EX-NO-9-RSA-Algorithm

AIM:

To Implement RSA Encryption Algorithm in Cryptography

Algorithm:

Step 1: Design of RSA Algorithm
The RSA algorithm is based on the mathematical difficulty of factoring the product of two large prime numbers. It involves generating a public and private key pair, where the public key is used for encryption, and the private key is used for decryption.

Step 2: Implementation in Python or C This algorithm can be implemented in languages like Python or C by performing large integer calculations for key generation, encryption, and decryption, utilizing libraries for modular arithmetic if necessary.

Step 3: Algorithm Description

1. Key Generation:
    - Select two large prime numbers (p) and (q).
    - Calculate (n = p \times q), which will be used as the modulus.
    - Compute the totient (\phi(n) = (p - 1)(q - 1)).
    - Choose a public exponent (e) such that (e) is coprime with (\phi(n)).
    - Compute the private key (d), which is the modular inverse of (e) mod (\phi(n)).
2. Encryption:
    - Convert the plaintext message (M) into a numerical form (m) (such that (0 \le m < n)).
    - Compute the ciphertext (c) using the formula: (c = m^e \mod n).
3. Decryption:
    - Use the private key (d) to recover (m) from (c) using: (m = c^d \mod n).
    - Convert (m) back into the original message (M).

Step 4: Mathematical Representation

- Encryption: (E(m) = m^e \mod n)
- Decryption: (D(c) = c^d \mod n)

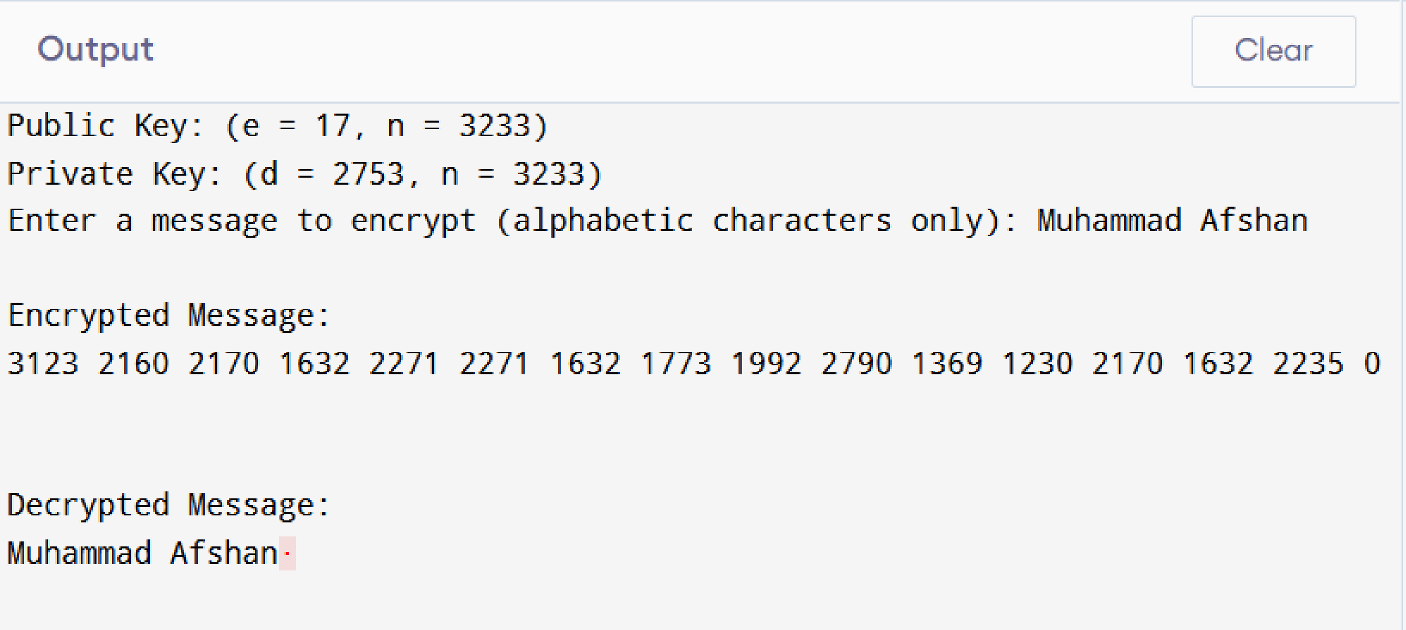
Step 5: **Security Foundation
The security of RSA relies on the difficulty of factoring large numbers; thus, choosing sufficiently large prime numbers for (p) and (q) is crucial for security.

Program:

```
Developed By: Muhammad Afshan A
Ref No.: 212223100035
```

```
#include <stdio.h>
#include <string.h>
#include <math.h>
#include <ctype.h>
#include <stdlib.h>
// Function to calculate GCD using the Euclidean algorithm
int gcd(int a, int b) {
    while (b != 0) {
        int temp = b;
        b = a % b;
        a = temp;
    }
    return a;
}
// Function to calculate (base^exp) % mod using modular exponentiation
long long mod_exp(long long base, long long exp, long long mod) {
    long long result = 1;
    while (exp > 0) {
        if (exp % 2 == 1)
            result = (result * base) % mod;
        base = (base * base) % mod;
        exp = exp / 2;
    }
    return result;
}
// Function to calculate the modular inverse of e mod phi using the extended Euclidean
int mod_inverse(int e, int phi) {
    int t = 0, newt = 1;
    int r = phi, newr = e;
    while (newr != 0) {
        int quotient = r / newr;
        int temp = t;
        t = newt;
        newt = temp - quotient * newt;
        temp = r;
        r = newr;
        newr = temp - quotient * newr;
    }
    if (r > 1) return -1; // e is not invertible
    if (t < 0) t = t + phi;
    return t;
}
int main() {
    // Step 1: Initialize prime numbers p and q (use larger primes for real-world appl
    int p = 61;
    int q = 53;

    // Step 2: Compute n = p * q and phi = (p-1) * (q-1)
    int n = p * q;
    int phi = (p - 1) * (q - 1);

    // Step 3: Choose an encryption key e such that 1 < e < phi and gcd(e, phi) = 1
    int e = 17; // A commonly used public exponent
    if (gcd(e, phi) != 1) {
        printf("e and phi(n) are not coprime!\n");
        return -1;
    }
    // Step 4: Compute the decryption key d, the modular inverse of e mod phi
    int d = mod_inverse(e, phi);
    if (d == -1) {
        printf("No modular inverse found for e!\n");
        return -1;
    }
    // Step 5: Display the public and private keys
    printf("Public Key: (e = %d, n = %d)\n", e, n);
    printf("Private Key: (d = %d, n = %d)\n", d, n);

    // Step 6: Get the message from the user
    char message[100];
    printf("Enter a message to encrypt (alphabetic characters only): ");
    fgets(message, sizeof(message), stdin);
    int len = strlen(message);
    if (message[len - 1] == '\n') message[len - 1] = '\0'; // Remove newline character

    // Step 7: Encrypt the message
    printf("\nEncrypted Message:\n");
    long long encrypted[100];
    for (int i = 0; i < len; i++) {
        int m = (int)message[i];  // Convert the character to its ASCII value
        encrypted[i] = mod_exp(m, e, n);  // Encrypt the ASCII value using RSA
        printf("%lld ", encrypted[i]);  // Print encrypted values
    }
    printf("\n");

    // Step 8: Decrypt the message
    printf("\nDecrypted Message:\n");
    for (int i = 0; i < len; i++) {
        int decrypted = (int)mod_exp(encrypted[i], d, n);  // Decrypt the ASCII value
        printf("%c", (char)decrypted);  // Convert the decrypted ASCII value back to a
    }
    printf("\n");
    return 0;
}
```

Output:



```
Public Key: (e = 17, n = 3233)
Private Key: (d = 2753, n = 3233)
Enter a message to encrypt (alphabetic characters only): Muhammad Afshan

Encrypted Message:
3123 2160 2170 1632 2271 2271 1632 1773 1992 2790 1369 1230 2170 1632 2235 0


Decrypted Message:
Muhammad Afshan
```

Result:

The program is executed successfully.