快速乘算法的实现

密码工程

2023 年 11 月 29 日

李 昕 202100460065

21 级密码 2 班

目录

1 实验要求

问题一 实现 Karatsuba 算法。

问题二 实现 Barrett reduction 和 Barrett 模乘算法。

问题三 实现 Montgomery 系列算法（约简，模乘，模幂）。

2 符号标记说明

符号	解释
$\oplus$	异或
$\lceil x \rceil$	向上取整
$\lfloor x \rfloor$	向下取整

3 实验准备

3.1 Karatsuba 算法的算法过程

Karatsuba 乘法是一种分治算法，使用递归的方式将两个大整数分别分解成较小的整数相乘的形式，然后再通过组合这些小的积得到原始积的算法。该算法的时间复杂度为 $O(n^{log_2(3)})$，比传统的手动竖式乘法更快。

Karatsuba 乘法的关键思想是递归地计算这些较小整数的乘积，并利用中间结果来计算最终的乘积。通过适当的组合和加法运算，可以在较少的乘法步骤中完成整个乘法运算，从而提高了计算效率。假设我们要计算两个大整数 x 和 y 的乘积，其中 x 和 y 都可以表示为：

$$x = a * B^{(n/2)} + b$$

$$y = c * B^{(n/2)} + d$$

计算 ac、bd 和 (a+b)*(c+d), 使用递归计算 (a+b)*(c+d), 则可将其变成 3 个乘法运算，以及六次加法，复杂度得到了降低。

3.2 Barrett 算法的算法过程 [1]

3.2.1 Barrett Reduction

Barrett 算法利用移位代替除法。在一般的计算机取模运算中，使用 $a-\lfloor\frac{a}{p}\rfloor*p$ 来计算 a 模 p，在这个计算中存在除法过程，计算开销较大，我们希望能用乘法替换除法，若把 $q=\lfloor\frac{a}{p}\rfloor$ 写作 $q=\lfloor a*p^{-1}\rfloor$，此处 p^{-1}，是用浮点数表示的 p 的倒数。但浮点数计算在计算机中仍然较。

在上述分析的基础上，指定一个整数 k ，以及一个整数 m，使得 $\frac{m}{2^k}\approx\frac{1}{p}$，则 q 约等于 $\frac{a*m}{2^k}$，此时除法运算被拆成了一次乘法和一次位移，从而规避了除法运算。

由于 $\frac{m}{2^k}\approx\frac{1}{p}$，故只需要预处理这个除法的结果 m，就可以在之后的重复计算中节省时间，为了防止算出的商超过实际的商，一般取 $m=\lfloor\frac{2^k}{p}\rfloor$。指定取 $k>=\lfloor 2log_2 p\rfloor$，使得 $2^k\approx p^2$.

此时需要证明，k 取上值时，$0<=a-pq<p$。注意到 $q=\frac{am}{2^k}$，因此 $pq=\frac{apm}{2^k}$，$a-pq=\frac{a}{2^k}(2^k-pm)$。由于 $m=\lfloor\frac{2^k}{p}\rfloor$，因此 $\frac{2^k}{p}-1<=m<=\frac{2^k}{p}$. 如果 $k>=\lfloor 2log_2 p\rfloor$，则 $2^k>=p^2$。由于 a 是 $modp$ 下两个数的乘积，不会达到 p^2 ，因此 $\frac{a}{2^k}$ 在 0，1 之间，于是 $a-pq=\frac{a}{2^k}(2^k-pm)$ 在 [0,p) 之间。

由上述推理，得到算法一：

Algorithm 1: Barrett_Mod(X,M)

input : 模数 $M=(m_{n-1},...,m_1,m_0)_r$，其中 $m_{n-1}\neq 0$ 且 $r=2^\delta$，整数 $X=(x_{2n-1},...,x_1,x_0)_r$，其中 $0\leq X<r^{2n}$ 且 $\mu=\lfloor r^{n+\alpha/M}\alpha\rfloor$，$\alpha,\beta\in N$

output $Z=X \mod M$

:

1 $Z\leftarrow 0,\quad k'\leftarrow\lfloor\lfloor X/r^{n+\beta}\rfloor(\mu/r^{\alpha-\beta})\rfloor$

2 $Z\leftarrow X-k'M$

3 **while** $Z\geq M$ **do**

4 $\quad Z\leftarrow Z-M$

5 **end**

6 **return** Z

3.2.2 Barrett 模乘

利用 3.1.1 中的分析，当给定整数 $X=\sum_{i=1}^{n}x_i r^i$ 和 $Y=\sum_{i=1}^{n}y_i r^i$，其中 $0=X,Y<M$，则 X,Y 在 M 下的模乘余数可以表示为：$Z\equiv XY modM\equiv\sum_{i=1}^{n}y_i r^i X modM$，因此模乘算法可以通过取模算法展开，如算法二：

Algorithm 2: Barrett_MultiMod(x,y,m)

input : 模数 $M = (m_{n-1}, \cdots, m_1, m_0)_r$，其中 $m_{n-1} \neq 0$ 且 $r = 2^s$，整数 $X = (x_{n-1}, \cdots, x_1, x_0)_r$ 和整数 $Y = (y_{n-1}, \cdots, y_1, y_0)_r$，其中 $0 \leq X, Y < M, \mu = \lfloor r^{n+\alpha}/M \rfloor$ 且 $\alpha, \beta \in N$

output $Z = XY \bmod M$

:

$Z \leftarrow 0$

for $i \leftarrow 1$ *to* n **do**

  $Z \leftarrow rZ + y_i X$

  $k' \leftarrow \lfloor LZ/r^{n+\beta} \rfloor \left(\mu/r^{\alpha-\beta}\right) \rfloor$

  $Z \leftarrow Z - k'M$

end

while $Z \geq M$ **do**

  $Z \leftarrow Z - M$

end

return Z

3.3 Montgomery multiplication 的算法过程 [2]

3.3.1 Montgomery 约简

Montgomery reduction 即 REDC，是蒙哥马利模乘和模幂的关键部分，REDC 的核心思想是避免做除法 $\lfloor a/N \rfloor$，但是仍能求得模约减的结果。作为代价，REDC 算的模约减结果包含了一个' 尾巴' R^{-1},。这也是为什么蒙哥马利模乘算法当中反复强调要在蒙哥马利形式下进行的根本原因。REDC 用于将一个在范围 [0, RN - 1] 内的数 X 转换为在范围 [0, N - 1] 内的数 S，满足 $S \equiv XR^{-}1 mod N$

算法的主要思想即为计算 $XR^{-1} \pmod N$,，这相当于 $\frac{X}{R} \pmod N$，要避免除法运算，从 3.2 对 R 的定义中我们知道，$R = 2^k$，所以 $\frac{X}{R} = X >> k$，同时，为了避免右移 k 位可能会抹掉 X 的低位中的一些 1，X 需要是 R 的整数倍，此时 X/R=$X >> K$，故需要找到一个 m，使得 $X + mN$ 是 R 的倍数，又由 gcd(R,N)=1，根据扩展欧几里得算法，有 $RR^{'} - NN^{'} = 1$ 且 $0 < N^{'} < R, 0 < R^{'} < N < R$,

$$X + mN \equiv 0 \pmod R$$

$$XN' + mNN' \equiv 0 \pmod R$$

$$XN' + m(RR' - 1) \equiv 0 \pmod R$$

$$XN' \equiv m \pmod{R}$$

通过上面方程组即可求得 m，当我们已经计算出来了蒙哥马利模式下的 $a', b', R\ X$，就可以执行主体部分，需要计算 $N' = -N^{-1}(modR), m = XN'(modR)$，再计算 $y = \frac{X+mN}{R}$，即将 X+mN 右移 k 位，此时若 y>N，y=y-N，返回 y 即可。

3.3.2 Montgomery 模乘

Montgomery 乘法的数学表达是:$A * B * R^{-1} modM$，其中，A、B 是与 M 同位长的大数，$R = 2^{bitlen}$(bitlen 指 M 位长)，R^{-1} 是指 R 相对于 M 的模逆，即 R^{-1} 是满足如下条件的数：$R * R^{-1} modM = 1$；这个条件成立的充要条件是 R 与 M 互素，这一点只需要 M 为奇数即可，所以 Montgomery 乘法通常适用于对奇数求模。

现在考虑如何计算两个整数 a 和 b 的模乘，即 $ab \bmod N$。这里利用蒙哥马利模乘达成这个计算目标。蒙哥马利模乘可以分三步进行计算:

1. 将输入 $a \cdot R^2, b \cdot R^2$ 转成蒙哥马利形式，即 $aR = REDC(aR^2), bR = REDC(bR^2)$

2. 做一次标准乘法得 $abR^2 = aR \cdot bR$

3. 最后做一次 REDC 得 $abR = REDC(abR^2)$ 特别的，上面三个步骤返回的是蒙哥马利形式的 ab，即 abR。若需要转换成正常形式的 ab，需要再做一次 REDC 得 ab=REDC(abR)

由上述分析，可列出 Montgomery 模乘伪代码如下:

Algorithm 3: Mon_MultiMod(a,b,N)

```
a' = aR mod N
b' = bR mod N
X = a'b'
X_1 ← Mon_reduction(X, R, N)
y ← Mon_reduction(X_1, R, N)
return y
```

3.3.3 Montgomery 模乘的复杂度分析

蒙哥马利乘法需要进行两次模运算和一次逆元计算。逆元计算可以使用扩展欧几里得算法实现，时间复杂度为 O (logn)。而模运算可以使用移位和加减操作实现，时间复杂度为 O (logn)。因此，蒙哥马利乘法的时间复杂度为 O (logn)。

在蒙哥马利算法的流程中，包含了两次模 N 运算（a' a (mod N), b' b (mod N)），与普通实现相比似乎并没有减少预算开销。但实际上，第一步可以看作是蒙哥马利算法

的预先计算，也就是在实际运行之前就把这个步骤完成。在实现中，可以提前进行这些预计算，然后在后续运算中使用这些预先计算的结果，从而加快整体运行速度，特别的，当需要进行大量的模乘运算时，可以通过并行运算进行预计算，这将大大节省运行时间。

4 实验过程

4.1 编写 Karatsuba 乘法代码

按照老师课堂上展示的伪代码，结合原理部分的分析，得到以下代码：

```
def Karatsuba(x, y):
    if len(str(x))==1 or len(str(y))==1:
        return x*y
    if x<=10 or y<=10:
        return x*y
    n = max(len(str(x)), len(str(y)))
    k = n//2
    x1 = x // 10**k
    x0 = x % 10**k
    y1 = x // 10**k
    y0 = x % 10**k
    z0 = Karatsuba(x0, y0)
    z2 = Karatsuba(x1, y1)
    z1 = Karatsuba((x1+x0), (y1+y0)) - z2 - z0
    xy = z2*10**(k*2) + z1*10**(k) + z0
    return xy
```

代码主要思想即为使用递归调用 Karatsuba 函数计算 z0、z1 和 z2。其中，z0 对应于 x0 和 y0 的乘积，z2 对应于 x1 和 y1 的乘积，z1 对应于 (x1+x0) 和 (y1+y0) 的乘积减去 z2 和 z0 的结果。z2、z1 和 z0 组合成最终的结果 xy。

4.2 编写 Barrett Reduction 代码

4.2.1 编写 Barrett Reduction 的 python 代码

定义函数 def barrett_reduction(dividend, modulus, divisor, mu)，其中，前两个数分别为被模数和模数，divisor, mu 为巴雷特约简中预计算的的中间值，这里将其设置为

函数传入变量，方便快速处理足够多的比特数相近的数据，同时，通过调整这两个常量，可以优化算法的性能。

具体来说，定义 divisor 是一个固定的值，它的取值为 2 的模数位数次方，即 2^{2048}。这个值用于对模数的位数进行标准化，使得每个位的权重都相同。通过将被除数 num 与 divisor 进行位移运算，可以快速地将 num 分割成多个部分。mu 是 divisor 的倒数对 modulus 的近似值，它可以通过一些数学方法计算得到，这个值被用于进一步优化计算，减少乘法操作的次数。通过将 q1 乘以 mu 进行位移运算，可以得到 q2 的近似值，进而减少乘法和除法的运算次数。

在函数中，首先计算巴雷特约简算法中的参数 k 值，k 为模数的位数的一半向下取整，即 $k = \lfloor log_2(modulus)/2 \rfloor$;

第二步，将被除数 dividend 分成两部分，高位部分为 q1，低位部分为 r1。将 q1 右移 $(k-1)*64$ 得到 q';

第三步，计算 q2，q2 为 $q1'$ 乘以 mu 得到的结果右移 $(k+1)*64$ 的值;

最后，计算 r2，r2 为 dividend 减去 q2 与 divisor 相乘的结果，若 r2 大于或等于 modulus，则 r2 减去 modulus，直至 r2 小于 modulus。

具体实现如下:

```
def barrett_reduction(dividend, modulus, divisor, mu):
    k = modulus.bit_length() // 2   #第一步
    q1 = dividend >> ((k - 1) * 64)    #第二步
    r1 = dividend % (2 ** ((k - 1) * 64))
    q2 = (q1 * mu) >> ((k + 1) * 64)    #第三步
    r2 = dividend - q2 * divisor   #第四步
    while r2 >= modulus:
        r2 -= modulus
    return r2
```

4.2.2 编写 Barrett Reduction 的测试部分

接下来，编写对应的测试代码，随机生成两个 2048bit 的数字作为模数与被模数，分别计算普通取模和巴雷特约简的结果，代码如下:

```
if __name__ == "__main__":
    num_bits = 2048
    num = random.getrandbits(num_bits)
```

```
    modulus = random.getrandbits(num_bits)
    divisor = 2 ** 2048
    mu = (2 ** (2 * (modulus.bit_length() // 2))) // divisor
    start_time = time.time()
    a = barrett_reduction(num, modulus, divisor, mu)
    end_time = time.time()
    print(f"Barrett reduction: {a}")
    start_time_mod = time.time()
    b = num % modulus
    end_time_mod = time.time()
    print(f"Modulus operator:  {b}")
    time_barrett = end_time - start_time
    time_modulus = end_time_mod - start_time_mod
    if a == b:
        print("Results are the same")
    else:
        print("Results are different")
    #print(f"Computation time of Barrett reduction: {time_barrett:.6f} seconds")
    #print(f"Computation time of modulus operator:  {time_modulus:.6f} seconds")
```

如果两种方法计算结果相同，输出 Results are the same，证明成功编写 Barrett Reduction 的代码，结果如下：

```
========== RESTART: F:\Study\大三上\密码工程实验\实验五\Barrett.py ==========
Barrett reduction: 25368548060931702804740450166284109485026798868092153087706776498439814062476806635724661664680115795937208620618958027354473350335835072484302366624399806843620534756861099071507260908156193074917672958532087897163480059944356653606587332833953576160864071435960453895319905860201762758379945783708197447535630217310147724374582865308952603983157407492909534822106255459886688513145276835258634734678780141256955098242267961722561391136117557985503039190741996097637964500264273244996039882098463603115129060774248043681292599835509058857922257455818260753031204618321638489611438820413520457969129863550152942493855>>> Barrett reduction: 25368548060931702804740450166284109485026798868092153087706776498439814062476806635724661664680115795937208620618958027354473350335835072484302366624399806843620534756861099071507260908156193074917672958532087897163480059944356653606587332833953576160864071435960453895319905860201762758379945783708197447535630217310147724374582865308952603983157407492909534822106255459886688513145276835258634734678780141256955098242267961722561391136117557985503039190741996097637964500264273244996039882098463603115129060774248043681292599835509058857922257455818260753031204618321638489611438820413520457969129863550152942493855Modulus operator:  25368548060931702804740450166284109485026798868092153087706776498439814062476806635724661664680115795937208620618958027354473350335835072484302366624399806843620534756861099071507260908156193074917672958532087897163480059944356653606587332833953576160864071435960453895319905860201762758379945783708197447535630217310147724374582865308952603983157407492909534822106255459886688513145276835258634734678780141256955098242267961722561391136117557985503039190741996097637964500264273244996039882098463603115129060774248043681292599835509058857922257455818260753031204618321638489611438820413520457969129863550152942493855Results are the same>>>
```

Figure 1: 测试结果

4.3 Barrett 模乘的 python 实现

根据 3.1.2 分析的原理，在巴雷特约简代码的基础上进行改动，先将被乘数 a 和乘数 b 进行巴雷特约简预处理，然后按照巴雷特模乘的公式 r = r1 * r2 - t * n 进行计算，得到 a * b (mod n) 的结果，即 r。最后对 r 进行调整，使其保持在合适的范围内。

```
import random
def barrett_mod_mul(a, b, n, mu):
    # 计算模数位数的一半
    k = (n.bit_length() + 1) // 2
    q1 = (a >> (k - 1) * 64)# 将 a 和 b 进行巴雷特约简预处理
    r1 = a % (1 << (k - 1) * 64)
    q2 = (b >> (k - 1) * 64)
    r2 = b % (1 << (k - 1) * 64)
    t = q1 * q2     # 计算 t = q1 * q2 (mod n)
    t1 = t % (1 << (k + 1) * 64)
    m = q1 * r2 + q2 * r1# 计算 m = q1 * r2 + q2 * r1 (mod n)
    m = m % (1 << (k + 1) * 64)
    # 计算 t = t + m * mu (mod n)
    t = (t1 + m * mu) % (1 << (k + 1) * 64)
    # 计算 r = a * b (mod n)
    r = (r1 * r2 - t * n) % n
    if r < 0:
        r += n
    return r

def test_barrett_mod_mul():
    num_bits = 2048
    a = random.getrandbits(num_bits)
    b = random.getrandbits(num_bits)
    n = random.getrandbits(num_bits)
    k = (n.bit_length() + 1) // 2
    mu = (1 << (k + 1) * 64) // n
    result = barrett_mod_mul(a, b, n, mu)
    expected = (a * b) % n
    print(result)
    print(expected)
    assert result == expected, f"Test case failed: {a} * {b} (mod {n}) = {result}, expected {expected}"
    print("test cases passed.")
```

```
34
35 test_barrett_mod_mul()
```

运行结果如下：

```
========== RESTART: F:/Study/大三上/密码工程实验/实验五/Barrett multiplication.p
y ==========
5436286808225267262519180981013523009425716712718472519963901417591349884717595
7
2568015360182117165640280430449142186778923231424286782581327784754641729084875
2
9577602804184085386080024238918351719345723610252327894844805745724849248337309
4
2992958106184647499959677280932706885402148830376589736421893636673133526053036
7
2654231389663685923352007484088276616330429131202183224906101476282158082045658
2
5817778029984775546072601241884262755945662761012683281868716230388306286087609
4
1496294848166551076752015443656303592355609337290325856631698856571646021114203
3
8917330215554881594166871447185920318934119040953456229 1
5436286808225267262519180981013523009425716712718472519963901417591349884717595
7
2568015360182117165640280430449142186778923231424286782581327784754641729084875
2
9577602804184085386080024238918351719345723610252327894844805745724849248337309
4
2992958106184647499959677280932706885402148830376589736421893636673133526053036
7
2654231389663685923352007484088276616330429131202183224906101476282158082045658
2
5817778029984775546072601241884262755945662761012683281868716230388306286087609
4
1496294848166551076752015443656303592355609337290325856631698856571646021114203
3
8917330215554881594166871447185920318934119040953456229 1
test cases passed.
```

Figure 2: 测试结果

4.4 Montgomery 约简的 python 实现

按照 3.2.2 部分的分析，编写约简代码，输入参数为 X、模数 N 和模基 R，首先计算模数 N 在模基 R 下的逆元素，即变量 $N_{pie} = R - inv(N, R)$，然后计算 $m = X * N_piemodR$，和 $y = (X + m * N)//R$，得到的 y 是 Montgomery 约简后的初步结果，如果 y > N，则 y -= N，从而确保 y 的值小于模数 N。

代码如下：

```
1 def Mon_reduction(X, R, N):
2     N_pie = R - inv(N, R)
3     m = X * N_pie % R
4     y = int((X + m * N) // R)
5     if y > N:
6         y -= N
7     return y
```

4.5 Montgomery 模乘的 python 实现

下面编写 Montgomery 模乘的实现函数，同样传入三个参数：输入参数 a、b 和模数 N，输出经过蒙哥马利乘法计算后的结果 y，下面是具体流程：

首先，通过调用函数 find_R(N) 找到合适的模基 R. 将输入参数 a 和 b 分别与模基 R 相乘，并对模数 N 取余，得到 $a\_hat = a * RmodN$ $b\_hat = b * RmodN$。

计算 $X = a\_hat * b\_hat$。通过调用函数 Mon_reduction(X, R, N) 进行 Montgomery 归约操作，得到 X_1 = Mon_reduction(X, R, N)。

再次调用函数 Mon_reduction(X_1, R, N) 进行 Montgomery 归约操作，得到最终的结果 y = Mon_reduction(X_1, R, N), 返回经过 Montgomery 归约计算后的结果 y。

我在 Montgomery 模乘的代码中同样使用了 Karatsuba 乘法，以达到更好的乘法性能优化。

具体实现如下：

```
def find_R(N):
    """R >= 2^k > N"""
    N_origin = N
    k = 0
    while N >= 1:
        k += 1
        N = N >> 1
    R = 2 ** k
    while 1:
        if gcd(N_origin, R) == 1:
            break
        else:
            R += 1
    return R

def Mon_MultMon(a, b, N):
    R = find_R(N)
    a_hat = a * R % N
    b_hat = b * R % N
    X = a_hat * b_hat
    X_1 = Mon_reduction(X, R, N)
    y = Mon_reduction(X_1, R, N)
    return y
```

4.6 Montgomery 模幂的 python 实现

传入三个参数：输入参数 a、指数 b 和模数 N，输出经过 Montgomery 幂运算计算后的结果 ans。

利用二进制表示法，将指数 b 分解为若干个 2 的幂次方相加的形式，并使用 Montgomery 乘法即可快速计算幂次方，代码如下：

```
def Mon_Mod(a, b, N):
    R = find_R(N)
    a_hat = a * R % N  # a -> Montgomery
    ans_hat = R % N    # initialize ans = 1 -> Montgomery
    while b != 0:
        if b & 1:
            ans_hat = Mon_reduction(ans_hat * a_hat, R, N)  # ans = ans * a % N
        b = b >> 1
        a_hat = Mon_reduction(a_hat * a_hat, R, N)  # a = a * a % N
    ans = Mon_reduction(ans_hat, R, N)
    return ans
```

4.7 编写 Montgomery 算法的测试部分

分别调用 2048bit 的蒙哥马利约简/模乘/模幂来测试，代码如下（特别的，由于计算的位数都为 2048bit，为了避免模逆运算触发 python 默认的递归 1000 次上限，调用 sys 函数 sys.setrecursionlimit(8000) 扩大递归上限）：

```
print("----------reduction----------")
sys.setrecursionlimit(8000)
num_bits = 2048
a = random.getrandbits(num_bits)
b = random.getrandbits(num_bits)
n = random.getrandbits(num_bits)
ans3 = a * inv(b, n) % n
print("Simple result:", ans3)
```

```
ans4 = Mon_reduction(a, b, n)
print("Mon reduction:", ans4)
print("Success" if ans3 == ans4 else "Fail")

print("----------Mod----------")
a = random.getrandbits(num_bits)
b = random.getrandbits(num_bits)
n = random.getrandbits(num_bits)
ans1 = a * b % n
ans2 = Mon_MultMon(a, b, n)
print("Simple result:", ans1)
print("Mon MultiMod :", ans2)
print("Success" if ans1 == ans2 else "Fail")

print("----------Power Mod----------")
a = random.getrandbits(32)
b = random.getrandbits(16)
n = random.getrandbits(1024)
ans5 = a ** b % n
ans6 = Mon_Mod(a, b, n)
print("Simple result:", ans5)
print("Mon MultiMod :", ans6)
print("Success" if ans5 == ans6 else "Fail")
```

运行得到以下结果，说明三个函数都运行成功：

Figure 3: 测试结果

参考文献

[1] Cao ZJ, Wu XJ: An improvement of the Barrett modular reduction algorithm. International Journal of ComputerMathematics.DOI:10.1080/00207160.2013.862237. Taylor Francis (2013)

[2] Koc CK, Acar T, Kaliski BS (1996) Analyzing and comparing Montgomery multiplication algorithms. IEEE Micro 16(3):26–33