密码学原理与实践

实

验

报

告

学院名称 智能与计算学部

专　　业 网络空间安全

学生姓名 刘宏伟

学　　号 3020205094

年 班 级 2021 级 1 班

2024 年 11 月 27 日

实验 1 设计一个 RSA 加密算法

实验原理

算法1 RSA公钥加密的密钥产生

摘要：每一个实体产生RSA的公开和对应的秘密密钥。

每个实体A做如下步骤：

(1) 产生两个大随机(不同)的素数p和q，两个素数的长度基本相等。

(2) 计算 $n = p \cdot q$ 和 $\varphi = (p-1)(q-1)$。

(3) 选择一个随机的整数e，$1 < e < \varphi$，满足最大公约数(e，φ) = 1。

(4) 使用扩展Euclidean算法计算唯一的整数d，$1 < d < \varphi$，满足$e \cdot d \equiv 1 (\bmod \varphi)$。

(5) A的公开密钥为(n，e); A的秘密密钥为d。

算法2 RSA 公钥加密

摘要：实体B加密一条明文消息m给实体A，实体A解密。

加密. B做如下步骤：

(1) 得到A的真实公开密钥(n，e)。

(2) 将明文消息m表示为在[0，$n-1$]之间的整数。

(3) 计算 $c \equiv m^e (\bmod n)$。

(4) 发送密文消息c给 A。

解密. 为了从密文消息c恢复明文消息m，A应该做如下操作：

(1) 使用秘密密钥d恢复明文消息$m \equiv c^d (\bmod n)$。

算法3 Fermat素性测试

输入：一个奇整数$n > 3$和一个安全参数$t \geq 1$。

输出：n是素数或合数的判断。

(1) For i from 1 to t do the following:

(1.1) 随机选择一个整数a，$2 \leq a \leq n-2$。

(1.2) 计算 $r \equiv a^{n-1} (\bmod n)$。

(1.3) 如果 $r \neq 1$，则 return("n是合数")。

(2) Return("n是素数")。

如果 n 是合数并且 $a^{n-1} \equiv 1 (\bmod n)$，则我们说$n$是对基$a$的一个伪素数。

算法4从左向右二进制算法

输入：整数g，n以及一个正整数$b=(b_t b_{t-1}\cdots b_1 b_0)_2$。

输出：$g^b(\bmod n)$。

(1) $A=1$。

(2) For i from t down to 0 do the following:

(2.1) $A\equiv A\cdot A(\bmod n)$。

(2.2) 如果$b_i=1$，则$A\equiv A\cdot g(\bmod n)$。

(3) Return (A)。

实验要求

实现加解密功能。

加密

输入：p，q，e，m

输出：c，d

解密

输入：p，q，d，c

输出：m，e

(需要对p和q做素性测试;需要对e和d做参数检查;输入参数为 unsigned long 型，考虑计算中间结果溢出情况的为佳)

实验过程

使用 C++在 Kali Linux 上链接 GMP 库完成，源代码如下：

myreport - practice_1.cpp

```cpp
#include<iostream>
#include<gmp.h>

// 计算 (base^exponent)%mod
void power(mpz_t result, mpz_t base, mpz_t exponent, mpz_t mod) {
    mpz_powm(result, base, exponent, mod);
}

// 判断一个数是否为素数
bool is_prime(mpz_t n) {
    return mpz_probab_prime_p(n, 10) > 0;
}

// 计算模反元素，用于生成私钥指数 d
bool mod_inverse(mpz_t result, mpz_t a, mpz_t mod) {
    return mpz_invert(result, a, mod) != 0;
}

// RSA加密
void rsa_encrypt(mpz_t c, mpz_t d, mpz_t p, mpz_t q, mpz_t e, mpz_t m) {
    if (!is_prime(p) || !is_prime(q)) {
        std::cout << "错误: p 和 q 必须是素数。" << std::endl;
        return;
    }

    mpz_t n, phi, p_minus_1, q_minus_1;
    mpz_inits(n, phi, p_minus_1, q_minus_1, NULL);

    // 计算 n = p * q
    mpz_mul(n, p, q);

    // 计算 φ(n) = (p-1) * (q-1)
    mpz_sub_ui(p_minus_1, p, 1);
    mpz_sub_ui(q_minus_1, q, 1);
    mpz_mul(phi, p_minus_1, q_minus_1);

    // 验证 e 是否与 φ(n) 互素，并计算私钥指数 d
    if (!mod_inverse(d, e, phi)) {
        std::cout << "错误: e 必须与 (p-1)*(q-1) 互素。" << std::endl;
        return;
    }

    // 计算密文 c = m^e mod n
    power(c, m, e, n);

    mpz_clears(n, phi, p_minus_1, q_minus_1, NULL);
}

// RSA解密
void rsa_decrypt(mpz_t m, mpz_t e, mpz_t p, mpz_t q, mpz_t d, mpz_t c) {
    if (!is_prime(p) || !is_prime(q)) {
        std::cout << "错误: p 和 q 必须是素数。" << std::endl;
        return;
    }

    mpz_t n;
    mpz_init(n);

    // 计算 n = p * q
    mpz_mul(n, p, q);

    // 计算明文 m = c^d mod n
    power(m, c, d, n);

    // 计算公钥指数 e
    mpz_t phi, p_minus_1, q_minus_1;
    mpz_inits(phi, p_minus_1, q_minus_1, NULL);

    // 计算 φ(n) = (p-1) * (q-1)
    mpz_sub_ui(p_minus_1, p, 1);
    mpz_sub_ui(q_minus_1, q, 1);
    mpz_mul(phi, p_minus_1, q_minus_1);

    // 计算 e
    mod_inverse(e, d, phi);

    mpz_clears(n, phi, p_minus_1, q_minus_1, NULL);
}

int main() {
    mpz_t p, q, e, m, d, c, decrypted_m;
    mpz_inits(p, q, e, m, d, c, decrypted_m, NULL);
    int choice;

    std::cout << "请选择操作: " << std::endl;
    std::cout << "1. 加密" << std::endl;
    std::cout << "2. 解密" << std::endl;
    std::cin >> choice;

    if (choice == 1) {
        gmp_printf("请输入第一个素数 p: ");
        gmp_scanf("%Zd", p);
        gmp_printf("请输入第二个素数 q: ");
        gmp_scanf("%Zd", q);
        gmp_printf("请输入公钥指数 e: ");
        gmp_scanf("%Zd", e);
        gmp_printf("请输入需要加密的明文 m: ");
        gmp_scanf("%Zd", m);

        rsa_encrypt(c, d, p, q, e, m);
        gmp_printf("加密后的密文为: %Zd\n", c);
        gmp_printf("私钥指数 d 为: %Zd\n", d);
    }
    else if (choice == 2) {
        gmp_printf("请输入第一个素数 p: ");
        gmp_scanf("%Zd", p);
        gmp_printf("请输入第二个素数 q: ");
        gmp_scanf("%Zd", q);
        gmp_printf("请输入私钥指数 d: ");
        gmp_scanf("%Zd", d);
        gmp_printf("请输入需要解密的密文 c: ");
        gmp_scanf("%Zd", c);

        rsa_decrypt(decrypted_m, e, p, q, d, c);
        gmp_printf("解密后的明文为: %Zd\n", decrypted_m);
        gmp_printf("公钥指数 e 为: %Zd\n", e);
    }
    else {
        std::cout << "无效选择，程序退出。" << std::endl;
    }

    mpz_clears(p, q, e, m, d, c, decrypted_m, NULL);
    return 0;
}
```

测试结果

```
┌──(pama㉿kali)-[~/Desktop/myreport]
└─$ ./practice_1
请选择操作:
1. 加密
2. 解密
1
请输入第一个素数 p: 2357
请输入第二个素数 q: 2551
请输入公钥指数 e: 3674911
请输入需要加密的明文 m: 3650502
加密后的密文为: 227060
私钥指数 d 为: 422191
```

```
┌──(pama㉿kali)-[~/Desktop/myreport]
└─$ ./practice_1
请选择操作:
1. 加密
2. 解密
2
请输入第一个素数 p: 885320963
请输入第二个素数 q: 238855417
请输入私钥指数 d: 116402471153538991
请输入需要解密的密文 c: 113535859035722866
解密后的明文为: 30120
公钥指数 e 为: 9007
```

实验 2 设计一个 RSA 小加密指数 e 的攻击算法

实验原理

为了改进加密效率，我们希望选择小的加密指数 e，例如，e=3。一组实体可能选择相同小加密指数e。如果一个实体 A希望发送同一条消息m给三个实体他们的公开模为n_1，n_2，n_3，而他们的加密指数都是 e=3，则A将发送$c_i \equiv m^3 (\bmod n_i)$，这里 i=1,2,3。由于这些模是两两互素，而攻击者可以在公共信道上得到c_1，c_2，c_3，因此可以根据中国剩余定理得到一个解$x, 0 \le x < n_1 \cdot n_2 \cdot n_3$，满足三个同余式

$$\begin{cases} x \equiv c_1 (\bmod n_1) \\ x \equiv c_2 (\bmod n_2) \\ x \equiv c_3 (\bmod n_3) \end{cases}。$$

由于$m^3 < n_1 \cdot n_2 \cdot n_3$，就必然有 $x = m^3$。因此，通过计算整数x的三次方根，攻击者可以恢复出明文消息m。因此，e = 3这样的小加密指数不能使用在同一消息加密群发的情形。小加密指数在加密小消息m时同样存在问题，这是因为如果$m < n^{1/e}$，则密文$c \equiv m^e (\bmod n)$就可以通过简单计算c的e次方根得到。

$$x \equiv 23(\bmod 105) \Rightarrow \begin{cases} x \equiv 2(\bmod 3) \\ x \equiv 3(\bmod 5) \\ x \equiv 2(\bmod 7) \end{cases}$$

中国剩余定理揭示这一过程是可逆的。

定理1设m_1，$m_2,\cdots$，m_k是k个两两互素的正整数，

$m = m_1 m_2 \cdots m_k$，$m = m_i M_i (i = 1,2,\cdots，k)$则同余式组

$x \equiv b_1(\bmod m_1)$，$x \equiv b_2(\bmod m_2),\cdots$，$x \equiv b_k(\bmod m_k)$

有唯一解

$$x \equiv M_1' M_1 b_1 + M_2' M_2 b_2 + \cdots + M_k' M_k b_k (\bmod m),$$

其中

$$M_i' M_i \equiv 1(\bmod m_i)(i = 1,2,\cdots，k)。$$

实验要求

给出 RSA 的公开参数和密文，可以恢复出群发明文。

输入：e=3，n_1，n_2，n_3，c_1，c_2，c_3

输出：m

（需要对 n_1、n_2、n_3 做互素测试；输入参数为 unsigned long 型，考虑计算中间结果溢出情况的为佳）

实验过程

使用 C++在 Kali Linux 上链接 GMP 库完成，源代码如下：

myreport - practice_2.cpp

```cpp
#include <iostream>
#include <gmp.h>

// 检查两个数是否互素
bool is_coprime(mpz_t a, mpz_t b) {
    mpz_t gcd;
    mpz_init(gcd);
    mpz_gcd(gcd, a, b);
    bool result = mpz_cmp_ui(gcd, 1) == 0;
    mpz_clear(gcd);
    return result;
}

// 计算模反元素
void modInverse(mpz_t a, mpz_t m, mpz_t inv) {
    mpz_t g, x, y;
    mpz_inits(g, x, y, NULL);
    mpz_gcdext(g, x, y, a, m);
    if (mpz_cmp_ui(g, 1) != 0) {
        std::cerr << "错误: 无法计算模反元素, 输入的值不是互素的。" << std::endl;
        exit(EXIT_FAILURE);
    }
    mpz_mod(inv, x, m); // 确保结果非负
    mpz_clears(g, x, y, NULL);
}

int main() {
    mpz_t e, n[3], c[3], m, N, prod, inv, temp;
    mpz_inits(e, m, N, prod, inv, temp, NULL);
    for (int i = 0; i < 3; i++) {
        mpz_inits(n[i], c[i], NULL);
    }

    std::cout << "请输入小加密指数 e (例如 3): ";
    mpz_inp_str(e, stdin, 10);

    std::cout << "请输入三个互素的模数 n1, n2, n3 (每行一个):" << std::endl;
    for (int i = 0; i < 3; i++) {
        mpz_inp_str(n[i], stdin, 10);
    }

    // 检查模数是否两两互素
    if (!is_coprime(n[0], n[1]) || !is_coprime(n[0], n[2]) || !is_coprime(n[1], n[2])) {
        std::cerr << "错误: 模数 n1, n2, n3 必须两两互素。" << std::endl;
        exit(EXIT_FAILURE);
    }

    std::cout << "请输入对应的密文 c1, c2, c3 (每行一个):" << std::endl;
    for (int i = 0; i < 3; i++) {
        mpz_inp_str(c[i], stdin, 10);
    }

    // 计算所有 n 的乘积 N
    mpz_set_ui(N, 1);
    for (int i = 0; i < 3; i++) {
        mpz_mul(N, N, n[i]);
    }

    // 使用中国剩余定理恢复合并的密文
    mpz_set_ui(m, 0);
    for (int i = 0; i < 3; i++) {
        // 计算 N / n[i]
        mpz_div(prod, N, n[i]);

        // 计算 (N / n[i]) 的模反元素
        modInverse(prod, n[i], inv);

        // temp = c[i] * (N / n[i]) * inv
        mpz_mul(temp, c[i], prod);
        mpz_mul(temp, temp, inv);

        // 累加到 m
        mpz_add(m, m, temp);
        mpz_mod(m, m, N);
    }

    // 计算 m 的 e 次根, 恢复原始明文
    if (!mpz_root(m, m, mpz_get_ui(e))) {
        std::cerr << "错误: 无法计算 e 次方根, 可能输入的参数不正确。" << std::endl;
        exit(EXIT_FAILURE);
    }

    std::cout << "恢复的明文消息 m 为: ";
    mpz_out_str(stdout, 10, m);
    std::cout << std::endl;

    mpz_clears(e, m, N, prod, inv, temp, NULL);
    for (int i = 0; i < 3; i++) {
        mpz_clears(n[i], c[i], NULL);
    }

    return 0;
}
```

测试结果

```
┌──(pama㉿kali)-[~/Desktop/myreport]
└─$ g++ practice_2.cpp -lgmp -o practice_2

┌──(pama㉿kali)-[~/Desktop/myreport]
└─$ ./practice_2
请输入小加密指数 e (例如 3): 3
请输入三个互素的模数 n1, n2, n3 (每行一个):
763813
828083
720761
请输入对应的密文 c1, c2, c3 (每行一个):
352596
408368
6728
恢复的明文消息 m 为: 123456
```

实验 3 设计一个生成强素数的算法

实验原理

定义1 设n为一个合数，若对于所有满足$(a, n)=1$的整数a，均有$a^{n-1} \equiv 1 \pmod n$，则称n为Carmicheal数。

事实1 (Carmicheal数的充分必要条件) 合数n是一个Carmicheal数，当且仅当满足下列两个条件：

(1) 整数n是一个无平方数，即n不能被任何素数的平方整除。

(2) 对于任何整除n的素数p，有$p-1$也能整除$n-1$。

事实 2 Carmicheal数的个数为无限多个。已知的最小Carmicheal数为$561 = 3 \cdot 11 \cdot 17$。

算法1(Miller - Rabin测试)

MILLER - RABIN (n, t)

输入：一个奇整数$n > 3$和一个安全参数$t \geq 1$。

输出：一个对于问题"n是素数否?"的答案"素数"或"合数"。

(1) 写 $n-1 = 2^s \cdot r$ 满足r是一个奇数。

(2) For i from 1 to t do the following:

(2.1) 选择一个随机整数a，$2 \leq a \leq n-2$。

(2.2) 计算$y \equiv a^r \pmod n$。

(2.3) 如果$y \neq 1$并且$y \neq -1$则做如下步骤：

(2.3.1) $j = 1$。

(2.3.2) While $j \leq s-1$ and $y \neq -1$ do the following:

(2.3.2.1) 计算$y \equiv y^2 \pmod n$。

(2.3.2.2) 如果$y = 1$，则 return ("合数")。

(2.3.2.3) $j = j+1$。

(2.3.3) 如果$y \neq -1$，则return ("合数")。

(3) Return ("素数")。

一个素数p被称为是强素数，如果存在整数r, s，和t

满足如下条件：

(1) $p-1$有一个大素数因子，为r；

(2) $p+1$有一个大素数因子，为s;并且

(3) $r-1$有一个大素数因子，为t。

实验要求

生成一个指定长度的强素数。

输入：强素数的比特长度 $0<l<=32$

输出：比特长度为 l 的强素数 p， $p+1$ 的大素因子 s， $p-1$ 的大素因子 r， $r-1$ 大素因子 t

（要求：s，r，t 比特长度约为$\lceil l/2 \rceil$，比特长度误差±4；如无符合要求的强素数程序应该给出明确说明）

实验过程

使用 C++在 Kali Linux 上链接 GMP 库完成，源代码如下：

myreport - practice_3.cpp

```cpp
#include <gmp.h>
#include <iostream>
#include <cmath>

// 求最大素因子函数
void largestPrimeFactor(mpz_t n, mpz_t result) {
    mpz_t i, max;
    mpz_inits(i, max, NULL);

    // 处理因数 2
    while (mpz_divisible_ui_p(n, 2)) {
        mpz_set_ui(max, 2);
        mpz_divexact_ui(n, n, 2);
    }

    // 检查奇数因子
    for (mpz_set_ui(i, 3); mpz_cmp(i, n) <= 0; mpz_add_ui(i, i, 2)) {
        while (mpz_divisible_p(n, i)) {
            mpz_set(max, i);
            mpz_divexact(n, n, i);
        }
    }

    mpz_set(result, max);
    mpz_clears(i, max, NULL);
}

// 主函数
int main() {
    mpz_t p, s, r, t, temp;
    mpz_inits(p, s, r, t, temp, NULL);
    gmp_randstate_t state;
    gmp_randinit_mt(state);

    unsigned int l;
    std::cout << "请输入强素数的比特长度 (0 < l <= 32): ";
    std::cin >> l;

    if (l <= 0 || l > 32) {
        std::cerr << "错误: 比特长度必须在 0 < l <= 32 范围内! " << std::endl;
        return 1;
    }

    // 随机生成指定比特长度的数
    mpz_urandomb(p, state, l);

    // 确保是奇数
    if (mpz_even_p(p)) {
        mpz_add_ui(p, p, 1);
    }

    while (true) {
        // 检查 p 是否为素数
        if (mpz_probab_prime_p(p, 25)) {
            // 计算 p+1 的最大素因子
            mpz_add_ui(temp, p, 1);
            largestPrimeFactor(temp, s);

            // 计算 p-1 的最大素因子
            mpz_sub_ui(temp, p, 1);
            largestPrimeFactor(temp, r);

            // 计算 r-1 的最大素因子
            mpz_sub_ui(temp, r, 1);
            largestPrimeFactor(temp, t);

            // 检查 s、r、t 的比特长度是否符合要求
            unsigned int half = std::ceil(l / 2.0);
            if (mpz_sizeinbase(s, 2) >= half - 4 && mpz_sizeinbase(s, 2) <= half + 4 &&
                mpz_sizeinbase(r, 2) >= half - 4 && mpz_sizeinbase(r, 2) <= half + 4 &&
                mpz_sizeinbase(t, 2) >= half - 4 && mpz_sizeinbase(t, 2) <= half + 4) {
                break;
            }
        }

        // 若不满足强素数条件，尝试下一个奇数
        mpz_add_ui(p, p, 2);
    }

    // 输出结果
    std::cout << "强素数 p: ";
    mpz_out_str(stdout, 10, p);
    std::cout << "\np+1 的最大素因子 s: ";
    mpz_out_str(stdout, 10, s);
    std::cout << "\np-1 的最大素因子 r: ";
    mpz_out_str(stdout, 10, r);
    std::cout << "\nr-1 的最大素因子 t: ";
    mpz_out_str(stdout, 10, t);
    std::cout << std::endl;

    mpz_clears(p, s, r, t, temp, NULL);
    gmp_randclear(state);

    return 0;
}
```

测试结果

实验 4 设计一个 k-ary 和窗口译码算法

实验原理

计算g^e的思想：

$e=(f_n f_{n-1}\cdots f_1 f_0)_2 \Rightarrow$ 二进制方法

$\Downarrow$

$e=(e_t e_{t-1}\cdots e_1 e_0)_b \Rightarrow k$-ary方法

e的二进制表示可以分成每个长度为k的块，因此，有$(t+1)\cdot k=n+1$。如果k不能整除$n+1$，则最多在指数的最前端添加$k-1$个0。

我们可以定义

$$e_i=(f_{i\cdot k+k-1} f_{i\cdot k+k-2}\cdots f_{i\cdot k})_2=\sum_{j=0}^{k-1} f_{i\cdot k+j}\cdot 2^j。$$

注意$0\le e_i\le 2^k-1$和$e=\sum_{i=0}^{t} e_i\cdot 2^{k\cdot i}$。$k$-ary方法首先计算$g^i$的值，这里$i=2,3,\cdots,2^k-1$。接着依次扫描$e$的$k$比特位。

算法3 从左向右 k-ary模幂

输入：g 和 正整数$e=(e_t e_{t-1}\cdots e_1 e_0)_b$，这里 $b=2^k, k\ge 1$。

输出：g^e。

(1) 预计算.

(1.1) $g_1\leftarrow g$。

(1.2) For i from 2 to (2^k-1) do：$g_i\leftarrow g_{i-1}\cdot g$。(因此，$g_i=g^i$。)

(2) $A\leftarrow 1$。

(3) For i from t down to 0 do the following：

(3.1) 如果$i\ne t$，则$A\leftarrow A^{2^k}$。

(3.2) 如果 $e_i\ne 0$，则$A\leftarrow A\cdot g_{e_i}$。

(4) Return(A)。

实验要求

对任意正整数 $e<2^{60}$ 给出它的 k-ary 和窗口表示。

输入：e, k （十进制输入）

输出：e 的分块和窗口宽度为 k 的译码表示

实验过程

使用 C++在 Kali Linux 上链接 GMP 库完成，源代码如下：

实验结果

```cpp
#include <iostream>
#include <vector>
#include <string>
#include <algorithm>

// 将十进制转换为二进制表示
std::string decToBinary(int n) {
    std::string binaryNum;
    while (n > 0) {
        binaryNum.push_back('0' + (n % 2)); // 将当前位存入字符串
        n /= 2;                             // 取下一位
    }
    // 反转以获得正确的二进制表示
    std::reverse(binaryNum.begin(), binaryNum.end());
    return binaryNum;
}

// 生成 k-ary 和窗口表示
std::vector<std::string> k_ary_and_window_representation(int e, int k) {
    std::string binary = decToBinary(e); // 将 e 转换为二进制字符串

    // 根据窗口大小 k 对二进制字符串进行填充
    int pad = k - (binary.length() % k);
    if (pad != k) { // 若需要填充，则前面补零
        binary = std::string(pad, '0') + binary;
    }

    // 将二进制字符串分成长度为 k 的块
    std::vector<std::string> chunks;
    for (int i = 0; i < binary.length(); i += k) {
        chunks.push_back(binary.substr(i, k)); // 每次截取长度为 k 的子串
    }

    return chunks;
}int main() {
    int e, k;

    std::cout << "请输入一个正整数 e: ";
    std::cin >> e;

    std::cout << "请输入窗口大小 k: ";
    std::cin >> k;

    // 生成 k-ary 和窗口表示
    std::vector<std::string> k_ary_representation = k_ary_and_window_representation(e, k);

    // 输出结果
    std::cout << "整数 " << e << " 的 k-ary 和窗口宽度为 " << k << " 的译码表示为: " << std::endl;
    for (size_t i = 0; i < k_ary_representation.size(); ++i) {
        std::cout << "窗口 " << i + 1 << ": " << k_ary_representation[i] << std::endl;
    }

    return 0;
}
```

实验结果

```
┌──(pama㉿kali)-[~/Desktop/myreport]
└─$ g++ practice_4.cpp -lgmp -o practice_4

┌──(pama㉿kali)-[~/Desktop/myreport]
└─$ ./practice_4
请输入一个正整数 e: 11749
请输入窗口大小 k: 3
整数 11749 的 k-ary 和窗口宽度为 3 的译码表示为:
窗口 1: 010
窗口 2: 110
窗口 3: 111
窗口 4: 100
窗口 5: 101
```

实验 5 设计一个 wNAF 译码算法

实验原理

算法11 计算一个正整数的窗口宽度为w的NAF

输入：窗口宽度w，一个正整数k。

输出：$\mathrm{NAF}_w(k)$。

(1) $i \leftarrow 0$。

(2) while $k \geq 1$ do

(2.1) 如果k是奇数，则$k_i \leftarrow k \bmod\mathrm{s}\, 2^w$，$k \leftarrow k - k_i$。

(2.2) 否则，$k_i \leftarrow 0$。

(2.3) $k \leftarrow k/2, i \leftarrow i+1$。

(3) Return($(k_{i-1}k_{i-2}\cdots k_1k_0)_{wNAF}$)。

解释.

k mods 2^w 表示一个整数u，u满足$u \equiv k(\bmod 2^w)$ 且$-2^{w-1} \leq u \leq 2^{w-1}$即绝对值最小余数。由于第(2.1)步，可以保证连续的w比特数据至多只有一个非零数字。

实验要求

对任意正整数 k<2^{60}给出它的 wNAF 表示。

输入：k, w （十进制输入）

输出：k 的窗口宽度为 w 的 wNAF 表示

实验过程

使用 C++在 Kali Linux 上链接 GMP 库完成，源代码如下：

myreport - practice_5.cpp

```cpp
#include <gmp.h>
#include <vector>
#include <iostream>

// 计算 wNAF 表示
std::vector<int> wNAF(mpz_t k, int w) {
    std::vector<int> wNAF_rep; // 用于存储 wNAF 表示
    mpz_t zero, temp, two_pow_w, two_pow_w_minus_1;
    mpz_inits(zero, temp, two_pow_w, two_pow_w_minus_1, NULL);

    // 计算 2^w 和 2^(w-1)
    mpz_ui_pow_ui(two_pow_w, 2, w);
    mpz_ui_pow_ui(two_pow_w_minus_1, 2, w - 1);

    while (mpz_cmp(k, zero) > 0) { // 当 k > 0
        if (mpz_odd_p(k)) { // 如果 k 是奇数
            // 计算 z_i = k mod 2^w
            mpz_mod(temp, k, two_pow_w);
            if (mpz_cmp(temp, two_pow_w_minus_1) >= 0) { // 如果 z_i >= 2^(w-1)
                mpz_sub(temp, temp, two_pow_w);          // z_i = z_i - 2^w
            }
            wNAF_rep.push_back(mpz_get_si(temp));        // 存储 z_i
            // 更新 k = k - z_i
            mpz_sub(k, k, temp);
        } else {
            wNAF_rep.push_back(0); // 如果 k 是偶数，z_i = 0
        }
        // k 右移 1 位
        mpz_tdiv_q_2exp(k, k, 1);
    }

    mpz_clears(zero, temp, two_pow_w, two_pow_w_minus_1, NULL);
    return wNAF_rep;
}

int main() {
    mpz_t k;
    mpz_init(k);

    int w;
    std::cout << "请输入一个正整数 k： ";
    gmp_scanf("%Zd", k);

    std::cout << "请输入窗口大小 w： ";
    std::cin >> w;

    // 计算 wNAF 表示
    std::vector<int> wNAF_rep = wNAF(k, w);

    // 输出结果
    std::cout << "整数 k 的窗口宽度为 " << w << " 的 wNAF 表示为： " << std::endl;
    for (int i = wNAF_rep.size() - 1; i >= 0; --i) {
        std::cout << wNAF_rep[i] << " ";
    }
    std::cout << std::endl;

    mpz_clear(k);
    return 0;
}
```

实验结果


```
┌──(pama㉿kali)-[~/Desktop/myreport]
└─$ g++ practice_5.cpp -lgmp -o practice_5

┌──(pama㉿kali)-[~/Desktop/myreport]
└─$ ./practice_5
请输入一个正整数 k: 1122334455
请输入窗口大小 w: 6
整数 k 的窗口宽度为 6 的 wNAF 表示为:
1 0 0 0 0 0 0 0 0 23 0 0 0 0 0 11 0 0 0 0 0 0 -9 0 0 0 0 0 0 0 -9
```

实验 6 研究 Shamir 窍门和扩展 Shamir 窍门的效率优势

实验原理

算法 1 Shamir窍门

输入：g，$h \in G$ 和正整数 $a=(a_t a_{t-1} \cdots a_1 a_0)_2$，$b=(b_t b_{t-1} \cdots b_1 b_0)_2$。

输出：$g^a \cdot h^b$。

(1) 计算并存储 $g \cdot h$。

(2) $A \leftarrow 1$。

(3) For i from t down to 0 do the following:

(3.1) 如果 $i \neq t$，则 $A \leftarrow A \cdot A$。

(3.2) 如果 $(a_i,\ b_i) \neq (0,0)$，则 $A \leftarrow A \cdot g^{a_i} \cdot h^{b_i}$。

(4) Return(A)。

算法 2 扩展Shamir窍门

输入：g，$h \in G$ 和正整数 $a=(a_t a_{t-1} \cdots a_1 a_0)_2$，$b=(b_t b_{t-1} \cdots b_1 b_0)_2$。

输出：$g^a \cdot h^b$。

(1) 将 a 和 b 译码为非邻接表表示 $a=(d_{t+1} d_t \cdots d_1 d_0)_{NAF}$，$b=(f_{t+1} f_t \cdots f_1 f_0)_{NAF}$。

(2) 计算和(或)存储 g^{-1}，h^{-1}，$g^{-1} \cdot h$，$g \cdot h^{-1}$，$g^{-1} \cdot h^{-1}$，$g \cdot h$。

(3) $A \leftarrow 1$。

(4) For i from $t+1$ down to 0 do the following:

(4.1) 如果 $i \neq t+1$，则 $A \leftarrow A \cdot A$。

(4.2) 如果 $(d_i,\ f_i) \neq (0,0)$，则 $A \leftarrow A \cdot g^{d_i} \cdot h^{f_i}$。

(5) Return(A)。

实验要求

计算多模幂，并统计用 Shamir 窍门和扩展 Shamir 窍门时需要的乘法次数。

输入：g，h，a，b，p<=2^{32}

输出：$g^a h^b \pmod p$；Shamir 窍门需要的乘法次数 m，扩展 Shamir 窍门需要的乘法次数 n（统计初始计算的乘法）

实验过程

使用 C++在 Kali Linux 上链接 GMP 库完成，源代码如下：

myreport - practice_6.cpp

```cpp
#include <iostream>
#include <gmp.h>

// 使用 Shamir 窍门计算 g^a * h^b (mod p) 并统计乘法次数
void compute_and_count(mpz_t g, mpz_t h, mpz_t a, mpz_t b, mpz_t p, int& m, int& n) {
    mpz_t result, temp;
    mpz_inits(result, temp, NULL);

    // 计算 g^a mod p 使用 Shamir 的窍门
    mpz_powm(result, g, a, p);
    m += mpz_popcount(a);  // 统计 g^a 需要的乘法次数

    // 计算 h^b mod p 使用 Shamir 的窍门
    mpz_powm(temp, h, b, p);
    m += mpz_popcount(b);  // 统计 h^b 需要的乘法次数

    // 计算 g^a * h^b mod p
    mpz_mul(result, result, temp);
    mpz_mod(result, result, p);
    m++;  // 计算乘法

    // 打印 Shamir 的窍门结果
    gmp_printf("使用 Shamir 的窍门计算 g^a * h^b mod p = %Zd\n", result);

    // 使用扩展 Shamir 的窍门计算 g^a * h^b mod p
    mpz_t g_pow_a, h_pow_b;
    mpz_inits(g_pow_a, h_pow_b, NULL);

    // 预计算 g^a 和 h^b
    mpz_powm(g_pow_a, g, a, p);
    mpz_powm(h_pow_b, h, b, p);
    n += mpz_popcount(a) + mpz_popcount(b);  // 统计预计算的乘法次数

    // 计算 g^a * h^b mod p
    mpz_mul(result, g_pow_a, h_pow_b);
    mpz_mod(result, result, p);
    n++;  // 计算乘法

    // 打印扩展 Shamir 的窍门结果
    gmp_printf("使用扩展 Shamir 的窍门计算 g^a * h^b mod p = %Zd\n", result);

    mpz_clears(result, temp, g_pow_a, h_pow_b, NULL);
}

int main() {
    mpz_t g, h, a, b, p;
    mpz_inits(g, h, a, b, p, NULL);

    // 提示用户输入
    std::cout << "请输入底数 g: ";
    gmp_scanf("%Zd", g);
    std::cout << "请输入底数 h: ";
    gmp_scanf("%Zd", h);
    std::cout << "请输入指数 a: ";
    gmp_scanf("%Zd", a);
    std::cout << "请输入指数 b: ";
    gmp_scanf("%Zd", b);
    std::cout << "请输入模数 p: ";
    gmp_scanf("%Zd", p);

    // 检查 p 是否符合要求
    if (mpz_cmp_ui(p, 0) <= 0 || mpz_cmp_ui(p, 1ULL << 32) > 0) {
        std::cerr << "模数 p 必须在 1 到 2^32 范围内! " << std::endl;
        mpz_clears(g, h, a, b, p, NULL);
        return 1;
    }

    int m = 0, n = 0;
    // 调用函数计算并统计乘法次数
    compute_and_count(g, h, a, b, p, m, n);

    // 输出乘法次数
    std::cout << "使用 Shamir 的窍门需要 " << m << " 次乘法\n";
    std::cout << "使用扩展 Shamir 的窍门需要 " << n << " 次乘法\n";

    mpz_clears(g, h, a, b, p, NULL);
    return 0;
}
```

实验结果

```
┌──(pama㉿kali)-[~/Desktop/myreport]
└─$ ./practice_6
请输入底数 g: 2
请输入底数 h: 5
请输入指数 a: 569858951
请输入指数 b: 734233321
请输入模数 p: 3586654197
使用 Shamir 的窍门计算 g^a * h^b mod p = 1472000767
使用扩展 Shamir 的窍门计算 g^a * h^b mod p = 1472000767
使用 Shamir 的窍门需要 34 次乘法
使用扩展 Shamir 的窍门需要 34 次乘法
```