数据结构-树上问题

ZeitHaum

2023 年 4 月 16 日

目录

1 基本概念

1. 无根树: 没有固定根节点的树。
2. 有根树: 指定固定根节点的无根树。
3. 深度: 结点到根节点的边数，记作 $h(v)$。
4. 高度: 一个树中所有结点的深度的最大值，记作 $h(T)$。
5. 完整二叉树: 每个结点的儿子数量要么为 0 要么为 1。
6. 完美二叉树: 所有叶节点深度均相同的完整二叉树。
7. 完全二叉树: 除了右下连续部分叶节点深度为树的高度减 1 外，其余叶节点深度均和树的高度相同的二叉树。
8. 左孩子右兄弟: 两个数组分别记录每个结点的最左儿子和右兄弟。
9. 最近公共祖先: 记为 $\mathrm{lca}(u, v)$。

2 基础例题

热身运动开始。

2.1 例 1

力扣-94. 二叉树的中序遍历 前序遍历、后序遍历类似，不过需要调换添加答案的时机。

```
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode() : val(0), left(nullptr), right(nullptr) {}
 *     TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
 *     TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left), right(right) {}
```

```
 * };
 */
class Solution {
public:
    vector<int>ans;

    void dfs(TreeNode* root){
        if(root==nullptr) return;
        dfs(root->left);
        ans.push_back(root->val);
        dfs(root->right);
    }

    vector<int> inorderTraversal(TreeNode* root) {
        ans.clear();
        dfs(root);
        return ans;
    }
};
```

2.2 例 2

力扣-100. 相同的树 同步 DFS 遍历步骤并遇到异常及时处理即可。

```
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode() : val(0), left(nullptr), right(nullptr) {}
 *     TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
 *     TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left), right(right) {}
 * };
 */
class Solution {
public:
```

```cpp
    bool check = true;
    void dfs_same(TreeNode* r1,TreeNode* r2){
        if(check==false) return;
        if(r1==nullptr || r2==nullptr){
            if(!(r1 == nullptr && r2==nullptr)) check = false;
            return;
        }
        if(r1->val!=r2->val) check = false;
        dfs_same(r1->left,r2->left);
        dfs_same(r1->right,r2->right);
    }

    bool isSameTree(TreeNode* p, TreeNode* q) {
        check = true;
        dfs_same(p,q);
        return check;
    }
};
```

2.3 例 3

力扣-101. 对称二叉树 类似与例 2，不过同步遍历时考虑对称性即可。

```cpp
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode() : val(0), left(nullptr), right(nullptr) {}
 *     TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
 *     TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left), right(right) {}
 * };
 */
class Solution {
public:
    bool ck = true;
```

```
    void dfs_symmetric(TreeNode* r1,TreeNode* r2){
        if(r1==nullptr || r2==nullptr){
            if(!(r1==nullptr && r2 == nullptr)) ck = false;
            return;
        }
        if(r1->val!=r2->val) ck = false;
        if(ck==false) return;
        dfs_symmetric(r1->left,r2->right);
        dfs_symmetric(r1->right,r2->left);
    }

    bool isSymmetric(TreeNode* root) {
        ck = true;
        if(root==nullptr) return ck;
        dfs_symmetric(root->left,root->right);
        return ck;
    }
};
```

2.4 例 4

力扣-105. 从前序与中序遍历序列构造二叉树 分治和递归思想的应用。

```
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode() : val(0), left(nullptr), right(nullptr) {}
 *     TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
 *     TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left), right(right) {}
 * };
 */
class Solution {
public:
    int n;
```

```
    map<int,int>f;

    TreeNode* buildTree(vector<int>& preorder, vector<int>&
        inorder) {
        function<void(int,int,int,int,TreeNode*)> dfs_build =
            [&](int pbegin,int pend,int ibegin,int iend,
            TreeNode* root){
            //左闭右开
            int i_ind = f[root->val];
            int left_len = i_ind-1 - ibegin + 1;
            int right_len = iend-1 - (i_ind+1) + 1;
            if(left_len!=0){
                int root_val = preorder[pbegin+1];
                TreeNode* left = new TreeNode(root_val);
                root->left = left;
                dfs_build(pbegin+1,pbegin+1+left_len,ibegin,
                    i_ind,left);
            }
            if(right_len!=0){
                int root_val = preorder[pbegin+1+left_len];
                TreeNode* right = new TreeNode(root_val);
                root->right = right;
                dfs_build(pbegin+1+left_len,pend,i_ind+1,iend,
                    right);
            }
        };
        n = preorder.size();
        f.clear();
        map<int,int>g;
        for(int i = 0;i<n;i++){
            g[inorder[i]] = i;
        }
        for(int i = 0;i<n;i++){
            f[preorder[i]] = g[preorder[i]];
        }
        TreeNode* root = new TreeNode(preorder[0]);
        dfs_build(0,n,0,n,root);
        return root;
    }
```

```
};
```

2.5 例 5

力扣-106. 从中序与后序遍历序列构造二叉树 与上一题类似，注意递归条件的更改 (前序遍历和后序遍历的区别)。

```
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode() : val(0), left(nullptr), right(nullptr) {}
 *     TreeNode(int x) : val(x), left(nullptr), right(nullptr)
    {}
 *     TreeNode(int x, TreeNode *left, TreeNode *right) : val(x
    ), left(left), right(right) {}
 * };
 */
class Solution {
public:
    int n;
    map<int,int>f;
    TreeNode* buildTree(vector<int>& inorder, vector<int>&
        postorder) {
        //declearation of dfs.
        function<void(int,int,int,int,TreeNode*)> dfs_build =
            [&](int ibegin,int iend,int pbegin,int pend,
            TreeNode* root){
            int i_ind = f[root->val];
            int left_len = i_ind-1 - ibegin + 1;
            int right_len = iend-1 - (i_ind+1) + 1;
            if(left_len!=0){
                TreeNode* left = new TreeNode(postorder[pbegin+
                    left_len-1]);
                root->left = left;
                dfs_build(ibegin,i_ind,pbegin,pbegin+left_len,
                    left);
```

```
            }
            if(right_len!=0){
                TreeNode* right = new TreeNode(postorder[pend
                    -2]);
                root->right = right;
                dfs_build(i_ind+1, iend,pbegin+left_len,pend-1,
                    right);
            }
        };
        //initialize of attributes.
        n = inorder.size();
        f.clear();
        map<int,int>g;
        for(int i = 0;i<n;i++){
            g[inorder[i]] = i;
        }
        for(int i = 0;i<n;i++){
            f[postorder[i]] = g[postorder[i]];
        }
        //calc ans.
        TreeNode* root = new TreeNode(postorder[n-1]);
        dfs_build(0,n,0,n,root);
        return root;
    }
};
```

3 *Morris 遍历

通过修改原始数据结构实现的空间 $O(1)$ 的算法。相比 DFS 和 BFS 空间 $O(n)$ 的算法空间效率很高，但是由于会修改原始数据结构不常用。

4 树的直径

定义为树 T 的最长路径，记为 $D(T)$。

4.1 算法一:DP

设 T_l, T_r 为左子树和右子树。考虑根节点对答案的贡献，可以得到转移方程

$$D(T) = \max(D(T_l), D(T_r), h(T_l) + h(T_r)) \tag{1}$$

时间复杂度 $O(n)$，如果还要得到一条直径的两个端点或者路径，还需要记录取最大时的信息。

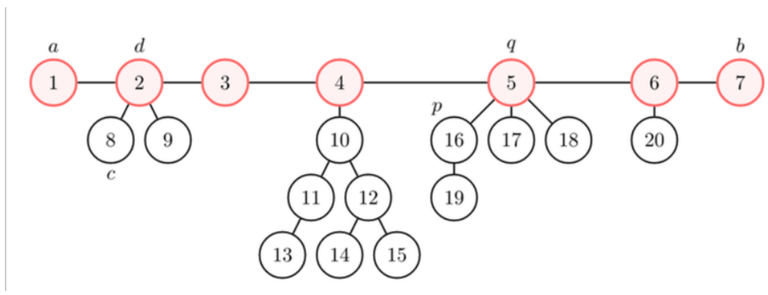
4.2 算法二: 遍历

记 $\mathrm{dis}(u, v)$ 为结点 u 到结点 v 的距离。

引理 1 *对于树上任意一个结点 x，记 $a = \max_i(dis(x, i))$，则 a 必为直径端点之一。*

证明:

首先一棵树可以视为直径 + 森林，如下图 (图片仅作示例):



而森林的每个顶点均在直径上。设 x 所在部分的顶点为 r_1。

(利用反证法) 假设 a 不为直径端点，s, t 为一条直径的两个端点。

不失一般性地可设 $\mathrm{dis}(x, s) \le \mathrm{dis}(x, t)$。

则有

$$\mathrm{dis}(x, a) > \mathrm{dis}(x, t) \Rightarrow \mathrm{dis}(r_1, a) > \mathrm{dis}(r_1, t) \Rightarrow \mathrm{dis}(s, a) > \mathrm{dis}(s, t). \tag{2}$$

这与假设相悖，于是原命题得证。

同样根据定义可证以下引理:

引理 2 *对于树上一个直径的端点 a，记 $b = \max_i(dis(a, i))$，则 $a \to b$ 的路径必为直径。*

于是只需按以上两个引理操作，分别以任意结点和第一次的结果进行两次树的遍历 (推荐 BFS 遍历)，每次寻找到距离根节点最远的结点，第二次的结果即是答案。

4.3 适用性

当题目所给为典型有根树状结构时，适合用算法一。如果是无根树 (图论结构)，适合用算法二。

4.4 例 6

力扣-543. 二叉树的直径

4.4.1 Solution

典型例题，给的数据结构为树，适用于算法一。

4.4.2 Code

```
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode() : val(0), left(nullptr), right(nullptr) {}
 *     TreeNode(int x) : val(x), left(nullptr), right(nullptr)
     {}
 *     TreeNode(int x, TreeNode *left, TreeNode *right) : val(x
     ), left(left), right(right) {}
 * };
 */
 //DP version
class Solution {
public:
    int ans = 0;
    int dfs_dp(TreeNode* root){
        if(root==nullptr) return 0;
```

```
        int h_l = dfs_dp(root->left);
        int h_r = dfs_dp(root->right);
        ans = max(ans,h_l+h_r);
        return max(h_l,h_r)+1;
    }

    int diameterOfBinaryTree(TreeNode* root) {
        dfs_dp(root);
        return ans;
    }
};
```

4.5 例 7

CodeForces-Round 862(Div2)-D

4.5.1 Solution

此题较为灵活，分析可得显然直径的端点是首次被合入的对象。设 T_i 表示以 i 为根时的树。

显然第 k 轮将会合并满足 $h(T_i) = k$ 的点，即连通区域减少 $\text{count}(h(T_i) = k)$(第一次合并减少 $\text{count}(h(T_i) = k) - 1$)。

根据引理1,$h(T_i) = k$ 等价于距离直径一端点距离较大值为 k。

也即 $\max(\text{dis}(i, s), \text{dis}(i, t)) = k$。

于是我们可以先找到一组直径的端点 s, t，即可计算 $h(T_i)$。

根据所给条件适宜使用算法二。

4.5.2 Code

```
#include <bits/stdc++.h>
using namespace std;

ostream& operator<<(ostream& out, vector<int>& arr){
    for(int i = 0;i<arr.size();i++){
        out<<arr[i]<<" ";
    }
    return out;
```

```
}

int main(){
    int n;
    cin>>n;
    vector adj(n+1,vector<int>(0));
    int u,v;
    for(int i = 0;i<n-1;i++){
        cin>>u>>v;
        adj[u].push_back(v);
        adj[v].push_back(u);
    }
    vector dis_1(n+1,-1);
    vector dis_2(n+1,-1);
    auto bfs = [&](vector<int>& height,int root){
        queue<int> q;
        q.push(root);
        height[root] = 0;
        vector<bool>vis(n+1,false);
        while(q.size()!=0){
            int qf = q.front();
            q.pop();
            vis[qf] = true;
            for(int i  = 0;i<adj[qf].size();i++){
                int now = adj[qf][i];
                if(!vis[now]){
                    q.push(now);
                    height[now] = height[qf]+1;
                    vis[now] = true;
                }
            }
        }
    };
    bfs(dis_2,1);
    int max_root = max_element(dis_2.begin(),dis_2.end()) - dis_2.begin();
    bfs(dis_1,max_root);
    max_root = max_element(dis_1.begin(),dis_1.end()) - dis_1.begin();
```

```
    bfs(dis_2,max_root);
    for(int i = 0;i<dis_1.size();i++){
        dis_1[i] = max(dis_1[i],dis_2[i]);
    }
    sort(dis_1.begin(),dis_1.end());
    vector ans(n+1,0);
    for(int i =1;i<dis_1.size();i++){
        ans[dis_1[i]]++;
    }
    cerr<<ans<<endl;
    ans[n] = n;
    bool enable = false;
    for(int i = ans.size()-2;i>=0;i--){
        int temp = 0;
        if(ans[i]!=0 && !enable){
            temp = 1;
            enable = true;
        }
        ans[i] = ans[i+1] - ans[i] + temp;
    }
    for(int i = 1;i<ans.size();i++){
        cout<<ans[i]<<" ";
    }
    cout<<endl;
}
```

4.6 例 8

洛谷-SDOI2013-直径 这道题是上述引理的综合应用。

第一问略。第二问需要求取所有直径都经过的边数，为此我们可以先将一条直径求取出来。

(利用树 = 直径 + 森林的做法) 记 s, t 为一条直径的两个端点，r 是直径上一点，T_r 是去除直径后以 r 为根节点的深度。

显然，当 $\exists x$ 使得在树 T_r 中有 $h(x) = \text{dis}(x, s)$ 时，路径 $s \to r$ 便不是所有直径都经过的边的一部分，对于端点 t 此结论也成立。

引理 3 对于上述描述的 x, 若存在，必有

$$h(x) = h(T_r) = \min(dis(r,s), dis(r,t)). \tag{3}$$

证明: 使用反证法，证明如果不满足上述情况必有假设 “s, t 为直径的两个端点” 断言为假即可。

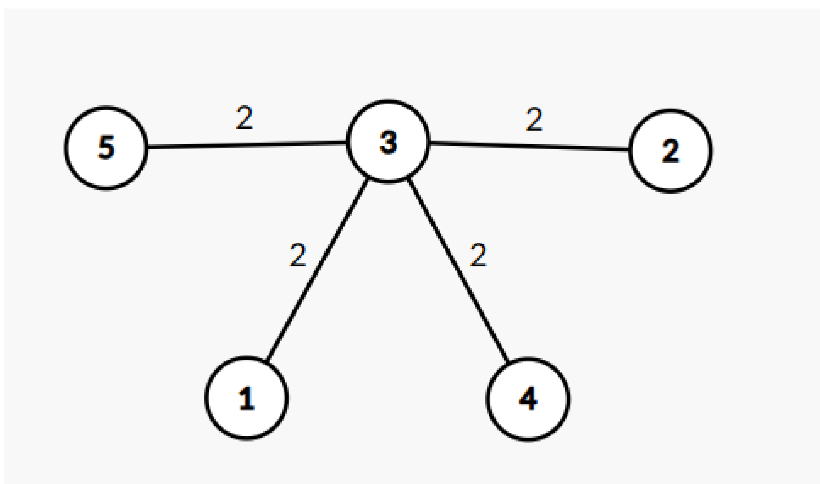
记

$$f(v) = \max_{\mathrm{dis}(u,v)=h(T_u)} (\mathrm{dis}(u,v)), v \in \{s,t\}. \tag{4}$$

于是

$$\mathrm{ans} = \mathrm{abs}(f(s) - f(t)). \tag{5}$$

上述算法在下图将会遇到特例，需要进行特别判定:



4.6.1 Code

```
#include<bits/stdc++.h>
using namespace std;
#define int long long
#define mp(x,y) make_pair((x),(y))

ostream& operator<<(ostream& out,const vector<int>& arr){
    for(int i = 0;i<arr.size();i++){
        out<<arr[i]<<" ";
    }
```

```
    return out;
}
ostream& operator<<(ostream& out,const vector<bool>& arr){
    for(int i = 0;i<arr.size();i++){
        out<<arr[i]<<" ";
    }
    return out;
}
ostream& operator<<(ostream& out,const pair<int,int>& p){
    out<<p.first<<","<<p.second<<" ";
    return out;
}
ostream& operator<<(ostream& out,const map<pair<int,int>,int>& dic){
    for(auto iter = dic.begin();iter!=dic.end();iter++){
        out<<iter->first<<":"<<iter->second<<" ";
    }
    return out;
}

signed main(){
    int n;
    cin>>n;
    //handling the input.
    const int ARR_SIZE = n+1;
    vector adj(ARR_SIZE,vector<int>(0));
    map<pair<int,int>,int>weight;
    int u,v,w;
    for(int i = 0;i<n-1;i++){
        cin>>u>>v>>w;
        adj[u].push_back(v);
        adj[v].push_back(u);
        weight[mp(u,v)] = w;
        weight[mp(v,u)] = w;
    }
    //Find a diameter point.
    vector<int>h(ARR_SIZE,0);
    vector<bool>vis(ARR_SIZE,false);
    auto clear_vis = [&]{
```

```
        for(int i = 0;i<vis.size();i++){
            vis[i] = false;
            h[i] = 0;
        }
    };
    auto clear_h = [&]{
        for(int i = 0;i<vis.size();i++){
            h[i] = 0;
        }
    };
    auto clear_vis_h = [&]{
        clear_vis();
        clear_h();
    };
    auto bfs_find = [&](vector<int>& h,int root){
        queue<int> q;
        q.push(root);
        vis[root] = true;
        while(q.size()>0){
            int qf = q.front();
            q.pop();
            for(int i = 0;i<adj[qf].size();i++){
                int now = adj[qf][i];
                if(vis[now]==false){
                    q.push(now);
                    vis[now] = true;
                    h[now] = h[qf] + weight[mp(qf,now)];
                }
            }
        }
        return max_element(h.begin(),h.end());
    };
    clear_vis_h();
    int da = bfs_find(h,1) - h.begin();
    clear_vis_h();
    int db = bfs_find(h,da) - h.begin();
    int d_len = h[db];
    //Find a diameter path.
    vector<int> diameter_path(0);
```

```
    for(int i = 0;i<vis.size();i++){
        vis[i] = false;
    }
    bool enable_path = false;
    function<void(int,int)> dfs_path = [&](int root,int
        now_weight){
        diameter_path.push_back(root);
        vis[root] = true;
        for(int i = 0;i<adj[root].size();i++){
            int now = adj[root][i];
            if(vis[now]==false){
                if(now_weight+ weight[mp(root,now)]==d_len &&
                    adj[now].size()==1){
                    diameter_path.push_back(now);
                    enable_path = true;
                    return;
                }
                dfs_path(now,now_weight+weight[mp(root,now)]);
                if(enable_path) return;
                diameter_path.pop_back();
            }
        }
    };
    dfs_path(da,0);
    int d_size = diameter_path.size();
    //Check connectivity.
    for(int i = 0;i<vis.size();i++){
        vis[i] = false;
    }
    for(int i =0;i<d_size;i++){
        vis[diameter_path[i]] = true;
    }
    int temp_h;
    vector<int> t_r(d_size,0);
    function<void(int,int)> dfs_check = [&](int root,int j){
        vis[root] = true;
        for(int i = 0;i<adj[root].size();i++){
            int now = adj[root][i];
            if(vis[now]==false){
```

```
                temp_h += weight[mp(root,now)];
                //Leaf Node;
                if(adj[now].size()==1){
                    t_r[j] = max(t_r[j],temp_h);
                }
                dfs_check(now,j);
                temp_h -= weight[mp(root,now)];
            }
        }
    };
    for(int i = 0;i<d_size;i++){
        temp_h = 0;
        dfs_check(diameter_path[i],i);
    }
    int fs = max_element(t_r.begin(),t_r.begin()+(d_size-1)
        /2+1) - t_r.begin();
    int ft = max_element(t_r.begin()+(d_size-1)/2+1,t_r.end())
        - t_r.begin();
    int ans2 = abs(fs - ft);
    //Special Judge
    int accu_w = 0;
    for(int i = 1;i<d_size;i++){
        accu_w += weight[mp(diameter_path[i-1],diameter_path[i
            ])];
        if(d_len - accu_w==accu_w){
            if(t_r[i]==accu_w) ans2 = 0;
            break;
        }
    }
    cout<<d_len<<endl;
    cout<<ans2<<endl;
}
/*
5
1 3 2
2 3 2
4 3 2
5 3 2

```

```
9
1 2 1
2 5 3
3 4 2
4 5 1
5 6 1
6 7 3
5 8 2
8 9 2
*/
```

5 最近公共祖先问题

两个结点的最近公共祖先为公共结点中深度最低的结点，记为 $\mathrm{lca}(u,v)$。性质：

1. $\mathrm{lca}(u,v)=u$，当且仅当 u 是 v 的祖先。
2. 对于点集 A,B，$\mathrm{lca}(A\cup B)=\mathrm{lca}(\mathrm{lca}(A),\mathrm{lca}(B))$.
3. $\mathrm{dis}(u,v)=h(u)+h(v)-2*h(\mathrm{lca}(u,v))$.

5.1 朴素算法

记 $\mathrm{fa}(x)$ 为结点 x 的根节点，根据性质 1，可以推出转移方程式：

$$\mathrm{lca}(u,v)=\begin{cases} u, & u=v \\ \mathrm{lca}(\mathrm{fa}(u),v), & h(u)>h(v) \\ \mathrm{lca}(u,\mathrm{fa}(v)), & h(u)<h(v) \\ \mathrm{lca}(\mathrm{fa}(u),\mathrm{fa}(v)), & h(u)=h(v) \end{cases} \tag{6}$$

预处理复杂度 $O(n)$，单次查询复杂度为 $O(h(T))$.

5.1.1 例 9

力扣-236. 二叉树的最近公共祖先

Code

```
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode(int x) : val(x), left(NULL), right(NULL) {}
 * };
 */
class Solution {
public:
    TreeNode* lowestCommonAncestor(TreeNode* root, TreeNode* p,
         TreeNode* q) {
        map<int,TreeNode*> fa;
        map<int,int> h;
        auto bfs = [&](){
            queue<TreeNode*> q;
            q.push(root);
            fa[root->val] = NULL;
            h[root->val] = 0;
            while(q.size()!=0){
                TreeNode* qf = q.front();
                q.pop();
                if(qf->left!=NULL){
                    fa[qf->left->val] = qf;
                    h[qf->left->val] = h[qf->val]+1;
                    q.push(qf->left);
                }
                if(qf->right!=NULL){
                    fa[qf->right->val] = qf;
                    h[qf->right->val] = h[qf->val]+1;
                    q.push(qf->right);
                }
            }
        };
        bfs();
        function<TreeNode*(TreeNode*,TreeNode*)> find_lca =
            [&](TreeNode*p ,TreeNode* q){
```

```
            if(p->val==q->val) return p;
            if(h[p->val]>h[q->val]) return find_lca(fa[p->val],q);
            else if(h[p->val]<h[q->val]) return find_lca(p,fa[q->val]);
            else return find_lca(fa[p->val],fa[q->val]);
        };
        return find_lca(p,q);
    }
};
```

5.2 倍增算法

5.3 子问题一: 二进制转换

首先先思考一个子问题: 对于一个整数 x, 如何将其转换为二进制?

对于这个问题，朴素的做法是依次除 2，即利用表达式

$$\text{binary\_str}(x) = \begin{cases} \text{binary\_str}(x/2) + \text{`0'}, & x\%2 = 0 \\ \text{binary\_str}(x/2) + \text{`1'}, & x\%2 = 1 \end{cases} \tag{7}$$

实践中，为了避免从字符串首部插入导致复杂度提升，常常先将字符插入至尾部，最后再进行翻转。尽管如此，单次求取 $x/2$ 常数还是比较大 (较比于加减法和乘法)。

如果我们预处理数列 $\{2^0, 2^1, ..., 2^k\}, k \leq \lfloor \log_2(n) \rfloor$ 的值，于是可有

$$\text{binary\_str}(x) = \begin{cases} \text{`0'} + \text{binary\_str}(x), & x < 2^k \\ \text{`1'} + \text{binary\_str}(x - 2^k), & x \geq 2^k \end{cases} \tag{8}$$

依次减少 k 的值，便可以求出二进制每位的值。

例 10: 转换二进制

```
#include <bits/stdc++.h>
using namespace std;

int pow2[31];

string binary_str1(int x){
```

```
    string rans;//初始设置ans为答案的翻转串
    while(true){
        if(x%2==0) rans.push_back('0');
        else rans.push_back('1');
        x = x/2;
        if(x==0) break;
    }
    reverse(rans.begin(),rans.end());
    return rans;
}

string binary_str2(int x){
    string ans;
    for(int i = 30;i>=0;i--){
        if(x>= pow2[i]){
            x -= pow2[i];
            ans.push_back('1');
        }
        else ans.push_back('0');
    }
    //如果要求去除前导0...
    int first_1 = 0;
    for(int i = 1;i<=31;i++){
        if(ans[i]=='1'){
            first_1 = i;
            break;
        }
    }
    return ans.substr(first_1,31 - first_1+1);
}

int main(){
    //预处理pow2
    for(int i = 0;i<31;i++){
        if(i==0) pow2[i] = 1;
        else pow2[i] = pow2[i-1] * 2;
    }
    int x = 13;
    cout<<binary_str1(x)<<endl<<binary_str2(x)<<endl;
```

```
}
```

5.4 思路

上述子问题给出了一种可能性，对于具备二分性质的数列可以通过预处理 2^k 数列来降低多次询问复杂度。

现在我们考虑利用这种方法优化朴素最近公共祖先问题。

首先我们考虑当 $h(u) \neq h(v)$ 的情况，不失一般性性地假设 $h(u) > h(v)$，于是必存在 x 使得 $h(\mathrm{fa}^x(u)) = h(v)$，其中

$$\mathrm{fa}^x(u) = \begin{cases} \mathrm{fa}^{x-1}(u), & x > 1 \\ \mathrm{fa}(u), & x = 1 \\ u, & x = 0. \end{cases} \tag{9}$$

对于任意 k, 根据二进制分解的原理有

$$\mathrm{fa}^x(u) = \begin{cases} \mathrm{fa}^x(u), & h(\mathrm{fa}^{2^k}(u)) < h(u) \\ \mathrm{fa}^{x-2^k}(\mathrm{fa}^{2^k}(u)), & h(\mathrm{fa}^{2^k}(u) \geq h(u)) \end{cases} \tag{10}$$

如果对于任意结点 u 我们提前预处理 $f^{2^k}(u)$ 的值，然后依次减小 k 的值即可。

单次查询复杂度: $O(\log(n))$, k 最大为 $\lfloor \log(n) \rfloor$。

现在我们考虑当 $h(u) = h(v)$ 时求最近公共祖先的操作，同理，必存在最小的 y，使得 $\mathrm{fa}^y(u) = \mathrm{fa}^y(v)$.

同理，根据二进制分解的原理, 对于 k 有

$$\mathrm{lca}(u, v) = \begin{cases} \mathrm{lca}(u, v), & \mathrm{fa}^{2^k}(u) = \mathrm{fa}^{2^k}(v) \\ \mathrm{lca}(\mathrm{fa}^{2^k}(u), \mathrm{fa}^{2^k}(v)), & \mathrm{fa}^{2^k}(u) \neq \mathrm{fa}^{2^k}(v) \end{cases} \tag{11}$$

最后需要考虑 $\mathrm{fa}^{2^k}(u)$ 的维护:

$$\mathrm{fa}^{2^k}(u) = \begin{cases} \mathrm{fa}(u), & k = 0 \\ \mathrm{fa}^{2^{k-1}}(\mathrm{fa}^{2^{k-1}}(u)), & k \geq 1. \end{cases} \tag{12}$$

注意需要考虑对非法值，即 $2^k > h(u)$ 时的处理，一般考虑赋为定义域之外的数。

5.5 例 11

P3379 【模板】最近公共祖先（LCA） 模板题，按照模板写即可。

```
#include <bits/stdc++.h>
using namespace std;

const int MAX_SIZE = 5e5+1;
int fa[MAX_SIZE][21];
int h[MAX_SIZE];

int main(){
    ios::sync_with_stdio(false);
    cin.tie(0);
    int n,m,root;
    cin>>n>>m>>root;
    vector adj(n+1,vector<int>(0));
    int u,v;
    for(int i = 0;i<n-1;i++){
        cin>>u>>v;
        adj[u].push_back(v);
        adj[v].push_back(u);
    }
    auto dfs = [&](auto self,int root,int depth)->void{
        for(int v: adj[root]){
            if(v==fa[root][0]) continue;
            fa[v][0] = root;
            h[v] = depth+1;
            self(self,v,depth+1);
        }
    };
    dfs(dfs,root,0);
    h[0] = -1;
    for(int i = 1;i<21;i++){
        for(int j = 1;j<=n;j++){
            fa[j][i] = fa[fa[j][i-1]][i-1];
        }
    }
    auto lca = [&](int u,int v){
        if(h[u]<h[v]) swap(u,v);
```

```
        if(h[u]!=h[v]){
            for(int i = 20;i>=0;i--){
                if(h[fa[u][i]]>=h[v]) u = fa[u][i];
            }
        }
        if(u==v) return u;
        else{
            for(int i = 20;i>=0;i--){
                if(fa[u][i] != fa[v][i]){
                    u = fa[u][i];
                    v = fa[v][i];
                }
            }
            return fa[u][0];
        }
    };
    for(int i = 0;i<m;i++){
        cin>>u>>v;
        cout<<lca(u,v)<<endl;
    }
}
```

6 线段树

6.1 概述

定义: 将区间信息存储在结点的一种数据结构。其支持的操作为

1. 区间修改，将区间 $[L,R]$ 修改为新值。
2. 区间查询，查询区间 $[L,R]$ 的信息。

6.2 操作需要满足的条件

线段树基于分治的原理。其维护的信息 (统称操作) 都需要满足以下条件:

1. 操作必须是二元的。

2. 操作必须满足结合律。

3. 操作必须有幺元。

几个满足线段树操作的例子: 加减法、最大值、最小值、GCD、LCM 等。

6.3 结构

常用的线段树满足以下性质:

1. 线段树是一颗完整二叉树。

2. 线段树的根节点编号为 1，维护区间 $[1, n]$ 的信息。

3. 对于区间 $[1, n]$, 线段树的叶子结点从左往右依次维护 $[1, 1], [2, 2]..., [n-1, n-1], [n, n]$ 的信息。

4. 对于线段树的每个中间结点，设该节点维护区间 $[x, y]$ 的信息，则其左儿子维护区间 $[x, \lfloor\frac{x+y}{2}\rfloor]$ 的信息，右儿子维护区间 $[\lfloor\frac{x+y}{2}\rfloor + 1, y]$ 的信息。

5. 线段树采取堆式存储，也即对于编号 x, 其左儿子编号为 $2x$, 右儿子编号为 $2x + 1$.

6.4 性质

注意: 这里的 n 指的是区间长度而不是常见的树的结点个数。

引理 4 线段树不存在两个结点不同但编号一样。

证明: 假设存在两个结点编号均为 x, 所以其均为 $\lfloor x/2 \rfloor$ 的子节点，且根据奇偶性可以判定其属于左儿子还是右儿子。

因此这两个结点是同一个结点，QED。

引理 5 线段树的结点个数小于等于 $2n - 1$。

证明:

观察可得递推式

$$f(n) = \begin{cases} 2f(n/2) + 1, & \text{n 是奇数}, \\ f((n-1)/2) + f((n+1)/2) + 1, & \text{n 是偶数}. \end{cases}$$

使用归纳法证明。当 $f(1)=1, f(2)=3$ 均成立。

考虑 $f(n)$,

当 n 为偶数时，$f(n) \le 2 \times (n/2 \times 2 - 1) + 1 \Leftrightarrow f(n) \le 2n-1$.

当 n 为奇数时，$f(n) \le (n-1)/2*2-1+(n+1)/2*2-1+1 \Leftrightarrow f(n) \le 2n-1$..

均成立，QED.

引理 6 线段树编号最大值小于 $4n$。

证明:

线段树深度最大不超过 $2^{\lceil \log(n) \rceil}$，因此最大编号不超过

$$1+2+4+....+2^{\lceil \log(n) \rceil}$$

即为 $2^{\lceil \log(n) \rceil+1} < 2^{\log(n)+2} = 4n$.

引理 7 线段树的高度不超过 $O(\log(n))$.

证明: 由性质5和完全二叉树的性质可证。-

6.5 PURQ 数据结构

PURQ ——”Point update range query”, 也即单点修改，区间查询。

首先要构建线段树，开 4 倍空间采取递归建树即可。

其次是区间查询,

设 e 为查询信息的幺元，$\circ$ 为查询信息合并的操作。

要查询的区间为 $[L,R]$, 当前所在叶节点维护的区间为 $[l,r]$, 信息为 $\inf(l,r)$.

有以下递推式:

$$Q(L,R,l,r) = \begin{cases} \inf(l,r), & [l,r] \subseteq [L,R] \\ e, & [l,r] \cap [L,R] = \emptyset \\ Q(L,R,l,(l+r)/2) \circ Q(L,R,(l+r)/2+1,r), & \text{else} \end{cases}$$

以上三种情况的条件也称为完全覆盖、不相交、部分覆盖。

单点更新和区间查询类似，见源代码。

6.5.1 例 13

模板题，需要注意区间关系的判断条件。力扣-307. 区域和检索 - 数组可修改

```
class NumArray {
public:
    vector<int>s;
    int n;

    NumArray(vector<int>& nums) {
        n = nums.size();
        s.resize(n*4+1);
        nums.insert(nums.begin(),0);
        auto build = [&](auto self,int l,int r,int p)->void{
            if(l==r){
                s[p] = nums[l];
                return;
            }
            int mid = (l+r)/2;
            self(self,l,mid,p*2);
            self(self,mid+1,r,p*2+1);
            s[p] = s[p*2] + s[p*2+1];
        };
        build(build,1,n,1);
    }

    void update(int index, int val) {
        index = index + 1;
        auto up = [&](auto self,int l,int r,int p)->void{
            if(l<=index && r>=index){
                if(l==r){
                    s[p] = val;
                    return;
                }
                int mid = (l+r)/2;
                self(self,l,mid,p*2);
                self(self,mid+1,r,p*2+1);
                s[p] = s[p*2] + s[p*2+1];
                return;
```

```
            }
            else{
                return;
            }
        };
        up(up,1,n,1);
    }

    int sumRange(int left, int right) {
        left = left + 1;
        right = right + 1;
        auto query = [&](auto self, int l,int r,int p)->int{
            if(left<=l && right>= r){
                return s[p];
            }
            else if(l>right || r<left){
                return 0;
            }
            else{
                int mid = (l+r)/2;
                return self(self,l,mid,p*2)+self(self,mid+1,r,p
                    *2+1);
            }
        };
        return query(query,1,n,1);
    }
};

/**
 * Your NumArray object will be instantiated and called as such
    :
 * NumArray* obj = new NumArray(nums);
 * obj->update(index,val);
 * int param_2 = obj->sumRange(left,right);
 */
```

6.6 RURQ 数据结构

RURQ ——"Range update range query" 数据结构，即区间修改，区间查询。

思考区间修改的问题，如果每次都去修改区间内的数据将会复杂度难以承受。

而如果只保存此次操作，改变对应结点的信息，便可以大大降低复杂度。

懒惰标记 (lazy propagation) 是优化区间更新的方法，其思想便是存储每次操作的结果，数组的更新只在进入到相应区间时发生。

懒惰标记维护的操作集合 F 需要满足以下条件，

1. F 含有单位变换 e，使得 $e(x) = x$.
2. F 中的元素是可复合的，也即对于 $f, g \in F$, 必有 $f \circ g \in F$.
3. 对于维护信息的操作 $\cdot$，F 中的元素是可分配的。也即对于 $f \in F$, $f(x \cdot y) = f(x) \cdot f(y)$。

具体编程实现见例题。

6.6.1 例 14

洛谷 - P3372 【模板】线段树 1

需要维护的操作为区间求和，记 $\text{lazy}(i)$ 表示线段树结点 i 未对子树更新部分。

一旦查询或更新到相应区间，$\text{lazy}(i)$ 存储的更新应该同步向下传递，而本结点的标记清空，并且递归结束后需要更新该区间维护的信息。

代码:

```
#include <bits/stdc++.h>
using namespace std;
#define MAX_N 100001
#define int long long

struct S{
    int val;
    int len;
    S():val(0),len(0){}
```

```
    S(int val,int len):val(val),len(len){}
    S operator+(const S& s){
        return {this->val+s.val,this->len+s.len};
    }
};
int da[MAX_N];
S s[4*MAX_N];
const S se = {0,0};
int lazy[4*MAX_N];
struct seg_tree{
    //将lazy 标记向下传递
    void push_down(int p){
        lazy[p*2] += lazy[p];
        lazy[p*2+1] += lazy[p];
        s[p*2].val += lazy[p]*s[p*2].len;
        s[p*2+1].val += lazy[p]*s[p*2+1].len;
        lazy[p] = 0;
    }
    //build
    void build(int l,int r,int p){
        if(l==r){
            s[p] = {da[l],1};
            return;
        }
        int mid = (l+r)/2;
        build(l,mid,p*2);
        build(mid+1,r,p*2+1);
        s[p] = s[p*2] + s[p*2+1];
    }
    //query
    S query(int l,int r,int p,int L,int R){
        if(L<=l && R>=r) return s[p];
        else if(l>R || r<L) return se;
        else{
            int mid = (l+r)/2;
            push_down(p);
            return query(l,mid,p*2,L,R) + query(mid+1,r,p*2+1,L
                ,R);
        }
```

```
    }
    //update
    void update(int l ,int r,int p,int L,int R,int val){
        if(L<=l && R>=r) {
            lazy[p] += val;
            s[p].val += s[p].len*val;
        }
        else if(l>R || r<L) return;
        else{
            int mid = (l+r)/2;
            push_down(p);
            update(l,mid,p*2,L,R,val);
            update(mid+1,r,p*2+1,L,R,val);
            s[p] = s[p*2] + s[p*2+1];
        }
    }
};

signed main(){
    ios::sync_with_stdio(false);
    cin.tie(0);
    int n,m;
    cin>>n>>m;
    seg_tree st;
    for(int i = 1;i<=n;i++){
        cin>>da[i];
    }
    st.build(1,n,1);
    int type;
    int L,R,val;
    for(int i = 1;i<=m;i++){
        cin>>type;
        if(type==1){
            cin>>L>>R>>val;
            st.update(1,n,1,L,R,val);
        }
        else{
            cin>>L>>R;
            cout<<st.query(1,n,1,L,R).val<<endl;
```

```
        }
    }
}
```

6.7 例 15

洛谷 - P3373 【模板】线段树 2

与上述题目不同的是需要考虑操作的定义。

考虑操作 $f(a,b,x)=(x+a)b$(先加后乘), 则

$$\begin{aligned} f(c,d,f(a,b,x)) &= (f(a,b,x)+c)d \\ &= ((x+a)b+c)d \\ &= (x+a+c/b)bd \\ &= f(a+c/b,bd,x) \end{aligned}$$

若 $b=0$, 此时便不满足区间更新所需的第二条性质。

尽管我们可以通过约定此时 $c/b=0$ 使其满足该性质，然而对于 c/b 这部分的小数仍然是我们不希望得到的 (不易维护)。

因此我们考虑定义 $f(a,b,x)=ax+b$(先乘后加), 则

$$\begin{aligned} f(c,d,f(a,b,x)) &= f(c,d,ax+b) \\ &= acx+bc+d \\ &= f(ac,bc+d) \end{aligned}$$

此操作具备良好的性质，便于维护。

代码:

```
#include <bits/stdc++.h>
using namespace std;
#define int long long
#define MAX_SIZE 100001

int p;
struct S
{
    int val;
    int len;
```

```
    S():val(0),len(0){}
    S(int val,int len):val(val),len(len){}
    S merge(S s){ return {(this->val+s.val)%p,this->len+s.len
        };}
};

struct F
{
    int add;
    int mul;
    F():add(0),mul(1){}
    F(int add,int mul):add(add),mul(mul){}
    F composition(F f){ return {((this->add*f.mul)%p + f.add)%p
        ,(this->mul*f.mul)%p};}
};

S apply(F f, S s){
    return {((s.val * f.mul)%p + (s.len * f.add)%p)%p,s.len};
}

int da[MAX_SIZE];
S s[MAX_SIZE*4];
F lazy[MAX_SIZE*4];
S se;
F fe;

struct seg_tree
{
    void build(int l,int r,int p){
        if(l==r){
            s[p] = {da[l],1};
            return;
        }
        int mid = (l+r)/2;
        build(l,mid,p*2);
        build(mid+1,r,p*2+1);
        s[p] = s[p*2].merge(s[p*2+1]);
    }

```

```
    void push_down(int p){
        lazy[p*2] =  lazy[p*2].composition(lazy[p]);
        lazy[p*2+1] = lazy[p*2+1].composition(lazy[p]);
        s[p*2] = apply(lazy[p],s[p*2]);
        s[p*2+1] = apply(lazy[p],s[p*2+1]);
        lazy[p] = fe;
    }

    S query(int l,int r,int p,int L,int R){
        if(L<= l && R>=r) return s[p];
        else if(l>R || r<L) return se;
        else{
            int mid = (l+r)/2;
            push_down(p);
            return query(l,mid,p*2,L,R).merge(query(mid+1,r,p
                *2+1,L,R));
        }
    }

    void update(int l,int r,int p,int L,int R,int add,int mul){
        if(L<=l && R>=r){
            lazy[p] = lazy[p].composition({add,mul});
            s[p] = apply({add,mul},s[p]);
        }
        else if(l>R || r<L) return;
        else{
            int mid = (l+r)/2;
            push_down(p);
            update(l,mid,p*2,L,R,add,mul);
            update(mid+1,r,p*2+1,L,R,add,mul);
            s[p] = s[p*2].merge(s[p*2+1]);
        }
    }
};


signed main(){
    ios::sync_with_stdio(false);
    cin.tie(0);
```

```
    int n,m;
    cin>>n>>m>>p;
    for(int i = 1;i<=n;i++){
        cin>>da[i];
    }
    seg_tree st;
    st.build(1,n,1);
    int type,L,R,add,mul;
    for(int i = 1;i<=m;i++){
        cin>>type;
        if(type==1){
            cin>>L>>R>>mul;
            st.update(1,n,1,L,R,0,mul);
        }
        else if(type==2){
            cin>>L>>R>>add;
            st.update(1,n,1,L,R,add,1);
        }
        else{
            cin>>L>>R;
            cout<<st.query(1,n,1,L,R).val<<endl;
        }
    }
}
```

6.8 线段树求解 lca 问题

6.8.1 子问题:RMQ 问题

RMQ——”Range Minimum/Maximum Query”，区间最大/最小值查询。

无修改，直接用 RQ 数据结构维护即可 (甚至不需要 PU)。

6.8.2 例 16

板子题。Balanced Lineup

代码:

```
#include <iostream>
```

```
#include <algorithm>
#define MAX_SIZE 50001
#define N_INF -1000001
#define P_INF 1000001
using namespace std;

struct S
{
    int mi;
    int mx;
    S():mi(P_INF),mx(N_INF){}
    S(int mi,int mx):mi(mi),mx(mx){}
};

inline S merge(S s1, S s2){
    S ret;
    ret.mi = min(s1.mi,s2.mi);
    ret.mx = max(s1.mx,s2.mx);
    return ret;
}

int da[MAX_SIZE];
S s[4*MAX_SIZE];
S se;

struct seg_tree
{
    void build(int l,int r,int p){
        if(l==r){
            s[p].mi = da[l];
            s[p].mx = da[l];
            return;
        }
        int mid = l+r>>1;
        build(l,mid,p<<1);
        build(mid+1,r,p<<1 | 1);
        s[p] = merge(s[p<<1],s[p<<1 | 1]);
    }

```

```
    inline S query(int l,int r,int L,int R,int p){
        if(L<= l && R>=r) return s[p];
        else if(l>R || r<L) return se;
        else{
            int mid = l+r>>1;
            return merge(query(l,mid,L,R,p<<1), query(mid+1,r,L,R,p<<1|1));
        }
    }
};


int main(){
    ios::sync_with_stdio(false);
    cin.tie(0);
    int n,Q;
    cin>>n>>Q;
    for(int i = 1;i<=n;i++){
        cin>>da[i];
    }
    seg_tree st;
    st.build(1,n,1);
    int L,R;
    for(int i = 1;i<=Q;i++){
        cin>>L>>R;
        S ans = st.query(1,n,L,R,1);
        cout<<ans.mx - ans.mi<<endl;
    }
}
```

6.8.3 思路

首先我们需要介绍一棵树的欧拉序列。

定义: 考虑一棵树的 dfs 遍历 (最后要回到根节点)，每遇到一个结点 (无论是否是回溯的) 便将序号其添加到序列中，最后得到的序列便是欧拉序列。这里的序号等于将第一次遇到该结点的时刻排序后的次序。

引理 8 *一棵树的欧拉序列长度为* $2n-1$.*(*n *是结点个数。)*

证明:

采取归纳法，当 $n=1$ 时，欧拉序列长度为 1，成立。

考虑一个子树 T_r 的根节点 r, 设其子树为 $T_1, T_2, ..., T_l$,

记函数 $f(T_r)$ 为子树欧拉序列长度，c(T_r) 为子树的大小,

从 r 出发回到 r 过程中经过 r 的次数为 $l+1$. 因此最后欧拉序列长度为

$$
\begin{aligned}
f(T_r) &= \sum_{i=1}^{l} f(T_i) + l + 1 \\
&= \sum_{i=1}^{l} (2c(T_i) - 1) + l + 1 \\
&= 2(c(T_r) - 1) - l + l + 1 \\
&= 2c(T_r) - 1
\end{aligned}
$$

于是原命题得证。

引理 9 *对于树上两点 u, v, u, v 的最近公共祖先是欧拉序列中 u, v 对应位置 (如果有多个，任取一个) 组成的区间中的最小值。*

证明: 首先 dfs 的过程中对于每一条边最多经历 2 次，其次最近公共祖先必在上述区间中。

令 r 为满足上述性质的结点，若 r 不为 u, v 的最近祖先，则 $r->\text{lca}(u,v)$ 将会被遍历至少 3 次 $(r->u, u->r, r->\text{lca}(u,v))$。

与 dfs 性质不符，因此原命题证明。

6.8.4 例 17

洛谷-P3379 【模板】最近公共祖先（LCA） 题面同例 11，模板题，使用 RMQ 和线段树求解版本。

Code:

```
#include <bits/stdc++.h>
using namespace std;
#define MAX_SIZE 500001
#define P_INF 1'000'000'000

int euler_map[MAX_SIZE];
```

```
int euler_map_inv[MAX_SIZE];
int euler_path[MAX_SIZE<<1];
int map_path_ind[MAX_SIZE];
bool vis[MAX_SIZE];

struct seg_tree
{
    int s[MAX_SIZE<<3];
    void build(int l,int r,int p){
        if(l==r){
            s[p] = euler_path[l];
            return;
        }
        int mid = l+r>>1;
        build(l,mid,p<<1);
        build(mid+1,r,p<<1|1);
        s[p] = min(s[p<<1],s[p<<1|1]);
    }

    int query(int l,int r,int L,int R,int p){
        if(L<=l && R>=r) return s[p];
        else if(l>R || r<L) return P_INF;
        else{
            int mid = l+r>>1;
            return min(query(l,mid,L,R,p<<1),query(mid+1,r,L,R,
                p<<1|1));
        }
    }
};

seg_tree st;

int main(){
    int n,m,root;
    cin>>n>>m>>root;
    vector adj(n+1,vector<int>(0));
    for(int i = 1;i<=n-1;i++){
        int u,v;
        cin>>u>>v;
```

```
        adj[u].push_back(v);
        adj[v].push_back(u);
    }
    vis[root] = true;
    euler_map[root] = 1;
    euler_map_inv[1] = root;
    int dfs_order = 1;
    int ep = 0;
    auto dfs_euler = [&](auto self,int root)->void{
        vis[root] = true;
        euler_path[++ep] = euler_map[root];
        map_path_ind[euler_map[root]] = ep;
        for(int v : adj[root]){
            if(vis[v]==false){
                euler_map[v] = ++dfs_order;
                euler_map_inv[dfs_order] = v;
                self(self,v);
                euler_path[++ep] = euler_map[root];
                map_path_ind[euler_map[root]] = ep;
            }
        }
    };
    dfs_euler(dfs_euler,root);
    st.build(1,2*n-1,1);
    for(int i = 0;i<m;i++){
        int u,v;
        cin>>u>>v;
        u = map_path_ind[euler_map[u]];
        v = map_path_ind[euler_map[v]];
        if(u>v) swap(u,v);
        cout<<euler_map_inv[st.query(1,2*n-1,u,v,1)]<<endl;
    }
}
```

7 树链剖分

7.1 概述

定义: 将整棵树剖分为若干条链状结构, 使用其余的数据结构维护信息。

树链剖分有很多种，如重链剖分、长链剖分、实链剖分，未明确说明的情况下本文的树链剖分都特指重链剖分。

树链剖分在以下的操作场景种很有用:

1. 更新结点 u 和 v 路径上的所有结点。
2. 查询结点 u 到 v 路径上的和、最大值、最小值等符合**结合律**的函数值。

7.2 数据结构定义

1. 重儿子: 儿子结点中子树最大的一个。
2. 轻儿子: 非重儿子的儿子。
3. 重边: 连接一个结点和其重儿子的边。
4. 轻边: 连接一个结点和其任意轻儿子的边。
5. 重链: 由重边组成的路径, 通常将只有没有重边相连的叶子结点也视为一条重链。
6. 重链头: 一条重链中深度最低的结点, 记 $\mathrm{top}(u)$ 表示结点 u 所在的重链的重链头。
7. 轻链: 由轻边组成的路径。

7.3 性质

引理 10 对于任意结点 x 和其祖先结点 y，从 $x->y$ 的路径中经过轻边的个数不超过 $O(\log(n))$.

证明:

假设 $u->v$ 是其经过的一条轻边，其中 u 是 v 的父亲结点。

根据定义，u 必存在一重儿子 v', 且 $\mathrm{size}(T'_v) \geq \mathrm{size}(T_v)$。

因此 $\mathrm{size}(T_u) \geq 2\mathrm{size}(T_v)$.

假设 l 表示 $u->v$ 经过轻边的条数，根据上述迭代过程，则有

$$\begin{aligned}\text{size}(T_y) \geq 2^l \text{size}(T_x) &\Leftrightarrow l \leq \log(\text{size}(T_y)/\text{size}(T_x)) \\ &\Leftrightarrow l \leq O(\log(n)).\end{aligned}$$

引理 11 对于树上任意结点 x 和 y，从 $x->y$ 的路径中经过轻边的个数不超过 $O(\log(n))$.

证明:
假设 $r = \text{lca}(x, y)$, 因此 $x->y = x->r->y$.
根据引理10, $x->r, r->y$ 均为 $O(\log(n))$, 原命题得证。

7.4 树链剖分解决 lca 问题

7.4.1 思路

分析可得 lca 新的递推表达式:

不失一般性，这里约定 $h(\text{top}(u)) \geq h(\text{top}(v))$, 可得 $\text{lca}(u, v)$ 在不同条件下的递推式:

$$\begin{cases} u, \text{top}(u) = \text{top}(v) \ \wedge \ h(u) \leq h(v) \\ v, \text{top}(u) = \text{top}(v) \ \wedge \ h(u) > h(v) \\ \text{lca}(\text{top}(\text{fa}(u)), v), \text{top}(u) \neq \text{top}(v) \end{cases} \tag{13}$$

$\text{top}(u)$ 的更新递推式如下:

$$\text{top}(u) = \begin{cases} \text{top}(\text{fa}(u)), & u\text{是重儿子}, \\ u, & u\text{是轻儿子}. \end{cases}$$

7.4.2 复杂度分析

预处理 top 等必要函数映射为 $O(n)$.

分析递推表达式，执行 13 的次数等于 u, v 之间路径轻边的个数，为 $O(\log(n))$.

因此单次查询 $\text{lca}(u, v)$ 复杂度为 $O(\log(n))$.

7.4.3 例 12

洛谷-P3379 【模板】最近公共祖先（LCA）

题面同例 11, 模板题，使用上述式子递推即可，相较于倍增法，使用树链剖分的常数较小。

代码:

```
#include <bits/stdc++.h>
using namespace std;

const int MAX_SIZE = 5e5+1;
int size_tr[MAX_SIZE];
int h[MAX_SIZE];
int top[MAX_SIZE];
int fa[MAX_SIZE];
int heavy_son[MAX_SIZE];
vector adj(MAX_SIZE,vector<int>(0));

void dfs_1(int root){
    int sz = 1;
    for(int v:adj[root]){
        if(v==fa[root]) continue;
        fa[v] = root;
        h[v] = h[root]+1;
        dfs_1(v);
        sz+= size_tr[v];
        if(heavy_son[root]==0 || size_tr[v]>size_tr[heavy_son[root]]){
            heavy_son[root] = v;
        }

    }
    size_tr[root] = sz;
};
void dfs_2(int root,int tp){
    for(int v:adj[root]){
        if(v==fa[root]) continue;
        if(v==heavy_son[root]){
            top[v]= tp;
```

```
            dfs_2(v,tp);
        }
        else{
            top[v] = v;
            dfs_2(v,v);
        }
    }
};
int lca(int u,int v){
    while(top[u]!=top[v]){
        if(h[top[u]]<h[top[v]]) swap(u,v);
        u = fa[top[u]];
    }
    return h[u]<h[v]?u:v;
};
int main(){
    h[0] = -1;
    ios::sync_with_stdio(false);
    cin.tie(0);
    int n,m,root;
    cin>>n>>m>>root;
    int u,v;
    for(int i = 0;i<n-1;i++){
        cin>>u>>v;
        adj[u].push_back(v);
        adj[v].push_back(u);
    }
    top[root] = root;
    dfs_1(root);
    dfs_2(root,root);
    for(int i = 0;i<m;i++){
        cin>>u>>v;
        cout<<lca(u,v)<<"\n";
    }
}
```

7.5 树链剖分与线段树

如果规定 dfs 每次必须先走重儿子，那么每条重链的结点在 DFS 序中将是连续的。

那么对于每条重链上的信息,我们可以使用线段树维护,最终查询 $u->v$ 的复杂度为 $O(n\log^2(n))$.

7.5.1 例 18

CSEC-Path Queries II

模板题，只需要维护简单的路径求和。

注: 这题最后一个测试点 TLE 了，经过长达几个小时的 DEBUG，总结出以下特性:

1. endl 速度及其缓慢，应该严格限制。
2. auto lambda 表达式对于运行速度的影响不大。
3. 关于左移位和右移位理论上会增加速度, 但是通常编译器可以优化, 因此影响不大。
4. scanf 速度慢于加了读入优化的 cin。

```
#include <bits/stdc++.h>
using namespace std;
#define MAX_SIZE 200'001
#define N_INF 0

int n;
int id_to_dfn[MAX_SIZE];
int dfn_to_id[MAX_SIZE];
int val[MAX_SIZE];
int dfs_order[MAX_SIZE];
int fa[MAX_SIZE];
int heavy_son[MAX_SIZE];
int size_tr[MAX_SIZE];
int top[MAX_SIZE];
int h[MAX_SIZE];

```

```
auto find_data(int i){
    return val[dfn_to_id[i]];
}

struct seg_tree
{
    int s[MAX_SIZE<<2];
    void build(int l,int r,int p){
        if(l==r){
            s[p] = find_data(l);
            return;
        }
        int mid = l+r>>1;
        build(l,mid,p<<1);
        build(mid+1,r,p<<1 | 1);
        s[p] = max(s[p<<1], s[p<<1|1]);
    }
    int query(int l,int r,int L,int R,int p){
        if(L<=l && R>=r) return s[p];
        else if(l>R || r<L) return N_INF;
        else{
            int mid =  l+r>>1;
            return max(query(l,mid,L,R,p<<1), query(mid+1,r,L,R,p<<1 | 1));
        }
    }
    void update(int l,int r,int index,int val,int p){
        if(l==r && l==index) s[p] = val;
        else if(l>index || r<index) return;
        else{
            int mid = l+r>>1;
            update(l,mid,index,val,p<<1);
            update(mid+1,r,index,val,p<<1 | 1);
            s[p] = max(s[p<<1], s[p<<1 | 1]);
        }
    }
};

seg_tree st;
```

```
int query_on_heavy_chain(int u,int v){
    if(id_to_dfn[u]>id_to_dfn[v]) swap(u,v);
    return st.query(1,n,id_to_dfn[u],id_to_dfn[v],1);
}

int merge_query(int u,int v){
    int ans = 0;
    while(top[u] !=top[v]){
        if(h[top[u]] < h[top[v]]) swap(u,v);
        ans = max(ans,query_on_heavy_chain(u,top[u]));
        u = fa[top[u]];
    }
    ans = max(ans,query_on_heavy_chain(u,v));
    return ans;
};
vector adj(0,vector<int>(0));
void dfs_1(int root){
    size_tr[root] = 1;
    for(int v:adj[root]){
        if(v==fa[root]) continue;
        fa[v] = root;
        h[v] = h[root]+1;
        dfs_1(v);
        size_tr[root] += size_tr[v];
        if(heavy_son[root]==0 || size_tr[heavy_son[root]]<
            size_tr[v]) heavy_son[root] = v;
    }
};
int main(){
    ios::sync_with_stdio(false);
    cin.tie(0);
    cout.tie(0);
    // freopen("Code_18Test.txt","r",stdin);
    // freopen("Code_18out.txt","w",stdout);
    // clock_t begin_time = clock();
    int q;
    cin>>n>>q;
    for(int i = 1;i<=n;i++){
        cin>>val[i];
```

```
    }
    adj.resize(n+1);
    for(int i = 1;i<=n-1;i++){
        int u,v;
        cin>>u>>v;
        adj[u].push_back(v);
        adj[v].push_back(u);
    }
    int root = 1;
    h[root] = 0;
    h[0] = -1;

    dfs_1(root);
    int dfs_or = 1;
    dfs_order[root] = 1;
    top[root] = root;
    id_to_dfn[root] = 1;
    dfn_to_id[1] = root;
    auto dfn_update = [&](int node){
        dfs_order[node] = ++dfs_or;
        id_to_dfn[node] = dfs_order[node];
        dfn_to_id[dfs_order[node]] = node;
    };
    auto dfs_2 = [&](auto self,int root)->void{
        if(heavy_son[root]!=0){
            dfn_update(heavy_son[root]);
            top[heavy_son[root]] = top[root];
            self(self,heavy_son[root]);
        }
        for(int v:adj[root]){
            if(v==fa[root] || v==heavy_son[root]) continue;
            dfn_update(v);
            top[v] = v;
            self(self,v);
        }
    };
    dfs_2(dfs_2,root);
    st.build(1,n,1);
    for(int i = 1;i<=q;i++){
```

```
        int type;
        cin>>type;
        if(type==1){
            int u;
            cin>>u>>val[u];
            st.update(1,n,id_to_dfn[u],val[u],1);
        }
        else{
            int u,v;
            cin>>u>>v;
            cout<<merge_query(u,v)<<"\n";
        }
    }
    // cerr<<(clock()-begin_time)/1000.0<<"\n";
}
```

8 参考资料

[1]. OI-wiki

[2]. Geeksforgeeks

[3]. CodeForces-Tutorial

[4]. CSDN 论坛

[5]. 算法导论 (第三版)

[6]. Wikipedia

[7]. AC Library