信息安全数学基础探究报告

2310711 陈子烨 信安

1. 素性检测

（1）问题背景：很多公钥密码算法都会用到大素数，生成指定位数大素数的步骤如下：①随机生成一个指定位数的大奇数 p’ 作为候选数。②判断奇数 p’ 是否为素数。可以看出，生成指定位数大素数的核心步骤是判断是否为素数。判断的过程就是**素性检测**。

（2）数理部分：

<1>相关概念：素性检测算法可分为两类：确定性素性检测算法、概率素性检测算法。确定性素性检测算法可以明确判断出大奇数 p’ 是合数还是素数，不存在出错的可能，但计算复杂度较高。概率素性探测算法中，如果判断 p'是合数，则 p'一定是合数，但如果判定 p'是素数，那么 p'在一定概率下是素数，也有可能是合数。虽然存在判断错误的可能，但概率素性探测算法的复杂度远远小于确定性素性探测算法。为了提高精度，可以对 p'进行多次测试，将出错概率控制在足够小的范围内。

<2>理论知识：

引理：令 n 为素数，那么对任意 $b\in Z_n^*$，有 $b^{(n-1)/2}\equiv(\frac{b}{n})(\mathrm{mod}\ n)$，其中$(\frac{b}{n})$是雅可比符号.

定义 1：令 n 为奇合数，并且(b,n) = 1，若有 $b^{(n-1)/2}\equiv(\frac{b}{n})(\mathrm{mod}\ n)$，则称 n 为关于基 b 的**欧拉伪素数**.

定理 1：令 n 为奇合数，那么在 Z_n^*中至少存在一半的数使得 $b^{(n-1)/2}\equiv(\frac{b}{n})(\mathrm{mod}\ n)$不成立.

定义 2：令 n 为奇合数，$n-1=2^s t$，2∤t，s≥1. 令 $b\in Z_n^*$，(b,n)=1. 若对于某

个 r(0 ≤r < s)，有 b^t≡1(mod n)或 b^2^rt≡-1(mod n)，则称 n 为关于基 b 的**强伪素数**.

定理 2：令 n 为奇合数，b∈Z_n^*，那么 n 是关于基 b 的强伪素数的概率不超过$\frac{1}{4}$.

（3）相关算法：

<1>**Solovay-Strassen 算法**

此算法利用雅可比符号进行素性检测。基本思想是：分别计算雅可比符号$(\frac{a}{p})$和 $a^{(p-1)/2}$(mod p)，将所得的值分别赋给 x 和 y，若 x≡y(mod p)，则输出素数，否则输出合数。此算法利用了欧拉判别条件：p 是奇素数，(a,p)=1，则 a 是模 p 的二次剩余的充要条件是 $a^{(p-1)/2}$≡1(mod p)，a 是模 p 的二次非剩余的充要条件是 $a^{(p-1)/2}$≡-1(mod p)。合数的结论必然正确，但素数的结论则不一定。c++语言实现的代码如下：

```
#include <iostream>
using namespace std;
int gcd(int a, int b) {
    return b == 0 ? a : gcd(b, a % b);
} //求最大公约数
int jacobi(int a, int n) {
    int result = 1;
    if (a < 0) { a = -a; if (n % 4 == 3) result = -result; }
    while (a != 0) {
        while (a % 2 == 0) {
            a /= 2; if (n % 8 == 3 || n % 8 == 5) result = -result;
        }
        swap(a, n);
        if (a % 4 == 3 && n % 4 == 3) result = -result;
        a %= n;
    }
    return n == 1 ? result : 0;
} //计算雅可比符号
int modularExponentiation(int base, int exp, int mod) {
    int result = 1;
    while (exp > 0) {
        if (exp % 2 == 1) result = (result * base) % mod;
```

```
            base = (base * base) % mod;
            exp /= 2;
        }
        return result;
    }  //模幂运算
    bool solovayStrassen(int n, int k) {
        if (n < 2 || (n != 2 && n % 2 == 0)) return false;
        for (int i = 0; i < k; ++i) {
            int a = rand() % (n - 1) + 1;
            int jacobiSymbol = jacobi(a, n);
            int x = modularExponentiation(a, (n - 1) / 2, n);
            if (x == 0 || (jacobiSymbol != 0 && x != 1 && x != n -
1)) return false;
        }
        return true;
    }  //Solovay-Strassen 素性检测
    int main() {
        int n, k;
        cout << "请输入想要进行 S-S 素性检测的大奇数：";
        cin >> n;
        cout << "请输入检测次数: ";
        cin >> k;
        cout << n << (solovayStrassen(n, k) ? "可能是素数" : "是
合数");
        return 0;
    }  //主函数。当最终输出“可能是素数”时，n 是合数的概率不超过 1/2^k
```

该算法具有以下特点：①基于雅可比符号。②利用欧拉判别条件。③复杂度相对较低，易于实现。④因为是概率素性检测，所以存在判断错误的情况。

<2> **Miller-Rabin 算法**

根据费马小定理 ,若 p 是素数 ,则对于一个小于 p 的整数有 $a^{p-1}(\text{mod } p)=1$，因此，给定整数 p，若能在区间内找到一个整数 a 使得 $a^{p-1}(\text{mod } p)=1$，则可说明 p 是一个合数。上述过程中需要计算 $a^{p-1}(\text{mod } p)$，为了简化计算，不妨设 $p-1=2t$（p 是奇数，则 p-1 是偶数），于是 $a^{p-1}=(a^t)^2$，接下来考察 $a^t(\text{mod } p)$，若结果为±1 ,则直接输出素数 ,然后结束(因为平方后就会得到 $a^{p-1}(\text{mod } p)=1$)。

以此类推，可以将 p 表示为 $p-1=2^kt$，所以 $a^{p-1}=(a^t)^{2^k}$。于是就可以依次考察 a^t，$(a^t)^2$，$(a^t)^{2^2}$，…，$(a^t)^{2^{k-1}}$，若序列中出现了一个-1，就输出素数并停止。

c++语言实现的代码如下：

```
    #include <iostream>
    using namespace std;
    long long mod_exp(long long base, long long exp, long long mod)
{
        long long result = 1;
        base %= mod;
        while (exp > 0) {
            if (exp & 1) result = result * base % mod;
            base = base * base % mod;
            exp >>= 1;
        }
        return result;
    }  // 快速模幂
    bool MillerRabin(long long n, int k) {
        if (n < 2) return false;
        if (n <= 3) return true;
        if (n % 2 == 0) return false;

        long long d = n - 1;
        int s = 0;
        while (d % 2 == 0) {
            d /= 2;
            s++;
        }
        for (int i = 0; i < k; i++) {
            long long a = 2 + rand() % (n - 4);
            long long x = mod_exp(a, d, n);
            if (x == 1 || x == n - 1) continue;

            bool composite = true;
            for (int r = 1; r < s; r++) {
                x = mod_exp(x, 2, n);
                if (x == n - 1) {
                    composite = false;
                    break;
                }
            }
            if (composite) return false;
        }
        return true;
    }  // Miller-Rabin 测试
```

```
int main() {
    long long n;
    int k;
    cout << "请输入想要进行 M-R 素性检测的大奇数： ";
    cin >> n;
    cout << "请输入检测次数: ";
    cin >> k;
    cout << n << (MillerRabin(n, k) ? "是素数" : "不是素数") ;
    return 0;
}  //主函数
```

该算法具有以下特点：①时间复杂度为 $O(k \cdot \log_3 n)$，较为高效。②原理依据了费马小定理。③能够对小素数进行特殊情况处理。④因为是概率素性检测，所以存在判断错误的情况，但概率极低。

（4） 密码学中的应用：

素性检测在密码学中主要应用于：①**随机数生成**：许多加密算法要求使用大素数生成密钥（如 RSA 加密），确保生成的随机数是素数对密钥的安全性至关重要。②**数字签名**：在实现数字签名的算法（如 DSA）时，需要使用素数来生成密钥和验证签名。③**公钥加密**：如 RSA、ElGamal 等加密算法中依赖于大素数的性质，使用素性检测确保密钥安全性。

Solovay-Strassen 算法在密码学中主要应用于①**大素数生成**：Solovay-Strassen 算法是一种有效的素性测试算法，通常用于生成大素数，这些素数在密钥生成过程中至关重要，如生成 RSA 和 DSA 的密钥对。②**增量式安全性**：由于 Solovay-Strassen 是一个随机化的算法，适用于实现需要一定概率保证的安全性场景，比如在高安全性要求的加密系统中，如数字货币的私钥生成等场景。

Miller-Rabin 算法在密码学中主要应用于①**公钥加密**：Miller-Rabin 算法

常用于 RSA、ElGamal 和其他基于数论的公钥加密系统中，以检验大整数的素性，从而确保生成的密钥是由素数组成，提升安全性。②**密码协议**：在密码学协议（如 Diffie-Hellman 密钥交换）中，需要大素数以保证协议的安全性，Miller-Rabin 提供了一个高效且可靠的方法来验证这些素数。③**后续安全性**：Miller-Rabin 能够在有条件地判定一个数为“强素数”，这对于某些密码算法（如 certain DSA 的实现）非常重要，能抵御特定类型的攻击。

2. 二次剩余问题

（1）知识回顾：

设 m 是大于 1 的整数，(a,m)=1，若 $x^2 \equiv a \pmod m$ 有解，则 a 是模 m 的二次剩余，否则，a 是模 m 的二次非剩余。判断二次剩余可借助：

1. **欧拉判别条件**：p 是奇素数，(a,p)=1，则 a 是模 p 的二次剩余的充要条件是 $a^{(p-1)/2} \equiv 1 \pmod p$，a 是模 p 的二次非剩余的充要条件是 $a^{(p-1)/2} \equiv -1 \pmod p$。

2. **勒让德符号**：设 p 是奇素数，(a,p)=1，勒让德符号为 $\left(\frac{a}{p}\right)=1$ (a 是模 p 的二次剩余)；-1(a 是模 p 的二次非剩余)。

3. **高斯引理**：设 p 是奇素数，(a,p)=1，如果下列 $\frac{p-1}{2}$ 个整数：a·1，a·2， a·3，……，$a\cdot\frac{p-1}{2}$ 模 p 后得到的最小正剩余中大于 $\frac{p}{2}$ 的个数是 m，则 $\left(\frac{a}{p}\right)=(-1)^m$。

4. **二次互反律**：$\left(\frac{p}{q}\right)\left(\frac{q}{p}\right) = (-1)^{\frac{p-1}{2}\cdot\frac{q-1}{2}}$，

或写为 $\left(\frac{q}{p}\right) = (-1)^{\frac{p-1}{2}\cdot\frac{q-1}{2}}\left(\frac{p}{q}\right)$。

5. **雅可比符号**：设正奇数 $m=p_1p_2\ldots p_r$ 是奇素数 p_i（i=1,2,…,r）的乘积，定义雅可比符号：$\left(\frac{a}{m}\right)=\left(\frac{a}{p1}\right)\left(\frac{a}{p2}\right)\ldots\left(\frac{a}{pr}\right)$。

（2）数理部分：

定义 1：若整数 n 满足 n=pq，其中 p 和 q 是素数且 $p\equiv 3 \pmod 4$，$q\equiv 3 \pmod 4$，

则称 n 为 **Blum 数**。

定理 1：如果 n 是一个 Blum 数，那么-1 是模 n 的二次非剩余，且$\left(\frac{-1}{n}\right)=1$。

定义 2：(二次剩余问题）令 n=pq 是两个大素数之积。对于一个任意选取的模 n 的二次剩余 a，称求出一个 x 使其满足 $x^2 \equiv a \pmod n$的问题为**二次剩余问题**。

定理 2：对于任意模 n 的二次剩余 a，方程 $x^2 \equiv a \pmod n$有 4 个模 n 下的解。这四个根为：$x \equiv \pm x_0 \pmod p$，$x \equiv \pm x_1 \pmod q$。

定理 3：对于每个模 n 的二次剩余 a，方程 $x^2 \equiv a \pmod n$的 4 个解中只有一个解是模 n 的二次剩余。

定理 4：对 n 进行分解等价于解决模 n 的二次剩余问题。

定义 3：(**二次剩余的判定**）令 n=pq 是两个大素数之积，对于一个随机选取的元素 $a \in Z_n^*$，判定 a 是否为模 n 的二次剩余是一个困难问题。

（3）相关算法：

<1>**定理 4 算法**

```
#include <iostream>
#include <cmath>
using namespace std;
//计算 a 和 b 的最大公因子
int gcd(int a, int b) {
    while (b != 0) {
        int t = b;
        b = a % b;
        a = t;
    }
    return a;
}
// 计算 x 在模 n 下的平方根
int modSqrt(int x, int n, int roots[]) {
    int count = 0;
    for (int i = 0; i < n; i++) {
        if ((i * i) % n == x) {
            roots[count++] = i;
```

```
            if (count == 4) break;
        }
    }
    return count;
}

// 算法 A：尝试找到 n 的因子
int factorizationAlgorithm(int n) {
    for (int i = 2; i <= sqrt(n); i++) {
        if (n % i == 0) {
            return i;
        }
    }
    return n;
}

// 算法 B：输出 x 在模 n 下的平方根
void squareRootAlgorithm(int n, int x) {
    int p = factorizationAlgorithm(n);
    int q = n / p;
    int roots[4];
    int count = modSqrt(x, n, roots);
    if (count == 0) {
        cout << "没有平方根\n";

        return; // 没有平方根
    }
    int b = roots[0];
    int a = 1;
    while ((a * a) % n != (n - 1)) {
        a++;
        if (a >= n) {
            cout << "没有找到满足条件的 a\n";
            return;
        }
    }
    if ((b * b) % n == 1) {
        int factor1 = gcd(n, a - b);
        int factor2 = gcd(n, a + b);
        cout << "找到因子: " << factor1 << " 或 " << factor2 <<
"\n";
    } else {
        cout << "没有找到因子\n";
    }
```

```
}

int main() {
    int n = 15;
    int x = 9; //x 是 n 的二次剩余

    cout << "对 n 的因子分解:\n";
    factorizationAlgorithm(n);
    cout << "计算 x 在模 n 下的平方根:\n";
    squareRootAlgorithm(n, x);
    return 0;
}
```

<2>**Rabin 算法**

Rabin 算法的主要思想是基于模二次剩余的数学特性 ,利用快速因子分解方法来进行大数的分解。用 c++语言实现如下：

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

// 计算 x^y (mod p)
int modExp(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; // base = base^2 mod p
    }
    return result;
}

// 计算 a 在模 n 下的平方根
int modSqrt(int a, int n) {
    int roots[2];
    int count = 0;
    for (int i = 0; i < n; i++) {
        if ((i * i) % n == a) {
            roots[count++] = i;
            if (count == 2) break;
        }
    }
    return count > 0 ? roots[0] : -1;
}
```

```
// Rabin 算法
void rabin(int n) {
    int p = 1;
    while (p != -1) {
        p = modSqrt(1, n);
        if (p != -1) {
            cout << "找到一个因子: " << gcd(n, n - p) << endl;
            return;
        }
    }
    cout << "没有找到因子\n";
}
int main() {
    int n = 21; // 示例输入
    cout << "对 n 的因子进行分解:\n";
    rabin(n);
    return 0;
}
```

该算法具有以下特点：①随机性：Rabin 算法使用随机选择的技术，增加了成功找到因子的概率，通过多次尝试可以提高成功率。②效率较高：该算法在大素数的因子分解方面具有良好的表现，适合处理大数。③简单性：Rabin 算法的实现相对直观，使用基础的数学概念，如模幂和平方根。④复杂度：一般情况下，该算法的时间复杂度为 O(√n)，但由于其随机性质，统计上的性能往往更优。

<3>**Goldwasser-Micali 算法（GM 加密）**

Goldwasser-Micali 算法是一种基于数论的公钥加密算法，其安全性基于二次剩余问题。该算法对加密消息进行编码，确保只有合法的私钥持有者能够解密。算法的具体过程如下：1.公钥和私钥生成：选择两个大素数 p 和 q，计算 $n=p\cdot q$，生成一个随机符号 s。2.加密：消息通过随机数编码为 $c=m\cdot r^2 \pmod n$，其中 r 是随机的 {0,1}的平方根，使用 s 来确定附加的加密属性。3.解密：使用私钥通过二次剩余的性质解密消息，检查 c 是不是二次剩余，利

用 s 提供的信息恢复原始消息。用 c++语言实现如下：

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

// 计算 a 在模 n 下的平方

int modSquare(int a, int n) {
    return (a * a) % n;
}

// 检查 b 在模 n 下是否是二次剩余

bool isQuadraticResidue(int b, int n) {
    for (int i = 0; i < n; i++) {
        if (modSquare(i, n) == b) {
            return true;
        }
    }
    return false;
}

// Goldwasser-Micali 加密算法

void encrypt(int m, int n, int& c) {
    int r = 2;
    c = (m * modSquare(r, n)) % n;
}

// Goldwasser-Micali 解密算法

int decrypt(int c, int s, int n) {
    if (isQuadraticResidue(c, n)) {

        return 1; // 如果 c 是二次剩余，返回 1（代表明文 1）

    } else {

        return 0; // 如果 c 不是二次剩余，返回 0（代表明文 0）

    }
}
int main() {
    int p = 3, q = 11;
    int n = p * q;
    int s = p % 4;
    int message = 1;
    int ciphertext;
    encrypt(message, n, ciphertext);

    cout << "加密后的密文: " << ciphertext << endl;

    int decryptedMessage = decrypt(ciphertext, s, n);

    cout << "解密后的明文: " << decryptedMessage << endl;
```

```
    return 0;
}
```

该算法具有以下特点：①安全性：Goldwasser-Micali 算法的安全性基础在于二次剩余问题，至今尚无有效的多项式时间算法可以解决。②随机性：算法使用随机数进行加密，这提供了更高的安全性，使得同一明文在多次加密时的密文不同。③加密信息量：算法允许在加密过程中将零和一编码为不同的密文，并通过解密判断出其原始信息。④计算复杂度：对于明文位的处理相对较快，但由于需要确认数是否为二次剩余，因此性能上可能受到一定限制。

（4）在密码学中的应用：

二次剩余问题在密码学中主要应用于：①**公钥加密**：二次剩余问题的不可解性是多种密码系统（如 Rabin 和 GM 加密）的安全基础。由于没有有效的算法能够在多项式时间内决定一个数是否为二次剩余，因此可以用来构建安全的加密协议。②**密钥交换**：二次剩余问题可以在某些密钥交换协议中用于生成共享密钥，尤其是在基于数学难题的协议中。③**零知识证明**：在某些零知识证明协议中，二次剩余被用作证明者向验证者展示他知道某个秘密，但仍能保持该秘密的隐私。

Rabin 算法在密码学中主要应用于①**加密**：Rabin 算法可以用于加密小型消息，特别是适用于处理二进制消息 (0 和 1)。②**数字签名**：Rabin 签名方案基于 Rabin 加密算法，可用于创建数字签名，确保消息的完整性和来源。③**整数因子分解问题**：Rabin 算法的实现部分依赖于整数因子分解的困难性，在某些密码系统中被用作挑战，提升了安全性。

Goldwasser-Micail 算法在密码学中主要应用于①**加密**：GM 算法用于加密数据，特别是在二进制数据的场景中。②**同态加密**：GM 算法具有一定的同态性，允许某些类型的运算在加密数据上进行，而不需要先解密。这对于加密云计算和隐私保护计算尤为重要。③隐私保护：GM 加密可以用于保护敏感信息，

同时允许相关的计算和查询操作。

3. 同态加密算法

（1）定义：

同态加密（Homomorphic Encryption）是一种特殊的加密算法，它允许在密文上直接进行计算，并且得到的结果在解密后与在原始明文上进行相同计算的结果一致。这种加密方式的核心特性是“同态性”，即加密和解密操作可以与某些数学运算相结合，而不会破坏数据的完整性和隐私性。

（2）密码原语：

<1>分类：

根据加密函数的运算类型将同态加密算法分为**部分同态加密**和**全同态加密**，部分同态加密又可分为**加法同态加密**和**乘法同态加密**。下面依次介绍：

加法同态加密：允许对加密后的数据进行加法操作。假设有两个明文消息 m1 和 m2，它们对应的加密结果为 E(m1)和 E(m2)，若满足 E(m1+m2)=E(m1)+E(m2)，则加法同态成立。同时存在解密函数 D 使得 D(E(m))=m，即对密文解密后可以得到明文。

例：明文消息 m1=3, m2=5，加密后得到 E(m1)=E(3)和 E(m2)=E(5)。计算 C=E(m1)+E(m2)。解密密文 C 得到 D(C)=D(E(m1)+E(m2))=D(E(3)+E(5))=8，所以结果为 3+5=8。

乘法同态加密：允许对加密后的数据进行乘法操作。假设有两个明文消息 m1 和 m2，它们对应的加密结果为 E(m1)和 E(m2)，若满足 E(m1×m2)=E(m1)×E(m2)，则乘法同态成立。同时存在解密函数 D 使得 D(E(m))=m，即对密文解密后可以得到明文。

例：明文消息 m1=4, m2=6，加密后得到 E(m1)=E(4)和 E(m2)=E(6)。计

算 C=E(m1)×E(m2)。解密密文 C 得到 D(C)=D(E(m1)×E(m2))=D(E(4)×E(6))=24，所以结果为 4×6=24。

全同态加密(FHE)：允许对加密后的数据进行任意计算（不仅包括加法和乘法，还支持两者的任意组合、任意次数以及任意函数的操作）。设 E 为加密函数，D 为解密函数，对于任意明文 m1 和 m2，如果有 f 为任意计算函数，则满足 D(E(f(m1, m2)))=f(m1, m2)。

<2>算法：

1.**Paillier 算法**（加法同态加密）：基于大数分解的困难性，依赖于全域模乘的加法同态特性，支持任意大小的明文。算法步骤如下：

①密钥生成 选择两个大素数 p 和 q ,计算 n=p×q ,计算λ=lcm(p-1, q-1)(最小公倍数)，随机选择一个整数 $g \in Z_{n\wedge 2}^{*}$ 满足(gcd(L($g^{\lambda} \bmod n^2$), n)=1, 其中 L(x)= $\frac{x-1}{n}$，计算 n^2 并确定公钥 pk=(n, g)，私钥 sk=λ。

②加密算法：给定一个明文消息 m，随机选择一个 $r \in Z_n^{*}$，计算密文 c 为 $c=g^m \times r^n \pmod{n^2}$。

③解密算法：明文 m= $\frac{L(c\wedge \lambda \bmod n\wedge 2)}{L(g\wedge \lambda \bmod n\wedge 2)}$ 。

C++语言实现 Paillier 算法如下：

```
#include <iostream>
Using namespace std;
class Paillier {
private:
    int p, q, n, n_squared, lambda;
    int L(int x) {
        return (x - 1) / n;
    }
    int lcm(int a, int b) {
        int gcd = 1;
        for (int i = 1; i <= a && i <= b; ++i) {
            if (a % i == 0 && b % i == 0) {
                gcd = i;
            }
        }
```

```
        return (a * b) / gcd;
    }

public:
    int g, publicKey, privateKey;

    // 构造函数 - 生成密钥对
    Paillier() {
        p = 11;
        q = 13; //实际中选大素数
        n = p * q;
        n_squared = n * n;
        lambda = lcm(p - 1, q - 1);
        g = n + 1;
        publicKey = n;
        privateKey = lambda;
    }

    // 加密函数
    int encrypt(int m) {
        int r = 7; //实际中应随机选择
        int c = (pow(g, m) * pow(r, n)) % n_squared;
        return c;
    }

    // 解密函数
    int decrypt(int c) {
        int u = pow(c, privateKey) % n_squared;
        return L(u);
    }
};
int main() {
    Paillier paillier;
    int m = 42;
    int c = paillier.encrypt(m);
    cout << "Encrypted message: " << c << endl;
    int decrypted_m = paillier.decrypt(c);
    cout << "Decrypted message: " << decrypted_m << endl;
    return 0;
}
```

2.**RSA 算法**（乘法同态加密）：一种非对称加密算法，基于数论中的素数分解难题，广泛用于安全数据传输。算法步骤如下：

①密钥生成 选择两个大素数 p 和 q ,计算 n=p×q ,计算φ(n)=(p-1)(q-1)(欧

拉函数)，选择一个整数 e (公钥指数)，满足 $1<e<\varphi(n)$且 $\gcd(e, \varphi(n))=1$，计算 d，使得 $d\equiv e^{-1} \bmod \varphi(n)$ (私钥指数)，即 d 是 e 模φ(n)的乘法逆元。公钥为(e, n)，私钥为(d, n)。

②加密算法：明文消息 m 被加密为密文 c：$c\equiv m^e \bmod n$。

③解密算法：密文 c 被解密为明文 $m\equiv c^d \bmod n$。

C++语言实现 RSA 算法如下：

```
#include <iostream>
using namespace std;
int gcd(int a, int b) {
    while (b != 0) {
        int t = b;
        b = a % b;
        a = t;
    }
    return a;
}
// 计算 a 的模逆，使用扩展欧几里得算法
int modInverse(int a, int m) {
    int m0 = m, t, q;
    int 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;
}
// RSA 加密和解密
class RSA {
private:
    int p, q, n, phi, e, d;
public:
    RSA(int prime1, int prime2) {
        p = prime1;
```

```
            q = prime2;
            n = p * q;
            phi = (p - 1) * (q - 1);
            e = 3;
            while (gcd(e, phi) != 1) {
                e += 2;
            }
            d = modInverse(e, phi);
        }

        // 加密函数

        int encrypt(int m) {
            return (pow(m, e)) % n;
        }

        // 解密函数

        int decrypt(int c) {
            return (pow(c, d)) % n;
        }
        void displayKeys() {
            cout << "Public Key: (n = " << n << ", e = " << e << ")"
<< endl;
            cout << "Private Key: (n = " << n << ", d = " << d <<
")" << endl;
        }
    };
    int main() {
        int p = 61;

        int q = 53; //实际中选用更大的素数

        RSA rsa(p, q);
        rsa.displayKeys();
        int message = 42;
        cout << "Original Message: " << message << endl;
        int encrypted = rsa.encrypt(message);
        cout << "Encrypted Message: " << encrypted << endl;
        int decrypted = rsa.decrypt(encrypted);
        cout << "Decrypted Message: " << decrypted << endl;
        return 0;
    }
```

3.**全同态加密算法：**LWE，BCV，GSW 等。

（3）应用场景：

1.**数据隐私保护**：在涉及敏感数据的场景中，同态加密可以确保即使数据被外部服务提供商处理，也不会泄露用户的隐私。如医院可以对患者的医疗记录进行加

密，并将加密数据发送给研究机构进行分析，以保护患者隐私；银行可以对客户的财务数据进行加密，并在不解密的情况下进行信用评估或欺诈检测。

2.**云计算**：同态加密使得用户可以在云端进行计算，而不必担心数据的安全性，具体包括：安全数据存储：用户将数据存储在云端时，可以保持数据加密状态，数据提供商只能处理加密数据，而无法访问明文数据。加密计算：用户可以将加密的数据上传到云服务，并要求服务提供商进行计算（如加法、乘法等），用户只需下载最终的加密结果并解密。

3.**机器学习**：同态加密在机器学习中可以用于：私有模型训练：通过在加密数据上进行训练，保护数据隐私。例如，可以在多个医院之间共享患者数据进行联合模型训练，而不泄露具体的患者信息。安全推理：模型可以在加密数据上进行推理，以确保输入的数据（如个人信息）在计算过程中不会被暴露。

4.**匿名货币和支付**：同态加密技术在加密货币和网络支付系统中也有应用：安全交易：保护交易双方的身份信息及交易细节，确保只有授权方能查看交易内容，同时允许进行必需的数学计算（如余额计算）。合规审计：在不泄露用户数据的情况下，可以对交易进行合规性审计。

5.**数据共享于合作计算**：在多方数据分析中，同态加密可以让不同组织在不泄露彼此数据的情况下进行合作：联合分析：多家机构可以在保持各自数据安全的前提下，利用同态加密技术进行联合数据分析，共同发现规律或趋势。隐私保护的市场研究：通过安全聚合用户数据，进行市场研究而不侵犯用户隐私。

（4）数学问题：

1.**整数因式分解**：整数因式分解是确定一个大整数的质因数的过程。这个问题在许多公钥加密方案中（如 RSA）被用作安全的基础。同态加密算法可能使用与整数因式分解相关的数学构造来保证密钥的安全性，因而破解该算法的难度与因式分解的难度密切相关。

2.**离散对数问题**：给定一个素数 p 和一个生成元 g，离散对数问题是找到一个整数 x，使得 $g^x \equiv y \pmod{p}$。许多加密方案的安全性依赖于解决离散对数问题的困难，这是同态加密算法设计的重要基础之一。

3.**格理论**：格理论研究的是在高维空间中定义的点的集合（称为格）。与其他数论问题相比，很多格问题在高维情况下更为复杂和困难，如近似最近向量问题（Approximate nearest vector problem, ANVP）和最短向量问题（Shortest Vector Problem, SVP）。许多基于格的同态加密方案（如 Gentry 的同态加密方案）利用了格问题的计算困难性来构造安全密钥和加密机制。

4.**多项式环**：在同态加密中，多项式环的结构是加密和解密过程中常用的数学工具。密文和运算通常使用多项式表示，这些多项式的计算可以在加密域中直接进行。对多项式的操作（例如加法和乘法）通常依赖于大数分解的复杂性，这使得加密和解密在理论上是可行的。