# 密码工程实验报告

实验名称 Montgomery multiplication and Barrett reduction

# 目录

**1 实验目的** **2**

**2 Montgomery 模乘实现** **2**

2.1 算法介绍 . . . . . . 2

2.2 算法原理 . . . . . . 2

2.3 代码实现 . . . . . . 4

2.3.1 C 代码实现 . . . . . . 5

2.3.2 python 代码实现 . . . . . . 7

**3 Barrett reduction 实现** **9**

3.1 算法原理 . . . . . . 9

3.2 C 代码实现 . . . . . . 10

3.3 python 代码实现 . . . . . . 10

**4 结果验证** **11**

**5 分工和实验心得** **13**

# 1 实验目的

本实验目的是实现 Montgomery multiplication and Barrett reduction. 其中 Montgomery 乘法是公钥算法实现中的一个核心的算法，其主要作用是为模乘运算进行加速。Barrett 算法则是一种大数的归约算法，它能够用乘积运算替代大数取模中耗时的除法部分，从而对大数取模运算进行加速。

# 2 Montgomery 模乘实现

## 2.1 算法介绍

**基本思想:**

- 对任意正整数 N，可选取 $R > N$ 且 $gcd(N, R) = 1$, 通过蒙哥马利乘法将 mod N 的运算转化为 mod R 的运算
- 为什么要把 mod N 转化为 mod R
- 因为选择恰当的 R 能更容易的进行计算，例如 mod 251→mod 256，这样 256 是 2 的幂次的形式，在计算机中处理就可以直接进行移位计算.

## 2.2 算法原理

蒙哥马利模乘依赖于蒙哥马利形式的数字的特殊表示，该算法使用 $a$ 和 $b$ 的蒙哥马利形式来有效的计算 $ab \pmod N$ 的蒙哥马利形式. 那么蒙哥马利乘法可由下面几步来完成.

- Find a auxiliary(辅助的) modulus $R$ such that $gcd(N, R) = 1$,and we have:

$$aR + bR \equiv (a + b)R \bmod N, aR - bR \equiv (a - b)R \bmod N.$$

- Calculate $\bar{a} \leftarrow aR \bmod N$ and $\bar{b} \leftarrow bR \bmod N$.

- Let $R'$ be an integer such that $RR' \equiv 1 \bmod N$,the integer $R^{-1}$ exists,since $R$ and $N$ are coprime.
- Our goal is to compute $\bar{c} \leftarrow \bar{c}\bar{b}R^{-1} \bmod N$.

下面我将举一个例子来说明蒙哥马利模乘的具体过程，从实例中进行分析能够更好的理解算法。

- Key point:105 mod 17
- It will be much simplified if $T$ is smaller than $N$ or close to $N$.
- It can be simplified if we consider $TR^{-1} \bmod N$ with $R = 100$.
- We are happy if 105|100.But it is not true.
- Solution:raise the $T$
- $105 \rightarrow 105 + 17m$ and $105 + 17m = 105 \bmod 17$
- Search for m such that $(105 + 17m)|100$.
  - $mN = -T \bmod R$
  - $m = -105 \cdot 17^{-1} = 105 \cdot 47 \bmod 100$ with $17 \cdot 17 = -1 \bmod 100$
  - $m = 105 \cdot 47 \bmod 100 = (105 \bmod 100) \cdot 47 \bmod 100 = 35$
- $105 \cdot 100^{-1} \bmod 100 = (105 \bmod 100) \cdot 47 \bmod 100 = 35$
- $105 * 100^{-1} \bmod 17 = (105 + 35 * 17)/100 = 700/100 = 7$
- Let's return back to the case $7 \cdot 15 \bmod 17$
- Calculate $\bar{7} = 700 = 3 \bmod 17$ and $\bar{15} = 1500 = 4 \bmod 17$
- Caculate $3 \cdot 4 = 12 \bmod 17$.

- Instead,we can calculate $12 \cdot 100^{-1} \bmod N = 12 \cdot 100^{-1} \bmod 17 = 11$

**算法优势:**

- 对于奇数 $N$, 选取 $R = 2^k > N$, 则必有 $gcd(R, N) = 1$
- $T \bmod R$ 可以转化为位与运算
- $t = (T + mN)/R$ 可以转化位右移运算

## 2.3 代码实现

为了计算蒙哥马利乘法，需要提前计算好下面的几个参数:
$r2 : R^2 \equiv r2 \bmod N$
$N' : NN' \equiv -1 \bmod R, N' \equiv N^{-1} \bmod R$
$N^{-1}$ : 可以使用扩展欧几里得求解
这里当 $N$ $R$ 选定后，$r2$ $N'$ 也可以确定.

- rl $= R \bmod N$ 的计算:
  $REDC\left(r2, R, N, N'\right) \equiv \left(R^2\right) R' \equiv R \bmod N$
- $a \bmod N$ 的计算:
  $\mathrm{REDC}\left(a^{*}r1, \mathrm{R}, \mathrm{N}, \mathrm{N}'\right) \equiv (aR)\mathrm{R}' \equiv \mathrm{a} \bmod \mathrm{N}$
- $aR \bmod N$ 的计算:
  $\mathrm{REDC}\left(a^{*}r2, R, N, N'\right) \equiv \left(aR^2\right) R' \equiv aR \bmod N$
- $ab \bmod N$ 计算:
  $aR, bR \rightarrow abR \rightarrow ab$
- $a/b \bmod N$ 转化为 $ab^{-1} \bmod N$

### 2.3.1 C 代码实现

在这里，由于涉及到大数运算，可以使用 C++ 中的大数运算 NTL 库实现.NTL 库的安装参考放在了参考文献中.NTL 库中的 ZZ 数据类型可以支持大数运算. 在实现过程中，为了使 R 的运算简便，这里将 R 取为 2 的幂次. 对于求逆的过程，可以使用扩展欧几里得算法进行实现，具体实现由下面的代码注释给出：

```
class Montgomery
{
private:

        uint RR;
        ZZ R;
        ZZ N;
        ZZ N_inv;//N_inv * N = -1 mod R
        ZZ Z;//零
        ZZ L;//R的掩码
        ZZ m;

public:
        void init(ZZ& N);//初始化模型
        void Map(ZZ& src, ZZ& dst);//将a和b计算aR和bR
        void InvMap(ZZ& src, ZZ& dst);//映射回源数据
        void Mul(ZZ& a, ZZ& b, ZZ& ab);//乘法计算
};


void Montgomery::init(ZZ& N)
{
        if ((N & ZZ(1)) != 1) //N应当是奇数，因为R取的2的幂次
                return;
        this->N = N;
        RR = NumBits(N);
        R = 1;
        R <<= RR; //R = 2^RR
        Z = 0; //零
```

```
        L = R - 1; //掩码
        ZZ d, s, t;
        XGCD(d, s, t, N, R); //d=1, s*N = 1 mod R
        N_inv = R - s; //N_inv * N = -1 mod R
}

// a*R mod N
void Montgomery::Map(ZZ& a, ZZ& bar_a)
{
        bar_a = a << RR; //a*R
        bar_a %= N; //a*R mod N
}


// a_bar * R_inv mod N
void Montgomery::InvMap(ZZ& T_bar, ZZ& T)
{
        T = T_bar; //T
        m = T * N_inv; //m = T * (-N_inv)
        m &= L; //m mod R 直接使用掩码实现，提升运算速度
        T += m * N; //T + m*N
        T >>= RR; //(T + m*N) * R_inv
        if (T > N)
                T -= N;
}



//Montgomery乘法
void Montgomery::Mul(ZZ& a, ZZ& b, ZZ& ab)
{
        ab = a * b; //T = a*R * b*R
        m = ab * N_inv; //m = T * (-N_inv)
        m &= L; //m mod R
        ab += m * N; //(T + m*N)
```

```
        ab >>= RR; //(T + m*N) * R_inv
        if (ab > N)
                ab -= N;
}
```

### 2.3.2 python 代码实现

在 python 的计算中，由于 python 可以直接支持大数的运算，因此这里可以不用考虑数据类型. 在实现过程中，首先实现了扩展欧几里得求逆算法，然后按照上面的原理实现模乘和模约简运算：

```
 #扩展欧几里得定理用于求逆元


def inverse_mod(a, p):
    old_s, s = 1, 0
    old_t, t = 0, 1
    old_r, r = a, p
    if p == 0:
        return 0
    else:
        while r != 0:
            q = old_r // r
            old_r, r = r, old_r - q * r
            old_s, s = s, old_s - q * s
            old_t, t = t, old_t - q * t
    return (old_s%p+p)%p



"""
Montgomery reduction
T:integer in T mod N
N:mod N
R:R,2^k
```

```
_N:INVERSE OF N MOD R
OUTPUT = T*R^(-1) MOD N
"""

def redc(t:int,n:int,r:int,_n:int)->int:
    assert 0 <= t <= r*n-1
    rbit_len = r.bit_length()-1
    r_mask = r-1
    m=((t&r_mask)*_n)&r_mask #M <- ((T MOD R)N')MOD R
    t = (t+m*n)>>rbit_len #T<-(T+MN)/R
    result = t if (t<n) else (t-n)
    assert 0<=result<n
    return result


#AMOD RETURN A MOD P

def amod(a:int,p:int)->int:
    a_p_len = max(a.bit_length(),p.bit_length())
    r_len = ((a_p_len+7)//8)*8 #对齐到8的整数倍
    r = 2**r_len #2^K
    r1 = r%p
    _n=inverse_mod(p,r)
    _n = -_n%r
    return redc(a*r1,p,r,_n)



def mult_mod(a:int,b:int,p:int)->int:
    a_p_len = max(a.bit_length(),b.bit_length(),p.bit_length())
    r_len = (a_p_len+7)//8*8
    r = 2**r_len #2^K
    r1 = r%p #R1 = R MOD P
    r2 = r1*r1%p #R2 = R*R MOD P
    _n = inverse_mod(p,r)
```

```
    _n = -_n%r

    ar = redc(a*r2,p,r,_n)
    br = redc(b*r2,p,r,_n)
    abr = redc(ar*br,p,r,_n)
    return redc(abr,p,r,_n)
```

# 3 Barrett reduction 实现

## 3.1 算法原理

这里 Barrett reduction 的目标是实现快速的模数运算，思想是将乘法和减法运算代替除法运算.

**基本思想:**

首先计算一个浮点数 $s = 1/n$. 然后有

$$a \bmod n = a - \lfloor as \rfloor n$$

$\lfloor as \rfloor$ 是对 as 乘积的下取整函数，当 $s$ 具有足够的精度时，最后计算出来的结果将是准确的.

Barrett reduction 将 $1/n$ 用 $m/2^k$ 的值进行粗略的估计，因为计算 $2^k$ 可以直接用右移位操作计算. 那么给出 $2^k$,m 的计算就可以由下面的式子给出:

$$\frac{m}{2^k} = \frac{1}{n} \Leftrightarrow m = \frac{2^k}{n}$$

因此，$m = \lfloor 2^k/n \rfloor$ 是一种更加通用的写法. 但是因为 $m/2^k \leqslant 1/n$，计算出的 q 值 (as 下取整) 可能会变得过小，这样 a 无法减足够的 n 的倍数，导致最终 a 的值不在 n 之内.

因为 $m/2^k$ 只是一个大概估计，a 的有效范围需要被考虑. 错误大概是 $e = \frac{1}{n} - \frac{m}{2^k}$，因此错误在 q 的计算中会被放大为 ae. 只要 $ae < 1$ 那么就认为模约减是有效的因此 $a < 1/e$. 通过选择更大的 k，a 的选择范围就能够增加，但是要注意不要溢出.

## 3.2 C 代码实现

Barrett reduction 的实现比较简单，因为涉及到大数的运算，这里还是使用 NTL 库进行完成. 具体细节在代码注释中给出:

```
    void Barrett(ZZ& n, ZZ& a, ZZ& res, uint& acc)
{
        ZZ q;
        uint k = NumBits(n) + acc;
        //由于m的存在，那么k一定要比n的bit数多
        //k的位数加上准确度常量acc，来控制e的大小，acc越大，e越小，准确度越高
        ZZ m = power2_ZZ(k)/n;
        //m = floor(2^k / n)
        q = (m * a) >> k; //q = floor(a/n) = a * m/2^k
        res = a - q * n; //a - q*n
        if (res >= n)
                res -= n;
        //如果出现a约化精度不够的问题，就继续减n
}
```

## 3.3 python 代码实现

python 代码和 c 代码实现的思路相同，在实现过程中，这里使用了移位运算代替 $2^k$, 这里值得注意的一点是 $2^k$ 转化为移位运算是 2 向左移 $k-1$ 位.

```
    acc = 1024
def Barrett(n,a):
    k = n.bit_length()+acc
    #由于M存在，那么K一定要比N的BIT数多
    #ACC用来控制E的大小，ACC越大，E越小，准确度也就越高
    m = (2<<(k-1))//n
    #M=FLOOR(2^K/N)
    q = (m*a)>>k
    #PRINT(Q)
    #Q = FLOOR(A/N) = A * M/2^K
```

```python
    res = a-q*n
    if(res>=n):
        res -=n
    return res
```

# 4 结果验证

首先我们使用老师给出的网站生成了 3 个 2048bit 的素数 a,b,N 来用于后续的实验.


图 1: 2048bit numbers

下面运行 C 语言程序，程序将分别使用 Montgomery multiplication 和 barrett reduction 计算 $a \cdot b \bmod N$ 和 $a \bmod N$ 的值，结果如下:

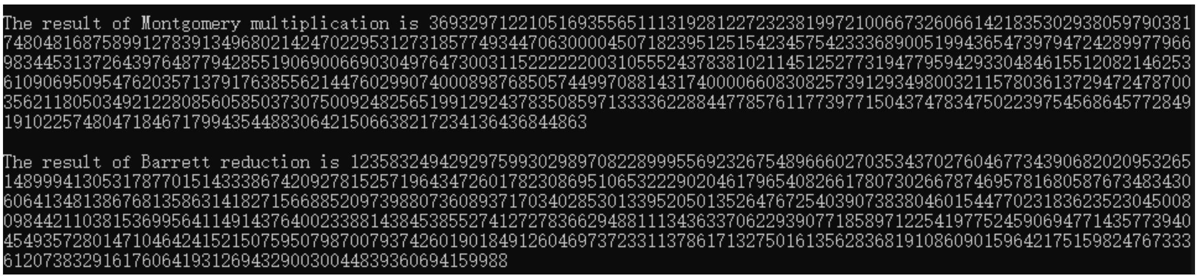


图 2: C Result

下面运行 python 程序，程序将分别使用 Montgomery multiplication 和 barrett reduction 计算 $a \cdot b \bmod N$ 和 $a \bmod N$ 的值，结果如下:

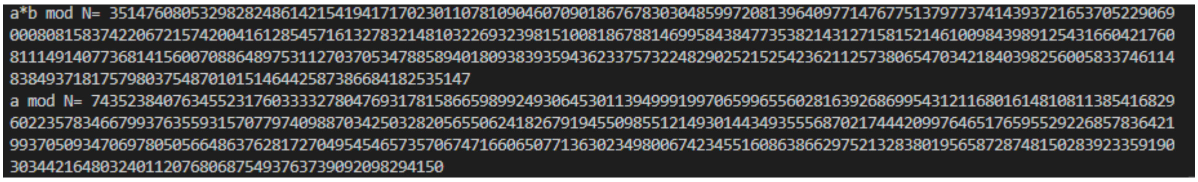


图 3: python Result

接下来我使用 sagemath 软件直接计算结果进行验证，可以发现验证一致.

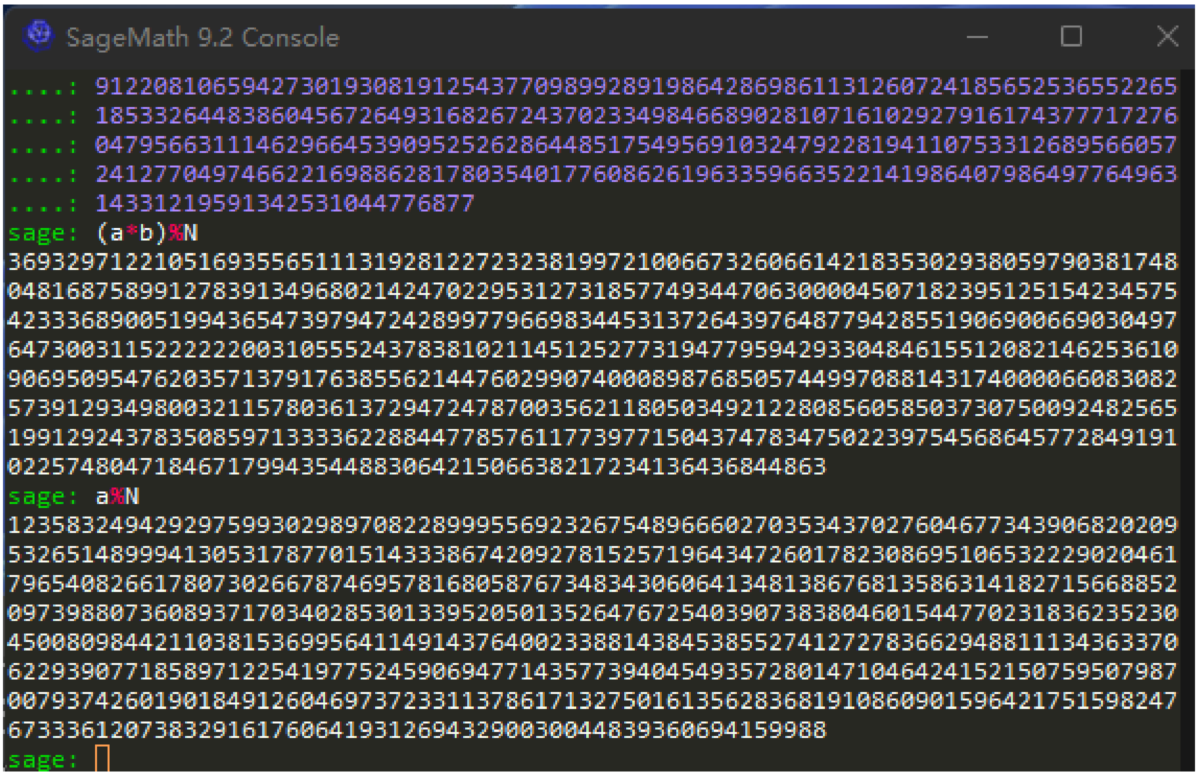


图 4: Verify

# 5 分工和实验心得

实验实现了 Montgomery multiplication 和 Barrett reduction，了解到了对大数模乘和模数运算的优化方法，进一步丰富了相关的密码学知识. 在实验的具体实现过程中，由于涉及到大数运算，学习了如何安装编译 NTL 库静态链接文件, 并调用相关库函数, 在实践中, 还学习了通过掩模运算和移位运算加快算法速度, 收获很大.

# 参考文献


[1] https://en.wikipedia.org/wiki/Barrett_reduction

[2] https://en.wikipedia.org/wiki/Montgomery_modular_multiplication

[3] https://asecuritysite.com/encryption/random3

[4] https://www.cnblogs.com/luoqiang111/p/qq1449904853.html

[5] https://zhuanlan.zhihu.com/p/66102259