algorithm template

zinan.xu.dev@gmail.com

December 22, 2025

Contents

数论

素性检测

```cpp
#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;
    }
}
```

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);
```

```cpp
                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

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

```cpp
namespace TreeIsomorphism {
    /*
     * 判断树是否同构，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];
        }
    };
}
```