

algorithm template

zinan.xu.dev@gmail.com

December 29, 2025

Contents

Contents	2
数论	3
素性检测	3
大合数分解	4
二次剩余-膜意义下开方	5
Nim积	7
树	9
树的直径	9
最近公共祖先	10
树同构	12

数论

素性检测

```
1 #include<vector>
2 namespace PrimeTest {
3     long long mul(long long a, long long b, long long mod){
4         return (__int128) a * b % mod;
5     }
6
7     long long Pow(long long a, long long b, long long mod){
8         //mod <= 10^18.
9         long long res = 1;
10        while(b){
11            if (b&1) res = mul(res, a, mod);
12            b >>= 1;
13            a = mul(a, a, mod);
14        }
15        return res;
16    }
17
18    std::vector<long long> pr = {2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31,
19      ↵ 37};
20
21    bool rabin_test(long long a, long long n, long long s, long long d){
22        long long u = Pow(a, d, n);
23        if (u == 1 or u == n - 1) return false;
24
25        for(long long i = 1; i < s; i++){
26            u = mul(u, u, n);
27            if (u == n - 1) return false;
28        }
29        return true;
30    }
31
32    bool rabin_miller(long long n){
33        if (n < 2) return false;
34        if (n % 2 == 0) return n==2;
35        long long res = 1;
36        long s = 0, d = n-1;
37        while(d%2==0) {
38            s++;
39            d>>=1;
40        }
```

```

41         for(long long i = 0;i<pr.size();i++){
42             if (n%pr[i] == 0) {
43                 return n == pr[i];
44             }
45             if (rabin_test(pr[i], n, s, d)){
46                 return false;
47             }
48         }
49         return true;
50     }
51 }
```

大合数分解

该板子依赖了【数论】 - 【素性检测】的板子

```

1 #include<vector>
2 #include<algorithm>
3 namespace Factorize {
4     /*
5      * 可以分解不超过 LONG_LONG_MAX 的合数的所有因子，期望复杂度为
6      *  $O(n^{1/4})$ 
7      */
8     using namespace PrimeTest;
9
10    /*
11     * 牛逼plus的gcd
12     * 由于不用做除法和取模，只用了二进制命令和减法，所以速度非常快
13     * 这里不是朴素的更相减损术，而是用了一些优雅的性质
14     */
15    long long gcd(long long _a, long long _b) {
16        unsigned long long a = abs(_a), b = abs(_b);
17        if (a == 0) return b;
18        if (b == 0) return a;
19        int shift = __builtin_ctzll(a|b); // 拿到了最多有多少个2
20        a >>= __builtin_ctzll(a); // 这里保证了a是奇数
21        do {
22            b >>= __builtin_ctzll(b); // 这里保证了b是奇数
23            if (a > b) std::swap(a, b);
24            b -= a; // 这里两个奇数相减，一定是偶数，在下一个循环中至少二进
25            // 制会少一位
26        } while (b);
27        return (a << shift);
28    }
29 }
```

```

28  /*
29   * 返回n的任意一个素因子
30   * 用一个优雅的近似随机数来判断是否是素数
31   * 如果随机过程均匀，期望复杂度是  $O(n^{1/4} * \log(n))$ 
32   */
33   long long pollard_single(long long n) {
34     if (rabin_miller(n)) return n;
35     if (n%2==0) return 2;
36     long long st = 0;
37     auto f = [&](long long x) { return (__int128(x) * x + st) % n; };
38     while (true) {
39       st++;
40       long long x = st, y = f(x);
41       while (true) {
42         long long p = gcd((y - x + n), n);
43         if (p == 0 or p == n) break;
44         if (p != 1) return p;
45         x = f(x);
46         y = f(f(y));
47       }
48     }
49   }
50
51  /*
52   * 返回n的所有素因子
53   * n的因子越多，该算法越快
54   * 近似复杂度可以用  $O(n^{1/4})$  估计
55   */
56   std::vector<long long> pollard(long long n) {
57     if (n==1) return {};
58     auto x = pollard_single(n);
59     if (x==n) return {x};
60     auto l = pollard(x);
61     auto r = pollard(n/x);
62     l.insert(l.end(), r.begin(), r.end());
63     return l;
64   }
65 }
```

二次剩余-膜意义下开方

```

1 #include <vector>
2 namespace SqrtMod {
3   long long mod, i2;
4   struct Complex {
```

```

5     long long a, b;
6     Complex(long long _a, long long _b) : a(_a), b(_b) {}
7     Complex operator*(const Complex& other) {
8         return Complex((a*other.a + i2*b%mod*other.b%mod)%mod,
9                         → (a*other.b + b*other.a)%mod);
10    }
11    Complex operator%(long long _mod) {
12        return *this;
13    }
14    };
15 // long long的快速幂和Complex的放一起了
16 template<typename T>
17 void qpow(T a, long long b, long long mod, T& ans) {
18     while (b) {
19         if (b & 1) ans = ans * a % mod;
20         a = a * a % mod;
21         b >>= 1;
22     }
23 }
24 /*
25 * 膜意义下的开方 (二次剩余) : 找到 x 使得  $x^2 \equiv a \pmod{p}$ 
26 * 复杂度:  $O(\log p)$ 
27 * 限制:
28 * -  $a < 2^{31}$ 
29 * -  $p < 2^{31}$ 
30 * -  $p$  是素数
31 * 返回:  $p$ 范围内的所有满足条件的x
32 */
33 std::vector<int> sqrt_mod(int a, int p) {
34     if(a<=1 || p == 2) return {a};
35     // 判断a是否是二次剩余
36     long long remind = 1;
37     if(qpow((long long)a, (p-1)/2, p, remind); remind != 1) {
38         return {-1};
39     }
40 }
41
42     long long b, d; // d = b*b-a 是二次非剩余
43     while(true) {
44         b = rand() % p;
45         d = (b*b - a) % p;
46         if(d < 0) d += p;
47         remind = 1;
48         qpow(d, (p-1)/2, p, remind);
49         if(remind != 1) break;

```

```

50         }
51         mod = p;
52         i2 = d;
53
54         // 计算  $(a+i)^{(p+1)/2}$ 
55         Complex data_(b, 1);
56         Complex power(1, 0);
57         qpow(data_, (p+1)/2, p, power);
58
59         int ans = power.a % p;
60         return (ans > p-ans ? std::vector<int>{p-ans, ans} :
61             std::vector<int>({ans, p-ans}));
62     }
63 }
```

Nim积

```

1 #include <cstring>
2 namespace NimProd {
3     using ull = unsigned long long;
4     ull cache[257][257];
5     /*
6      * 计算nim积
7      * 限制:
8      * - x,y 在unsigned long long 范围内
9      * 复杂度:  $O(\log^2(\max(x, y)))$ 
10     */
11     ull solve(ull x, ull y, int p = 32) {
12         if (x <= 1 || y <= 1) return x * y;
13         if (p < 8 && cache[x][y]) return cache[x][y]; // 小缓存优化
14
15         ull a = x >> p;
16         ull b = ((1ull << p) - 1) & x;
17         ull c = y >> p;
18         ull d = ((1ull << p) - 1) & y;
19
20         ull bd = solve(b, d, p >> 1);
21         ull ac = solve(solve(a, c, p >> 1), 1ull << p >> 1, p >> 1);
22
23         ull ans = ((solve(a ^ b, c ^ d, p >> 1) ^ bd) << p) ^ ac ^ bd;
24         if (p < 8) cache[x][y] = cache[y][x] = ans;
25         return ans;
26     }
27     void init() {
```

```
28     memset(cache, 0, sizeof(cache));  
29     for(int i = 0;i<256;i++) {  
30         for(int j = 0;j<256;j++) {  
31             solve(i, j);  
32         }  
33     }  
34 }  
35 }  
36 }
```

树

树的直径

```
1 #include <vector>
2 #include <tuple>
3 namespace TreeDiameter {
4     /*
5      * 无向正权树的最大直径，限制：
6      * 1. 直径需要 $\leq LONG\_LONG\_MAX$ 
7      * 2. 单颗树，而非森林
8      */
9     using namespace std;
10    using Graph = vector<vector<pair<int, long long>>>; // 起点对应的
11    // 边(终点 & 权值)
12    /*
13     * Input: 树，起始点
14     * Output: 离起始点最大的距离，对应的点
15     * 复杂度:  $O(\text{边数})$ 
16     */
17    pair<long long, int> dfs(const vector<vector<pair<int, long long>>>
18    // &g, int cur, int par = -1) {
19        pair<long long, int> ret(0, cur);
20        for (auto e : g[cur]) {
21            if (e.first == par) continue;
22            auto cost = dfs(g, e.first, cur);
23            cost.first += e.second;
24            ret = max(ret, cost);
25        }
26        return ret;
27    /*
28     * Input: 树
29     * Output: 直径起点，直径终点，直径长度
30     */
31    tuple<int, int, long long> tree_diameter(const
32        vector<vector<pair<int, long long>>> &g) {
33        auto u = dfs(g, 0, -1).second;
34        long long dist;
35        int v;
36        tie(dist, v) = dfs(g, u, -1);
37        return make_tuple(u, v, dist);
38    }
```

```

39  /*
40   * 会搜索出一条从cur到goal的路径，结果会放在path里面
41   * Input: 树
42   * Output: 路径
43   * 复杂度:  $O(\text{边数})$ 
44   */
45   void path_restoration(const vector<vector<pair<int, long long>>> &g,
46     vector<int> &path, int cur, int par, int &goal) {
47     path.push_back(cur);
48     if (cur == goal) {
49       goal = -1;
50       return;
51     }
52     for (auto e : g[cur]) {
53       int nxt = e.first;
54       if (nxt == par) continue;
55       path_restoration(g, path, nxt, cur, goal);
56       if (goal == -1) return;
57     }
58     if (goal == -1) {
59       return;
60     }
61     path.pop_back();
62   }
63 }
64 }
```

最近公共祖先

```

1 #include <vector>
2
3 namespace LCA {
4 #define V vector
5   /*
6    * 最近公共祖先，限制：
7    * 1. root必须为0 / 1
8    * 2. 总复杂度  $O(q * \log(n) + n * \log(n))$  // q次查询，一共n个节点，建树过
9    * 程  $n * \log(n)$ ，每次查询  $\log(n)$ 
10   */
11   using namespace std;
12   class Tree {
13     private:
14       int n_;
```

```

14     int root_;
15     int lg;
16     V<int> depth;
17     V<V<int>> father;
18     V<V<int>> son;
19     /*
20      * 从跟节点dfs, 来构建depth数组
21      * 复杂度: O(n)
22      */
23     void dfs(int now, int pre = -1, int dep = 1) {
24         depth[now] = dep;
25         father[0][now] = pre;
26         for(auto s : son[now]) {
27             if(s==pre) continue;
28             dfs(s, now, dep+1);
29         }
30     }
31     /*
32      * 构建祖先关系, 倍增构建
33      * 复杂度: O(n*log(n))
34      */
35     void build_father() {
36         for(int i = 1; i < lg; i++) {
37             for(int j = root_; j < n_ + (root_==1); j++) {
38                 father[i][j] = (father[i-1][j] == -1) ? -1 :
39                               father[i-1][father[i-1][j]];
40             }
41         }
42     }
43     public:
44     Tree(int root, int n) {
45         root_ = root;
46         n_ = n;
47         lg = 1;
48         while((1<<lg) < n) lg++;
49         depth.resize(n+(root==1));
50         father = V<V<int>>(lg, V<int>(n+(root==1), -1));
51         son = V<V<int>>(n+(root==1), V<int>(0));
52     }
53     /*
54      * 增加一个父子关系, now的父亲是pre
55      * 复杂度: O(1)
56      */
57     void add(int pre, int now) {
58         son[pre].push_back(now);

```

```

59             son[now].push_back(pre);
60         }
61
62         /*
63          * 完整建树，需要在add完所有父子关系才可以调用
64          * 复杂度:  $O(n \log(n))$ 
65          */
66         void build() {
67             this->dfs(root_);
68             this->build_father();
69         }
70
71         /*
72          * 查询u和v的最近公共祖先
73          * 复杂度:  $O(\log(n))$ 
74          */
75         int query_lca(int u, int v) {
76             if(depth[u] > depth[v]) {
77                 swap(u, v);
78             }
79             int depth_diff = depth[v] - depth[u];
80             for(int i = 0; i < lg; i++) {
81                 if(depth_diff & (1<<i)) {
82                     v = father[i][v];
83                 }
84             }
85             if(u == v) {
86                 return u;
87             }
88             for(int i = lg - 1; i >= 0; i--) {
89                 if(father[i][u] != father[i][v]) {
90                     u = father[i][u];
91                     v = father[i][v];
92                 }
93             }
94             return father[0][u];
95         }
96     };
97 }

```

树同构

```

1 #include <vector>
2 #include <map>
3 #include <algorithm>

```

```

4  namespace TreeIsomorphism {
5      /*
6      * 判断有根树是否同构, 限制:
7      * 1. root只能为0或者1
8      * 2. 复杂度  $O(n \log(n))$ 
9      */
10     using namespace std;
11     class Tree {
12     private:
13         int n_, root_;
14         vector<vector<int>> son;
15         map<vector<int>, int> mp;
16         vector<int> t;
17         /*
18         * 建树过程
19         * 每次会从root开始遍历, 记录每个子节点的编号, 并从map中判断是否已
20         * 经存在同构结构
21         * 为了保证同构, 所以需要一个sort来保证子节点编号不影响结果
22         */
23         void dfs(int root, int pre) {
24             vector<int> v(1, 0);
25             for(auto i : son[root]) {
26                 if(i==pre) continue;
27                 dfs(i, root);
28                 v.push_back(t[i]);
29             }
30             sort(v.begin(), v.end());
31             if(mp.find(v) == mp.end()) {
32                 mp[v] = mp.size();
33             }
34             t[root] = mp[v];
35         }
36     public:
37         Tree(int n, int root) {
38             n_ = n;
39             root_ = root;
40             t = vector<int>(n+(root==1), 0);
41             son.resize(n+(root==1));
42         }
43         /*
44         * 添加边
45         * 这里只要是无向边就行了, 不需要保证u是v的父亲
46         */
47         void add_edge(int u, int v) {

```

```

49             son[u].push_back(v);
50             son[v].push_back(u);
51         }
52
53     /*
54      * 构建树
55      * 这里会计算出每个节点的结构，并给一个编号，保证编号相同的节点对应
56      * 的子树是同构的
57      */
58     void build() {
59         dfs(root_, -1);
60     }
61
62     /*
63      * 返回有多少个不同构的子树
64      */
65     int get_size() {
66         return mp.size();
67     }
68
69     /*
70      * 返回每个子树的编号
71      * Input: 每个节点的编号
72      * Output: 每个子树的结构编号
73      * 仅保证相同结构的子树返回的编号一致，不保证节点更多的子树编号更大
74      */
75     int get_mask(int i) {
76         return t[i];
77     }
78 }

```