# ****1.约定****

x%y为x取模y，即x除以y所得的余数，当x<y时，x%y=x，所有取模的运算对象都为整数。  
x^y表示x的y次方。乘方运算的优先级高于乘除和取模，加减的优先级最低。  
见到x^y/z这样，就先算乘方，再算除法。  
A/B，称为A除以B，也称为B除A。  
若A%B=0，即称为A可以被B整除，也称B可以整除A。  
A\*B表示A乘以B或称A乘B，B乘A，B乘以A……都一样。

复习一下小学数学  
公因数：两个不同的自然数A和B，若有自然数C可以整除A也可以整除B，那么C就是A和B的公因数。

公倍数：两个不同的自然数A和B，若有自然数C可以被A整除也可以被B整除，那么C就是A和B的公倍数。

互质数：两个不同的自然数，它们只有一个公因数1，则称它们互质。

费马是法国数学家，又译“费尔马”，此人巨牛，他的简介请看下面。不看不知道，一看吓一跳。

费马人物简介：<http://baike.baidu.com/view/6303430.htm?fromId=5194&redirected=seachword>

# 2.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)。

但是我查了很多资料见到的公式都是这个样子：

(N^(P-1))%P=1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  
============================================

# 3.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  
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。  
=====================================================

# 4.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。

**[cpp]** [view plain](http://blog.csdn.net/arvonzhang/article/details/8564836" \o "view plain)[copy](http://blog.csdn.net/arvonzhang/article/details/8564836" \o "copy)

1. /\*计算n^p\*/
2. unsigned power(unsigned n,unsigned p)
3. {
4. **for**(**int** i=0;i<p;i++) n\*=n;
5. **return** n;
6. }

该死的乘法，是时候优化一下了！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：

**[cpp]** [view plain](http://blog.csdn.net/arvonzhang/article/details/8564836" \o "view plain)[copy](http://blog.csdn.net/arvonzhang/article/details/8564836" \o "copy)

1. unsigned Power(unsigned n,unsigned p)
2. {
3. unsigned main=n; //用main保存结果
4. unsigned odd=1; //odd用来计算“剩下的”乘积
5. **while** (p>1)
6. {//一直计算，直到指数小于或等于1
7. **if**((p%2)!=0)
8. {// 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
9. 乘到结果中
10. odd\*=main; //把“剩下的”乘起来
11. }
12. main\*=main; //主体乘方
13. p/=2; //指数除以2
14. }
15. **return** main\*odd; //最后把主体和“剩下的”乘起来作为结果
16. }

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!

完美版：

**[cpp]** [view plain](http://blog.csdn.net/arvonzhang/article/details/8564836" \o "view plain)[copy](http://blog.csdn.net/arvonzhang/article/details/8564836" \o "copy)

1. unsigned Power(unsigned n, unsigned p)
2. { // 计算n的p次方
3. unsigned odd = 1; //oddk用来计算“剩下的”乘积
4. **while** (p > 1)
5. { // 一直计算到指数小于或等于1
6. **if** (( p & 1 )!=0)
7. { // 判断p是否奇数，偶数的最低位必为0
8. odd \*= n; // 若odd为奇数，则把“剩下的”乘起来
9. }
10. n \*= n; // 主体乘方
11. p /= 2; // 指数除以2
12. }
13. **return** n \* odd; // 最后把主体和“剩下的”乘起来作为结果
14. }

========================================================

# 5."蒙格马利”快速幂模算法

后面我们会用到这样一种运算：(X^Y)%Z。但问题是当X和Y很大时，只有32位的整型变量如何能够有效的计算出结果？

考虑上面那份最终的优化代码和再上面提到过的积模分解公式，我想你也许会猛拍一下脑门，吸口气说：“哦，我懂了！”。

下面的讲解是给尚没有做出这样动作的同学们准备的：

X^Y可以看作Y个X相乘，即然有积模分解公式，那么我们就可以把Y个X相乘再取模的过程分解开来，比如：(17^25)%29则可分解为：( ( 17 \* 17 ) % 29 \* ( 17 \* 17 ) % 29 \* ……

如果用上面的代码将这个过程优化，那么我们就得到了著名的“蒙格马利”快速幂模算法：

**[cpp]** [view plain](http://blog.csdn.net/arvonzhang/article/details/8564836" \o "view plain)[copy](http://blog.csdn.net/arvonzhang/article/details/8564836" \o "copy)

1. unsigned Montgomery(unsigned n, unsigned p, unsigned m)
2. { // 快速计算 (n ^ e) % m 的值，与power算法极类似
3. unsigned r = n % m; // 这里的r可不能省
4. unsigned k = 1;
5. **while** (p > 1)
6. {
7. **if** ((p & 1)!=0)
8. {
9. k = (k \* r) % m; // 直接取模
10. }
11. r = (r \* r) % m; // 同上
12. p /= 2;
13. }
14. **return** (r \* k) % m; // 还是同上
15. }

上面的代码还可以优化。下面是蒙格马利极速版：

**[cpp]** [view plain](http://blog.csdn.net/arvonzhang/article/details/8564836" \o "view plain)[copy](http://blog.csdn.net/arvonzhang/article/details/8564836" \o "copy)

1. unsigned Montgomery(unsigned n,unsigned p,unsigned m)
2. { //快速计算(n^p)%m的值
3. unsignedk=1;
4. n%=m;
5. **while**(p!=1)
6. {
7. **if**(0!=(p&1))k=(k\*n)%m;
8. n=(n\*n)%m;
9. p>>=1;
10. }
11. **return**(n\*k)%m;
12. }

=====================================================

# 6.怎么判断一个数是否为素数？

## 1)笨蛋的作法：

**[cpp]** [view plain](http://blog.csdn.net/arvonzhang/article/details/8564836" \o "view plain)[copy](http://blog.csdn.net/arvonzhang/article/details/8564836" \o "copy)

1. **bool** IsPrime(unsigned n)
2. {
3. **if** (n<2)
4. {
5. //小于2的数即不是合数也不是素数
6. **throw** 0;
7. }
8. **for** (unsigned i=2;i<n;++i)
9. {
10. //和比它小的所有的数相除，如果都除不尽，证明素数
11. **if** (n%i==0)
12. {
13. //除尽了，则是合数
14. **return** **false**;
15. }
16. }
17. **return** **true**;
18. }

一个数去除以比它的一半还要大的数，一定除不尽，所以还用判断吗？？

## 2)下面是小学生的做法：

**[cpp]** [view plain](http://blog.csdn.net/arvonzhang/article/details/8564836" \o "view plain)[copy](http://blog.csdn.net/arvonzhang/article/details/8564836" \o "copy)

1. **bool** IsPrime(unsigned n)
2. {
3. **if** (n<2)
4. {
5. //小于2的数即不是合数也不是素数
6. **throw** 0;
7. }
8. **for**(unsigned i=2;i<n/2+1;++i)
9. {
10. // 和比它的一半小数相除，如果都除不尽，证明素数
11. **if** ( 0 == n % i )
12. {
13. // 除尽了，合数
14. **return** **false**;
15. }
16. }
17. **return** **true**; // 都没除尽，素数
18. }

一个合数必然可以由两个或多个质数相乘而得到。那么如果一个数不能被比它的一半小的所有的质数整除，那么比它一半小的所有的合数也一样不可能整除它。建立一个素数表是很有用的。

## 3)下面是中学生的做法：

**[cpp]** [view plain](http://blog.csdn.net/arvonzhang/article/details/8564836" \o "view plain)[copy](http://blog.csdn.net/arvonzhang/article/details/8564836" \o "copy)

1. **bool** IsPrime2(unsigned n)
2. {
3. **if** ( n < 2 )
4. { // 小于2的数即不是合数也不是素数
5. **throw** 0;
6. }
7. **static** unsigned aPrimeList[] = { // 素数表
8. 1, 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41,
9. 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97, 113,
10. 193, 241, 257, 337, 353, 401, 433, 449, 577, 593, 641,
11. 673, 769, 881, 929, 977, 1009, 1153, 1201, 1217, 1249,
12. 1297,1361, 1409, 1489, 1553, 1601, 1697, 1777, 1873,
13. 1889, 2017, 2081, 2113, 2129, 2161, 2273, 2417, 2593,
14. 2609, 2657, 2689, 2753, 2801, 2833, 2897, 3041, 3089,
15. 3121, 3137, 3169, 3217, 3313, 3329, 3361, 3457, 3617,
16. 3697, 3761, 3793, 3889, 4001, 4049, 4129, 4177, 4241,
17. 4273, 4289, 4337, 4481, 4513, 4561, 4657, 4673, 4721,
18. 4801, 4817, 4993, 5009, 5153, 5233, 5281, 5297, 5393,
19. 5441, 5521, 5569, 5857, 5953, 6113, 6257, 6337, 6353,
20. 6449, 6481, 6529, 6577, 6673, 6689, 6737, 6833, 6961,
21. 6977, 7057, 7121, 7297, 7393, 7457, 7489, 7537, 7649,
22. 7681, 7793, 7841, 7873, 7937, 8017, 8081, 8161, 8209,
23. 8273, 8353, 8369, 8513, 8609, 8641, 8689, 8737, 8753,
24. 8849, 8929, 9041, 9137, 9281, 9377, 9473, 9521, 9601,
25. 9649, 9697, 9857
26. };
28. **const** **int** nListNum = **sizeof**(aPrimeList)/**sizeof**(unsigned);//计算素数表里元素的个数
29. **for** (unsigned i=2;i<nListNum;++i )
30. {
31. **if**(n/2+1<aPrimeList[i])
32. {
33. **return** **true**;
34. }
35. **if**(0==n%aPrimeList[i])
36. {
37. **return** **false**;
38. }
39. }
40. /\*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\*/
41. **for** (unsigned i=aPrimeList[nListNum-1];i<n/2+1;i++ )
42. {
43. **if** (0==n%i)
44. {
45. // 除尽了，合数
46. **return** **false**;
47. }
48. }
49. **return** **true**;
50. }

        还是太糟了，我们现在要做的对于大型素数的判断，那个素数表倒顶个P用！当然，我们可以利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:

## 6)下面是专家的做法：

**[cpp]** [view plain](http://blog.csdn.net/arvonzhang/article/details/8564836" \o "view plain)[copy](http://blog.csdn.net/arvonzhang/article/details/8564836" \o "copy)

1. **bool** IsPrime3(unsigned n)
2. {
3. **if** ( n < 2 )
4. {
5. // 小于2的数即不是合数也不是素数
6. **throw** 0;
7. }
8. **static** unsigned aPrimeList[] = {
9. 2, 3, 5, 7, 11, 17, 19, 23, 29, 31, 41,
10. 43, 47, 53, 59, 67, 71, 73, 79, 83, 89, 97
11. };
12. **const** **int** nListNum = **sizeof**(aPrimeList) / **sizeof**(unsigned);
13. **for** (**int** i=0;i<nListNum;++i)
14. {
15. // 按照素数表中的数对当前素数进行判断
16. **if** (1!=Montgomery(aPrimeList[i],n-1,n)) // 蒙格马利算法
17. {
18. **return** **false**;
19. }
20. }
21. **return** **true**;
22. }

        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。

        现在我们发现了重要的一点，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。

================================================================

# 7.拉宾米勒测试

        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。

下面是代码：

**[cpp]** [view plain](http://blog.csdn.net/arvonzhang/article/details/8564836" \o "view plain)[copy](http://blog.csdn.net/arvonzhang/article/details/8564836" \o "copy)

1. **bool** RabbinMillerTest( unsigned n )
2. {
3. **if** (n<2)
4. {
5. // 小于2的数即不是合数也不是素数
6. **throw** 0;
7. }
8. **const** unsigned nPrimeListSize=**sizeof**(g\_aPrimeList)/**sizeof**(unsigned);//求素数表元素个数
9. **for**(**int** i=0;i<nPrimeListSize;++i)
10. {
11. // 按照素数表中的数对当前素数进行判断
12. **if** (n/2+1<=g\_aPrimeList[i])
13. {
14. // 如果已经小于当前素数表的数，则一定是素数
15. **return** **true**;
16. }
17. **if** (0==n%g\_aPrimeList[i])
18. {
19. // 余数为0则说明一定不是素数
20. **return** **false**;
21. }
22. }
23. // 找到r和m，使得n = 2^r \* m + 1;
24. **int** r = 0, m = n - 1; // ( n - 1 ) 一定是合数
25. **while** ( 0 == ( m & 1 ) )
26. {
27. m >>= 1; // 右移一位
28. r++; // 统计右移的次数
29. }
30. **const** unsigned nTestCnt = 8; // 表示进行测试的次数
31. **for** ( unsigned i = 0; i < nTestCnt; ++i )
32. {
33. // 利用随机数进行测试，
34. **int** a = g\_aPrimeList[ rand() % nPrimeListSize ];
35. **if** ( 1 != Montgomery( a, m, n ) )
36. {
37. **int** j = 0;
38. **int** e = 1;
39. **for** ( ; j < r; ++j )
40. {
41. **if** ( n - 1 == Montgomery( a, m \* e, n ) )
42. {
43. **break**;
44. }
45. e <<= 1;
46. }
47. **if** (j == r)
48. {
49. **return** **false**;
50. }
51. }
52. }
53. **return** **true**;
54. }