密码学实验报告 实验一

2019 年 3 月 5 日

1 厄拉多塞筛法

1.1 算法原理

一个数N如果是合数，那么它的所有因子不超过$\sqrt{N}$，一个直观的求小于等于N的所有素数的朴素方法是对$[2,N]$内的所有数字进行素数判定，其时间复杂度为$O\left(N*\sqrt{N}\right)$，当N很大的时候，程序效率较差，所以采用厄拉多塞(Eratosthenes)筛法来“筛选”素数。该方法先将2~N从小到大排列，选中2，然后将2的其他倍数都筛去；再选中下一个未被筛去的数3，将3的其他倍数都筛去，下一个未被筛去的数是5，将5的其他倍数筛去……以此类推，一直到没有数可以被选择为止，剩下未被筛去的数即为2~N中的所有素数。

1.2 算法实现的伪代码

厄拉多塞筛法 计算小于等于N的所有素数

输入： $N>0$

输出： 小于等于N的所有素数列表P

```
for i ← 2 to N do
    P[i] ← i
end for
for i ← 2 to ⌊√N⌋ do
    if P[i] ≠ 0 then
        for j ← i + i to N by i do
            if P[j] mod P[i] = 0 then
                P[j] = 0
            end if
        end for
    end if
end for
```

1.3 时间和空间复杂度分析

相比朴素求解方法的$O\left(N*\sqrt{N}\right)$的时间复杂度，厄拉多塞筛法的时间复杂度很小，为$O(N\ln\ln N)$，证明如下：

对于待筛选的最大数n，根据素数分布定理，其中的素数个数约等于$\frac{n}{\ln n}$，设第i个素数约为$i\ln i$，则算法总运行次数为：

$$\sum_{i=1}^{\frac{n}{\ln n}}\frac{n}{i\ln i}=n\sum_{i=1}^{\frac{n}{\ln n}}\frac{1}{i\ln i}$$

用积分估计上式，有如下估计：

$$\sum_{i=1}^{\frac{n}{\ln n}}\frac{1}{i\ln i}=O\left(\int_{2}^{\frac{n}{\ln n}}\frac{1}{i\ln i}di\right)$$

由微积分基本定理：

$$O\left(\int_{2}^{\frac{n}{\ln n}}\frac{1}{i\ln i}di\right)=O(\ln\ln n)$$

所以：

$$\sum_{i=1}^{\frac{n}{\ln n}}\frac{n}{i\ln i}=O(n\ln\ln n)$$

证毕。

由于需要筛选小于等于n的所有素数，需要建立一个与n同等大小的数组，故厄拉多塞筛法的空间复杂度为$O(n)$。

1.4 算法的优化以及对比分析

虽然厄拉多塞筛法比经典的素数判定算法效率高很多，但也存在改进的余地，我们利用以下几个方面对厄拉多塞筛法进行改进。

1.4.1 去偶存奇

除 2 以外的偶数均不是素数，所以构造数据序列时，直接去掉偶数只留奇数，从而可以减少一半的数据，即节约了存储空间，也减少了筛选偶数对象所需的时

间。当筛选小于等于N的素数时，只需构造大小为$\lfloor N/2 \rfloor$的数组，把$3,5,7,9,\cdots$等依次存入，任意数字n与下标i有固定的映射关系

$$n = 2 * i + 3$$

筛除时，用数组前部$\sqrt{N}$范围内的小素数去除后面的数字，如果能够整除则是合数，将该数从数据序列中剔除。

1.4.2 按步长值筛选合数

在如 1.4.1 所述的只有奇数的数据序列中，合数的分布有一定规律，按照因子即可直接通过调整数组下标改变时的步长来筛选数据，按照上述下标和数字的映射关系，对于将要筛选的素数n，需要筛除的倍数从$3(i+1)$开始，接着以$2*i+3$递增。

1.4.3 优化筛除起点

通过进一步的分析，按照下标对合数进行筛除时，有重复筛除的情况存在：第一个筛除的是 3 的倍数，在筛除 5 的倍数时，有一部分数字已经被筛过了，所以对任意素数n（对应下标为i），只需从它的n倍数开始筛除，筛除起点为

$$i * (2 * i + 6) + 3$$

1.4.4 缩小筛除范围

再进一步分析，由于筛除起点后移，对于长度为N的数据序列，假定当下标$i = \sqrt{N}$对应的数为素数，筛除起点为$\sqrt{N} * \left(2 * \sqrt{N} + 6\right) + 3 = 2 * N + 6 * \sqrt{N} + 3$，早已越界，因此筛除$3 \sim \sqrt{2N}/2$范围内的素数的合数，就能筛去$3 \sim N$范围内所有的合数，缩小了检验范围。

1.5 探究理论上无限的素数筛选方法

在传统筛法算法中，获取N以内的素数需要消耗与N等量大小的内存，这导致N过大时，容易造成内存溢出的问题，极大的限制了素数筛选的可行范围。针对此问题，需要有一种突破计算机内存分配限制的算法，从而提升寻找素数的范围。下面引入一种分区间筛选的算法。

传统筛法寻找素数时，由于没有任何已知素数，因此需要从第一个素数 2 开

始计算，筛选序列的构造也必须从 2 开始，很大程度上限制了筛选的范围。若已知$2\sim n$范围内素数，则可以通过遍历一次素数集合快速计算出n^2以内全部的素数。对于区间$[m,n]$，需要使用$\sqrt{n}$内的素数进行筛选，所以该算法的时间复杂度为$O\left((n-m)\ln\ln\sqrt{n}\right)$，略低于厄拉多塞筛法。

算法执行过程为：首先设置筛选区间$[m,n]$，初始化筛选数组$a[m,n]$，获取$\lfloor\sqrt{N}\rfloor$以内的素数集合P，用P中的每个素数p对区间$[m,n]$内的整数筛选，即可得到$[m,n]$内所有的素数集合。事实上，若要筛去区间$[m,n]$中素因子为p的数字，则因子i只需取$\lfloor m/p\rfloor+1$到$\lfloor n/p\rfloor+1$之间的所有数字即可。显然地，该算法也可以通过上述几节中提到了去偶存奇等方法进行进一步的优化。

运用分区间筛法，通过对较大的区间进行合理的划分并迭代计算可以大幅扩大筛选的范围，并且可以将每一次新的筛选结果存入素数集中，扩大的素数集将为更大数量级的筛选提供保证。例如需要计算 1000 亿以内的素数，假设内存空间仅允许单次进行 10 亿大小的区间筛选，则可以将 1000 亿的区间以 10 亿大小等分为 100 个子区间，首先用厄拉多塞筛法得到$2\sim10^{10}$内的素数集并保存，然后用区间筛法对之后的每个子区间进行计算，每轮计算完成后将结果添加到已有的素数集中，为以后的筛选提供素数。经过反复迭代即可获得 1000 亿以内的素数集。

1.6 测试样例及运行结果

我们选取了规模分别在10^5、10^6、10^7、10^8的输入数据来检验三种算法筛选出全部素数所需的运行时间，结果如下表所示。

表 1 三种算法实验结果（单位：秒）

数据个数	经典判断法	厄拉多塞筛法	优化后筛法
10^5	0.31	0.03	0.01
10^6	8.49	0.45	0.17
10^7	236.93	4.86	1.90
10^8	N/A	55.56	20.83

对于分区间筛选法，我们探究了在规模为10^{10}的输入数据下算法的运行效率，首先以10^7为划分大小将上述区间等分，运行该算法筛选素数，理论上需要进行 1000 次筛选才能得到10^{10}以内的全部素数，几轮迭代后的时间曲线如下图：

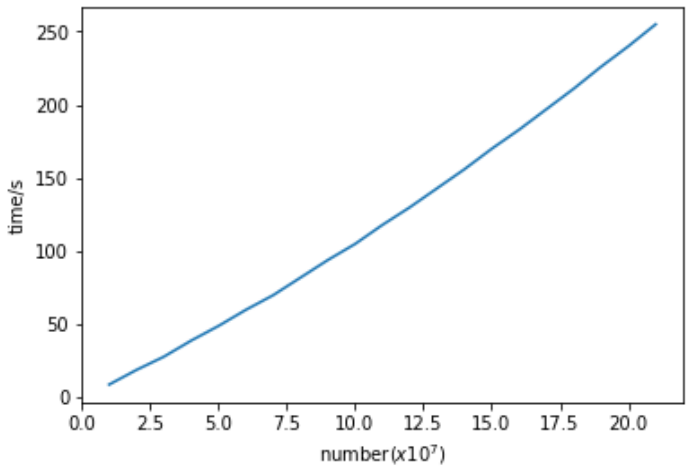

图 1 分区间筛选时间与迭代数量关系

在 Python 中使用一次多项式曲线拟合后，估计得到计算所需用时约为 12305 秒。我们再用10^8为划分大小等分区间，理论上需要 100 次筛选得到10^{10}以内的全部素数，几轮迭代后的时间曲线如下图：

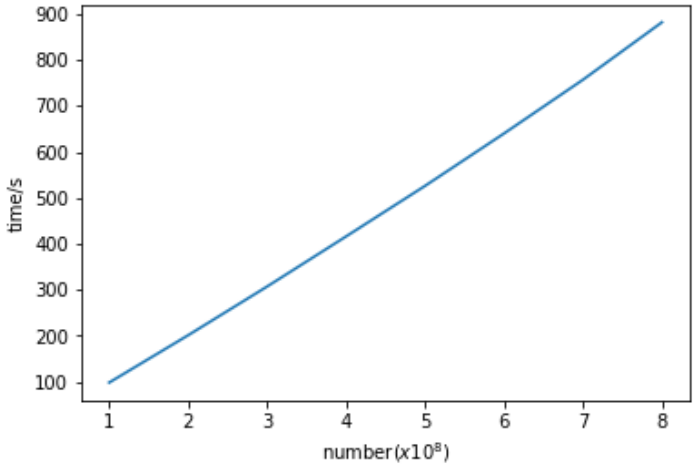

图 2 分区间筛选时间与迭代数量关系

同样用一次多项式做曲线拟合，估计得到计算所需用时约为 11139 秒，可见用于划分区间的数越大，计算效率会更高一些。分区间筛选提供了理论上无限的素数筛选方式，并且时间复杂度与筛法相同。

最后，为了检验素数筛选算法的正确性，我们使用经典检验方法、厄拉多塞筛法和改进后的算法计算了 1000 以内的素数，将得到的结果与标准素数表比对，证明计算的结果是正确的。

2 欧几里得算法

2.1 欧几里得算法原理

欧几里得算法依赖于$d = \gcd(a,b) = \gcd(b,r)$的正确性，其中$a = q * b + r$，下面对此进行证明：

设d是a和b的公因数，则$d|a$且$d|b$，于是$d|(a - q * b)$，也就是$d|r$，又因为$r = a - q * b$，所以d是b和r的公因数。

设d是b和r的公因数，则$d|b$且$d|r$，于是$d|(q * b + r)$，所以$d|a$，所以d是a和b的公因数。

综上所述，a和b的所有公因数和b和r的所有公因数是相同的，那么d是a和b的最大公因数，当且仅当d是b和r的最大公因数。

2.2 欧几里得算法实现的伪代码

欧几里得算法 计算两数的最大公因数

输入： 整数$a, b > 0$

输出： $d = \gcd(a,b)$

```
while b ≠ 0 do
    t ← b
    b ← a mod b
    a ← t
end while
```

2.3 时间和空间复杂度分析

欧几里得算法在平均情况下的复杂度分析需要大量复杂的数学运算分析，此处只给出最终的迭代平均次数结果：

$$\tau(a) = \frac{12}{\pi^2}\ln 2 \ln a + 1.467$$

我们可以简单考虑一种最差情况：m和n是两个相邻的斐波那契数，欧几里得算法需要迭代的次数相当于该斐波那契数列的长度，根据斐波那契数列通项公式：

$$F(n) = \frac{1}{\sqrt{5}}\left[\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n\right]$$

可见对于N，需要迭代的次数约为$n = O(\log N)$。

显然地，欧几里得算法的空间复杂度为$O(1)$。

2.4 扩展欧几里得算法的原理

扩展欧几里得算法是对欧几里得算法的扩展，已知整数a和b，扩展欧几里得算法可以在求得a, b的最大公约数d的同时，找到整数x, y满足：

$$ax + by = d$$

还可以利用扩展欧几里得算法计算模逆元，用于中国剩余定理或 RSA 加密算法的计算。

该算法在原算法的基础上增加了两个递归等式：

$$s_k = s_{k-2} - q_k s_{k-1}$$

$$t_k = t_{k-2} - q_k s_{k-1}$$

算法需要初始化$s_{-2} = 1, t_{-2} = 0, s_{-1} = 0, t_{-1} = 1$，算法终止于$r_N = 0$，上述等式的整数$x, y$分别由$s_N, t_N$给出。

2.5 扩展欧几里得算法实现的伪代码

2.5.1 迭代法

扩展欧几里得算法 计算两数的最大公因数和满足等式的整数

输入： 整数$a, b > 0$

输出： $d = \gcd(a, b)$和x, y满足$ax + by = d$

```
function exgcd(a, b)
    s0 ← 1, t0 ← 0
    s1 ← 0, t1 ← 1
    q ← ⌊a/b⌋
    r0 ← b, r1 ← a mod b
    while r1 ≠ 0 do
        x0 ← x1, x1 ← x0 − q * x1
        y0 ← y1, y1 ← y0 − q * y1
        q ← ⌊r0/r1⌋
        r0 ← r1, r1 ← r0 mod r1
    end while
    return d ← r0, x ← x1, y ← y1
end function
```

2.5.2 回代法

扩展欧几里得算法 计算两数的最大公因数和满足等式的整数

输入： 整数$a, b > 0, \forall x, y$
输出： $d = \gcd(a, b)$和x, y满足$ax + by = d$

```
function exgcd(a, b, x, y)
    if b = 0 then
        x ← 1, y ← 0
        return a
    end if
    d ← exgcd(b, a mod b, y, x)
    y ← y − ⌊a/b⌋ ∗ x
    return d
end function
```

2.6 两种扩展欧几里得算法实现的对比分析

两种算法的时间复杂度与欧几里得算法相同，均为$O(\log n)$，空间复杂度也均为$O(1)$，但是由于具体实现的不同，两种算法可能会有常数级的复杂度差别，迭代法在正向计算的同时就可以求出x, y，而回代法还需要反向计算才能得到x, y，在这里我们使用递归的形式实现了回代法，所以回代法的效率可能略低于迭代法，而在实现上以及空间复杂度略优于迭代法。

2.7 测试样例及运行结果

我们选取了规模为10^{500}、10^{1000}和10^{1500}的大数对三种算法效率进行比较，结果如下表所示。

表 2 三种欧几里得算法的运行时间（单位：毫秒）

数据规模	欧几里得算法	扩展欧几里得(迭代)	扩展欧几里得(回代)
10^{500}	0.48	0.93	0.95
10^{1000}	1.53	2.68	2.95
10^{1500}	2.88	6.14	7.80

可以看得出，三种算法的有同阶的渐进时间复杂度，但是由于运算时所需的计算步骤数不同，所以会有常量倍的时间复杂度差别，扩展欧几里得算法一般要比普通欧几里得算法耗时多，而对于两种不同的扩展欧几里得算法实现，由于回代法需要递归计算，所以耗时略多于迭代法。

下面用举例的方式验证算法正确性：

使用三种不同的方法计算 58 和 689 的最大公因数d，以及x, y使

$$58x + 689y = d$$

得到的结果均为：$d = 1, x = 297, y = -25$。

3 模幂算法

3.1 算法原理

在计算$a^n(mod\ \ m)$时，朴素的模幂算法是将a连乘，在连乘的同时不断对m取模，其成立的前提依赖于以下数论定理：

$$(a*b)\%m=\big((a\%m)*(b\%m)\big)\%m$$

这种算法的时间复杂度为$O(n)$，我们还可以使用快速幂的方法来进行优化。

设n的二进制表示为

$$n=a_0+a_1\cdot 2+\cdots+a_r\cdot 2^r$$

将底数a连续平方，计算

$$t_0\equiv a(mod\ \ n)$$

$$t_1\equiv t_0^2\equiv a^2(mod\ \ n)$$

$$t_2\equiv t_1^2\equiv a^{2^2}(mod\ \ n)$$

$$\cdots$$

$$t_r\equiv t_{r-1}^2\equiv a^{2^r}(mod\ \ n)$$

再将全部结果相乘，计算

$$a^n=t_{i_1}\times t_{i_2}\times\cdots\times t_{i_s}(mod\ \ n)$$

其中$\big(a_{i_1},\cdots,a_{i_s}\big)$是$(a_0,a_1,\cdots,a_r)$中所有的 1。即可得到最终运算结果。

3.2 常规模幂算法实现的伪代码

模幂算法 计算在模m下a^n的值

输入： a,n,m
输出： $a^n(mod\ \ m)$

```
function power(a, n, m)
    res ← 1
    for i ← 1 to N do
        res ← res * a mod m
    end for
    return res
end function
```

3.3 快速模幂算法实现的伪代码

快速模幂算法 计算在模m下a^n的值

输入： a, n, m

输出： $a^n (mod\ \ m)$

```
function quick_power(a, n, m)
    res ← 1
    while n ≠ 0 do
        if n mod 2 ≠ 0 then
            res ← res * a mod m
        end if
        a ← a * a mod m
        n ← ⌊n/2⌋
    end while
    return res
end function
```

3.4 两种模幂算法实现的对比分析

对于常规模幂算法，由于需要计算 n 次乘法，故时间复杂度为$O(n)$，空间复杂度为$O(1)$。

对于快速模幂算法，每次迭代中用 2 除 n，则最差情况的计算步数是$\log_2 n$，故该算法的时间复杂度为$O(\log n)$。由于在算法具体实现中只需要存储最终的结果变量，故空间复杂度为$O(1)$。

对于很大的数字来说，显然快速模幂算法的性能远优于常规模幂算法，故在计算中通常使用快速模幂方法。

3.5 测试样例及运行结果

我们选取了规模为10^{10^5}、10^{10^6}和10^{10^7}的大数对两种算法效率进行比较，模数为 10007，运行结果如下表所示。

表 3 两种模幂算法的运行时间（单位：毫秒）

数据规模	常规模幂算法	快速模幂算法
$n = 10^5$	8.6482	0.0059
$n = 10^6$	88.3808	0.0067
$n = 10^7$	866.7110	0.0075

最后再使用两种算法计算$13^{10000} (mod\ \ 10007)$的结果均为 3123，证明算法正确。

4 中国剩余定理

4.1 中国剩余定理基本原理

中国剩余定理的叙述如下：

设$m_1, m_2, \cdots, m_r$是两两互素的自然数，令$m = m_1 m_2 \cdots m_r = m_i M_i$，即$M_i = m_1 \cdots m_{i-1} m_{i+1} \cdots m_r$，$i = 1,2, \cdots, r$，则方程组

$$\begin{cases} x \equiv b_1 (mod\ \ m_1) \\ x \equiv b_2 (mod\ \ m_2) \\ \quad\quad \cdots \\ x \equiv b_r (mod\ \ m_r) \end{cases}$$

在模m的意义下有唯一解

$$x \equiv M_1' M_1 b_1 + M_2' M_2 b_2 + \cdots + M_r' M_r b_r (mod\ \ m)$$

其中M_i'是整数，使$M_1' M_i = 1 (mod\ \ m_i), i = 1,2, \cdots, r$。算法证明略。

在具体实现时，计算M_i'可利用先前的扩展欧几里得算法来求M_i的逆元。

4.2 算法实现的伪代码

中国剩余定理 计算一元线性同余方程组的解

输入： 参数数组b, m，方程个数n
输出： 同余方程的解x

```
function mod_reverse(a, m)
    d, x, y ← exgcd(a, m)
    return x mod m
end function

function crt(b, m, n)
    ans ← 0
    lcm ← 1
    for i ← 1 to n do
        lcm ← lcm * m[i]
    end for
    for i ← 1 to n do
        tpn ← ⌊lcm/m[i]⌋
        tpr ← mod_reverse(tpn, m[i])
        ans ← (ans + tpn * tpr * b[i]) mod lcm
    end for
    return ans
end function
```

4.3 测试样例及运行结果

尝试使用该算法求解同余方程组

$$\begin{cases} x \equiv 1(mod\ 4) \\ x \equiv 2(mod\ 9) \\ x \equiv 3(mod\ 11) \end{cases}$$

得到该方程组的解$x = 245$，证明算法正确。

5 总结

在本次实验中，我动手实现了四个密码学问题中经典的算法，巩固了对算法的理解，增强了自己的实践能力。厄拉多塞筛法利用合数必是小素数的倍数的特点将素数直接筛除出去，极大地提高了程序效率。但筛选法有明显可改进的余地，我们尝试了去偶存奇、按步长筛数、优化筛除起点和缩小检验范围等方面对厄拉多塞筛法进行了改进，改进后的筛选法减少了程序筛选素数时所需运行的次数，提高了程序运行效率。我们还探究了理论上无限的素数筛选方法，实现了分区间筛法，用已知的素数集在某个区间内寻找新的素数，同时进一步优化，突破了传统的筛法在范围上的局限。通过将传统筛法和分区间筛法相结合并合理划分区间，可以大幅提升筛选素数的范围。我们对比了两种不同的扩展欧几里得算法实现，从程序的执行原理上分析效率的差别。通过使用快速幂算法，大幅提升了模幂运算的效率。最后利用模逆元函数实现了中国剩余定理，能够求解一元线性同余方程组。但是在一些程序的实现细节上，仍然有改进空间。同时也由于 Python 是一种效率较低的语言，程序的优化可能并不能很好地体现在运行速度上，还需要进行进一步的探究。