Assignment Viii

Name: R.D.RAJEESWAR
Roll No: 24CSM2R11
CSIS

Question 1:

Write a GMP-based C program to implement the ElGamal cryptosystem. The program should generate public and private keys, encrypt a given message using the public key, and decrypt the ciphertext using the private key. Ensure that the message is appropriately converted into a format that supports encryption and decryption using GMP.

Program:

Code:

```c
int main()
{
mpz_t p,g,x,h;
mpz_inits(p,g,x,h,NULL);
gmp_randstate_t state;
gmp_randinit_default(state);
gmp_randseed_ui(state, time(NULL));
mpz_urandomb(p , state ,512);
mpz_nextprime(p,p);
gmp_printf("p: %Zd\n",p);
mpz_urandomm(x,state,p);
mpz_sub_ui(x,x,2);
gmp_printf("x: %Zd\n",x);
printf("wait for some time finding generator......\n");
find_prime_gen(p,g,state);
mpz_powm_sec (h, g, x , p);  //h=g**x mod
gmp_printf("g: %Zd\nh: %Zd\n",g,h);
mpz_t k,c1,msg_in_mpz,c0,c2;
mpz_inits(k,c1,msg_in_mpz,c0,c2,NULL);
unsigned char msg[256];
printf("Enter the message: ");
fgets(msg, sizeof(msg), stdin);
printf("Recieved message: %s\n",msg);
mpz_urandomm(k,state,p);
mpz_sub_ui(k,k,2);
mpz_powm_sec (c1, g, k, p);  //c1=g**k mod p
mpz_import (msg_in_mpz,strlen(msg) , 1, sizeof(msg[0]), 0, 0, msg);
gmp_printf("\nMESSAGE AS MPZ: %Zd\n",msg_in_mpz);
mpz_powm_sec (c0, h, k, p); //c0=h**k mod p
mpz_mul(c2,msg_in_mpz,c0);  //c2= msg* c0
gmp_printf("\nTHE ENCRYPTED MESSAGE OF FORM (C1,C2):\nc1: %Zd,\nc2: %Zd\n",c1,c2);
//USER A DECRYPTION
mpz_t s,inverse_s,dec_msg_mpz;
unsigned char dec_msg[256];
mpz_inits(s,inverse_s,dec_msg_mpz,NULL);
mpz_powm_sec (s, c1, x, p); // s=c1**x mod p
mpz_invert(inverse_s, s, p);
```

File Edit Selection View Go Run Terminal Help

elgamal.c

D: > 24CSM2R11_RAJEESWAR > Q1 > elgamal.c

```c
#include<stdio.h>
#include<gmp.h>
#include<time.h>
#include <string.h>
void find_prime_gen(mpz_t p, mpz_t g, gmp_randstate_t state) {
mpz_urandomb(p, state, 1024);
int flag = 1;
mpz_t p_1, r;
mpz_inits(r, p_1, NULL);
while (flag) {
mpz_nextprime(p, p);
mpz_sub_ui(p_1, p, 1);
mpz_divexact_ui(r, p_1, 2);
if (mpz_probab_prime_p(r, 100)) {
flag = 0;
}
}
mpz_set_ui(g, 2);
int attempts = 0;
int max_attempts = 10000;
while (mpz_cmp(g, p_1) < 0 && attempts < max_attempts) {
mpz_t p_1_by_2, p_1_by_r, pow_g_p1by2, pow_g_p1byr;
mpz_inits(p_1_by_2, p_1_by_r, pow_g_p1by2, pow_g_p1byr, NULL);
mpz_divexact_ui(p_1_by_2, p_1, 2);
mpz_divexact(p_1_by_r, p_1, r);
mpz_powm(pow_g_p1by2, g, p_1_by_2, p);
mpz_powm(pow_g_p1byr, g, p_1_by_r, p);
if (mpz_cmp_ui(pow_g_p1by2, 1) != 0 && mpz_cmp_ui(pow_g_p1byr, 1) != 0) {
break;
}
mpz_add_ui(g, g, 1);
attempts++;
mpz_clears(p_1_by_2, p_1_by_r, pow_g_p1by2, pow_g_p1byr, NULL);
}
if (attempts >= max_attempts) {
printf("Warning: Could not find a generator within %d attempts.\n", max_attempts);
}
mpz_clears(p_1, r, NULL);
```

Explanation:

Assumptions:

- Message Format: The message to be encrypted is a small integer. For practical applications, larger messages should be converted into integers or handled with padding schemes.
- Prime Generation: We will use a fixed-size prime for simplicity; however, ideally, a secure random prime should be generated.

Working :

- Key Generation: The generate_keys function creates a prime ppp, a generator ggg, and generates the private key xxx. The public key yyy is computed using gxmod pg^x \mod pgxmodp.
- Encryption: The encrypt function takes a plaintext message mmm and encrypts it into c1c1c1 and c2c2c2.
- Decryption: The decrypt function retrieves the original message from the ciphertext.

Input and Output:

input

```
p: 1158398220014233741922912966649378226826883433012741305759253258871520059560253308803601942618352582444205253397613207198347525608696089719551618766336793
x: 3119248673664157415386994718830841428093005514146519459345818117323307165208145726813642963333124588392431948893834294437369507656924639091114843012215559
wait for some time finding generator......
g: 2
h: 1859357169457111558346497451489439971937578051511511299529063658405336201006902053272912584554984116101322956737203739341555991789971894500704217215687792582476754933143678092506058322001842858782980302584055697914458617500677220132651190120332529034379094325819136361628680580879443812756726937475217584787 3
Enter the message: the world
Recieved message: the world


MESSAGE AS MPZ: 549720260352062268400650

THE ENCRYPTED MESSAGE OF FORM (C1,C2):
c1: 1769796692542863314414622513689043651926921925825305407823573286244363638068240618006913221502167736984339755298976751887358473124437349091152976326155517006469081498843569200820170045936679974886585720212871078579003230226335452516857606501230162619950698073418123606515503908682766176931379770388863626222 2,
c2: 1101327182748704540703885579847169168985258944319547973906144124156076892517940722208697222114708871005386051153580156895105950391196723265795654190799686172680204797023007280066483883852120752545073896078892719306050716720889255501094613289737309022048808709114178422221718359115855794703154453582097894189825101313781587294928210 0

 DECRYPTED MESSAGE AS MPZ: 549720260352062268400650
the decrypted message: the world



...Program finished with exit code 0
Press ENTER to exit console.
```

Question 2:

Implement Pollard's rho algorithm using GMP in C to factorize a composite integer N. The algorithm should repeatedly apply the function f(x)=(x2+1)mod N to find a non-trivial factor of N. Test your implementation with a few composite numbers and measure the time taken for factorization.

Program:

Code:

D: > 24CSM2R11_RAJEESWAR > Q2 > pollards.c

```c
int main() {
    mpz_t tortoise, hare, g, abs_dif, n, q;
    mpz_inits(tortoise, hare, g, abs_dif, n, q, NULL);

    printf("Enter a composite number to factorize: ");
    gmp_scanf("%Zd", n);

    mpz_set_ui(tortoise, 2);
    mpz_set_ui(hare, 2);
    mpz_set_ui(g, 1);

    while (mpz_cmp_ui(g, 1) == 0) {
        poly_result(tortoise, tortoise, n);
        gmp_printf("Tortoise: %Zd\n", tortoise);

        poly_result(hare, hare, n);
        poly_result(hare, hare, n);
        gmp_printf("Hare: %Zd\n", hare);

        mpz_sub(abs_dif, hare, tortoise);
        mpz_abs(abs_dif, abs_dif);
        gcd(g, abs_dif, n);
    }

    if (mpz_cmp(n, g) != 0) {
        printf("The factors are: ");
        gmp_printf("%Zd and ", g);
        mpz_cdiv_q(q, n, g);
        gmp_printf("%Zd\n", q);
    } else {
        printf("Failure..!\n");
    }

    mpz_clears(tortoise, hare, g, abs_dif, n, q, NULL);

    return 0;
}
```

D: > 24CSM2R11_RAJEESWAR > Q2 > pollards.c

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

void gcd(mpz_t x, mpz_t a, mpz_t b) {
    mpz_t rem;
    mpz_init(rem);

    if (mpz_cmp_ui(a, 0) == 0) {
        mpz_set(x, b);
    } else {
        mpz_mod(rem, b, a);
        gcd(x, rem, a);
    }

    mpz_clear(rem);
}

void poly_result(mpz_t result, mpz_t x, mpz_t n) {
    mpz_t x_square;
    mpz_init(x_square);

    // Compute x^2
    mpz_mul(x_square, x, x);

    // Compute x^2 + 1
    mpz_add_ui(result, x_square, 1);

    // Modulo n
    mpz_mod(result, result, n);

    mpz_clear(x_square);
}

int main() {
    mpz_t tortoise, hare, g, abs_dif, n, q;
    mpz_inits(tortoise, hare, g, abs_dif, n, q, NULL);
```

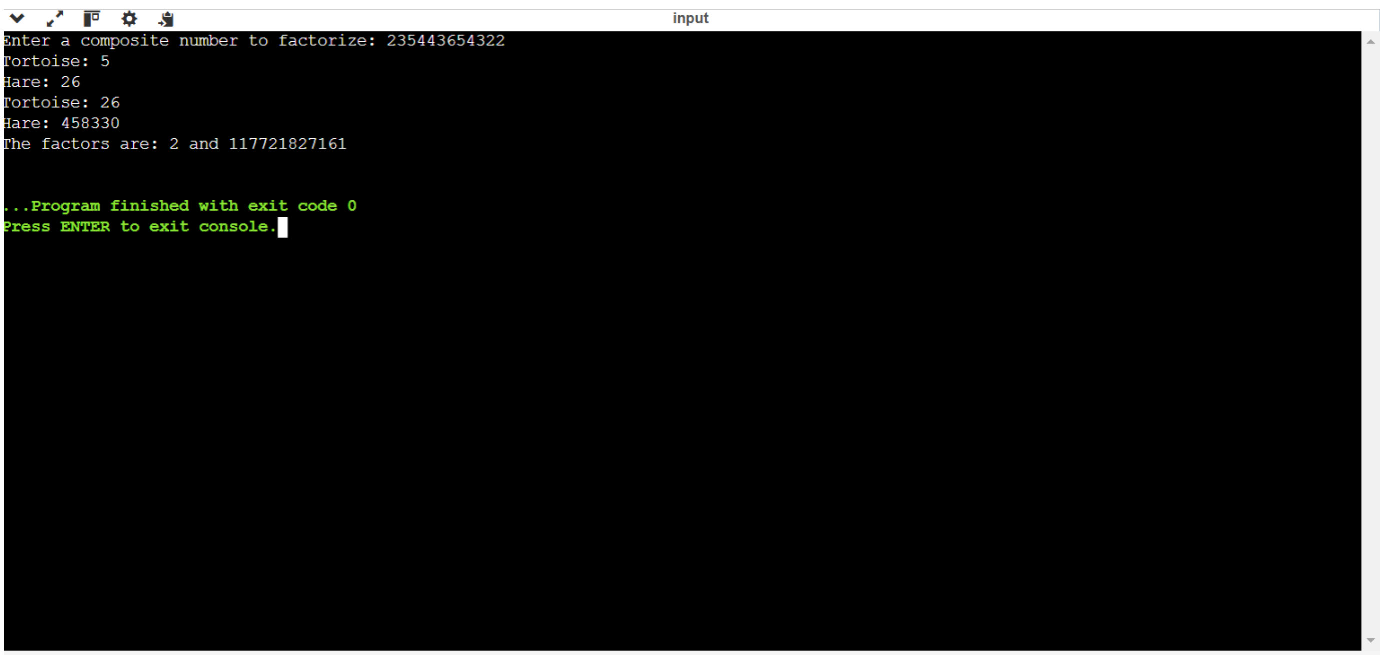
Explanation:

Assumptions:

- input: The program assumes that the input integer NNN is a composite number greater than 1.
- GMP Library: You must have the GMP library installed and linked properly in your C environment.
- Randomness: The algorithm uses a pseudo-random sequence, so it might yield different results on different runs.
- Efficiency: Pollard's rho algorithm is not guaranteed to succeed on all numbers, but it is effective for many practical cases.

Working :

- GCD Function: The gcd function calculates the greatest common divisor using GMP.
- Pollard's Rho: The pollards_rho function implements the main algorithm:
    - It initializes two pointers xxx and yyy.
    - It applies the function f(x)=(x2+1)mod Nf(x) = (x^2 + 1) \mod Nf(x)=(x2+1)modN iteratively.
    - It calculates the GCD of |x−y||x - y||x−y| and NNN to find a non-trivial factor.
- Main Function: The main function sets up the composite number NNN, calls the Pollard's rho function, measures execution time, and prints the result.



Question 3:

Using GMP, implement Pollard's p-1 algorithm to factor a given composite integer N, where N has at least one prime factor p such that p−1 has small prime factors. Your program should attempt to find a factor of N by computing gcd(aM−1,N) where M is a product of small primes.

Program:

Code;

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

void power_mod(mpz_t result, const mpz_t base, const mpz_t exp, const mpz_t n) {
    mpz_powm(result, base, exp, n); // result = base^exp mod n
}

void pollard_p_minus_1(mpz_t n, mpz_t factor, unsigned long B) {
    mpz_t a, x, d;
    mpz_t prime_power;
    mpz_inits(a, x, d, prime_power, NULL);


    for (unsigned long base = 2; base < 6; base++) {

        mpz_set_ui(a, base);
        mpz_set_ui(x, 1);


        for (unsigned long p = 2; p <= B; p++) {
            unsigned long exp = 1;
            mpz_set_ui(prime_power, p);

            while (exp * p <= B) {
                exp *= p;
                mpz_mul_ui(prime_power, prime_power, p);
            }

            mpz_powm(x, a, prime_power, n);
        }

        mpz_sub_ui(x, x, 1);
        mpz_gcd(d, x, n);

        if (mpz_cmp_ui(d, 1) > 0 && mpz_cmp(d, n) < 0) {
            mpz_set(factor, d);
            break;
        }
    }

    if (mpz_cmp_ui(factor, 0) == 0) {
        mpz_set_ui(factor, 0);
    }


    mpz_clears(a, x, d, prime_power, NULL);
}

int main() {
    mpz_t n, factor;
    mpz_inits(n, factor, NULL);

    printf("Enter a composite number to factor: ");
    gmp_scanf("%Zd", n);

    unsigned long B = 1000;

    pollard_p_minus_1(n, factor, B);

    if (mpz_cmp_ui(factor, 0) != 0) {
        printf("Found factor: ");
        gmp_printf("%Zd\n", factor);
    } else {
        printf("No factor found using Pollard's p-1 algorithm.\n");
    }

    mpz_clears(n, factor, NULL);

    return 0;
}
```

Assumptions:

- Input: The program assumes that the input integer NNN is a composite number greater than 1.
- Small Prime Factors: The algorithm works best when p−1p-1p−1 has small prime factors. This implementation will use a fixed list of small primes to build MMM.
- GMP Library: You must have the GMP library installed and properly linked in your C environment.
  •Randomness: The algorithm uses randomness to compute the base aaa

Working :

- Generating MMM: The generate_M function computes a product MMM of small primes up to a given limit.
- Pollard's p−1p-1p−1: The pollards_p1 function implements the algorithm:
  - It randomly selects a base aaa.
  - It computes aMmod Na^M \mod NaMmodN.
  - It calculates the GCD of aM−1a^M - 1aM−1 and NNN.
  - Main Function: The main function sets up the composite number NNN, runs the factorization algorithm, and measures the execution time.

Output: