**第一章 树及其应用**

中国共产党第十九次全国代表大会（简称十九大）于2017年10月18日至10月24日在北京召开。本次会议由党的全国代表大会选举产生了新一届中央委员会和中央纪律检查委员会，由中央委员会决定中央军事委员会组成成员，通过中央书记处成员，选举产生中央政治局、中央政治局常务委员会和中央委员会总书记。

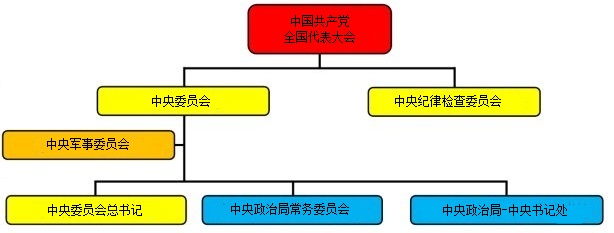


图1.1 中共中央组织结构图

我们看中共中央的组织结构图，如图1.1，自上而下形成了一种层次结构，从上往下看，类似一棵倒着生长的树，最上面是树根，最下面是树叶，而且整个结构图没有形成闭环，这样的结构，我们称之为树。树结构在现实生活中十分常见，比如家族的族谱，公司组织结构等等。

**§1.1 树的相关概念及其性质**

**一、树的概念**

我们现在所提到的所谓“树”，是数据结构中的一种。树可以分为有根树与无根树两种。有根树有一个确定的根节点。无根树的根不确定，也就是说任何一个点都可以作为该树的根。

对于有根树，必须明确树的根，“中国共产党全国代表大会”就是图1.1的“**根**”。树根是整棵树的起点，从树根开始，通过树枝（称为**树边**）向下可以逐步扩展出很多其他的**子节点**，其中有些子节点还能继续向下扩展延伸，这些节点成为树的“**中间节点**”，图1.1的“中央委员会”。有些子节点不能再向下扩展了，我们称为树的“**叶节点**”或简称“**叶子**”，图1.1 的“中央纪律委员会”等。

无根树的例子也很多。例如，n-1条公路将n个城市联通在一起，若将城市看成点，将公路看成边，则这个城市公路网络就是一棵树，但是这棵树没有明确根，因此他是一棵无根树。

对于有根树而言，除了根、叶子等，还有一些概念。

**父亲和儿子**：父亲和儿子是相对的，若某节点能往下生成一些节点，则该节点为父亲，被产生的节点为儿子。

**祖先**：从某节点出发，顺着某条路径往上走到根节点，这条路径上经过的所有节点，都是该节点的祖先。

**兄弟**：具有相同父亲的所有节点，互相称为兄弟。

**节点的度**：节点儿子的个数，称为节点的度。显然，叶子的度为0。

**节点层次（或节点深度）**：根节点处在第1层，根的子节点则为第2层，以此类推。也有一些场合为了方便会把根的深度定义为0。

**树的高度（或树的深度）**：叶节点的最大深度为树的高度（或深度）。

**堂兄弟：**同一层次的所有节点，互相称为堂兄弟。

**子树**：从某个节点出发，往下扩展所产生的图，称为该节点为根的子树。子树节点的个数称为**子树的大小**。如图1.2“中央委员会结构图”就是图1.1的一个子树。

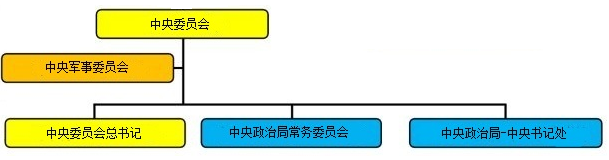


图1.2 中央委员会组织结构图

**森林：**独木不成林，两棵以上的树组成的集合称为森林。

容易发现，大量定义都是针对有根树的。为了便于分析，很多时候我们遇到无根树问题时，都会任意选择一个节点（一般选择编号为1的点）当成有根树来方便讨论。

**二、树的性质**

1．对于有根树，除根节点外，其余节点有且仅有一个父节点。

2．n个节点的树有且仅有n−1条边。

3．树是不存在环的连通图。

4．树中任意两个节点之间有且仅有一条简单路径。

**因此，我们可以利用树的基本性质来判断某问题的数据结构是不是一棵树结构**

**例1.1** 求树中每个点的儿子个数，假设节点1为树的根。

输入：第1行一个整数n表示树的节点个数，接下来n-1行，每行两个整数x, y，表示x为y的父节点。

输出：n个整数，第i个整数为节点i的儿子个数

有根树以如下形式给出：

**解法：**x是y的父亲，则对x的儿子加1，简单统计即可。

scanf("%d", &n);

for (int i = 1, x, y; i < n; ++i) {

scanf("%d%d", &x, &y);

++cnt[x];

}

for (int i = 1; i <= n; ++i) printf("%d ", cnt[i]);

**例1.2** 求树中每个点的儿子个数，假设节点1为树的根。

输入：第1行一个整数n表示树的节点个数，接下来n-1行，每行两个整数x, y,表示x和y节点之间有一条树边（**不保证x是y的父亲**）。

输出：n个整数，第i个整数为节点i的儿子个数

**解法：**若某个节点有一条边相连，则先对儿子个数加一，最后，树中每个节点（根节点除外）的儿子数等于与它相连的边数减1。

scanf("%d", &n);

for (int i = 1, x, y; i < n; ++i) {

scanf("%d%d", &x, &y);

++cnt[x]; ++cnt[y];

}

++cnt[1]; //根节点特殊处理

for (int i = 1; i <= n; ++i) printf("%d ", cnt[i] - 1);

**§1.2 树的存储及遍历****法**

**一、 有根树的双亲表示法**

除根节点以外其他节点有且只有一个父节点，因此，我们可以把每一条树边存储在其子节点上，记录形式为：i节点的父亲是j节点。如图1.3：

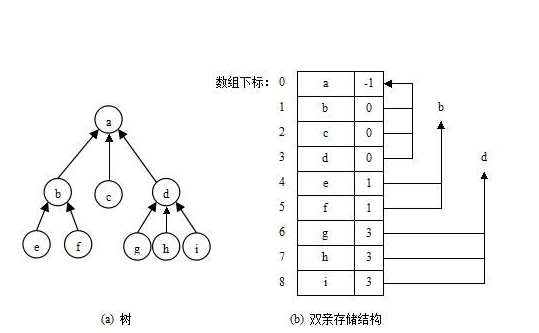


图1.3 树的父亲表示法存储示意图

上述存储含义为：father[a]=-1(表示不存在),father[b]=a ,father[b]=a , father[b]=a father[c]=b,……

双亲表示法存储结构的实现代码:

int fa[MAXN];

void link(int x, int y) { // y是x的父亲

fa[x]=y;

}

**二、 有根树的图存储法**

树其实是一种特殊的图，可以把一条树边看作一条父亲指向儿子(或者儿子指向父亲)的有向边。我们采用邻接矩阵或邻接表来存储这棵树。具体做法在第四章将会有详细介绍，这里只做简单说明。

**邻接矩阵**：我们可以使用一个n \* n的bool数组mp，mp[x][y]为true则表示x, y之间有边，为false则表示两个点之间没有边。

邻接矩阵存储代码：

bool mp[MAXN][MAXN];

void link(int x, int y) { // 连一条由x指向y的边

mp[x][y] = 1;

}

**邻接表**：同样，我们可以采用邻接表来存储一个点连出的多条树边。如图1.4：

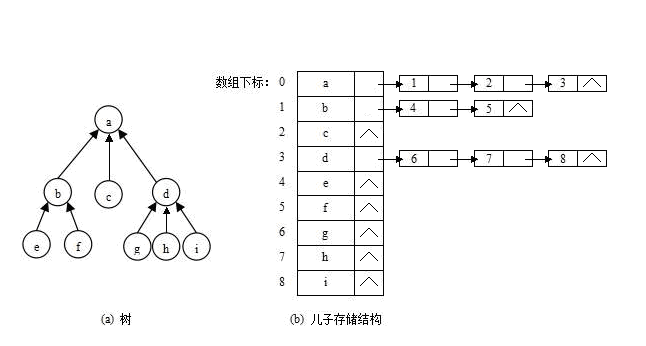


图1.4 树的邻接表存储示意图

上图表示下标为0的a节点，有指向b、c、d三个节点（编号为1,2,3）的边，这三条边通过链表来存储。其他节点类似。

邻接表存储代码：

int m,;

int fi[MAXN]; //fi存储节点的儿子个数

int to[MAXN]; //to存储节点的具体每个儿子

int ne[MAXN]; //ne是节点儿子的链接，指向该节点的下一个儿子

void link(int x, int y) { // 连一条由x指向y的边

to[++m] = y; ne[m] = fi[x]; fi[x] = m;

}

使用c++的模板**std::vector<int>**，我们可以更方便的存储。

实现代码：

vector<int> g[MAXN];

void link(int x, int y) { // 连一条由x指向y的边

g[x].push\_back(y);

}

**三、 无根树的图存储法**

因为无根树没有确定的根，所以一条边相连的两个点也没有明确的父子之分。我们可以用类似存储有根树的图存储法来存储无根树，与有根树不同的是，每连接一条边，我们要存相应的两条边。如下图1.5。

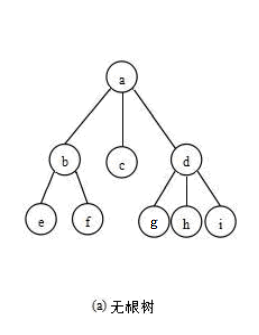
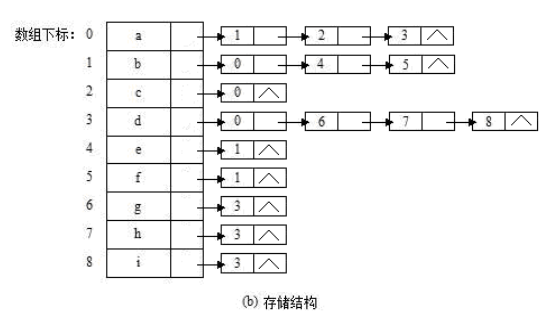


图1.5 无根树的邻接表存储法示意图

**邻接表存储结构的实现代码：**

int m, fi[MAXN], to[MAXN \* 2], ne[MAXN \* 2];

void link(int x, int y) {

to[++m] = y; ne[m] = fi[x]; fi[x] = m;

to[++m] = x; ne[m] = fi[y]; fi[y] = m;

}

**注意**到我们的to和ne数组开了两倍的空间。这是因为边数实际上为2(n-1)。这是处理树的问题中的一个易错点。

**四、树的深度优先遍历**

遍历就是按某种顺序依次访问图中的每一个节点。深度优先遍历是遍历的一种方法，对树的深度优先遍历方法是：访问当前节点→深度遍历当前节点的第一个儿子→深度遍历当前节点的下一个儿子，直到所有节点都访问为止。

图1.6标识了深度优先遍历的过程。具体遍历路径如下：

* 1. 先访问根节点a。
  2. 沿着a存储的第一条边（因为是链表存储，所以是读入的最后一条边）走到d，访问d。
  3. 沿着a的父子边遍历i，访问i。
  4. i没有孩子，则i遍历完毕，返回到d。
  5. 沿着d的父子边遍历h，访问h。
  6. h没有孩子，则h遍历完毕，返回到d。
  7. 沿着d的父子边遍历g，访问g。
  8. g没有孩子，则g遍历完毕，返回到d。
  9. d的所有儿子均已遍历，则d遍历完成，返回到a。
  10. …… 以此类推,遍历以c为根的子树, 返回到a。
  11. …… 以此类推,遍历以b为根的子树, 返回到a。
  12. a的所有儿子均已遍历，树遍历完成。

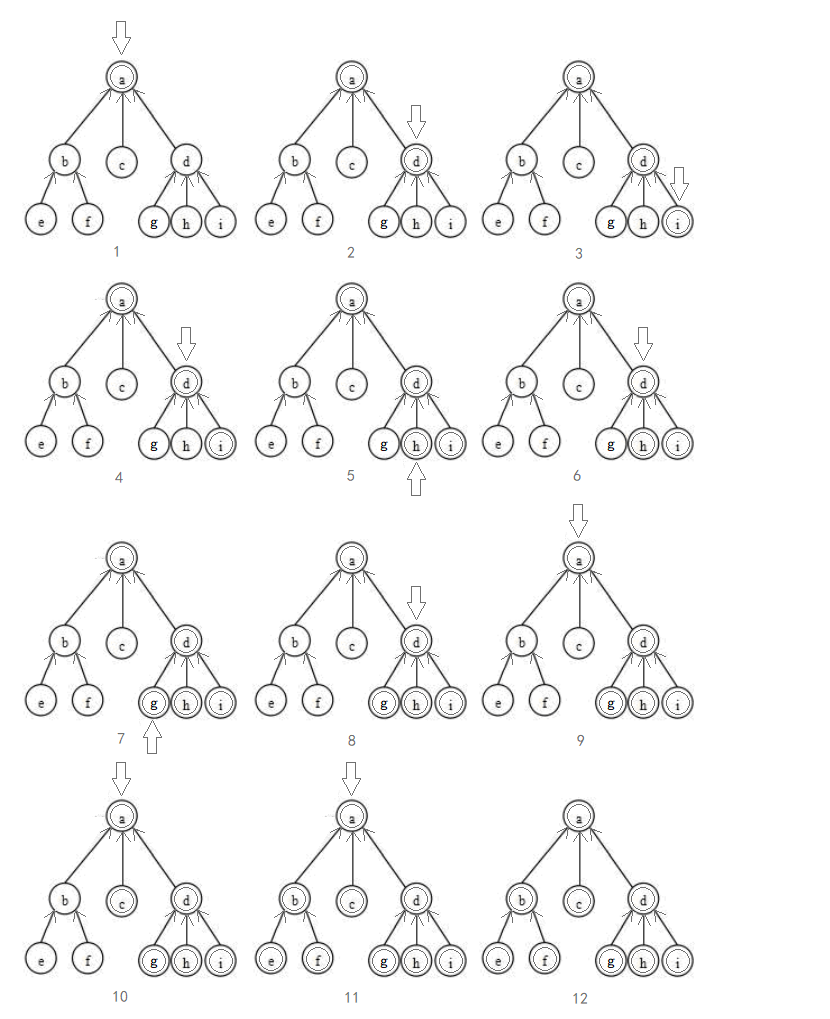


图1.6 树的深度优先遍历过程

有根树的深度优先遍历代码：

int m, fi[MAXN], to[MAXN \* 2], ne[MAXN \* 2];

void dfs(int x) {

visit(x); // 在访问x的儿子前做一些事

for ( int i = fi[x]; i; i = ne[i] ) dfs(to[i]); // 访问x的儿子

…. // 在访问x的儿子后做一些事

}

dfs(root) //主程序调用。

无根树的深度优先遍历代码：

void dfs(int x, int fa) {

visit(x); // 在访问x的儿子前做一些事

for ( int i = fi[x]; i; i = ne[i] ) if (to[i] != fa) dfs(to[i], x)

…. // 在访问x的儿子后做一些事

}

dfs(root, 0) //主程序调用，一般我们将根的父亲设置为0或-1。

**例1.3** 求树中每棵子树的大小以及每个节点的深度（假设节点1为根）。

输入：第一行一个整数n，表示树中节点个数，然后n-1行每行两个数x和y，表示节点x和节点y之间有一条边，但不保证x是y的父亲。

输出：共n行，每i两个数，分别表示以节点i为根的子树大小和该节点i的深度。

**解法**：每个点的子树大小为它的所有儿子的子树大小之和再加1。每个点的深度为它父亲的深度再加1。

int n, si[MAXN], de[MAXN];

int tot, fi[MAXN], to[MAXN \* 2], ne[MAXN \* 2];

void link(int x, int y) {

to[++tot] = y; ne[tot] = fi[x]; fi[x] = tot;

}

void getsi(int x, int fa) {

si[x] = 1; de[x] = de[fa] + 1;

for (int i = fi[x]; i; i = ne[i]) if (to[i] != fa) dfs(to[i], x), si[x] += si[to[i]];

}

int main() {

scanf("%d", &n);

for (int i = 1, x, y; i < n; ++i) {

scanf("%d%d", &x, &y);

link(x, y); link(y, x);

}

getsi(1, 0);

for (int i = 1; i <= n; ++i) printf("%d %d\n", si[i], de[i]);

return 0;

}

五、树的几种存储方式的优劣对比（主要针对c++）

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 双亲表示法 | 邻接矩阵 | 邻接表 | vector |
| 优点 | 理解容易，实现简单，存储消耗空间最小 | 容易理解，实现简单。 | 消耗空间小。 | 实现简单，消耗空间较小。 |
| 缺点 | 储存的信息量太少，很多操作无法完成。 | 消耗空间大。n-1条边却使用了n2的空间，浪费明显，大部分题目无法接受 | 较难理解，实践中在特定情况下相比vector速度较慢 | 如果要熟练运用需要深入了解c++ STL的实现。 |

在程序设计竞赛中，对树的存储通常采取后两种方法居多。若没有特殊说明，下文默认使用邻接表存储结构

**§1.3最近公共祖先（**LCA**）**

我们之前了解到“**祖先**”概念，若两个节点的祖先相同，则叫该节点的**公共祖先**。距离两个节点最近的公共祖先称为**最近公共祖先**（Lowest Common Ancestors），简称LCA。

显然，两个点的LCA只有一个，且一定是两个点到根的路径中重复部分最下端的点。

**算法一：采取两个点逐渐向根移动的方法，求出LCA。**

具体步骤如下：

1. 求出每个节点的深度。

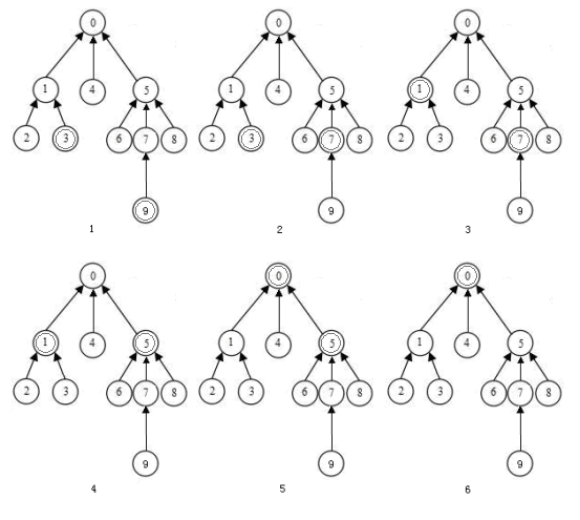
2. 询问的两个节点是否重合，若重合，则LCA已求出。

3. 否则，则选择两个点中深度较大一个，并移动到它的父亲。

重复执行2，3步。

图1.7，展示了算法一的执行流程。

图1.7 算法一求LCA示意图



执行流程如下：

1.两个指针位于3号节点与9号节点。

2.此时9号节点深度较大，向上移动到它的父亲7号节点。

3.此时两个节点深度一样，移动3号节点的指针到它的父亲1号节点。

4.此时7号节点深度较大，向上移动到5号节点。

5.此时两个节点深度一样，移动1号节点到根节点0号节点。

6.此时5号节点深度较大，向上移动到0号节点，两个指针重合，0号点为LCA。

算法代码：

int de[MAXN], fa[MAXN];

void dfs(int x, int fath) { // 遍历这棵树并求出每个点的深度和每个点的父亲

fa[x] = fath; de[x] = de[fath] + 1;

for (int i = fi[x]; i; i = ne[i])

if (to[i] != fath) dfs(to[i], x);

}

int getlca( int x, int y) { // 求LCA

while ( x != y ) {

if ( de[x] > =de[y] ) x = fa[x];

else y = fa[y]; //将深度较大的点的指针上移

}

return x;

}

**算法一：**求LCA的时间复杂度与两点间的距离有关，极限情况可达到O(n)。虽然该算法时间复杂度较高，但该算法也是有一定的应用价值的。

其一它实现简单，可在算法竞赛快速实现和正确的程序对拍。

其二随机产生的树的高度期望是log2n级别的，有些时候树是随机的或者树高不太高，我们也可以使用该算法。

其三是这个算法允许树动态改变。我们只需要知道每个点的父亲和深度，就可以方便的求出两个点的LCA（当然如果更进一步的我们不知道点的深度，也可以先从一个点走到根，把路径上的点都标记了，再从另一个点向根走走到标记的点为止）。另一个可以处理动态情况的数据结构是**动态树**，但它实现复杂而且常数因子较大。所以在这个特定情况下，该算法具有不可替代的作用。

**算法二：倍增法求LCA。**本算法是对算法一中一步步走的改进，核心实质是让两个节点每次向上都2的倍数步，具体操作如下：。

第一步：求出倍增数组anc[MAXN][MAXLOG]。其中MAXLOG是最小的满足2x ≥MaxDeep（最深节点深度）的x，anc[i][j]为节点i向上走2 j步后能走到的节点。我们规定根节点的父亲是它自己，这样根节点往上走还是在根节点。对于j=0，anc[i][j]就是节点i的父亲。对于j>0，anc[i][j]等于anc[anc[i][j-1]][j-1]（即节点i往上走2 j-1步后再往上走2 j-1步）。

第二步，把两个点移到同一深度。假设要求LCA(x,y)，不失一般性，令de[x]>=de[y]（否则我们交换x，y），先让x往上走de[x]-de[y]步。我们将这个差表示成二进制，就可以通过倍增数组往上走2的幂次步（即对于二进制为1的第i位（规定二进制数最左边为第0位），要往上走2i步，即调用x=anc[x][i]），那么可以在log的时间复杂度内到达目标深度。或者说，类似的，我们也可以从大往小扫描i，每次如果anc[x][i]深度不小于y，我们就跳x。容易发现两种做法效果是一样的。读者可以根据自己的喜好选择。

第三步，求出LCA。假设x与y向上走最小的L步后是同一节点，也就意味着，x与y向上走最大的L-1步，也是不同的节点。我们可以从大到小枚举往上走2i步，如果当前x与y向上走2i步后为同一点，则停止，否则一起往上走。这样，我们就能在log的时间复杂度内使x与y都向上走L-1步。根据倍增数组是2的幂次这个特性，这样的做法可以看成一个通过二分法来求解走最大L-1步的过程。用这样的方法从大到小决策MAXLOG次，直到完全不能向上走了为止，我们再让x与y各向上走一步，则为LCA。

算法代码:

int de[MAXN], anc[MAXN][MAXLOG];

void dfs(int x, int fa) {

de[x] = de[fa] + 1; anc[x][0] = fa;

for (int i = 1; i < MAXLOG; ++i) anc[x][i] = anc[anc[x][i - 1]][i - 1];

for (int i = fi[x]; i; i = ne[i]) if (to[i] != fa) dfs(to[i], x);

}

int getlca(int x, int y) {

if (de[x] < de[y]) swap(x, y);

for (int i = MAXLOG - 1; ~i; --i) if (de[anc[x][i]] >= de[y]) x = anc[x][i];

if (x == y) return x;

for (int i = MAXLOG - 1; ~i; --i) if (anc[x][i] != anc[y][i]) x = anc[x][i], y = anc[y][i];

return anc[x][0];

}

**算法三：转化为欧拉序列上的 RMQ 问题，采用ST算法。**

**欧拉序列**：每经过一次节点，都进行统计一次DFS 序列。

**RMQ**：指一类查询连续区间最小（大）值的问题

**ST算法**：用来求解 RMQ 问题的快速方法。

**分析**：±1RMQ LCA 转化而来的 RMQ 问题其实是个特殊的 ±1RMQ 问题，有O(N) - O(1) 的算法，但是因为常数巨大在竞赛中一般不用，其理论价值大于实际价值。

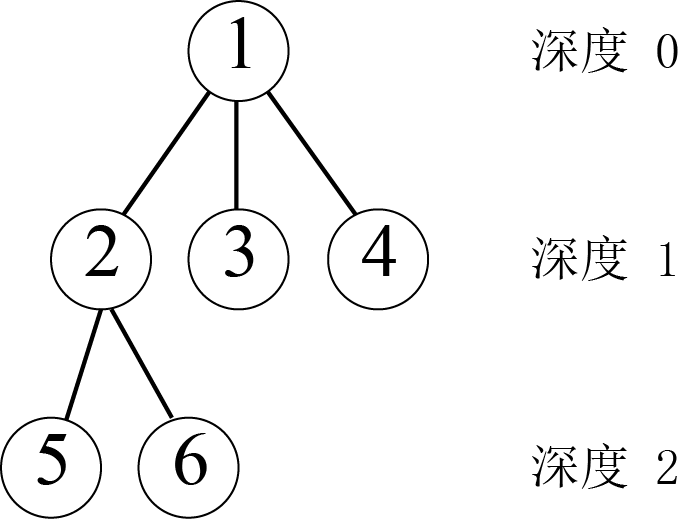


图1.7 求树的欧拉序列示例图

图1.7所示的树的欧拉序列和深度优先遍历序列（dfs序）分别为:

1. 欧拉序列 1 2 5 2 6 2 1 3 1 4 1
2. dfs序列 0 1 2 1 2 1 0 1 0 1 0

每次求 LCA(x,y) 时，只要知道它们在欧拉序列中的位置，假设分别为 pos[x], pos[y]，那么它们的 LCA 就是 [pos[x], pos[y]] 区间中深度最小的点。只要在深度序列上用 ST算法做RMQ 即可快速求出。假如 x,y 在欧拉序列上不止出现一次，只要任选其中一次来计算即可，因为 [pos[x], pos[y]] 区间中的点即使不在 x 到 y 的路径上，也一定在 x子树或者 y子树内，深度大于x,y的深度，不影响结果。树的欧拉序列中每条边出现两次（从上往下一次，从下往上一次），所以时间复杂度为O(N)。

该算法效率:预处理时间复杂度为O(N\*log2N)，单次询问时间复杂度 O为(1)，总时间复杂度为 O(N\*log2N + Q)，空间复杂度为O(N\*log2N)

**算法四：Taran算法**

1. DFS 整棵树。每个节点 x 一开始属于只有该节点本身的集合 Sx（并查集）
2. DFS(x)时，每次访问完子树y时，把 Sy 合并到 Sx
3. x 的所有子节点访问完，标记 x 为已经访问
4. 遍历所有关于 x 的询问 (x, y)，如果 y 已被访问，则这个询问的答案为并查集中的 Find(y)

算法伪代码：

function DFS(x)

for all y ∈ Child[x] do

DFS(y)

Union(y,x)

end for

Vis[x] ← True

for all y ∈ Query[x] do

if Vis[y] then

Ans(x, y) ← Find(y)

end if

end for

end function

算法时间复杂度为O((N + Q)α(N))。这是一个离线算法，仅用于时间限制极其严格、倍增算法会超时的情况。

因查集的实现在第三章讲述，因此这里不做详细描述，操作时，需要小心栈溢出。

**例1.4** 求树上两个点之间的距离。距离定义为树上两个点之间唯一简单路径上边的条数。

**解法**：可以先求出这两个点的LCA，则距离等于de[x]+de[y]-2\*de[LCA]。

算法代码：

int getdis( int x,int y) {

return de[x]+de[y]-2\*de[getlca(x, y)];

}

**总结：求LCA算法的比较（主要针对c++，默认**N个点，Q次询问**）**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 暴力法 | 倍增法 | 转RMQ问题 | Tarjan |
| 时间复杂度 | O(N \* Q) | O((N + Q) log2N) | O(N\*log2N + Q) | O(N + Q) |
| 优点 | 容易理解和实现，存储空间小。支持动态的树。 | 容易理解和实现，支持动态的树（增加叶子）。询问次数少的情况下，比好适合。 | 单次询问是O(1)的。在询问次数多时。该算法较好。 | 时间复杂度最优。一般适用离线情况。 |
| 缺点 | 最坏情况下时间复杂度很高。 | 占用空间较大。时间复杂度直接与询问次数相关。 | 占用空间较大。算术实现相比倍增法略复杂。 | 算法常数因子大，不适用在线询问。实现也不易，几乎完全被替代。 |

**§1.4 树的简单应用**

**一、 括号序列与树结构的相互转化**

观察以下这个括号序列：（（（）（））（）（（）（）（）））

在这个括号序列中，左右括号一一对应。

括号层层嵌套，如果把一对括号看成一个节点，其直接套住的括号对看作是其子节点，则这个结构就是一棵树。

上图的括号序列在层次上表示为：



那么。对应到树结构上，如图1.8。

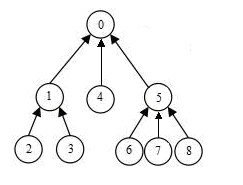


图1.8 括号序列转化为树的结构图

**解法：**我们可以根据读取括号序列，用递归的方法把它转化为树。

相关代码：

int f[MAXN], now;

char s[MAXN\*2];

void solve(int fa) {

while (s[now] != ’)’) { // 扫过这层括号所嵌套的内容，直到遇到右括号为止

++now; // 发现了新的一层括号

link(fa, ++n); // 建立树边

solve(n); // 递归处理新的一层括号的内容

}

++now;

}

同样，我们也能通过深度优先遍历输出一棵树的括号序列。

每向下遍历到一个节点，就在序列末端增加一个左括号（表示开始进入这个节点子树的范围）。当访问完一个节点的子树，在回溯到它的父亲节点前，在序列末端加上一个右括号（表示这个节点的子树的范围已完成）。

相关代码：

void dfs(int x) {

putchar(‘(‘); //开始进入节点的子树的范围

for (int i = fi[x]; i; i = ne[i]) dfs(to[i]); //递归遍历

putchar(‘)’); //结束该节点子树的范围

}

**二、求树的直径**

树中两点间的不重复经过的边和点道路称为两点的路径，路径的长度（路径上所经边的长度和）称为两点的距离。

我们知道圆的直径是一个圆的最长的一条弦。那么，树的直径是树中两点间的最长路径。通常，我们用一个无序点对(x, y)表示一条树的直径。树的直径可能有很多条。

树的直径满足如下性质：

1.若有多条直径，则所有的直径之间皆有公共点。

证明：如果存在两条直径没有公共点，我们一定可以用这两条直径的四个端点中的某两个构造出一条更长的直径。

2.直径的两个端点一定是叶子。

证明：如果存在一个端点不是叶子，我们可以取那个端点的子节点，代替那个当前点作为直径，则可构造出一条更长的直径。

3.树中距离直径最远的点，至少有一个是该直径的两个端点。

证明：如果不是，那么我们一定可以用与之距离最远的点更新直径。

4.对树上任意一个点，与之距离最远的每一个点，至少有一个直径的端点。

证明：设与之距离最远的点为x，任取一条直径(u, v)，则易证(u, x)或(v, x)必定至少有一个是直径。

**解法一**：通过两次遍历找出树的一条直径。

第一次遍历，找出距离某个节点（例如根节点）最远的一个点x。

第二次遍历，找出距离节点x最远的一个点y。

x到y的简单路径，即为树的一条直径。

注意：为了找出距离某个点最远的点，这棵树应该看作无根树，一个节点连向父亲的边也要存入邻接表中！

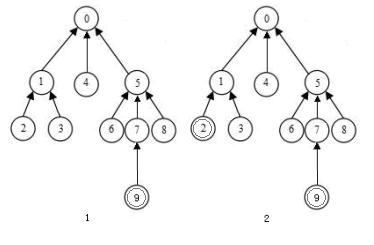


图1.9 求树的直径示例图

求图1.9这棵树的一条直径。步骤如下：

1.先找到距离根节点0最远的点（之一），即9号节点。

2.找到距离9号节点最远的点（之一），即2号节点，2到9的路径即为直径。

直径为2-1-0-5-7-9，一般情况下，我们只要记录下直径的两个端点即可。

相关代码：

int x, y, de[N];

void getde(int x, int fa) {

de[x] = de[fa] + 1;

for (int i = fi[x]; i; i = ne[i]) if (to[i] != fa) getde(to[i], x);

}

void work() {

getde(1, 0);

x = 1; for (int i = 2; i <= n; ++i) if (de[i] > de[x]) x = i;

getde(x, 0);

y = 1; for (int i = 2; i <= n; ++i) if (de[i] > de[y]) y = i;

printf("%d %d\n", x, y);

printf("%d\n", de[y]);

}

**解法二**：通过求LCA，找出树的一条直径。

显然，一条直径上的所有点有一个共同的LCA。在DFS的过程中对于每一个点，我们考虑以它为LCA的可能的直径。即，我们只需要维护出以每个点为顶端的最长链和次长链。然后用最长链加上次长链更新直径即可。

相关代码：

int len = 0;

int dfs(int x, int fa) {

int mx = 0, mmx = 0;

for (int i = fi[x]; i; i = ne[i]) if (to[i] != fa) {

int tmp = dfs(to[i], x);

if (tmp > mx) mmx = mx, mx = tmp;

else if (tmp > mmx) mmx = tmp;

}

len = max(len, mmx + mx);

return mx;

}

**例1.5** 给定一棵树，对于每一个点，输出离它最远的点到它的距离。

**解法**：根据直径的性质4，我们只要求出任意一条直径，对于每个点，比较两个端点哪个距离该点较远即可。

**例1.6** 给定一棵树，动态加入叶子，输出每次加入叶子以后树的直径长度，n≤105。

**解法**：使用求LCA的倍增法。容易发现，加入叶子的时候，我们可以方便的维护出anc数组和de数组。也就是说，我们可以在O(log2n) 的复杂度内求出两点间的距离。根据直径的性质3，我们维护加入叶子前的任意一条直径（x，y）。每当加入叶子之后， 新产生的距离最远的点对间的距离不会超过max(dis(x, new\_leaf), dis(y, new\_leaf))。亦即，我们只需要用(x, new\_leaf) (y, new\_leaf)去更新(x, y)即可。

**三、求树的中心**

圆的中心就是圆心，树的中心是什么呢？

树的中心类似于圆的圆心，树的中心是该点到树中其他点的距离，最远距离最小。同样，树的中心可能有多个。

树的中心满足如下性质：

1.它一定在某条树的直径上，并且到直径两端的距离差不超过1。

2.如果树有多条直径，则树的中心只有一个，为直径的中点。

在求出了节点深度、直径的两个端点x、y，x与y的最近公共祖先z之后，我们可以利用树的中心与树的直径之间的关系，求出树的中心。

设直径长度为R，我们要在x到y的路径上找到与x点距离为R/2的点。

假如R/2小于x到z的距离，则该点在x到z的路径上（我们可以从x往z找），否则在z到y的路径上（我们可以从z往y找）。

相关代码：

int work() {

int len = de[x] + de[y] - 2 \* de[z];

if (de[x] - de[z] > len / 2) {

for (int i = 1; i <= len / 2; ++i) x = fa[x];

return x;

}

for (int i = 1; i <= len - len / 2; ++i) y = fa[y];

return y;

}

**四、求树的重心**

一棵具有n个节点的无根树，若以某个节点为整棵树的根，它的每个儿子的子树大小都小于等于n/2，则称这个点为该树的重心。

我们可以先给无根树确定一个根（如节点1），求出以每个点为根的子树的大小。

这样，我们枚举每一个节点，看看它往下的子树（即以它的每个儿子为根的子树）与往上的子树（即整棵树去除以其为根的子树的部分）的大小是否都小于等于n/2。

图1.10，定0号点为无根树的根，枚举5号节点，虚线框出的部分为向下的子树，实线框出的部分为向上的子树。

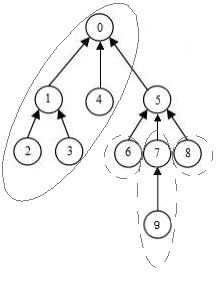


图1.10 求树的重心示例图

相关代码：

int si[MAXN];

int getsi(int x){ //求初次定根后的子树大小

si[x]=1;

for (int i = fi[x]; i; i = ne[i]) if (to[i] != fa[i]) {

fa[to[i]] = x;

si[x] += getsi(to[i]);

}

return si[x];

}

bool check(int x) { //检验是否是树的重心

if (n - si[x] > n / 2) return 0;

for (int i = fi[x]; i; i = ne[i])

if ((fa[to[i]] == x) && (si[to[i]] > n/2))

return 0;

return 1;

}

树的重心具有如下性质：

1.若求树中某个点到其他点的距离之和，则重心到其他点的距离和最小，如果有两个重心，则它们的距离和一样。

­2.一棵树添加或者删除一个节点，树的重心最多只移动一条边的位置。

3.把两棵树通过某一点相连得到一棵新的树，新树的重心必然在连接原来两棵树重心的路径上。

**例1.7** 给出一棵有根树，求出每棵子树的重心（各求一个即可）。

**解法**：如果直接暴力求每棵子树的重心，时间复杂度为O(N2)。需要优化。根据树的重心的性质，我们可以从下往上对子树求解。假设现在要求以u为根的子树的重心，我们先求出u的每个子节点的子树重心。显然，以u为根的子树的重心位于u点到这些子树重心的路径上。假如每个儿子的子树大小都小于等于si[u]的一半，则u点可以作为重心；否则，至多有一个儿子的子树大小大于si[u]的一半，重心显然不可能在其他子树中（否则分出来的联通块大小就一定会大于si[u]的一半），所以，重心一定在这个儿子的子树的重心到u点的路径上，我们只要从下往上枚举检验就行了。因为对重心的枚举一直是从下向上的，且不会重复枚举一个已经被判定为不是重心的点，所以每个点只会被淘汰（即被检验为非重心节点）1次；而对于每一棵子树，只会将一个点检验为重心（因为找到重心就停止向上枚举了），时间复杂度降为O(N)。

相关代码

bool check(int x,int y) { //检验是否是子树y的重心

if (si[y] - si[x] > si[y] / 2) return 0;

for (int i = fi[x]; i; i = ne[i])

if (si[to[i]] > si[y] / 2)

return 0;

return 1;

}

void getbaryct(int x) { //求每棵子树的重心

int p = -1;

for (int i = fi[x]; i; i = ne[i]) {

getbaryct(to[i]);

if(si[to[i]] > si[x] / 2) p = to[i];

}

if (p == -1) baryct[x] = x; else baryct[x] = baryct[p];

while(!check(baryct[x],x)) baryct[x] = fa[baryct[x]];

}

**五、 树形动态规划的一类经典问题**

树形动态规划（树形DP）是一类经典问题。一般的DP都是出现在序列上的，但是也有不少序列上DP可以被拓展到树上。树形DP一般就是在树的DFS遍历中维护了一些信息。

因为DP将在后面的章节中详细学习，这里我们只简略的提出一类经典问题。

对于每个点的状态数等于这个点的size的一类树形DP，一个很重要的思考方向在于dfs序的应用，下面算法的时间复杂度其实是O(n2)，其原因是任意一对点只会被枚举到一次。通过这个思路，我们可以解决很多（几乎所有）类似的树形DP问题。

void dfs(int x, int fa) {

si[x] = 1;

for (int i = fi[x]; i; i = ne[i]) if (to[i] != fa) {

dfs(to[i], x);

for (int u = 0; u <= si[x]; ++u)

for (int v = 0; v <= si[to[i]]; ++v)

状态f[x][u + v] 根据问题实际情况进行转移;

si[x] += si[to[i]];

}

}

**§1.5 树的统计**

**例1.8** 统计树上所有无序点对之间的距离和。x，y的距离定义为x到y路径上所有边的边权和。点数不超过105，边权为1。

**解法一：**暴力枚举一个点，以它为根DFS一遍，把每个点离根的距离累加到答案上。复杂度O(n2)。

**解法二**：考虑对每一条边的贡献。一条边(a, b)会被经过的次数等于以b为根的子树a的大小乘上以a为根的子树b的大小。复杂度O(n)。

算法实现:

int ans = 0;

void dfs(int x, int fa) {

ans += si[x] \* (n - si[x]);

for (int i = fi[x]; i; i = ne[i]) if (to[i] != fa) dfs(to[i], x);

}

本题可以把贡献拆开来计算，所以我们仅仅使用了两行简洁的代码就完成了它。可是大多数时候，我们面对的是更为复杂的统计问题。

**例1.9** 一棵以节点1为根的树，每个点有一个颜色col[x]。对每个点x，有一个询问query[x]，表示你需要求出以x为根的子树中有多少个颜色为query[x]的点。点数不超过105。

**提示**：在思考这道例题之前，我们先来考虑这样一个基本问题。有n个集合，每个集合大小为1。执行n-1次操作，每次操作任意指定两个大小分别为a，b的集合,**花费min(a, b)的代价合并a、b集合，即删去a、b集合，并加入大小为a + b的集合**。显然,n-1次操作后只会剩下一个集合。那么这些操作的代价和最大是多少呢？

**提示分析**：显然每次操作的代价不会超过n，所以总代价不会超过n2。但实际我们有更优秀的一个上界O(n\*log2n)。当合并大小为a，b (a < =b)的集合时，我们说a被合并了。对于任意一个元素，每当它被合并了时，它所在的集合大小都至少乘了2。而最终集合大小为n，也就是说每个元素会被合并不超过log2n次。那么总共就会花费不超过n\*log2n 的代价。

**原题分析**：如果你已经对基本问题有了一定的思考，就会发现，其实集合的合并过程可以看成一棵树的生成过程，而一棵树也可以看出是一个集合的合并过程。于是不难想到，如果我们尝试把之前的复杂度分析套用到树上，或许就可以在n\*log2n的时间内解出这道题。

对于每个节点，它的信息是由其所有的子树信息（集合）合并起来再加上自己的信息。它所代表的集合的大小可以看作是它子树的大小。所以我们直接在DFS的过程中把一个点的所有儿子的信息合并到它自身上即可。

代码实现：

unordered\_map <int, int> mp[MAXN];

void merge(int x, int y) {

if (mp[x].size() < mp[y].size()) mp[x].swap(mp[y]);

for (auto u: mp[y]) mp[x][u.first] += u.second;

}

void dfs(int x, int fa) {

++mp[x][col[x]];

for (int i = fi[x]; i; i = ne[i]) if (to[i] != fa) dfs(to[i], x), merge(x, to[i]);

ans[x] = mp[x][query[x]];

}

事实上这种做法有一个更正式的名字，叫做启发式合并。顾名思义，就是把小的集合并到大的上面去。

另外，存在另一种叫做dsu on tree的做法，与启发式合并原理相同，功能略少，空间更小，速度更快。限于篇幅，在此不做介绍。有兴趣的读者可以自行学习。

同时，存在一种叫做树链剖分的做法，它也是利用了这个原理。

顺便一提，当我们合并大小分别为a，b的集合计算贡献的时候，如果我们必须花费a+b的代价，有一种更为复杂的点（边）分治的做法，仍然保证O(n\*log2 n)的复杂度。

**例1.10** 一棵以节点1为根的树，对每个点x，有一个询问query[x]，表示你需要求出以x为根的子树中有多少个距离query[x]的点。点数不超过106。

**提示**：我们先来考虑类似例1.9中的提示问题。不同之处在于，“**花费min(a, b)的代价合并a，b集合，即删去a b集合，加入大小为max(a, b)+1的集合**”。

**提示分析**：显然max(a, b)+1 <= a + b，所以有一个O(n\*log2 n)的上界。但实际上还有一个更优秀的上界O(n)。一开始支付n的代价在每个集合上存储1点能量，并且时刻保持大小为k的集合上存储了k点能量。当合并大小为a，b (a < =b)的集合时，我们用存储在大小为a的集合上的a点能量来支付min(a, b)=a的代价，产生了一个大小为max(a, b)+1=b+1的新集合，所以多支付1的代价存储在合并后的集合上以满足之前的定义。把所有支付的代价加起来，发现一共花了2 \* n-1的代价。这就是这些操作代价和的一个上界。

**原题分析**：对于每个点，它的信息是由它的所有子树的信息（集合）合并起来再加上自己的信息。它所代表的集合大小可以看作是它子树中的不同深度的个数，即最大深度减去它的深度。所以我们直接再DFS的过程中把一个点的所有儿子的信息合并到它自己上即可。

代码实现：

unordered\_map<int, int> mp[MAXN];

void Merge(int x, int y) {

if (mp[x].size() < mp[y].size()) mp[x].swap(mp[y]);

for (auto u: mp[y]) mp[x][u.first] += u.second;

}

void dfs(int x, int fa, int de) {

++mp[x][de];

for (int i = fi[x]; i; i = ne[i]) if (to[i] != fa) dfs(to[i], x, de + 1), merge(x, to[i]);

ans[x] = mp[x][de + query[x]];

}

实际上这种做法有一个更广为人知的同族兄弟，叫做长链剖分。

**总结：树上统计各类算法比较**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 技巧法 | 启发式合并  dsu on tree | 类长链剖分 | 点分治  边分治 |
| 合并代价 | 无 | min(a, b) | min(a, b) | a+b |
| 合并产生的集合大小 | 无 | a+b | max(a, b) + 1 | a+b |
| 最坏情况下时间复杂度 | 无 | O(n\*log2 n) | O(n) | O(n\*log2 n) |
| 特点 | 针对少量特定的题目的特定方法。 | 实现简单，常数小。初学者较容易握。 | 实现简单，常数小，复杂度优秀，但应用范围较小。 | 常数一般，功能完整。初学者不太容易掌握。 |

注：本节中提到的树链剖分以及点分治/边分治算法，都是解决树的问题中的有力工具。然而相对来说难度较高。这里为了和一些其它的算法做对比，做了简单的介绍。学有余力的读者可以自行了解。

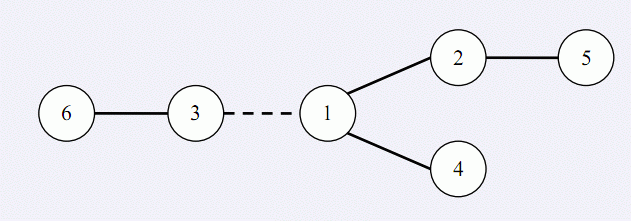
**§1.6 竞赛真题分析**

由于树的结构简洁优美，它受到了不少出题人的青睐。其中的不少技巧都是在特定题目中才会出现的。以下是一些典型的例题。由于是实际竞赛中的题目，它们不可避免的结合了其它的知识，可能难度比较大。如觉得有困难，可以在阅读了后面的章节后再综合考虑。

**例1.11道路修建[[1]](#footnote-0)**

**题目描述**

在 W 星球上有 n 个国家。为了各自国家的经济发展，他们决定在各个国家之间建设双向道路使得国家之间连通。但是每个国家的国王都很吝啬，他们只愿意修建恰好 n–1条双向道路。 每条道路的修建都要付出一定的费用， 这个费用等于道路长度乘以道路两端的国家个数之差的绝对值。例如，图1.11中，虚线所示道路两端分别有 2 个、4个国家，如果该道路长度为 1，则费用为1×|2–4|=2。图中圆圈里的数字表示国家的编号。



**图1.11**

由于国家的数量十分庞大，道路的建造方案有很多种，同时每种方案的修建费用难以用人工计算，国王们决定找人设计一个软件，对于给定的建造方案，计算出所需要的费用。请你帮助国王们设计一个这样的软件。

**输入格式**

输入的第一行包含一个整数 n，表示 W 星球上的国家的数量，国家从 1 到 n 编号。 接下来 n–1 行描述道路建设情况，其中第 i 行包含三个整数 ai、bi和 ci，表 示第 i 条双向道路修建在 ai与 bi两个国家之间，长度为 ci。  
**输出格式**

输出一个整数，表示修建所有道路所需要的总费用。

**样例输入**

6

1 2 1

1 3 1

1 4 2

6 3 1

5 2 1

**样例输出**

20

**数据范围**

2≤n≤1000000，1≤ai，bi≤n，0≤ci≤1000000

**分析**

因为树已经给出，所以可以任选一个根建树，记录以每个点为根的子树有多少节点，每条边的费用就是a[i].d\*abs((N-f[i].d)-f[i].d)，f[i].d是以i为根的子树的节点数，a[i].d是边的长度。

因为N很大，所以递规建树会爆栈，所以先记录下BFS序，此时从后往前找就是从叶往根找，用当前节点更新父节点。

**例1.12 逃学的小孩[[2]](#footnote-1)**

**题目描述**

Chris家的电话铃响起了，里面传出了Chris的老师焦急的声音：“喂，是Chris的家长吗？你们的孩子又没来上课，不想参加考试了吗？”一听说要考试，Chris的父母就心急如焚，他们决定在尽量短的时间内找到Chris。他们告诉Chris的老师：“根据以往的经验，Chris现在必然躲在朋友Shermie或Yashiro家里偷玩《拳皇》游戏。现在，我们就从家出发去找Chris，一旦找到，我们立刻给您打电话。”说完砰地一声把电话挂了。

Chris居住的城市由N个居住点和若干条连接居住点的双向街道组成，经过街道x需花费Tx分钟。可以保证，任两个居住点间有且仅有一条通路。Chris家在点C，Shermie和Yashiro分别住在点A和点B。Chris的老师和Chris的父母都有城市地图，但Chris的父母知道点A、B、C的具体位置而Chris的老师不知。

为了尽快找到Chris，Chris的父母会遵守以下两条规则：

如果A距离C比B距离C近，那么Chris的父母先去Shermie家寻找Chris，如果找不到，Chris的父母再去Yashiro家；反之亦然。

Chris的父母总沿着两点间唯一的通路行走。

显然，Chris的老师知道Chris的父母在寻找Chris的过程中会遵守以上两条规则，但由于他并不知道A，B，C的具体位置，所以现在他希望你告诉他，最坏情况下Chris的父母要耗费多长时间才能找到Chris？

**输入格式**

第一行是两个整数N和M，分别表示居住点总数和街道总数。以下M行，每行给出一条街道的信息。第i+1行包含整数Ui、Vi、Ti，表示街道i连接居住点Ui和Vi，并且经过街道i需花费Ti分钟。街道信息不会重复给出。  
**输出格式**

输出仅一行，包含一个整数T，表示最坏情况下Chris的父母需要花费T分钟才能找到Chris。

**样例输入**

4 3

1 2 1

2 3 1

3 4 1

**样例输出**

4

**数据范围**

3≤N≤200000，1≤Ui, Vi≤N，1≤Ti≤1000000000

**分析**

经分析，对于A，B而言，dis[A][C]，dis[B][C]谁小谁大是不重要的，因为如果要先去A，在dis[A][C]>dis[B][C]的情况下完全可以将AB调换，因此关键在于AB的路径选择。

结论：最优方案中，AB必定为树中的最长链。

于是，只要先在树中找一条最长链，链两端就是AB，然后枚举C，求最优解即可。

求最长链可以通过两次搜索完成，同时可以完成A到其他的距离的初始化，然后由B出发遍历每一个点，计算出B到其他点的距离，这样计算总距离就可以在O(1)的时间内完成了。

总时间复杂度O(n)。

**例1.13道路重建[[3]](#footnote-2)**

**题目描述**

在上个5月一场极其严重的大地震后，奶牛们重建了农民杰克有N个谷仓的农场。但是奶牛们没有时间去修建更多的道路，所以现在每个谷仓之间只有一条路。因此，农场的交通系统可以认为是一棵树。

农民杰克想知道地震能造成多少破坏——最少需要被破坏的道路总数使得1棵包含P个节点的子树被隔离。

**输入格式**

第1行包含2个整数，N和P。

第2到N行，每行包含两个整数i和j，表示节点i是节点j的父亲。

**输出格式**

共一行包含一个整数，表示最少需要被破坏的道路总数使得1棵包含P个节点的子树被隔离。

**样例输入**

11 6

1 2

1 3

1 4

1 5

2 6

2 7

2 8

4 9

4 10

4 11

**样例输出**

2

**样例解释**

如果道路1-4和1-5被破坏，那么一个包含节点（1,2,3,6,7,8）的子树会被隔离。

**数据范围**

1≤N≤150 , 1≤P≤N

**分析**

树形DP + 背包。由于给定的结构是树，就要想到树的递归特性，而树形dp的优美之处是可以利用子树的状态来转移，来求得根的状态。本题要求求最少删除几条边使得子树节点个数为p，我们只要算出每个以节点i为根的树中节点个数为p的最少删除边数，求个最小值就好。其实我们可以这样想，每棵以i为根的树有sum种物品（sum为他以及与他的子孙节点的个数），必须要删除k条边才能使得这棵子树有j个节点（1≤j≤sum)，那么每个物品j的费用是k，价值是j，这样问题就转换为在树上的分组背包，总共有n组物品，每次都从以i为根的物品组中选择一个物品进行转移，每组选择一个物品。由于根节点固定了是1，我把树看成有向的树，也就是每次求解都不管父节点。

现在设dp[i][j]表示以i为根的子树中节点个数为j的最少删除边数（从分组背包角度理解就是到转移到第i组价值为j的最少费用）

状态转移方程:

dp[i][1] = tot (tot为他的子节点个数)

dp[i][j] = min(dp[i][j],dp[i][k]-1+dp[s][j-k]) (1≤i≤n，2≤j≤sum[节点总和],1≤k＜j,s为i子节点)(i中已有k个节点并从s中选择j-k个，算最少删除边数，s选上所以i→s的边不需删除，所以-1)

**例1.14货车运输[[4]](#footnote-3)**

**题目描述**

A 国有n座城市，编号从1到 n，城市之间有 m 条双向道路。每一条道路对车辆都有重量限制，简称限重。现在有q辆货车在运输货物，司机们想知道每辆车在不超过车辆限重的情况下，最多能运多重的货物。

**输入格式**

第一行有两个用一个空格隔开的整数n、m，表示 A 国有 n 座城市和 m 条道路。

接下来 m 行每行 3 个整数 x、y、z，每两个整数之间用一个空格隔开，表示从 x 号城市到 y 号城市有一条限重为 z 的道路，注意：x 不等于 y，两座城市之间可能有多条道路。

接下来一行有一个整数 q，表示有 q 辆货车需要运货。

接下来 q 行，每行两个整数 x、y，之间用一个空格隔开，表示一辆货车需要从 x 城市运输货物到 y 城市，注意：x 不等于 y。

**输出格式**

输出共有q 行，每行一个整数，表示对于每一辆货车，它的最大载重是多少。如果货车不能到达目的地，输出-1。

**样例输入**

4 3

1 2 4

2 3 3

3 1 1

3

1 3

1 4

1 3

**样例输出**

3

-1

3

**数据范围**

对于 30%的数据，0＜n＜1,000，0＜m＜10,000，0＜q＜1,000；

对于 60%的数据，0＜n＜1,000，0＜m＜50,000，0＜q＜1,000；

对于 100%的数据，0＜n＜10,000，0＜m＜50,000，0＜q＜30,000，0≤z≤100,000。

**分析**

一条路的瓶颈是限重最小的边。把边按限重从大到小一条一条地加进去，使图在极大连通的原则下尽量保留限重较大的边。这有点类似 Kruskal 算法。事实上，这一过程求出的就是原图的最大瓶颈生成森林。所有询问的最优解一定在原图的最大生成森林上。

问题转化为：如何快速求出一条树上路径权值最小的边 .

最简单的方法就是暴力模拟，事实上这个方法在比赛中的确能得到不错的分数。倍增算法是暴力模拟法优化得来的，那么这个问题能否套用倍增算法呢？对于节点 x，我们还要记录 mi[x][i]，代表从 x 到 p[x][i] 的路径上最小边的权值 mi[x][i] = min(mi[x][i−1], mi[p[x][i−1],i−1]) 每次我们把 x 往上爬 2i 步时，都用 mi[x][i] 来更新答案 这样就可以用倍增求 LCA 类似的方法求出路径上的最小边权了。

**例1.15 灾难[[5]](#footnote-4)**

**题目描述**

阿米巴是小强的好朋友。

阿米巴和小强在草原上捉蚂蚱。小强突然想，如果蚂蚱被他们捉灭绝了，那么吃蚂蚱的小鸟就会饿死，而捕食小鸟的猛禽也会跟着灭绝，从而引发一系列的生态灾难。

学过生物的阿米巴告诉小强，草原是一个极其稳定的生态系统。如果蚂蚱灭绝了，小鸟照样可以吃别的虫子，所以一个物种的灭绝并不一定会引发重大的灾难。

我们现在从专业一点的角度来看这个问题。我们用一种叫做食物网的有向图来描述生物之间的关系：

一个食物网有 N个点，代表 N 种生物，如果生物 x 可以吃生物 y，那么从 y 向 x 连一个有向边。

这个图没有环。

图中有一些点没有连出边，这些点代表的生物都是生产者，可以通过光合作用来生存； 而有连出边的点代表的都是消费者，它们必须通过吃其他生物来生存。

如果某个消费者的所有食物都灭绝了，它会跟着灭绝。

我们定义一个生物在食物网中的“灾难值”为，如果它突然灭绝，那么会跟着一起灭绝的生物的种数。

举个例子：在一个草场上，生物之间的关系是：

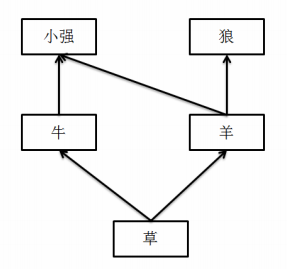


图1.12

如果小强和阿米巴把草原上所有的羊都给吓死了，那么狼会因为没有食物而灭绝，而小强和阿米巴可以通过吃牛、牛可以通过吃草来生存下去。所以，羊的灾难值是 1。但是，如果草突然灭绝，那么整个草原上的5 种生物都无法幸免， 所以，草的灾难值是 4。  
 给定一个食物网，你要求出每个生物的灾难值。  
**输入格式**

第一行是一个正整数 N，表示生物的种数。生物从1 标号到 N。

接下来 N 行，每行描述了一个生物可以吃的其他生物的列表，格式为用空格隔开的若干个数字，每个数字表示一种生物的标号，最后一个数字是 0 表示列表的结束。   
**输出格式**

输出包含N行，每行一个整数，表示每个生物的灾难值。   
**样例输入**

5

0

1 0

1 0

2 3 0

2 0

**样例输出**

4

1

0

0

0

**数据范围**

对 50%的数据，1≤N≤10000。

对 100%的数据，1≤N≤65534。

保证输入的食物网没有环。

**分析**

定理：生物之间的灭绝的结构形成了一个森林，森林上的一个节点的灭绝会且仅会导致以它为根的子树的灭绝。

对于生产者，不妨给它添加一个假想的食物——太阳。那么定理中的森林就变成了一棵树，可以称之为“灭绝树”。

下面说明如何通过增量法把灭绝树建出来，同时也是对灭绝树的存在性的证明。首先，把食物网按从猎物到捕食者的顺序拓扑排序，有且仅有太阳没有任何入边，所以拓扑序中的第一个就是太阳。把太阳加入灭绝树，此时灭绝树中只有太阳一个点。

之后，依次考虑每个生物 i。已经构建好了排序在i之前的生物组成的灭绝树了，假设 i 的食物有 p1, p2, p3, ..., pk（这些节点在拓扑序中比 i 靠前），i 会灭绝，当且仅当 p1, p2, p3, ..., pk 全部灭绝，当且仅当 LCA(p1, p2, p3, ..., pk) 灭绝。

于是可以在树上加上 LCA 到 i 的边，把 i 加到树中。

处理完所有的生物，我们得到的树就是整个图的灭绝树。

一旦得到灭绝树，每个生物的灾难值就可以通过以它为根的子树的大小减 1 来计算。

可以通过一次 DFS 来解决。

复杂度分析（LCA 使用倍增算法）：拓扑排序 O(N + M)，求 LCA 共 M 次，每次 O(log2 N)，加点共 N 次，每次 O(log2N)，DFS O(N)，总 O((N + M)log2 N)。

课后习题：

XJOI3326：树的深度again <http://www.hzxjhs.com:83/problem/3326>

XJOI3329：树上两点距离 <http://www.hzxjhs.com:83/problem/3329>

XJOI3333：每个点子树大小 <http://www.hzxjhs.com:83/problem/3333>

XJOI3335：树上最远点 <http://www.hzxjhs.com:83/problem/3335>

XJOI3429：字典树 <http://www.hzxjhs.com:83/problem/3429>

XJOI1050：树的重心节点 <http://www.hzxjhs.com:83/problem/1050>

XJOI3361：树2 <http://www.hzxjhs.com:83/problem/3361>

XJOI3361：树3 <http://www.hzxjhs.com:83/problem/3362>

1. NOI2011 Day2试题 [↑](#footnote-ref-0)
2. NOI2003试题 [↑](#footnote-ref-1)
3. USACO 2002 February Green试题 [↑](#footnote-ref-2)
4. **NOIP2013试题** [↑](#footnote-ref-3)
5. ZJOI2012 [↑](#footnote-ref-4)