密码学实验报告 7

张天辰 17377321

2019 年 5 月 16 日

1 大数运算——C++ 语言实现

1.1 大数运算算法

首先，为了存储大数又兼顾效率，采用如下的数据结构：将大整数切割成一段一段，即每次把大数模 10^9 的结果存储在列表中，再把大数除以 10^9。这样，它就会被 9 位 9 位存储起来。

在完成了对大数的构造之后，就需要进行各种关系的重写，并且重载运算符。对于大小关系，只需要比较两个数的分段数量，如果相同再逐个比较每个分段即可。

对于大数加法，只需要从对齐的低位分段开始逐段相加即可，每次相加之后判断进位。对于大数减法，也是类似的，只不过逐段相减之前要先判断能不能减，如果不能减则要向高位借位。对于乘法，需要按照类似竖式计算的方法进行逐段乘法和移位相加。以上三种算法都基于竖式计算，比较直观。

除法计算如果直接采用仿照竖式的方法，效率较低。我参考了网上的一种高效算法。首先，有这样的结论：

$$\frac{\overline{a_{n-1}a_{n-2}\cdots a_1a_0}}{\overline{b_{n-1}b_{n-2}\cdots b_1b_0}} \leqslant \frac{\overline{a_{n-1}a_{n-2}\cdots a_i}}{\overline{b_{n-1}b_{n-2}\cdots b_i}}$$

于是，可以通过取被除数和除数的高位来估计商，并根据这个估计的商更新被除数，通过计算逼近实际的商。具体算法如下，对于 A/B 有：

1 计算 $C_0 = A/B$，得到商 V_0。令 $A_1 = B * V_0 - A$。

2 计算 $C_1 = A_1/B$，得到商 V_1。令 $A_2 = B * V_1 - A_1$。

3 $\cdots$

4 计算 $C_n = A_n/B$，得到商 V_n。$V_n * B - A = 0$。

则实际的商约为 $V_0 - V_1 + V_2 \cdots$。实际上，真实的商比上述估计值小一或相等，只需估计后进行调整、验证即可。对于大数的除法，只要先模仿竖式计算进行，在每一步的除法中应用上述算法即可。

关于模幂运算，只是把之前写过的快速幂算法搬过来即可。因为重载了运算符，所以几乎不用做改动。

1.2 算法实现

我写的实际算法是类方法，这里变换成普通函数。

Algorithm 1 大整数比较

```
function COMPARE(a, b)
    len1 ← a.data.size()
    len2 ← b.data.size()
    if len1 ≠ len2 then
        return len1 > len2?1 : −1
    end if
    for cmp from len1 − 1 to 0 do
        if a.data[cmp] ≠ b.data[cmp] then
            return a.data[cmp] > b.data[cmp]?1 : −1
        end if
    end for
    return 0
end function
```

Algorithm 2 大整数加法

```
function ADD(a, b)
    len1 ← a.data.size()
    len2 ← b.data.size()
    result ← null
    for cnt from 0 to minlen1, len2 do
        result 写入 a.data[cnt] + b.data[cnt] + carry
        计算进位 carry
    end for
    for cnt from minlen1, len2 to maxlen1, len2 do
        result 写入 maxa, b[cnt] + carry
        计算进位 carry
    end for
    result 写入余下的 carry。
    return result
end function
```

Algorithm 3 大整数减法

```
function SUBTRACT(a, b)
    保证 a > geqslantb，否则交换 a, b
    len1 ← a.data.size()
    len2 ← b.data.size()
    result ← null
    for cnt from 0 to len2 do
        if a.data[cnt] < b.data[cnt] + carry then
            result 写入 a.data[cnt] − b.data[cnt] − carry + MAX
```

$carry = 1$
else
$result$ 写入 $a.data[cnt] - b.data[cnt] - carry$
$carry = 0$
end if
end for
for cnt from $len2$ to $len1$ **do**
if $a.data[cnt] < carry$ **then**
$result$ 写入 $a.data[cnt] - carry + MAX$
$carry = 1$
else
$result$ 写入 $a.data[cnt] - carry$
$carry = 0$
end if
end for
return $result$
end function

Algorithm 4 大整数乘法

function MULTIPLY(a, b)
$len1 \leftarrow a.data.size()$
$len2 \leftarrow b.data.size()$
保证 $a > geqslantb$，否则交换 a, b
for $cnt2$ from 0 to $len2$ **do**
在 $temp$ 低位写入 $cnt2$ 组 0，用于移位
$b.data[cnt2]$ 依次乘 $a.data$ 各组，写入 $temp$，计算进位
将 $temp$ 结果写入 $result$
end for
return $result$
end function

Algorithm 5 大整数除法

function DIVIDE(a, b)
$len1 \leftarrow a.data.size()$
$len2 \leftarrow b.data.size()$
保证 $a > geqslantb$，否则返回 0
选出 a 的高 $len2 - 1$ 位写入 $beDivide$
for 每次从 a 中选出一块拼接 $beDivide$ 和 b 进行除法 **do**
$value \leftarrow$PARTDIVIDE$(beDivide, b)$
$beDivide \leftarrow beDivide - b * value$

```
        value 写入 result 低位
    end for
    return result
end function
function partDivide(beDivide, b)
    temp1 ← partDivide
    temp2 ← b
    while temp1 ⩾ temp2 do
        if len(temp1.data) > len(temp2.data) then
            temp1 取高 2 组，temp2 取高 1 组除
        else
            temp1, temp2 取高 2 组除；如果不足就取 1 组
        end if
        更新 temp1
        将商按照正确的符号写入 result
    end while
    调整 result 使其正确
    return result
end function
```

取模操作实际上就是除法、减法和乘法的组合，这里略去。此外，模幂运算复用之前实验的快速幂算法逻辑，只不过在另一种语言上重写。因为重载了运算符，改动并不大，这里也从略。

1.3 算法测试

我没有测试加法减法除法，因为这些都可以在乘法和模幂运算中得到测试。这里的测试数据都是我在做完 RSA 实验之后从里面选取的数据。

1. 乘法测试

   这里采用了 RSA 密钥生成程序给出的两个大素数 p, q。它们相乘达到了 1024bit。输出第一行为运行时间，可以看出速度相当快；第二行为其与正确答案是否相等的比较，为 1 代表相同，结果正确。

2. 模幂测试

   同样采用 RSA 解密的中间数据。其中 n 为 1024bit，其余的 c, d 大小相仿，在 1000bit 以上。由代码可以看出，运行时间约 5 秒。我在 python 下对同样的数据进行了测试。首先可以证明结果的正确性，其次，python 对这样的数据几乎是在一瞬之间给出答案，说明我的代码在除法上与 python 还有很大的差距。

```cpp
int main(int argc, const char * argv[]) {
    BigInteger
        p("102692825601333568171084073434398636849176067673469971069983997255133098703903315620509812479753956698570046407679186728009955526953397015947550076653790 99");
```

```cpp
int main(int argc, const char * argv[]) {
    BigInteger
        p("1026928256013335681710840734343986368491760676734699710699839972551330987039033156205098124797
        5395669857004640767918672800995552695339701594755007665379099");
    BigInteger
        q("1140788152752573133826182529001210100132714764277184238101637045939995588803012924265278764257
        2757757018243217871807023755437966997834802463768757457636411");
    BigInteger
        d("2232994078584682478109815104146584133431307730335005398339199855732145633089755577421237529421
        4474708057792661081316923431086832139465173782834126385268649108515405057016421089800245128856734
        1953123970487521744363349880157925343752507293688086152022546931049063767777853623688581100475228
        2550524533243241082 7");
    BigInteger
        n("1171507588186874715537182507120317170394872934234944148734449316956859062176618973045084644704
        8917118657605939373596465202728080045748580136768197957045830376149271053404134242246679652148502
        8971706653063121598021718557354736363418743555536174350765795912889223353384778734543571754593736
        392809271522820773689");
    time_t start, end;
    start = clock();
    BigInteger n1 = p * q;
    end = clock();
    cout<<(double)(end - start) / CLOCKS_PER_SEC<<endl;
    cout<<(n == n1)<<endl;
    return 0;
```

```
0.000198
1
Program ended with exit code: 0
```

图 1: 大数乘法测试

```cpp
int main(int argc, const char * argv[]) {
    BigInteger
        c("4841452323869829528547179938298799163791493856219301490289015385003044188042883934607105951565
        3499509252249257856692449071732649412416739147176616265241735381054084429325074723213517775131473
        5002326694039476244465546358648967731076943635919547755320035675475176536176956094275084669699725
        76587715171890704707");
    BigInteger
        d("2232994078584682478109815104146584133431307730335005398339199855732145633089755577421237529421
        4474708057792661081316923431086832139465173782834126385268649108515405057016421089800245128856734
        1953123970487521744363349880157925343752507293688086152022546931049063767777853623688581100475228
        25505245332432410827");
    BigInteger
        n("1171507588186874715537182507120317170394872934234944148734449316956859062176618973045084644704
        8917118657605939373596465202728080045748580136768197957045830376149271053404134242246679652148502
        8971706653063121598021718557354736363418743555536174350765795912889223353384778734543571754593736
        392809271522820773689");
    time_t start, end;
    start = clock();
    BigInteger m = c.power(d, n);
    end = clock();
    cout<<m.toString()<<endl;
    cout<<(double)(end - start) / CLOCKS_PER_SEC<<endl;
    return 0;
```

```
5537346354412453409978822342087812629042789178975963771851163063012506
9849666991086051107478019920449352768132825935813825507580157451065571
5717700839066600229827488883766702092916880082258892131125860930362
2047675255018617669355911949479546201869905632843300109820696841116840
97188809557304285713782838 7
4.86131
Program ended with exit code: 0
```

图 2: 大数模幂测试

2 RSA 加解密——Python 实现

2.1 RSA 逻辑与 OAEP 填充

RSA 是基于大整数分解难题的公钥算法。OAEP 又称最佳非对称加密填充，是一种利用掩膜生成函数和哈希函数构造的填充方法，可以很好地防止选择密文攻击。相对 RSA 破解的难度而言，进行 OAEP 填充是有意义的。关于 OAEP 的填充算法，具体可以参见 RSA 标准。

RSA 算法的基本内容如下：选取两个大素数 p, q，计算 $n = pq$，$\phi(n) = (p-1)(q-1)$。选取 e 满足 $\gcd(e, \phi(n)) = 1$，及 d 为 e 模 $\phi(n)$ 的逆。把 e, n 作为公钥，p, q, d 作为私钥。

加密过程为 $c = m^e \mod n$，解密过程为 $m = c^d \mod n$。其中解密过程可以利用中国剩余定理优化如下：要解 $c^d \mod n$ 即要解

$$x \equiv c^d \mod p \qquad x \equiv c^d \mod q$$

设

$$r1 \equiv d \mod (p-1) \qquad r2 \equiv d \mod (q-1)$$

则就转化为解

$$x \equiv c^{r1} \mod p \qquad x \equiv c^{r2} \mod q$$

利用中国剩余定理可求解。

2.2 算法实现

1. 密钥生成

   密钥生成可以单独进行。为了后续计算、填充的方便，这里保证 n 为 1024bit。具体实现方式如下：满足长度为 1024bit 的数范围如下：$[2^{1023}, 2^{1024}-1]$，因此 p, q 的范围就是 $[2^{\frac{1023}{2}}, 2^{512})$。$2^{\frac{1023}{2}} < 2^{511} * 1.5 = 2^{511} + 2^{510}$。

   为了保证在 p, q 都是奇数的时候再进行素性检验，可以将 p, q 的生成方式变为 $2k+1$ 的形式。将上述范围的 2 的幂次减 1，就是 k 的范围。p, q 重合或者 $pq > 2^1024$ 的概率太小，忽略不计。

Algorithm 6 RSA 密钥生成

---

function GENERATEKEY
    $p \leftarrow$GENERATEPRIME
    $q \leftarrow$GENERATEPRIME
    $n \leftarrow p * q$
    $\phi \leftarrow (p-1) * (q-1)$
    $e \leftarrow$RANDINT$(3, \phi - 1)$ 直到 $\gcd(e, \phi) = 1$
    $d \leftarrow$REV(e, ϕ)
    分别写公钥文件和私钥文件
end function
function GENERATEPRIME
    while 1 **do**

```
        temp ←RANDINT(2^510 + 2^509, 2^511)
        p ← 2 * temp + 1
        if MILLERRABIN(p) then
            return p
        end if
    end while
end function
```

2. OAEP 填充

   实在没什么特别的地方，无非是按照标准给出的算法做。这里给出流程图。解码时从 *maskedDB* 通过 *MGF* 及 *maskedSeed* 获得 *seed* 之后就显然了。

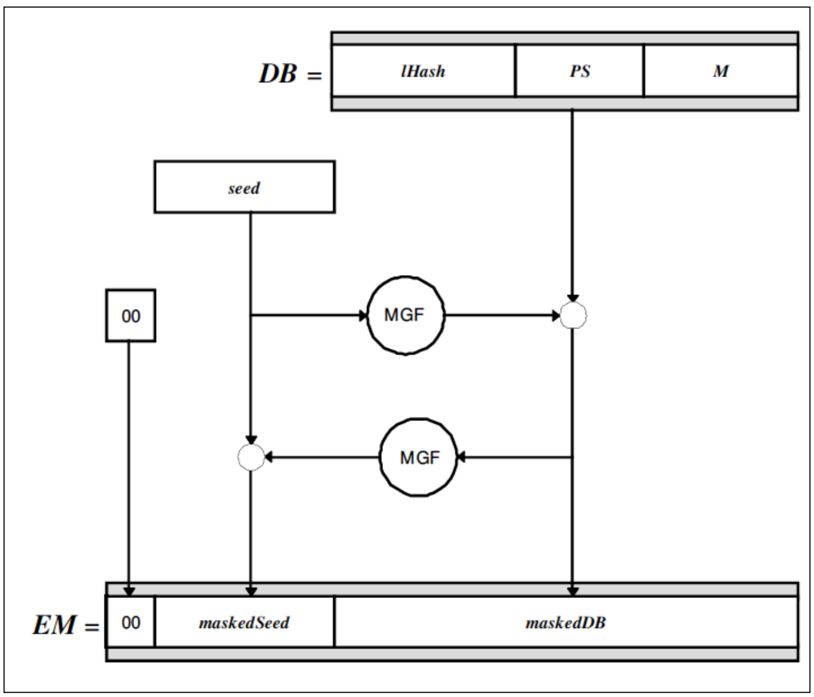

图 3: OAEP 流程

3. MGF 函数

   同样也是按照标准给出的算法逐步实现。

4. RSA 加解密

   首先，因为我在 OAEP 里的 hash 函数使用了 sha-256，根据 OAEP 编码的计算规则和范围，将明文 60 字节分组比较恰当。把文件读入后按照 60 字节分组，并且在最后进行填充。填充规则为填充若干个 0，并在最后一个字节写入填充的 0 的个数。

Algorithm 7 RSA

```
function ENCRYPT(plainList)
```

    for $p \in plainList$ **do**
        $plain \leftarrow$ 对 p 进行 OAEP 编码
        $cipherList.append(plain^e \mod n)$
    end for
end function
function DECRYPT($cipherList$)
    读文件得到 p, q, d
    for $c \in cipherList$ **do**
        $r1 \leftarrow d\%(p-1)$
        $r2 \leftarrow d\%(q-1)$
        $p1 \leftarrow$MODPOWER($c, r1, p$)
        $p2 \leftarrow$MODPOWER($c, r2, q$)
        $plain \leftarrow$CRT$([p1, p2], [p, q], n)$
        $temp \leftarrow$ 对 $plain$ 进行 OAEP 解码
        $plainList.append(temp)$
    end for
end function

2.3 测试结果

我实现了一个简单的 GUI 界面用于加密和解密，选取了一张约 400KB 的图片进行加密，再进行解密。如图 4，右边为加密后的文件，左边为解密得到的原文件，从预览图就可看出成功解密。图 4还显示了 GUI 界面的设计。加密时只需点击生成密钥就可生成公私钥对，解密时把私钥文件和代码放在同一文件夹即可。唯一的不足是代码执行速度太慢，加密这个文件用了约 40s，解密用了约 30s，需要后期的进一步优化。

图 4: RSA 测试

3 简单背包密码体制

3.1 背包密码体制及其攻击

背包密码体制的私钥为一个超递增背包序列。根据模数 m 和乘数 w，其中 $\gcd(m, w) = 1$，可以构造公钥：对私钥里的每个 r，对应公钥的权重为 $p \equiv rw \mod w$。利用公钥作为背包对消息加密得到密文。将密文在模 m 下乘以 w 关于 m 的逆，再解超递增背包问题就可以得到明文。

背包密码体制的攻击为只要选出 w, m 使得能把公钥背包变成超递增背包，就可以用那个背包和选出的 w, m 解密。奇怪的地方在于，我用这种方法编写的程序有可能得到错误的明文，初步猜测是 m 太小导致。

3.2 算法实现

Algorithm 8 背包密码

function GENERATEKEY(len)
  生成长度为 $plain$ 二进制长度的超递增背包，为私钥
  随机产生 m, w 使得 $\gcd(w, m) = 1$
  私钥的每一项模 m 乘 w，得到公钥
end function
function ENCRYPT($plain$)
  用公钥背包加密 $plain$ 的二进制串
end function
function DECRYPT($cipher$)
  $c \leftarrow cipher * \overline{w} \mod m$
  用私钥背包解 c
end function

Algorithm 9 背包密码攻击

function ATTACK($pub, cipher$)
  随机生成 m, w 使得 $\gcd(m, w) = 1$
  如果公钥模 m 乘 $\overline{w}$ 没得到超递增背包，就返回上一步
  用得到的私钥进行解密 $cipher$
end function

3.3 测试样例

测试使用的数据是数字，可以自动生成公私钥加密或解密。没有写成加密解密分开实现的形式，将进行改进。

攻击算法只能解决长度为 9 以下的背包，因为使用了随机数所以耗时不等，奇怪的是可能解密出错，正在研究。

```
WangtekiMacBook-Air:Exp7 WangJM$ python3 BagCrypto.py
Plain number: 1305938172
Cipher:
11279230732227
Decrypt Cipher into Plain:
1305938172
```

图 5: 背包密码测试

```
WangtekiMacBook-Air:Exp7 WangJM$ python3 AttackBag.py
Input plain number: 209
Cipher:
7302
Attack accomplished.
Private Key:
[10, 41, 165, 402, 1186, 3187, 8004, 19278]
Plain:
209
```

图 6: 背包密码攻击测试