algorithm template

zinan.xu.dev@gmail.com

January 5, 2026

Contents

数论

素性检测

```
#include<vector>
namespace PrimeTest {
    long long mul(long long a, long long b, long long mod){
        return (__int128) a * b % mod;
    }

    long long Pow(long long a, long long b, long long mod){
        //mod <= 10^18.
        long long res = 1;
        while(b){
            if (b&1) res = mul(res, a, mod);
            b >>= 1;
            a = mul(a, a, mod);
        }
        return res;
    }

    std::vector<long long> pr = {2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31,
    ↪  37};

    bool rabin_test(long long a, long long n, long long s, long long d){
        long long u = Pow(a, d, n);
        if (u == 1 or u == n - 1) return false;

        for(long long i = 1; i < s; i++){
            u = mul(u, u, n);
            if (u == n - 1) return false;
        }
        return true;
    }

    bool rabin_miller(long long n){
        if (n < 2) return false;
        if (n % 2 == 0) return n==2;
        long long res = 1;
        long  s = 0, d = n-1;
        while(d%2==0) {
            s++;
            d>>=1;
        }

```

```
        for(long long i = 0;i<pr.size();i++){
            if (n%pr[i] == 0) {
                return n == pr[i];
            }
            if (rabin_test(pr[i], n, s, d)){
                return false;
            }
        }
        return true;
    }
}
```

大合数分解

该板子依赖了【数论】-【素性检测】的板子

```
#include <vector>
#include <algorithm>
namespace Factorize {
    /*
     * 可以分解不超过 LONG_LONG_MAX 的合数的所有因子，期望复杂度为
     ↪  O(n^{1/4})
     */
    using namespace PrimeTest;

    /*
     * 牛逼plus的gcd
     * 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
     * 这里不是朴素的更相减损术，而是用了一些优雅的性质
     */
    long long gcd(long long _a, long long _b) {
        unsigned long long a = abs(_a), b = abs(_b);
        if (a == 0) return b;
        if (b == 0) return a;
        int shift = __builtin_ctzll(a|b); // 拿到了最多有多少个2
        a >>= __builtin_ctzll(a); // 这里保证了a是奇数
        do {
            b >>= __builtin_ctzll(b); // 这里保证了b是奇数
            if (a > b) std::swap(a, b);
            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
            ↪  制会少一位
        } while (b);
        return (a << shift);
    }

```

```
    /*
     * 返回n的任意一个素因子
     * 用一个优雅的近似随机数来判断是否是素数
     * 如果随机过程均匀，期望复杂度是 O(n^{1/4} * log(n))
     */
    long long pollard_single(long long n) {
        if (rabin_miller(n)) return n;
        if (n%2==0) return 2;
        long long st = 0;
        auto f = [&](long long x) { return (__int128(x) * x + st) % n; };
        while (true) {
            st++;
            long long x = st, y = f(x);
            while (true) {
                long long p = gcd((y - x + n), n);
                if (p == 0 or p == n) break;
                if (p != 1) return p;
                x = f(x);
                y = f(f(y));
            }
        }
    }

    /*
     * 返回n的所有素因子
     * n的因子越多，该算法越快
     * 近似复杂度可以用 O(n^{1/4}) 估计
     */
    std::vector<long long> pollard(long long n) {
        if(n==1) return {};
        auto x = pollard_single(n);
        if (x==n) return {x};
        auto l = pollard(x);
        auto r = pollard(n/x);
        l.insert(l.end(), r.begin(), r.end());
        return l;
    }
}
```

二次剩余-膜意义下开方

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

```cpp
    long long a, b;
    Complex(long long _a, long long _b) : a(_a), b(_b) {}
    Complex operator*(const Complex& other) {
        return Complex((a*other.a + i2*b%mod*other.b%mod)%mod, (a*other.b + b*other.a)%mod);
    }
    Complex operator%(long long _mod) {
        return *this;
    }
};

// long long的快速幂和Complex的放一起了
template<typename T>
void qpow(T a, long long b, long long mod, T& ans) {
    while (b) {
        if (b & 1) ans = ans * a % mod;
        a = a * a % mod;
        b >>= 1;
    }
}

/*
 * 膜意义下的开方（二次剩余）: 找到 x 使得 x^2 \equiv a (mod p)
 * 复杂度: O(log p)
 * 限制:
 *   - a < 2^31
 *   - p < 2^31
 *   - p 是 素数
 * 返回: p范围内的所有满足条件的x
 */
std::vector<int> sqrt_mod(int a, int p) {
    if(a<=1 || p == 2) return {a};
    // 判断a是否是二次剩余
    long long remind = 1;
    if(qpow((long long)a, (p-1)/2, p, remind); remind != 1) {
        return {-1};
    }

    long long b, d; // d = b*b-a 是二次非剩余
    while(true) {
        b = rand() % p;
        d = (b*b - a) % p;
        if(d < 0) d += p;
        remind = 1;
        qpow(d, (p-1)/2, p, remind);
        if(remind != 1) break;
```

```cpp
        }
        mod = p;
        i2 = d;

        // 计算(a+i)^{(p+1)/2}
        Complex data_(b, 1);
        Complex power(1, 0);
        qpow(data_, (p+1)/2, p, power);

        int ans = power.a % p;
        return (ans > p-ans ? std::vector<int>{p-ans, ans} : std::vector<int>({ans, p-ans}));
    }
}

```

Nim积

```cpp
#include <cstring>
namespace NimProd {
    using ull = unsigned long long;
    ull cache[257][257];
    /*
     * 计算nim积
     * 限制:
     *       - x,y 在unsigned long long 范围内
     * 复杂度: O(log^2(max(x, y)))
     */
    ull solve(ull x, ull y, int p = 32) {
        if (x <= 1 || y <= 1) return x * y;
        if (p < 8 && cache[x][y]) return cache[x][y]; // 小缓存优化

        ull a = x >> p;
        ull b = ((1ull << p) - 1) & x;
        ull c = y >> p;
        ull d = ((1ull << p) - 1) & y;

        ull bd = solve(b, d, p >> 1);
        ull ac = solve(solve(a, c, p >> 1), 1ull << p >> 1, p >> 1);

        ull ans = ((solve(a ^ b, c ^ d, p >> 1) ^ bd) << p) ^ ac ^ bd;
        if (p < 8) cache[x][y] = cache[y][x] = ans;
        return ans;
    }
    void init() {
```

```
        memset(cache, 0, sizeof(cache));
        for(int i = 0;i<256;i++) {
            for(int j = 0;j<256;j++) {
                solve(i, j);
            }
        }
    }
}

```

树

树的直径

```cpp
#include <vector>
#include <tuple>
namespace TreeDiameter {
    /*
     * 无向正权树的最大直径，限制：
     * 1. 直径需要<= LONG_LONG_MAX
     * 2. 单颗树，而非森林
     */
    using namespace std;
    using Graph = vector<vector<pair<int, long long>>>; // 起点对应的边(终点 & 权值)

    /*
     * Input: 树，起始点
     * Output: 离起始点最大的距离，对应的点
     * 复杂度: O(边数)
     */
    pair<long long, int> dfs(const vector<vector<pair<int, long long>>> &g, int cur, int par = -1) {
        pair<long long, int> ret(0, cur);
        for (auto e : g[cur]) {
            if (e.first == par) continue;
            auto cost = dfs(g, e.first, cur);
            cost.first += e.second;
            ret = max(ret, cost);
        }
        return ret;
    }
    /*
     * Input: 树
     * Output: 直径起点，直径终点，直径长度
     */
    tuple<int, int, long long> tree_diameter(const vector<vector<pair<int, long long>>> &g) {
        auto u = dfs(g, 0, -1).second;
        long long dist;
        int v;
        tie(dist, v) = dfs(g, u, -1);
        return make_tuple(u, v, dist);
    }

```

```
    /*
     * 会搜索出一条从cur到goal的路径，结果会放在path里面
     * Input: 树
     * Output: 路径
     * 复杂度: O(边数)
     */
    void path_restoration(const vector<vector<pair<int, long long>>> &g, vector<int> &path, int cur, int par, int &goal) {
        path.push_back(cur);
        if (cur == goal) {
            goal = -1;
            return;
        }

        for (auto e : g[cur]) {
            int nxt = e.first;
            if (nxt == par) continue;
            path_restoration(g, path, nxt, cur, goal);
            if (goal == -1) return;
        }

        if (goal == -1) {
            return;
        }
        path.pop_back();
    }
}

```

最近公共祖先

```
#include <vector>

namespace LCA {
#define V vector
    /*
     * 最近公共祖先，限制：
     * 1. root必须为0 / 1
     * 2. 总复杂度 O(q*log(n) + n*log(n)) // q次查询，一共n个节点，建树过程 n*log(n)，每次查询 log(n)
     */
    using namespace std;
    class Tree {
    private:
        int n_;
```

```cpp
        int root_;
        int lg;
        V<int> depth;
        V<V<int>> father;
        V<V<int>> son;
        /*
         * 从跟节点dfs，来构建depth数组
         * 复杂度：O(n)
         */
        void dfs(int now, int pre = -1, int dep = 1) {
            depth[now] = dep;
            father[0][now] = pre;
            for(auto s : son[now]) {
                if(s==pre) continue;
                dfs(s, now, dep+1);
            }
        }
        /*
         * 构建祖先关系，倍增构建
         * 复杂度：O(n*log(n))
         */
        void build_father() {
            for(int i = 1; i < lg; i++) {
                for(int j = root_; j < n_ + (root_==1); j++) {
                    father[i][j] = (father[i-1][j] == -1) ? -1 : father[i-1][father[i-1][j]];
                }
            }
        }
    public:
        Tree(int root, int n) {
            root_ = root;
            n_ = n;
            lg = 1;
            while((1<<lg) < n) lg++;
            depth.resize(n+(root==1));
            father = V<V<int>>(lg, V<int>(n+(root==1), -1));
            son = V<V<int>>(n+(root==1), V<int>(0));
        }

        /*
         * 增加一个父子关系，now的父亲是pre
         * 复杂度：O(1)
         */
        void add(int pre, int now) {
            son[pre].push_back(now);
```

```
                son[now].push_back(pre);
            }

            /*
             * 完整建树，需要在add完所有父子关系才可以调用
             * 复杂度：O(n*log(n))
             */
            void build() {
                this->dfs(root_);
                this->build_father();
            }

            /*
             * 查询u和v的最近公共祖先
             * 复杂度：O(log(n))
             */
            int query_lca(int u, int v) {
                if(depth[u] > depth[v]) {
                    swap(u, v);
                }
                int depth_diff = depth[v] - depth[u];
                for(int i = 0; i < lg; i++) {
                    if(depth_diff & (1<<i)) {
                        v = father[i][v];
                    }
                }
                if(u == v) {
                    return u;
                }
                for(int i = lg - 1; i >= 0; i--) {
                    if(father[i][u] != father[i][v]) {
                        u = father[i][u];
                        v = father[i][v];
                    }
                }
                return father[0][u];
            }
        };
}
```

树同构

```
#include <vector>
#include <map>
#include <algorithm>
```

```cpp
namespace TreeIsomorphism {
    /*
     * 判断有根树是否同构，限制：
     * 1. root只能为0或者1
     * 2. 复杂度 O(n*log(n))
     */
    using namespace std;
    class Tree {
    private:
        int n_, root_;
        vector<vector<int>> son;
        map<vector<int>, int> mp;
        vector<int> t;
        /*
         * 建树过程
         * 每次会从root开始遍历，记录每个子节点的编号，并从map中判断是否已经存在同构结构
         * 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
         */
        void dfs(int root, int pre) {
            vector<int> v(1, 0);
            for(auto i : son[root]) {
                if(i==pre) continue;
                dfs(i, root);
                v.push_back(t[i]);
            }
            sort(v.begin(), v.end());
            if(mp.find(v) == mp.end()) {
                mp[v] = mp.size();
            }
            t[root] = mp[v];
        }

    public:
        Tree(int n, int root) {
            n_ = n;
            root_ = root;
            t = vector<int>(n+(root==1), 0);
            son.resize(n+(root==1));
        }

        /*
         * 添加边
         * 这里只要是无向边就行了，不需要保证u是v的父亲
         */
        void add_edge(int u, int v) {
```

```
            son[u].push_back(v);
            son[v].push_back(u);
        }

        /*
         * 构建树
         * 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
         ↪  的子树是同构的
         */
        void build() {
            dfs(root_, -1);
        }

        /*
         * 返回有多少个不同构的子树
         */
        int get_size() {
            return mp.size();
        }

        /*
         * 返回每个子树的编号
         * Input: 每个节点的编号
         * Output: 每个子树的结构编号
         * 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
         */
        int get_mask(int i) {
            return t[i];
        }
    };
}
```

图论

单源最短路

```cpp
#include <vector>
#include <set>
#include <algorithm>
namespace ShortestPath {
    /*
     * 单源最短路
     * 限制:
     *       - 复杂度O (N+M)
     *       - 无负权
     *       - 最长路径不超过1e18
     *       - 有向图 / 无向图都可以
     *       - 点的标号从0开始
     */
    class Graph {
    private:
        int N, M; // n point, m edge
        std::vector<std::vector<std::pair<long long, int>>> sons; // first = value, second = to
        std::vector<long long> distance; // 记录每个点的最近距离
        std::set<std::pair<long long, int>> q; // 一个临时的queue
        std::vector<int> per; // 记录从s开始到当前位置的最短路径中, 当前节点的前一个节点是什么

    public:
        Graph(int n, int m) : N(n), M(m) {
            sons = std::vector<std::vector<std::pair<long long, int>>>(N);
        }

        void add_edge(int s, int t, long long v) {
            sons[s].push_back({v, t});
        }

        /*
         * 单源最短路, 复杂度O(N+M)
         * 会从s开始建图, 构建每个从s可达的点的距离 & 路径
         */
        void build(int s) {
            per = std::vector<int>(N, -1);
            distance = std::vector<long long>(N, (long long) 1e18);
            q.clear();
```

```
            distance[s] = 0;
            q.insert({0, s});
            while(!q.empty()) {
                auto [v, p] = *q.begin();
                q.erase(q.begin());
                if(distance[p] < v) continue;

                for(auto [v2, p2] : sons[p]) {
                    if(distance[p2] > distance[p] + v2) {
                        q.erase({distance[p2], p2});
                        per[p2] = p;
                        distance[p2] = distance[p] + v2;
                        q.insert({distance[p2], p2});
                    }
                }
            }
        }

        /*
         * build完成后才可以调用，并且二者s需要相同，纯找路径
         * 复杂度O(path_length)
         */
        std::pair<long long, std::vector<int>> shortest_path(int s, int t) {
            std::pair<long long, std::vector<int>> ret;
            ret.first = distance[t];
            if(ret.first == 1e18) return ret; // 不存在最短路
            ret.second.push_back(t);
            while(ret.second[ret.second.size()-1] != s) {
                ret.second.push_back(per[ret.second[ret.second.size()-1]]);
            }
            std::reverse(ret.second.begin(), ret.second.end());
            return ret;
        }
    };
}
```

有向图-强连通分量

```
#include <cassert>
#include <vector>
namespace SCC {
    /*
```

```
 * 找出有向图的强连通分量
 * 限制:
 *      - 点的标号从0开始
 *      - 复杂度 O(N+M)
 */
class Graph{
private:
    int N;
    std::vector<std::vector<int>> g; // 有向图
    std::vector<std::vector<int>> g2; // 反向图
    std::vector<int> s; // 第一次dfs的时候记录dfs序, 然后反向dfs
    std::vector<bool> vis;

    void dfs(int u) {
        vis[u] = true;
        for(auto v : g[u]) {
            if(!vis[v]) dfs(v);
        }
        s.push_back(u);
    }
    void dfs2(int a, int c) {
        vis[a] = true;
        sccs[c].push_back(a);
        for(auto v : g2[a]) {
            if(!vis[v]) dfs2(v, c);
        }
    }
public:
    std::vector<std::vector<int>> sccs; // 保存了有哪些强连通分量, 每
    ↪ 个list是一个强连通分量

    Graph(int n) : N(n) {
        g.resize(n);
        g2.resize(n);
        vis = std::vector<bool>(n, false);
        s = std::vector<int>(0);
        sccs = std::vector<std::vector<int>>(0);
    }
    void add_edge(int s, int t) {
        g[s].push_back(t);
        g2[t].push_back(s);
    }
    void build() {
        for(int i = 0;i<N;i++) {
            if(!vis[i]) dfs(i);
        }
```

```
            vis = std::vector<bool>(N, false);
            for(int i = s.size()-1;i>=0;i--) {
                if(!vis[s[i]]) {
                    sccs.push_back(std::vector<int>());
                    dfs2(s[i], sccs.size()-1);
                }
            }
        }
    };
}
```