原创 威尔逊定理&费马小定理

2019-06-07 17:20:15 _-Y-_-Y-_ 阅读数 73 更多

威尔逊定理和费马小定理

Wilson’s Theorem

如果p是素数，则 $(p-1)! \equiv -1(mod\ \ p)$
例如 当$p=11$时
$(p-1)!=10\times 9\times 8\times 7\times 6\times 5\times 4\times 3\times 2\times 1=1\times (2\times 6)\times (3\times 4)\times (5\times 9)\times (7\times 8)\times 10$
$10!\equiv -1(mod 11)$

证明：当$p=2$时，（$p-1$）！ $\equiv -1(mod\ \ p)$
当$p>2$且p是素数时，因为同余方程 $ax\equiv 1(mod\ \ m)$有解当且仅当$GCD(a,m)=1$ 且所有解都模m同余
又因为 p是素数，正整数a是其自身模p的逆当且仅当$a\equiv 1$（$mod\ \ p$）或者 $a\equiv -1$（$mod\ \ p$）
因为$1\le\ a<\ p$ 所以 模p的逆是自身数的只有1和$p-1$，因此可以吧2到 $p-2$ 分成$\frac{p-3}{2}$组整数对，每组乘以模p余1;
所以$2\times 3\times ...(p-2)\equiv 1(mod\ \ p)$, 则可以推出$(p-1)!=-1(mod\ \ p)$

费马小定理

如果 p是素数， a是正整数，且$GCD(a,p)=1$， 则 $a^{p-1}\equiv 1(mod\ \ p)$

证明 $p-1$ 整数 $a,2a,...(p-1)a$是不能被p整除的，且其中任何两个数模 p不同余。
所以，（$p-1$）整数 $a,2a,...(p-1)a$模p的余数为$1,2...p-1$。
因此$a\times 2a\times 3a\times 4a...\times (p-1)a\equiv 1\times 2\times 3\times ...\times p-1(mod\ \ p)$
即 $a^{p-1}\times$（$p-1)!\equiv (p-1)!$ $(modp)$
因为 $GCD((p-1)!$ ， $p)=1$，又因为 $ac\equiv bc(mod\ \ m)a\equiv b(mod\ \ m)$。
可推出
$a^{p-1}\equiv 1(mod\ \ p)$

如果 p 是素数， a是正整数，$a^p\equiv a(mod\ \ p)$.
如果 $a=3,p=5$，则 $3^4\equiv 1(mod\ \ 5)$
如果 $a=6,p=3$，则 $6^3\equiv 6(mod\ \ 3)$

例1

https://cn.vjudge.net/problem/ZOJ-3785

AC code

第一种方法： 暴力跑，找规律

```
#include <algorithm>
#include <cstring>
#include <cstdio>
#include <iostream>
using namespace std;
char c[8][10]={"Saturday","Sunday","Monday","Tuesday","Wednesday","Thursday","Friday"};
typedef long long ll;
ll pow(ll x,ll n,ll mod)//快速幂
{
    ll res=1;
    while(n>0)
    {
       if(n%2==1)
       {
         res=res*x;
         res=res%mod;
       }
       x=x*x;
```

```
      x=x%mod;
      n>>=1;
    }
    return res;
}
int main(){
  int m;
  while(1){
    scanf("%d", &m);
    int ans=0;
    for(int i=1;i<=500;i++){
      printf("%d ", ans);
      ans=(ans+pow(i,i,7))%7;
      if(i%m==0) cout<<endl;
    }
  }
  return 0;
}
```

第二种，算出循环周期

因为 $(n^n)\%7 = \{(n\%7)^n\}\%7$

令 $m = n\%7$ 因为 $GCD(m,7) = 1$ 根据费马小定理 $(m^6)\%7 = 1$，所以$m^n = k \times n^6 \times n^t$

令 $t = n\%6$ 所以 $(n^n)\%7 = \{(n\%7)^n\}\%7 = (m^t)\%7$

$0 \le m < 7, 0 \le t < 6$ 所以一共有 42 种 又因为其实位置不同，所以一共有 $42 \times 7 = 294$ 种

```
#include <algorithm>
#include <cstring>
#include <cstdio>
using namespace std;
int mp[500]={0,1,5,4,1,4,5,5,6,0,4,6,0,6,6,0,2,0,1,6,0,0,1,5,6,3,0,6,6,0,1,4,6,5,6,6,0,2,4,5,
0,6,6,0,4,3,0,3,4,4,5,6,3,5,6,5,5,6,1,6,0,5,6,6,0,4,5,2,6,5,5,6,0,3,5,4,5,5,6,1,
3,4,6,5,5,6,3,2,6,2,3,3,4,5,2,4,5,4,4,5,0,5,6,4,5,5,6,3,4,1,5,4,4,5,6,2,4,3,4,4,
5,0,2,3,5,4,4,5,2,1,5,1,2,2,3,4,1,3,4,3,3,4,6,4,5,3,4,4,5,2,3,0,4,3,3,4,5,1,3,2,
3,3,4,6,1,2,4,3,3,4,1,0,4,0,1,1,2,3,0,2,3,2,2,3,5,3,4,2,3,3,4,1,2,6,3,2,2,3,4,0,
2,1,2,2,3,5,0,1,3,2,2,3,0,6,3,6,0,0,1,2,6,1,2,1,1,2,4,2,3,1,2,2,3,0,1,5,2,1,1,2,
3,6,1,0,1,1,2,4,6,0,2,1,1,2,6,5,2,5,6,6,0,1,5,0,1,0,0,1,3,1,2,0,1,1,2,6,0,4,1,0,
0,1,2,5,0,6,0,0,1,3,5,6,1,0
};
char c[8][10]={"Saturday","Sunday","Monday","Tuesday","Wednesday","Thursday","Friday"};
typedef long long ll;
ll pow(ll x,ll n,ll mod)//快速幂
{
    ll res=1;
    while(n>0)
    {
       if(n%2==1)
       {
         res=res*x;
         res=res%mod;
       }
       x=x*x;
       x=x%mod;
       n>>=1;
    }
    return res;
}
int main(){
  int t,n;
  scanf("%d", &t);
  mp[1]=1;
  while(t--){
    scanf("%d", &n);
    printf("%s\n", c[mp[n%294]]);
  }
  return 0;
}

```

Miller-Rabin 素性测试

（伪素数）设 a 是一个正整数，如果n是一个正合数，并且 $a^n \equiv a(mod\ \ n)$，则称 n 为以 a 为基的伪素数。
（绝对伪素数） 如果一个正合数 n 对于所有满足 $GCD（a, \ n） = 1$的正整数 a 都有
$a^{n-1} \equiv 1（mod\ \ n）$也成为Carmichael数

来自 https://www.cnblogs.com/dalt/p/8436883.html
费马小定理中指出，对于任意素数p，以及对于模p的剩余类环 $1, 2, ..., p-1$ 中的任意数x，
都满足 $x^p = x\,(mod\ \ p)$。
因此我们可以用这个小小的技巧来排除大量的合数。
譬如对于素数5，我们发现 $(2^5)\%5 = 2$，而对于6，有 $(2^6)\%6 = 4$。
但是费马小定理中p是素数是 pow (x , p)=x(mod p) 的充分条件，而非必要条件
比如$5^6 = 5\,(mod\ \ 6)$，但我们不能说6是素数
因此我们需要从模p的剩余类环中选取更多的数进行测试，以增强结果的可信度，
只要存在一个数x不满足 $x^p = x\,(mod\ \ p)$，那么p就绝不可能是素数。
但是还是存在一类极端的合数 p，使得对于任意 $1, .., p-1$ 中的 x 都满足 $x^p = x\,(mod\ \ p)$，
这类合数称为Carmichael数，一个例子就是561。
由于这类数的存在，使得我们用费马小定理完全无法正确断定一个数为素数还是合数。

而Miller-Rabin算法的出世使得相当一类的满足费马小定理的合数无法通过素数测试。
Miller-Rabin算法基于一个事实，若$x^2 = 1\ \ (mod\ \ p)$，那么若p是素数，则$(x-1)(x+1) = 0\,(mod\ \ p)$，除非$p$为$2$，
否则(x-1)与(x+1)在模p的性质下是不相等的，无论p是否为2，都可以保证有(x-1)=0(mod p)或者(x+1)=0(mod p)（因为模p剩余类环是整环），即x
$p-1$。
因此我们可以在p通过数x的费马测试后，即$x^p = p\,(mod\ \ p)$，也可以写作$x^{p-1} = 1\,(mod\ \ p)$，
若$p-1$是偶数，那么可以继续通过$x^{((p-1)/2)}$ 是否等于 1或$(p-1)$ 来进行测试。
如果测试通过还可以继续判断是否满足 $x^{2k} = 1(modp)$，从而继续进行判断。
只要一环判断不通过，那么就保证p是合数。
例如
$2^{560} = 1\,(mod\ \ 561)$ 满足
$2^{280} = 1\,(mod\ \ 561)$ 满足
$2^{140} = 67\,(mod\ \ 561)$ 不满足

算法流程

（1）对于偶数和$0，1，2$ 可以直接判断。
（2）设要测试的数为 x 我们取一个较小的质数a 设s, t 满足 $2^s \times t = x-1$（其中t是奇数）。
（3）我们先算出 a^t，然后不断地平方并且进行二次探测（进行 s 次）。
（4）如果最后不满足费马小定律则说明 x 为合数。
（5）多次取不同的 a 进行 Miller-Rabin 素数测试，这样可以使正确性更高

备注

（1）我们可以多选择几个 a，如果全部通过，那么 x大概率是质数。

（2） Miller-Rabin 素数测试中，“大概率”意味着概率非常大，基本上可以放心使用。

（3）当 a 取遍小等于 30 的所有素数时，可以证明 int 范围内的数不会出错。

（4）代码中我用的 int 类型，不过实际上 Miller-Rabin 素数测试可以承受更大的范围。

（5）另外，如果是求一个 long long 类型的平方，可能会爆掉，因此有时我们要用 龟速乘，不能直接乘。

code

```
#include <bits/stdc++.h>
using namespace std;
int prime[10]={2,3,5,7,11,13,17,19,23,29};
typedef long long ll;
ll pow(ll x,ll n,ll mod){//快速幂
    ll res=1;
    while(n>0){
        if(n%2==1){
          res=res*x;
          res=res%mod;
        }
        x=x*x;
        x=x%mod;
        n>>=1;
```

```
    }
    return res;
}
ll mulit(ll a,ll b,ll c){//龟速乘
    ll ans=0;
    ll res=a;
    while(b){
      if(b&1)
        ans=(ans+res)%c;
      res=(res+res)%c;
      b>>=1;
    }
    return ans;
}
bool Miller_Rabin(int x)     //判断素数
{
    int i,j,k;
    int s=0,t=x-1;
    if(x==2)  return true;   //2是素数
    if(x<2||!(x&1))  return false;     //如果x是偶数或者是0,1，那它不是素数
    while(!(t&1))  //将x分解成(2^s)*t的样子
    {
        s++;
        t>>=1;
    }
    for(i=0;i<10&&prime[i]<x;++i)      //随便选一个素数进行测试
    {
        int a=prime[i];
        int b=pow(a,t,x);      //先算出a^t
        for(j=1;j<=s;++j)    //然后进行s次平方
        {
            k=mulit(b,b,x);   //求b的平方
            if(k==1&&b!=1&&b!=x-1)     //用二次探测判断
              return false;
            b=k;
        }
        if(b!=1)  return false;   //用费马小定律判断
    }
    return true;   //如果进行多次测试都是对的，那么x就很有可能是素数
}
int main()
{
    int x;
    scanf("%d",&x);
    if(Miller_Rabin(x))  printf("Yes");
    else  printf("No");
    return 0;
}
```

例 2

https://cn.vjudge.net/problem/POJ-3641#author=ChineseOJ

AC code

```
#include <algorithm>
#include <cstdio>
#include <cstring>
using namespace std;
long long a,p;
int prime[10]={2,3,5,7,11,13,17,19,23,29};
typedef long long ll;
ll pow(ll x,ll n,ll mod){//快速幂
    ll res=1;
    while(n>0){
       if(n%2==1){
         res=res*x;
         res=res%mod;
       }
```

```
            x=x*x;
            x=x%mod;
            n>>=1;
        }
        return res;
}
ll mulit(ll a,ll b,ll c){//龟速乘
        ll ans=0;
        ll res=a;
        while(b){
          if(b&1)
            ans=(ans+res)%c;
          res=(res+res)%c;
          b>>=1;
        }
        return ans;
}
bool Miller_Rabin(int x)      //判断素数
{
    int i,j,k;
    int s=0,t=x-1;
    if(x==2)  return true;   //2是素数
    if(x<2||!(x&1))  return false;     //如果x是偶数或者是0,1，那它不是素数
    while(!(t&1))  //将x分解成(2^s)*t的样子
    {
        s++;
        t>>=1;
    }
    for(i=0;i<10&&prime[i]<x;++i)      //随便选一个素数进行测试
    {
        int a=prime[i];
        int b=pow(a,t,x);      //先算出a^t
        for(j=1;j<=s;++j)    //然后进行s次平方
        {
            k=mulit(b,b,x);   //求b的平方
            if(k==1&&b!=1&&b!=x-1)     //用二次探测判断
              return false;
            b=k;
        }
        if(b!=1)  return false;   //用费马小定律判断
    }
    return true;   //如果进行多次测试都是对的，那么x就很有可能是素数
}
bool judge(){
  if(Miller_Rabin(p)) return false;
  if(pow(a,p,p)==a) return true;
  else return false;
}
int main(){
  while(scanf("%ld %lld", &p, &a)&&a||p){
    if(judge()) printf("yes\n");
    else printf("no\n");
  }
  return 0;
}
```

例3

https://cn.vjudge.net/problem/HDU-2138

AC code

```
#include <algorithm>
#include <cstdio>
#include <cstring>
using namespace std;
int prime[10]={2,3,5,7,11,13,17,19,23,29};
typedef long long ll;
ll pow(ll x,ll n,ll mod){//快速幂
    ll res=1;
```

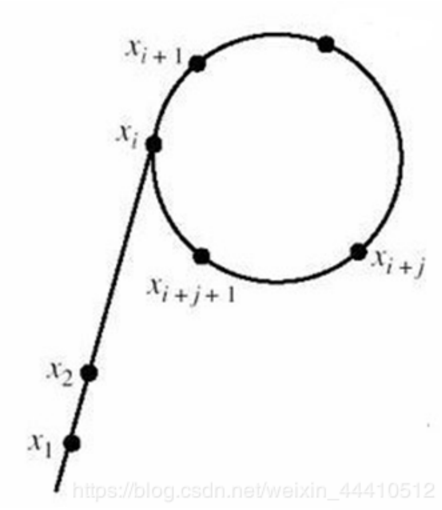
```
    while(n>0){
       if(n%2==1){
         res=res*x;
         res=res%mod;
       }
       x=x*x;
       x=x%mod;
       n>>=1;
    }
    return res;
}
ll mulit(ll a,ll b,ll c){//龟速乘
    ll ans=0;
    ll res=a;
    while(b){
      if(b&1)
        ans=(ans+res)%c;
      res=(res+res)%c;
      b>>=1;
    }
    return ans;
}
bool Miller_Rabin(int x)     //判断素数
{
    int i,j,k;
    int s=0,t=x-1;
    if(x==2)  return true;   //2是素数
    if(x<2||!(x&1))  return false;     //如果x是偶数或者是0,1，那它不是素数
    while(!(t&1))  //将x分解成(2^s)*t的样子
    {
        s++;
        t>>=1;
    }
    for(i=0;i<10&&prime[i]<x;++i)      //随便选一个素数进行测试
    {
        int a=prime[i];
        int b=pow(a,t,x);      //先算出a^t
        for(j=1;j<=s;++j)    //然后进行s次平方
        {
            k=mulit(b,b,x);   //求b的平方
            if(k==1&&b!=1&&b!=x-1)     //用二次探测判断
              return false;
            b=k;
        }
        if(b!=1)  return false;   //用费马小定律判断
    }
    return true;   //如果进行多次测试都是对的，那么x就很有可能是素数
}
int main(){
  long long n,a;
  while(~scanf("%lld", &n)){
    long long ans=0;
    while(n--){
      scanf("%lld", &a);
      if(Miller_Rabin(a)) ans++;
    }
    printf("%lld\n", ans);
  }
  return 0;
}
```


Pollard_rho算法

Pollard_rho算法是一个随机算法，Pollard_rho算法对于一个整数n，首先使用Miller_Rabin算法判断是否是素数，若n是素数，则记录一个素因子：
如果n不是素数，则按照下述方法分解n的一个因子d:
先取一个随机整数$c, 1 \leq c < n$然后另取一个随机数x_1，$1 \leq x_1 < n$
然后计算序列 $x_1, x_2, x_3, x_4 \ldots x_i, x_{i+1} \ldots$
其中，令$y = x_{i-1} x_i = (x_{i-1} * (x_{i-1} + c)\% n$得出：
每生成一项x_i后求GCD（$y - x_i$，n），继续对p和$\frac{n}{p}$递归搜索，直到搜到素数为止：

若$GCD(y-x,n)$是1，则重复上述操作。
这样的过程一直进行至出现了以前出现过的某个x为止；

由于这个算法因为在找寻随机数的过程中会出现成环的情况，类似希腊字母□的形状（如图所示），因而得名Pollard_rho算法。



例5

https://cn.vjudge.net/problem/POJ-1811

AC code

```
#include <algorithm>
#include <cstdio>
#include <cstring>
#define maxn 10000
using namespace std;
typedef long long ll;
ll prime[10]={2,3,5,7,11,13,17,19,23,29};
ll factor[maxn];
ll tot;
ll mulit(ll a,ll b,ll c){//龟速乘
    ll ans=0;
    ll res=a;
    while(b){
      if(b&1)
        ans=(ans+res)%c;
      res=(res+res)%c;
      b>>=1;
    }
    return ans;
}
ll pow(ll x,ll n,ll mod){//快速幂
    ll res=1;
    while(n>0){
       if(n%2==1){
         res=mulit(res,x,mod);
       }
       x=mulit(x,x,mod);
       n>>=1;
    }
    return res;
}
bool Miller_Rabin(ll x)     //判断素数
{
    ll i,j,k;
    ll s=0,t=x-1;
    if(x==2)  return true;   //2是素数
    if(x<2||!(x&1))  return false;     //如果x是偶数或者是0,1，那它不是素数
    while(!(t&1)){  //将x分解成(2^s)*t的样子
        s++;
```

```
        t>>=1;
    }
    for(i=0;i<10&&prime[i]<x;++i){      //随便选一个素数进行测试
        ll a=prime[i];
        ll b=pow(a,t,x);      //先算出a^t
        for(j=1;j<=s;++j){    //然后进行s次平方
            k=mulit(b,b,x);   //求b的平方
            if(k==1&&b!=1&&b!=x-1)      //用二次探测判断
              return false;
            b=k;
        }
        if(b!=1)  return false;   //用费马小定律判断
    }
    return true;   //如果进行多次测试都是对的，那么x就很有可能是素数
}
ll gcd(ll a,ll b){
    if(a==0)return 1;
    if(a<0) return gcd(-a,b);
    while(b){
        long long t=a%b;
        a=b;
        b=t;
    }
    return a;
}
ll Pollard_rho(ll x,ll c){
    ll i=1,k=2;
    ll x0=rand()%x;
    ll y=x0;
    while(1){
        i++;
        x0=(mulit(x0,x0,x)+c)%x;
        long long d=gcd(y-x0,x);
        if(d!=1&&d!=x) return d;
        if(y==x0) return x;
        if(i==k){
            y=x0;
            k+=k;
        }
    }
}
void findfac(ll n){
    if(Miller_Rabin(n)){
        factor[tot++]=n;
        return;
    }
    ll p=n;
    while(p>=n) p=Pollard_rho(p,rand()%(n-1)+1);
    findfac(p);
    findfac(n/p);
}
int main(){
  ll t,n;
  scanf("%lld", &t);
  while(t--){
    scanf("%lld", &n);
    if(Miller_Rabin(n)){
      printf("Prime\n");
      continue;
    }
    tot=0;
    findfac(n);
    ll ans=factor[0];
    for(int i=1; i<tot; i++)
        if(factor[i]<ans)
            ans=factor[i];
    printf("%lld\n",ans);
  }

  return 0;

```

```
}
```

有 0 个人打赏

文章最后发布于: 20