PS：**本来没有决心把这个东西写完的，结果早上写到一半，出去吃个饭，没保存，回来手一抖直接关掉了，好不容易写了一大半了，只能重新写了，坑爹啊，但就是这个插曲，本来还没有决心的我，一下子却坚定了信念，一点要把这个东西写完。就这样开始吧**

BY：Lee

下面，我们重新开始

==========

如何判断一个数是否是素数呢

==========

也许你会认为这是一个简单的问题，但事实上，世界上任何一个问题，都没有你想象中的那么简单 1 + 1 是否等于 2 ，这便是一个简单而又复杂的问题，呵呵。

突然想把这个东西换一种风格来写了，就这样扯淡扯下去吧。扯的时候文章中多少有内容来自于网络，没有侵权的意思，如果作者看到还请见谅。

==========

下面正式进入正题

==========

一、朴素判断素数

==========

1. 这种方法被誉为 笨蛋的做法：

```
int isprime1(int n)
{
   if(n < 2)    return 0;   //小于2的数就不是质数了，这个不要忽略
   for(int i = 2; i <= n-1; ++i)
   {
      if(n%i == 0) return 0;// n 与所有 比 n 小的数相除，除得尽的话就是合数
   }
   return 1;              //都除不尽，为素数
}
```

一个数去除以比它的一半还要大的数，一定除不尽的，这还用判断吗？？

很容易发现的，这种方法判断素数，对于一个整数n，需要 n-2 次判断，时间复杂度是O(n)在n非常大或者测试量很大的时候，这种笨蛋做法肯定是不可取的。

2. 改进一下下　小学生的做法：

```
int isprime2(int n)
{
   if(n < 2) return 0;
   for(int i = 2; i < n/2+1; ++i)
   {
      if(n%i == 0) return 0;
   }
   return 1;
}
```

3. 再改进一下　聪明的小学生的做法

```
int isprime3(int n)
{
   if(n < 2) return 0;
   for(int i = 2; i*i <= n; ++i)     //for(int i = 2; i <= sqrt(n); ++i)
   {
      if(n%i == 0) return 0;
   }
   return 1;
}
```

对于一个小于n的整数X，如果n不能整除X,则n必定不能整除n/X。反之相同
一个明显的优化，就是只要从2枚举到 √ n 即可。
因为在判断2的同时也判断了n/2。到√ n时就把2到n-1都判断过了。

在这里，这个聪明的小学生还用了i*i <= n 来代替 sqrt（n），
这里是避免了调用函数sqrt（ ），其消耗时间很大，
特别是在大量数据测试的时候消耗很明显。
这个算法的时间复杂度，与最前面的笨蛋做法就好多了，

不过这里好像用sqrt()也没问题啊，，，，这个就不太清楚了。
但是做一个测试发现，如果是这样额话，每一次判断都要计算i*i，

而如果只调用sqrt()函数的话，只需要计算一次。故还是sqrt()函数好一些啊。
对于一个整数N，需要测试√ n-1 次，所以本算法的时间复杂度 O(√ n)。

4. 再改进一下 牛逼的小学生的做法

```
int isprime4(int n)
{
   if(n < 2) return 0;
   if(n == 2) return 1;
   for(int i = 3 ; i*i <= n   ; i+=2)   //for(int i = 3;   i <= sqrt(n);   ++i)
   {
      if(n%1 == 0) return 0;
   }
   return 1;
}
```

最后，来看一下这个牛逼的小学生，
确实，对于一个小学生，能够这么牛逼的想到这么多优化，
已经很强大了。
不过其实没什么必要。。。。。。
这里 的 i+=2，是因为，偶数除了2之外，是不可能是素数的、
所以从3开始，直接 +2 。进一步优化。
这个大概就是 朴素判断素数 方法的最佳优化了。（也许你还有更好的优化）

所以，如果是对于一般的素数判断的话，用上面那个代码吧

小学生毕业了，到了中学，会有怎样的成长呢？
下面来看看中学生们是怎么样判断的。

==========

二、 埃拉托斯特尼 筛选法

==========

埃拉托色尼选筛法(the Sieve of Eratosthenes)简称埃氏筛法，是古希腊数学家埃拉托色尼(Eratosthenes 274B.C.～194B.C.)提出的一种筛选法。

是针对自然数列中的自然数而实施的，用于求一定范围内的质数，
它的容斥原理之完备性条件是p=H~。
可参考：
http://zh.wikipedia.org/wiki/%E5%9F%83%E6%8B%89%E6%89%98%E6%96%AF%E7%89%B9%E5%B0%BC%E7%AD%9B%E6%B3%95

基本原理：
筛素数的基本方法是用来筛选出一定范围内的素数素数筛法的基本原理，利用的是素数p只有1和p这两个约数，并且一个数的约数一定不大于本身，素数筛法的过程：
把从1开始的、某一范围内的正整数从小到大顺序排列，
1不是素数，首先把它筛掉。
剩下的数中选择最小的数是素数，然后去掉它的倍数。
依次类推，直到筛子为空时结束。

求解用途：
素数筛法经常作为一道题的一部分用来打一定范围内素数表，
然后利用素数表作为基础解题。

*/

1. 老实的中学生的做法

```
#define MAX 10001
bool isprime[MAX];
void TheSieveofEratosthees()
{
   int i, j;
   for(i = 2; i < MAX; i++)
      isprime[i] = 1;
   for(i =2;   i< MAX; i++)
   {
      if(isprime[i])
         for(j = i+i;j < MAX; j+=i)
            isprime[j] = 0;
   }
}
```

执行完本算法之后，isprime[i]中如果是1，表示i为素数，0，表示i不是素数。

所以呢，这个算法执行完一遍之后，就可以在O(1)的时间内判断出MAX以内的任意数，是不是素数了，所以这个算法消耗的时间可以说全部在筛选上了。初看这个算法，会觉得这个算法的时间复杂度是O(N^2)，，但其实不是的，在第二个循环中，每次递增的i，当i越来也大的时候，j很快就能超过MAX的，筛选法的实际复杂度是 O(n*log(logn))

2. 有思想的中学生的做法

```
#define MAX 10001
bool isprime[MAX];
int p[MAX];
void prime(int n)
{
   memset(isprime, 0, sizeof(isprime));
   memset(p, 0, sizeof(p));
   int np = 0, i, j;
   for(i = 2; i <= n; i++)
   {
      if(!isprime[i]) p[np++] = i;
      for(j = 0; j< np &&p [j]*i <= n; j++)
      {
         isprime[p[j]*i] = 1;
         if(i % p[j] == 0) break;
      }
   }
}
```

*

这个算法的关键在于if(i % p[j] == 0) break;，它使得任何一个合数，只能被它最小的质因数标记过一次，再一次进行优化。所以整个算法是线性的。但考虑到log(log(100000000))还不到3，
故这个线性算法其实也只有理论的价值罢了。
其实我不这样认为。 这样其实可以当成素数表来用，因为定义了一个数组p，存放的都是素数。

讲到这里，你应该对判断素数这个问题有了一个新的认识了。
既要考虑时间上的问题。又要考虑空间上的问题。
也就是这并不是一个无脑问题。

==========

三、朴素判断 + 筛选法

==========

那位聪明的小学生已经将 朴素法优化到很好了。再深入理解，你确实会发现一个本质问题。
从2到√ n 中，存在很多不必要的判断，比如，n不能被2整除的话，n必然不能被4整除，必然不能被2的倍数整除。所以，我们再结合筛选法，优化处理小于√ n 的所有素数。这样在大量测试数据的时候，效率就提高很多了。

看看代码：

```
#define MAX XXXXX
bool isprime[MAX];
int p[MAX];
void prime(int n)
{
    memset(isprime, 0, sizeof(isprime));
    memset(p, 0, sizeof(p));
    int np = 0, i, j;
    for(i = 2; i <= n; i++)
    {
        if(!isprime[i]) p[np++] = i;
        for(j = 0; j < np && p[j]*i <= n; j++)
        {
            isprime[p[j]*i] = 1;
            if(i % p[j] == 0) break;
        }
    }
}

int Isprime(int n)
{
    if(n < 2) return 0;
```

```
    for(int i = 0; p[i]*p[i] <= n; i++)
        if(n%p[i] == 0)
            return 0;
    return 1;
}
```

上面这个代码，就是将 两种方法完美结合了，
上面的算法，总的时间复杂度理论上是0(√ n)。
但是字常数上已经得到很大的优化了，效率上也比原来的朴素快了好多。
别人统计是快了几十倍吧。这个我不清楚。

但是你会发现始终有一个问题，单纯用筛选法耗空间太多，用朴素法耗时间太多，有没有其他的办法？事实证明是有的。
下面将会接受两种方法，一种是费马测试，一种是米勒拉宾测试。
这两个测试就有点难了，至少对于我来说有点难了。

==========

四、费马素数测试

==========

==========

费马小定理：

有N为任意正整数，P为素数，且N不能被P整除（显然N和P互质），
则有：
N^P%P=N(即：N的P次方除以P的余数是N)

公式变形：
(N^(P-1))%P=1

网络分析：
后来分析了一下，两个式子其实是一样的，可以互相变形得到，
原式可化为：(N^P-N)%P=0
(即：N的P次方减N可以被P整除，因为由费马小定理知道N的P次方除以P的余数是N)

把N提出来一个，N^P就成了你N*(N^(P-1))，
那么(N^P-N)%P=0可化为：(N*(N^(P-1)-1))%P=0
请注意上式，含义是：N*(N^(P-1)-1)可以被P整除
又因为N*(N^(P-1)-1)必能整除N（这不费话么!）
所以，N*(N^(P-1)-1)是N和P的公倍数，小学知识了^_^

又因为前提是N与P互质，而互质数的最小公倍数为它们的乘积，
所以一定存在正整数M使得等式成立：
N*(N^(P-1)-1)=M*N*P
两边约去N，化简之：
N^(P-1)-1=M*P
因为M是整数，显然：
(N^(P-1)-1)%P=0
即：
N^(P-1)%P=1
==========

积模分解公式

先有一个引理，如果有：X%Z=0，即X能被Z整除，则有：
(X+Y)%Z=Y%Z
这个不用证了吧...

设有X、Y和Z三个正整数，则必有：(X*Y)%Z=((X%Z)*(Y%Z))%Z

想了很长时间才证出来，要分情况讨论才行：

1.当X和Y都比Z大时，必有整数A和B使下面的等式成立：
X=Z*I+A（1）
Y=Z*J+B（2）
不用多说了吧，这是除模运算的性质！
将（1）和（2）代入(X*Y)modZ得：((Z*I+A)(Z*J+B))%Z
乘开，再把前三项的Z提一个出来，变形为：(Z*(Z*I*J+I*A+I*B)+A*B)%Z（3）
因为Z*(Z*I*J+I*A+I*B)是Z的整数倍.....晕，又来了。
概据引理，（3）式可化简为：(A*B)%Z
又因为：A=X%Z，B=Y%Z，代入上面的式子，就成了原式了。

2.当X比Z大而Y比Z小时，一样的转化：

X=Z*I+A
代入(X*Y)%Z得：
(Z*I*Y+A*Y)%Z
根据引理，转化得：(A*Y)%Z
因为A=X%Z，又因为Y=Y%Z，代入上式，即得到原式。
同理，当X比Z小而Y比Z大时，原式也成立。

3.当X比Z小，且Y也比Z小时，X=X%Z，Y=Y%Z，所以原式成立。
==========

快速计算乘方的算法

如计算2^13，则传统做法需要进行12次乘法。
/*计算n^p*/

```
unsigned power(unsigned n,unsigned p)
{
    for(int i=0;i<p;i++) n*=n;
    return n;
}
```

把2*2的结果保存起来看看，是不是成了：4*4*4*4*4*4*2
再把4*4的结果保存起来：16*16*16*2
一共5次运算，分别是2*2、4*4和16*16*16*2

这样分析，我们算法因该是只需要计算一半都不到的乘法了。
为了讲清这个算法，再举一个例子2^7：2*2*2*2*2*2*2
两两分开：(2*2)*(2*2)*(2*2)*2
如果用2*2来计算，那么指数就可以除以2了，不过剩了一个，稍后再单独乘上它。
再次两两分开，指数除以2： ((2*2)*(2*2))*(2*2)*2
实际上最后一个括号里的2 * 2是这回又剩下的，那么，稍后再单独乘上它
现在指数已经为1了，可以计算最终结果了：16*4*2=128

优化后的算法如下：

```
unsigned Power(unsigned n,unsigned p)
{
   unsigned main=n; //用main保存结果
```

```
    unsigned odd=1; //odd用来计算“剩下的”乘积
    while (p>1)
    {//一直计算，直到指数小于或等于1
        if((p%2)!=0)
        {// 如果指数p是奇数，则说明计算后会剩一个多余的数，那么在这里把它乘到结果中
            odd*=main; //把“剩下的”乘起来
        }
        main*=main; //主体乘方
        p/=2; //指数除以2
    }
    return main*odd; //最后把主体和“剩下的”乘起来作为结果
}
```

够完美了吗？不，还不够！看出来了吗？main是没有必要的，
并且我们可以有更快的代码来判断奇数。
要知道除法或取模运算的效率很低，
所以我们可以利用偶数的一个性质来优化代码，
那就是偶数的二进制表示法中的最低位一定为0!
其实位运算这个东西我一直也没高太懂，有时间要系统得搞清楚

```
//完美版：
unsigned Power(unsigned n, unsigned p)
{ // 计算n的p次方
    unsigned odd = 1; //oddk用来计算“剩下的”乘积
    while (p > 1)
    { // 一直计算到指数小于或等于1
        if (( p & 1 )!=0)
        { // 判断p是否奇数，偶数的最低位必为0
            odd *= n; // 若odd为奇数，则把“剩下的”乘起来
        }
        n *= n; // 主体乘方
        p /= 2; // 指数除以2
    }
    return n * odd; // 最后把主体和“剩下的”乘起来作为结果
}
```

```
有一个东西叫做素数表，我们来看看一位中学生的做法
bool IsPrime2(unsigned n)
{
    if ( n < 2 )
    { // 小于2的数即不是合数也不是素数
        throw 0;
    }
    static unsigned aPrimeList[] = { // 素数表
        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, 113,
        193, 241, 257, 337, 353, 401, 433, 449, 577, 593, 641,
        673, 769, 881, 929, 977, 1009, 1153, 1201, 1217, 1249,
        1297,1361, 1409, 1489, 1553, 1601, 1697, 1777, 1873,
        1889, 2017, 2081, 2113, 2129, 2161, 2273, 2417, 2593,
        2609, 2657, 2689, 2753, 2801, 2833, 2897, 3041, 3089,
        3121, 3137, 3169, 3217, 3313, 3329, 3361, 3457, 3617,
        3697, 3761, 3793, 3889, 4001, 4049, 4129, 4177, 4241,
        4273, 4289, 4337, 4481, 4513, 4561, 4657, 4673, 4721,
        4801, 4817, 4993, 5009, 5153, 5233, 5281, 5297, 5393,
        5441, 5521, 5569, 5857, 5953, 6113, 6257, 6337, 6353,
        6449, 6481, 6529, 6577, 6673, 6689, 6737, 6833, 6961,
        6977, 7057, 7121, 7297, 7393, 7457, 7489, 7537, 7649,
        7681, 7793, 7841, 7873, 7937, 8017, 8081, 8161, 8209,
        8273, 8353, 8369, 8513, 8609, 8641, 8689, 8737, 8753,
        8849, 8929, 9041, 9137, 9281, 9377, 9473, 9521, 9601,
        9649, 9697, 9857
    };

    const int nListNum = sizeof(aPrimeList)/sizeof(unsigned);//计算素数表里元素的个数
    for (unsigned i=2;i<nListNum;++i )
    {
        if(n/2+1<aPrimeList[i])
        {
            return true;
        }
        if(0==n%aPrimeList[i])
```

```
        {
            return false;
        }
    }
    /*由于素数表中元素个数是有限的，那么对于用素数表判断不到的数，就只有用笨蛋办法了*/
    for (unsigned i=aPrimeList[nListNum-1];i<n/2+1;i++ )
    {
        if (0==n%i)
        { // 除尽了，合数
            return false;
        }
    }
    return true;
}
```

如果我们现在要对非常非常大的数进行判断，素数表显得无能为力了。
还有就是可以用动态的素数表来进行优化，
这就是大学生的做法了。
但是动态生成素数表的策略又复杂又没有效率，

所以我们还是直接跳跃到专家的做法吧：

根据上面讲到的费马小定理，对于两个互质的素数N和P，必有：N^(P-1)%P=1
那么我们通过这个性质来判断素数吧，当然，你会担心当P很大的时候乘方会很麻烦。不用担心！我们上面不是有个快速的幂模算法么？好好的利用蒙格马利这位大数学家为我们带来的快乐吧！

算法思路是这样的：
对于N，从素数表中取出任意的素数对其进行费马测试，
如果取了很多个素数，N仍未测试失败，那么则认为N是素数。当然，测试次数越多越准确，
但一般来讲50次就足够了。
另外，预先用“小学生”的算法构造一个包括500个素数的数组，
先对Q进行整除测试，
将会大大提高通过率，方法如下:

```
bool IsPrime3(unsigned n)
{
```

```
    if ( n < 2 )
    { // 小于2的数即不是合数也不是素数
        throw 0;
    }
    static unsigned aPrimeList[] = {
        2, 3, 5, 7, 11, 17, 19, 23, 29, 31, 41,
        43, 47, 53, 59, 67, 71, 73, 79, 83, 89, 97
    };
    const int nListNum = sizeof(aPrimeList) / sizeof(unsigned);
    for (int i=0;i<nListNum;++i)
    { // 按照素数表中的数对当前素数进行判断
        if (1!=Montgomery(aPrimeList[i],n-1,n)) // 蒙格马利算法
        {
            return false;
        }
    }
    return true;
}
```

****

OK，这就专家的作法了。

等等，什么？好像有点怪，看一下这个数29341，

它等于13 * 37 * 61，显然是一个合数，但是竟通过了测试！！

哦，抱歉，我忘了在素数表中加入13，37，61这三个数，

我其实是故意的，我只是想说明并费马测试并不完全可靠。

（PS：确实，费马测试毕竟只是一个测试。）

****

来看一下费马素数测试如何使用

```
int Montgomery(int n,int p,int m)          //蒙格马利快速幂模算法，用于后面素数判断
{ //快速计算(n^e)%m的值,即逐次平方法
   intk=1;
   n%=m;
   while(p!=1)
   {
     if(0!=(p&1))
```

```
        k=(k*n)%m;
        n=(n*n)%m;
        p>>=1;
    }
    return(n*k)%m;
}
void prime(int n)                    //生成一个我们想要的小范围的素数表，用于素数判断
{
    np = 0;
    for (int i = 2; i <= n; i++)
    {
        if (!isprime[i]) p[np++] = i;
        for (int j = 0; j < np && p[j]*i <= n; j++)
        {
            isprime[p[j]*i] = 1;
            if( i % p[j] == 0) break;
        }
    }
}
bool IsPrime(int n)                  //利用蒙格马利快速幂模算法来判断素数
{
    if ( n < 2 )
    { // 小于2的数即不是合数也不是素数
        return false;
    }
    for (int i=0;i<np;++i)
    { // 按照素数表中的数对当前素数进行判断
        if (1!=Montgomery(p[i],n-1,n))//蒙格马利算法
        {
            return false;
        }
    }
    return true;
}
```

我们发已经现了重要的一点，费马定理是素数的必要条件而非充分条件。

这种不是素数，但又能通过费马测试的数字还有不少，
数学上把它们称为卡尔麦克数，
现在数学家们已经找到所有10 ^ 16以内的卡尔麦克数，
最大的一个是9585921133193329。
我们必须寻找更为有效的测试方法。
数学家们通过对费马小定理的研究，并加以扩展，
总结出了多种快速有效的素数测试方法，
目前最快的算法是拉宾米勒测试算法，
下面介绍拉宾米勒测试。

*/

==========

五、米勒 -拉宾素性测试

==========

拉宾米勒测试是一个不确定的算法，只能从概率意义上判定一个数可能是素数，但并不能确保。算法流程如下:

1.选择T个随机数A，并且有A<N成立。
2.找到R和M，使得N=2*R*M+1成立。
快速得到R和M的方式：N用二进制数B来表示，令C=B-1。因为N为奇数（素数都是奇数），所以C的最低位为0，从C的最低位的0开始向高位统计，一直到遇到第一个1。这时0的个数即为R，M为B右移R位的值。
3.如果A^M%N=1，则通过A对于N的测试，然后进行下一个A的测试
4.如果A^M%N!=1，那么令i由0迭代至R，进行下面的测试
5.如果A^((2^i)*M)%N=N-1则通过A对于N的测试，否则进行下一个i的测试
6.如果i=r，且尚未通过测试，则此A对于N的测试失败，说明N为合数。
7.进行下一个A对N的测试，直到测试完指定个数的A

通过验证得知，当T为素数，并且A是平均分布的随机数，那么测试有效率为1 / (4 ^ T)。如果T > 8那么测试失误的机率就会小于10^(-5)，这对于一般的应用是足够了。如果需要求的素数极大，或着要求更高的保障度，可以适当调高T的值。
在网上找了很多代码，都感觉看不太懂。碰巧找到一个模板，相当好用。决定以后就背这个模板了。

```
__int64 qpow(int a,int b,int r)//快速幂
{
    __int64 ans=1,buff=a;
```

```
    while(b)
    {
        if(b&1)ans=(ans*buff)%r;
        buff=(buff*buff)%r;
        b>>=1;
    }
    return ans;
}
bool Miller_Rabbin(int n,int a)//米勒拉宾素数测试
{
    int r=0,s=n-1,j;
    if(!(n%a))
        return false;
    while(!(s&1)){
        s>>=1;
        r++;
    }
    __int64 k=qpow(a,s,n);
    if(k==1)
        return true;
    for(j=0;j<r;j++,k=k*k%n)
        if(k==n-1)
            return true;
    return false;
}
bool IsPrime(int n)//判断是否是素数
{
    int tab[]={2,3,5,7};
    for(int i=0;i<4;i++)
    {
        if(n==tab[i])
            return true;
        if(!Miller_Rabbin(n,tab[i]))
            return false;
    }
    return true;
}
```

```
bool RabbinMillerTest( unsigned n )
{
    if (n<2)
    { // 小于2的数即不是合数也不是素数
        throw 0;
    }

    const unsigned nPrimeListSize=sizeof(g_aPrimeList)/sizeof(unsigned);//求素数表
元素个数
    for(int i=0;i<nPrimeListSize;++i)
    {// 按照素数表中的数对当前素数进行判断
        if (n/2+1<=g_aPrimeList[i])
        {// 如果已经小于当前素数表的数，则一定是素数
            return true;
        }
        if (0==n%g_aPrimeList[i])
        {// 余数为0则说明一定不是素数
            return false;
        }
    }
    // 找到r和m，使得n = 2^r * m + 1;
    int r = 0, m = n - 1; // ( n - 1 ) 一定是合数
    while ( 0 == ( m & 1 ) )
    {
        m >>= 1; // 右移一位
        r++; // 统计右移的次数
    }
    const unsigned nTestCnt = 8; // 表示进行测试的次数
    for ( unsigned i = 0; i < nTestCnt; ++i )
    { // 利用随机数进行测试，
        int a = g_aPrimeList[ rand() % nPrimeListSize ];
        if ( 1 != Montgomery( a, m, n ) )
        {
            int j = 0;
            int e = 1;
```

```
            for ( ; j < r; ++j )
            {
                if ( n - 1 == Montgomery( a, m * e, n ) )
                {
                    break;
                }
                e <<= 1;
            }
            if (j == r)
            {
                return false;
            }
        }
    }
    return true;
}
```

由于能用逐次平方法在O(logn)的时间内算出a^b mod c.米勒拉宾的算法时间主要是花在这里了，所以米勒拉宾算法的时间复杂度是O(logn).对于朴素判断优化的O(√ n)来说，要快得多了。
感谢米勒拉宾这位大神吧。

```
// __int64 在USACO上用不了，直接改成long long
```

相关知识：

最大公约数只有 1 和它本身的数叫做质数（素数）——这个应该知道吧？-_-b

至今为止，没有任何人发现素数的分布规律，也没有人能用一个公式计算出所有的素数。关于素数的很多的有趣的性质或者科学家的努力

我不在这里多说，大家有兴趣的话可以到百度或 google 搜一下。我在下面列出了一个网址，上面只有个大概。更多的知识需要大家一点一点

地动手收集。

http://www.scitom.com.cn/discovery/universe/home01.html

1.高斯猜测，n 以内的素数个数大约与 n/ln(n)相当，或者说，当 n 很大时，两者数量级相同。这就是著名的素数定理。

2.十七世纪费马猜测，2 的 2^n 次方+1，n=0，1，2...时是素数，这样的数叫费马素数，可惜当 n=5 时，2^32+1 就不是素数，

至今也没有找到第六个费马素数。

3.18 世纪发现的最大素数是 2^31-1，19 世纪发现的最大素数是 2^127-1，20 世纪末人类已知的最大素数是 2^859433-1，用十进制表示，这是一个 258715 位的数字。
4.孪生素数猜想：差为 2 的素数有无穷多对。目前知道的最大的孪生素数是 1159142985×2^2304 - 1 和 1159142985×2^2304 + 1。
5.歌德巴赫猜想：大于 2 的所有偶数均是两个素数的和，大于 5 的所有奇数均是三个素数之和。其中第二个猜想是第一个的自然推论，因此歌德巴赫猜想又被称为 1+1 问题。我国数学家陈景润证明了 1+2，即所有大于 2 的偶数都是一个素数和只有两个素数因数的合数的和。国际上称为陈氏定理。

感谢那些在博客里分享自己的成果和点滴的人们，我们的世界，我们自己懂！！！！！！