("Enter the plaintext: ");
    scanf("%99s", plaintext);

    printf("Enter the key: ");
    scanf("%99s", key);

    playfair_cipher(plaintext, key, ciphertext);
    printf("Ciphertext: %s\n", ciphertext);

    playfair_decipher(ciphertext, key, decryptedtext);
    printf("Decrypted text: %s\n", decryptedtext);

    return 0;
}
```

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

long gcd(long a, long b) {
    while (b != 0) {
        long t = b;
        b = a % b;
        a = t;
    }
    return a;
}

long mod_exp(long base, long exp, long mod) {
    long result = 1;
    base = base % mod;
    while (exp > 0) {
        if (exp % 2 == 1) {
            result = (result * base) % mod;
        }
        exp = exp >> 1;
        base = (base * base) % mod;
    }
    return result;
}

long mod_inverse(long a, long m) {
    long m0 = m, t, q;
    long x0 = 0, x1 = 1;
    if (m == 1) return 0;
    while (a > 1) {
        q = a / m;
        t = m;
        m = a % m;
        a = t;
        t = x0;
        x0 = x1 - q * x0;
        x1 = t;
    }
    if (x1 < 0) x1 += m0;
    return x1;
}

void rsa_generate_keys(long *e, long *d, long *n) {
    long p = 61, q = 53;
    *n = p * q;
    long phi = (p - 1) * (q - 1);
    *e = 17;
    while (gcd(*e, phi) != 1) {
        (*e)++;
    }
    *d = mod_inverse(*e, phi);
}

long rsa_encrypt(long msg, long e, long n) {
```

```
    return mod_exp(msg, e, n);
}

long rsa_decrypt(long cipher, long d, long n) {
    return mod_exp(cipher, d, n);
}

int main() {
    long e, d, n;
    rsa_generate_keys(&e, &d, &n);

    long msg = 65;
    printf("Original message: %ld\n", msg);

    long cipher = rsa_encrypt(msg, e, n);
    printf("Encrypted message: %ld\n", cipher);

    long decrypted = rsa_decrypt(cipher, d, n);
    printf("Decrypted message: %ld\n", decrypted);

    return 0;
}
```

```c
#include <stdio.h>
#include <math.h>

int power(int base, int exp, int mod) {
    int result = 1;
    base = base % mod;
    while (exp > 0) {
        if (exp % 2 == 1) {
            result = (result * base) % mod;
        }
        exp = exp >> 1;
        base = (base * base) % mod;
    }
    return result;
}

int is_primitive_root(int candidate, int p) {
    for (int i = 1; i < p - 1; i++) {
        if (power(candidate, i, p) == 1) {
            return 0;
        }
    }
    return 1;
}

int find_primitive_root(int p) {
    for (int g = 2; g < p; g++) {
        if (is_primitive_root(g, p)) {
            return g;
        }
    }
    return -1;
}

int main() {
    int p;
    printf("Enter a prime number: ");
    scanf("%d", &p);

    if (p <= 1) {
        printf("Input must be a prime number greater than 1\n");
        return 0;
    }

    int root = find_primitive_root(p);
    if (root == -1) {
        printf("No primitive root found for the prime number %d\n", p);
    } else {
        printf("A primitive root of %d is: %d\n", p, root);
    }

    return 0;
}
```