密码学实验报告 1

张天辰 17377321

2019 年 3 月 7 日

1 Eratosthenes 筛法

1.1 递归算法基本原理

为求 $1 \sim n$ 的所有素数，可采用如下方法：先求出 $1 \sim \sqrt{n}$ 中所有素数，再把这些素数的倍数删去，留下的数就是 $1 \sim n$ 中所有素数。依照这样的算法进行递归，可以设置一个递归出口，即在 $n < 10$ 时直接给出相应的素数。

1.2 算法实现

算法 1 递归 Eratosthenes 筛法

输入: 数字上限 n

输出: $1 \sim n$ 范围内所有的素数

function BADERATOSTHENES(n)
    $standard = [2, 3, 5, 7]$
    if n $<$ 10 **then**
        return $pinstandard : p \leqslant n$
    else
        $primes \leftarrow$BADERATOSTHENES($[\sqrt{n}]$)
        $flag[1, 2, \cdots, n] \leftarrow False$
        for each p in primes **do**
            $j \leftarrow 2 * p$
            while $j \leqslant n$ **do**
                $flag[j] \leftarrow True$
                $j \leftarrow j + p$
            end while
        end for
        return $[p : flag[p] == False]$
    end if
end function

1.3 优化后的算法

递归算法进行了过多的重复计算。为此不再使用递归算法，而是把从 2 开始，把表中 2 的倍数删去，再找到下一个元素 3，把其倍数删去，直到找到 $\sqrt{n}$ 位置。每次把 i 的倍数删去时，可从 i^2 开始，因为小于 i^2 的 i 的倍数一定有小于 i 的素因子，因此已经被删去。

算法 2 优化 Eratosthenes 筛法

输入: 数字上限 n

输出: $1 \sim n$ 范围内所有的素数

```
function ERATOSTHENES(n)
    flag[1...n] ← False
    for each i ∈ [2, [√n]] do
        if flag[i] == False then
            j ← i * i
            while j ⩽ n do
                flag[i] ← True
                j ← j + i
            end while
        end if
    end for
    return [p : flag[p] == False]
end function
```

1.4 复杂度分析

在对素数 p 的倍数进行删除时，内层代码执行了 $\frac{n}{p} - 1$ 次，因此总消耗为:

$$\sum_{p \leqslant \sqrt{n}} \frac{n}{p} - 1 = n \sum_{p \leqslant \sqrt{n}} \frac{1}{p} - \pi(n)$$

其中 $\pi(n)$ 表示不超过 n 的素数个数。根据素数定理，$\pi(n) = O(\frac{n}{\ln n})$ 由 Mertens 第二定理:

$$\lim_{x \to \infty} \sum_{p \leqslant \sqrt{n}} \frac{1}{p} - \ln \ln n = M$$

其中 M 是 Meissel-Mertens 常数，约为 0.26。因此时间复杂度为 $O(n \log \log n)$ 对算法进行上述优化不会影响复杂度。空间复杂度为 $O(n)$。

1.5 再次优化——欧拉筛

欧拉筛算法为让每个合数只被其最小素因子筛一次，从而不重复筛选。此时，时间复杂度被优化为 $O(n)$。算法如下：如果 i 是 $prime[j]$ 的倍数，则设 $i = k \times prime[j]$，$i \times prime[j+1] = prime[j] \times k \times prime[j+1]$，当 $i = k \times prime[j+1]$ 时删去，因此算法到此就跳出循环。

算法 3 Euler 筛法

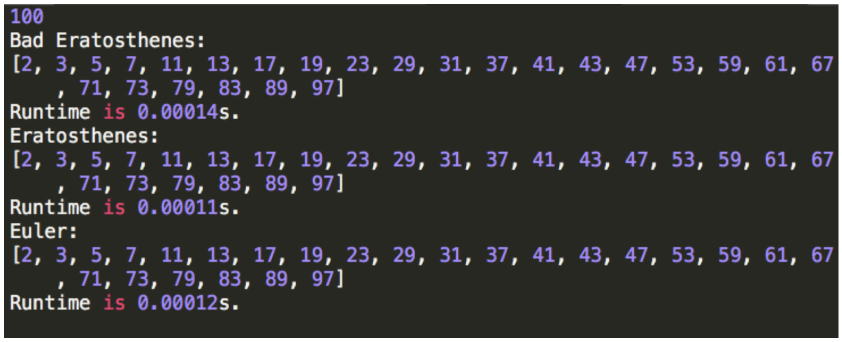
输入: 数字上限 n

输出: $1 \sim n$ 范围内所有的素数

```
function EULER(n)
    primes ← []
    flag[1...n] ← True
    for each i ∈ [2, n] do
        if flag[i] == True then
            primes.append(i)
        end if
        for each p in primes do
            if i * p < n then
                Break
            end if
            flag[i * p] ← False
            if i%p == 0 then
                Break
            end if
        end for
    end for
    return primes
end function
```

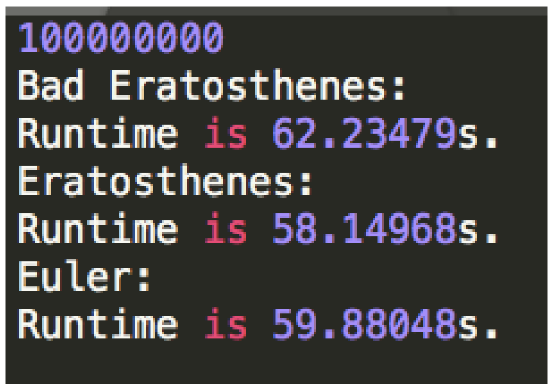
1.6 测试样例

先验证算法正确性，见图 1。在确认正确性后，为节省时间不再输出结果，只比较时间，见图 2



```
100
Bad Eratosthenes:
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67
    , 71, 73, 79, 83, 89, 97]
Runtime is 0.00014s.
Eratosthenes:
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67
    , 71, 73, 79, 83, 89, 97]
Runtime is 0.00011s.
Euler:
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67
    , 71, 73, 79, 83, 89, 97]
Runtime is 0.00012s.
```

图 1: 筛法



```
100000000
Bad Eratosthenes:
Runtime is 62.23479s.
Eratosthenes:
Runtime is 58.14968s.
Euler:
Runtime is 59.88048s.
```

图 2: 筛法（大数）

2 Euclid 算法

2.1 Euclid 算法原理

Euclid 算法又称为辗转相除法，是一种求最大公因数的方法。其算法设计基于如下事实：若 a/b 的带余除法写成如下形式

$$a = q \times b + r$$

则有

$$gcd(a, b) = gcd(b, r)$$

如此可将除法规模不断缩小，以达到求最大公因子的目的。

2.2 算法实现

算法 4 Euclid 算法

输入: $num1$, $num2$

输出: $gcd(num1, num2)$

```
function EUCLID(num1, num2)
    if num1 < num2 then
        num1, num2 ← num2, num1
    end if
    while num1%num2 ≠ 0 do
        a, b = b, num1%num2
    end while
    return num2
end function
```

2.3 算法复杂度分析

显然，该算法空间复杂度为 $O(1)$，下面分析时间复杂度：

不妨设输入的两个数为 a 和 b，$a \geqslant b$，并设需要做 n 次除法。令 $r_0 = a, r_1 = b$。则：

$$r_0 = q_1 r_1 + r_2 \quad r_1 = q_2 r_2 + r_3 \quad \cdots \quad r_{n-1} = q_n r_n$$

容易发现

$$q_1, q_2, q_3, \quad \cdots \quad q_{n-1} \geqslant 1, q_n \geqslant 2$$

则

$$r_n \geqslant 1 = f_2 \quad r_{n-1} \geqslant 2r_n \geqslant 2f_2 = f_3 \quad r_{n-2} \geqslant f_2 + f_3 = f_4 \quad \cdots$$

其中 f_n 表示 Fibonacci 数列第 n 项。故

$$r_1 \geqslant f_{n+1} > \alpha^{n-1}$$

其中 $\alpha = \frac{\sqrt{5}-1}{2}$。因此

$$\lg r_1 = \lg b > \lg \alpha^{n-1} > \frac{n-1}{5}$$

所以

$$n \leqslant 5 \lg r_1 = 5 \lg b$$

因此 Euclid 算法的时间复杂度为 $O(\lg mina, b)$

2.4 扩展 Euclid 算法

Bézout 定理给出，对于任意整数 a, b，总存在整数 s, t，使得 $sa + tb = gcd(a, b)$。Euclid 算法并不能给出如上的线性系数 s, t，但只需将算法稍作优化，便得到可以同时得到最大公因子和上述线性系数的扩展 Euclid 算法。

实现线性系数的求解可以有两种方法：一种是在递归求解最大公因子后回溯求解系数；另一种是在循环时直接计算系数，其原理如下：令 $r_{-1} = a, r_0 = b$

$$r_{i-2} = q_i r_{i-1} + r_i \quad i = 1, 2, \cdots, n$$

设 $r_i = ax_i + by_i$ 可推出

$$x_i = x_{i-2} - q_i x_{i-1} \quad y_i = y_{i-2} - q_i y_{i-1}$$

2.5 扩展算法实现

算法 5 Euclid 算法回溯

输入: $num1, num2$

输出: $gcd(num1, num2)$，线性系数 $coe1, coe2$

```
function backEuclid(num1, num2)
    if num1 < num2 then
        change ← True
        big, small ← num2, num1
    else
        change ← False
        big, small ← num1, num2
    end if
    if small == 0 then
        return big, 1, 0
    else
        rest, coeBig_n, coeSmall_n ← backEuclid(small, big%small)
        coeBig ← coeSmall_n
        coeSmall ← coeBig_n − coeSmall_n * [big/small]
        if change == True then
            return rest, coeSmall, coeBig
        else
            return rest, coeBig, coeSmall
        end if
    end if
end function
```

算法 6 扩展 Euclid 算法

输入： $num1, num2$

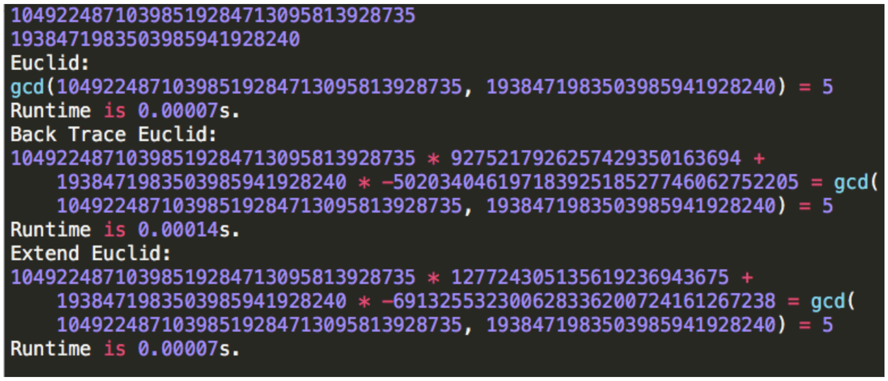
输出： $gcd(num1, num2)$，线性系数 $coe1, coe2$

```
function extendEuclid(num1, num2)
    if num1 < num2 then
        change ← True
        big, small ← num2, num1
    else
        change ← False
        big, small ← num1, num2
    end if
    rest, coeBig, coeSmall ← big, 1, 0
    rest_n, coeBig_n, coeSmall_n ← small, 1, 0
    while rest_n ≠ 0 do
        q ← rest//rest_n
        temp1 ← rest − q ∗ rest_n
        temp2 ← coeBig − q ∗ coeBig_n
        temp1 ← coeSmall − q ∗ coeSmall_n
        rest, coeBig, coeSmall ← rest_n, coeBig_n, coeSmall_n
        rest_n, coeBig_n, coeSmall_n ← temp1, temp2, temp3
    end while
    if change == True then
        return rest, coeSmall, coeBig
    else
        return rest, coeBig, coeSmall
    end if
end function
```

2.6 两种算法比较

回溯算法和扩展算法的时间复杂度相仿，均与普通欧几里得算法相同。但是因为回溯算法需要递归调用函数，因此比使用循环的算法需要更多的时间。在空间方面，递归算法也较为劣势，每次递归都要开辟另外的空间，复杂度为 $O(mina, b)$；而扩展算法复杂度为 $O(1)$。

2.7 测试案例



```
10492248710398519284713095813928735
19384719835039859941928240
Euclid:
gcd(10492248710398519284713095813928735, 19384719835039859941928240) = 5
Runtime is 0.00007s.
Back Trace Euclid:
10492248710398519284713095813928735 * 9275217926257429350163694 +
    19384719835039859941928240 * -5020340461971839251852774606275205 = gcd(
    10492248710398519284713095813928735, 19384719835039859941928240) = 5
Runtime is 0.00014s.
Extend Euclid:
10492248710398519284713095813928735 * 127724305135619236943675 +
    19384719835039859941928240 * -691325532300628336200724161267238 = gcd(
    10492248710398519284713095813928735, 19384719835039859941928240) = 5
Runtime is 0.00007s.
```

图 3: Euclid 算法

3 快速模幂算法

3.1 基本流程

快速模幂算法又称为“平方乘算法”。顾名思义，算法基本由平方操作和乘操作两部分组成。首先，将指数写成二进制的形式。此后，从二进制指数的高位向低位遍历，如果遇到 0，就将目前的结果平方后取模；否则就在平方后再模乘上原始底数。如此操作的原理是容易在数学上验证的。因为平方操作相当于把二进制指数左移一位，也就相当于在右侧加 0。如果再乘上原始底数，就相当于在右侧加 1。这样的算法避免了暴力算法的逐次模乘，大大提高了效率。

3.2 算法实现

算法 7 暴力模幂算法

输入： 底数 $base$, 指数 $power$, 模 mod

输出： $base^{power} \pmod{mod}$

```
function MODPOWER(base, power, mod)
    result ← 1
    for each i ∈ [1, power] do
        result ← (result * base) %mod
    end for
    return result
end function
```

算法 8 快速模幂算法

输入： 底数 $base$, 指数 $power$, 模 mod

输出： $base^{power} \pmod{mod}$

```
function MODPOWER(base, power, mod)
    binary[] ← bin(power)
    result ← 1
    for each i ∈ binary do
        if i ==' 0' then
            result ← result^2  %mod
        else
            result ← result^2  %mod
            result ← (result * base)  %mod
        end if
    end for
    return result
end function
```

3.3 算法对比分析

暴力模幂算法需要 n 次模乘，时间复杂度为 $O(n)$，空间复杂度为 $O(1)$；快速模幂算法（平方乘算法）需要做约 log_2n 次模乘，最多不超过 $2 \times log_2n$ 次模乘，时间复杂度为 $O(log_2n)$，空间复杂度为 $O(1)$。

两种算法速度在数字较大时差距特别明显。如图 4，在幂指数大小达到约 10^9 时，暴力算法需要 $271s$，而快速的算法只需要 $0.00003s$，平方乘算法优越性可见一斑。

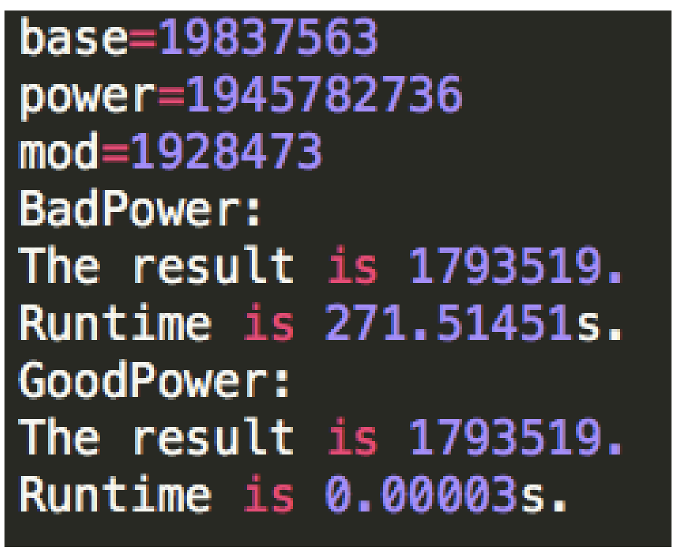


图 4: 两种模幂算法结果对比

4 中国剩余定理

4.1 中国剩余定理简述

设整数 $m_1, m_2, \cdots, m_n$ 两两互素，则对任意的整数 $a_1, a_2, \cdots, a_n$，方程组

$$\begin{cases} x \equiv a_1 \quad (mod\ m_1) \\ x \equiv a_2 \quad (mod\ m_2) \\ \quad \vdots \\ x \equiv a_n \quad (mod\ m_n) \end{cases} \tag{1}$$

有同余意义下唯一解，且可如下构造：

设

$$M = \prod_{i=1}^{n} m_i,$$

并设 $M_i = M/m_i, \quad \forall i = 1, 2, \cdots, n$。设 M_i' 为满足 $M_i' M_i \equiv 1 \quad (mod\ m_i)$ 的整数，则方程组的解为：

$$x \equiv \sum_{i=1}^{n} M_i' M_i a_i \quad (mod\ M)$$

4.2 算法实现

算法 9 中国剩余定理

输入: 余数表 $remainders[]$, 模数表 $mods[]$, 模数表积 $modProduct$

输出: 解 $root$

function REVERSE(n, mod)
  $gcd, coeN, coeMod \leftarrow$EXTENDEUCLID$(n, mod)$
  return coeN
end function
function CRT$(remainders, mods, modProduct)$
  $root \leftarrow 0$
  for each $i \in [1, len(mods)]$ **do**
    $rest \leftarrow modProduct/mods[i]$
    $rev \leftarrow$REVERSE$(rest, mods[i])$
    $root \leftarrow root + rev * rest * remainders[i]$
    $root \leftarrow root\%modProduct$
  end for
  return $root$
end function

CRT 算法需要调用之前写过的扩展欧几里得算法，可获得数论倒数（逆元），就是算法中得到了 n 的系数。模数之积 $modProduct$ 可以在输入时一并求出，故不在函数里给出，而是作为函数的参数给出。此外，本算法假设给出的模都是两两互素的。

4.3 测试样例

如图 5所示，输入方程数和每个方程的余数及模，可求出方程的解，且用时很快。

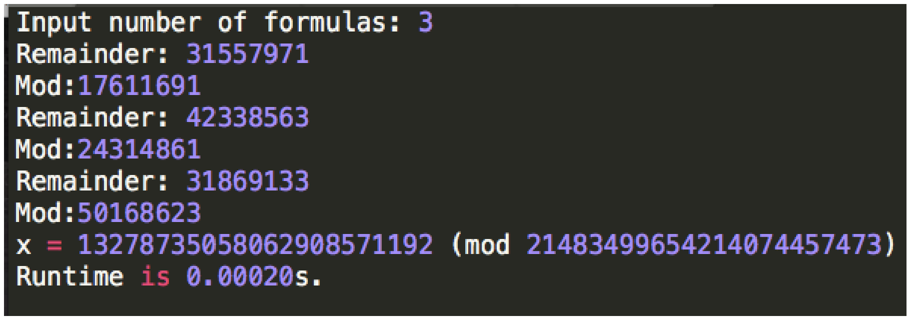


图 5: 中国剩余定理测试

5 感想

本次实验让我加深了对几个重要算法的理解，并且加强了 python 编程技术。在计算 Eratosthenes 筛法的时间复杂度时，我查阅了一些资料才计算出来，简单的算法的时间复杂度并不简单。

然而，在编写高效算法之前必须编写低效算法十分不合理。尤其是 Eratosthenes 筛法的低效算法（递归算法）其实根本就不存在，在 wiki 上也查不到，让我浪费了很多时间。我认为这些低效的过时算法应该被摒弃。对于更大的数，筛法业已十分低效，不再适合作为应用算法，所以没必要在此计较。

花费了很多时间完成本次实验和报告。功不唐捐。