$p = 3 \cdot 2^{1274} - 1$ 是一个素数

杨文颜

2025 年 2 月 20 日

1 唉，我怎么知道这是一个素数的呢？

众所周知，大数的分解很难，但是素数的判别却很容易. 最简单的方法是做一些伪素数测试. 比如说可以使用著名的 Fermat 小定理. 也就是

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

对于这样的高次幂的取余计算，我们可以使用传统的逐次平方法，对于 a^n 的计算这只要 $O(\log n)$ 步，非常方便. 如果我们得到了

$$a^n \not\equiv a \pmod{n}$$

那么我们可以将 a 作为 n 不是素数的证据. 通常来说，这样的证据非常之多，完全足够我们验证，随便挑一个 2 可能都足够作为证据了.

那也未必！

但此时有例外发生. 第一个找不出这样的证据的数是 $561 = 3 \cdot 11 \cdot 17$. 虽然如此，但是

$$a^{561} \equiv a \pmod{561}.$$

这是一种能冒充素数的数字，所以 Fermat 小定理的逆不成立. 如果你找不到 561 的分解中出现的数字就没有办法分解它. 当然这个时候你可能会说这个因子 3 太小了. 但还有素因子更大的这样的数字，比如

$$340561 = 13 \cdot 17 \cdot 23 \cdot 67.$$

事实上，所有形如 $(6k+1)(12k+1)(18k+1)$，其中三个因子都是素数的数都满足要求. 这样的数字有个名字叫做 Carmichael 数. Erdös 曾经猜测有无穷多个 Carmichael 数，目前这个结论已经得到证明. Carmichael 数十分的稀有，但是已经知道在 1 到 n 中间也至少有 n^c 这么多个，c 是一个目前改进到比 1/3 大一点点的常数.

为了不放过这些漏网之鱼，我们需要使用所谓的 Rabin-Miller 测试. 这是一种以上 Fermat 小定理判别方式的改良. 原理是这样的，我们知道对于 $\gcd(a,p) = 1$ 的 a 素数 p 一定有

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

那么如果我们写 $p - 1 = 2^s \cdot c$，其中 c 是一个素数，那么要么有

$$a^c \equiv 1 \pmod{p}.$$

要么有

$$a^{2^r c} \equiv -1 \pmod{p} \text{对于某个} 0 \le r \le s-1.$$

这是因为若 p 是一个素数那么 $x^2 \equiv 1 \pmod{p}$ 仅有 $x \equiv \pm 1 \pmod{p}$ 作为解. 如果这两点都不满足，那么 a 就又能成为一个证据. 这个时候 Carmichael 数就被成功识别出来了. 还是以 561 为例，2 此时就可以被摆上证人席了. 注意到 $561 - 1 = 2^4 \cdot 35$，

$$2^{35} \equiv 263 \pmod{561}$$

$$2^{70} \equiv 166 \pmod{561}$$

$$2^{140} \equiv 67 \pmod{561}$$

$$2^{280} \equiv 1 \pmod{561}$$

-1 被跳过了！所以只需要 2 作为证据就可以证实 561 不是一个素数了.

有一个证明表明，对于 Rabin-Miller 测试，理论上说至少有一定（非常高的）比例的数都可以作为证据，所以可以很放心的使用它. 常见的用法是随机挑几个 a，然后进行检验，如果都不是证据，就说明 p “极为可能”就是一个素数.

2 那也未必！

我就知道有杠精要来抬杠了！哪怕就 0.1% 的概率不是概率吗，你这是典型的投机主义，犯了右倾错误！别急，我有对策，那就是

使用真正的素性测试！

我们之前的测试方法，基本上可以称为合数测试. Miller-Rabin 测试是一个概率性算法，也就是说不论输入如何它的运行时间是确定的，但问题在于我们不能保证输出的正确性. 只有输出为合数的时候它的结果才是绝对准确的.

但是我们有真正的素性测试. 这个测试方法就是经典的 Pocklington-Lehmer 测试.

命题 2.1. *假设 $n > 1$ 是一个整数，$n - 1 = rs$ 且 $r \geq \sqrt{n}$. 假设对于每一个 r 的素因子 l，我们都能找到整数 a_l 满足以下条件：*

$$a_l^{n-1} \equiv 1 \pmod{n}, \quad \gcd\left(a_l^{(n-1)/l} - 1, n\right) = 1.$$

那么，n 是素数.

证明. 假设 p 是一个 n 的素因子. 我们假设 $l^e || r$，l 是 r 的素因子. 假设 $b \equiv a_l^{(n-1)/l^e} \pmod{p}$. 那么

$$b^{l^e} \equiv 1 \pmod{p}, \quad b^{l^{e-1}} \not\equiv 1 \pmod{p}.$$

因此 b 在乘法群 $(\mathbb{Z}/p\mathbb{Z})^*$ 中的阶是 l^e. 因此 $l^e | p - 1$. 让 l 取遍 r 的素因子，就得到了 $r | p - 1$. 特别地，

$$p > r \geq \sqrt{n}.$$

如果 n 是合数，那么至少有一个不超过 $\sqrt{n}$ 的素因子，矛盾，因此 n 只能是素数. □

本质上来说这个命题有点像是寻找原根，但是 a_l 的选择有一定空间不一定要求相同，所以比找原根要容易一些. 这个命题的逆也是对的，确实每一个素数都可以这么干. 但是这里有一个问题，我要怎么做才能知道足够多的 $n - 1$ 的因子，以至于能找到 $r \geq \sqrt{n}$ 并且还要知道 r 的全部素因子？答案自然是

没办法.

世界上有些东西不能强求，不如就此放下. 但是，有的时候总会遇到一些惊喜. 数学世界里有一种神奇的魔法，叫做椭圆曲线. 最早人们并没有把椭圆曲线引入到素数测试这种看起来毫无关联的主题中. 但是 Goldwasser 和 Kilian 看到了其中的精妙之处. 我们首先需要一个定理.

定理 2.2 (Hasse 定理)**.** *对于有限域 $\mathbb{F}_p$ 上的椭圆曲线 E 来说，E 上点的个数 $\#E(\mathbb{F}_p) \in [p+1-2\sqrt{p}, p+1+2\sqrt{p}]$.*

这个区间也被称为 Hasse 区间. 接下来我们有一种用椭圆曲线来判定一个数是素数的方法.

定理 2.3 (简单版本的椭圆曲线素性测试)**.** *假设 $n > 1$ 是一个整数. 并且 E 是一个 $\pmod n$ 的椭圆曲线. 假设存在素数 $l_1, l_2, \cdots, l_r$ 并且有限点 $P_i \in E(\mathbb{Z}_n)$ 满足条件：*

- $l_i P_i = \infty, 1 \le i \le k$.
- $\prod_{i=1}^{r} l_i > (n^{1/4}+1)^2$.

那么 n 是一个素数.

证明的逻辑与 Pocklington-Lehmer 测试是一致的. 这里我们感受到这个操作的精妙之处. 如果我们仔细思考一下，假如 E 被换成一个具有奇性的曲线，并且 n 真的是一个素数，那么事实上可以有 $E(\mathbb{Z}_n) \simeq \mathbb{Z}_n^*$ 具有阶 $n-1$. 所以本质上来说，传统的检验方法是在这个乘法群上干活，但 $n-1$ 这个数字没有办法控制. 通过椭圆曲线的手法，我们可以“偷梁换柱”，换一个阶数在 n 附近的 Abel 群再来操作. Hasse 定理给出了我们一个大致可以操作的范围.

这个时候还需要一个有效性的问题. 为了找到一个能够使用的 E，通常我们可以随机生成一些 $\pmod n$ 的椭圆曲线. 生成的方式是随机生成一个有限点 $P \in \mathbb{Z}_n \times \mathbb{Z}_n$，之后再去得到一个可能的 P 满足的三次方程. 一番操作之后我们可能希望这个 P 在 E 中的阶略比 $(n^{1/4}+1)^2$ 这个数大一点点. 因为我们同样也需要 l 的素性，假如它跟 n 差不多大那么它没有什么价值. 为了得到 P 的阶，我们也有一些好的办法，这里不赘述. 当然，如果 P 的阶是一个差不多这么大的 l 的倍数，那用相应的 P 的倍数替换掉也行.

我们举一个例子. 假如我们要来证明 $n = 907$ 是一个素数. 考虑一个 $\pmod n$ 椭圆曲线 $y^2 = x^3 + 10x - 2$. 我们知道其上有一个点是 $(1,3)$. 通过一些操作可以得到 $(1,3)$ 这个点的阶是 $923 = 13 \cdot 71$. 这个时候我们考虑 $P = 13(1,3)$，其阶数就是 71. 因此我们直接选取 $l = 71 > (907^{1/4}+1)^2 \approx 42.1$.

值得注意的是，为了让这个方法完备，你还要证明你选取的 l 是一个素数，比如这里是 71. 而好的情况下它通常已经只有 $O(\sqrt{n})$ 的大小，所以你可以逐步地缩小，直到你可以直接试除为止.

如今这种方法也有优化. Schoof 算法是一种能在多项式时间内计算椭圆曲线的阶的方法，用它可以更快速找到合适的阶数来选出 l. 通过复乘的理论，Atkin 和 Morain 还给出了一些更好的选取 E 的方法.

3 主播主播，你的方法确实很强，但还是太吃操作了！

有的兄弟，有的. 正所谓大道至简. 当你还在为了寻找椭圆曲线 E 的阶发愁的时候，只能说明你没有遇到合适的. 但假如你足够幸运，或者足够有观察力，你可以不需要递归地去算 l，不需要去到处寻找 E.

越单纯的原理，往往背后包含的智慧就越深奥.

上个世纪有一个数学家叫做 Pomerance，他在 2012 年成为了美国数学会的会士. 在 1987 年他的一篇文章 *Very Short Primality Proofs*（翻译过来就是超短的素性证明）中提出了这个思想 [2]. 对于有一些特定类型的素数，毫无疑问有着非常简单的素性证明方式. 比如说 Fermat 数，

$$p = 2^{2^n} + 1.$$

你只需要检查 $3^{(p-1)/2} \equiv -1 \pmod p$ 成立就可以说明它是一个素数了. 换言之我们不需要去寻找特定的证据 a. 类似地，对于 Mersenne 数也就是形如

$$p = 2^n - 1.$$

为了证明它是一个素数，还有专门的 Lucas-Lehmer 测试来提供一个非常简单的方法. Pomernance 的想法就是，每一个素数 p，我都有一种非常简单的，只需要 $O(\log p)$ 步的乘法的，直接判定它是一个素数的方法. 注意，和上面这

些只能判定特殊的数是不是素数的方法不同，这里的 p 可以是任何一个素数. 这里的情况比 Goldwasser-Kilian 测试的不同的点在于，Goldwasser-Kilian 测试的确非常有效，但需要用 $O(\log^2 p)$ 步的乘法. 这是因为计算点的阶的过程需要一些迭代的过程，包括还有素性证明自身的迭代过程，也就意味着它尽管已经优化到了多项式时间，可以用于实际，但是在随机选取的过程中，证明方式不是唯一的，可能还有更加简单的证明方法.[1]

是的，Pomerance 的想法是对的，其原理也类似于 Hasse 定理，只不过稍微有一点差别，就是

定理 3.1 (Hasse 定理的“逆”). *取定一个素数 p，对于每一个 Hasse 区间内的整数 $N \in [p+1-2\sqrt{p}, p+1+2\sqrt{p}]$，都存在一个椭圆曲线 $E/\mathbb{F}_p$ 使得 $E(\mathbb{F}_p)$ 是一个阶为 N 的循环群.*

接下来我要挑选一个 2 的幂次 $2^k > (p^{1/4}+1)^2 \geq 2^{k-1}$. 这个数字大概是 $\sqrt{p}$ 到 $2\sqrt{p}$ 之间这么大，而 Hasse 区间有 $4\sqrt{p}$ 的长度，可以证明对于 $p > 31$，Hasse 区间中都包含一个数，使得它是 2^k 的倍数. 是的，Pomerance 想要就挑出这样一个特定的数 N 出来，假如根据上面 Hasse 定理的“逆”挑出了一个这样的 E 使得 $E(\mathbb{F}_p)$ 是一个阶为 N 的循环群，那么只要在在这样的 E 上挑出一个点，使得它的阶为 2^k，就可以达到同样的效果. 这样整个证明过程就在 $O(\log p)$ 步乘法内完成了.

但是，往往找到一个 Pomerance 的“完美”证明不太容易，因为假如你什么都不知道，找到这样的 E 可能要花费远远更多的时间.

那也未必！

是的，虽然这个方法初看起来只能拿来观赏，但是我们确实在一些情况下不需要任何技巧就可以写出一个 Pomerance 证明. 当然，类似地，只是一些特定形式的数.

4 利用 Pomerance 证明写出的超大素数

最后一节是一个纯粹实践性的问题，那就是，在不使用任何黑箱技巧的前提下，纯粹通过 $\pmod n$ 的加减乘除运算，写一个超过 2^{1000} 大小的素数. 这并不是一个完全无聊的问题，因为寻找相当大的素数对于如今的密码学来说很重要. 当然，找 Mersenne 素数肯定是一个简单的方案. 但这次我们用 Pomerance 证明来寻找形如 $3 \cdot 2^k - 1$ 这样的素数. 因为大小为 n 的数是素数的概率差不多是 $1/\log n$，这里由于肯定不是 2 和 3 的倍数所以大致选中的概率会翻 3 倍，为了节约资源，我们就在 k 在 $1001 \sim 1400$ 这个范围内进行搜索，我们预测差不多能找到一个左右.

首先，我们先通过试除法，去除掉一些显然不在考察范围内的对象，这里我们设置的是没有小于 2^{17} 的素因子.

```
from sage.all import *
limit = 2^17
primes_list = list(prime_range(1, limit))
satisfying_k = []
for k in range(1001, 1401):
    found_prime = False
    target_value = 3 * 2^k
    for p in primes_list:
        F = GF(p)
        result = F(target_value) - 1
        if result == 0:
            found_prime = True
            break
    if not found_prime:
```

[1] 这里指的是作为证据的证明的步骤长度，如果你要把寻找证明的时间包含进去，那么改进后的 AKS 方法能够做到 $O(\log^4 p)$ 这么好.

```
        satisfying_k.append(k)
print("所有满足条件的 k:", satisfying_k)
```

最后的输出如下：

```
所有满足条件的 k: [1027, 1030, 1035, 1043, 1046, 1055, 1056, 1058, 1060, 1068, 1071, 1079,
    1080, 1086, 1099, 1106, 1107, 1116, 1120, 1140, 1144, 1159, 1163, 1164, 1166, 1167, 1176,
    1178, 1183, 1186, 1188, 1190, 1207, 1210, 1218, 1238, 1239, 1251, 1258, 1259, 1260, 1264,
    1266, 1271, 1274, 1275, 1278, 1286, 1287, 1291, 1296, 1300, 1318, 1324, 1326, 1351, 1359,
    1371, 1379, 1383, 1390, 1394, 1399]
```

这里有 63 个不同的候选. 现在让我们来挑选椭圆曲线 E. 此时我的推荐是 $E: y^2 = x^3 + 8$. 这条椭圆曲线的优势在于，对于任意的 $p \equiv 2 \pmod 3$，都有

$$\#E(\mathbb{F}_p) = p + 1.$$

因为这是一个具有复乘的椭圆曲线. 当然也可以直接证明这一点.

$$\#E(\mathbb{F}_p) = p + 1 + \sum_{x \in \mathbb{F}_p} \left(\frac{x^3 + 8}{p} \right) = p + 1.$$

这是因为 $|\mathbb{F}_p^*| = p - 1$ 不被 3 整除因此 $x^3 + 8$ 恰好取遍 $\mathbb{F}_p$ 中的元素.

我们先挑选 E 上一个明显的点，当然你不能挑 $(-2, 0)$ 这样的在 $\mathbb{Q}$ 上都是 2-torsion 点的东西，我们就挑一个最简单的，$P = (1, 3)$（其他的有理点其实差不多因为 $\operatorname{rank}(E/\mathbb{Q}) = 1$）. 现在我们把它乘以 3 倍. 这个可以手算完成，算出来的答案应该是

$$3P = (\frac{433}{121}, -\frac{9765}{1331}).$$

现在我希望知道 $3P$ 的阶是多少. 我希望它是一个很大的 2 的方幂. 为了验证这一点，我们只需要不停地把 $3P$ 加倍. 椭圆曲线上点加倍的公式我们在这里回顾一下.

$$Q = (x_1, y_1), 2Q = (x_3, y_3), x_3 = m^2 - 2x_1, y_3 = m(x_1 - x_3) - y_1, m = \frac{3x_1^2 + A}{2y_1}.$$

这里当然 A 就是 0. 我们干脆就把 $3P$ 加倍 1500 次（虽然 k 次可能就看出来了，但其实消耗一点电脑的资源也未尝不可）.

```
p = 3 * 2^1027 - 1
R = IntegerModRing(p)
x = R(433) / R(121)
y = R(-9765) / R(1331)
for k in range(0, 1500):
    m = 3 * x * x / 2 / y
    x_new = m * m - 2 * x
    y = m * (x - x_new) - y
    x = x_new
print("程序正常输出。")
```

这里的样例是 $k = 1027$ 时候的程序. 如果程序正常输出，那么就说明 p 应该不是一个素数. 不然的话因为 $p = 3 \cdot 2^k - 1 \equiv 2 \pmod 3$，所以 $3P$ 的阶一定要能整除 2^k. 所以不可能加倍 1500 次之后没有任何反应. 但如果 p 是一个合数的话就能够说得通，因为这个时候比如 $p = p_1 p_2$ 是两个素数的乘积，那么

$$E(\mathbb{Z}_p) = E(\mathbb{Z}_{p_1}) \oplus E(\mathbb{Z}_{p_2})$$

的阶就是 $(p_1-1)(p_2-1)$. 所以 $3P$ 可能不停加倍之后会有重复（当然 1500 次之内几乎可以肯定重复不了），也有可能在加倍很多次之后直接得到一个非有限点. 但总的来说很大概率这个程序就能够正常运行.

如果你将 1027 换成上述 63 个候选中的其他 62 个，这个程序都能正常输出. 但如果你用的是 $k=1274$，那么你会看到报错信息：

```
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
Cell In[2], line 6
      4 y = R(-Integer(9765)) / R(Integer(1331))
      5 for k in range(Integer(0), Integer(1500)):
----> 6     m = Integer(3) * x * x / Integer(2) / y
      7     x_new = m * m - Integer(2) * x
      8     y = m * (x - x_new) - y
File /ext/sage/10.4/src/sage/structure/element.pyx:1734, in
    sage.structure.element.Element.__truediv__()
   1732 cdef int cl = classify_elements(left, right)
   1733 if HAVE_SAME_PARENT(cl):
-> 1734     return (<Element>left)._div_(right)
   1735 if BOTH_ARE_ELEMENT(cl):
   1736     return coercion_model.bin_op(left, right, truediv)
File /ext/sage/10.4/src/sage/rings/finite_rings/integer_mod.pyx:2256, in
    sage.rings.finite_rings.integer_mod.IntegerMod_gmp._div_()
   2254       71428571429
   2255     """
-> 2256     return self._mul_(~right)
   2257
   2258 def __int__(self):
File /ext/sage/10.4/src/sage/rings/finite_rings/integer_mod.pyx:2342, in
    sage.rings.finite_rings.integer_mod.IntegerMod_gmp.__invert__()
   2340 """
   2341 if self.is_zero():
-> 2342     raise ZeroDivisionError(f"inverse of Mod(0, {self._modulus.sageInteger}) does not
    exist")
   2343
   2344 cdef IntegerMod_gmp x
ZeroDivisionError: inverse of Mod(0,
    975743643249787413930491280690383019245180835681253575498666782850338255594377527938647447662405392328
    does not exist
```

此时仔细看是发生了除以 0 的错误. 现在我们修改一下程序为

```
p = 3 * 2^1274 - 1
R = IntegerModRing(p)
x = R(433) / R(121)
y = R(-9765) / R(1331)
```

```
for k in range(0, 1271):
    m = 3 * x * x / 2 / y
    x_new = m * m - 2 * x
    y = m * (x - x_new) - y
    x = x_new
print(x + 2, y)
```

得到了

```
0 0
```

换言之 $3\cdot 2^{1271}P=(-2,0)$，也就是说 $3\cdot 2^{1272}P=\infty$. 这其实已经足够说明 $p=3\cdot 2^{1274}-1$ 是一个素数. 当然，其实也可以找到一个恰好阶是 $3\cdot 2^{1274}$ 阶的元素（如果你喜欢），当然在这里我认为没有必要，因为试除法已经断绝了小因子的可能性了.

当然，我们最后用常规方法验证一下.

```
p = 3 * 2^1274 - 1
is_prime = is_prime(p)
print(is_prime)
```

最后自然返回了 True 的结果.

写在最后

正文的总结和思考

我对算法数论一无所知，以下内容完全是一个外行的人的理解. 有限域上的椭圆曲线的一个应用的哲学就在于和 singular curve 的对比，将传统的加法乘法群替换为了一个有一定自由度的 Abel 群. 这种哲学也体现在用椭圆曲线来做大数分解的 ECF 算法中.

Pomerance proof 的应用是正文后半段的重头戏. 其原理十分简单，但却可以有很多应用. 寻找 $p=3\cdot 2^{1274}-1$ 这样的素数虽然看起来没什么用但是这却将上面的哲学体现的淋漓尽致.

实际上早在 1975 年 Pratt 就已经证明了证明一个数是素数是一个 NP 的问题. 但 Pomerance 用一个最简单的道理说明了其实每一个素数都可以找到一个只有线性长度的证明. 不需要广义 Riemann 假设这样的大道理，这几乎是一个可以演示给任何一个知道一点初等数论的人的 trick.

那么其实这就引出了一个最重要的问题，就是在这个证明过程中要寻找具有特定数量点的椭圆曲线. 尽管椭圆曲线上的点数量的计数现在已经有了 Schoof 算法这样的方法，但我们并不能指望仅仅通过随机抽取来查找. 正如我们看到的，对于特定类型的数可以使用特定的具有复乘法的 $E/\mathbb{Q}$ 的曲线直接 reduction 得到. 如果一般要找一个有特定数量点的椭圆曲线，Stevenhagen 曾经思考过这个问题并且给出了一些方案. 其方法也和我们之前的一篇文章中提到的 j 函数的特殊值有关系（没错又是这个经典的问题），当然也有一些复乘的理论，我不了解这方面的工作就不多说了. 当然总的来说对于过于大的数寻找 E 和 p 这并不容易.

当然我们可以演示一个简单的版本.

彩蛋：寻找具有特定数量点的椭圆曲线

一个有趣的事情是对于一个比较小的数来说找一个具有特定数量点的椭圆曲线倒是实际上可行的. 比如我的一个同学的生日是 2005 年 2 月 27 日，让我们来尝试找到一个椭圆曲线上面刚好有 2005227 个点[2]. 要做到这个事情并

[2]这个看起来有趣又无厘头的应用来自 [2].

不是十分容易，因为首先我们需要找一个素数 p 比较接近 2005227. 经过我的一番试验之后我选定了 $p = 2006341$. 接下来我们需要两个 j 函数的特殊值[3]，就是

$$j\left(\frac{1+\sqrt{-91}}{2}\right) = -\left(2^2 \cdot 3 \cdot \epsilon^4(3\epsilon - 8)\right)^3,$$

$$j\left(\frac{1+\sqrt{-91}}{2}\right) - 1728 = -7\left(2^3 \cdot 3^3 \cdot 11\epsilon^5(\epsilon - 2)\right)^2.$$

其中

$$\epsilon = \frac{3+\sqrt{13}}{2}.$$

$\mathbb{F}_p$ 上 13 是一个二次剩余，所以我们可以定义这样一条椭圆曲线

$$y^2 = x^3 - 3cx - 2c.$$

其中

$$c = \frac{j}{j - 1728}.$$

那么这条椭圆曲线就有 2005227 个点. 我们来验证一下. 首先注意到 $x^2 \equiv 13 \pmod{2006341}$ 的一个解是 $x \equiv 253905 \pmod{2006341}$. 然后我们就可以写出程序.

```
p = 2006341
R = GF(p)
eps = (R(3) + R(253905)) / R(2)
j = - (2^5 * 3 * eps^4 * (3 * eps - 8))^3
print(j)
l = - R(7) * (2^3 * 3^3 * 11 * eps^5 * (eps - 2))^2
print(l)
c = j / l
print(c)
E = EllipticCurve([-3 * c,-2 * c])
print(E)
N = E.count_points()
print(N)
```

程序运行结果如下.

```
1488822
1487094
1798713
Elliptic Curve defined by y^2 = x^3 + 622884*x + 415256 over Finite Field of size 2006341
2005227
```

参考文献

[1] C. Pomerance, *Very short primality proofs*, Math. of Comp. 48(1987), 315-322.

[2] Heng Huat Chan, Elisavet Konstantinou, Aristides Kontogeorgis, Chik How Tan, *What is your "birthday elliptic curve"?*, Finite Fields and Their Applications, Volume 18, Issue 6, 2012, 1232-1241.

[3] W. E. H. Berwick, *Modular Invariants*, Proc. Lond. Math. Soc. 28 (1927), 53-69.

[3]这些常数的值可以在 Berwick 的表格中查到 [3].