链滴

Pollard'"'"'s rho 算法

作者：DoctorLo

原文链接：https://ld246.com/article/1569220928268

来源网站：链滴

Pollard's rho算法

摘要

Pollar's Rho算法是一种用于分解质因数的算法，对于一个被分解的数N，假设N的最小质因数为p（p neq N），那么Pollar's Rho算法能够在O(\sqrt N * α(N))的期望时间复杂度内将N分解为两个不是1数的乘积，其中α(N)是求解两个数的gcd时间复杂度，并且该算法对于空间要求很低。

首先要明确的是，Pollar's Rho 算法是随机算法，其基于Miller-Rabin算法，一般在求大整数的唯一解时会用到，因为 试除法 ^{[1]}的O(\sqrt N)复杂度太高。

本文将首先从算法的适用模型讲起，详细介绍算法流程，最后会给出一道例题以及详细注释了的代码板，当然还有模板的使用说明。

[1]试除法：一种将整数唯一分解成质因数乘积的方法，时间复杂度为O(\sqrt N)

算法简介

该算法最早于1975年由John M. Pollard提出，而Richard Brent于1980年提出了改进版本。虽然不目前最快的算法，但它要比试除法快上多个量级。实现它的思想同样可以用于其他地方。

问题模型： 给定一个整数n，试着将n分解为若干素数相乘的形式。n小于2^{60}。

分析： 如果用试除法，显然时间复杂度过高，现在我们希望得到一个快速的方法；类比大素数判断的iller-Rabin算法，Miller-Rabin算法依靠着费马小定理以及二次检测定理实现了快速判断一个大整数否为素数的算法，实际上对于大整数的唯一分解也同样有着类似的巧妙的算法，就是本文介绍的Pollar's rho算法。

算法思想

基本思路：

对于给定的一个整数n，显然如果n为素数（Miller-Rabin算法判断），那么算法结束，返回唯一素因n。

否则，pollard's rho算法会试着找到当然数n的一个因子a（a并不一定是素数），然后递归 Pollard_ro(a) 和 Pollard_rho(n/a)，即将n分为了两个部分，将原问题变成了规模更小的两个子问题。

如何求因子a?

一般情况下可能会本能的想到枚举，但这样过于耗时。寻找因子是该算法中最重要的一步，该算法中用随机化算法来查找因子a。假设此时n仅有2个因子 p 和 q，那么如果我们用随机数来找n的因子，功率为1/n。接下来我们思路是如何提升成功率。

寻找一个因子a，等价于寻找是否存在k个数，使得其中有x_i - x_j = a，由于**生日悖论** 我们可以得知当 = \sqrt N时，该概率是50%，所以我们将可能性从\frac{1}{n}提升到了\sqrt{\frac{1}{n}}。

但不幸的是对于10位的整数，k = 1e5时，仍要做k^2 = 10^{10}次比较，幸运的是，还有更好的方。

我们仍然选取k个数：x_1,x_2,...,x_k，但我们不再询问是否存在x_i - x_j可以整除n，转而询问是否存在cd(x_i - x_j , n) > 1 的情况。换句话说，我们问x_i - x_j和n是否存在一个平凡的最大公约数。

如果我们询问有多少个整数能整除n，那么答案显然只有两个：p和q。

但是如果我们问有多少个数使得gcd(x_i-x_j) > 1，答案便很多了，如：p, 2p, 3p, 4p, ... , (q-1)p , q , q ,...,(p-1)q。准确的说，有p+q-2个。

所以，一个简单的策略如下：

- 在区间[2 , n-2]中随机选 k 个数， x_1,x_2,x_3,...,x_k。
- 判断是否存在gcd(x_i - x_y,n) > 1，若存在，gcd(x_i - x_y,n) 是n的一个因子（p或q）。

但是这样还有一个问题，就是我们大约要选取n^{1/4}个数，数量还是太大，以至于不能存放在内存。

Pollard's rho算法的解决策略：

为了解决无法储存太多数的问题，Pollard's rho Algorithm只将两个数存放在内存中。具体思路是：们并不随机生成k个数并两两比较，而是一个一个地生成并检查连续的两个数。反复执行这个步骤并望能够得到我们想要的数。

我们使用一个函数来生成伪随机数。

换句话说，我们不断使用一个函数f来生成（可以这样形容）随机数。当然并不是所有的函数都能这样，但是有一个函数可以：

      f(x) = (x^2 + a) mod N

（其中的 a 可以用随机数生成，当然这不是讨论的重点。）

我们从x_1 = 2开始，让x_2 = f(x_1) , x_3 = f(x_2),.....，通项为：x_{n+1} = f(x_n)

于是顺着这个策略，依据上述分析，不断判断gcd(x_n - x_{n-1}, n)是否大于1即可。

存在的问题：

大多情况下，这种算法是可以正常运行的，但是对于某些数据，会出现无线的死循环，原因在于函数f在自环。

于是问题变成了“如何判断环的出现”。一种方法是记录所有出现过的数，当然这会耗费大量内存，舍弃；另一种方法是Floyd发明的算法，这里可以举一个有趣的例子说明“假设A和B在一个很长的圆轨道上走，那么我们如何判断B是否走完一圈呢？我们可以让B的速度是A的二倍，他们同时出发，当第一次追上A，就知道B至少已经走了一圈”，同样的道理运用到该算法中，框架如下：

```
a = 2;
b = 2;
while(b != a){
    a = f(a);    //一倍速
    b = f(f(b));//二倍速
    p = GCD(|b-a| , n);
    if(p > 1) return "Found factor: p";
}
return "Failed";
```

算法流程

简单梳理一下，给出主要框架而不涉及理论证明。

假设 n 为待分解的数，我们将分解出来的所有因子存放在factors[110]数组中，当然存放的是无序的因子。

findFac(long long n) 函数：

负责将 n 分解成素因子相乘的形式，并将结果存放在factors数组中。

- 如果n本身就是素数，那么将n存放在factor便可结束并返回。
- 如果n不是素数，那么通过 pollard_rho()函数 找到n的一个因子p(不一定是素因子)，递归findFac(p和findFac(n/p)

pollard_rho(long long x, long long a)函数：

返回x的一个因子（不一定是素数），若失败则返回x。

利用Floyd发明的类似“二倍速的算法”，具体参考上述伪代码。

注： 当然还有一些辅助函数，例如快速幂、Miller-Rabin算法、欧几里得算法等。

算法模板

```
/*-----------------------------------------
 POJ 1811
 最后更新：2019/8/11
 说明：该代码包含了取随机值函数rand()以及求最小值函数min()
 注：适用范围是2^61,至少要保证中间结果不会溢出long long
 (中间结果最多是2*n)
 -----------------------------------------*/
#include<cstdio>
#include<cstdlib>
#include<iostream>
typedef long long ll;
/*-----------------------------------------
 利用 Miller-Rabin进行素性测试
 -----------------------------------------*/
int testnum[] = {2,7,61,3,5,11,13,19};
ll fmul(ll a,ll b,ll p){
    /*返回a * b % p*/
    a %= p,b %= p;          //防止超出精度
    ll res = 0;
    while(b){
        if(b&1) res += a, res %= p;
        a <<= 1;            //a = a*2
        if(a >= p) a %= p;
        b >>= 1;            //b = b/2
    }
    return res;
}

ll qpow(ll a,ll b,ll p){
    /*返回a^b % p*/
    ll res = 1;
    while(b){
        if(b&1) res = fmul(res,a,p);
        a = fmul(a,a,p);
        b >>= 1;
    }
    return res;
```

```
}

bool isPrime(ll n){
    /*Miller-Rabin判定x是否为素数*/
    if(n == 2) return true;
    if(n < 2 || n%2 == 0) return false;
    ll d = n-1, a, x, y;int t = 0;
    while((d&1) == 0) d >>= 1,t++;
    for(int i = 0;i < 7;i++){
        a = testnum[i];
        if(n == a) return true;
        x = qpow(a,d,n);
        for(int j = 0;j < t;j++){
            y = fmul(x,x,n);
            if(y == 1 && x != 1 && x != n-1) return false;
            x = y;
        }
        if(x != 1) return false;
    }
    return true;
}
/*-----------------------------------------------
 利用 pollard rho 算法进行质因数分解
 ----------------------------------------------*/

ll factors[110];    //用来存放被分解的因数(无序)
int tot = 0;        //因子个数
ll gcd(ll a,ll b){
    /* 返回a和b的最大公约数 */
    if(a == 0) return 1;
    if(a < 0) return gcd(-a,b);
    while(b){
        ll t = a%b;
        a = b; b = t;
    }
    return a;
}
ll pollard_rho(ll x,ll c){
    /* 返回 x 的一个因子或 x 本身 */
    ll i = 1,k = 2;
    ll tx = rand()%x;
    ll y = tx;
    while(true){
        i++;
        tx = (fmul(tx,tx,x)+c)%x;
        ll d = gcd(y-tx,x);
        if(d != 1 && d != x) return d;
        if(y == tx) return x;       //寻找失败
        if(i == k) y = tx, k += k;
    }
}
void findFac(ll n){
    /* 对 n 进行质因数分解 */
    if(isPrime(n)){
```

```
        factors[++tot] = n;
        return ;
    }
    ll p = n;
    /* 通过pollard_rho算法找到 n 的一个因子 p */
    while(p >= n) p = pollard_rho(p,rand()%(n-1)+1);
    findFac(p);     //递归分解
    findFac(n/p);
}
int main(){
    int t;ll n,ans;
    scanf("%d",&t);
    while(t--){
        scanf("%lld",&n);
        if(isPrime(n)) puts("Prime");
        else{
            tot = 0; ans = 1e18; findFac(n);
            for(int i = 1;i <= tot;i++) ans = std::min(ans,factors[i]);
            printf("%lld\n",ans);
        }
    }
    return 0;
}
```