《信息安全数学基础》探究报告

姓名：陆皓喆　学号：2211044　专业：信息安全

一、大整数分解问题

1.1 数学问题

大整数分解问题属于NP问题，也是目前学界一直没有完全解决的一个难题。对于给定的一个大整数N，它是两个大素数的乘积，但其因子p和q未知，我们将寻找p和q，使其满足$N = p \cdot q$。它是一个在计算复杂性理论和密码学中广泛研究的问题。

假设我们有一个大整数N，大整数分解问题就是要找到N的所有质因数的过程。质因数是指不能被其他整数整除的质数，而大整数可以被分解为质因数的乘积。例如，对于整数$N = 6$，其质因数分解为2×3。

我们都知道，已知两个很大的素数去将其相乘，是一件十分简单的事；但是当事情反一下，已知一个大数是两个大素数的乘积，这样的话我们就不是很好求解这两个素数了。

1.2 算法思想及特点

我们对于大整数分解，目前有许多的解决方法，下面我会将这些常见的算法一一列举。

1.2.1 暴力穷举算法

算法介绍

暴力穷举算法是一种最简单、最容易理解的分解大整数的算法，简单来说，就是通过暴力举例p的值，去进行试除，如果除下来是一个整数，那么说明我们的大整数N可以被分解为p和N/p，然后我们一直进行这样的步骤，直到剩下的数是素数，不能被一个因子除下为止。举一个例子，我们需要分解一个整数100，那么我们需要从2开始一个一个尝试除法，我们先从2开始，发现可以除下来，所以我们就将其写入因数的列表中，同时将100变成50，继续重复上面的步骤，直到这个数也变成素数就停止，这样我们就可以获得$100 = 2 \times 2 \times 5 \times 5$这样的结果。

算法步骤

1. 我们首先需要给出一个大整数N。
2. 然后，我们从最小的因子2开始，一个一个尝试，看该因子是否能被N整除。如果可以整除的话，我们相当于找到了一个质因数。
3. 如果找到一个质因数，将其记录下来，并将N除以该质因数得到一个新的整数$N^{'}$。
4. 重复步骤2和步骤3，直到N被分解为质因数的乘积为止。如果N无法再被分解，说明它本身就是一个质数，即N为最后一个质因数。
5. 最后得到的所有质因数就是N的分解结果。

算法伪代码

我们在此处给出该算法的伪代码，很显然，该代码的输入是一个整数N，然后输出是一个列表，分别记录了该整数的所有因子。

```
Input: 大整数 N

Procedure BruteForceFactorization(N):
    Initialize an empty list of factors

    // 从2开始尝试所有可能的因子
    for i = 2 to N-1:
        // 如果 i 是 N 的因子
        if N % i == 0:
            Add i to the list of factors
            Set N = N / i

            // 重复尝试同一个因子，直到它不再是 N 的因子
            while N % i == 0:
                Add i to the list of factors
                Set N = N / i

            // 如果 N 变为1，表示所有因子已找到
            if N == 1:
                Exit the loop

    // 如果 N 不为1，说明 N 是一个素数
    if N > 1:
        Add N to the list of factors

    Return the list of factors

Output:一串质因数，是由大整数分解而来的
```

我们可以看到，我们$input$一个大整数，然后就会输出一个该整数分解后生成的$list$。

算法评价

我们不难发现，这个算法的时间复杂度非常的高，达到了$O(\sqrt{N})$，其中的N是我们需要分解的整数。如果N比较小的话，我们还是能够在一定的时间内将其进行分解，但是如果当N很大的时候，我们就不能够对其进行分解了。所以在大整数的分解问题中，我们往往不采用这种暴力的算法。

1.2.2 费马分解算法

算法介绍

费马分解算法是一种相比于前面的暴力求解算法稍微改进一些的算法。该算法是一种基于费马小定理的分解整数的算法。该算法是在试图将一个大整数N分解为两个较小整数之间的乘积时使用的。费马分解算法的基本思想是利用费马小定理来搜索满足条件的平方数差，将一个奇数表示为两个平方数之差的形式。

同样的我们举一个例子，比如说我们要分解$n = 119143$， 因为$345^2 < 119143 < 346^2$，所以$k = 346$，$346^2 - 119143 = 573$，$347^2 - 119143 = 1266$，$348^2 - 119143 = 1961$，$349^2 - 119143 = 2658$，$350^2 - 119143 = 3357$，$351^2 - 119143 = 4058$，$352^2 - 119143 = 4761 = 69^2$，所以，$n = (352 - 69) \cdot (352 + 69) = 283 \times 421$。我们通过猜测其中的一个质数，然后通过计算剩下的数能否开方，如果可以开方，那么说明我们的分解成功了。

算法步骤

1. 首先是将大整数开方，取整数部分，得出其所在的范围
2. 然后从整数部分加一开始，用整数部分取平方，再减去原数
3. 计算能否开平方，如果可以就说明分解成功
4. 如果不能开平方的话就说明分解失败，我们继续对整数做加一操作，直到分解成功为止

算法伪代码

我们给出以下的伪代码，通过输入一个大整数，去进行分解，输出一串质因数的乘积。

```
Input: 大整数 N

Procedure FermatFactorization(N):
    // 如果 N 是偶数，直接找到因子 2
    if N % 2 == 0:
        Return [2, N/2]

    // 初始化 a 为大整数的平方根向上取整
    a = ceil(sqrt(N))
    b_square = a*a - N

    // 检查 b_square 是否是一个完全平方数
    while b_square is not a perfect square:
        a = a + 1
        b_square = a*a - N

    // 计算 N 的因子
    b = sqrt(b_square)
    factor_1 = a + b
    factor_2 = a - b

    Return [factor_1, factor_2]

Output: [factor1,factor2,...factorN]
```

我们可以发现，我们输入一个大整数，就可以输出一串被分解的数字。

算法评价

费马分解算法的性能取决于a的选取，我们通常需要尝试不同的a的值才能将N分解，所以对于不同的N来说的话，时间复杂度都是不同的，这取决于我们所选取的a的值。但是，费马分解算法在两个质数相差过大时可能会导致分解失败，所以在一般的大数分解中，除非告诉我们两个素数相差非常小，否则使用费马分解算法很有可能分解不出来，我们可能需要采取其他的算法来进行分解。

1.2.3 $Pollard\ \rho-1$算法

算法介绍

1974年，Pollard基于费马小定理，提出了该方法。该方法不具有一般的分解算法能力，但是也为后期的许多因数分解方法打下了基础，比如说1975年的$Pollard\ \rho$算法，后面的基于其提出的椭圆曲线分解算法等等。

算法步骤

1. 首先给定一个大整数N
2. 我们选择一个a，其满足$1 < a < \rho - 1$
3. 计算$a^{k!} \equiv m(mod\ n)$，因为存在整数j使得$k! = j(\rho - 1)$
4. 得到$m \equiv a^{k!} \equiv a^{j(p-1)} \equiv (a^{\rho-1})^j \equiv 1^j \equiv 1(mod\ \rho)$
5. 也就是$\rho|(m-1)$，因此$(m-1,n) > 1$，我们只需要m与1不关于n同余就可以了
6. 这样我们就求出了我们的非平凡因子$(m-1,n)$

算法伪代码

本处我们提供C++脚本。

```
#include <NTL/ZZ.h>
using namespace NTL;
using namespace std;
#include <iostream>
#include <cstdlib>
#include <math.h>
#include <time.h>
#include <stdlib.h>

void pollard(ZZ N) {
    cout << "N可以分解为：  ";
    while (1) {
        ZZ B = ZZ(10);
        ZZ a = ZZ(2);
        a = PowerMod(a, B, N);
        ZZ d = GCD(a - 1, N);
        if (d > 1 && d < N) {
            cout << d << " ";
            N = N / d;
        }
        else {
            cout << N;
            return;
        }
    }
}

int main() {
    ZZ N;
    cout << "请输入待分解的数：";
    cin >> N;
    pollard(N);
}
```

算法分析

我们一般来说，在此处选取的a的值都是2，为了使其保证与p互素。时间复杂度一般，只要满足我们的b必须要大于$p-1$的所有因子，不然就会导致分解失败。所以，在一般的加密大整数中，我们一般都会选取$p=2p_1+1, q=2q_1+1$这样的素数来抵御我们的$Pollard\ \rho-1$攻击。

1.2.4 $Pollard\ \rho$算法

算法介绍

$Pollard\ \rho$算法是一种随机化算法，用于分解大整数。该算法由$John\ Pollard$于1975年提出，基于整数序列的随机性质来寻找整数的因子。$Pollard\ \rho$算法利用了序列的随机性质，通过找到序列中出现的循环来寻找整数的因子。当序列中出现循环时，可以使用$Floyd$循环检测算法或$Brent$循环检测算法来检测循环的位置。

算法步骤

1. 给定一个待分解的大整数N。
2. 随机选择一个起始值x_0。
3. 定义两个函数$f(x)$和$g(x)$。其中$f(x)$是一个对x进行某种变换的函数，$g(x)$是$f(x)$对应的函数。
4. 使用递归的方式生成一个整数序列：$x[i+1]=f(x[i])$，$y[i+1]=g(x[i])$。
5. 在序列中，不断计算$y[i]$和$y[2i]$之间的最大公约数$c=gcd(y[i]-y[2i],N)$。
6. 如果$c=1$，则回到步骤4，选择不同的起始值x_0。
7. 如果$c=N$，则回到步骤2，选择不同的起始值x_0。
8. 如果c是N的一个因子，则停止算法，并将c作为N的一个质因数，完成分解。

算法伪代码

```
typedef long long LL;
LL Pollard_Rho(LL n,LL c)
{
    LL i=1,j=2,x=rand()%(n-1)+1,y=x;//随机初始化一个基数（p1）
    while(1)
    {
        i++;
        x=(modmul(x,x,n)+c)%n;//玄学递推
        LL p=gcd((y-x+n)%n,n);
        if(p!=1&&p!=n)return p;//判断
        if(y==x)return n;//y为x的备份，相等则说明遇到圈，退出
        if(i==j)
        {
            y=x;
            j<<=1;
        }//更新y，判圈算法应用
    }
}
void find(LL n,LL c)//同上，n为待分解数，c为随机常数
{
    if(n==1)return;
    if(Miller_Rabin(n))//n为质数
```

```
    {
        //保存，根据不同题意有不同写法，在此略去
        return;
    }
    LL x=n,k=c;
    while(x==n)x=Pollard_Rho(x,c--);//当未分解成功时，换个c带入算法
    find(n/x,k);
    find(x,k);
    //递归操作
}
```

此处我们提供在网上找到的C++代码，用于进行该算法的计算。

算法评价

$Pollard\ \rho$算法的性能取决于起始值x_0的选择和函数$f(x)$的设计。在实践中，通常需要多次尝试不同的起始值和函数来增加成功分解整数N的机会。但$Pollard\ \rho$算法并不是一个确定性算法，它可能在某些情况下失败，即无法将整数N分解为质因数。对于特定的整数N，$Pollard\ \rho$算法的效率可能会有所不同。在实际应用中，通常与其他分解算法结合使用，以提高分解的成功率。

1.2.5 数域筛选算法

算法介绍

数域筛选算法（$Number\ Field\ Sieve$）是一种高效的大整数分解算法，广泛应用于现代密码学中。它利用了数论和代数数论的高级技术，在相对较短的时间内分解较大的整数。这也是目前最高效的大整数分解算法，在数字很大时时间复杂度优于其他的方法。

算法步骤

1. 给定一个待分解的大整数N。
2. 选择一个合适的平方数B，使得$B^2 > N$。
3. 选择一个数域($field$)F，它是一个包含实数和复数的集合，其中包含了用于构建曲线方程的元素。
4. 在数域F上，选择一个合适的曲线方程，例如$y^2 = x^3 + ax + b$，其中a和b是数域F中的常数。
5. 在曲线上选择一个基准点P，作为算法的起始点。
6. 生成一组点序列，通过对基准点进行椭圆曲线运算和模N的取模运算，得到一系列的点（x, y）。
7. 对于每个点(x, y)，计算其离散对数，即找到一个整数k，使得$P \times k = (x, y)$。
8. 通过计算离散对数得到一组线性方程。
9. 将这些线性方程构建成一个矩阵A。
10. 使用高级的线性代数技术，例如高斯消元法和列选主元法，对矩阵A进行处理，得到一个较为简化的矩阵。
11. 在简化的矩阵中，寻找一个非零向量，使得对应的列向量之积等于1。
12. 使用找到的向量，得到一个关系式，将其转换为模N的等式。
13. 通过解决这个模N的等式，找到一个非平凡的因子。
14. 将N除以这个因子得到一个新的整数N'。
15. 重复上述步骤，直到N被完全分解为质因数的乘积。

算法伪代码

```
Input: 大整数 N
Procedure NumberFieldSieveFactorization(N):
    // 计算 N 的平方根向上取整
    sqrt_N = ceil(sqrt(N))
    // 初始化素数上限和指数上限
    prime_limit = ceil(exp(sqrt(log(N) * log(log(N)))))
    exponent_limit = ceil(2 * sqrt(log(N) * log(log(N))))
    // 生成素数表
    primes = GeneratePrimeList(prime_limit)
    // 构建指数矩阵
    exponent_matrix = InitializeExponentMatrix(N, primes, exponent_limit)
    // 执行数域筛选
    Sieve(exponent_matrix)
    // 查找线性相关关系
    relations = FindLinearRelations(exponent_matrix)
    // 提取因子
    factors = ExtractFactors(relations)
    Return factors
Output:分解之后的因子
```

算法分析

数域筛法是最快的(渐进意义下)整数分解方法。用于解决IFP和DLP问题，应用于信息安全方面加密算法的破解。这个算法在大整数的分解下的表现是很好的，但其时间复杂度仍然很高，并且对于非常大的整数来说，仍然需要大量的计算资源和时间。在小整数下的表现可能就不如其他一些分解方法了，比如二次筛法等等。

1.3 密码学中的应用

大整数分解在密码学中应用范围很广，尤其是在公钥密码学中的非对称加密，典型的就是RSA算法了。因为我们RSA算法的破解的关键点就是我们能否求出私钥d，也就是我们能否求出我们所需要的大整数的欧拉函数phi，也就是说，只要我们能够分解大整数，我们就可以求出其欧拉函数，进一步我们就可以破解私钥，就可以破解RSA加密体制了。可惜的是，我们到目前还没有一套完整的体系能够在一定的时间复杂度内去对整数进行分解，所以我们在公钥和私钥长度过大的时候，可能就无法破解我们的RSA体系了。

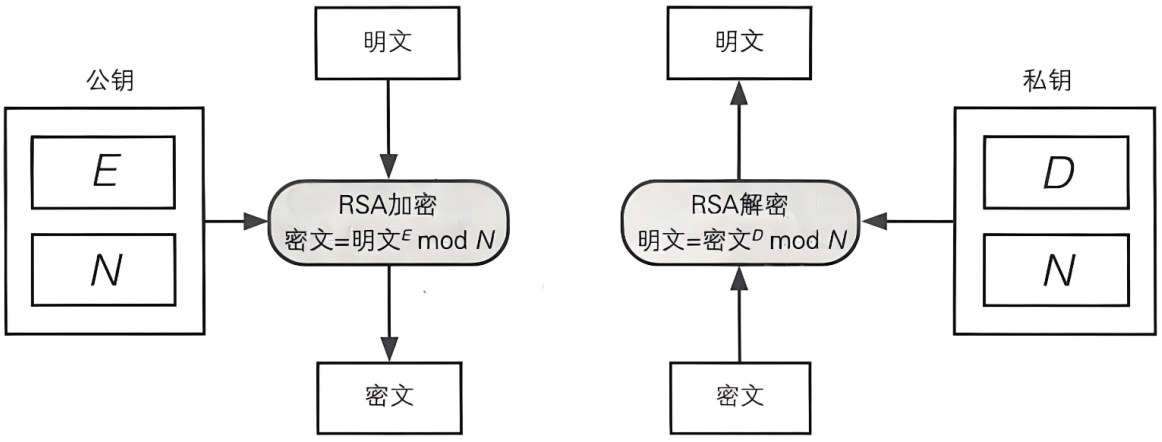
大数因子分解是国际数学界几百年来尚未解决的难题，也是现代密码学中公开密钥RSA算法密码体制建立的基础。也就是说，只要我们成功的解决了大整数的分解问题，我们就完全解决了RSA加密体制，也就是说这套密码体系就不能够在适用了。

当然，也不是所有的RSA密钥体系都牢不可破。在位数较大，且公钥私钥之间呈现一定位数关系的时候，我们还是有许多的方法去破译密码体系的，这部分我们将在下一部分RSA问题中详细阐述。

二、RSA问题

2.1 数学问题

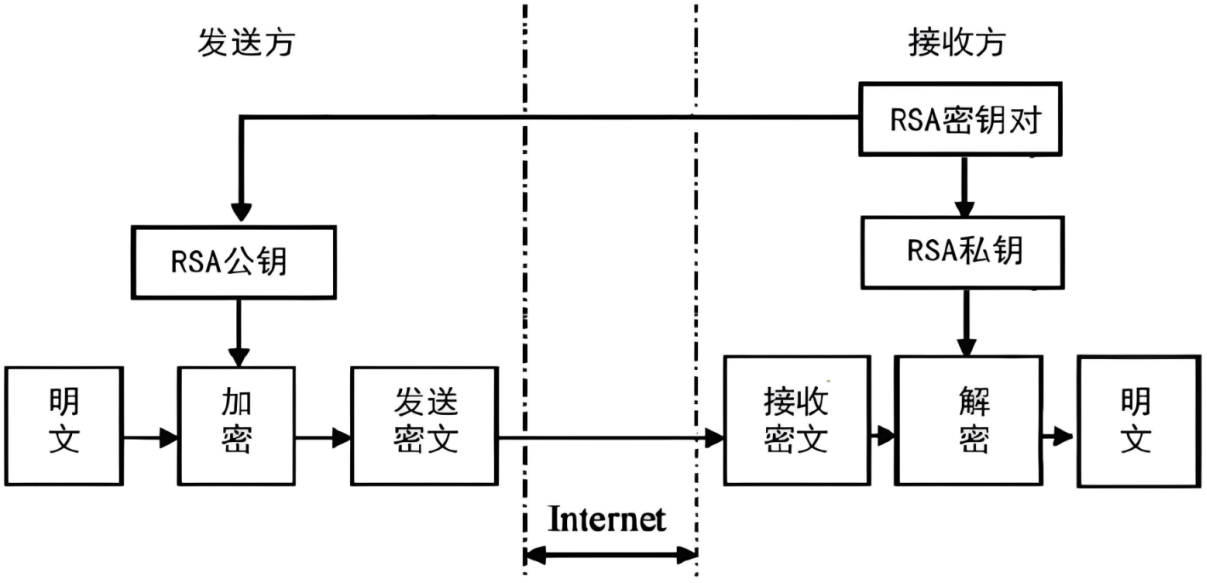
RSA加密算法是一种非对称加密算法，以其创始人$Ron\ Rivest$、$Adi\ Shamir$和$Leonard\ Adleman$的名字命名。该算法的安全性基于大素数难以分解的数学原理。RSA算法的安全性是基于大素数因数分解问题的难度。在数论中，有一个简单的事实：将两个大素数相乘非常容易，但要将它们的乘积进行因式分解却非常困难。因此，可以将这两个大素数的乘积公开作为加密密钥。



我们在RSA算法中会有一些常用的字母，我们在介绍之前，先简单提一下那些字母表示的意思。

> p：就是第一个大质数
>
> q：就是第二个大质数
>
> n：一般来说，就是pq的乘积
>
> $phi(n)$：就是n的欧拉函数
>
> m：一般是需要加密的原文
>
> c：一般是加密完之后的密文
>
> d：私钥，用于解密
>
> e：公钥，用于加密

我们的信息拥有者手头有的是需要加密的原文m和公钥e和大整数n，信息接受者手上有的是加密完之后的密文c和私钥d，用于解密。我们只需要知道我们的私钥，就可以逆向推出我们的原始信息。



但是由于大整数分解的困难性，我们无法通过求解d的值去破解私钥，所以这也使RSA加密体系存在了这么多年，在我们现在的电脑上，有些加密体制还是基于$RSA3072$的，可见其安全性与易懂性。在此处我们不详细介绍该原理，我们将其放在算法思想及特点里。

2.2 算法思想及特点

下面这部分，我会详细地介绍该加密算法的思想、原理和特点。

首先，在介绍之前，我会给出一些初等数论中常见的结论与公理，本文中遇到这些定理与结论我们将不再证明。

> **质数**：只能被1和自身整除的正整数，且大于1。
>
> **互质**：两个正整数的最大公约数为1，则它们互质。
>
> **模运算**：一种整数运算，其结果是被除数除以除数后的余数。
>
> **欧拉函数**$\varphi(n)$：表示小于n且与n互质的正整数的个数。

2.2.1 算法思想与原理

我们首先需要生成密钥，我们有以下的步骤：

> 1. **选择质数**：随机选择两个大且不同的质数p和q。
> 2. **计算模数**：计算p和q的乘积$n = p \times q$。这个n将作为公钥和私钥的一部分，并且是公开的。
> 3. **计算欧拉函数**：计算$\varphi(n) = (p-1) \times (q-1)$。注意，$\varphi(n)$是私钥生成的关键部分，但不应该被公开。
> 4. **选择加密指数**：选择一个整数e，使得$1 < e < \varphi(n)$，并且e与$\varphi(n)$互质。这个e将作为公钥的一部分，用于加密操作。
> 5. **计算解密指数**：找到一个整数d，使得$(e \times d - 1)$能被$\varphi(n)$整除。换句话说，求解模反元素d，满足$e \times d \equiv 1 (mod\ \varphi(n))$。这个$d$将作为私钥的一部分，用于解密操作。

至此，我们得到了公钥(n, e)和私钥(n, d)。公钥可以公开分发给任何人，而私钥必须严格保密，不然整个密码体制就被破坏了。

接下来我们需要对信息进行加密，举个例子，我们需要加密的信息是`Mathematics is the queen of sciences, and arithmetic [number theory] is the queen of mathematics.`

那么这就是我们的需要加密的文字m了。接下来我们生成一组质数，然后算出我们的乘积，另外需要给出我们的e，再利用d传输给解密人，这样我们双方就可以进行加解密了。

我们先对其进行加密，通过$C = M^e\ (mod\ n)$来进行计算，求出了我们的密文c，值为

```
102306866548166297283990570728719380746274356547349456313911134525511009884527595077619658312746154540918040609904792766267540939554230661255790442398718139161414133218935455395440009626652377478022047762911840522170980051387391261774896710421983974318857176556467366545165306790641364917195790444235849931561
```

然后我们将c传输给对方，对方就可以通过c，d，n来解出我们的原文m了！通过计算$M = C^d(\ mod\ n)$就可以求得m的值：

```
721156075061547613364110141099856850563950957473214919673494232477495653879136119681837293285641323913019261856399366962867926733241410500465446031610582623751314859721369608219089460972326894980564507282258352507478729034946935770 37
```

，再将其转化为$ASCII$码就可以获得原先的明文了。

而对于其他人来说，只要不拥有我们的私钥，即使我们截获了我们的密文，没有私钥也是几乎不可能求解出原先的明文的，这也保证了我们的加密体系的安全性。

下面简单说说这个模运算的原理，即RSA加密的证明：

我们需要证明的就是，密文经过私钥的幂取模的结果等于明文，即如下等式成立：

$$x = y^d\ mod\ n$$

根据加密算法得到的密文 y，一定满足如下等式：

$$y = x^e\ mod\ n$$

证明

我们首先证明**当x和n互素的情况：**

我们使用欧拉定理就可以证明了，证明过程如下所示：

$$\begin{aligned} y^d \ mod \ n &= x^{ed} \ mod \ n \\ &= x^{km+1} \ mod \ n \\ &= (x^m \ mod \ n)^k \ x \ mod \ n \\ &= (x^{\phi(n)} \ mod \ n)^k \ x \ mod \ \ n \\ &= x \ mod \ n \end{aligned}$$

> 1. 等式两边同时取d次幂，再取模；
> 2. e和d在模m的域上互为逆元，所以$ed = km + 1$;
> 3. 乘法取模的结合律；
> 4. m为n的欧拉函数，即$m = \phi(n) = \phi(p)\phi(q)$;
> 5. 由于x和n互素，所以根据欧拉定理，有$x^{\phi(n)} mod \ n = 1$,直接代入得证；

当x和n不互素的情况：

当x 和 n 不互素时，由于 $n = pq$，所以x 要么是p 的倍数，要么是q 的倍数，但是不可能同时是两者的倍数，因为这样一来，$x^e \ mod \ n = 0$，就无法加密了。

那么，我们假设 $x = x'p$，这里显然 $gcd(x, q) = 1$，并且可以得到 $qx \ mod \ n = 0$
根据欧拉定理，有：

$$x^{\phi(q)} \equiv 1 (mod \ q)$$

由于$ed \ mod \ m = 1$，所以我们可以令 $ed = km + 1 = k\phi(p)\phi(q) + 1$
对欧拉定理的同余式两边同时乘上 $k\phi(p)$，则有：

$$x^{k\phi(p)\phi(q)} \equiv 1 (mod \ q)$$

则：

$$x^{k\phi(p)\phi(q)} = iq + 1$$

然后我们就可以推导出以下的结果了：

$$\begin{aligned} y^d mod \ n &= x^{ed} \ mod \ n & (1) \\ &= x^{km+1} \ mod \ n & (2) \\ &= x^{k\phi(p)\phi(q)} x \ mod \ n & (3) \\ &= (iq + 1)x \ mod \ n & (4) \\ &= (iqx + x) \ mod \ n & (5) \\ &= x \ mod \ n & (6) \end{aligned}$$

所以综上所述，我们完成了对其的证明！

2.2.2 RSA的安全性

RSA的安全性是我们使用的时候必须考虑的，因为一旦生成的数据有一定的规律，就很容易被破解，我们在下一个版块就会具体罗列一些常见的$RSA \ attack$方式。

> - **密钥长度**：为了保持RSA算法的安全性，必须选择足够大的密钥长度。在现代标准中，通常推荐使用至少2048位的密钥长度，以抵抗已知的攻击方法。
> - **随机数生成**：在密钥生成过程中使用的随机数必须具有良好的随机性，以避免潜在的安全漏洞。

- **参数选择**：选择合适的质数p和q以及加密指数e对于算法的安全性至关重要。通常建议使用安全的参数生成方法来避免常见的陷阱和弱点。
- **已知攻击与防御**：尽管RSA算法被广泛认为是安全的，但仍存在潜在的攻击风险。例如，侧信道攻击可以通过观察加密或解密操作的物理特征（如时间、功耗等）来推测密钥信息。为了防范这些攻击，可以采取相应的防御措施，如使用掩码技术来隐藏关键操作的特征。
- **算法实现与更新**：在实际应用中，需要注意RSA算法的正确实现和及时更新。错误的实现或使用过时的算法库可能导致安全漏洞。因此，建议使用经过充分测试和验证的加密算法库，并定期更新以应对新出现的安全威胁。

2.2.3 常见的$RSA\ attack$方式

$Wiener\ Attack$

$\varphi(n)=(p-1)(q-1)=pq-(p+q)+1=N-(p+q)+1$

$\because p,q$非常大，$\therefore pq\gg p+q,\therefore \varphi(n)\approx N$

$\because ed\equiv 1mod\varphi(n),\therefore ed-1=k\varphi(n)$,这个式子两边同除$d\varphi(n)$可得:

$\frac{e}{\varphi(n)}-\frac{k}{d}=\frac{1}{d\varphi(n)}$

$\because \varphi(n)\approx N,\therefore \frac{e}{N}-\frac{k}{d}=\frac{1}{d\varphi(n)}$,同样$d\varphi(n)$是一个很大的数，所以$\frac{e}{N}$略大于$\frac{k}{d}$

因为e和N是知道的，所以计算出$\frac{e}{N}$后，比它略小的$\frac{k}{d}$,可以通过计算$\frac{e}{N}$的连分数展开，依次算出这个分数每一个渐进分数，由于$\frac{e}{N}$略
大于$\frac{k}{d}$,$Wiener$ 证明了，该攻击能精确的覆盖$\frac{k}{d}$.

在e过大或过小的情况下，可使用算法从e中快速推断出d的值。可以解决$q<p<2q,d<\frac{1}{3}N^{\frac{1}{4}}$的问题。

对应脚本：

```
#Sage
def factor_rsa_wiener(N, e):
    N = Integer(N)
    e = Integer(e)
    cf = (e / N).continued_fraction().convergents()
    for f in cf:
        k = f.numer()
        d = f.denom()
        if k == 0:
            continue
        phi_N = ((e * d) - 1) / k
        b = -(N - phi_N + 1)
        dis = b ^ 2 - 4 * N
        if dis.sign() == 1:
            dis_sqrt = sqrt(dis)
            p = (-b + dis_sqrt) / 2
            q = (-b - dis_sqrt) / 2
            if p.is_integer() and q.is_integer() and (p * q) % N == 0:
                p = p % N
                q = q % N
                if p > q:
                    return (p, q)
                else:
                    return (q, p)
```

低加密指数广播攻击（$Hastad$攻击）

如果一个用户使用同一个加密指数e加密了同一个密文，并发送给了其他e个用户。那么就会产生广播攻击。

脚本：

```
#sage
def chinese_remainder(modulus, remainders):
 Sum = 0
    prod = reduce(lambda a, b: a*b, modulus)
 for m_i, r_i in zip(modulus, remainders):
        p = prod // m_i
     Sum += r_i * (inverse_mod(p,m_i)*p)
    return Sum % prod
```

我们分别输入对应的模数和对应的e就可以解出我们的结果了。

共模攻击

当n不变的情况下，知道n, e_1, e_2, c_1, c_2可以在不知道d_1, d_2的情况下，解出m。

首先假设e_1, e_2互质，

即 $\gcd(e_1, e_2) = 1$

此时则有 $e_1 s_1 + e_2 s_2 = 1$

式中，s_1, s_2皆为整数，但是一正一负。

通过扩展欧几里德算法，我们可以得到该式子的一组解（s_1, s_2），假设s_1为正数,s_2为负数。

因为 $c_1 = m^{e_1} \bmod n, c_2 = m^{e_2} \bmod n$

所以 $(c_1^{s_1} c_2^{s_2}) \bmod n = ((m^{e_1} \bmod n)^{s_1} (m^{e_2} \bmod n)^{s_2}) \bmod n$

根据模运算性质，可以化简为 $(c_1^{s_1} c_2^{s_2}) \bmod n = ((m^{e_1})^{s_1} (m^{e_2})^{s_2}) \bmod n$

即 $(c_1^{s_1} c_2^{s_2}) \bmod n = (m^{e_1 s_1 + e_2 s_2}) \bmod n$

又前面提到 $e_1 s_1 + e_2 s_2 = 1$

所以 $(c_1^{s_1} c_2^{s_2}) \bmod n = m \bmod n$

即 $c_1^{s_1} c_2^{s_2} = m$

破解脚本：

```
import gmpy2 as gp
def egcd(a, b):
    if a == 0:
        return (b, 0, 1)
    else:
        g, y, x = egcd(b % a, a)
        return (g, x - (b // a) * y, y)
n =
c1 =
c2 =
e1 =
e2 =
s = egcd(e1, e2)
s1 = s[1]
s2 = s[2]
if s1<0:
    s1 = - s1
    c1 = gp.invert(c1, n)
elif s2<0:
    s2 = - s2
```

```
    c2 = gp.invert(c2, n)

m = pow(c1,s1,n)*pow(c2,s2,n) % n
print(hex(m)[2:])
print(bytes.fromhex(hex(m)[2:]))
```

Boneh and Durfee attack

e 非常大接近于N,即d 较小时。与低解密指数攻击类似，比低解密指数攻击($Wiener\ Attack$)更强，可以解决$\frac{1}{3}N^{\frac{1}{4}} \leq d \leq N^{0.292}$的问题

$$\because ed = k\varphi + 1$$

$$\therefore k\varphi + 1 \equiv 0 \pmod{e} \Rightarrow k(N+1-p-q)+1 \equiv 0 \pmod{e} \Rightarrow 2k(\frac{N+1}{2} + \frac{-p-q}{2}) \equiv 0 \pmod{e}$$

设$A = \frac{N+1}{2}, y = \frac{-p-q}{2}, x = 2k$,有$f(k,y) = 1 + x \cdot (A+y)$

如果在模e下解得该方程的根x, y,由$ed = 1 + x \cdot (A+y)$可以得到d。

Coppersmith Attack

已知d的低位:

如果知道d的低位，低位约为n的位数的$\frac{1}{4}$ ($\frac{n.nbits()}{4}$)就可以恢复d。

```
#Sage
def partial_p(p0, kbits, n):
    PR.<x> = PolynomialRing(Zmod(n))
    nbits = n.nbits()
    f = 2^kbits*x + p0
    f = f.monic()
    roots = f.small_roots(X=2^(nbits//2-kbits), beta=0.4)  # find root < 2^(nbits//2-
kbits) with factor >= n^0.4
    if roots:
        x0 = roots[0]
        p = gcd(2^kbits*x0 + p0, n)
        return ZZ(p)
def find_p(d0, kbits, e, n):
    X = var('X')
    for k in range(1, e+1):
        results = solve_mod([e*d0*X - k*X*(n-X+1) + k*n == X], 2^kbits)
        for x in results:
            p0 = ZZ(x[0])
            p = partial_p(p0, kbits, n)
            if p and p != 1:
                return p
if __name__ == '__main__':
    n =
    e =
    c =
    d0 =
    beta = 0.5
    nbits = n.nbits()
    kbits = d0.nbits()
    print("lower %d bits (of %d bits) is given" % (kbits, nbits))
    p = int(find_p(d0, kbits, e, n))
```

```
    print("found p: %d" % p)
    q = n//int(p)
    print("d:", inverse_mod(e, (p-1)*(q-1)))
```

已知m的高位：

e足够小，且部分明文泄露时，可以采用$Coppersmith$单变量模等式的攻击，如下：

$c = m^e \ mod \ n = (mbar + x_0)^e \bmod n$,其中$mbar = (m >> kbits)$

当$|x_0| \leq N^{\frac{1}{e}}$时，可以在$\log N$和$e$的多项式时间内求出$x_0$.

```
#Sage
n =
e =
c =
mbar =
kbits =
beta = 1
nbits = n.nbits()
print("upper {} bits of {} bits is given".format(nbits - kbits, nbits))
PR.<x> = PolynomialRing(Zmod(n))
f = (mbar + x)^e - c
x0 = f.small_roots(X=2^kbits, beta=1)[0]  # find root < 2^kbits with factor = n
print("m:", mbar + x0)
```

已知p的高位：

知道一半的位数就可以破解了。

```
#Sage
n =
p4 =  #p去0的剩余位
pbits = 1024
kbits = pbits - p4.nbits()
print(p4.nbits())
p4 = p4 << kbits
PR.<x> = PolynomialRing(Zmod(n))
f = x + p4
roots = f.small_roots(X=2^kbits, beta=0.4)
if roots:
    p = p4 + int(roots[0])
    q = n//p
    print(f'n: {n}')
    print(f'p: {p}')
    print(f'q: {q}')
```

Common Private Exponent

加密用同样的私钥并且私钥比较短，从而导致了加密系统被破解。

我们已知：

$$\begin{cases} e_1 d = 1 + k_1 \varphi(N_1) \\ e_2 d = 1 + k_2 \varphi(N_2) \\ \vdots \\ e_r d = 1 + k_r \varphi(N_r) \end{cases}$$

其中，$N_1 < N_2 < \cdots < N_r < 2N_1$。

我们构造格：

$$B_r = \begin{bmatrix} M & e_1 & e_2 & \cdots & e_r \\ 0 & -N_1 & 0 & \cdots & 0 \\ 0 & 0 & -N_2 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & -N_r \end{bmatrix}$$

其中$M = \lfloor N_r^{\frac{1}{2}} \rfloor$。

再利用LLL算法进行规约得到$|b_1| = Md$,则 $d = \frac{|b_1|}{M}$,从而解密密文得到明文。

使用条件：

$$d < N_r^{\delta_r},\ \delta_r < \frac{1}{2} - \frac{1}{2(r+1)} - \log_{N_r}(6)$$

2.3 密码学中的应用

我们在$windows$系统上很多地方都能见到RSA加密的身影，包括很多开源项目中，为了设置私有属性，很多都是通过添加RSA密钥来设置密码的，比如$github$等。下面我们就展示一下常见的生成RSA密钥对的方法。

```
import java.security.*;
import java.security.spec.PKCS8EncodedKeySpec;
import java.security.spec.X509EncodedKeySpec;
import java.util.Base64;

public class RSAExample {

    // 生成密钥对
    public static KeyPair generateKeyPair() throws NoSuchAlgorithmException {
        KeyPairGenerator keyPairGenerator = KeyPairGenerator.getInstance("RSA");
        keyPairGenerator.initialize(2048); // 设置密钥长度为2048位
        return keyPairGenerator.generateKeyPair();
    }

    // 将私钥转换为字符串形式以便存储
    public static String privateKeyToString(PrivateKey privateKey) {
        byte[] encoded = privateKey.getEncoded();
        return Base64.getEncoder().encodeToString(encoded);
    }

    // 从字符串形式恢复私钥
    public static PrivateKey stringToPrivateKey(String privateKeyStr) throws
GeneralSecurityException {
        byte[] encoded = Base64.getDecoder().decode(privateKeyStr);
        PKCS8EncodedKeySpec keySpec = new PKCS8EncodedKeySpec(encoded);
        KeyFactory keyFactory = KeyFactory.getInstance("RSA");
        return keyFactory.generatePrivate(keySpec);
    }
```

```
    // 将公钥转换为字符串形式以便存储
    public static String publicKeyToString(PublicKey publicKey) {
        byte[] encoded = publicKey.getEncoded();
        return Base64.getEncoder().encodeToString(encoded);
    }

    // 从字符串形式恢复公钥
    public static PublicKey stringToPublicKey(String publicKeyStr) throws
GeneralSecurityException {
        byte[] encoded = Base64.getDecoder().decode(publicKeyStr);
        X509EncodedKeySpec keySpec = new X509EncodedKeySpec(encoded);
        KeyFactory keyFactory = KeyFactory.getInstance("RSA");
        return keyFactory.generatePublic(keySpec);
    }

    // 使用公钥加密数据
    public static byte[] encrypt(PublicKey publicKey, byte[] data) throws
GeneralSecurityException {
        Cipher cipher = Cipher.getInstance("RSA");
        cipher.init(Cipher.ENCRYPT_MODE, publicKey);
        return cipher.doFinal(data);
    }

    // 使用私钥解密数据
    public static byte[] decrypt(PrivateKey privateKey, byte[] encryptedData) throws
GeneralSecurityException {
        Cipher cipher = Cipher.getInstance("RSA");
        cipher.init(Cipher.DECRYPT_MODE, privateKey);
        return cipher.doFinal(encryptedData);
    }

    public static void main(String[] args) {
        try {
            // 生成密钥对
            KeyPair keyPair = generateKeyPair();
            PublicKey publicKey = keyPair.getPublic();
            PrivateKey privateKey = keyPair.getPrivate();

            // 将密钥转换为字符串并打印
            String publicKeyStr = publicKeyToString(publicKey);
            String privateKeyStr = privateKeyToString(privateKey);
            System.out.println("公钥: " + publicKeyStr);
            System.out.println("私钥: " + privateKeyStr);

            // 模拟加密和解密过程
            String originalMessage = "这是一个需要加密的消息";
            System.out.println("原始消息: " + originalMessage);

            // 加密
            byte[] encryptedData = encrypt(publicKey, originalMessage.getBytes());
            System.out.println("加密后的数据: " +
Base64.getEncoder().encodeToString(encryptedData));

            // 解密
            PrivateKey restoredPrivateKey = stringToPrivateKey(privateKeyStr);
            byte[] decryptedData = decrypt(restoredPrivateKey, encryptedData);
            System.out.println("解密后的消息: " + new String(decryptedData));

        } catch (Exception e) {
```

```
            e.printStackTrace();
        }
    }
}
```

以上的代码完成了生成2048位RSA密钥的过程，还完成了保存密钥、使用公钥加密数据以及使用私钥解密数据。

首先，我们生成了一个RSA密钥对，然后将公钥和私钥转换为字符串形式以便存储或传输。接着模拟了一个加密和解密的过程：使用公钥加密一条消息，然后使用私钥解密这条消息。实际应用中应该使用更安全的方式来存储和传输密钥，比如使用安全的密钥存储库或硬件安全模块（HSM）。此外，对于大量的数据加密，推荐使用对称加密算法（如AES），并使用RSA等非对称算法来安全地传输对称加密密钥。

RSA算法作为一种非对称加密算法，在多个领域有广泛的应用，主要包括：

> - **网络通信安全**：RSA算法可以用于保护网络通信的安全，比如$HTTPS$、SSH等协议都使用了RSA算法来加密通信过程中的数据，以此确保数据在传输过程中的安全性。
> - **数字签名**：RSA算法也可以用于数字签名，保证数据的完整性和真实性。在电子商务中，商家就可以使用RSA算法对订单进行数字签名，确保订单的真实性和完整性，防止数据被篡改或伪造。
> - **身份认证**：RSA算法还可以用于身份认证，比如在网银等场景中，用户可以使用RSA算法生成一对公私钥，将公钥发送给银行，银行使用公钥对数据进行加密，只有用户拥有私钥才能解密，从而实现身份认证。
> - **电子邮件加密**：RSA算法同样可以用于电子邮件加密，确保邮件的机密性和安全性。只有持有私钥的收件人才能解密和阅读邮件内容。
> - **VPN（虚拟私人网络）**：RSA算法可以用于创建VPN，保护网络通信的隐私和安全。通过RSA算法加密VPN连接中的数据，可以确保数据在公共网络上的安全性。
> - **数字证书**：RSA算法还可以用于数字证书，用于认证和验证数字签名。数字证书是一种电子文档，用于证明公钥的拥有者的身份，通常用于网站的身份验证和安全通信。

综上所述，RSA算法是一种广泛使用的公钥加密算法，它的安全性基于大数分解和离散对数等数学难题。该算法利用一对密钥（公钥和私钥）进行加密和解密操作，其中公钥可以公开分发，用于加密信息，而私钥必须保密，用于解密信息。RSA算法的核心思想在于通过一系列数学运算，将明文转换为密文，并且只有持有相应私钥的人才能解密出原始明文。

在实际应用中，RSA算法通常用于数字签名、身份验证和数据加密等场景。它的优点在于易于实现和理解，同时具有较高的安全性。然而，随着计算能力的不断提升和新型攻击手段的出现，RSA算法也面临着一些安全挑战。为了应对这些挑战，研究者们不断提出改进方案和新算法来增强RSA算法的安全性。

三、椭圆曲线在密码学中的应用

3.1 密码原语

3.1.1 椭圆曲线的定义

椭圆曲线就是三次平滑代数平面曲线，用代数几何的语言说就是亏格为1的代数曲线。域F上的椭圆曲线E就是满足下列非奇异的$Weierstrass$方程的所有点(x,y)的集合：

$$y^2+a_1xy+a_3y=x^3+a_2x^2+a_4x+a_6$$

当域的特征不为$2,3$时，$Weierstrass$方程可以转化为下面的形式：

$$y^2=x^3+ax+b$$

其中要求判别式为：

$$\Delta = 4a^3 + 27b^2 \neq 0$$

要注意的是，椭圆曲线与椭圆并不是完全相同的，因为椭圆曲线是一个二次方程，而椭圆却是一个图形而已。唯一存在联系的地方就是，在求椭圆的弧长的时候，弧长的积分与椭圆曲线有一定的联系，积分如下所示：

$$\int \frac{1}{x^3 + ax^2 + bx + c} \mathrm{d}x$$

通过切割线法则，可以在椭圆曲线上定义一个群运算，从而使得椭圆曲线上点的全体构成一个加法群。在一个特征不为$2, 3$的域上，这些点在下面定义的加法运算下构成一个$Abelian$群：

群运算的恒等元是O(称为无穷远点),设P和Q 是椭圆曲线上的两个点，若 $P = O$,则$-P = O$,且$P + Q = Q + P = Q$; 令$P = (x_1, y_1)$, $Q = (x_2 , y_2)$,则$-P = (x_1 , -y_1)$;如果$Q \neq -P$,则$P + Q = (x_3, y_3)$, 这里$x_3 = \mu^2 - x_1 - x_2, y_3 = \mu(x_1 - x_3)$ -y_1,其中当$Q \neq P$时, $\mu = \frac{y_2 - y_1}{x_2 - x_1}$;当$Q = P$时, $\mu = \frac{3x_1^2 + a}{2y_1}$。

以上就是椭圆曲线的定义，我们再来说一说几种常见的椭圆曲线。

3.1.2 常见的椭圆曲线

通过查阅资料可以知道，常用的椭圆曲线有很多种，上面所提到的只是其中一种椭圆曲线的标准曲线。一般材料会以维尔斯特拉斯曲线（$Weierstrass\ Curve$）为例介绍椭圆曲线的基本概念和运算原理，这是因为**任意椭圆曲线都可以写为$Weierstrass\ Curve$形式**。实际上，椭圆曲线还包括多种其他的类型，如蒙哥马利曲线（$Montgomery\ Curve$）、扭曲爱德华曲线（$Twisted\ Edwards\ Curve$）等。

下面我们就来简单介绍一下常见的这三种曲线的表示形式以及如何互相转化。

首先是标准曲线，椭圆曲线的一般形式可表示为：$E: y^2 = x^3 + Ax + B$，其中$A, B \in F_p, 4A^3 + 27B^2 \neq 0$ 。一般称上式为维尔斯特拉斯形式的椭圆曲线方程。

蒙哥马利形式的椭圆曲线方程定义为：$Kt^2 = s^3 + Js^2 + s$，其中$K, J \in F_p, B(A^2 - 4) \neq 0$ 。

扭曲爱德华形式的椭圆曲线方程定义为：$av^2 + w^2 = 1 + dv^2w^2$，其中$a, d \neq 0, a \neq d$ 。

这三种椭圆曲线之间都可以互相转化。下面我给出在我最近的CTF竞赛时遇到的问题，题目需要将几种椭圆曲线相互转换，我对此编写了代码脚本：

```
def twisted_to_Montgomery(x, y, a, d, p):
    """The map for twisted Edwards curves point to Montgomery curves point
    x, y: twisted Edwards curves point
    a, d: twisted Edwards curves sach that a*x^2 + y^2 = 1 + d*x^2*y^2
    P: The field K(p)
    Return: mapped point (u, v) and the Montgomery curves parmteres (A, B)
    """
    A = 2*(a + d) * inverse_mod(a-d, p) % p
    B = 4 * inverse_mod(a-d, p)

    u = (1+y) * inverse_mod((1-y), p) % p
    v =  u * inverse_mod(x, p)

    assert B*v**2 % p == (u**3 + A*u**2 + u) % p; "Error"

    return u, v, A, B

def Montgomery_to_Weierstrass(x, y, A, B, p):
    """The map from Montgomery curves point to Weierstrass curves point
    x, y: Montgomery curves point
    A, B: Montgomery curves sach that B*v^2 = u^3 + A*u^2 + u
    P: The field K(p)
```

```
    """
    a = (3 - A**2) * inverse_mod(3*B**2, p) % p
    b = (2*A**3 - 9*A) * inverse_mod(27*B**3, p) % p

    t = (3*x + A) * inverse_mod(3*B, p) % p
    v = y * inverse_mod(B, p) % p

    assert v**2 % p == (t**3 + a*t + b) % p; "Error"

    return t, v, a, b

def twisted_to_Weierstrass(x, y, a, d, p):
    """The map for twisted Edwards curves point to Weierstrass curves point
    x, y: twisted Edwards curves point
    a, d: twisted Edwards curves sach that a*x^2 + y^2 = 1 + d*x^2*y^2
    P: The field K(p)
    Return: mapped point (t, v) and the Weierstrass curves parmteres (a, b)
    """
    u, v, A, B = twisted_to_Montgomery(x, y, a, d, p)
    return Montgomery_to_Weierstrass(u, v, A, B, p)
```

以上就是三种常见的椭圆曲线相互转化的脚本，我们直接使用这些脚本就可以实现相互的转化。

3.2 具体的应用场景

3.2.1 $Elliptic\ Curve\ Diffie-Hellman$

通过素数域下的椭圆曲线的标量乘法来实现。流程如下：

$Alice$ 和 Bob 生成各自的公私钥。假设$Alice$ 的私钥是d_A ,公钥则为$H_A = d_AG$,Bob 的则是d_B和$H_B = d_BG$。他们用的同一个基点G、同一个整数有限域，同一条椭圆曲线。

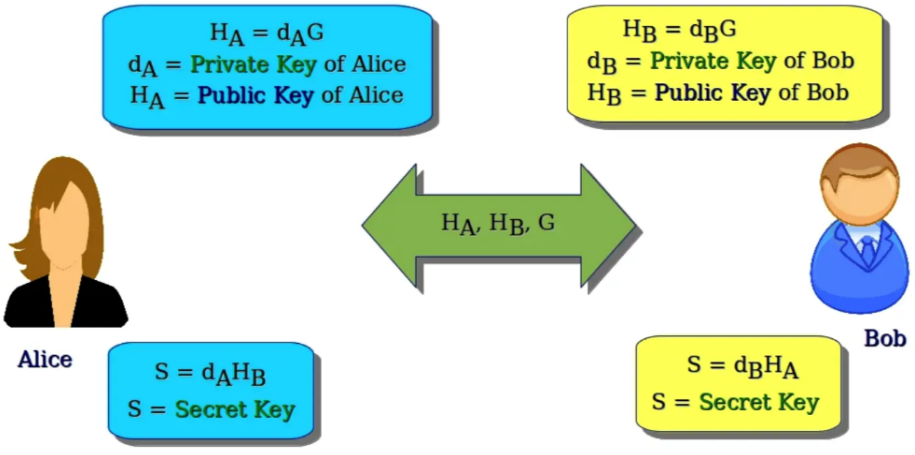
$Alice$ 和Bob通过不安全的信道交换公钥H_A和H_B 。

$Alice$ 计算$S = d_AH_B$,Bob 计算$S = d_BH_A$,共享密钥就是S 。对称加密算法 AES或者 $3DES$ 只用S的一个坐标如x坐标作为密钥即可。

$$S = d_AH_B = d_BH_A = d_Ad_BG$$

要想破解密钥就好比"已知G,aG,bG,求a和b？",当a和b很大的时候，破解是很困难的，这也被称为椭圆曲线的离散对数问题，即 $ECDLP$。

具体的流程图如下所示：



3.2.2 $ECDSA$

$ECDSA$ 是DSA 算法的变种，常用于数字签名。$Alice$ 通过椭圆曲线算法生成公私钥d_A, H_A , 对要签名的消息通过哈希算法生成摘要z (z为整数),然后 $Alice$ 用私钥d_A按照下面步骤对摘要z生成签名。Bob 通过公钥H_A验证签名。

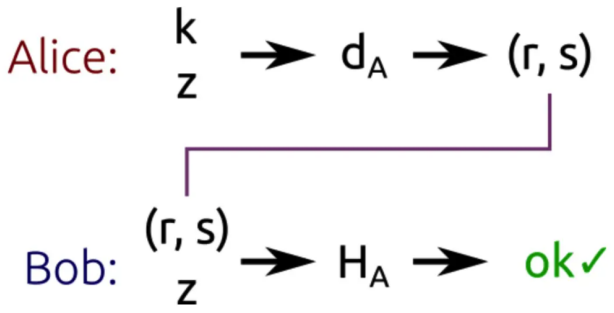
具体的生成签名方式如下所示：

> **1)Alice 选取一个随机数 k,其中**$k \in \{1, 2, \dots n-1\}$**,n为子群的阶。**
>
> **2)计算**$P = kG$**,G 是曲线的基点。**
>
> **3)计算**$r = x_P \bmod n$**,如果**$r = 0$**,则回到第1步重新选择一个 k 重试。**
>
> **4)计算**$s = k^{-1}(z + rd_A) \bmod n$**。如果**$s = 0$**,则回到第1步重新选择一个k 重试。**
>
> **5)**(r, s)**就是最终的签名对。**

在生成签名后，我们需要校验签名。我们通过以下的方式去校验签名：

> **1) 计算** $u_1 = s^{-1}z \bmod n$。
>
> **2) 计算** $u_2 = s^{-1}r \bmod n$。
>
> **3) 计算** $P = u_1G + u_2H_A$。
>
> **4) 如果** $x_P \bmod n = r$**，则签名有效**

下面是该方法的简单示意图：



我们来证明一下该方法的正确性：

> **证明**
>
> 因为 $H_A = d_AG$，于是：
>
> $$P = u_1G + u_2H_A = u_1G + u_2d_AG = (u_1 + u_2d_A)G$$
> $$= ((s^{-1}z + s^{-1}rd_A)\ mod\ n$$
>
> 而之前定义有：$s = k^{-1}(z + rd_A) \bmod n \Rightarrow k = s^{-1}(z + rd_A) \bmod n$。
>
> 因此 $P = (s^{-1}(z + d_A)\ mod\ n$，这与生成签名时一致，故而证明完毕。

$ECDSA$是DSA的一种变化形式，经常用于数字签名等工作。

3.3 包含的数学问题

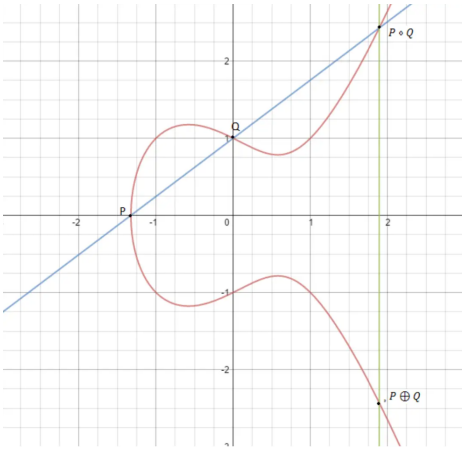
3.3.1椭圆曲线中的群论

椭圆曲线中的点加法

椭圆曲线上的点经过一种特定的加法运算可以让椭圆曲线在实数域构成一个群。首先就是我们的点加法。

我们先定义一下无穷远点的运算法则。定义一个无穷远点 O，即经过椭圆上任意一点的与X轴垂直的直线都经过该点。可能有人疑惑垂直于X轴的直线是平行线，为啥可以定义为都经过O点？因为在非欧几何中，可认为平行线在无穷远处会交于一点。

椭圆曲线点加法：椭圆曲线上经过P和Q两个点的直线与椭圆曲线的交点记作$R = P \diamond Q$,根据定义有$P \diamond Q = Q \diamond P$以及$O \diamond (O \diamond P) = P$。继而定义椭圆曲线点加法：$P \bigoplus Q = O \diamond (P \diamond Q)$,即加法结果是经过点$P \diamond Q$且与$X$轴垂直的直线与椭圆曲线的另外一个交点，简单来说，就是交点R关于X轴的对称点。下图就是做点加法的具体操作。



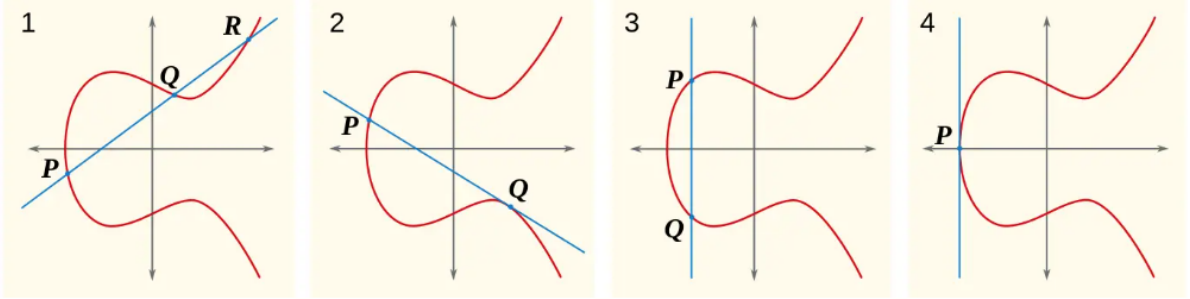
椭圆曲线群:定义为椭圆曲线在实数域上的点集以及点加法 $(E, \bigoplus)$，我们在此处就不详细证明其群的性质了，篇幅较长。

由此可知，椭圆曲线上的点在椭圆曲线加法运算上构成了一个阿贝尔群。增加了单位元后，椭圆曲线方程改为：

$$E = \{(x, y) \in \mathbb{R}^2 \mid y^2 = x^3 + ax + b,\ 4a^3 + 27b^2 \neq 0\} \bigcup \{O\}$$

由定义可知，$P \bigoplus Q = O \diamond (P \diamond Q) = O \diamond R = -R$,所以，最终加法只需要计算交点$R$的逆元$-R$即可。几种特殊情况说明：

> - 如果P, Q不是切点且不是互为逆元，则有第三个交点R,故$P \bigoplus Q = -R$。
> - 如果P或者Q是切点，则PQ就是椭圆曲线的一条切线。假如Q是切点，则有$Q \bigoplus Q = -P$。
> - 如果P和Q连线垂直于X轴，即$P = -Q$,则跟曲线没有第三个交点，可以认为是交于无穷远点O,故而$P \bigoplus Q = O$ 。
> - 如果$P = Q$,则过它们的直线就是椭圆曲线过点P的切线，该直线一般来说跟椭圆曲线有另一个交点R。如果恰好该切线如图4这样跟曲线没有其他交点，则可以认为交点为O,即此时$P \bigoplus P = O$ 。



上图就说明了几种特殊情况，当点在某些特殊的位置上的时候，就会有一些特殊的计算方法。

椭圆曲线中的代数加法

上面的一部分，我们定义了椭圆曲线几何上意义的点加法，需要转换为代数加法以方便计算。**要注意的是，这并不是两个点的坐标简单相加。**

假设直线PQ的斜率m,然后将直线方程$y=mx+c$代入曲线可以得到$(mx+c)^2=x^3+ax+b$,转换成标准式，根据韦达定理$x_P+x_Q+x_R=m^2$,即而可求得$R(x_R,y_R)$。

$$x_R=m^2-x_P-x_Q$$
$$y_R=y_P+m(x_R-x_P)$$

斜率m计算需要区分两种情况，当$P=Q$时求椭圆曲线在P点的切线斜率(求导)即可：

$$m=\frac{y_Q-y_P}{x_P-x_Q}\quad(P\neq Q)$$
$$m=\frac{3x_P^2+a}{2y_P}\quad(P=Q)$$

下面给出椭圆曲线的代数加法的python脚本：

```
    def add(self, P, Q):
        mul_inv = lambda x: pow(x, -1, self.p)
        x1, y1 = P
        x2, y2 = Q
        if P!=Q:
            l=(y2-y1)*inverse(x2-x1,self.p)
        else:l=(3*x1**2+2*self.a*x1+1)*inverse(2*self.b*y1,self.p)
        temp1 = (self.b*l**2-self.a-x1-x2)
        temp2 = ((2*x1+x2+self.a)*l-self.b*l**3-y1)
        x3 = temp1
        y3 = temp2
        return x3, y3
```

椭圆曲线中的标量乘法

我们在椭圆曲线的加密过程中，不能只使用加法运算，这样很容易被破译，所以我们还需要利用一些标量乘法来进行运算。下面简单的说一说它的原理：

我们首先使用一个椭圆曲线A，先对其自身作叠加操作，然后不停地做向量加法运算，直到加到n次为止。这样我们就完成了标量乘法的运算。

$$A\bigoplus A=B\rightarrow A\bigoplus B=C\rightarrow A\bigoplus C=D\rightarrow\dots$$

不过，当n很大时，执行 n 次加法需要$O(n)$时间，效率有问题。因为椭圆曲线点加法在实数域构成阿贝尔群，满足交换律和结合律，于是可以通过$[Double-and-add]$算法进行优化。比如$n=151$,其二进制表示为10010111_2,通过优化只要7次倍乘和4次加法计算即可，时间复杂度降到$O(logn)$。这是一个很好的单向函数，正向计算容易，而反向和蛮力计算复杂。

$$151P=2^7P\bigoplus 2^4P\bigoplus 2^2P\bigoplus 2^1P\bigoplus 2^0P$$

令$Q=nP$,则Q作为公钥，n 为私钥。如果要破解该密钥，问题就是"$Q=nP$,如果已知P和 Q,如何求解n"? 这个问题是比较困难的。不过由于在实数域上曲线连续，可能会更容易找到一些规律进行破解。而且实数域上数值大小没有限制、浮点数等问题而导致计算效率问题，在实际应用中常将椭圆曲线限制到一个有限域内，将曲线变成离散的点，这样即方便了计算也加大了破解难度。

下面给出椭圆曲线的标量乘法的python脚本：

```
    def mul(self, x, P):
        Q = SECRET
        while x > 0:
            if x & 1:
                Q = self.add(Q, P)
            P = self.add(P, P)
            x >>= 1
        return Q
```

3.3.2 有限域椭圆曲线

有限域的椭圆曲线是上面的椭圆曲线的加强版，相当于限制在了一个模n的范围内作椭圆曲线运算。有限域椭圆曲线在CTF中非常常见，对于其原理我们在前面一部分都已经提到了，就是加上一个模运算，这里就不再赘述了。我们还是给出在python脚本下的椭圆有限域运算：

加法：

```
    def add(self, P, Q):
        mul_inv = lambda x: pow(x, -1, self.p)
        x1, y1 = P
        x2, y2 = Q
        if P!=Q:
            l=(y2-y1)*inverse(x2-x1,self.p)%self.p
        else:l=(3*x1**2+2*self.a*x1+1)*inverse(2*self.b*y1,self.p)%self.p
        temp1 = (self.b*l**2-self.a-x1-x2)%self.p
        temp2 = ((2*x1+x2+self.a)*l-self.b*l**3-y1)%self.p
        x3 = temp1
        y3 = temp2
        return x3, y3
```

乘法：

```
    def mul(self, x, P):
        Q = SECRET
        x = x % self.p
        while x > 0:
            if x & 1:
                Q = self.add(Q, P)
            P = self.add(P, P)
            x >>= 1
        return Q
```

对于安全性而言，相比于RSA加密，我们的椭圆曲线还是相对来说比较安全的，因为我们是在一个有限域下做运算，涉及到很多复杂的运算，比如说离散对数，标准曲线等等。这也使破译难度上升了一个等级。但是如果我们不注意生成的椭圆曲线上的点，也会出现很多的破解手段，此处由于过为复杂，我只在此列举几种我在CTF竞赛时遇到的较难的问题。

第一个就是，生成的点可能在椭圆曲线的奇异曲线上，这会导致密钥变得非常不安全。我们有以下的脚本可以破解在奇异曲线上的点的自身的两个未知数。

```
def attack(p, a2, a4, a6, Gx, Gy, Px, Py):
    """
    Solves the discrete logarithm problem on a singular curve (y^2 = x^3 + a2 * x^2 + a4
* x + a6).
    :param p: the prime of the curve base ring
```

```
    :param a2: the a2 parameter of the curve
    :param a4: the a4 parameter of the curve
    :param a6: the a6 parameter of the curve
    :param Gx: the base point x value
    :param Gy: the base point y value
    :param Px: the point multiplication result x value
    :param Py: the point multiplication result y value
    :return: l such that l * G == P
    """
    x = GF(p)["x"].gen()
    f = x ** 3 + a2 * x ** 2 + a4 * x + a6
    roots = f.roots()

    # Singular point is a cusp.
    if len(roots) == 1:
        alpha = roots[0][0]
        u = (Gx - alpha) / Gy
        v = (Px - alpha) / Py
        return int(v / u)

    # Singular point is a node.
    if len(roots) == 2:
        if roots[0][1] == 2:
            alpha = roots[0][0]
            beta = roots[1][0]
        elif roots[1][1] == 2:
            alpha = roots[1][0]
            beta = roots[0][0]
        else:
            raise ValueError("Expected root with multiplicity 2.")

        t = (alpha - beta).sqrt()
        u = (Gy + t * (Gx - alpha)) / (Gy - t * (Gx - alpha))
        v = (Py + t * (Px - alpha)) / (Py - t * (Px - alpha))
        return int(v.log(u))

    raise ValueError(f"Unexpected number of roots {len(roots)}.")
```

这种破解方法利用了其在奇异曲线上，导致会有一些加密上的缺陷，从而进行破译。还有很多的破译方式，在此处由于篇幅原因，我们就不一一列举了。