

数据结构-树上问题

ZeitHaum

2023 年 4 月 16 日

目录

1 基本概念	1
2 基础例题	1
2.1 例 1	1
2.2 例 2	2
2.3 例 3	3
2.4 例 4	4
2.5 例 5	6
3 *Morris 遍历	7
4 树的直径	7
4.1 算法一:DP	8
4.2 算法二: 遍历	8
4.3 适用性	9
4.4 例 6	9
4.4.1 Solution	9
4.4.2 Code	9
4.5 例 7	10
4.5.1 Solution	10
4.5.2 Code	10
4.6 例 8	12
4.6.1 Code	13
5 最近公共祖先问题	18
5.1 朴素算法	18
5.1.1 例 9	18
5.2 倍增算法	20
5.3 子问题一: 二进制转换	20
5.4 思路	22
5.5 例 11	23
6 线段树	24

6.1	概述	24
6.2	操作需要满足的条件	24
6.3	结构	25
6.4	性质	25
6.5	PURQ 数据结构	26
6.5.1	例 13	27
6.6	RURQ 数据结构	29
6.6.1	例 14	29
6.7	例 15	32
6.8	线段树求解 lca 问题	35
6.8.1	子问题:RMQ 问题	35
6.8.2	例 16	35
6.8.3	思路	37
6.8.4	例 17	38
7	树链剖分	41
7.1	概述	41
7.2	数据结构定义	41
7.3	性质	41
7.4	树链剖分解决 lca 问题	42
7.4.1	思路	42
7.4.2	复杂度分析	42
7.4.3	例 12	43
7.5	树链剖分与线段树	45
7.5.1	例 18	45
8	参考资料	49

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 $\text{lca}(u, v)$ 。

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。

```
1  /**
2   * Definition for a binary tree node.
3   * struct TreeNode {
4   *     int val;
5   *     TreeNode *left;
6   *     TreeNode *right;
7   *     TreeNode() : val(0), left(nullptr), right(nullptr) {}
8   *     TreeNode(int x) : val(x), left(nullptr), right(nullptr)
9   *     {}
10    *     TreeNode(int x, TreeNode *left, TreeNode *right) : val(x
11    *     ), left(left), right(right) {}
```

```

10  * };
11  */
12  class Solution {
13  public:
14      vector<int>ans;
15
16      void dfs(TreeNode* root){
17          if(root==nullptr) return;
18          dfs(root->left);
19          ans.push_back(root->val);
20          dfs(root->right);
21      }
22
23      vector<int> inorderTraversal(TreeNode* root) {
24          ans.clear();
25          dfs(root);
26          return ans;
27      }
28  };

```

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。

```

1  /**
2   * Definition for a binary tree node.
3   * struct TreeNode {
4   *     int val;
5   *     TreeNode *left;
6   *     TreeNode *right;
7   *     TreeNode() : val(0), left(nullptr), right(nullptr) {}
8   *     TreeNode(int x) : val(x), left(nullptr), right(nullptr)
9   *         {}
10    *     TreeNode(int x, TreeNode *left, TreeNode *right) : val(x
11    *         ), left(left), right(right) {}
12    * };
13    */
14    class Solution {
15    public:

```

```

14     bool check = true;
15     void dfs_same(TreeNode* r1,TreeNode* r2){
16         if(check==false) return;
17         if(r1==nullptr || r2==nullptr){
18             if(!(r1 == nullptr && r2==nullptr)) check = false;
19             return;
20         }
21         if(r1->val!=r2->val) check = false;
22         dfs_same(r1->left,r2->left);
23         dfs_same(r1->right,r2->right);
24     }
25
26     bool isSameTree(TreeNode* p, TreeNode* q) {
27         check = true;
28         dfs_same(p,q);
29         return check;
30     }
31 };

```

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。

```

1  /**
2   * Definition for a binary tree node.
3   * struct TreeNode {
4   *     int val;
5   *     TreeNode *left;
6   *     TreeNode *right;
7   *     TreeNode() : val(0), left(nullptr), right(nullptr) {}
8   *     TreeNode(int x) : val(x), left(nullptr), right(nullptr)
9   *     {}
10    *     TreeNode(int x, TreeNode *left, TreeNode *right) : val(x
11    *     ), left(left), right(right) {}
12    * };
13    */
14    class Solution {
15    public:
16        bool ck = true;

```

```

15 void dfs_symmetric(TreeNode* r1,TreeNode* r2){
16     if(r1==nullptr || r2==nullptr){
17         if(!(r1==nullptr && r2 == nullptr)) ck = false;
18         return;
19     }
20     if(r1->val!=r2->val) ck = false;
21     if(ck==false) return;
22     dfs_symmetric(r1->left,r2->right);
23     dfs_symmetric(r1->right,r2->left);
24 }
25
26 bool isSymmetric(TreeNode* root) {
27     ck = true;
28     if(root==nullptr) return ck;
29     dfs_symmetric(root->left,root->right);
30     return ck;
31 }
32 };

```

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。

```

1 /**
2  * Definition for a binary tree node.
3  * struct TreeNode {
4  *     int val;
5  *     TreeNode *left;
6  *     TreeNode *right;
7  *     TreeNode() : val(0), left(nullptr), right(nullptr) {}
8  *     TreeNode(int x) : val(x), left(nullptr), right(nullptr)
9  *     {}
10    *     TreeNode(int x, TreeNode *left, TreeNode *right) : val(x
11    *     ), left(left), right(right) {}
12    * };
13 */
14 class Solution {
15 public:
16     int n;

```

```

15     map<int,int>f;
16
17     TreeNode* buildTree(vector<int>& preorder, vector<int>&
18         inorder) {
19         function<void(int,int,int,int,TreeNode*)> dfs_build =
20             [&](int pbegin,int pend,int ibegin,int iend,
21                 TreeNode* root){
22                 //左闭右开
23                 int i_ind = f[root->val];
24                 int left_len = i_ind-1 - ibegin + 1;
25                 int right_len = iend-1 - (i_ind+1) + 1;
26                 if(left_len!=0){
27                     int root_val = preorder[pbegin+1];
28                     TreeNode* left = new TreeNode(root_val);
29                     root->left = left;
30                     dfs_build(pbegin+1,pbegin+1+left_len,ibegin,
31                         i_ind,left);
32                 }
33                 if(right_len!=0){
34                     int root_val = preorder[pbegin+1+left_len];
35                     TreeNode* right = new TreeNode(root_val);
36                     root->right = right;
37                     dfs_build(pbegin+1+left_len,pend,i_ind+1,iend,
38                         right);
39                 }
40             };
41     n = preorder.size();
42     f.clear();
43     map<int,int>g;
44     for(int i = 0;i<n;i++){
45         g[inorder[i]] = i;
46     }
47     for(int i = 0;i<n;i++){
48         f[preorder[i]] = g[preorder[i]];
49     }
50     TreeNode* root = new TreeNode(preorder[0]);
51     dfs_build(0,n,0,n,root);
52     return root;
53 }

```



```
49 };
```

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）。

```
1  /**
2   * Definition for a binary tree node.
3   * struct TreeNode {
4   *     int val;
5   *     TreeNode *left;
6   *     TreeNode *right;
7   *     TreeNode() : val(0), left(nullptr), right(nullptr) {}
8   *     TreeNode(int x) : val(x), left(nullptr), right(nullptr)
9   *         {}
10    *     TreeNode(int x, TreeNode *left, TreeNode *right) : val(x
11    *         ), left(left), right(right) {}
12    * };
13    */
14    class Solution {
15    public:
16        int n;
17        map<int,int>f;
18        TreeNode* buildTree(vector<int>& inorder, vector<int>&
19        postorder) {
20            //declaration of dfs.
21            function<void(int,int,int,int,TreeNode*)> dfs_build =
22            [&](int ibegin,int iend,int pbegin,int pend,
23            TreeNode* root){
24                int i_ind = f[root->val];
25                int left_len = i_ind-1 - ibegin + 1;
26                int right_len = iend-1 - (i_ind+1) + 1;
27                if(left_len!=0){
28                    TreeNode* left = new TreeNode(postorder[pbegin+
29                    left_len-1]);
30                    root->left = left;
31                    dfs_build(ibegin,i_ind,pbegin,pbegin+left_len,
32                    left);
```

```

26         }
27         if(right_len!=0){
28             TreeNode* right = new TreeNode(postorder[pend
29                 -2]);
30             root->right = right;
31             dfs_build(i_ind+1, iend, pbegin+left_len, pend-1,
32                 right);
33         }
34     };
35     //initialize of attributes.
36     n = inorder.size();
37     f.clear();
38     map<int,int>g;
39     for(int i = 0;i<n;i++){
40         g[inorder[i]] = i;
41     }
42     for(int i = 0;i<n;i++){
43         f[postorder[i]] = g[postorder[i]];
44     }
45     //calc ans.
46     TreeNode* root = new TreeNode(postorder[n-1]);
47     dfs_build(0,n,0,n,root);
48     return root;
49 }
50 };

```

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

$$D(T) = \max(D(T_l), D(T_r), h(T_l) + h(T_r)) \quad (1)$$

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。

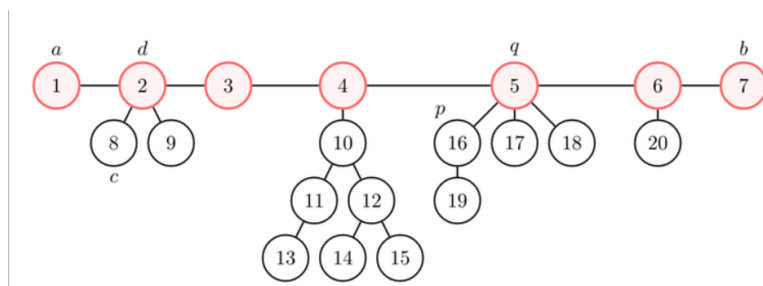
4.2 算法二: 遍历

记 $\text{dis}(u, v)$ 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 $a = \max_i(\text{dis}(x, i))$, 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 (图片仅作参考):



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 $\text{dis}(x, s) \leq \text{dis}(x, t)$ 。

则有

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

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 $b = \max_i(\text{dis}(a, i))$, 则 $a \rightarrow b$ 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

```
1  /**
2   * Definition for a binary tree node.
3   * struct TreeNode {
4   *     int val;
5   *     TreeNode *left;
6   *     TreeNode *right;
7   *     TreeNode() : val(0), left(nullptr), right(nullptr) {}
8   *     TreeNode(int x) : val(x), left(nullptr), right(nullptr)
9   *         {}
10    *     TreeNode(int x, TreeNode *left, TreeNode *right) : val(x
11    *         ), left(left), right(right) {}
12    * };
13    */
14    //DP version
15    class Solution {
16    public:
17        int ans = 0;
18        int dfs_dp(TreeNode* root){
19            if(root==nullptr) return 0;
```

```

18     int h_l = dfs_dp(root->left);
19     int h_r = dfs_dp(root->right);
20     ans = max(ans, h_l+h_r);
21     return max(h_l, h_r)+1;
22 }
23
24 int diameterOfBinaryTree(TreeNode* root) {
25     dfs_dp(root);
26     return ans;
27 }
28 };

```

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

```

1 #include <bits/stdc++.h>
2 using namespace std;
3
4 ostream& operator<<(ostream& out, vector<int>& arr){
5     for(int i = 0; i<arr.size(); i++){
6         out<<arr[i]<<" ";
7     }
8     return out;

```

```

9  }
10
11 int main(){
12     int n;
13     cin>>n;
14     vector adj(n+1,vector<int>(0));
15     int u,v;
16     for(int i = 0;i<n-1;i++){
17         cin>>u>>v;
18         adj[u].push_back(v);
19         adj[v].push_back(u);
20     }
21     vector dis_1(n+1,-1);
22     vector dis_2(n+1,-1);
23     auto bfs = [&](vector<int>& height,int root){
24         queue<int> q;
25         q.push(root);
26         height[root] = 0;
27         vector<bool>vis(n+1,false);
28         while(q.size()!=0){
29             int qf = q.front();
30             q.pop();
31             vis[qf] = true;
32             for(int i = 0;i<adj[qf].size();i++){
33                 int now = adj[qf][i];
34                 if(!vis[now]){
35                     q.push(now);
36                     height[now] = height[qf]+1;
37                     vis[now] = true;
38                 }
39             }
40         }
41     };
42     bfs(dis_2,1);
43     int max_root = max_element(dis_2.begin(),dis_2.end()) -
        dis_2.begin();
44     bfs(dis_1,max_root);
45     max_root = max_element(dis_1.begin(),dis_1.end()) - dis_1.
        begin();

```

```

46     bfs(dis_2,max_root);
47     for(int i = 0;i<dis_1.size();i++){
48         dis_1[i] = max(dis_1[i],dis_2[i]);
49     }
50     sort(dis_1.begin(),dis_1.end());
51     vector ans(n+1,0);
52     for(int i =1;i<dis_1.size();i++){
53         ans[dis_1[i]]++;
54     }
55     cerr<<ans<<endl;
56     ans[n] = n;
57     bool enable = false;
58     for(int i = ans.size()-2;i>=0;i--){
59         int temp = 0;
60         if(ans[i]!=0 && !enable){
61             temp = 1;
62             enable = true;
63         }
64         ans[i] = ans[i+1] - ans[i] + temp;
65     }
66     for(int i = 1;i<ans.size();i++){
67         cout<<ans[i]<<" ";
68     }
69     cout<<endl;
70 }

```

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 $s \rightarrow r$ 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

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

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。

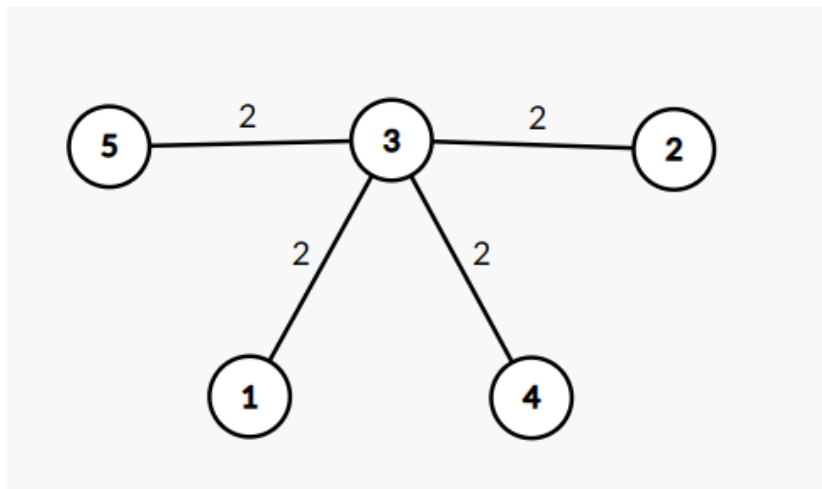
记

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

于是

$$\text{ans} = \text{abs}(f(s) - f(t)). \quad (5)$$

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

```

1 #include<bits/stdc++.h>
2 using namespace std;
3 #define int long long
4 #define mp(x,y) make_pair((x),(y))
5
6 ostream& operator<<(ostream& out, const vector<int>& arr){
7     for(int i = 0; i<arr.size(); i++){
8         out<<arr[i]<<" ";
9     }

```



```

10     return out;
11 }
12 ostream& operator<<(ostream& out,const vector<bool>& arr){
13     for(int i = 0;i<arr.size();i++){
14         out<<arr[i]<<" ";
15     }
16     return out;
17 }
18 ostream& operator<<(ostream& out,const pair<int,int>& p){
19     out<<p.first<<","<<p.second<<" ";
20     return out;
21 }
22 ostream& operator<<(ostream& out,const map<pair<int,int>,int>&
    dic){
23     for(auto iter = dic.begin();iter!=dic.end();iter++){
24         out<<iter->first<<":"<<iter->second<<" ";
25     }
26     return out;
27 }
28
29 signed main(){
30     int n;
31     cin>>n;
32     //handling the input.
33     const int ARR_SIZE = n+1;
34     vector adj(ARR_SIZE,vector<int>(0));
35     map<pair<int,int>,int>weight;
36     int u,v,w;
37     for(int i = 0;i<n-1;i++){
38         cin>>u>>v>>w;
39         adj[u].push_back(v);
40         adj[v].push_back(u);
41         weight[mp(u,v)] = w;
42         weight[mp(v,u)] = w;
43     }
44     //Find a diameter point.
45     vector<int>h(ARR_SIZE,0);
46     vector<bool>vis(ARR_SIZE,false);
47     auto clear_vis = [&]{

```

```

48         for(int i = 0;i<vis.size();i++){
49             vis[i] = false;
50             h[i] = 0;
51         }
52     };
53     auto clear_h = [&]{
54         for(int i = 0;i<vis.size();i++){
55             h[i] = 0;
56         }
57     };
58     auto clear_vis_h = [&]{
59         clear_vis();
60         clear_h();
61     };
62     auto bfs_find = [&](vector<int>& h,int root){
63         queue<int> q;
64         q.push(root);
65         vis[root] = true;
66         while(q.size()>0){
67             int qf = q.front();
68             q.pop();
69             for(int i = 0;i<adj[qf].size();i++){
70                 int now = adj[qf][i];
71                 if(vis[now]==false){
72                     q.push(now);
73                     vis[now] = true;
74                     h[now] = h[qf] + weight[mp(qf,now)];
75                 }
76             }
77         }
78         return max_element(h.begin(),h.end());
79     };
80     clear_vis_h();
81     int da = bfs_find(h,1) - h.begin();
82     clear_vis_h();
83     int db = bfs_find(h,da) - h.begin();
84     int d_len = h[db];
85     //Find a diameter path.
86     vector<int> diameter_path(0);

```

```

87     for(int i = 0;i<vis.size();i++){
88         vis[i] = false;
89     }
90     bool enable_path = false;
91     function<void(int,int)> dfs_path = [&](int root,int
        now_weight){
92         diameter_path.push_back(root);
93         vis[root] = true;
94         for(int i = 0;i<adj[root].size();i++){
95             int now = adj[root][i];
96             if(vis[now]==false){
97                 if(now_weight+ weight[mp(root,now)]==d_len &&
                    adj[now].size()==1){
98                     diameter_path.push_back(now);
99                     enable_path = true;
100                     return;
101                 }
102                 dfs_path(now,now_weight+weight[mp(root,now)]);
103                 if(enable_path) return;
104                 diameter_path.pop_back();
105             }
106         }
107     };
108     dfs_path(da,0);
109     int d_size = diameter_path.size();
110     //Check connectivity.
111     for(int i = 0;i<vis.size();i++){
112         vis[i] = false;
113     }
114     for(int i =0;i<d_size;i++){
115         vis[diameter_path[i]] = true;
116     }
117     int temp_h;
118     vector<int> t_r(d_size,0);
119     function<void(int,int)> dfs_check = [&](int root,int j){
120         vis[root] = true;
121         for(int i = 0;i<adj[root].size();i++){
122             int now = adj[root][i];
123             if(vis[now]==false){

```

```

124         temp_h += weight[mp(root,now)];
125         //Leaf Node;
126         if(adj[now].size()==1){
127             t_r[j] = max(t_r[j],temp_h);
128         }
129         dfs_check(now,j);
130         temp_h -= weight[mp(root,now)];
131     }
132 }
133 };
134 for(int i = 0;i<d_size;i++){
135     temp_h = 0;
136     dfs_check(diameter_path[i],i);
137 }
138 int fs = max_element(t_r.begin(),t_r.begin()+(d_size-1)
139                     /2+1) - t_r.begin();
140 int ft = max_element(t_r.begin()+(d_size-1)/2+1,t_r.end())
141         - t_r.begin();
142 int ans2 = abs(fs - ft);
143 //Special Judge
144 int accu_w = 0;
145 for(int i = 1;i<d_size;i++){
146     accu_w += weight[mp(diameter_path[i-1],diameter_path[i]
147                       )]];
148     if(d_len - accu_w==accu_w){
149         if(t_r[i]==accu_w) ans2 = 0;
150         break;
151     }
152 }
153 cout<<d_len<<endl;
154 cout<<ans2<<endl;
155 }
156 /*
157 5
158 1 3 2
159 2 3 2
160 4 3 2
161 5 3 2

```

```

160 | 9
161 | 1 2 1
162 | 2 5 3
163 | 3 4 2
164 | 4 5 1
165 | 5 6 1
166 | 6 7 3
167 | 5 8 2
168 | 8 9 2
169 | */

```

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 $\text{lca}(u, v)$ 。

性质:

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

5.1 朴素算法

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

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

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

```

1  /**
2   * Definition for a binary tree node.
3   * struct TreeNode {
4   *     int val;
5   *     TreeNode *left;
6   *     TreeNode *right;
7   *     TreeNode(int x) : val(x), left(NULL), right(NULL) {}
8   * };
9   */
10 class Solution {
11 public:
12     TreeNode* lowestCommonAncestor(TreeNode* root, TreeNode* p,
13         TreeNode* q) {
14         map<int,TreeNode*> fa;
15         map<int,int> h;
16         auto bfs = [&]() {
17             queue<TreeNode*> q;
18             q.push(root);
19             fa[root->val] = NULL;
20             h[root->val] = 0;
21             while(q.size() != 0) {
22                 TreeNode* qf = q.front();
23                 q.pop();
24                 if(qf->left != NULL) {
25                     fa[qf->left->val] = qf;
26                     h[qf->left->val] = h[qf->val] + 1;
27                     q.push(qf->left);
28                 }
29                 if(qf->right != NULL) {
30                     fa[qf->right->val] = qf;
31                     h[qf->right->val] = h[qf->val] + 1;
32                     q.push(qf->right);
33                 }
34             }
35         };
36         bfs();
37         function<TreeNode*(TreeNode*,TreeNode*)> find_lca =
38             [&](TreeNode* p,TreeNode* q) {

```

```

37         if(p->val==q->val) return p;
38         if(h[p->val]>h[q->val]) return find_lca(fa[p->val],
           q);
39         else if(h[p->val]<h[q->val]) return find_lca(p,fa[q
           ->val]);
40         else return find_lca(fa[p->val],fa[q->val]);
41     };
42     return find_lca(p,q);
43 }
44 };

```

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

$$\text{binary_str}(x) = \begin{cases} \text{binary_str}(x/2) + '0', & x\%2 = 0 \\ \text{binary_str}(x/2) + '1', & x\%2 = 1 \end{cases} \quad (7)$$

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 $\{2^0, 2^1, \dots, 2^k\}, k \leq \lfloor \log_2(n) \rfloor$ 的值，于是可有

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

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

```

1 #include <bits/stdc++.h>
2 using namespace std;
3
4 int pow2[31];
5
6 string binary_str1(int x){

```

```
7     string rans; //初始设置ans为答案的翻转串
8     while(true){
9         if(x%2==0) rans.push_back('0');
10        else rans.push_back('1');
11        x = x/2;
12        if(x==0) break;
13    }
14    reverse(rans.begin(),rans.end());
15    return rans;
16 }
17
18 string binary_str2(int x){
19     string ans;
20     for(int i = 30;i>=0;i--){
21         if(x>= pow2[i]){
22             x -= pow2[i];
23             ans.push_back('1');
24         }
25         else ans.push_back('0');
26     }
27     //如果要求去除前导0...
28     int first_1 = 0;
29     for(int i = 1;i<=31;i++){
30         if(ans[i]=='1'){
31             first_1 = i;
32             break;
33         }
34     }
35     return ans.substr(first_1,31 - first_1+1);
36 }
37
38 int main(){
39     //预处理pow2
40     for(int i = 0;i<31;i++){
41         if(i==0) pow2[i] = 1;
42         else pow2[i] = pow2[i-1] * 2;
43     }
44     int x = 13;
45     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, 不失一般性地假设 $h(u) > h(v)$, 于是必存在 x 使得 $h(\text{fa}^x(u)) = h(v)$, 其中

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

对于任意 k , 根据二进制分解的原理有

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

如果对于任意结点 u 我们提前预处理 $\text{fa}^{2^k}(u)$ 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 $\text{fa}^y(u) = \text{fa}^y(v)$ 。

同理, 根据二进制分解的原理, 对于 k 有

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

最后需要考虑 $\text{fa}^{2^k}(u)$ 的维护:

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

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。

```

1  #include <bits/stdc++.h>
2  using namespace std;
3
4  const int MAX_SIZE = 5e5+1;
5  int fa[MAX_SIZE][21];
6  int h[MAX_SIZE];
7
8  int main(){
9      ios::sync_with_stdio(false);
10     cin.tie(0);
11     int n,m,root;
12     cin>>n>>m>>root;
13     vector adj(n+1,vector<int>(0));
14     int u,v;
15     for(int i = 0;i<n-1;i++){
16         cin>>u>>v;
17         adj[u].push_back(v);
18         adj[v].push_back(u);
19     }
20     auto dfs = [&](auto self,int root,int depth)->void{
21         for(int v: adj[root]){
22             if(v==fa[root][0]) continue;
23             fa[v][0] = root;
24             h[v] = depth+1;
25             self(self,v,depth+1);
26         }
27     };
28     dfs(dfs,root,0);
29     h[0] = -1;
30     for(int i = 1;i<21;i++){
31         for(int j = 1;j<=n;j++){
32             fa[j][i] = fa[fa[j][i-1]][i-1];
33         }
34     }
35     auto lca = [&](int u,int v){
36         if(h[u]<h[v]) swap(u,v);

```

```
37         if(h[u]!=h[v]){
38             for(int i = 20;i>=0;i--){
39                 if(h[fa[u][i]]>=h[v]) u = fa[u][i];
40             }
41         }
42         if(u==v) return u;
43         else{
44             for(int i = 20;i>=0;i--){
45                 if(fa[u][i] != fa[v][i]){
46                     u = fa[u][i];
47                     v = fa[v][i];
48                 }
49             }
50             return fa[u][0];
51         }
52     };
53     for(int i = 0;i<m;i++){
54         cin>>u>>v;
55         cout<<lca(u,v)<<endl;
56     }
57 }
```

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. 操作必须有么元。

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 $[1, 1], [2, 2], \dots, [n-1, n-1], [n, n]$ 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

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

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, $f(n) \leq 2 \times (n/2 \times 2 - 1) + 1 \Leftrightarrow f(n) \leq 2n - 1$.

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

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

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

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 为查询信息的么元, \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 $\text{inf}(l, r)$.

有以下递推式:

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

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](#)

```

1  class NumArray {
2  public:
3      vector<int>s;
4      int n;
5
6      NumArray(vector<int>& nums) {
7          n = nums.size();
8          s.resize(n*4+1);
9          nums.insert(nums.begin(),0);
10         auto build = [&](auto self,int l,int r,int p)->void{
11             if(l==r){
12                 s[p] = nums[l];
13                 return;
14             }
15             int mid = (l+r)/2;
16             self(self,l,mid,p*2);
17             self(self,mid+1,r,p*2+1);
18             s[p] = s[p*2] + s[p*2+1];
19         };
20         build(build,1,n,1);
21     }
22
23     void update(int index, int val) {
24         index = index + 1;
25         auto up = [&](auto self,int l,int r,int p)->void{
26             if(l<=index && r>=index){
27                 if(l==r){
28                     s[p] = val;
29                     return;
30                 }
31                 int mid = (l+r)/2;
32                 self(self,l,mid,p*2);
33                 self(self,mid+1,r,p*2+1);
34                 s[p] = s[p*2] + s[p*2+1];
35                 return;

```

```

36         }
37         else{
38             return;
39         }
40     };
41     up(up,1,n,1);
42 }
43
44 int sumRange(int left, int right) {
45     left = left + 1;
46     right = right + 1;
47     auto query = [&](auto self, int l,int r,int p)->int{
48         if(left<=l && right>= r){
49             return s[p];
50         }
51         else if(l>right || r<left){
52             return 0;
53         }
54         else{
55             int mid = (l+r)/2;
56             return self(self,l,mid,p*2)+self(self,mid+1,r,p
                    *2+1);
57         }
58     };
59     return query(query,1,n,1);
60 }
61 };
62
63 /**
64  * Your NumArray object will be instantiated and called as such
65  * :
66  * NumArray* obj = new NumArray(nums);
67  * obj->update(index,val);
68  * int param_2 = obj->sumRange(left,right);
69  */

```

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:

```
1 #include <bits/stdc++.h>
2 using namespace std;
3 #define MAX_N 100001
4 #define int long long
5
6 struct S{
7     int val;
8     int len;
9     S():val(0),len(0){}
```



```

10     S(int val,int len):val(val),len(len){}
11     S operator+(const S& s){
12         return {this->val+s.val,this->len+s.len};
13     }
14 };
15 int da[MAX_N];
16 S s[4*MAX_N];
17 const S se = {0,0};
18 int lazy[4*MAX_N];
19 struct seg_tree{
20     //将lazy 标记向下传递
21     void push_down(int p){
22         lazy[p*2] += lazy[p];
23         lazy[p*2+1] += lazy[p];
24         s[p*2].val += lazy[p]*s[p*2].len;
25         s[p*2+1].val += lazy[p]*s[p*2+1].len;
26         lazy[p] = 0;
27     }
28     //build
29     void build(int l,int r,int p){
30         if(l==r){
31             s[p] = {da[l],1};
32             return;
33         }
34         int mid = (l+r)/2;
35         build(l,mid,p*2);
36         build(mid+1,r,p*2+1);
37         s[p] = s[p*2] + s[p*2+1];
38     }
39     //query
40     S query(int l,int r,int p,int L,int R){
41         if(L<=l && R>=r) return s[p];
42         else if(l>R || r<L) return se;
43         else{
44             int mid = (l+r)/2;
45             push_down(p);
46             return query(l,mid,p*2,L,R) + query(mid+1,r,p*2+1,L
47                 ,R);

```

```

48     }
49     //update
50     void update(int l ,int r,int p,int L,int R,int val){
51         if(L<=l && R>=r) {
52             lazy[p] += val;
53             s[p].val += s[p].len*val;
54         }
55         else if(l>R || r<L) return;
56         else{
57             int mid = (l+r)/2;
58             push_down(p);
59             update(l,mid,p*2,L,R,val);
60             update(mid+1,r,p*2+1,L,R,val);
61             s[p] = s[p*2] + s[p*2+1];
62         }
63     }
64 };
65
66 signed main(){
67     ios::sync_with_stdio(false);
68     cin.tie(0);
69     int n,m;
70     cin>>n>>m;
71     seg_tree st;
72     for(int i = 1;i<=n;i++){
73         cin>>da[i];
74     }
75     st.build(1,n,1);
76     int type;
77     int L,R,val;
78     for(int i = 1;i<=m;i++){
79         cin>>type;
80         if(type==1){
81             cin>>L>>R>>val;
82             st.update(1,n,1,L,R,val);
83         }
84         else{
85             cin>>L>>R;
86             cout<<st.query(1,n,1,L,R).val<<endl;

```

```

87     }
88   }
89 }

```

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:

```

1  #include <bits/stdc++.h>
2  using namespace std;
3  #define int long long
4  #define MAX_SIZE 100001
5
6  int p;
7  struct S
8  {
9      int val;
10     int len;

```

```

11     S():val(0),len(0){}
12     S(int val,int len):val(val),len(len){}
13     S merge(S s){ return {(this->val+s.val)%p,this->len+s.len
14         };}
15
16 struct F
17 {
18     int add;
19     int mul;
20     F():add(0),mul(1){}
21     F(int add,int mul):add(add),mul(mul){}
22     F composition(F f){ return {((this->add*f.mul)%p + f.add)%p
23         ,(this->mul*f.mul)%p};}
24 };
25 S apply(F f, S s){
26     return {(s.val * f.mul)%p + (s.len * f.add)%p,s.len};
27 }
28
29 int da[MAX_SIZE];
30 S s[MAX_SIZE*4];
31 F lazy[MAX_SIZE*4];
32 S se;
33 F fe;
34
35 struct seg_tree
36 {
37     void build(int l,int r,int p){
38         if(l==r){
39             s[p] = {da[l],1};
40             return;
41         }
42         int mid = (l+r)/2;
43         build(l,mid,p*2);
44         build(mid+1,r,p*2+1);
45         s[p] = s[p*2].merge(s[p*2+1]);
46     }
47

```

```

48     void push_down(int p){
49         lazy[p*2] = lazy[p*2].composition(lazy[p]);
50         lazy[p*2+1] = lazy[p*2+1].composition(lazy[p]);
51         s[p*2] = apply(lazy[p],s[p*2]);
52         s[p*2+1] = apply(lazy[p],s[p*2+1]);
53         lazy[p] = fe;
54     }
55
56     S query(int l,int r,int p,int L,int R){
57         if(L<= l && R>=r) return s[p];
58         else if(l>R || r<L) return se;
59         else{
60             int mid = (l+r)/2;
61             push_down(p);
62             return query(l,mid,p*2,L,R).merge(query(mid+1,r,p
               *2+1,L,R));
63         }
64     }
65
66     void update(int l,int r,int p,int L,int R,int add,int mul){
67         if(L<=l && R>=r){
68             lazy[p] = lazy[p].composition({add,mul});
69             s[p] = apply({add,mul},s[p]);
70         }
71         else if(l>R || r<L) return;
72         else{
73             int mid = (l+r)/2;
74             push_down(p);
75             update(l,mid,p*2,L,R,add,mul);
76             update(mid+1,r,p*2+1,L,R,add,mul);
77             s[p] = s[p*2].merge(s[p*2+1]);
78         }
79     }
80 };
81
82
83 signed main(){
84     ios::sync_with_stdio(false);
85     cin.tie(0);

```

```

86     int n,m;
87     cin>>n>>m>>p;
88     for(int i = 1;i<=n;i++){
89         cin>>da[i];
90     }
91     seg_tree st;
92     st.build(1,n,1);
93     int type,L,R,add,mul;
94     for(int i = 1;i<=m;i++){
95         cin>>type;
96         if(type==1){
97             cin>>L>>R>>mul;
98             st.update(1,n,1,L,R,0,mul);
99         }
100        else if(type==2){
101            cin>>L>>R>>add;
102            st.update(1,n,1,L,R,add,1);
103        }
104        else{
105            cin>>L>>R;
106            cout<<st.query(1,n,1,L,R).val<<endl;
107        }
108    }
109 }

```

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:

```

1 #include <iostream>

```

```

2  #include <algorithm>
3  #define MAX_SIZE 50001
4  #define N_INF -1000001
5  #define P_INF 1000001
6  using namespace std;
7
8  struct S
9  {
10     int mi;
11     int mx;
12     S():mi(P_INF),mx(N_INF){}
13     S(int mi,int mx):mi(mi),mx(mx){}
14 };
15
16 inline S merge(S s1, S s2){
17     S ret;
18     ret.mi = min(s1.mi,s2.mi);
19     ret.mx = max(s1.mx,s2.mx);
20     return ret;
21 }
22
23 int da[MAX_SIZE];
24 S s[4*MAX_SIZE];
25 S se;
26
27 struct seg_tree
28 {
29     void build(int l,int r,int p){
30         if(l==r){
31             s[p].mi = da[l];
32             s[p].mx = da[l];
33             return;
34         }
35         int mid = l+r>>1;
36         build(l,mid,p<<1);
37         build(mid+1,r,p<<1 | 1);
38         s[p] = merge(s[p<<1],s[p<<1 | 1]);
39     }
40 }

```

```

41     inline S query(int l,int r,int L,int R,int p){
42         if(L<= l && R>=r) return s[p];
43         else if(l>R || r<L) return se;
44         else{
45             int mid = l+r>>1;
46             return merge(query(l,mid,L,R,p<<1), query(mid+1,r,L
47                             ,R,p<<1|1));
48         }
49     };
50
51
52 int main(){
53     ios::sync_with_stdio(false);
54     cin.tie(0);
55     int n,Q;
56     cin>>n>>Q;
57     for(int i = 1;i<=n;i++){
58         cin>>da[i];
59     }
60     seg_tree st;
61     st.build(1,n,1);
62     int L,R;
63     for(int i = 1;i<=Q;i++){
64         cin>>L>>R;
65         S ans = st.query(1,n,L,R,1);
66         cout<<ans.mx - ans.mi<<endl;
67     }
68 }

```

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, \dots, 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:

```
1 #include <bits/stdc++.h>
2 using namespace std;
3 #define MAX_SIZE 500001
4 #define P_INF 1'000'000'000
5
6 int euler_map[MAX_SIZE];
```

```

7  int euler_map_inv[MAX_SIZE];
8  int euler_path[MAX_SIZE<<1];
9  int map_path_ind[MAX_SIZE];
10 bool vis[MAX_SIZE];
11
12 struct seg_tree
13 {
14     int s[MAX_SIZE<<3];
15     void build(int l,int r,int p){
16         if(l==r){
17             s[p] = euler_path[l];
18             return;
19         }
20         int mid = l+r>>1;
21         build(l,mid,p<<1);
22         build(mid+1,r,p<<1|1);
23         s[p] = min(s[p<<1],s[p<<1|1]);
24     }
25
26     int query(int l,int r,int L,int R,int p){
27         if(L<=l && R>=r) return s[p];
28         else if(l>R || r<L) return P_INF;
29         else{
30             int mid = l+r>>1;
31             return min(query(l,mid,L,R,p<<1),query(mid+1,r,L,R,
32                 p<<1|1));
33         }
34     };
35
36     seg_tree st;
37
38     int main(){
39         int n,m,root;
40         cin>>n>>m>>root;
41         vector adj(n+1,vector<int>(0));
42         for(int i = 1;i<=n-1;i++){
43             int u,v;
44             cin>>u>>v;

```

```

45         adj[u].push_back(v);
46         adj[v].push_back(u);
47     }
48     vis[root] = true;
49     euler_map[root] = 1;
50     euler_map_inv[1] = root;
51     int dfs_order = 1;
52     int ep = 0;
53     auto dfs_euler = [&](auto self,int root)->void{
54         vis[root] = true;
55         euler_path[++ep] = euler_map[root];
56         map_path_ind[euler_map[root]] = ep;
57         for(int v : adj[root]){
58             if(vis[v]==false){
59                 euler_map[v] = ++dfs_order;
60                 euler_map_inv[dfs_order] = v;
61                 self(self,v);
62                 euler_path[++ep] = euler_map[root];
63                 map_path_ind[euler_map[root]] = ep;
64             }
65         }
66     };
67     dfs_euler(dfs_euler,root);
68     st.build(1,2*n-1,1);
69     for(int i = 0;i<m;i++){
70         int u,v;
71         cin>>u>>v;
72         u = map_path_ind[euler_map[u]];
73         v = map_path_ind[euler_map[v]];
74         if(u>v) swap(u,v);
75         cout<<euler_map_inv[st.query(1,2*n-1,u,v,1)]<<endl;
76     }
77 }

```

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 $\text{top}(u)$ 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 $x \rightarrow y$ 的路径中经过轻边的个数不超过 $O(\log(n))$ 。

证明:

假设 $u \rightarrow v$ 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 $\text{size}(T_{v'}) \geq \text{size}(T_v)$ 。

因此 $\text{size}(T_u) \geq 2\text{size}(T_v)$ 。

假设 l 表示 $u \rightarrow v$ 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 $x \rightarrow y$ 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 $x \rightarrow y = x \rightarrow r \rightarrow y$.

根据引理 10, $x \rightarrow r, r \rightarrow y$ 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:

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

$\text{top}(u)$ 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:

```

1  #include <bits/stdc++.h>
2  using namespace std;
3
4  const int MAX_SIZE = 5e5+1;
5  int size_tr[MAX_SIZE];
6  int h[MAX_SIZE];
7  int top[MAX_SIZE];
8  int fa[MAX_SIZE];
9  int heavy_son[MAX_SIZE];
10 vector adj(MAX_SIZE, vector<int>(0));
11
12 void dfs_1(int root){
13     int sz = 1;
14     for(int v:adj[root]){
15         if(v==fa[root]) continue;
16         fa[v] = root;
17         h[v] = h[root]+1;
18         dfs_1(v);
19         sz+= size_tr[v];
20         if(heavy_son[root]==0 || size_tr[v]>size_tr[heavy_son[
21             root]]){
22             heavy_son[root] = v;
23         }
24     }
25     size_tr[root] = sz;
26 };
27 void dfs_2(int root,int tp){
28     for(int v:adj[root]){
29         if(v==fa[root]) continue;
30         if(v==heavy_son[root]){
31             top[v]= tp;

```

```

32         dfs_2(v, tp);
33     }
34     else{
35         top[v] = v;
36         dfs_2(v, v);
37     }
38 }
39 };
40 int lca(int u, int v){
41     while(top[u] != top[v]){
42         if(h[top[u]] < h[top[v]]) swap(u, v);
43         u = fa[top[u]];
44     }
45     return h[u] < h[v] ? u : v;
46 };
47 int main(){
48     h[0] = -1;
49     ios::sync_with_stdio(false);
50     cin.tie(0);
51     int n, m, root;
52     cin >> n >> m >> root;
53     int u, v;
54     for(int i = 0; i < n - 1; i++){
55         cin >> u >> v;
56         adj[u].push_back(v);
57         adj[v].push_back(u);
58     }
59     top[root] = root;
60     dfs_1(root);
61     dfs_2(root, root);
62     for(int i = 0; i < m; i++){
63         cin >> u >> v;
64         cout << lca(u, v) << "\n";
65     }
66 }

```

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 $u \rightarrow v$ 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。

```
1 #include <bits/stdc++.h>
2 using namespace std;
3 #define MAX_SIZE 200'001
4 #define N_INF 0
5
6 int n;
7 int id_to_dfn[MAX_SIZE];
8 int dfn_to_id[MAX_SIZE];
9 int val[MAX_SIZE];
10 int dfs_order[MAX_SIZE];
11 int fa[MAX_SIZE];
12 int heavy_son[MAX_SIZE];
13 int size_tr[MAX_SIZE];
14 int top[MAX_SIZE];
15 int h[MAX_SIZE];
16
```



```

17 auto find_data(int i){
18     return val[dfn_to_id[i]];
19 }
20
21 struct seg_tree
22 {
23     int s[MAX_SIZE<<2];
24     void build(int l,int r,int p){
25         if(l==r){
26             s[p] = find_data(l);
27             return;
28         }
29         int mid = l+r>>1;
30         build(l,mid,p<<1);
31         build(mid+1,r,p<<1 | 1);
32         s[p] = max(s[p<<1], s[p<<1|1]);
33     }
34     int query(int l,int r,int L,int R,int p){
35         if(L<=l && R>=r) return s[p];
36         else if(l>R || r<L) return N_INF;
37         else{
38             int mid = l+r>>1;
39             return max(query(l,mid,L,R,p<<1), query(mid+1,r,L,R
40                 ,p<<1 | 1));
41         }
42     void update(int l,int r,int index,int val,int p){
43         if(l==r && l==index) s[p] = val;
44         else if(l>index || r<index) return;
45         else{
46             int mid = l+r>>1;
47             update(l,mid,index,val,p<<1);
48             update(mid+1,r,index,val,p<<1 | 1);
49             s[p] = max(s[p<<1], s[p<<1 | 1]);
50         }
51     }
52 };
53
54 seg_tree st;

```

```

55 int query_on_heavy_chain(int u,int v){
56     if(id_to_dfn[u]>id_to_dfn[v]) swap(u,v);
57     return st.query(1,n,id_to_dfn[u],id_to_dfn[v],1);
58 }
59
60 int merge_query(int u,int v){
61     int ans = 0;
62     while(top[u] !=top[v]){
63         if(h[top[u]] < h[top[v]]) swap(u,v);
64         ans = max(ans,query_on_heavy_chain(u,top[u]));
65         u = fa[top[u]];
66     }
67     ans = max(ans,query_on_heavy_chain(u,v));
68     return ans;
69 };
70 vector adj(0,vector<int>(0));
71 void dfs_1(int root){
72     size_tr[root] = 1;
73     for(int v:adj[root]){
74         if(v==fa[root]) continue;
75         fa[v] = root;
76         h[v] = h[root]+1;
77         dfs_1(v);
78         size_tr[root] += size_tr[v];
79         if(heavy_son[root]==0 || size_tr[heavy_son[root]]<
            size_tr[v]) heavy_son[root] = v;
80     }
81 };
82 int main(){
83     ios::sync_with_stdio(false);
84     cin.tie(0);
85     cout.tie(0);
86     // freopen("Code_18Test.txt","r",stdin);
87     // freopen("Code_18out.txt","w",stdout);
88     // clock_t begin_time = clock();
89     int q;
90     cin>>n>>q;
91     for(int i = 1;i<=n;i++){
92         cin>>val[i];

```

```

93     }
94     adj.resize(n+1);
95     for(int i = 1;i<=n-1;i++){
96         int u,v;
97         cin>>u>>v;
98         adj[u].push_back(v);
99         adj[v].push_back(u);
100    }
101    int root = 1;
102    h[root] = 0;
103    h[0] = -1;
104
105    dfs_1(root);
106    int dfs_or = 1;
107    dfs_order[root] = 1;
108    top[root] = root;
109    id_to_dfn[root] = 1;
110    dfn_to_id[1] = root;
111    auto dfn_update = [&](int node){
112        dfs_order[node] = ++dfs_or;
113        id_to_dfn[node] = dfs_order[node];
114        dfn_to_id[dfs_order[node]] = node;
115    };
116    auto dfs_2 = [&](auto self,int root)->void{
117        if(heavy_son[root]!=0){
118            dfn_update(heavy_son[root]);
119            top[heavy_son[root]] = top[root];
120            self(self,heavy_son[root]);
121        }
122        for(int v:adj[root]){
123            if(v==fa[root] || v==heavy_son[root]) continue;
124            dfn_update(v);
125            top[v] = v;
126            self(self,v);
127        }
128    };
129    dfs_2(dfs_2,root);
130    st.build(1,n,1);
131    for(int i = 1;i<=q;i++){

```

```
132     int type;
133     cin>>type;
134     if(type==1){
135         int u;
136         cin>>u>>val[u];
137         st.update(1,n,id_to_dfn[u],val[u],1);
138     }
139     else{
140         int u,v;
141         cin>>u>>v;
142         cout<<merge_query(u,v)<<"\n";
143     }
144 }
145 // cerr<<(clock()-begin_time)/1000.0<<"\n";
146 }
```

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](#)