

Lab-3

Task 1:

Write a program to implement the Miller-Rabin primality test. Test it with various values of 'n'.

Theory:

The Miller-Rabin primality test or Rabin Miller primality test is a probabilistic primality test. It determines whether a given number is likely to be prime or not. For a number n , the steps for Rabin-Miller Primality test are as follows:

Step 1. Find $n - 1 = 2^k * m$, where m is an odd number

Step 2. Choose a such that $1 < a < n - 1$

Step 3. Compute $b_0 = a^m \bmod n$, ..., $b_n = b_{n-1}^2 \bmod n$

Step 4. If b_i is $+1$ the number is composite and if b is -1 the number is probably prime.

Source Code:

```
#include <iostream>
#include <stdlib.h>
using namespace std;

long long mulmod(long long, long long, long long);
long long modulo(long long, long long, long long);
bool Miller(long long, int);

int main()
{
    do
    {
        int iteration = 10;
        long long num;
        cout << "Enter integer to test primality: ";
        cin >> num;
        if (Miller(num, iteration))
            cout << num << " is prime" << endl;
        else
            cout << num << " is not prime" << endl;

        char choice;
        cout << "Do you want to continue? (y/n): ";
```

```

        cin >> choice;
        if (choice == 'n' || choice == 'N')
            break;

    } while (true);
    cin.get();
    return 0;
}

long long mulmod(long long a, long long b, long long m)
{
    long long x = 0,
               y = a % m;
    while (b > 0)
    {
        if (b % 2 == 1)
        {
            x = (x + y) % m;
        }
        y = (y * 2) % m;
        b /= 2;
    }
    return x % m;
}

long long modulo(long long base, long long e, long long m)
{
    long long x = 1;
    long long y = base;
    while (e > 0)
    {
        if (e % 2 == 1)
            x = (x * y) % m;
        y = (y * y) % m;
        e = e / 2;
    }
    return x % m;
}

bool Miller(long long p, int iteration)
{
    if (p < 2)
    {
        return false;
    }

```

```

if (p != 2 && p % 2 == 0)
{
    return false;
}
long long s = p - 1;
while (s % 2 == 0)
{
    s /= 2;
}
for (int i = 0; i < iteration; i++)
{
    long long a = rand() % (p - 1) + 1, temp = s;
    long long mod = modulo(a, temp, p);
    while (temp != p - 1 && mod != 1 && mod != p - 1)
    {
        mod = mulmod(mod, mod, p);
        temp *= 2;
    }
    if (mod != p - 1 && temp % 2 == 0)
    {
        return false;
    }
}
return true;
}

```

Output:

```
D:\College\5th Semester\Cryp × + ∨
Enter integer to test primality: 37
37 is prime
Do you want to continue? (y/n): y
Enter integer to test primality: 28
28 is not prime
Do you want to continue? (y/n): y
Enter integer to test primality: 99
99 is not prime
Do you want to continue? (y/n): n

-----
Process exited after 16.32 seconds with return value 0
Press any key to continue . . . |
```

Conclusion:

Hence, in this way we can use and implement Miller-Rabin Primality test in the laboratory.

Task 2:

Calculate $\phi(n)$ (Euler's Totient Function) for a given positive integer 'n.' Verify its correctness for multiple values of 'n.'

Theory:

Euler's Totient function $\phi(n)$ for an input n is the count of numbers in $\{1, 2, 3, \dots, n-1\}$ that are relatively prime to n , i.e., the numbers whose GCD (Greatest Common Divisor) with n is 1.

Source Code:

```
#include <iostream>
using namespace std;

void computeTotient(int);

int main()
{
    int n;
    do
    {
        cout << "Enter a positive integer: ";
        cin >> n;
        computeTotient(n);

        cout << "Do you want to continue? (y/n): ";
        char ch;
        cin >> ch;
        if (ch == 'n' || ch == 'N')
            break;

    } while (true);
    return 0;
}

void computeTotient(int n)
{
    long long phi[n + 1];
    for (int i = 1; i <= n; i++)
        phi[i] = i;
    for (int p = 2; p <= n; p++)
    {
```

```

        if (phi[p] == p)
        {
            phi[p] = p - 1;
            for (int i = 2 * p; i <= n; i += p)
            {
                phi[i] = (phi[i] / p) * (p - 1);
            }
        }
    }

    cout << "Totient value of " << n << ": " << phi[n] << endl;
}

```

Output:

```

D:\College\5th Semester\Cryp x + v
Enter a positive integer: 36
Totient value of 36: 12
Do you want to continue? (y/n): y
Enter a positive integer: 5
Totient value of 5: 4
Do you want to continue? (y/n): y
Enter a positive integer: 76
Totient value of 76: 36
Do you want to continue? (y/n): y
Enter a positive integer: 79
Totient value of 79: 78
Do you want to continue? (y/n): n

-----
Process exited after 14.79 seconds with return value 0
Press any key to continue . . . |

```

Conclusion:

Hence, in this way we can calculate Euler Totient Function in the laboratory.

Task 3:

Write a program to apply Fermat's Little Theorem to check if a given number, is a probable prime.

Theory:

Fermat's little theorem states that if p is a prime number, then for any integer a , the number a^{p-1} is an integer multiple of p .

p is prime and ' a ' is a positive integer not divisible by p then

$$a^{p-1} = 1 \pmod{p}$$

Source Code:

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

#define ll long long

ll modulo(ll base, ll exponent, ll mod) {
    ll x = 1;
    ll y = base;
    while (exponent > 0) {
        if (exponent % 2 == 1)
            x = (x * y) % mod;
        y = (y * y) % mod;
        exponent = exponent / 2;
    }
    return x % mod;
}

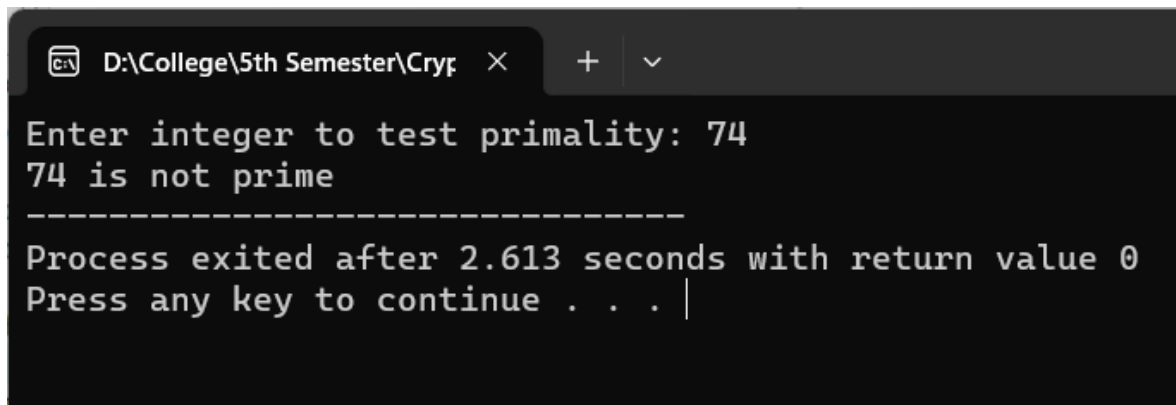
int Fermat(ll p, int iterations) {
    int i;
    if (p == 1) {
        return 0;
    }
    for (i = 0; i < iterations; i++) {
        ll a = rand() % (p - 1) + 1;
        if (modulo(a, p - 1, p) != 1) {
            return 0;
        }
    }
    return 1;
}
```

```

}
int main() {
    int iteration = 50;
    ll num;
    printf("Enter integer to test primality: ");
    scanf("%lld", &num);
    if (Fermat(num, iteration) == 1)
        printf("%lld is probably prime ", num);
    else
        printf("%lld is not prime ", num);
    return 0;
}

```

Output:



```

D:\College\5th Semester\Cryp >
Enter integer to test primality: 74
74 is not prime
-----
Process exited after 2.613 seconds with return value 0
Press any key to continue . . . |

```

Conclusion:

Hence, in this way we can apply Fermat's little Theorem in the laboratory.

Task 4:

Write a program to generate a public and private key using RSA algorithm. Also, encrypt a message "CAB College" and again decrypt it using the algorithm.

Theory:

RSA algorithm is an asymmetric cryptography algorithm. Asymmetric actually means that it works on two different keys i.e. Public Key and Private Key. As the name describes that the Public Key is given to everyone and the Private key is kept private.

RSA uses the following steps for encryption purposes:

1. Select two large prime numbers, p and q .
2. Multiply these numbers to find $n = p \times q$, where n is called the modulus for encryption and decryption.
3. Choose a number e less than n , such that n is relatively prime to $(p - 1) \times (q - 1)$. It means that e and $(p - 1) \times (q - 1)$ have no common factor except 1. Choose " e " such that $1 < e < \phi(n)$, e is prime to $\phi(n)$,
i.e. $\gcd(e, \phi(n)) = 1$
4. If $n = p \times q$, then the public key is $\langle e, n \rangle$. A plaintext message m is encrypted using public key $\langle e, n \rangle$. To find ciphertext from the plain text following formula is used to get ciphertext C .
$$C = m^e \bmod n$$

Here, m must be less than n . A larger message ($>n$) is treated as a concatenation of messages, each of which is encrypted separately.
5. To determine the private key, we use the following formula to calculate the d such that:
$$De \bmod \phi(n) = 1$$
6. The private key is $\langle d, n \rangle$. A ciphertext message c is decrypted using private key $\langle d, n \rangle$. To calculate plain text m from the ciphertext c following formula is used to get plain text m .
$$m = c^d \bmod n$$

Source Code:

```
#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;
}
string decoder(vector<int> encoded)
{
    string s;
    for (auto& num : encoded)
        s += decrypt(num);
    return s;
}
int main()
{
    primefiller();
    setkeys();
    string message = "CAB College";

    vector<int> coded = encoder(message);
    cout << "Initial message:\n" << message;
    cout << "\n\nThe encoded message(encrypted by public "

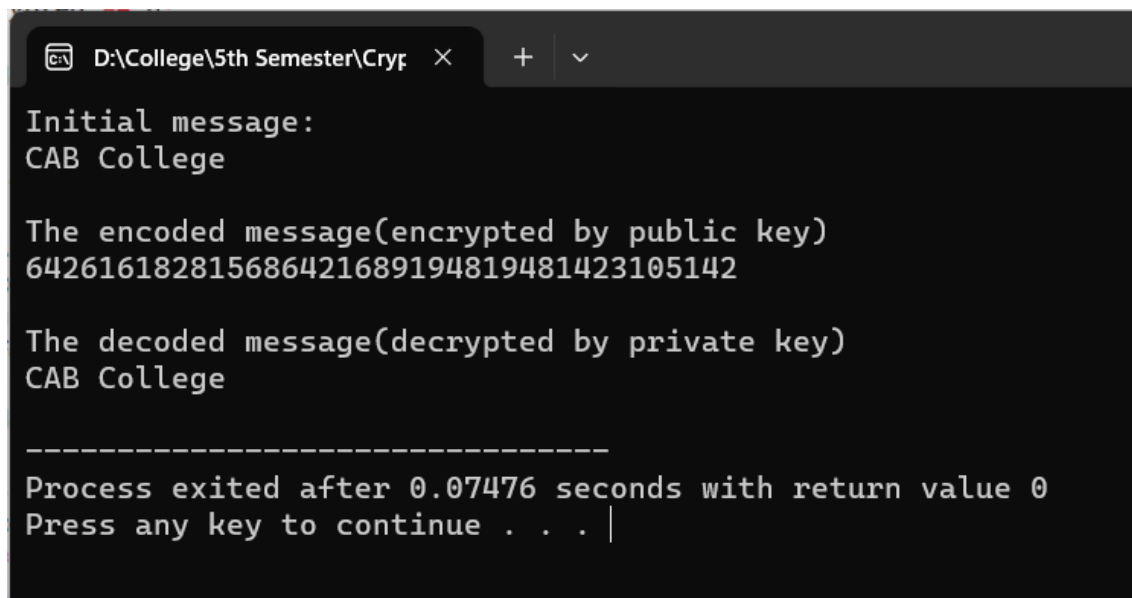
```

```

        "key)\n";
    for (auto& p : coded)
        cout << p;
    cout << "\n\nThe decoded message(decrypted by private "
        "key)\n";
    cout << decoder(coded) << endl;
    return 0;
}

```

Output:



```

Initial message:
CAB College

The encoded message(encrypted by public key)
642616182815686421689194819481423105142

The decoded message(decrypted by private key)
CAB College

-----
Process exited after 0.07476 seconds with return value 0
Press any key to continue . . . |

```

Conclusion:

Hence, in this way we can implement encryption and decryption using RSA algorithm in the laboratory.

Task 5:

Write a program to calculate the Key for two persons using the Diffie Hellman Key exchange algorithm.

Theory:

The Diffie-Hellman algorithm is used to establish a shared secret that can be used for secret communications while exchanging data over a public network using the elliptic curve to generate points and get the secret key using the parameters.

For the sake of simplicity and practical implementation of the algorithm, we will consider only 4 variables, one prime P and G (a primitive root of P) and two private values a and b. P and G are both publicly available numbers. Users (say Alice and Bob) pick private values a and b and they generate a key and exchange it publicly. The opposite person receives the key and that generates a secret key, after which they have the same secret key to encrypt.

Alice	Bob
Public Keys available = P, G	Public Keys available = P, G
Private Key Selected = a	Private Key Selected = b
Key generated = $x = G^a \text{ mod } P$	Key generated = $y = G^b \text{ mod } P$
Exchange of generated keys takes place	
Key received = y	key received = x
Generated Secret Key = $k_a = y^a \text{ mod } P$	Generated Secret Key = $k_b = x^b \text{ mod } P$
Algebraically, it can be shown that $k_a = k_b$	
Users now have a symmetric secret key to encrypt	

Source Code:

```
#include <cmath>
#include <iostream>

using namespace std;

long long int power(long long int a, long long int b,
                    long long int P)
{
    if (b == 1)
        return a;

    else
        return (((long long int)pow(a, b)) % P);
}

// Driver program
int main()
{
    long long int P, G, x, a, y, b, ka, kb;

    // Both the persons will be agreed upon the
    // public keys G and P

    P = 23; // A prime number P is taken
    cout << "The value of P : " << P << endl;

    G = 5; // A primitive root for P, G is taken
    cout << "The value of G : " << G << endl;

    // Alice will choose the private key a
    a = 4; // a is the chosen private key
    cout << "The private key a for Alice : " << a << endl;

    x = power(G, a, P); // gets the generated key

    // Bob will choose the private key b
    b = 3; // b is the chosen private key
    cout << "The private key b for Bob : " << b << endl;

    y = power(G, b, P); // gets the generated key

    // Generating the secret key after the exchange
    // of keys
```

```

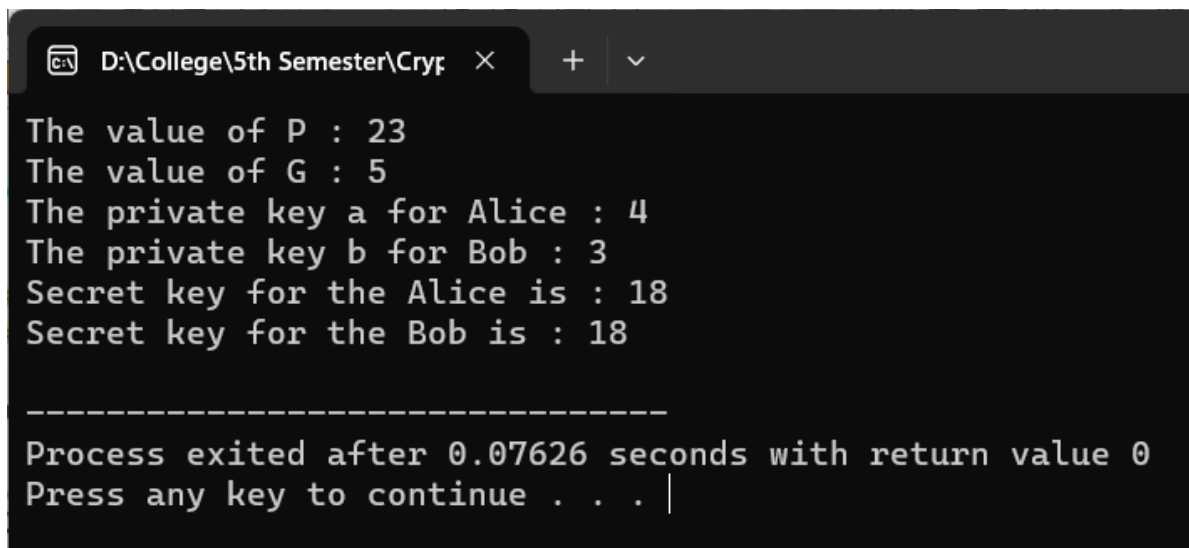
    ka = power(y, a, P); // Secret key for Alice
    kb = power(x, b, P); // Secret key for Bob
    cout << "Secret key for the Alice is : " << ka << endl;

    cout << "Secret key for the Bob is : " << kb << endl;

    return 0;
}

```

Output:



The screenshot shows a terminal window with a dark background. The title bar at the top indicates the file path is 'D:\College\5th Semester\Cryp'. The output of the program is as follows:

```

The value of P : 23
The value of G : 5
The private key a for Alice : 4
The private key b for Bob : 3
Secret key for the Alice is : 18
Secret key for the Bob is : 18

-----
Process exited after 0.07626 seconds with return value 0
Press any key to continue . . . |

```

Conclusion:

Hence, in this way we can observe key generation and sharing using Diffie-Helman Key Sharing Algorithm in the laboratory.