# 聊聊如何检测素数

作者 张洋 | 发布于 2012-08-28

素数　概率　数学

最近看到一则颇为有趣的新闻，说北大一名大一新生，以素数为标准选手机号，受到广大网友膜拜。其实素数的检测算法是很有趣的，并且会涉及到数论、概率算法等诸多内容，一直觉得素数探测算法是了解概率算法很好的入口。本文和大家简单聊聊如何确定一个数是素数。

## 素数

### 素数的定义

素数是这样被定义的：

一个大于1的整数，如果不能被除1和它本身外的其它正整数整除，则是素数（又称质数）。

与素数相关的定义还有合数：

一个大于1的整数，如果不是素数则是合数。其中能整除这个数的正整数叫做约数，不等于1也不等于合数本身的约数叫做非平凡约数。

注意1既不是素数又不是合数。

举几个例子：

2是素数，因为除1和2外没有其它正整数可以整除2。

3也是素数。

4不是素数，因为2可以整除4。

11是素数，除1和11外没有正整数可以整除它。

15不是素数，3和5可以整除15。

### 素数的性质

素数有一些有趣的性质，下面不加证明的列几条。

素数有无穷多个。

设f(n)为定义在大于1的整数集合上的函数，令f(n)的值为不大于n的素数的个数，则：

$$\lim_{n\to\infty} \frac{f(n)}{n/\ln n} = 1$$

这个函数叫做素数分布函数，反映了素数的分布律。换言之，可以认为大于1的前n个正整数中，素数的个数大约是$n/\ln n$。

## 检测素数

所谓素数检测，就是给定任意一个大于1的整数，判断这个数是否素数。

### 因子检测法

最直观的素数检测算法就是因子检测法。说白了，就是从2到n-1一个个拿来试，看能否整除n，如果有能整除的（找到一个因子），则输出不是质数，否则则认为n为质数。当然，实际上不需要试探到n-1，只要到$\sqrt{n}$就好了，原因如下：

设$n = a \times b$，且a、b均为n的非平凡约数，显然$a > \sqrt{n}$和$b > \sqrt{n}$不可能同时成立，因为同时成立时a*b就会大于n，所以，如果n存在非平凡约数，则至少有一个小于等于$\sqrt{n}$，因此只要遍历到$\sqrt{n}$就可以了。

因子检测法的实现代码如下（python）：

```
def prime_test_factor(n):
    if n == 1:
        return False
    for i in range(2, 1 + int(floor(sqrt(n)))):
        if n % i == 0:
            return False
    return True
```

做几个测试：

```
print prime_test_factor(2) #True
print prime_test_factor(11) #True
print prime_test_factor(15) #False
print prime_test_factor(2147483647) #True
```

很明显，因子检测法的时间复杂度为$O(\sqrt{n})$，一般来看，这个时间复杂度已经很不错了，不过对于超级大的数（例如RSA加密中找几百位的素数是很正常的），这个复杂度还是太大了。

例如对于下面的整数：

6864797660130609714981900799081393217269435300143305409394463459185543183397656052122559640661454554977296311391480858037121987999716643812574028291115057151

哪位壮士可以试试用因子检测法检测这个数是质数还是合数，估计这辈子结果是出不来了，下辈子也悬。所以需要更高效的素数检测算法。

# 费马检测

坦白说，对于大素数的探测，目前并没有非常有效的确定性算法。不过借助费马定理，可以构造一种有效的概率算法来进行素数探测。

## 费马定理

首先看一下什么是费马定理。这条定理是史上最杰出的业余数学家费马发现的一条数论中的重要定理， 这条定理可以表述为：

如果p为素数，则对任何小于p的正整数a有

$$a^{p-1} \equiv 1(mod\ p)$$

根据基本数理逻辑，一个命题正确，当且仅当其逆否命题正确。所以费马定理蕴含了这样一个事实：如果某个小于p的正整数不符合上述公式，则p一定不是素数；令人惊讶的是，费马定理的逆命题也“几乎正确”，也就是说如果所有小于p的正整数都符合上述公式，则p“几乎就是一个素数”。当然，“几乎正确”就意味着有出错的可能，这个话题我们后续再来讨论。至少从目前来看，费马定理给我们提供了一条检测素数的方法。

下面再通过例子说明一下费马定理表达的意义，例如我们知道7是一个素数，则：

$$\begin{aligned}
1^6 &= 1 && \equiv 1(mod\ 7)\\
2^6 &= 64 && \equiv 1(mod\ 7)\\
3^6 &= 729 && \equiv 1(mod\ 7)\\
4^6 &= 4096 && \equiv 1(mod\ 7)\\
5^6 &= 15625 && \equiv 1(mod\ 7)\\
6^6 &= 46656 && \equiv 1(mod\ 7)
\end{aligned}$$

其它素数可以可以用类似方法验证，关于这个定理的严格证明本文不再给出。

所以可以使用如下方法进行大素数探测：选择一个底数（例如2），对于大整数p，如果2^(p-1)与1不是模p同余数，则p**一定**不是素数；否则，则p**很可能**是一个素数。

至于出现假阳性（即合数被判定为素数）的概率，已有研究表明，随着整数趋向于无穷，这个概率趋向于零，在以2为底的情况下，512位整数碰到假阳性的概率为1/10^20，而在1024位整数中，碰到假阳性的概率为1/10^41。因此如果使用此法检测充分大的数，碰到错误的可能性微乎其微。

## 模幂的快速算法

仅有费马定理还不能写检测算法，因为对于大整数p来说，a^(p - 1) (mod p)不是一个容易计算的数字，例如上上面那个超大整数来说，直接计算2的那么多次幂真是要死人了，其效果一点不比因子分解法好。所以寻找一种更有效的取模幂算法。通常来说，重复平方法是一个不错的选择。下面通过例子介绍一下这个方法。

假设现在要求2的10次方，一种方法当然是将10个2连乘，不过还有这样一种计算方法：

10的二进制表示是1010，因此：

$$2^{10} = 2^{[1010]_2}$$

现初始化结果d=2^0=1，我们希望通过乘上某些数变换到2^10，变换序列如下：

$$\begin{aligned}
&2^{[0]_2} \\
&2^{[0]_2} \times 2^{[0]_2} && \times 2 = 2^{[1]_2} \\
&2^{[1]_2} \times 2^{[1]_2} && = 2^{[10]_2} \\
&2^{[10]_2} \times 2^{[10]_2} && \times 2 = 2^{[101]_2} \\
&2^{[101]_2} \times 2^{[101]_2} && = 2^{[1010]_2}
\end{aligned}$$

可以看到这样一个规律：对中间结果d自身进行平方，等于在二进制指数的尾部“生出”一个0；对中间结果d自身进行平方再乘以底数，等于在二进制指数尾部“生出”一个1。靠这样不断让指数“生长”，就可以构造出幂。如果在每次运算时取模，就可以得到模幂了，下面是这个算法的python实现：

```
def compute_power(a, p, m):
    result = 1
    p_bin = bin(p)[2:]
    length = len(p_bin)
    for i in range(0, length):
        result = result**2 % m
        if p_bin[i] == '1':
            result = result * a % m

    return result
```

这个算法的复杂度正比于a、p和m中位数最多的数的二进制位数，要远远低于朴素的模幂求解法。

例如，下面的代码在我的机器上瞬间可以完成：

```
compute_power(2,
68647976601306097149819007990813932172694353001433054093944634591855431833976560521225596406614545549772963113914808580371219879997166438125740282911150571 50,
68647976601306097149819007990813932172694353001433054093944634591855431833976560521225596406614545549772963113914808580371219879997166438125740282911150571 51)
```

而用直观方法计算如此大指数的幂基本是不可能的。

## 费马检测的实现

有了上的铺垫，下面可以实现费马检测了：

```
def prime_test_fermat(p):
    if p == 1:
        return False
    if p == 2:
        return True
    d = compute_power(2, p - 1, p)
    if d == 1:
        return True
    return False
```

以下是一些测试：

```
print prime_test_fermat(7) #True
print prime_test_fermat(11) #True
print prime_test_fermat(15) #False
print prime_test_fermat(121) #False
print prime_test_fermat(561) #True
print
prime_test_fermat(68647976601306097149819007990813932172694353001433054093944634591855431833976560521225596406614545549772963113914808580371219879997166438125740282911150571 51) #True
```

需要注意的是，倒数第二个结果实际是错的，因为561可以分解为3和187。

相对来说，因子分解法适合比较小的数的探测，可以给出准确的结论，但是对于大整数效率不可接受，例如上面最后一个超大整数，因子分解法基本不可行；费马测试当给出否定结论时，是准确的，但是肯定结论有可能是错误的，对于大整数的效率很高，并且误判率随着整数的增大而降低。

# Miller-Rabin检测

上文说，费马检测失误的概率随着整数不断增大而趋向于0，看似是对大素数检测很好的算法。那么我们考虑另外一个问题：如果一个数p是合数，a是小于p的正整数且a不满足费马定理公式，那么a叫做p是合数的一个证据，问题是，对于任意一个合数p，是否总存在证据？

答案是否定的。例如561这个数，可以分解为3乘以187，但是如果你试过会发现所有小于561的正整数均符合费马定理公式。这就意味着，费马检测对于561是完全失效的。类似561这样是合数但是可以完全欺骗费马检测的数叫做Carmichael数。Carmichael数虽然密度不大（前10亿个正整数中约600个），但是已经被证明有无穷多个。Carmichael数的存在迫使需要一种更强的检测条件配合单纯费马检测使用，其中Miller-Rabin检测是目前应用比较广泛的一种。

Miller-Rabin检测依赖以下定理：

如果p是素数，x是小于p的正整数，且x^2 = 1 mod p，则x要么为1，要么为p-1。

简单证明：如果x^2 = 1 mod p，则p整除x^2 - 1，即整除(x+1)(x-1)，由于p是素数，所以p要么整除x+1，要么整除x-1，前者则x为p-1，后者则x为1。

以上定理说明，如果对于任意一个小于p的正整数x，发现1（模p）的非平凡平方根存在，则说明p是合数。

对于p-1，我们总可以将其表示为u2^t，其中u是奇数，t是正整数。此时：

$$a^{p-1} = a^{u2^t} = \left(a^u\right)^{2^t}$$

也就是可以通过先算出a^u，然后经过连续t次平方计算出a^(p-1)，并且，在任意一次平方时发现了非平凡平方根，则断定p是合数。

例如，560 = 35 * 2^4，所以可设u=35，t=4：

$$\begin{aligned} 2^{35} &\mod 561 = 263 \\ 263^2 &\mod 561 = 166 \\ 166^2 &\mod 561 = 67 \\ 67^2 &\mod 561 = 1 \end{aligned}$$

由于找到了一个非平凡平方根67，所以可以断言561是合数。因此2就成为了561是合数的一个证据。

一般的，Miller-Rabin算法的python实现如下：

```python
def miller_rabin_witness(a, p):
    if p == 1:
        return False
    if p == 2:
        return True

    n = p - 1
    t = int(floor(log(n, 2)))
    u = 1
    while t > 0:
        u = n / 2**t
        if n % 2**t == 0 and u % 2 == 1:
            break
        t = t - 1

    b1 = b2 = compute_power(a, u, p)
    for i in range(1, t + 1):
        b2 = b1**2 % p
        if b2 == 1 and b1 != 1 and b1 != (p - 1):
            return False
        b1 = b2
    if b1 != 1:
        return False

    return True

def prime_test_miller_rabin(p, k):
    while k > 0:
        a = randint(1, p - 1)
        if not miller_rabin_witness(a, p):
            return False
        k = k - 1
    return True
```

其中miller_rabin_witness用于确认a是否为p为合数的证据，prime_test_miller_rabin共探测k次，每次随机产生一个1至p-1间的整数。只要有一次发现p为合数的证据就认为p为合数，否则认为p为素数。一些测试：

```python
print prime_test_miller_rabin(7, 5) #True
print prime_test_miller_rabin(21, 5) #False
print prime_test_miller_rabin(561, 50) #False
```

```
print prime_test_miller_rin(6864797660130609714981900799081393217269435300143305409394463459185543183397656052122559640661454554977296311391480858037121987999716643812574028291115057151, 50) #True
```

Miller-Rabin检测也同样存在假阳性的问题，但是与费马检测不同，MR检测的正确概率不依赖被检测数p（排除了Carmichael数失效问题），而仅依赖于检测次数。已经证明，如果一个数p为合数，那么Miller-Rabin检测的证据数量不少于比其小的正整数的3/4，换言之，k次检测后得到错误结果的概率为(1/4)^k，例如上面最后一个大整数，Miller-Rabin检测认为其实素数，我设k为50，也就是说它被误认为素数的概率为(1/4)^50。这个概率有多小呢，小到你不可想象。直观来说，大约等于一个人连续中得5次双色球头奖的概率。

# 参考文献


[1] 算法导论

[2] http://www.wikipedia.org/

[3] http://www.matrix67.com/blog/archives/234

[4] http://www.bigprimes.net/