密码学第二次实验报告

# 素性测试算法

## 原理

直接检验一个数是否为素数，可以用素数筛法，但是筛法的复杂度通常是多项式级别，很大时不实用。因此，就有了素性测试算法。素性测试算法有确定性素性测试算法，可以确定地判断出是否为素数；也有概率性素性测试算法，每次测试中，若返回“合数”，则一定是合数，否则可能为素数，且判断错误的概率有常数上界。只要足够多次应用概率性素性测试算法，即可得到一个，它是合数的概率任意小。

在实践中概率性素性测试算法比较常用，因为它运行时间可以接受，而且在测试次数足够多的情况下，可靠性较高。

以下算法都没有重点讨论为偶数的情况，这是因为为偶数时情况显然。

### Fermat素性测试算法

根据Fermat定理，对任意素数，。根据Euler定理，对任意素数，。因此可以在的完全剩余系中随机选取，首先计算。若，则返回“合数”。计算，检测是否成立。如果不成立，则返回“合数”，否则返回“不确定”。

### Miller-Rabin素性测试算法

奇素数满足以下三个性质：

的解只有和

因此可以在的简化剩余系中随机选取，检查是否成立。若不成立，则再检查是否成立。若二者都不成立，则返回“合数”，否则返回“不确定”。这里和容易计算。

### Solovay-Stassen素性测试算法

根据Euler准则，对任意素数，。但是，对任意正奇数，Jacobi符号也容易计算，且在为素数时与相应的Legendre符号相等。

因此可以在的简化剩余系中随机选取，检查是否成立。若不成立，则返回“合数”，否则返回“不确定”。其实在算出时就已经可以检查它是否属于了。如果不是，那么一定是合数。

## 伪代码

以下伪代码都是返回bool类型的值。若返回True，则不确定是否为素数。若返回False，则一定为合数。

### Fermat素性测试算法

**def** format(n: int, iterations: int) -> bool:

和为偶数的特殊情况判断

重复以下步骤次，若有一次返回“合数”，则**return** False:

选取中的随机整数

若，**return** “合数”

若，**return** “合数”

**return** True

### Miller-Rabin 素性测试算法

**def** miller\_rabin(n: int, iterations: int) -> bool:

和为偶数的特殊情况判断

计算和，

重复以下步骤次，若有一次不返回“不确定”，则**return** False:

选取中的随机整数

若，**return**“不确定”

若，**return**“不确定”

**return** True

### Solovay-Stassen素性测试算法

**def** solovay\_stassen(n: int, iterations: int) -> bool:

和为偶数的特殊情况判断

重复以下步骤次，若有一次返回“合数”，则**return** False:

选取中的随机整数

若，

若且，**return** “合数”

若，**return** “合数”

**return** True

## 分析

以下都认为模幂算法使用平方乘快速幂算法，设迭代次数为。

### Fermat素性测试算法

每轮迭代需要做一次欧几里得算法和模幂算法。因此时间复杂度为，空间复杂度为。

### Miller-Rabin素性测试算法

迭代前算出和，这可以通过不断对除以完成。时间复杂度为，空间复杂度为。

每轮迭代需要计算次模幂运算和次乘法运算。这里只执行次乘法运算是因为各可以通过把不断乘以得到。因此时间复杂度为，空间复杂度为。

因此总时间复杂度为，空间复杂度为。

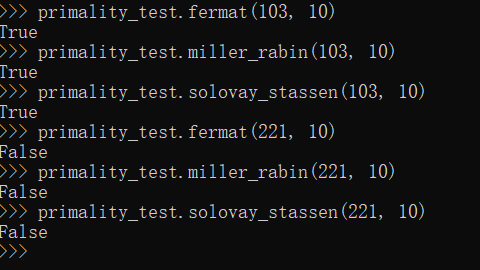
### Solovay-Stassen素性测试算法

每轮迭代需要执行一次模幂算法和一次求Jacobi符号算法。由二次互反律可知，求Jacobi符号的时间复杂度为。由于求Jacobi符号只需要存放要求的结果，因此空间复杂度为。因此每轮迭代的时间复杂度为，空间复杂度为。

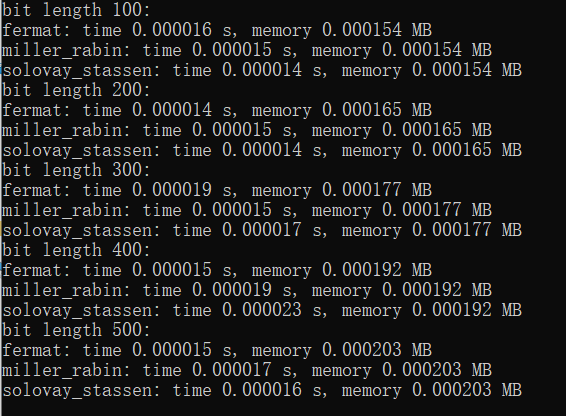
总时间复杂度为，空间复杂度为。

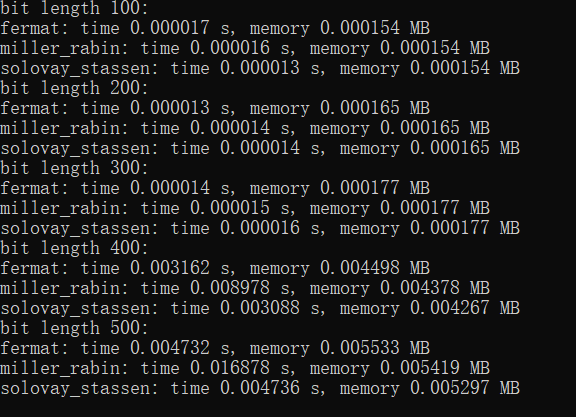
## 测试

首先测试三种素性测试算法的正确性。分别选择和进行测试。已知103为素数，201为合数。迭代次数为10次。



然后比较三种素性测试算法的性能表现。通过随机生成大整数来比较性能表现。迭代次数为20次。





可以看出，在比较实用的随机整数位长（100到500位）下，若测试的数据是合数，三种算法都有比较优秀的性能表现，但是需要的时间都太短，无法通过数据估计出时间复杂度随的变化关系。随着位长线性增大，需要的内存也线性增大，这与理论分析相符。在随机整数位长比较大时，Miller-Rabin素性测试算法需要的时间略多。这也与理论分析相符。

若测试的数据可能是素数，Miller-Rabin素性测试算法明显比另外两种算法需要的时间多，这是因为需要的测试次数较多，使得算法需要的时间更符合对最坏情况算法时间复杂度的分析，这也与理论分析相符。而且同样与理论分析吻合的是三种素性测试算法各自需要的时间和内存随位长的线性变化规律。

## 优化

### 通用

实际上，这三种概率性素性测试算法都可以归纳出这样的步骤：如果某步返回了“某种情况”，就不需要判断了，直接返回“对应的情况”。所以最简单的优化思路都是若某步发现了能提前返回的情况，就直接返回。这样能降低平均情况的时间复杂度。

由于每一次迭代之间数据没有相互依赖，因此可以并行计算每一次迭代的结果，再综合。也可以用线程通信的方式，在某个线程计算出可以直接返回结果的情况时，通知其它线程停止运算，以及通知主线程返回对应的值。

### Miller-Rabin素性测试算法

其实在原来的算法中，计算了两次。可以只计算一次再判断余数，然后不停地对原来的平方，即可求出各。

# 有限域上的四则运算

## 原理

对有限域，一种比较简明的方法是把每个元素用上的次数不超过的多项式表示。这样只需要用位即可表示一个多项式，而且加减运算可以用按位异或实现。

但是这些多项式若要构成一个域，必须定义乘法运算，且这些多项式去掉零元关于乘法构成一个交换群。这可以通过规定和的乘法为实现，其中，不可约且。这里的乘法是常规的上的多项式乘法。抽象代数中可以证明这样的定义可以使这些多项式去掉零元关于乘法构成一个交换群。

要想实现乘多项式，首先可以实现乘以。根据多项式乘法和位运算的规则，若最高次项不为0，直接乘以即可；否则乘以以后要减去。由中加法运算的性质，减去一遍和得到的结果相同。这样就定义了乘以。乘以任意一个多项式，可以根据它的次数和系数，先把原来多项式乘以的结果算出来（为任意整数），再把相应的结果加起来。这样就实现了多项式乘法。

求多项式的乘法逆元可以查表，也可以通过群的性质。若用多项式表示，去掉零元后有个元素。对，其中为的乘法单位元。这样与互为乘法逆元，能在固定时间内求出的乘法逆元。

除某个多项式，只需要乘以它的乘法逆元即可。

上面提到的多项式，可以给定，也可以使用常用的多项式。和中默认的多项式分别为和。

## 伪代码

以下的伪代码都省略了相应函数的参数检查。的相关代码与的类似，因此只写出的代码。

### 加减法

**def** gf\_2\_4\_add(a: int, b: int) -> int:

**return** a ^ b

gf\_2\_4\_sub = gf\_2\_4\_add

### 乘以

**def** gf\_2\_4\_mult\_x(a: int, prim\_poly: int=GF\_2\_4\_PRIM\_POLY) -> int:

**if** a的最高位为 0:

**return** a << 1

**else**:

**return** a左移后减去prim\_poly的结果

### 乘法

**def** gf\_2\_4\_mult(a: int, b: int, prim\_poly: int=GF\_2\_4\_PRIM\_POLY) -> \

int:

result = 0x0

a\_mult = a

从低到高遍历b的每一位:

**if** 该位为1:

result = result + a\_mult

a\_mult乘以x

**return** result

### 求逆元

**def** gf\_2\_4\_get\_inverse(a: int, prim\_poly: int=GF\_2\_4\_PRIM\_POLY) -> \

int:

**return**

### 除法

**def** gf\_2\_4\_div(a: int, b: int, prim\_poly: int=GF\_2\_4\_PRIM\_POLY) -> \

int:

**return** 乘以

## 分析

### 加减法

由于只执行一次异或运算，所以时空复杂度都为。

### 乘以

由于最坏情况下的位运算次数也是确定的，只需要存放结果的额外空间，所以时空复杂度都为。

### 乘法

由于乘以的次数和每次的最大运算次数都是确定的，只需要存放结果的额外空间，所以时空复杂度都为。

### 求逆元

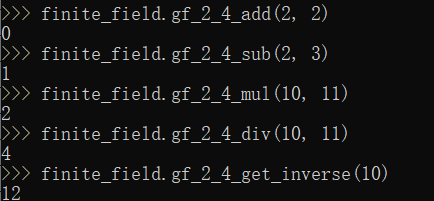
由于运算次数确定，只需要存放结果的额外空间，所以时空复杂度都为。

### 除法

由于乘法和求逆元时空复杂度都为，而除法只用到这两种运算各一次，因此时空复杂度都为。

## 测试

选择进行测试。



经过验算，可以得出结果都正确。

## 优化

### 通用

其实所有的运算，都可以通过查表的方式完成，这需要一定的空间，也可以把运算时间减小。但是对于加减法没必要这么做，因为异或运算已经足够快了，而且可以在数字电路中比较方便地实现。这样做的缺点是无法改变本原多项式，但是其实这在实践中并不是一个缺点，因为常见的使用有限域的加密标准一般都把要用哪个多项式作为本原多项式规定好了。

但是查表应该让表做得尽量小（或者尽量避免让表跨内存页或其它内存调度的基本单位），否则现代处理器内部以及操作系统的内存调度机制可能会起作用，形成一个侧信道攻击面。

### 乘以

对乘以这个运算来说，其实不应该一位一位做判断，而更应该查表。因为有限域上的运算通常被用在加密算法中，必须减小旁路攻击的攻击面，这属于预防被动攻击。一位一位判断会使处理器的分支预测器进行预测，从而使加密每个数字的时间存在差异，这种情况可能会导致泄露关于加密内容的信息。因此，查表是个更好的方法。

跟查表同等地防止类似测信道攻击的一种方法是采用某些位运算技巧，但是并没有查表简单和容易理解。

### 乘法

对乘法这个运算来说，也有对乘数一位一位判断的操作，也可能会暴露类似的攻击面。解决方法和乘以也类似。

### 求逆元

求逆元更实际的方法是查表而不是求，因为本原多项式在密码算法的实践中规定好了。但是还有一种优化是通过逻辑化简来实现，这里可以使用计算机自动计算出逻辑表达式，并实现出组合逻辑数字电路。缺点是同样可能受到侧信道攻击，尤其是电路设计人员经验不足的时候。

# 求模的原根

## 原理

因此判断是否有原根，只需要判断是否为4种规定的形式即可。判断为2或4很简单，但是判断是否符合的格式需要一些操作。注意到，因此可以先检查是否为整数，再检查它是否为素数。检查是否为素数，用到了素性测试算法。考虑到平时应用的数位数一般为400十进制位左右，20次的素性测试迭代次数比较适当。

检查是否为的原根，可以用检验。设的所有质因子为，若，则，不是的原根。该命题的逆命题易知也成立。

遍历所有可能的，就能求出所有原根。

## 伪代码

### 获得关于原根的信息

# 返回格式为 (是否有原根, 是否为素数的幂, , )

**def** get\_prim\_root\_stats(n: int) -> tuple:

为偶数时的三种情况判断

result = (**None**, **None**, **None**, **None**)

**if** n % 2 == 0:

to\_test = n // 2

**else**:

to\_test = n

对从到1的每个:

若为整数且通过了素性测试检验:

result[0] = **True**

result[1] = (n % 2 == 0)

result[2] =

result[3] =

**return** result

**return** (**False**, **False**, **None**, **None**)

### 求原根列表

**def** get\_prim\_root\_list(n: int) -> int:

stats = get\_prim\_root\_stats(n)

p = stats[2]

l = stats[3]

result = []

**if not** stats[0]:

返回错误

处理为偶数的情况

得到

对2到之间的所有:

若:

判断下一个

否则:

result.append()

return result

### 求最小的原根

**def** get\_prim\_root(n: int) -> int:

stats = get\_prim\_root\_stats(n)

p = stats[2]

l = stats[3]

result = []

**if not** stats[0]:

返回错误

处理为偶数的情况

得到

对2到之间的所有:

若:

判断下一个

否则:

返回

返回错误

## 分析

假设素性检验采用Miller-Rabin素性检验算法，检验次数为常数20。

### 获得关于原根的信息

对每个接受测试的，求出和判断其为整数的时空复杂度都是，素性检验的时间复杂度为，空间复杂度为。

对所有的，的范围为，因此总时空复杂度为。

### 求原根列表

由为奇素数知，，因此对进行质因数分解时可以只分解的质因数，这就大大降低了运算量。幂运算也可以采用类似平方快速幂的算法，而且求出即可求出，因此求的时间复杂度和空间复杂度都为.

可以假设求的质因数是采用朴素的从到的枚举求法，这样最坏情况下时间复杂度是。

所有的不同因数个数是不知道的，但是至少有。若一个数的不同因数个数为，则由质数分布规律，第个质数可以认为大约是。它的最小值是前个质数相乘的积，数量级大约是。而的因数个数依赖于的因数个数。可以证明，最多有个不同的质因数，因此也有个不同的质因数。

一共有个，每个最多要做次幂乘。因此做幂乘的时空复杂度都为。

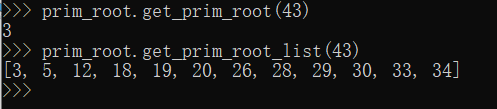
因此总时间复杂度为，空间复杂度为。这里得到的时空复杂度并不准确，因为做幂乘的最多次数和之间有联系，这并没有在对求和时表现出来。

### 求最小的原根

分析类似上面，但是时空复杂度更不准确，因为只返回一个原根，而数值最小的原根与本身的关系并不明确。时空复杂度与上面相同。

## 测试

以为例。



经过验算，结果正确。

## 优化

### 获得关于原根的信息

由于，所以可以缓存。这样能减少约一半的计算量。

### 求原根列表

求的所有因数时只需求的所有因数即可，因为。同样根据上式，可以通过一次平方幂运算快速求出。

求时，若为形式，由可得，这种情况也能化归成求的情况。

模的原根必定满足与互质，因为，通过这一点可以简化计算。

# 求上的本原多项式

## 原理

上的本原多项式就是各系数且互素的多项式。所以求上的本原多项式可以简化为求一组互素的系数。这里可以随机生成一组系数，然后剔除不互素的；也可以采用模板生成本原多项式。

## 伪代码

### 随机生成法

**def** z\_x\_gen\_prim\_poly\_random (deg: int, max\_coeff: int) -> list:

随机在到范围内生成系数

**return** [(0, ), (1, ), ,(deg – 1, ), (deg, 1)]

### 模板生成法

**def** z\_x\_gen\_prim\_poly\_simple(deg: int) -> list:

**return** [(0, 2), (deg, 1)]

## 分析

### 随机生成法

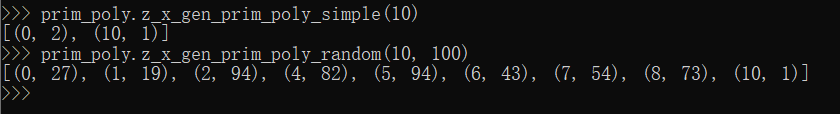
时空复杂度显然为。

### 模板生成法

显然时空复杂度为。

## 测试

生成次数为10，最大系数为100的本原多项式。



经过验算，结果正确。

# 心得体会

## 素性测试算法

素性测试算法其实我以前接触过一两种，就是没有真的去编程实现它。这次实现了素性测试算法，我感觉在实践中完全是另外一个境界，因为这个算法是一个随机算法，输出不仅与输入有关。

## 有限域上的四则运算

有限域上的四则运算其实比较简单，只要能弄明白把用多项式表示具体是怎么回事。但是实际上实现四则运算，还是要有一定妥协的，比如时间和空间的妥协。

## 求模的原根

求原根这个算法，想是好想，主要是效率不高。我想出了几种优化的方法，能够减少计算复杂度的常数部分，有一两个地方降低了复杂度的量级，但是我感觉好像还不一定够。

求出时空复杂度，是我最不会的地方。主要还是因为对素数的分布和函数的不了解。给出的上界也比较松散。

## 求上的本原多项式

求本原多项式，根本上是生成一组互质的数。可以用随机生成再删除的办法，也可以用从比较小的多项式构建的方法。实际上随机生成再删除的方法，也是一种随机算法。

## 算法评估与优化

这次的算法时空复杂度分析更加复杂，比上次更难。优化也是这样，必须要抓住算法的计算本性，才能更好地优化，比如求原根中对的质因数分解。

## 系统设计与维护

这次有效的错误机制也发挥了作用，在写新代码以及调试的过程中，代码出错了，本层代码和上层代码的错误机制都能有效地警告，避免隐性bug的出现。

良好的系统设计也让增加新功能变得简单。这次我准备加入上多项式运算的功能，因为当时并没有把上的本原多项式写成模数，所以直接改变变量名字和函数声明就可以了。我写求原根代码的时候，想加入一个辅助函数判断是不是模的原根，利用已知的函数逻辑，大约10行代码就可以实现了。想要评测素性测试代码，利用写好的评测模块，10行有效代码就能完成评测。所以关注以下系统的可维护性还是有意义的。

## 对课程的建议

感觉下次实验目标可以明确一点，因为这次我看到“本原多项式”这个题目，不知道是求哪个环上的本原多项式，问了老师才知道是上的。

然后我感觉可以先把题目公开，然后再讲后面的基础讲解，或者也可以先说一下我们的实验情况，这样我们就能有个目标了。再就是因为一些数学上的基础支持，我们在上学期其实已经接触过了。

我的最后一个建议就是助教学长检查我们好像有点慢了……可以利用现成的OJ系统自动评测，或者让我们写个简短的评测代码，写完之后运行，助教学长看一下结果，然后对原理提问，这样速度可能会快一点。

## 总结

这次的密码学实验难度更大，越来越具有挑战性了，但是我也感觉更有意思了。这也算是我们学习的Python第一次用在实践当中，其实挺刺激的。我会不怕困难，继续面对接下来的挑战。