Algorithm Tutorials

Primality Testing : Non-deterministic Algorithms

By **innocentboy**
TopCoder Member

Introduction

Primality testing of a number is perhaps the most common problem concerning number theory that topcoders deal with. A prime number is a natural number which has exactly two distinct natural number divisors: 1 and itself. Some basic algorithms and details regarding primality testing and factorization can be found here.

The problem of detecting whether a given number is a prime number has been studied extensively but nonetheless, it turns out that all the deterministic algorithms for this problem are too slow to be used in real life situations and the better ones amongst them are tedious to code. But, there are some probabilistic methods which are very fast and very easy to code. Moreover, the probability of getting a wrong result with these algorithms is so low that it can be neglected in normal situations.

This article discusses some of the popular probabilistic methods such as Fermat's test, Rabin-Miller test, Solovay-Strassen test.

Modular Exponentiation

All the algorithms which we are going to discuss will require you to efficiently compute $(a^b)\%c$ (where a,b,c are non-negative integers). A straightforward algorithm to do the task can be to iteratively multiply the result with 'a' and take the remainder with 'c' at each step.

```
/* a function to compute (a^b)%c */
int modulo(int a,int b,int c){
    // res is kept as long long because intermediate results might overflow in "int"
    long long res = 1;
    for(int i=0;i<b;i++){
        res *= a;
        res %= c; // this step is valid because (a*b)%c = ((a%c)*(b%c))%c
    }
    return res%c;
}
```

However, as you can clearly see, this algorithm takes O(b) time and is not very useful in practice. We can do it in O(log(b)) by using what is called as exponentiation by squaring. The idea is very simple:

$$a^b = \begin{cases} (a^2)^{(b/2)} & \text{if b is even and b > 0} \\ a*(a^2)^{((b-1)/2)} & \text{if b is odd} \\ 1 & \text{if b = 0} \end{cases}$$

This idea can be implemented very easily as shown below:

```
/* This function calculates (a^b)%c */
int modulo(int a,int b,int c){
    long long x=1,y=a; // long long is taken to avoid overflow of intermediate results
    while(b > 0){
        if(b%2 == 1){
            x=(x*y)%c;
        }
        y = (y*y)%c; // squaring the base
        b /= 2;
    }
    return x%c;
}
```

Notice that after i iterations, b becomes $b/(2^i)$, and y becomes $(y^{(2^i)})\%c$. Multiplying x with y is equivalent to adding 2^i to the overall power. We do this if the i^{th} bit from right in the binary representation of b is 1. Let us take an example by computing $(7^{107})\%9$. If we use the above code, the variables after each iteration of the loop would look like this: (a = 7, c = 9)

```
iterations     b     x     y
    0    107     1     7
    1     53     7     4
    2     26     1     7
    3     13     1     4
    4      6     4     7
    5      3     4     4
    6      1     7     7
    7      0           4     4
```

Now b becomes 0 and the return value of the function is 4. Hence $(7^{107})\%9 = 4$.

The above code could only work for a,b,c in the range of type "int" or the intermediate results will run out of the range of "long long". To write a function for numbers up to 10^18, we need to compute (a*b)%c when computing a*b directly can grow larger than what a long long can handle. We can use a similar idea to do that:

```
      (2*a)*(b/2)                  if b is even and b > 0
a*b = a + (2*a)*((b-1)/2)     if b is odd
```

```
    0       if b = 0
```

Here is some code which uses the idea described above (you can notice that its the same code as exponentiation, just changing a couple of lines):

```
/* this function calculates (a*b)%c taking into account that a*b might overflow */
long long mulmod(long long a,long long b,long long c){
    long long x = 0,y=a%c;
    while(b > 0){
        if(b%2 == 1){
            x = (x+y)%c;
        }
        y = (y*2)%c;
        b /= 2;
    }
    return x%c;
}
```

We could replace x=(x*y)%c with x = mulmod(x,y,c) and y = (y*y)%c with y = mulmod(y,y,c) in the original function for calculating (a^b)%c. This function requires that 2*c should be in the range of long long. For numbers larger than this, we could write our own BigInt class (java has an inbuilt one) with addition, multiplication and modulus operations and use them.

This method for exponentiation could be further improved by using Montgomery Multiplication. Montgomery Multiplication algorithm is a quick method to compute (a*b)%c, but since it requires some pre-processing, it doesn't help much if you are just going to compute one modular multiplication. But while doing exponentiation, we need to do the pre-processing for 'c' just once, that makes it a better choice if you are expecting very high speed. You can read about it at the links mentioned in the reference section.

Similar technique can be used to compute (a^b)%c in O(n^3 * log(b)), where a is a square matrix of size n x n. All we need to do in this case is manipulate all the operations as matrix operations. Matrix exponentiation is a very handy tool for your algorithm library and you can see problems involving this every now and then.

Fermat Primality Test

Fermat's Little Theorem
According to Fermat's Little Theorem if p is a prime number and a is a positive integer less than p, then

$$a^p = a \pmod p$$
or alternatively:
$$a^{(p-1)} = 1 \pmod p$$

Algorithm of the test
If p is the number which we want to test for primality, then we could randomly choose a, such that a < p and then calculate ($a^{(p-1)}$)%p. If the result is not 1, then by Fermat's Little Theorem p cannot be prime. What if that is not the case? We can choose another a and then do the same test again. We could stop after some number of iterations and if the result is always 1 in each of them, then we can state with very high probability that p is prime. The more iterations we do, the higher is the probability that our result is correct. You can notice that if the method returns composite, then the number is sure to be composite, otherwise it will be probably prime.

Given below is a simple function implementing Fermat's primality test:

```
/* Fermat's test for checking primality, the more iterations the more is accuracy */
bool Fermat(long long p,int iterations){
    if(p == 1){ // 1 isn't prime
        return false;
    }
    for(int i=0;i<iterations;i++){
        // choose a random integer between 1 and p-1 ( inclusive )
        long long a = rand()%(p-1)+1;
        // modulo is the function we developed above for modular exponentiation.
        if(modulo(a,p-1,p) != 1){
            return false; /* p is definitely composite */
        }
    }
    return true; /* p is probably prime */
}
```

More iterations of the function will result in higher accuracy, but will take more time. You can choose the number of iterations depending upon the application.

Though Fermat is highly accurate in practice there are certain composite numbers p known as Carmichael numbers for which all values of a<p for which gcd(a,p)=1, ($a^{(p-1)}$)%p = 1. If we apply Fermat's test on a Carmichael number the probability of choosing an a such that gcd(a,p) != 1 is very low (based on the nature of Carmichael numbers), and in that case, the Fermat's test will return a wrong result with very high probability. Although Carmichael numbers are very rare (there are about 250,000 of them less than 10^{16}), but that by no way means that the result you get is always correct. Someone could easily challenge you if you were to use Fermat's test :). Out of the Carmichael numbers less than 10^{16}, about 95% of them are divisible by primes < 1000. This suggests that apart from applying Fermat's test, you may also test the number for divisibility with small prime numbers and this will further reduce the probability of failing. However, there are other improved primality tests which don't have this flaw as Fermat's. We will discuss some of them now.

Miller-Rabin Primality Test

Key Ideas and Concepts

1. Fermat's Little Theorem.
2. If p is prime and x^2 = 1 (mod p), then x = +1 or -1 (mod p). We could prove this as follows:

$$x^2 = 1 \pmod p$$
$$x^2 - 1 = 0 \pmod p$$
$$(x-1)(x+1) = 0 \pmod p$$

Now if p does not divide both (x-1) and (x+1) and it divides their product, then it cannot be a prime, which is a contradiction. Hence, p will either divide (x-1) or it will divide (x+1), so x = +1 or -1 (mod p).

Let us assume that p - 1 = 2^d * s where s is odd and d >= 0. If p is prime, then either a^s = 1 (mod p) as in this case, repeated squaring from a^s will always yield 1, so $(a^{(p-1)})$%p will be 1; or $a^{(s*(2^r))}$ = -1 (mod p) for some r such that 0 <= r < d, as repeated squaring from it will always yield 1 and finally $a^{(p-1)}$ = 1 (mod p). If none of these hold true, $a^{(p-1)}$ will not be 1 for any prime number a (otherwise there will be a contradiction with fact #2).

Algorithm
Let p be the given number which we have to test for primality. First we rewrite p-1 as (2^d)*s. Now we pick some a in range [1,n-1] and then check whether a^s = 1 (mod p) or $a^{(s*(2^r))}$ = -1 (mod p). If both of them fail, then p is definitely composite. Otherwise p is probably prime. We can choose another a and repeat the same test. We can stop after some fixed number of iterations and claim that either p is definitely composite, or it is probably prime.

A small procedure realizing the above algorithm is given below:

```
/* Miller-Rabin primality test, iteration signifies the accuracy of the test */
bool Miller(long long p,int iteration){
    if(p<2){
        return false;
    }
    if(p!=2 && p%2==0){
        return false;
    }
    long long s=p-1;
    while(s%2==0){
        s/=2;
    }
    for(int i=0;i<iteration;i++){
        long long a=rand()%(p-1)+1,temp=s;
        long long mod=modulo(a,temp,p);
        while(temp!=p-1 && mod!=1 && mod!=p-1){
            mod=mulmod(mod,mod,p);
            temp *= 2;
        }
        if(mod!=p-1 && temp%2==0){
            return false;
        }
    }
    return true;
}
```

It can be shown that for any composite number p, at least (3/4) of the numbers less than p will witness p to be composite when chosen as 'a' in the above test. Which means that if we do 1 iteration, probability that a composite number is returned as prime is (1/4). With k iterations the probability of test failing is $(1/4)^k$ or $4^{(-k)}$. This test is comparatively slower compared to Fermat's test but it doesn't break down for any specific composite numbers and 18-20 iterations is a quite good choice for most applications.

Solovay-Strassen Primality Test

Key Ideas and Concepts

1. Legendre Symbol: This symbol is defined for a pair of integers a and p such that p is prime. It is denoted by (a/p) and calculated as:

   ```
         = 0    if a%p = 0
   (a/p) = 1    if there exists an integer k such that k^2 = a ( mod p )
         = -1   otherwise.
   ```

   It is proved by Euler that:

   $(a/p) = (a^{((p-1)/2))} \% p$

   So we can also say that:

   $(ab/p) = (ab^{((p-1)/2))} \% p = (a^{((p-1)/2))}\%p * (b^{((p-1)/2))}\%p = (a/p)*(b/p)$

2. Jacobian Symbol: This symbol is a generalization of Legendre Symbol as it does not require 'p' to be prime. Let a and n be two positive integers, and n = $p1^{k1}$ * .. * pn^{kn}, then Jacobian symbol is defined as:

   $(a/n) = ((a/p1)^{k1}) * ((a/p2)^{k2}) * ..... * ((a/pn)^{kn})$

   So you can see that if n is prime, the Jacobian symbol and Legendre symbol are equal.
   There are some properties of these symbols which we can exploit to quickly calculate them:
   1. (a/n) = 0 if gcd(a,n) != 1, Hence (0/n) = 0. This is because if gcd(a,n) != 1, then there must be some prime pi such that pi divides both a and n. In that case (a/pi) = 0 [by definition of Legendre Symbol].
   2. (ab/n) = (a/n) * (b/n). It can be easily derived from the fact (ab/p) = (a/p)(b/p) (here (a/p) is the Legendry Symbol).
   3. if a is even, than (a/n) = (2/n)*((a/2)/n). It can be shown that:

      ```
            = 1 if n = 1 ( mod 8 ) or n = 7 ( mod 8 )
      (2/n) = -1 if n = 3 ( mod 8 ) or n = 5 ( mod 8 )
            = 0 otherwise
      ```

4. (a/n) = (n/a)*(-1$^{((a-1)(n-1)/4))}$ if a and n are both odd.

The algorithm for the test is really simple. We can pick up a random a and compute (a/n). If n is a prime then (a/n) should be equal to (a$^{((n-1)/2))}$%n [as proved by Euler]. If they are not equal then n is composite, and we can stop. Otherwise we can choose more random values for a and repeat the test. We can declare n to be probably prime after some iterations.

Note that we are not interested in calculating Jacobi Symbol (a/n) if n is an even integer because we can trivially see that n isn't prime, except 2 of course.

Let us write a little code to compute Jacobian Symbol (a/n) and for the test:

```
//calculates Jacobian(a/n) n>0 and n is odd
int calculateJacobian(long long a,long long n){
    if(!a) return 0; // (0/n) = 0
    int ans=1;
    long long temp;
    if(a<0){
        a=-a;    // (a/n) = (-a/n)*(-1/n)
        if(n%4==3) ans=-ans; // (-1/n) = -1 if n = 3 ( mod 4 )
    }
    if(a==1) return ans; // (1/n) = 1
    while(a){
        if(a<0){
            a=-a;    // (a/n) = (-a/n)*(-1/n)
            if(n%4==3) ans=-ans;    // (-1/n) = -1 if n = 3 ( mod 4 )
        }
        while(a%2==0){
            a=a/2;    // Property (iii)
            if(n%8==3||n%8==5) ans=-ans;
        }
        swap(a,n);    // Property (iv)
        if(a%4==3 && n%4==3) ans=-ans; // Property (iv)

        a=a%n; // because (a/p) = (a%p / p ) and a%pi = (a%n)%pi if n % pi = 0
        if(a>n/2) a=a-n;
    }
    if(n==1) return ans;
    return 0;
}

/* Iterations determine the accuracy of the test */
bool Solovoy(long long p,int iteration){
    if(p<2) return false;
    if(p!=2 && p%2==0) return false;
    for(int i=0;i<iteration;i++){
        long long a=rand()%(p-1)+1;
        long long jacobian=(p+calculateJacobian(a,p))%p;
        long long mod=modulo(a,(p-1)/2,p);
        if(!jacobian || mod!=jacobian){
            return false;
        }
    }
    return true;
}
```

It is shown that for any composite n, at least half of the a will result in n being declared as composite according to Solovay-Strassen test. This shows that the probability of getting a wrong result after k iterations is (1/2)^{k}. However, it is generally less preferred than Rabin-Miller test in practice because it gives poorer performance.

The various routines provided in the article can be highly optimized just by using bitwise operators instead of them. For example /= 2 can be replaced by ">>= 1", "%2" can be replaced by "&1" and "*= 2" can be replaced by "<<=1". Inline Assembly can also be used to optimize them further.

Practice Problems

Problems involving non-deterministic primality tests are not very suitable for the SRM format. But problems involving modular exponentiation and matrix exponentiation are common. Here are some of the problems where you can apply the methods studied above:

PowerDigit (TCO 06 Online Round 2)
MarbleMachine (SRM 376)
DrivingAround (SRM 342)
PON
PRIC
SOLSTRAS
DIVSUM2 [this one also involves Pollard's Rho Algorithm]

References and Further Reading

http://www.cse.iitk.ac.in/users/manindra/algebra/primality_v6.pdf
http://security.ece.orst.edu/papers/j37acmon.pdf
http://icpc.baylor.edu/Past/icpc2004/RegReport/guan.cse.nsysu.edu.tw/data/montg.pdf