Pollard-Rho大数分解

组合随机采样

可以通过改变问题为满足答案的**组合**使得答案的概率大大提高。

例如“生日悖论”：如果单纯地在某个班级里找到1月1日出生的人，那概率不高，但是如果求班级里是否有两个生日相同的人，那概率就大大提高了。

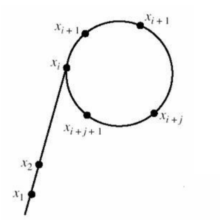
应用在在大数分解中，随机找出 N 的一个质因数的概率极低，我们考虑修改问题：找到 $\gcd(k,N)>1$ 的一个 k，这样 $g=\gcd(k,N)$ 就是 N 的一个因子（不一定是质因子），概率就提高了许多。想要得到质因子，只需递归对 g 和 N/g 分解即可。

我们不妨选取一组数 $x_1,x_2,\cdots,x_n$，若 $\gcd(|x_i-x_j|,N)>1$，就找到了这样的因子 $\gcd(|x_i-x_j|,N)$. 可以证明，需要选取的数的个数为 $O(N^{1/4})$，但是这组数如何选择仍然是一个问题。

Pollard-Rho

不妨选取一个伪随机数序列。$\textbf{Pollard}$ 设计了这样一个序列：$x_n^2=(x_{n-1}^2+c) \bmod N$，其中 c 是一个随机的常数，x_1 是 $[1,N-1]$ 内的随机数。每次检验 $|x_i-y|$ 与 N 的 $\gcd$ 是否 $\neq 1$，如果是，那么返回差值（这里 y 是下标 $j<i$ 的某个 x_j，算法导论上取 2 的幂次）。

容易知道这个序列会出现循环节，如果画出来，形成希腊字母 ρ 的形状：



根据生日悖论的分析，在序列出现回路之前预计要执行的步数为 $\Theta(\sqrt{n})$.

如果我们走遍了环而仍没有得到分解，那么我们就需要更改 c 重新计算。当然，整个过程之前先 $\textbf{Miller-Rabin}$ 测试是否是素数。

Floyd判环法

设置快慢指针，慢指针走一步快指针走两步，如果走在一起了说明有环。

在 $\textbf{Pollard-Rho}$ 的过程中可用来判断是否已经绕了环一圈。

倍增优化

每次都求 $\gcd$ 太费时了，我们每个 2 的幂次求 $\gcd$。具体的，把下标位于 $[2^{k-1},2^k)$ 之中的 $|x-y|$ 乘起来模 N 与 N 求 $\gcd$，容易知道，但凡这些数里面有一个与 N 的 $\gcd$ 不为 1，那么乘积与 N 的 $\gcd$ 也不为 1. 乘积为 0 的时候说明绕了一圈，分解失败。

实际操作中，取 2^k 不超过 128 有较好的表现。

Code

ATT：

- 如果不能用 `__int128` 改成快速乘也可（牺牲了时间复杂度）。

```cpp
mt19937 rnd(time(NULL));
namespace Miller_Rabin{
    inline LL fpow(LL bs, LL idx, LL mod){
        bs %= mod;
        LL res = 1;
        while(idx){
            if(idx & 1) res = (__int128)res * bs % mod;
            bs = (__int128)bs * bs % mod;
            idx >>= 1;
        }
        return res;
    }
    bool test(LL n){
        if(n < 3)   return n == 2;
        if(!(n & 1))    return false;
        LL u = n - 1, t = 0;
        while(u % 2 == 0)   u /= 2, t++;
        int testTime = 10;
        while(testTime--){
            LL v = rnd() % (n - 2) + 2;
            v = fpow(v, u, n);
            if(v == 1 || v == n - 1)    continue;
            int j; for(j = 0; j < t; j++, v = (__int128)v * v % n)
                if(v == n - 1)  break;
            if(j >= t)  return false;
        }
        return true;
    }
}

namespace Pollard_Rho{
    vector<LL> factors;
    // LL mxfactor = 0;
    inline LL solve(LL n){
        LL c = rnd() % (n - 1) + 1;
        LL x = 0, y = 0, val = 1;
        for(LL k = 1; ; k <<= 1, y = x, val = 1){
            for(int i = 1; i <= k; i++){
                x = ((__int128)x * x + c) % n;
                val = (__int128)val * abs(x - y) % n;
                if(val == 0)    return n;
                if(i % 127 == 0){
                    LL g = gcd(val, n);
                    if(g > 1)   return g;
                }
            }
            LL g = gcd(val, n);
            if(g > 1)   return g;
        }
    }
    void factorize(LL n){
        if(n < 2)   return;
        // if(n <= mxfactor)    return;
        if(Miller_Rabin::test(n)){
            factors.emplace_back(n);
            // mxfactor = max(mxfactor, n);
            return;
        }
        LL p = n;
        while(p == n)   p = solve(n);
        while(n % p == 0)   n /= p;
        factorize(p), factorize(n);
    }
}
```