



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 $\text{mod } N$ 的运算转化为 $\text{mod } R$ 的运算
- 为什么要把 $\text{mod } N$ 转化为 $\text{mod } R$
- 因为选择恰当的 R 能更容易的进行计算, 例如 $\text{mod } 251 \rightarrow \text{mod } 256$, 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:

$$aR + bR \equiv (a + b)R \pmod{N}, aR - bR \equiv (a - b)R \pmod{N}.$$

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

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

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地理解算法。

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

- 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.

- $r1 = R \bmod N$ 的计算:

$$\text{REDC}(r2, R, N, N') \equiv (R^2) R' \equiv R \bmod N$$

- $a \bmod N$ 的计算:

$$\text{REDC}(a * r1, R, N, N') \equiv (aR) R' \equiv a \bmod N$$

- $aR \bmod N$ 的计算:

$$\text{REDC}(a * r2, R, N, N') \equiv (aR^2) R' \equiv aR \bmod N$$

- $ab \bmod N$ 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：

```

1  class Montgomery
2  {
3  private:
4
5      uint RR;
6      ZZ R;
7      ZZ N;
8      ZZ N_inv; //  $N_{inv} * N = -1 \bmod R$ 
9      ZZ Z; // 零
10     ZZ L; // R 的掩码
11     ZZ m;
12
13 public:
14     void init(ZZ& N); // 初始化模型
15     void Map(ZZ& src, ZZ& dst); // 将 a 和 b 计算  $aR$  和  $bR$ 
16     void InvMap(ZZ& src, ZZ& dst); // 映射回源数据
17     void Mul(ZZ& a, ZZ& b, ZZ& ab); // 乘法计算
18 };
19
20
21 void Montgomery::init(ZZ& N)
22 {
23     if ((N & ZZ(1)) != 1) // N 应当是奇数，因为 R 取的 2 的幂次
24         return;
25     this->N = N;
26     RR = NumBits(N);
27     R = 1;
28     R <<= RR; //  $R = 2^{RR}$ 
29     Z = 0; // 零

```

```

30     L = R - 1; //掩码
31     ZZ d, s, t;
32     XGCD(d, s, t, N, R); //d=1, s*N = 1 mod R
33     N_inv = R - s; //N_inv * N = -1 mod R
34 }
35
36 // a*R mod N
37 void Montgomery::Map(ZZ& a, ZZ& bar_a)
38 {
39     bar_a = a << RR; //a*R
40     bar_a %= N; //a*R mod N
41 }
42
43
44 // a_bar * R_inv mod N
45 void Montgomery::InvMap(ZZ& T_bar, ZZ& T)
46 {
47     T = T_bar; //T
48     m = T * N_inv; //m = T * (-N_inv)
49     m &= L; //m mod R 直接使用掩码实现, 提升运算速度
50     T += m * N; //T + m*N
51     T >>= RR; //(T + m*N) * R_inv
52     if (T > N)
53         T -= N;
54 }
55
56
57
58 //Montgomery乘法
59 void Montgomery::Mul(ZZ& a, ZZ& b, ZZ& ab)
60 {
61     ab = a * b; //T = a*R * b*R
62     m = ab * N_inv; //m = T * (-N_inv)
63     m &= L; //m mod R
64     ab += m * N; //(T + m*N)

```

```

65         ab >>= RR; //(T + m*N) * R_inv
66         if (ab > N)
67             ab -= N;
68     }

```

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:

```

1  #扩展欧几里得定理用于求逆元
2
3
4  def inverse_mod(a, p):
5      old_s, s = 1, 0
6      old_t, t = 0, 1
7      old_r, r = a, p
8      if p == 0:
9          return 0
10     else:
11         while r != 0:
12             q = old_r // r
13             old_r, r = r, old_r - q * r
14             old_s, s = s, old_s - q * s
15             old_t, t = t, old_t - q * t
16         return (old_s%p+p)%p
17
18
19
20  """
21  MONTGOMERY REDUCTION
22  T:INTEGER IN T MOD N
23  N:MOD N
24  R:R,2^K

```



```

25  _N:INVERSE OF N MOD R
26  OUTPUT = T*R(-1) MOD N
27  ""
28
29  def redc(t:int,n:int,r:int,_n:int)->int:
30      assert 0 <= t <= r*n-1
31      rbit_len = r.bit_length()-1
32      r_mask = r-1
33      m=((t&r_mask)*_n)&r_mask #M <- ((T MOD R)N')MOD R
34      t = (t+m*n)>>rbit_len #T<-(T+MN)/R
35      result = t if (t<n) else (t-n)
36      assert 0<=result<n
37      return result
38
39
40  #AMOD RETURN A MOD P
41
42  def amod(a:int,p:int)->int:
43      a_p_len = max(a.bit_length(),p.bit_length())
44      r_len = ((a_p_len+7)//8)*8 #对齐到8的整数倍
45      r = 2**r_len #2^k
46      r1 = r%p
47      _n=inverse_mod(p,r)
48      _n = -_n%r
49      return redc(a*r1,p,r,_n)
50
51
52
53  def mult_mod(a:int,b:int,p:int)->int:
54      a_p_len = max(a.bit_length(),b.bit_length(),p.bit_length())
55      r_len = (a_p_len+7)//8*8
56      r = 2**r_len #2^k
57      r1 = r%p #R1 = R MOD P
58      r2 = r1*r1%p #R2 = R*R MOD P
59      _n = inverse_mod(p,r)

```

```

60     _n = -_n%r
61
62     ar = redc(a*r2,p,r,_n)
63     br = redc(b*r2,p,r,_n)
64     abr = redc(ar*br,p,r,_n)
65     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 $m/2^k \leq 1/n$ ，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：

```

1  void Barrett(ZZ& n, ZZ& a, ZZ& res, uint& acc)
2  {
3      ZZ q;
4      uint k = NumBits(n) + acc;
5      //由于m的存在，那么k一定要比n的bit数多
6      //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
7      ZZ m = power2_ZZ(k)/n;
8      //m = floor(2^k / n)
9      q = (m * a) >> k; //q = floor(a/n) = a * m/2^k
10     res = a - q * n; //a - q*n
11     if (res >= n)
12         res -= n;
13     //如果出现a约化精度不够的问题，就继续减n
14 }

```

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。

```

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

```

```

11     res = a-q*n
12     if(res>=n):
13         res -=n
14     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.

```

a=2432389404436587303124545809487419026069712709234629708795016676705550206675610367498513521901439846187314625799372887
658099485116303530894041801218877201151608395083266601892287235887223137003956905055086852887151063182445395515010452576
609096094730229129516840038437927285875844618599042701781486878441824794395595117322009205815778983868494009365343343826
335343529061268579920822548985478380743234543227901586150836540638923625539228230487813228146018653336862062253202217448
68899070155593593126770348298799137801513643570060005151636423573043203526096646101119902093062747937066329911220061490
8410209484873267496

b=2744874330004467664436501031072860052892232626783070764481506519714003711418832356014681021615637455490266062390125512
763840391522658016655487858422992254218078181656929901442117128336986583292695608470664909952674850965822775963997672420
129185545195183469409374943873444609764195664162368731553802704714800502078935284499037583037246530239528766348011441962
737054686662892618337433023300499153115473011327148447322141675774359884482981910382257857308588592829197226861832723850
536372298546068638890130135184279651154313950166288198333100298907349929003132732753897016907629720735964356322908366962
0115218642415688357

N=5772015387518224259553890253146048672866112900670067513151616001646941919018520680429996554900336320775748706152996695
949867282879890676789527329270559945623905538572959850278892422749383897089745344266859480614507311264079523002069342269
392496739807170589499515024769483334564273384579471656769122081065942730193081912543770989928919864286986113126072418565
253655226518533264483860456726493168267243702334984668902810716102927916174377717276047956631114629664539095252628644851
754956910324792281941107533126895660572412770497466221698862817803540177608626196335966352214198640798649776496314331219
591342531044776877

```

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:

```

The result of Montgomery multiplication is 36932971221051693556511131928122723238199721006673260661421835302938059790381
748048168758991278391349680214247022953127318577493447063000045071823951251542345754233368900519943654739794724289977966
983445313726439764877942855190690066903049764730031152222220031055524378381021145125277319477959429330484615512082146253
610906950954762035713791763855621447602990740008987685057449970881431740000660830825739129349800321157803613729472478700
356211805034921228085605850373075009248256519912924378350859713333622884477857611773977150437478347502239754568645772849
19102257480471846717994354488306421506638217234136436844863

The result of Barrett reduction is 1235832494292975993029897082289995569232675489666027035343702760467734390682020953265
148999413053178770151433386742092781525719643472601782308695106532229020461796540826617807302667874695781680587673483430
606413481386768135863141827156688520973988073608937170340285301339520501352647672540390738380460154477023183623523045008
098442110381536995641149143764002338814384538552741272783662948811134363370622939077185897122541977524590694771435773940
454935728014710464241521507595079870079374260190184912604697372331137861713275016135628368191086090159642175159824767333
612073832916176064193126943290030044839360694159988

```

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：

```
a*b mod N= 3514760805329828248614215419417170230110781090460709018676783030485997208139640977147677513797737414393721653705229069
000808158374220672157420041612854571613278321481032269323981510081867881469958438477353821431271581521461009843989125431660421760
811149140773681415600708864897531127037053478858940180938393594362337573224829025215254236211257380654703421840398256005833746114
838493718175798037548701015146442587386684182535147
a mod N= 74352384076345523176033327804769317815866598992493064530113949991997065996556028163926869954312116801614810811385416829
602235783466799376355931570779740988703425032820565506241826791945509855121493014434935556870217444209976465176595529226857836421
993705093470697805056648637628172704954546573570674716606507713630234980067423455160863866297521328380195658728748150283923359190
303442164803240112076806875493763739092098294150
```

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。

```
SageMath 9.2 Console
.....: 91220810659427301930819125437709899289198642869861131260724185652536552265
.....: 18533264483860456726493168267243702334984668902810716102927916174377717276
.....: 04795663111462966453909525262864485175495691032479228194110753312689566057
.....: 24127704974662216988628178035401776086261963359663522141986407986497764963
.....: 14331219591342531044776877
sage: (a*b)%N
36932971221051693556511131928122723238199721006673260661421835302938059790381748
04816875899127839134968021424702295312731857749344706300004507182395125154234575
42333689005199436547397947242899779669834453137264397648779428551906900669030497
6473003115222220031055524378381021145125277319477959429330484615512082146253610
90695095476203571379176385562144760299074000898768505744997088143174000066083082
57391293498003211578036137294724787003562118050349212280856058503730750092482565
19912924378350859713333622884477857611773977150437478347502239754568645772849191
02257480471846717994354488306421506638217234136436844863
sage: a%N
12358324942929759930298970822899955692326754896660270353437027604677343906820209
53265148999413053178770151433386742092781525719643472601782308695106532229020461
79654082661780730266787469578168058767348343060641348138676813586314182715668852
09739880736089371703402853013395205013526476725403907383804601544770231836235230
45008098442110381536995641149143764002338814384538552741272783662948811134363370
62293907718589712254197752459069477143577394045493572801471046424152150759507987
00793742601901849126046973723311378617132750161356283681910860901596421751598247
67333612073832916176064193126943290030044839360694159988
sage: □
```

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>