

AKS算法

黑帆

PRIMES is in P

近几年最著名的就算**AKS**算法了。由三个印裔的哥们设计的（算法的名称取自他们的首字母）。所以就在国庆期间翻译了这篇论文。中文翻译过来大意叫做**在多项式时间内的素性确定算法**。其实算法本身并不复杂，理解起来也不难。但是其中有一个多项式消减问题。国庆期间大概花了一个小时左右写好了算法流程本身。然而多项式消减这个操作我陆陆续续写了两周

左右。

其实本来也花不到那么长的时间，只是范了**程序员**的通病，慢慢的把一个小程序越写越大，到基本功能完成时写了**2000**行左右的**python**代码（如果用**C/C++**的话，应该在**3000**行左右）。实现了一个简易的符号系统（其中对外的输出使用**Latex**的数学公式表示），可以做**二项式展开**，**多项式合并**，以及**多项式的加减乘除**操作。在上周天的感觉在数值为分数时**自动配方**不是很好，又重写了一个叫做**yvalue**的数值类，把浮点数与整数在保存时，就转化成分数的形式。在求值时才展开。还可以处理类似 $\frac{15}{\frac{5}{7}}$ 这样的形式。连分数的处理是OK的。

多项式的消减的逻辑还是很烧脑的。本来**AKS**只需要一个一元**n**次的展开即可，出于一个**程序员的态度**，我还是完成了**多元多次式**的运算。下图显示了。**P2 (mod P3)**后的商式**P4_q**与余式**P4_r**。其中**P2**是 $(x + a)^7$ 展开后的结果。

```
C:\Users\devilogic\Anaconda3\python.exe C:/Users/devilogic/workspace
P2 = x^7+7x^6a+21x^5a^2+35x^4a^3+35x^3a^4+21x^2a^5+7xa^6+a^7
P3 = x^5-1
P4_q = 21a^2+x^2+7ax
P4_r = a^7+7xa^6+21x^2a^5+35x^4a^3+35x^3a^4+21a^2+x^2+7ax
```

下图是我实现的**AKS**算法的流程输出，感觉可以当做教学课件了。现在网上基本找不到**AKS**的完整真正实现，有些实现都是直接把**X**代入一个固定的数值，这样其实不能算是**AKS**算法，只能算是该算法的一个应用。这样在数值小的时候没问题，当要确定的一个数过大时就没有用了。其中多项式消减操作就是为了避免处理数值过大后大数运算带来的效率问题。

下图是程序的输出截图：

```
C:\Users\devilogic\Anaconda3\envs\python34\python.exe C:/Users/devilogic/workspace/hacker/hkfiction/aks.py
检查'31'是否为素数
步骤1, 确定31是否是纯次幂
步骤2, 找到一个最小的r, 符合o_r(31) > (log31)^2
r = 29
步骤3, 如果 1 < gcd(a, 31) < 31, 对于一些 a <= 29, 输出合数
步骤4, 如果n=31 <= r=29, 输出素数
步骤5
遍历a从1到\sqrt(\phi(r=29))logn=31$
如果(X+a)^31 != X^31+a mod {X^29-1, 31}$输出合数
构造多项式(X+a)^31, 并且进行二项式展开
X^31+31X^30a+465X^29a^2+4495X^28a^3+31465X^27a^4+169911X^26a^5+736281X^25a^6+2629575X^24a^7+7888725X^23a^8+20
构造消减多项式 X^29-1
进行运算 (X+a)^31 mod (X^29-1)
得到余式: a^31+31Xa^30+465X^2a^29+4495X^28a^3+4495X^3a^28+31465X^27a^4+31465X^4a^27+169911X^26a^5+169911X^5a^2
进行运算'余式' mod 31 得到式(A)
A = a^31+X^2
B = x^31+a mod x^29-1
B = X^2+a
C = A - B = a^31-1a
遍历a = 1 to 26
检查每个'a^31-1a = 0 (mod 31)'
检查a = 1
检查a = 2
检查a = 3
检查a = 4
检查a = 5
检查a = 6
检查a = 7
检查a = 8
检查a = 9
检查a = 10
检查a = 11
检查a = 12
检查a = 13
检查a = 14
检查a = 15
检查a = 16
检查a = 17
检查a = 18
检查a = 19
检查a = 20
检查a = 21
检查a = 22
检查a = 23
检查a = 24
检查a = 25
检查a = 26
步骤6 输出素数
YES
```

这份代码中，需要使用大数运算库 `gmpy2` 的支持。运行环境为 `python34` 和 `python27`。3.6目前不支持原因在于 `gmpy2` 不支持3.6的版本。另外还需要说明的是程序有 `bug` 没有修订，`gmpy2` 没有在所有的运算中使用，空了时间再填补上吧。

下面译文还有一些需要说明的是有些是我自己的注释，使用以下这种方式：

这里是我的注释

1.介绍

由于在现代密码学中需要非常大的素数来参与运算，所以素数的判定非常的重要。

我们使用**PRIMES**来代替素数集合。素数定义本身已经给出了一种算法来确定一个整数 $n \in PRIMES$ 。尝试使用 n

去除以每个整数 $m \leq \lfloor \sqrt{n} \rfloor$ ，如果有任意 m 可以整除 n 。那么 n 为合数。否则它是一个素数。

$m \leq \lfloor \sqrt{n} \rfloor$ 。为什么 $\lfloor \sqrt{n} \rfloor$ 是它的界限。一个很简单
的证明是，一个合数是最少两个素数相乘组成的。设 $n = pq$ ，其中 $p = q$ ，
那么设 $p = q = \sqrt{n}$ ；如果 $p < q$ ，那么 $p < \lfloor \sqrt{n} \rfloor$ 。如果假设
 $p > \lfloor \sqrt{n} \rfloor$ 。那么 $pq > n$ 。所以与条件矛盾。

这种测试算法在古希腊被称为 **埃拉托色尼** [1] (ca. 240 BC) 分解法。它需要遍历所有小于 \sqrt{n} 的素数，这很无效，需要花费大概 $\Omega(\sqrt{n})$ 步去决定 n 是否是一个素数。一个更加有效的测试是基于 **费马小定理** [2]：对于任意素数 p 与一个任意不能整除它的整数 a ，存在 $a^{p-1} \equiv 1 \pmod{p}$ 。给定一个整数 a 去确定整数 n 是否是一个素数时，需要对 a 反复进行 $(n - 1)^{th}$ 次平方，但是，它仍然不是一个总是正确的测试，因为有些 a 在 n 是合数是也可以通过这个测试(这些 a 被称作 **卡米切尔斯** [3])。无论如何，费马小定理都是有效素性测试算法的基础。

直到在1960年，**复杂理论** [4] 出现，从而将**问题的复杂性**进行标准化并且将**复杂性问题的种类**进行了定义(例如素性测试就是一个复杂性问题)。**素性测试**才被重视起来。开始这个问题被认为是一个 $co - NP$ [5] 问题：如果 n 不是一个素数，那么 n 拥有一个非平凡因子将很容易的被验证。在1974年，Pratt认为这个问题是一个 NP 问题(因此将它放入 $NP \cap co - NP$)。

在1975, Miller 使用一个基于费马小定理的属性去证明**扩展黎曼假设** [6]。很快的他的测试被Rabin修改为一个无条件但是是随机多项式时间的算法。在1974年索洛韦和Strassen也独立的提出了一个不同的随机多项式时间算法，这个算法使用了一个素数 n ，对于每一个 a 存在 $\left(\frac{a}{n}\right) = a^{\frac{n-1}{2}} \pmod{n}$ (($-$) 是雅可比符号 [7])。他们的算法在**扩展黎曼假设**的条件下也可以直接确定素数。后来有基于不同特性的随机多项式时间测试素数的算法。

在1983年，Adleman，Pomerance，与 Rumely 在素性测试方面取得了一个决定性的进展，这个算法可以运行在 $(\log n)^{O(\log \log \log n)}$ 时间(之前所有的算法都是在指数时间)。他们的算法(感觉上)是Miller算法的更加通用的形式并且使用更高级的相互作用规则。在1986年，Goldwasser与Kilian提出一个基于椭圆曲线 [8] 在大多数输入(所有的输入都被认为是合乎条件的)条件下可以运行在多项式时间。这使素性测试变得很容易(指导那时，所有的随机算法产生的验证结果都仅仅是复合性的)。基于他们的想法，一个类似的算法被Aktin发明。Adleman与Huang修改Goldwasser-Kilian的随机算法期望在所有的输入条件下运行在多项式时间。

一个无条件多项式时间确定素数的算法是这个研究的最终目标。尽管到现在这项研究已经取得了极大的成绩，但是离最终目标还有段距离。在这篇论文中，我们达到了最终目的。我们在确定一个数是否是素数在 $O \sim (\log^{15/2} n)$ 时间内。在一些条件下，我们的算法会做的更好：普遍认为在苏菲日耳曼素数 [9] 的密度下(如果素数 p 那么 $2p + 1$ 也是素数)，我们的算法仅花费 $O \sim (\log^6 n)$ 步。我们的算法是基于在有限域多项式环的费马小定理。证明我们定理的正确性只需要一些简单的代数功能(除了在 p 与 $p - 1$ 之间有一个非常大的素因子的筛选法的结果并且甚至不需要证明这个算法是工作在 $O \sim (\log^{21/2} n)$ 这个很短的时间内)。之前的算法证明要复杂的多。

在第二节，我们简单阐述了这个算法的核心思想。在第三节，我们明确了一些标记的使用。在第四节，我们证明了这个算法的正确性。在第五个小节，我们分析这个算法的性能。在第六节我们探讨对这个算法优化的方法。

其实这节就是介绍一下多项式时间确定素数算法的在历史上的取得的进展。

2.核心思路

我们的测试是基于**费马小定理**。它也是随机多项式时间算法的基础。

这里我们先花些时间来探讨下**费马小定理**，这是很有必要的。

如果 p 是一个素数，那么对于所有的整数 a ，数 $a^p - a$ 是 p 的整数倍。又可以表示为：

$$a^p \equiv a \pmod{p}$$

举例说明，如果 $a = 2$ 并且 $p = 7$ 那么 $2^7 = 128$ ，并且 $128 - 2 = 7 \times 18$ 。

如果 a 与 p 互素，上边的等式可以两边消去一个 a 则**费马小定理**的形式为

$$a^{p-1} \equiv 1 \pmod{p}$$

举例说明，如果 $a = 2$ 并且 $p = 7$ ，那么 $2^6 = 64$ 并且 $64 - 1 = 63$ 是7的整数倍。

以上就是**费马小定理**的基本内容。论文中的第二节就是其使用**多项式理论**对其证明的一个过程。

引理 2.1 使 $a \in \mathbb{Z}, n \in \mathbb{N}, n \geq 2$ 并且 $(a, n) = 1$ 。当且仅当

$$(X + a)^n \equiv X^n + a \pmod{n} \quad (1)$$

则 n 是素数。

其实以上就是**费马小定理的一般化形式**。

证明 循环遍历 $0 < i < n$ ， x^i 的系数在 $((X + a)^n - (X^n + a))$ 是 $\binom{n}{i} a^{n-i}$ 。

忘记组合公式的同学看这里： $\binom{n}{i} = \frac{n!}{i!(n-i)!}$

如果 n 是一个素数。则 $\binom{n}{i} \equiv 0 \pmod{n}$ 因此所有的系数为0。

个人比较讨厌，**没有来由的数学等式**，那就让我们展开来讨论一下这个等式。在探讨之前先把二项式定理放到前面膜拜一下牛顿老人家。

二项式展开式定理：令 x, y 为实数， n 是一个正整数。则

$$(x + y)^n = \sum_{i=0}^n \binom{n}{i} x^i y^{n-i}$$

展开等式(1)的左边得到如下的等式：

$$C_n^0 a^0 X^n + C_n^1 a^1 X^{n-1} + \dots + C_n^{n-1} a^{n-1} X + C_n^n a^n X^0$$

其中除了第一项与最后一项都不能被 n 整除外，其余的项当模 n 时，它们都被削减掉了。当削减掉中间的项后等式变成剩下的就是 $X^n + a^n$

如果 n 是一个合数。假设 n 的一个素因子 q 并且使得 $q^k \mid n$ 。然而 q^k 不能整除 $\binom{n}{q}$ 并且与 a^{n-q} 互素，因此 X^q 的系数非 0 ($\pmod n$)。因此 $((X+a)^n - (X^n + a^n))$ 在模 n 的环 [10] Z_n 中为非 0。

以上的等式暗示了一个简单的素性测试方法：给定一个输入 n ，选择一个数 a 并且测试等式(1)是否相等。但是，这需要花费 $\Omega(n)$ 。

$\Omega(n)$ 表示对于任意一个 $c \geq c_0$ ，则 $c_0 n \leq f(n)$ 。

因为上述等式的系数项有 n 个，最坏的情况是当模 n 后(此时， n 是一个合数)，每个系数项都为非 0。

因为在最坏的情况下计算左边等式 n 个系数。一个简化方法是等式(1)两边都模一个 $X^r - 1$ 多项式，其中的 $r < n$ 。等同于测试如下等式：

$$(X+a)^n = X^n + a \pmod{X^r - 1, n} \quad (2)$$

根据引理 2.1，对于所有的 a 与 n ，当 n 为素数时都满足等式(2)。现在的问题是有很少的 a 与 r 在 n 为合数时也满足此等式。

总体来说就是：我们选择一个合适的数 r ，如果等式(2)满足，那么对于一些少量的 a ， n 是合数，对于大量的 a 与一个合数 r 关于在 $\log n$ 多项式时间有界，因此我们获得一个在多项式时间内确定素数的算法。

3. 标记与预读

关于 P , NP 以及 $co - NP$ 问题的详细解释参见参考资料。

用 Z_n 表明模 n 的环。 F_p 是模 p 的有限域 [11]，其中 p 是素数。

回想一下如果 p 是素数， $h(X)$ 是一个在 F_p 域的 d 次可削减多项式，那么 $F_p[X]/(h(X))$ 是以 p^d 为阶的有限域。我们将使用 $f(X) = g(X) \pmod{h(X), n}$ 来表示在环 $Z_n[X]/(h(X))$ 内的等式 $f(X) = g(X)$ 。

上述等式就是表示 $f(X)$ 与 $g(X)$ 首先对模 $h(X)$ 做削减操作，随后在做模 n 的操作，保证结果在环 Z_n 中。

我们使用符号 $O(t(n))$ 来表示 $O(t(n) \cdot \text{poly}(\log t(n)))$ ，其中 $t(n)$ 表示变量 n 的函数。例如， $O(\log^k n) = O(\log^k n \cdot \text{poly}(\log \log n)) = O(\log^{k+\epsilon} n)$ 对于任意一个 $\epsilon > 0$ 。我们使用 \log 表示以 2 为底的对数，自然对数使用 \ln 来表示。

\mathcal{N} 与 \mathcal{Z} 分别表明自然数集与整数集。给定 $r \in \mathcal{N}$, $a \in \mathcal{Z}$ ，其中 $(a, r) = 1$ ，其中 k 是满足元素 a 模 r 群的最小的阶， $a^k \equiv 1 \pmod r$ 。它使用 $o_r(a)$ 来表示。对于 $r \in \mathcal{N}$ ， $\phi(r)$ 是欧拉函数表示满足所有小于 r 的素数的个数。其

中对于每一个 a , $(a, n) = 1$, 则 $o_r(a) \mid \phi(r)$ 。

引理3.1 使用 $LCM(m)$ 表示前 m 数的最小公倍数。对于 $m \geq 7$:

$$LCM(m) \geq 2^m$$

4. 算法与证明

输入一个大于1的正整数 n 。

1. 如果对于每一个 $a \in \mathcal{N}$ 与 $b > 1$, 则 $n = a^b$, 输出合数。
2. 找到一个最小的整数 r , 使得 $o_r(n) > \log^2 n$ 。
3. 如果对于一些 $a \leq r$, 则 $1 < (a, n) < n$, 输出合数。
4. 如果 $n \leq r$, 输出素数。
5. 循环测试每一个 $a = 1$ 到 $\lfloor \sqrt{\phi(r)} \log n \rfloor$, 如果 $(X + a)^n \not\equiv X^n + a \pmod{X^r - 1, n}$ 输出合数。
6. 输出素数。

定理 4.1 以上算法返回素数, 当且仅当 n 是素数。

在以下得小节中, 我们应用这个定理到所有得引理中。下面得定理就是一个简单得应用。

引理 4.2 如果 n 是一个素数, 以上算法将返回素数。

证明: 如果 n 是一个素数, 那么步骤1到3将绝不会返回合数。通过引理 2.1, 进入到循环中也不能返回合数。因此算法将返回素数在步骤4或者步骤6。

以上引理的反向证明需要花一点功夫。如果算法在步骤4返回素数, 那么前3个步骤没有都发现 n 的非平凡因子。因此仅剩一种情况是步骤6。

这个算法拥有两个重要的步骤(2与5): 步骤2中寻找一个合适的 r 而步骤5验证整数 a 是否符合等式(2)。在此之前我们首先要确定 r 的量级。

步骤2与5是这个算法的一个难点, 所以作者帮忙找出了一个很合适 r 值,
就是引理 4.3 提出的理论。

引理 4.3^[12] 存在 $r \leq \max\{3, \lceil \log^5 n \rceil\}$ 则 $o_r(n) > \log^2 n$ 。

证明: 当 $n = 2$, $r = 3$ 时, 满足所有条件。当 $n > 2$ 时, 那么 $\lceil \log^5 n \rceil > 10$ 并且根据引理 3.1。观察到对于一个最大的 k 值对于任意 $m^k \leq B = \lceil \log^5 n \rceil$, $m \geq 2$, 是 $\lfloor \log B \rfloor$ 。现在考虑将不下式整除的最小的 r

$$n^{\lfloor \log B \rfloor} \cdot \prod_{i=1}^{\lfloor \log^2 n \rfloor} (n^i - 1)$$

根据以上的观察 (r, n) 将不能被任何 r 的素因子整除, 除非 r 可以被 $n^{\lfloor \log B \rfloor}$ 整除。因此 $\frac{r}{(r, n)}$ 也不能被以上结果整除。 r 是

最小的不能被上述结果整除的值，由此可见 $(r, n) = 1$ 。更进一步的，由于对于每个 $1 \leq i \leq \lfloor \log^2 n \rfloor$, r 都不能被 $n^i - 1$ 整除以及 $o_r(n) > \log^2 n$ 。所以，

$$n^{\lfloor \log B \rfloor} \cdot \prod_{i=1}^{\lfloor \log^2 n \rfloor} (n^i - 1) < n^{\lfloor \log B \rfloor + \frac{1}{2} \log^2 n \cdot (\log^2 n - 1)} \leq n^{\log^4 n} \leq 2^{\log^5 n} \leq 2^B$$

(保证第二个不等式的所有的 $n \geq 2$)，根据引理 3.1，数 B 的第一个最小公倍数至少是 2^B 。因此， $r \leq B$ 。

由于 $o_r(n) > 1$ ，必然存在 n 的一个素因子 p ，使得 $o_r(p) > 1$ 。由于步骤3或者步骤4可以确定 n 的素性。因此，我们有 $p > r$ 。由于 $(n, r) = 1$ (步骤3或者步骤4将确定 n 的类型)， $p, n \in Z_r^*$ 。在这节剩余的部分将讨论数 p 与 r 的问题。而且，使得 $l = \lfloor \sqrt{\phi(r) \log n} \rfloor$ 。

步骤5用于验证等式 l 。由于在这个步骤算法不输出合数，所以，我们有：

$$(X + a)^n \equiv X^n + a \pmod{X^r - 1, n}$$

对于每个 $a, 0 \leq a \leq l$ (如果 $a = 0$ ，那么等式必然成立)。意味着：

$$(X + a)^n \equiv X^n + a \pmod{X^r - 1, n} \quad (3)$$

对于每个 $0 \leq a \leq l$ ，根据引理 2.1，我们有：

$$(X + a)^p \equiv X^p + a \pmod{X^r - 1, p} \quad (4)$$

对于每个 $0 \leq a \leq l$ ，根据等式(3)与等式(4)：

$$(X + a)^{\frac{n}{p}} \equiv X^{\frac{n}{p}} + a \pmod{X^r - 1, p} \quad (5)$$

对于每个 $0 \leq a \leq l$ ，在上述等式中， n 与 $\frac{n}{p}$ 两者的表现都与素数 p 一样。我们给这种属性取个名称：

定义 4.4 对于一个多项式 $f(x)$ 与一个数 $m \in \mathcal{N}$ ，我们称 m 是内省的(introspective)对于 $f(x)$ 当：

$$[f(X)]^m \equiv f(X^m) \pmod{X^r - 1, p}$$

当 $0 \leq a \leq l$ 对于 $X + a$ 在等式(5)与等式(4)中 $\frac{n}{p}$ 与 p 是内省的，还是比较清晰的。

下面的引理展示了内省数是符合乘法法则的。

引理 4.5 如果 m 与 m' 对于 $f(X)$ 是内省的，那么 $m \cdot m'$ 也是。

证明：由于 m 对于 $f(X)$ 是内省的，我们有：

$$[f(X)]^{m \cdot m'} \equiv [f(X^m)]^{m'} \pmod{X^r - 1, p}$$

同样，由于 m' 对于 $f(X)$ 是内省的，我们有 (将 X 替换成 X^m)：

$$[f(X^m)]^{m'} = f(X^{m \cdot m'}) \pmod{X^{m \cdot r} - 1, p} \quad (1)$$

$$= f(X^{m \cdot m'}) \pmod{X^r - 1, p} \quad (2)$$

上式中 $X^r - 1$ 可被 $X^{m \cdot r} - 1$ 整除。

合并上面两个等式，我们可以得到：

$$[f(X)]^{m \cdot m'} \equiv f(X^{m \cdot m'}) \pmod{X^r - 1, p}$$

所以多项式的内省数 m 是符合乘法原则的。

引理4.6 如果内省数 m 对于每个多项式 $f(X), g(X)$ 成立，那么对于它们的积也成立。

证明：

$$[f(X) \cdot g(X)]^m = [f(X)]^m \cdot [g(X)]^m \quad (3)$$

$$= f(X^m) \cdot g(X^m) \pmod{X^r - 1, p} \quad (4)$$

上述两个引理意味在每个在集合 $I = \{(\frac{n}{p})^i \cdot p^j \mid i, j \geq 0\}$ 中的数对于多项式集合

$P = \{\prod_{a=0}^l (X + a)^{e_a} \mid e_a \geq 0\}$ 都成立。现在我们定义基于以上两个集合的两个群。

第一个群是 I 模 r 的所有剩余类。由于通过观察， $(n, r) = (p, r) = 1$ ，因此它是 Z_r^* (模 r 的既约剩余类) 的一个子群。设 G 来表示这个群并且 $|G| = t$ (表示群 G 的数量)。 G 是通过 n 与 p 模 r 产生的，所以其阶 $o_r > \log^2 n$ 则 $t > \log^2 n$ 。为了定义第二个群，我们需要一些关于在有限域上的分圆多项式的基础知识。

使用 $Q_r(X)$ 来表示在 F_p 上的 r 次分圆多项式。多项式 $Q_r(X)$ 可以被 $X^r - 1$ 整除并且其因子是阶为 $o_r(p)$ 的不可约因子。让 $h(X)$ 表示这样一个不可约因子。由于 $o_r(p) > 1$ ，所以 $h(X)$ 的阶大于 1。第二个群是所有在 P 中模 $h(x)$ 与 p 可被消减的多项式。使用 \mathcal{G} 来表示它。这个群是通过在域 $F = F_p[X]/(h(X))$ 中的多项式 $X, X + 1, X + 2, \dots, X + l$ 产生的，并且是群 F 的整数倍的子群。

下面的引理证明了群 \mathcal{G} 长度的一个下界。

引理4.7 (Hendrik Lenstra Jr.) $|\mathcal{G}| \geq \binom{t+l}{t-l}$

证明：首先要说明的是 $h(X)$ 是分圆多项式 $Q_r(X)$ 的一个因子， X 是在 F 中的 r 次本原单位根。

现在我们的说明的是任意两个在 P 中阶小于 t 的多项式在 \mathcal{G} 中的映射都有不同的值。让 $f(X)$ 与 $g(X)$ 表示在 P 中的这样两个多项式。假设在域 F 中 $f(X) = g(X)$ ，设 $m \in I$ 。我们有在域 F 中 $[f(X)]^m = [g(X)]^m$ 。由于 m 是 f 与 g 的内省数，并且 $h(X)$ 可以被 $X^r - 1$ 整除，所以在域 F 中我们得到：

$$f(X^m) = g(X^m)$$

这意味着对于每个 $m \in \mathcal{G}$ ， X^m 是多项式 $Q(Y) = f(Y) - g(Y)$ 的一个根。由于 $(m, r) = 1$ (G 是 Z_r^* 的一个子群)，每个 X^m 都是 r 次本原单位根。因此在 F 中当 $|G| = t$ 时， $Q(Y)$ 有相异根。可是通过选择 f 与 g 可以使得 $Q(Y)$ 的阶小于 t 。这是这与假设矛盾，在 F 中 $f(X) \neq g(X)$ 。

对于在 F_p 中的每一个 $1 \leq i \neq j \leq l$ ，其中 $l = \lfloor \sqrt{\phi(r)} \log n \rfloor < \sqrt{r} \log n < r$ 并且 $p > r$ 。因此在 F 中 $X, X + 1, X + 2, \dots, X + l$ 两两相异。又因为 h 的阶大于 1，在 F 中对于每个 a ($0 \leq a \leq l$)， $X + a \neq 0$ 。因此在 \mathcal{G} 中至少存在 $l + 1$ 个不同的一阶多项式，因此在 G 中至少存在 $\binom{t+l}{t-l}$ 个相异的阶 $< t$ 的多项式。

在这种情况下， n 不是 p 的纯幂次形式，而 G 的长度的一个上界也可以确定：

引理4.8 如果 n 不是 p 的纯幂次形式，那么 $|G| \leq n^{\sqrt{t}}$

证明，考虑以下 I 的子集：

$$\hat{I} = \{(\frac{n}{p})^i \cdot p^j \mid 0 \leq i, j \leq \lfloor \sqrt{t} \rfloor\}$$

如果 n 不是 p 的纯幂次形式，那么集合 \hat{I} 有 $(\lfloor \sqrt{t} \rfloor + 1)^2 > t$

个，其中每个元素两两相异。由于 $|G| = t$ ，其中至少两个在 \hat{I} 中的数模 r 同余。设 $m_1 > m_2$ 。因此我们有：

$$X^{m_1} \equiv X^{m_2} \pmod{X^r - 1}$$

设 $f(X) \in P$, 那么

$$[f(X)]^{m_1} = f(X^{m_1}) \pmod{X^r - 1, p} \quad (5)$$

$$= f(X^{m_2}) \pmod{X^r - 1, p} \quad (6)$$

$$= [f(X)]^{m_2} \pmod{X^r - 1, p} \quad (7)$$

所以在域 F 中

$$[f(X)]^{m_1} = [f(X)]^{m_2}$$

因此 $f(X) \in \mathcal{G}$ 是多项式 $Q'(Y) = Y^{m_1} - Y^{m_2}$ 在域 F 中的一个根。 $f(X)$ 是 \mathcal{G} 中的任意元素，因此在 F 中多项式 $Q'(Y)$ 至少有 $|\mathcal{G}|$ 个两两相异的根。 $Q'(Y)$ 的阶是 $m_1 \leq (\frac{n}{p} \cdot p)^{\lfloor \sqrt{t} \rfloor} \leq n^{\sqrt{t}}$ 。这说明 $|\mathcal{G}| \leq n^{\sqrt{t}}$ 。

通过预估出来的 $|\mathcal{G}|$ 的规模，我们现在证明算法的正确性：

引理4.9 如果算法返回素数，那么 n 是素数。

证明，假设算法返回素数，根据**引理4.7**意味着对于 $t = |G|$ 与 $l = \lfloor \sqrt{\phi(r) \log n} \rfloor$ ：

$$|\mathcal{G}| \geq \binom{t+l}{t-l} \quad (8)$$

$$\geq \binom{l+1 + \lfloor \sqrt{t} \log n \rfloor}{\lfloor \sqrt{t} \log n \rfloor} (t > \sqrt{t} \log n) \quad (9)$$

$$\geq \binom{2\lfloor \sqrt{t} \log n \rfloor + 1}{\lfloor \sqrt{t} \log n \rfloor} (l = \lfloor \sqrt{\phi(r) \log n} \rfloor \geq \lfloor \sqrt{t} \log n \rfloor) \quad (10)$$

$$> 2^{\lfloor \sqrt{t} \log n \rfloor + 1} (\lfloor \sqrt{t} \log n \rfloor > \lfloor \log^2 n \rfloor \geq 1) \quad (11)$$

$$\geq n^{\sqrt{t}} \quad (12)$$

根据**引理4.8**，如果 n 不是 p 的纯幂次形式，那么 $|\mathcal{G}| \leq n^{\sqrt{t}}$ 。因此对于一些 $k > 0$, $n = p^k$ 。如果 $k > 1$ 那么算法将在第一步返回合数。因此， $n = p$ 。

以上就是整个算法的完整证明。

到这里为止就是**AKS**的原理部分，后边的部分是效率分析与未来改进的地方，没兴趣的可以不做了解了。这里我们稍微做下总结，并且按照程序员的角度一步步的分析并给出 *Python* 语言的实现。

```

1. #!/usr/bin/python
2. # coding:utf-8
3. import numpy as np
4. import math
5. import ysym
6. from ysym.ypolynomial import *
7. import gmpy2
8.
9. def is_prime_by_AKS(n):
10.     """
11.     使用AKS算法确定n是否是一个素数

```

```

12.     True:n是素数
13.     False:n是合数
14.     """
15.
16.     def __is_integer__(n):
17.         """
18.             判断一个数是否是整数
19.         """
20.         i = int(n)
21.         f = n - i
22.         return not f
23.
24.     def __phi__(n):
25.         """
26.             欧拉函数，测试小于n并与n互素的个数
27.         """
28.         res = n
29.         a = n
30.         for i in range(2, a+1):
31.             if a % i == 0:
32.                 res = res // i * (i - 1)
33.                 while a % i == 0:
34.                     a //= i
35.             if a > 1:
36.                 res = res // a * (a - 1)
37.         return res
38.
39.     def __gcd__(a, b):
40.         """
41.             计算a b的最大公约数
42.         """
43.         if b == 0:
44.             return a
45.         return __gcd__(b, a % b)
46.
47.     print("步骤1, 确定%d是否是纯次幂" % n)
48.     for b in range(2, math.floor(math.log2(n))+1):
49.         a = n** (1/b)
50.         if __is_integer__(a):
51.             return False
52.
53.     print("步骤2, 找到一个最小的r, 符合o_r(%d) > (log%d)^2" % (n, n))
54.     maxk = math.floor(math.log2(n)**2)
55.     maxr = max(3, math.ceil(math.log2(n)**5))
56.     nextR = True
57.     r = 0
58.     for r in range(2, maxr):
59.         if nextR == False:
60.             break
61.         nextR = False
62.         for k in range(1, maxk+1):
63.             if nextR == True:
64.                 break
65.             nextR = (gmpy2mpz(n**k % r) == 0) or (gmpy2mpz(n**k % r) == 1)
66.     r = r - 1 # 循环多增加了一层
67.     print("r = %d" % r)

```

```

68.
69.     print("步骤3, 如果  $1 < \gcd(a, n) < d$ , 对于一些  $a \leq d$ , 输出合数" % (n, n, r))
70.     for a in range(r, 1, -1):
71.         g = __gcd__(a, n)
72.         if g > 1 and g < n:
73.             return False
74.
75.     print("步骤4, 如果  $n=d \leq r=d$ , 输出素数" % (n, r))
76.     if n <= r:
77.         return True
78.
79.     print("步骤5")
80.     print("遍历a从1到 $\sqrt{\phi(r=d)} \log n=d$ " % (r, n))
81.     print("如果  $(X+a)^d \neq X^d + a \pmod{X^{d-1}, d}$  输出合数" % (n, n, r, n))
82.     # 构造P =  $(X+a)^n \pmod{X^{r-1}}$ 
83.
84.     print("构造多项式  $(X+a)^d$ , 并且进行二项式展开" % n)
85.     X = multi_yssymbols('X')
86.     a = multi_yssymbols('a')
87.     X_a_n_expand = binomial_expand(ypolynomial1(X, a), n)
88.     print(X_a_n_expand)
89.     X.pow(r)
90.     reduce_poly = ypolynomial1(X, ysymbol(value=-1.0))
91.     print("构造消减多项式 %s" % reduce_poly)
92.     print("进行运算  $(X+a)^d \pmod{X^{d-1}}$ " % (n, r))
93.     r_equ = ypolynomial_mod(X_a_n_expand, reduce_poly)
94.     print("得到余式: %s" % r_equ)
95.     print("进行运算'余式' mod %d 得到式(A)" % n)
96.     A = ypolynomial_reduce(r_equ, n)
97.     print("A = %s" % A)
98.     print("B =  $x^d + a \pmod{x^{d-1}}$ " % (n, r))
99.     B = ypolynomial(multi_yssymbols('X', power=31), a)
100.    B = ypolynomial_mod(B, reduce_poly)
101.    print("B = %s" % B)
102.    C = ypolynomial_sub(A, B)
103.    print("C = A - B = %s" % C)
104.    maxa = math.floor(math.sqrt(__phi__(r)) * math.log2(n))
105.    print("遍历a = 1 to %d" % maxa)
106.    print("检查每个'%s = 0 (mod %d)' % (C, n)")
107.    for a in range(1, maxa+1):
108.        print("检查a = %d" % a)
109.        C.set_variables_value(a=a)
110.        v = C.eval()
111.        if v % n != 0:
112.            return False
113.
114.    print("步骤6 输出素数")
115.    return True
116.
117. if __name__ == "__main__":
118.     n = 31
119.     print("检查'%d'是否为素数" % n)
120.     result = is_prime_by_AKS(n)
121.     if result is True:
122.         print("YES")
123.     else:
124.         print("NO")

```

```

125.     else:
126.         pass

```

5.时间复杂性分析与优化

这里先列出在 Z_n 上运算的复杂度以备参考

以下的算法复杂度分析是直接给出的，这种计算是基于 m 比特位的数进行加减乘除运算是 $O \sim (m)$ 时间内的。简单得讲，在两个系数最多 m 比特位的 d 阶多项式上进行以上运行能在 $O \sim (d \cdot m)$ 步能完成。

引理 5.1 这个算法的渐进时间为 $O \sim (\log^{21/2} n)$ 。

证明，算法的**步骤1**花费大约 $O \sim (\log^3 n)$

算法的**步骤2**，我们要找到一个满足 $o_r(n) > \log^2 n$ 的 r 。这需要遍历测试每个 $r(n^k \not\equiv 1 \pmod{r})$ ，其中 $k \leq \log^2 n$ 。对于一个特殊的 r ，模 r 的乘法最多花费 $O(\log^2 n)$ 因此这将花费 $O \sim (\log^2 n \log r)$ 。根据**引理 4.3** 我们仅需要尝试 $O(\log^5 n)$ 次不同的 r 。因此**步骤2**总共需要花费 $O \sim (\log^7 n)$ 。

第三个步骤计算 r 个数的 \gcd 。每个 \gcd 需要花费 $O(\log n)$ ，所以**步骤3**总共花费的 $O(r \log n) = O(\log^6 n)$ 。**步骤4** 仅需要花费 $O(\log n)$ 。

在**步骤5**中，我们需要验证 $\lfloor \sqrt{\phi(r)} \log n \rfloor$ 个等式。每个等式需要花费 $O(\log n)$ 倍的系数的长度为 $O(\log n)$ 的 d 阶多项式时间。因此每个等式能被在 $O \sim (r \log^2 n)$ 个时间步内被验证。因此**步骤5**的时间复杂度是

$$O \sim (r \sqrt{\phi(r)} \log^3 n) = O \sim (r^{\frac{3}{2}} \log^3 n) = O \sim (\log^{21/2} n)。因此这个步骤占整个算法花费的绝大部分时间。$$

这个算法性能提高的关键字在于对 r 的估计(**引理 4.3**)。当然最好的情况是当 $r = O(\log^2 n)$ 并且算法的复杂度在 $O \sim (\log^6 n)$ 。事实上，有两个猜想猜测这样一个 r 。

Artin 猜想：给定任意数非完全平方的 $n \in \mathcal{N}$ ，那么素数的数量 $q \leq m$ 对于 $o_q(n) = q - 1$ 是逐渐趋向于 $A(n) \cdot \frac{m}{\ln m}$ ，这里 $A(n)$ 是 **Artin 常数**， $A(n) > 0.35$ 。

Sophie-Germain 素数分布猜想：对于素数个数 q 小于等于 m ，那么 $2q + 1$ 仍然是一个素数并且渐进于 $\frac{2C_2 m}{\ln^2 m}$ ，这里 C_2 是孪生素数常量（估计接近 **0.66**）。具有这种属性的素数被称为 **Sophie-Germain 素数**。

Artin 猜想-当 $r = O(\log^2 n)$ 时对于 $m = O \sim (\log^2 n)$ 有效。在对 **Artin 猜想**的证明已经取得了一些进展 [GM84, GMM85, HB86]，这个猜想在一定条件的**广义黎曼假设**下成立是被承认的。

如果第二个猜想成立，我们应当断定 $r = O \sim (\log^2 n)$ ：

根据 **Sophie-Germain 素数分布**，至少存在 $\log^2 n$ 个素数在 $8\log^2 n$ 与 $c\log^2 n (\log \log n)^2$ 之间，其中 c 是一个合适的常数。对于这样一个素数 q ， $o_q(n) \leq 2$ 或者 $o_q(n) \geq \frac{q-1}{2}$ 。任意 q 对于每个 $o_q(n) \leq 2$ 都被 $n^2 - 1$ 整除，因此像这样的素数 q 的数量上界是 $O(\log n)$ 。这意味着必然存在一个素数 $r = O \sim (\log^2 n)$ 以致 $o_r(n) > \log^2 n$ 。这样的一个 r 将花费 $O \sim (\log^6 n)$ 的时间复杂度。

对于这个猜想也取得了一些进展。设 $P(m)$ 表示 m 的最大素因子。[Gol69] 说明了 $P(q-1) > q^{\frac{1}{2}+c}$ ，其中 q 是素数并且 $c \approx \frac{1}{12}$ ，同时具有正的密度。对于这个理论的进一步提高，Fouvry 做了以下工作：

引理 5.2[Fou85] 存在常熟 $c > 0$ 与 n_0 ，对于所有的 $x \geq n_0$ ：

$$|\{q \mid q \text{是素数}, q \leq x, P(q-1) > q^{\frac{2}{3}}\}| \geq c \frac{x}{\ln x}$$

以上引理当指数大于**0.6683**时被证实[BH96]。使用以上引理我们可以提高我们的算法性能。

引理 5.3 这个算法的时间复杂度为 $O \sim (\log^{15/2} n)$

证明，通过以上的讨论，一个高密度的素数分布 $P(q-1) > q^{\frac{2}{3}}$ (q 是素数) 意味着**步骤2**可以找到一个 $r = O(\log^3 n)$ ，其中 $o_r(n) > \log^2 n$ 。这将把这个算法的复杂度降低到 $O \sim (\log^{15/2} n)$ 。

最近，Hendrik Lenstra与Carl Pomerance[LP03]提出了这个算法的修改版本并证明了算法的复杂度在 $O \sim (\log^6 n)$ 。

6. 未来要做的事情

在我们的算法中，**步骤5**的循环需要运行 $\lfloor \sqrt{\phi r \log n} \rfloor$ 次确保群 \mathcal{G} 的规模足够大。如果我们可以找到一个比集合 $(X + a)$ 产生的群更小的规模则这个迭代的次数可以被减少。

如果[BP01]提出的猜想被验证并且在 $r \leq 100, n \leq 10^{10}$ 范围内[KS02]被证实，则我们的算法复杂度可以在 $O \sim (\log^3 n)$ 。

猜想 如果 r 是一个不能整除 n 的素数，并且如果

$$(X - 1)^n \equiv X^n - 1 \pmod{X^r - 1, n} \quad (6)$$

那么 n 是素数或者 $n^2 \equiv 1 \pmod{r}$ 。

如果这个猜想是正确的，我们可以修改我们的算法首先寻找一个不能整除 $n^2 - 1$ 的数 r 。以致 r 能在范围 $[2, 4 \log n]$ 内。这是因为素数的积小于 x 并且至少是 e^x ([Apo97])。然后我们可以测试全等式 (6) 成立或者不成立。验证这个全等式需要花费 $O \sim (r \log^2 n)$ 。这将有 $O \sim (\log^3 n)$ 的时间复杂度。

近期，Hendrik Lenstra与Carl Pomerance[LP03b]提出一些参数表明以上猜想是错误的。但是，对于一些变量这个猜想仍然是正确的（例如，如果我们强行指定 $r > \log n$ ）

感谢

作者感谢的一些话，这里就不翻译了。

参考资料

[AB03] M. Agrawal and S. Biswas. Primality and identity testing via Chinese remaindering. Jl. of the ACM, 50:429–443, 2003.

[AH92] L. M. Adleman and M.-D. Huang. Primality testing and two dimensional Abelian varieties over finite fields. Lecture Notes in Mathematics, 1512, 1992.

[AKS03] Manindra Agrawal, Neeraj Kayal, and Nitin Saxena. PRIMES is in P. Preprint (<http://www.cse.iitk.ac.in/news/primality/v3.ps>), February 2003.

- [Apo97] T. M. Apostol. *Introduction to Analytic Number Theory*. Springer-Verlag, 1997.
- [APR83] L. M. Adleman, C. Pomerance, and R. S. Rumely. On distinguishing prime numbers from composite numbers. *Ann. Math.*, 117:173–206, 1983.
- [Atk86] A. O. L. Atkin. Lecture notes of a conference, boulder (colorado). Manuscript, August 1986.
- [BH96] R. C. Baker and G. Harman. The Brun-Titchmarsh Theorem on average. In *Proceedings of a conference in Honor of Heini Halberstam, Volume 1*, pages 39–103, 1996.
- [BP01] Rajat Bhattacharjee and Prashant Pandey. Primality testing. Technical report, IIT Kanpur, 2001. Available at <http://www.cse.iitk.ac.in/research/btp2001/primality.html>.
- [Car10] R. D. Carmichael. Note on a number theory function. *Bull. Amer. Math. Soc.*, 16:232–238, 1910.
- [Fou85] E. Fouvry. Theoreme de Brun-Titchmarsh; application au theoreme de Fermat. *Invent. Math.*, 79:383–407, 1985.
- [GK86] S. Goldwasser and J Kilian. Almost all primes can be quickly certified. In *Proceedings of Annual ACM Symposium on the Theory of Computing*, pages 316–329, 1986.
- [GM84] R. Gupta and M. Ram Murty. A remark on Artin’ s conjecture. *Inventiones Math.*, 78:127–130, 1984.
- [GMM85] R. Gupta, V. Kumar Murty, and M. Ram Murty. The Euclidian algorithm for S integers. In *CMS Conference Proceedings*, pages 189–202, 1985.
- [Gol69] M. Goldfeld. On the number of primes p for which $p+a$ has a large prime factor. *Mathematika*, 16:23–27, 1969.
- [HB86] D. R. Heath-Brown. Artin’ s conjecture for primitive roots. *Quart. J. Math. Oxford*, (2) 37:27–38, 1986.
- [KS02] Neeraj Kayal and Nitin Saxena. Towards a deterministic polynomialtime test. Technical report, IIT Kanpur, 2002. Available at <http://www.cse.iitk.ac.in/research/btp2002/primality.html>.
- [KSS02] Adam Kalai, Amit Sahai, and Madhu Sudan. Notes on primality test and analysis of AKS. Private communication, August 2002.
- [Lee90] J. V. Leeuwen, editor. *Handbook of Theoretical Computer Science*, Volume A. Elsevier, 1990.
- [Len02] H. W. Lenstra, Jr. Primality testing with cyclotomic rings. Unpublished (<http://cr.yp.to/papers.html#aks> has an exposition of Lenstra’ s argument), August 2002.

- [LN86] R. Lidl and H. Niederreiter. Introduction to finite fields and their applications. Cambridge University Press, 1986.
- [LP03a] H. W. Lenstra, Jr. and Carl Pomerance. Primality testing with gaussian periods. Private communication, March 2003.
- [LP03b] H. W. Lenstra, Jr. and Carl Pomerance. Remarks on Agrawal' s conjecture. Unpublished (<http://www.aimath.org/WWN/primesinp/articles/html/50a/>), March 2003.
- [Mac02] Martin Macaj. Some remarks and questions about the AKS algorithm and related conjecture. Unpublished (<http://thales.doa.fmph.uniba.sk/macaj/aksremarks.pdf>), December 2002.
- [Mil76] G. L. Miller. Riemann' s hypothesis and tests for primality. J. Comput. Sys. Sci., 13:300–317, 1976.
- [Nai82] M. Nair. On Chebyshev-type inequalities for primes. Amer. Math. Monthly, 89:126–129, 1982.
- [Pra75] V. Pratt. Every prime has a succinct certificate. SIAM Journal on Computing, 4:214–220, 1975.
- [Rab80] M. O. Rabin. Probabilistic algorithm for testing primality. J. Number Theory, 12:128–138, 1980.
- [SS77] R. Solovay and V. Strassen. A fast Monte-Carlo test for primality. SIAM Journal on Computing, 6:84–86, 1977.
- [vzGG99] Joachim von zur Gathen and Jürgen Gerhard. Modern Computer Algebra. Cambridge University Press, 1999

-
- [1] 埃拉托色尼（公元前三世纪的希腊天文学家、数学家和地理学家） ↵
- [2] https://en.wikipedia.org/wiki/Fermat%27s_little_theorem ↵
- [3] https://en.wikipedia.org/wiki/Carmichael_number ↵
- [4] https://en.wikipedia.org/wiki/Complexity_theory ↵
- [5] <https://en.wikipedia.org/wiki/Co-NP> ↵
- [6] https://en.wikipedia.org/wiki/Generalized_Riemann_hypothesis ↵
- [7] https://en.wikipedia.org/wiki/Jacobi_symbol ↵
- [8] https://en.wikipedia.org/wiki/Elliptic_curve ↵
- [9] https://en.wikipedia.org/wiki/Sophie_Germain_prime ↵
- [10] [https://en.wikipedia.org/wiki/Ring_\(mathematics\)](https://en.wikipedia.org/wiki/Ring_(mathematics)) ↵
- [11] [https://en.wikipedia.org/wiki/Field_\(mathematics\)](https://en.wikipedia.org/wiki/Field_(mathematics)) ↵
- [12] 例如， $r \leq \lceil \log^5 n \rceil$ ，相应的当 $n \leq 5,690,034$ 时满足步骤4。 ↵

