最小生成树算法 2024/07/22

Kruskal **加边法**（并查集思想） 复杂度 $O(m \log m)$

（1）边权从小到大排序.

（2）把每个点看作是独立的集合.

（3）按边权从小到大选边，只要边两点不在同一集合就合并.

（4）重复（3），直到选出 $n-1$ 条边.

```
struct edge {
    int u,v,w;
}e[];

int getf(int x) {
    return x==f[x]?x:f[x]=getf(f[x]);
}

void merge(int a,int b) {
    int tx=getf(a),ty=getf(b);
    if (tx!=ty) {
        f[tx]=ty;
    }
}

bool cmp(edge x,edge y) {
    return x.w<y.w;
}

main() {
    for (int i=1;i<=m;i++) {
        cin>>e[i].u>>e[i].v>>e[i].w;
        f[i]=i;
    }
    sort(e+1,e+m+1,cmp);
    for (int i=1;i<=m;i++) {
        if (getf(e[i].u)!=getf(e[i].v)) {
            merge(e[i].u,e[i].v);
            ans+=e[i].w;
            cnt++;
        }
        if (cnt==n-1) {
            break;
        }
    }
}
```

Prim **加点法**

（1）从图上任意一点开始.

`book[]` 标记是否加入生成树，`dis[]` 标记 `i` 点离生成树最小的距离.

（2）找距离生成树最近的点（不在生成树上）.

（3）标记这个点在生成树中.

（4）用这个点更新 `dis[]`.

（5）重复（3）（4），直到所有点都被标记.

邻接矩阵：

```
main() {
    for (int i=1;i<=n;i++) {
        dis[i]=e[1][i];
    }
    while (cnt<n) {
        mn=0x7f; // infinity
        for (int i=1;i<=n;i++) {
            if (book[i]==0&&dis[i]<mn) {
                mn=dis[i];
                j=i;
            }
        }
        book[j]=1;
        for (int i=1;i<=n;i++) {
            if (book[i]==0&&dis[i]<e[j][i]) {
                dis[i]=e[j][i];
            }
        }
    }
}
```

最短路径算法

Floyd **算法**　复杂度 $O(n^3)$

```
void floyd() {
    for (int i=1;i<=n;i++) {
        for (int j=1;j<=n;j++) {
            for (int k=1;k<=n;k++) {
                a[i][j]=min(a[i][j],a[i][k]+a[k][j]);
            }
        }
    }
}
```

Dijkstra **算法**　复杂度 $O(n^2)$

```
void dijkstra(int s) {
    int mn,n;
    for (int i=1;i<=n;i++) {
        vis[i]=0;
        dis[i]=way[s][i];
    }
    vis[s]=1;
    for (int i=1;i<=n;i++) {
        int u;
        mn=0x3f3f3f3f; // infinity
        u=-1;
        for (int j=1;j<=n;j++) {
            if (vis[j]==0&&dis[j]<mn) {
                u=j;
                mn=dis[j];
            }
        }
        if (u==-1) {
            break;
        }
        vis[u]=1;
        for (int j=1;j<=n;j++) {
            if (vis[j]==0) {
                if (dis[u]+way[u][j]<dis[j]) {
                    dis[j]=dis[u]+way[u][j];
                }
            }
        }
    }
}
```

Bellman-Ford **算法**　复杂度 $O(mn)$

对每一条边进行 $n-1$ 轮标记，会重复进队且有意义.

```
void bf() {
    for (int k=1;k<n;k++) {
        bool flag=false;
        for (int i=1;i<=m;i++) {
            if (dis[v[i]]>dis[u[i]]+w[i]) {
                dis[v[i]]=dis[u[i]]+w[i];
                flag=true;
            }
        }
        if (!flag) {
            break;
        }
    }
    bool flag=false;
    for (int i=1;i<=m;i++) {
        if (dis[v[i]]>dis[u[i]]+w[i]) {
            dis[v[i]]=dis[u[i]]+w[i];
            flag=true;
        }
    }
    if (!flag) {
        break;
    }
}
```

SPFA **算法**　复杂度 $O(km)$，其中 k 可能为 2，n、m 约为 10^5 时不适用.

（1）取队首 u，对它的出边松弛.

（2）如果 $u \to v$ 使 `dis[v]` 更小，并且 v 不在队列，把 v 入队.

（3）重复（1）（2），直到队空.

需要重复进队.

```
int in[],dis[],vis[];
bool SPFA(int s) {
    for (int i=1;i<=3*n;i++) {
        dis[i]=1e10;
    }
    queue<int> Q;
    Q.push(s);
    vis[s]=true;
    dis[s]=0;
    while (!Q.empty()) {
        int now=Q.front();
        Q.pop();
        vis[now]=false;
        for (int i=head[now];i!=0;i=des[i].next) {
            int to=eds[i].to;
            if (dis[to]>dis[now]+eds[i].cost) {
                dis[to]=dis[now]+eds[i].cost;
                if (!vis[to]) {
                    vis[to]=true;
                    Q.push(to);
                    if (++in[to]>n) {
                        return false;
                    }
                }
            }
        }
    }
    return true;
}
```

查找强连通分量 2024/07/23

连通图：图上任意两点可达.
强连通：有向图上两点互相可达，这两点强连通.
强连通图：有向图上任意两点互相可达，这个图是强连通图.
弱连通图：有向图看作无向图，任意两点互相可达，这个图是弱连通图.
强连通分量（Strongly Connected Components，SCC）：极大的强连通子图.
树边：每次搜索找到一个还没有访问过的结点的时候就形成了一条树边.
返祖边：也叫回边，即指向祖先结点的边（用栈来判断）.
横叉边：主要是在搜索的时候遇到了一个已经访问过且不在栈中的结点.
前向边：在搜索的时候遇到子树中的结点（`dfn[u]<dfn[v]`）的时候形成的.

Tarjan **算法**

`dfn[]` 时间戳 `low[]` 追溯值（最早的时间戳）

（1）任选一点 u 开始 DFS，记录下当前的 `dfn[u]` 和 `low[u]` 的值为 `num`，把 u 入栈.

（2）遍历 u 的子节点 v. $\begin{cases} 1.\ v\text{没有被搜索过} \to \texttt{dfs(v) low[u]=min(low[u],low[v])} \\ 2.\ v\text{被搜索过，判断}v\text{在不在栈里，}u \to v\text{返祖为}\texttt{low[u]-min(low[u],dfn[v])} \end{cases}$

（3）`low[u]` 确定，`if (low[u]==dfn[u])`，以 u 为根的树是一个强连通分量，栈里 u 及其后入栈的点全部出栈.

```
vector<int> e[maxn];
int dfn[],low[],tot;
int stk[],instk[],top;
int scc[],siz[].cnt;

void tarjan(int x) {
    dfn[x]=low[x]=++tot;
    stk[++top]=x;
    instk[x]=1;
    for (int y:e[x]) {
        if (!dfn[y]) {
            tarjan(y);
            low[x]=min(low[x],low[y]
        } else if {
            low[x]=min(low[x],dfn[y])
        }
    }
    if (dfn[x]==low[x]) {
        ++cnt;
        while (stk[top+1]!=x) {
            int v=stk[top--];
            instk[v]=0;
            siz[cnt]++;
        }
    }
}
```

二分图匹配 2024/07/24

匈牙利算法（二分图最大匹配）

参考：二分图匹配——通俗易懂 - cnblog

板子题 - HDU 2063 **过山车**

RPG girls 今天和大家一起去游乐场玩，终于可以坐上梦寐以求的过山车了. 可是，过山车的每一排只有两个座位，而且还有条不成文的规矩，就是每个女生必须找个个男生做 partner 和她同坐. 但是，每个女孩都有各自的想法，举个例子吧，Rabbit 只愿意和 XHD 或 PQK 做 partner，Grass 只愿意和 linle 或 LL 做 partner，PrincessSnow 愿意和水域浪子或伪酷儿做 partner. 考虑到经费问题，boss 刘决定只让找到 partner 的人去坐过山车，其他的人，嘿嘿，就站在下面看着吧. 聪明的 Acmer，你可以帮忙算算最多有多少对组合可以坐上过山车吗?

Input

输入数据的第一行是三个整数 K，M，N，分别表示可能的组合数目，女生的人数，男生的人数.0<K≤1000，1≤N 和 M≤500. 接下来的 K 行，每行有两个数，分别表示女生 A_i 愿意和男生 B_j 做 partner. 最后一个 0 结束输入.

Output

对于每组数据，输出一个整数，表示可以坐上过山车的最多组合数.

```cpp
const int N=505;
int line[N][N];
int girl[N],used[N];
int k,m,n;
bool found(int x) {
    for (int i=1;i<=n;i++) {
        if (line[x][i]&&!used[i]) {
            used[i]=1;
            if (girl[i]==0||found(girl[i])) {
                girl[i]=x;
                return 1;
            }
        }
    }
    return 0;
}

int main() {
    int x,y;
    while (scanf("%d",&k)&&k) {
        scanf("%d %d",&m,&n);
        memset(line,0,sizeof(line));
        memset(girl,0,sizeof(girl));
        for (int i=0; i<k; i++) {
            scanf("%d %d",&x,&y);
            line[x][y]=1;
        }
        int sum=0;
        for (int i=1; i<=m; i++) {
            memset(used,0,sizeof(used));
            if (found(i)) sum++;
        }
        printf("%d\n",sum);
    }
    return 0;
}
```

二分图最小点覆盖

最小点覆盖：选最少的点，满足每条边至少有一个端点被选.

König **定理**：二分图中，最小点覆盖 = 最大匹配.

二分图最大独立集

最大独立集：选最多的点，满足两两之间没有边相连.

二分图中，最大独立集 $= n -$ 最小点覆盖.

数论

费马小定理

若 p 为素数，$\gcd(a,p)=1$，则 $a^{p-1}\equiv 1 \pmod{p}$.

即 $\forall a \in \mathbf{Z}$，有 $a^p \equiv a \pmod{p}$.

快速幂

(1) mod 乘简单，除困难.

(2) $a^{p-1}\equiv 1 \pmod{p}$，p 为质数，$\gcd(a,p)=1$.

(3) $\dfrac{1}{a} \equiv x \pmod{p}$.

(4) p 为质数，$\boxed{a^{-1}\equiv a^{p-2}}$.

递归法

```cpp
long long binpow(long long a,long long b) {
    if (b==0) return 0;
    long long res=binpow(a,b/2);
    if (b%2) return res*res*a;
    else return res*res;
}
```

字符串哈希

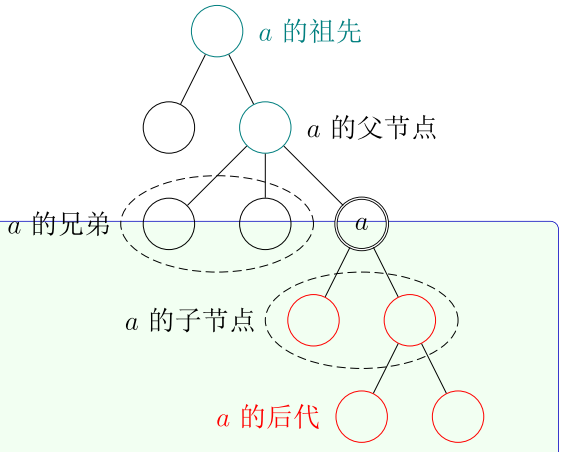
两个质数：10^9+7 和 998244353

定义 $H(\texttt{string})=\texttt{int}$，$H[i]=P\cdot H[i-1]+i-1$.

```cpp
using std::string;

const int M = 1e9 + 7;
const int B = 233;

typedef long long ll;

int get_hash(const string& s) {
    int res = 0;
    for (int i = 0; i < s.size(); ++i) {
        res = ((ll)res * B + s[i]) % M;
    }
    return res;
}

bool cmp(const string& s, const string& t) {
    return get_hash(s) == get_hash(t);
}
```

树 2024/07/25

无根树：一个没有固定根结点的树.
有根树：在无根树的基础上，指定一个结点称为根，则形成一棵有根树.
父亲（parent node）：对于除根以外的每个结点，定义为从该结点到根路径上的第二个结点. 根结点没有父结点.
祖先（ancestor）：一个结点到根结点的路径上，除了它本身外的结点. 根结点的祖先集合为空.
子结点（child node）：若 u 是 v 的父亲，则 v 是 u 的子结点. 子结点的顺序一般不加以区分，二叉树是一个例外.
结点的深度（depth）：到根结点的路径上的边数.
树的高度（height）：所有结点的深度的最大值.
兄弟（sibling）：同一个父亲的多个子结点互为兄弟.
后代（descendant）：如果 u 是 v 的祖先，那么 v 是 u 的后代.



二叉树的 DFS 遍历

先序遍历

```cpp
void preorder(BiTree* root) {
    if (root) {
        cout << root->key << " ";
        preorder(root->left);
        preorder(root->right);
    }
}
```

中序遍历

```cpp
void inorder(BiTree* root) {
    if (root) {
        inorder(root->left);
        cout << root->key << " ";
        inorder(root->right);
    }
}
```

后序遍历

```cpp
void postorder(BiTree* root) {
    if (root) {
        postorder(root->left);
        postorder(root->right);
        cout << root->key << " ";
    }
}
```

树的直径

树上任意两节点之间最长的简单路径即为树的直径.
若树上所有边边权均为正，则树的所有直径中点重合.

最近公共祖先（LCA）

两个节点的最近公共祖先，就是这两个点的公共祖先里面，离根最远的那个.

性质

1. $\mathrm{LCA}(\{u\}) = u$.
2. u 是 v 的祖先，当且仅当 $\mathrm{LCA}(u, v) = u$.
3. 如果 u 不为 v 的祖先并且 v 不为 u 的祖先，那么 u、v 分别处于 $\mathrm{LCA}(u, v)$ 的两棵不同子树中.
4. 前序遍历中，$\mathrm{LCA}(S)$ 出现在所有 S 中元素之前，后序遍历中 $\mathrm{LCA}(S)$ 则出现在所有 S 中元素之后.
5. 两点集并的最近公共祖先为两点集分别的最近公共祖先的最近公共祖先，即 $\mathrm{LCA}(A \cup B) = \mathrm{LCA}(\mathrm{LCA}(A), \mathrm{LCA}(B))$.
6. 两点的最近公共祖先必定处在树上两点间的最短路上.
7. $d(u, v) = h(u) + h(v) - 2h(\mathrm{LCA}(u, v))$，其中 d 是树上两点间的距离，h 代表某点到树根的距离.

Tarjan 算法

一种离线算法，需要使用并查集记录某个结点的祖先结点.

（1）首先接受输入边（邻接链表）、查询边（存储在另一个邻接链表内）. 查询边其实是虚拟加上去的边，为了方便，每次输入查询边的时候，将这个边及其反向边都加入到 `queryEdge[]` 里.

（2）然后对其进行一次 DFS 遍历，同时使用 `visited[]` 数组进行记录某个结点是否被访问过、`parent[]` 记录当前结点的父亲结点.

（3）其中涉及到了回溯思想，我们每次遍历到某个结点的时候，认为这个结点的根结点就是它本身. 让以这个结点为根节点的 DFS 全部遍历完毕了以后，再将这个结点的根节点设置为这个结点的父一级结点.

（4）回溯的时候，如果以该节点为起点，`queryEdge` 查询边的另一个结点也恰好访问过了，则直接更新查询边的 `LCA` 结果.

（5）最后输出结果.

倍增算法

板子题 - HDU 2586 How Far Away?

Problem Description

There are n houses in the village and some bidirectional roads connecting them. Every day people always like to ask like this "How far is it if I want to go from house A to house B?" Usually it hard to answer. But luckily int this village the answer is always unique, since the roads are built in the way that there is a unique simple path ("simple" means you can't visit a place twice) between every two houses. Yout task is to answer all these curious people.

Input

First line is a single integer T (T$\leqslant$10), indicating the number of test cases.

For each test case, in the first line there are two numbers n (2$\leqslant$n$\leqslant$40000) and m (1$\leqslant$m$\leqslant$200), the number of houses and the number of queries. The following $n-1$ lines each consisting three numbers i, j, k, separated bu a single space, meaning that there is a road connecting house i and house j, with length k (0<k$\leqslant$40000). The houses are labeled from 1 to n.

Next m lines each has distinct integers i and j, you areato answer the distance between house i and house j.

Output

For each test case, output m lines. Each line represents the answer of the query. Output a bland line after each test case.

```cpp
#define MXN 40005
std::vector<int> v[MXN];
std::vector<int> w[MXN];

int fa[MXN][31], cost[MXN][31], dep[MXN];
int n, m, a, b, c;

void dfs(int root, int fno) {
    fa[root][0] = fno;
    dep[root] = dep[fa[root][0]] + 1;
    for (int i = 1; i < 31; ++i) {
        fa[root][i] = fa[fa[root][i - 1]][i - 1];
        cost[root][i] = cost[fa[root][i - 1]][i - 1] + cost[root][i - 1];
    }
    int sz = v[root].size();
    for (int i = 0; i < sz; ++i) {
        if (v[root][i] == fno) continue;
        cost[v[root][i]][0] = w[root][i];
        dfs(v[root][i], root);
    }
}

int lca(int x, int y) {
    if (dep[x] > dep[y]) swap(x, y);
    int tmp = dep[y] - dep[x], ans = 0;
    for (int j = 0; tmp; ++j, tmp >>= 1)
        if (tmp & 1) ans += cost[y][j], y = fa[y][j];
    if (y == x) return ans;
    for (int j = 30; j >= 0 && y != x; --j) {
        if (fa[x][j] != fa[y][j]) {
            ans += cost[x][j] + cost[y][j];
            x = fa[x][j]; y = fa[y][j];
        }
    }
    ans += cost[x][0] + cost[y][0];
    return ans;
}

void Solve() {
    memset(fa, 0, sizeof(fa));
    memset(cost, 0, sizeof(cost));
    memset(dep, 0, sizeof(dep));
    for (int i = 1; i <= n; ++i) {
        v[i].clear(); w[i].clear();
    }
    for (int i = 1; i < n; ++i) {
        scanf("%d %d %d", &a, &b, &c);
        v[a].push_back(b); v[b].push_back(a);
        w[a].push_back(c); w[b].push_back(c);
    }
    dfs(1, 0);
    for (int i = 0; i < m; ++i) {
        scanf("%d %d", &a, &b); printf("%d\n", lca(a, b));
    }
}

int main() {
    int T; scanf("%d", &T);
    while (T--) Solve();
    return 0;
}
```

树的重心

如果在树中选择某个节点并删除，这棵树将分为若干棵子树，统计子树节点数并记录最大值. 取遍树上所有节点，使此最大值取到最小的节点被称为整个树的重心.

求法

```cpp
int size[MAXN],    // 这个节点的「大小」(所有子树上节点数 + 该节点)
    weight[MAXN],  // 这个节点的「重量」,即所有子树「大小」的最大值
    centroid[2];   // 用于记录树的重心(存的是节点编号)

void GetCentroid(int cur, int fa) {  // cur 表示当前节点 (current)
    size[cur] = 1;
    weight[cur] = 0;
    for (int i = head[cur]; i != -1; i = e[i].nxt) {
        if (e[i].to != fa) {  // e[i].to 表示这条有向边所通向的节点.
            GetCentroid(e[i].to, cur);
            size[cur] += size[e[i].to];
            weight[cur] = max(weight[cur], size[e[i].to]);
        }
    }
    weight[cur] = max(weight[cur], n - size[cur]);
    if (weight[cur] <= n / 2) {  // 依照树的重心的定义统计
        centroid[centroid[0] != 0] = cur;
    }
}
```

树上随机游走

给定一棵有根树，树的某个结点上有一个硬币，在某一时刻硬币会等概率地移动到邻接结点上，问硬币移动到邻接结点上的期望距离.

向父结点走的期望距离（来源于 OI Wiki）

设 $f(u)$ 代表 u 结点走到其父结点 p_u 的期望距离，则有：

$$f(u)=\frac{w(u,p_u)+\sum\limits_{v\in \text{son}_u}(w(u,v)+f(v)+f(u))}{d(u)}$$

分子中的前半部分代表直接走向了父结点，后半部分代表先走向了子结点再由子结点走回来然后再向父结点走；分母 $d(u)$ 代表从 u 结点走向其任何邻接点的概率相同.

化简如下：

$$\begin{aligned}f(u)&=\frac{w(u,p_u)+\sum\limits_{v\in \text{son}_u}(w(u,v)+f(v)+f(u))}{d(u)}\\&=\frac{w(u,p_u)+\sum\limits_{v\in \text{son}_u}(w(u,v)+f(v))+(d(u)-1)f(u)}{d(u)}\\&=w(u,p_u)+\sum_{v\in \text{son}_u}(w(u,v)+f(v))\\&=\sum_{(u,t)\in E}w(u,t)+\sum_{v\in \text{son}_u}f(v)\end{aligned}$$

对于叶子结点 l，初始状态为 $f(l)=w(p_l,l)$.

当树上所有边的边权都为 1 时，上式可化为：

$$f(u) = d(u) + \sum_{v \in \text{son}_u} f(v)$$

即 u 子树的所有结点的度数和，也即 u 子树大小的两倍 -1（每个结点连向其父亲的边都有且只有一条，除 u 与 p_u 之间的边只有 1 点度数的贡献外，每条边会产生 2 点度数的贡献）.

向子结点走的期望距离

设 $g(u)$ 代表 p_u 结点走到其子结点 u 的期望距离，则有：

$$g(u) = \frac{w(p_u, u) + (w(p_u, p_{p_u}) + g(p_u) + g(u)) + \sum_{s \in \text{sibling}_u} (w(p_u, s) + f(s) + g(u))}{d(p_u)}$$

分子中的第一部分代表直接走向了子结点 u，第二部分代表先走向了父结点再由父结点走回来然后再向 u 结点走，第三部分代表先走向 u 结点的兄弟结点再由其走回来然后再向 u 结点走；分母 $d(p_u)$ 代表从 p_u 结点走向其任何邻接点的概率相同. 化简如下：（初始状态为 $g(\text{root}) = 0$）

$$\begin{aligned}
g(u) &= \frac{w(p_u, u) + (w(p_u, p_{p_u}) + g(p_u) + g(u)) + \sum_{s \in \text{sibling}_u} (w(p_u, s) + f(s) + g(u))}{d(p_u)} \\
&= \frac{w(p_u, u) + w(p_u, p_{p_u}) + g(p_u) + \sum_{s \in \text{sibling}_u} (w(p_u, s) + f(s)) + (d(p_u) - 1)g(u)}{d(p_u)} \\
&= w(p_u, u) + w(p_u, p_{p_u}) + g(p_u) + \sum_{s \in \text{sibling}_u} (w(p_u, s) + f(s)) \\
&= \sum_{(p_u, t) \in E} w(p_u, t) + g(p_u) + \sum_{s \in \text{sibling}_u} f(s) \\
&= \sum_{(p_u, t) \in E} w(p_u, t) + g(p_u) + \left(f(p_u) - \sum_{(p_u, t) \in E} w(p_u, t) - f(u) \right) \\
&= g(p_u) + f(p_u) - f(u)
\end{aligned}$$

无权树写法

```cpp
vector<int> G[maxn];

void dfs1(int u, int p) {
    f[u] = G[u].size();
    for (auto v : G[u]) {
        if (v == p) continue;
        dfs1(v, u); f[u] += f[v];
    }
}

void dfs2(int u, int p) {
    if (u != root) g[u] = g[p] + f[p] - f[u];
    for (auto v : G[u]) {
        if (v == p) continue;
        dfs2(v, u);
    }
}
```

树上启发式合并

启发式算法是基于人类的经验和直观感觉，对一些算法的优化.

```cpp
const int N = 2e5 + 5;
vector<int> g[N];

int sz[N], big[N], col[N], L[N], R[N], Node[N], totdfn;
int ans[N], cnt[N], totColor;

void add(int u) { if (cnt[col[u]] == 0) ++totColor; cnt[col[u]]++; }

void del(int u) { cnt[col[u]]--; if (cnt[col[u]] == 0) --totColor; }

int getAns() { return totColor; }

void dfs0(int u, int p) {
    L[u] = ++totdfn;
    Node[totdfn] = u;
    sz[u] = 1;
    for (int v : g[u])
        if (v != p) {
            dfs0(v, u);
            sz[u] += sz[v];
            if (!big[u] || sz[big[u]] < sz[v]) big[u] = v;
        }
    R[u] = totdfn;
}

void dfs1(int u, int p, bool keep) {
    for (int v : g[u])  // 计算轻儿子的答案
        if (v != p && v != big[u])
            dfs1(v, u, false);
    if (big[u])  // 计算重儿子答案并保留计算过程中的数据（用于继承）
        dfs1(big[u], u, true);
    for (int v : g[u])
        if (v != p && v != big[u])
            for (int i = L[v]; i <= R[v]; i++)  // 子树结点可以直接遍历
                add(Node[i]);
    add(u); ans[u] = getAns();
    if (keep == false)
        for (int i = L[u]; i <= R[u]; i++)
            del(Node[i]);
}

int main() {
    int n; scanf("%d", &n);
    for (int i = 1; i <= n; i++) scanf("%d", &col[i]);
    for (int i = 1; i < n; i++) {
        int u, v;
        scanf("%d%d", &u, &v);
        g[u].push_back(v);
        g[v].push_back(u);
    }
    dfs0(1, 0); dfs1(1, 0, false);
    for (int i = 1; i <= n; i++) printf("%d%c", ans[i], " \n"[i == n]);
    return 0;
}
```

分块 2024/07/26

把 n 个元素分成 $\sqrt{n}$ 块，时间复杂度为 $O(n\sqrt{n})$.

变量定义：

```
int n; // 元素个数
int id[N], len; // id[i] 表示第 i 个数据的编号，len 表示块长

// L(x) 表示第 x 个块的左端点，R(x) 表示第 x 个块的右端点
inline int L(int x) {
    return len * (x - 1) + 1;
}

inline int R(int x) {
    return min(len * x, n) ;
}
```

预处理：

```
len = sqrt(n) ;
// 每一块的区间 [1,len],[len+1,2len],...,[xlen+1,n]
for (int i = 1; i <= n; i ++) {
    id[i] = (i - 1) / len + 1;
}
```

修改与查询：

```
// modify 和 query 函数
void function(int l, int r) {
    // [l,r] 在同一个块内
    if (id[l] == id[r]) {
        for (int i = l; i <= r; i++) {
            // 暴力处理
        }
        return;
    }
    // 左端不完整的块
    for (int i = l; id[i] == id[l]; i++) {
        // 暴力处理
    }
    // 中间完整的块
    for (int i = id[l] + 1; i < id[r]; i++) {
        //
    }
    // 右端不完整的块
    for (int i = r; id[i] == id[r]; i--) {
        // 暴力处理
    }
}
```