ACM Template

DUT ACM Lab

2021 年 10 月 6 日

目录

第一章 STL 使用

1.1 set-multiset-map

1.1.1 set

```
set<T> e;
e.clear();
e.insert();
e.size();
e.lower_bound(); // iter
e.upper_bound(); // iter
e.find();
```

1.1.2 multiset

```
mlutiset<T> e;
// the same to up
e.equal_range(); // pair
e.count();s
```

1.1.3 map

```
map<T1, T2> e;
// the same to up
e[T1] = T2;
```

1.2 unrodered STL

第二章　图论

2.1　二分图

2.1.1　二分图最大匹配

2.1.1.1　实现

第一种做法

匈牙利算法，复杂度 $\Theta(nm)$

```cpp
const int mx_n = 1005;
bool mp[mx_n][mx_n];
bool vis[mx_n];
int pre[mx_n];

bool dfs(cint loc) {
    for(int i=n+1; i<=n+m; i++) {
        if(mp[loc][i] && !vis[i]) {
            vis[i] = 1;
            if(!pre[i] || dfs(pre[i])) {
                pre[i] = loc;
                return 1;
            }
        }
    }
    return 0;
}

int main() {
    int ans = 0;
    for(int i=1; i<=n; i++) {
        memset(vis, 0, sizeof vis);
        ans += dfs(i);
    }
    cout << ans << endl;
    return 0;
}
```

第二种做法

转化为网络流，复杂度依赖选择

2.1.1.2　性质

最大独立集 = n - 最大匹配

最小点覆盖 = n - 最大独立集

2.2 树上问题

2.2.1 树的重心

2.2.1.1 实现

对于树上的每一个点，计算其所有子树中最大的子树节点数，这个值最小的点就是这棵树的重心。

```
void dfs(cint loc, cint fa) {
    int pre = 0;
    son[loc] = 1;
    for(int v: to[loc]) {
        if(v != fa) {
            dfs(v, loc);
            pre = max(pre, son[v]);
            son[loc] += son[v];
        }
    }
    pre = max(pre, n-son[loc]);
    if(pre < st) {
        st = pre; # 最大儿子的最小值
        ans = loc; # 最大儿子的编号
    }
}
```

2.2.1.2 性质

1. 一棵树最多有两个重心，且相邻
2. 以树的重心为根时，所有子树的大小都不超过整棵树大小的一半
3. 在一棵树上添加或删除一个叶子，那么它的重心最多只移动一条边的距离
4. 把两棵树通过一条边相连得到一棵新的树，那么新的树的重心在连接原来两棵树的重心的路径上
5. 树中所有点到某个点的距离和中，到重心的距离和是最小的；如果有两个重心，那么到它们的距离和一样

2.2.1.3 一点证明

首先证明如果树有两个重心，则它们必相邻

设两点分别为 a ，b ，且它们之间的简单路径经过的点的个数不为 0 ，即它们不相邻

不妨认为 a 在树中的深度大于 b

那么，点 a 向上的子树大小一定**大于**点 b 向上的子树，同时**大于**点 b 向下不经过点 a 的子树

同理，点 b 向下经过点 a 的子树一定**大于**点 a 向下的子树

所以，最大值仅由这两棵子树决定，而只要两点不相邻，上述总是成立的

不难发现，这两种子树，在 a 或 b 沿着两点间简单路径移动时会减小，不符合重心的定义

再证最多只有两个重心

显然，如果树的重心大于两个，至少有一对重心无法相邻

2.2.2 点分治

2.2.2.1 思路

如果在统计树上信息时，可以将子树内的信息单独统计，将多个子树的信息合并统计，那么可以考虑树分治。

如果对于每一次分治，复杂度为 $\Theta(N)$ ，那么 k 次递归的复杂度就是 $\Theta(kN)$

如果能保证递归次数为 log 级别，那么复杂度就会是 $\Theta(N\log N)$ ，而从重心分治就可以保证最多递归 $\log N$ 次

同时可以发现，在递归时保存所有以重心为根的子树的信息的空间复杂度也是 $\Theta(N\log N)$ 的

不会受到影响的信息有简单路径

会受到影响的信息有 LCA

如果题目需要求的信息会受到根节点选取的影响，还是不要使用点分治为好

2.2.2.2 实现

预定义部分

```
int h[10010], nx[20020], to[20020], w[20020], cnt_; // 链式前向星数组
int son[10010]; // 经过处理后的每个点的儿子个数
bool vis[10010]; // 该点是否在分治时作为子树的根
int id; // 当前所处理的树的重心
int snode; // 当前所处理树的节点数量

void add(cint f, cint t, cint co) {
    nx[++cnt_] = h[f];
    h[f] = cnt_;
    to[cnt_] = t;
    w[cnt_] = co;
}
```

统计以某点为根且不跨越其余重心的子树大小

```
int gsiz(cint loc, cint fa) {
    int sum = 1;
    for(int i=h[loc]; i; i=nx[i])
        if(to[i] != fa && !vis[to[i]]) {
            sum += gsiz(to[i], loc);
        }
    return sum;
}
```

寻找树的重心

```
void gp(cint loc, cint fa) {
    int pre = 0;
    son[loc] = 1;
    for(int i=h[loc]; i; i=nx[i])
        if(to[i] != fa && !vis[to[i]]) {
            gp(to[i], loc);
            pre = max(pre, son[to[i]]);
            son[loc] += son[to[i]];
            if(id) return;
```

```
        }
    pre = max(pre, snode - son[loc]);
    // 树的重心可能有两个，此处任取了一个
    if(pre <= snode/2) {
        id = loc;
        return;
    }
}
```

统计跨越重心的答案

```
void check(cint loc, cint fa) {
    // 统计跨越重心的答案
    for(int i=h[loc]; i; i=nx[i])
        if(to[i] != fa && !vis[to[i]]) {
            check(to[i], loc);
        }
}
```

合并子树

```
void update(cint loc, cint fa) {
    // 合并子树
    for(int i=h[loc]; i; i=nx[i]) {
        if(to[i] != fa && !vis[to[i]]) {
            update(to[i], loc);
        }
    }
}
```

解决问题

```
void sol(cint loc) {
    vis[loc] = 1;
    // 初始化计算答案与合并子树时需要用到的东西
    for(int i=h[loc]; i; i=nx[i]) {
        if(!vis[to[i]]) {
            check(to[i], loc);
            update(to[i], loc);
        }
    }
    for(int i=h[loc]; i; i=nx[i]) {
        if(!vis[to[i]]) {
            snode = gsiz(to[i], loc);
            id = 0;
            gp(to[i], loc);
            sol(id);
        }
    }
}
```

主函数里的一点东西

```
int main() {
    snode = n;
```

```
    gp(1, 1);
    sol(id);
}
```

2.2.3 边分治

2.2.4 点分树

2.2.5 树上启发式合并

2.2.5.1 思路

当子树可以单独处理，且子树信息转移到父节点较为容易时可以考虑

任意一条路径上轻边个数不超过 $\log N$

每一条轻边连接的轻子树会额外访问子树中所有的点一次，那么每个点至多被额外访问 $\Theta(\log N)$ 次

理论复杂度 $\Theta(N \log N)$

2.2.5.2 实现

预定义部分

```
vector<int> to[100100]; // 邻接表
int son[100100]; // 子树大小
int bson[100100]; // 重儿子
```

寻找重儿子

```
void fd_son(cint loc, cint fa) {
    son[loc] = 1;
    for(int v: to[loc]) {
        if(v != fa) {
            fd_son(v, loc);
            son[loc] += son[v];
            if(son[v] > son[bson[loc]]) bson[loc] = v;
        }
    }
}
```

递归主体

```
void clear() {
    // do somethings
}

void sol(cint loc, cint fa) {
    for(int v: to[loc]) {
        if(v != fa && v != bson[loc]) {
            sol(v, loc);
            clear(); // 清空函数
        }
    }
    if(bson[loc]) sol(bson[loc], loc);
    for(int v: to[loc]) {
        if(v != fa && v != bson[loc]) {
```

```cpp
            check(v, loc, a[loc]);
            update(v, loc);
        }
    }
    // 此处注意插入当前节点
}
```

统计答案

```cpp
void cacu(cint r, cint x) {
    // do somethings
}

void check(cint loc, cint fa, cint co) {
    // cacu
    for(int v: to[loc]) {
        if(v != fa) check(v, loc, co);
    }
}
```

合并子树

```cpp
void ins(cint r, cint x) {
    // do somethings
}

void update(cint loc, cint fa) {
    // ins
    for(int v: to[loc]) {
        if(v != fa) update(v, loc);
    }
}
```

主函数的一些部分

```cpp
int main() {
    fd_son(1, 1);
    sol(1, 1);
}
```

2.3 网络流

2.3.1 最大流算法

2.3.1.1 增广路算法

2.3.1.1.1 dinic

复杂度为 $\Theta(n^2m)$ ，特别地，对于二分图匹配复杂度为 $\Theta(m\sqrt{n})$

```cpp
const int mxn = 202; // 最大点数
const int mxm = 5005; // 最大边数
struct dinic {
    int s, t; // 源, 汇
    ll w[mxm*2];
    int h[mxn], nx[mxm*2], to[mxm*2], cnt=1; // 链式前向星, cnt=1
        方便找到反边
```

```cpp
    int cur[mxn]; // 当前弧优化辅助数组
    int label[mxn]; // 分层图标号
    void clear_map() { cnt = 1; for(int i=1; i<=mxn; i++) h[i] = 0; }
    void add(cint f, cint t, cint co) {
        nx[++cnt] = h[f]; h[f] = cnt; to[cnt] = t; w[cnt] = co;
        nx[++cnt] = h[t]; h[t] = cnt; to[cnt] = f; w[cnt] = 0;
    }
    bool bfs() {
        memset(label, 0x3f, sizeof label); // 初始化标号
        memcpy(cur, h, sizeof h); // 当前弧优化
        queue<int> q;
        q.push(s);
        label[s] = 0;
        while(!q.empty()) {
            int r = q.front();
            q.pop();
            for(int i=h[r]; i; i=nx[i])
                if(w[i] && label[to[i]] > label[r]+1) {
                    label[to[i]] = label[r] + 1;
                    q.push(to[i]);
                }
        }
        return label[t] < label[mxn-1];
    }
    ll dfs(cint loc, ll cap) {
        if(loc == t) return cap;
        ll sum = 0;
        for(int i=cur[loc]; i; i=nx[i]) {
            cur[loc] = i;
            if(label[to[i]] == label[loc] + 1) {
                ll d = dfs(to[i], min(cap, w[i]));
                if(d) { w[i] -= d; w[i^1] += d; cap -= d; sum += d; }
                if(!cap) break;
            }
        }
        return sum;
    }
    ll max_flow(cint s, cint t) {
        this->s = s;
        this->t = t;
        ll flow = 0;
        while(bfs()) { flow += dfs(s, inf_ll); }
        return flow;
    }
};
```

2.3.1.2 预流推进算法

第三章　数论

3.1　逆元

3.1.1　逆元的一些实现

注意，不是所有时候都有逆元

3.1.1.1　单个数的逆元

```
ll ksm(ll m, int c) {
    ll ans = 1;
    while(c) {
        if(c&1) ans = (ans*m) % mod;
        c >>= 1;
        m = (m*m) % mod;
    }
    return ans;
}

ll inv(ll x) { return ksm(x, mod-2); }
```

3.1.1.2　阶乘的线性逆元

```
ll ksm(ll m, int c) {
    ll ans = 1;
    while(c) {
        if(c&1) ans = (ans*m) % mod;
        c >>= 1;
        m = (m*m) % mod;
    }
    return ans;
}
void sol_inv() {
    fac[0] = 1;
    for(int i=1; i<=mx_n; i++) fac[i] = fac[i-1] * i % mod;
    inv[mx_n] = ksm(fac[mx_n], mod-2);
    for(int i=mx_n-1; i; i--) inv[i] = inv[i+1] * (i+1) % mod;
}
```

3.1.1.3　1 到 n 的线性逆元

```
inv[1] = 1;
for(int i=2; i<=n; i++) inv[i] = (mod-mod/i) * inv[mod%i] % mod;
```

3.1.2 逆元存在性

3.1.2.1 p 与 b 不互质

逆元不存在

3.1.2.2 模数 p 为质数

根据费马小定理，数 b 的逆元为 b^{p-2}

3.1.2.3 模数 p 不为质数

如果 p 与 b 互质，数 b 的逆元为 $b^{\varphi(p)-1}(mod\ \ p)$

3.2 筛法

3.2.1 埃氏筛（素数）

复杂度 $\Theta(N\log\log N)$

3.2.2 欧拉筛（素数，积性函数）

复杂度 $\Theta(N)$

3.2.2.1 素数

每一个合数可以被唯一的分解为一个最小质数和另一个合数的乘积
正确性证明每一个数仅被分解到一次且每一个数都被分解到就好
同时得到每个数的最小因数

```
const int mx_n = 100000000;
int vis[mx_n+1000];
int prim[mx_n+1000], cnt;

void liner_sieve(cint x) {
    int rt = 0;
    for(int i=2; i<=x; i++) {
        if(!vis[i]) {
            prim[++cnt] = i;
            vis[i] = i;
        }
        for(int j=1; j<=cnt; j++) {
            if(1ll*prim[j]*i > x) break;
            if(prim[j] > vis[i]) break;
            vis[prim[j]*i] = prim[j];
        }
    }
}
```

省空间的写法

```
const int mx_n = 100000000;
int n, q;
bool vis[mx_n+1000];
int prim[mx_n+1000], cnt;

```

```
void liner_sieve(cint x) {
    int rt = 0;
    for(int i=2; i<=x; i++) {
        if(!vis[i]) {
            prim[++cnt] = i;
        }
        for(int j=1; j<=cnt; j++) {
            if(1ll*prim[j]*i > x) break;
            vis[prim[j]*i] = 1;
            if(!(i%prim[j])) break;
        }
    }
}
```

3.2.2.2 积性函数

对于积性函数 f ，有 $f(1)=1$ 且当 $\gcd(a,b)=1$ 时有 $f(ab)=f(a)f(b)$
用欧拉筛筛积性函数大概有以下几个步骤

1. 对于质数 p ，求出 $f(p)$
2. 对于 $\gcd(p,q)=1$ 的情况，求出 $f(pq)=f(p)f(q)$
3. 对于 $\gcd(p,q)\neq 1$ 的情况，求出 $f(pq)$ 的值（对于完全积性函数，这一步可以归到 2 中）

   或者说对于质数 p ，求出 $f(p^k)$ 的值

```
// 求欧拉函数
void liner_sieve(cint x) {
    int rt = 0;
    for(int i=2; i<=x; i++) {
        if(!vis[i]) {
            prim[++cnt] = i;
            phi[i] = i-1; // 1
        }
        for(int j=1; j<=cnt; j++) {
            if(1ll*prim[j]*i > x) break;
            vis[prim[j]*i] = 1;
            if(!(i%prim[j])) {
                phi[i*prim[j]] = phi[i] * prim[j]; // 3
                break;
            }
            else phi[i*prim[j]] = phi[i] * phi[prim[j]]; // 2
        }
    }
}
```

3.3 MR 素性检验

3.4 狄利克雷卷积

第四章　多项式

4.1　快速傅里叶变换

第五章　计算几何

第六章　博弈理论

第七章　数据结构

7.1　树状数组

```cpp
int bnode[mx_n];

int lowbit(int &x) { return x&-x; }

void add(int x, cint co) {
    while(x <= mx_n) {
        bnode[x] += co;
        x += lowbit(x);
    }
}

int query(int x) {
    int ans = 0;
    while(x) {
        ans += bnode[x];
        x -= lowbit(x);
    }
    return ans;
}
```

注意，树状数组无法直接处理 **0** ，需要处理一下

第八章　组合统计

8.1　常见的恒等式与组合结论

8.1.1　结论

1. K_n 的生成数的个数为 n^{n-2}
2. n 个点的凸多边形对角线的交点最多为 C_n^4 ，其中不超过两条对角线在内部的任何一点相交

8.1.2　恒等式

1. $\Sigma_{i=1}^{n} C_i^x = C_{n+1}^{x+1}$

第九章　典型题型

9.1　最长上升子序列

9.1.1　做法

第一种：

dp，复杂度 $\Theta(N^2)$ ，优点是可以记录子序列

```
// Nope
```

第二种：

贪心，复杂度 $\Theta(N \log N)$ ，优点是复杂度低

```
int mx[mx_n];
int r = 0;
memset(mx, 0x3f, sizeof mx);
mx[0] = 0;
for(int i=1; i<=m; i++) {
    int id = lower_bound(mx, mx+r+1, c[i]) - mx;
    mx[id] = c[i];
    r = max(r, id);
}
# 最 后 mx 数 组 合 法 的 最 大 的 下 标 就 是 答 案
```

第十章　杂项

10.1　位运算

1. $a + b = a \& b + a | b$
2. 奇数个数的**与和**一定小于等于其**异或和**，偶数个数的与和为 1 的位置，异或和在该位置一定为 0

10.2　莫队算法

10.2.1　普通莫队

10.2.1.1　普通莫队

一些定义，其中 $block\_siz = \frac{n}{\sqrt{m}}$

```cpp
int block_siz; // 块大小
struct query {
    int l, r; // 询问区间
    int bl, br; // 区间端点所属的块的编号
    int id;
    int ans;
    void init(cint x) {
        id = x;
        cin >> l >> r;
        bl = l / block_siz;
        br = r / block_siz;
    }
} b[200200];
```

主要部分

```cpp
for(int i=1; i<=q; i++) {
    while(r < b[i].r) ++r, add(r);
    while(l > b[i].l) --l, add(l);
    while(r > b[i].r) --r, dele(r+1);
    while(l < b[i].l) ++l, dele(l-1);
    b[i].ans = siz;
}
```

奇偶化排序优化

```cpp
bool cmp1(const query&a, const query&b) {
    return a.bl == b.bl ? ( a.bl&1 ? a.r < b.r : a.r > b.r) : a.bl
        < b.bl;
}
```

10.2.2 回滚莫队

10.3 cdq 分治