树形动态规划讲稿

1.树形动态规划入门

动态规划一般用于解决最优性问题、计数类问题以及较少见的存在性问题。当在树形结构上解决以上三类问题时，就称为树形动态规划。更准确地说，状态之间的转移关系呈现一种树形结构时，称为树形动态规划。

树本身是一个递归结构，父亲与儿子之间天然形成一种大问题与所包含的子问题的递归关系。树的根就对应着整个问题了。动态规划的状态转移方程也是一个递归式。因此，在分析树形 DP 的最优子结构时，要特别注意这种结构与形式的对应关系。

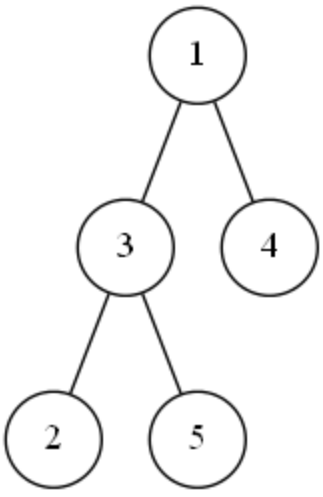
树形结构可分为二叉树与多叉树。相较而言，二叉树简单，容易建立状态转移方程。通过例子加以说明：

例 1 二叉苹果树(bzoj1375)

【问题描述】

有一棵苹果树，如果树枝有分叉，一定是分 2 个叉，就是说没有只有 1 个儿子的结点。这棵树共有 N 个结点（叶子点或者树枝分叉点），结点编号 1～N，树根编号一定是 1。

我们用一根树枝两端连接的结点的编号来描述一根树枝的位置。下面是一颗有 4 根树枝的树：



现在这颗树枝条太多了，需要剪枝。但是一些树枝上长有苹果。

给定需要保留的树枝数量，求出最多能留住多少苹果。

【输入格式】

第 1 行：2 个数，N 和 Q(1<=Q<= N,1<N<=100)。N 表示树的结点数，Q 表示要保留的树枝数量。

接下来 N-1 行描述树枝的信息。每行 3 个整数，前两个是它连接的结点的编号。第 3 个数是这根树枝上苹果的数量。

每根树枝上的苹果不超过 30000 个。

【输出格式】
第 1 行：1 个数，表示最多能留住的苹果的数量。

【样例输入】
5 2
1 3 1
1 4 10
2 3 20
3 5 20

【样例输出】
21

【问题分析】
先撇下如何建树的问题，集中精力思考树形动态规划的相关问题。首先定性分析最优子结构。

如果只保留一条树枝，该如何决策？显然应该比较“保留根->左儿子的树枝”与“保留根->右儿子的树枝”，然后取其中大的一个。

如果要保留二条树枝呢？我们应当考虑：把二条树枝都保留在根的左子树上，把二条树枝都保留在根的右子树上，左、右子树各保留一条树枝等三种情况。而每个子问题都需要是最优解，才能在比较之后，取其中大的一个作为整个问题的解。

当考虑要保留更多的树枝时，我们需要在左、右子树之间分配要保留的树枝数。每种分配方案对应个两个子问题，并且子问题的最优解相加，得到当前分配方案的解。整个问题的解就是所有分配方案的解中最优的一个。

由此，我们定性地得出了最优子结构。下面，准确地描述状态，并定量写出状态方程。

F[i][j]表示以结点 i 为根的子树，保留 j 条树枝能得到的最多苹果数。题目中苹果长在树枝上，为了方便存储数据，把苹果“下移”到儿子结点上。根据树的性质，每条树枝与一个子结点一一对应，这样的转移是可行的。

枚举分配给左子树的树枝数 k，枚举范围从 0～j。从 i 到左儿子 lch 要保留一条树枝，lch 实际得到的树枝数为 k-1，对应的子问题可表示为 F[i.lch][k-1]。同理，分配给右子树的树枝数则为 j-k，对应的子问题为 F[i.rch][j-k-1]。由此，得出状态转移方程：

F[i][j] = max{F[i.lch][k-1] + F[i.rch][j-k-1] | 0<=k<=j}

接下来讨论边界状态。首先讨论参数 j，它对状态的约束性更强。从状态方程中决策变量 k 的枚举范围可知，j 的边界值可为-1 和 0。

当 j=-1 时表示左子树不保留树枝，这时状态的值为 0，即：F[i][-1] = 0。

当 j=0 时，表示保留了 i 与父亲之间的树枝，而树枝上的苹果下移到结点 i 中，故返回 i 中的苹果数。即：F[i][0] = tree[i].cnt。

当 j>=1 时，就需要在左、右树之间分配，不再是边界状态了。

当 i 是叶结点时，无论此时 j 的值为多大（如果是自上而下书写边界条件，成功穿透 j=-1, 0 两个条件的检验，此时 j>=1），都应该只能返回结点 i 中的苹果数，即 F[i][j]=tree[i].cnt。

目标状态很明显为：F[1][Q]。

最后着重说一下树形 DP 的实现特点。实现普通 DP，可以有两种实现方式：自底向上的递推和自顶向下的记忆化搜索。无论何种方式，用表格（数组）来记录子问题的解都是必须

的。树形 DP 由于状态一般定义在结点上，规划实际是对整棵树所有结点的遍历过程。而树的遍历可分为递归形式的深度优先遍历 DFS（无论何种根序的遍历都是 DFS）和递推形式的宽度优先遍历 BFS。树的 BFS，是从根开始，逐层向子代结点遍历。但是，树形 DP 中父、子结点间的状态一般正好对应着问题与子问题。我们几乎不可能从问题的解去反推子问题的解，因此一般不能用 BFS 去实现树形 DP。（如果某个问题能把状态方程逆转方向，则此时用 BFS 做树形 DP 就是可行的）。

DFS 则不同。它会依次递归地求解各个子问题，然后在向上一层返回前，根据已经求得的子问题的最优解，去构造出当前问题的最优解。因此，DFS 天生适用于树形 DP，这是递归的形式应用于递归的结构的完美例子。

用 DFS 实现树形 DP，状态之间不再能划分出清晰的层次（阶段），无法逐层递推，而只能记忆化搜索：在递归函数的入口位置，查看表格中相应表项是否已经计算过。若是，则直接返回该表项的值。若不是，则再实施计算；在出口位置，把计算得出的结果填入表项，以便后续调用时直接返回该值。参考实现如下：

```
int dp(int i, int j)              //动态规划
{
    int k, t1, t2, ret=0;
    //处理边界
    if(j< 0) return 0;
    if(j==0) return tree[i].cnt;

    if(f[i][j] > 0) return f[i][j];       //直接返回表项的值

    if(tree[i].lch == 0 && tree[i].rch == 0)
    {
        f[i][j] = tree[i].cnt;
        return tree[i].cnt;
    }
    for(k=0; k<=j; k++)
    {
        t1 = dp(tree[i].lch, k-1);
        t2 = dp(tree[i].rch, j-k-1);
        ret = max(ret, t1+t2);
    }
    ret += tree[i].cnt;
    f[i][j] = ret ;
    return ret;
}
```

至此，关于树形动态规划的分析与实现步骤讨论完毕。最后，针对本题的输入数据格式，说一下建树的方法。

输入数据只描述了树的每一条边，而没有具体说明每个结点的左、右儿子编号。这就需要先把边的信息存入一个邻接矩阵，再从根（结点 1）开始，沿着树边遍历一次，明确每个结点的左、右儿子。遍历用 DFS 或者 BFS 都可以，下面的代码用 BFS。

```
bool visit[MAXN];
queue<int>q;
```

```
void bfs(int rt)            //建树,rt 是根
{
    int i, j;
    q.push(rt);
    visit[rt] = true;
    while(!q.empty())
    {
        i = q.front();   q.pop();
        for(j=1; j<=n; j++)
        {
            if(!visit[j] && mat[i][j] >= 0)
            {
                if(tree[i].lch == 0)
                {
                    tree[i].lch = j;
                    tree[j].cnt = mat[i][j];
                }
                else
                {
                    tree[i].rch = j;
                    tree[j].cnt = mat[i][j];
                }
                visit[j] = true;
                q.push(j);
            }
        }
    }
}
```

很多时候，树形 DP 都不是在二叉树上的，每个结点的儿子数量不固定，可多可少。如果定义的状态的某一维参数，需要在各个儿子之间分配，为了便于分配，常把多叉树转为二叉树。在具体实现时，一般读入多叉树的数据，直接构建成二叉树。这里的二叉树，含义是“左儿子，右兄弟”，相当于多叉树的一种存储方式。

例 2 选课(bzoj1376)

【问题描述】

在大学里每个学生，为了达到一定的学分，必须从很多课程里选择一些课程来学习，在课程里有些课程必须在某些课程之前学习，如高等数学总是在其它课程之前学习。现在有 N 门功课，每门课有个学分，每门课有一门或没有直接先修课（若课程 a 是课程 b 的先修课即只有学完了课程 a，才能学习课程 b）。一个学生要从这些课程里选择 M 门课程学习，问他能获得的最大学分是多少？

【输入格式】

第 1 行：2 个整数 N,M 用空格隔开。(1<=N<=200,1<=M<=150)

接下来的 N 行，第 i+1 行包含 2 个整数 ki 和 si，ki 表示第 i 门课的直接先修课，si 表示第 i 门课的学分。若 ki=0 表示没有直接先修课（1<=ki<=N, 1<=si<=20）。

【输出格式】
第 1 行：1 个整数，表示选 M 门课程的最大得分。

【样例输入】
7 4
2 2
0 1
0 4
2 1
7 1
7 6
2 2

【样例输出】
13

【问题分析】

这道题并没有直接给出一棵树。如果课程 i 是课程 j 的直接先修课，则称 i 是 j 的父亲。这样，所有课程就形成一个先修课关系的森林，如图 1 所示。新增一个虚拟的课程 n+1，指向所有无先修课的课程，作为整个树的根，如图 2 所示。显然，这是一棵多叉树。

根据前面关于树形 DP 状态定义的经验，定义状态为：F[i][j]，表示在以 i 为根的子树上选择 j 门课程的最大得分。

参数 j 作为约束条件，需要在各个儿子结点上分配。当儿子数多于 2 时，不便于分配。如果把多叉树转为二叉树，则这一困难就解决了。

不过注意，在建立状态转移方程时，结点 i 的右子树实际是它的所有兄弟子树。左子树才是它的真正儿子。因此，分配 j 时左、右子树不完全是一致的。j 可以全部分配 i 的右儿子，相当于不分配给子树 i。当分配给 i 的左儿子时，一定要选择课程 i，因为 i 是左儿子的先修课。状态转移方程为：

F[i][j] = max{F[i.rch][j], F[i.lch][k]+F[i.rch][j-1-k]+tree[i].sco |0<=k<=j-1 }

参考代码如下：

```
int dp(int i, int j)       //以 i 为根的树，选 j 门课的最大学分和
{
    int k, t1, t2, t3 = 0, ret = 0;
    //处理边界
    if(i==0 || j==0)       return 0;
    if(f[i][j] > 0) return f[i][j];
    if(tree[i].lch == 0 && tree[i].rch == 0)
    {
        f[i][j] = tree[i].sco;
        return f[i][j];
    }
```

```
    //如果有右兄弟，则 i 及左子树可以不选
    if(tree[i].rch != 0)
        ret = dp(tree[i].rch, j);
    //一定要选 i，剩下的 j-1 门课在左、右子分配
    for(k = 0;   k <= j-1; k++)
    {
        t1 = dp(tree[i].lch, k);            //easy to debug
        t2 = dp(tree[i].rch, j-k-1);
        t3 = max(t3, t1+t2);
    }
    t3 += tree[i].sco;
    ret = max(ret, t3);
    f[i][j] = ret;
    return ret;
}
```

输入数据，把多叉树转成二叉树的代码如下。每个结点中的 fa 域标识它是否有先修课。所有没有先修课的结点串成虚拟结点 n+1 的左子树。

```
void read()
{
    int i, j, k, u;
    cin >> n >> m;
    root = n+1;                         //虚拟的根结点编号
    for(i = 1; i<=n; i++)
    {
        cin >> j >> k;
        tree[i].sco = k;            //课程 i 的学分
        if(j == 0) continue;
        if(tree[j].lch == 0)       //左儿子，右兄弟
        {
            tree[j].lch = i;       //存为先行课的左儿子
            tree[i].fa = j;        //指向“真实”父亲
        }
        else
        {
            u = tree[j].lch;
            while(tree[u].rch != 0) //找到右儿子为空的结点
                u = tree[u].rch;
            tree[u].rch = i;
            tree[i].fa = j;        //指向“真实”父亲
        }
    }
    //虚拟结点 n+1 为整棵树的根，把森林串成二叉树
    for(i=1; i<=n; i++)
    {
```

```
            if(tree[i].fa ==0)
            {
                if(tree[root].lch ==0)
                {
                    tree[root].lch = i;
                    tree[i].fa = root;
                }
                else
                {
                    u = tree[root].lch;
                    while(tree[u].rch != 0)
                        u = tree[u].rch;
                    tree[u].rch = i;
                    tree[i].fa = root;
                }
            }
        }
    }
```

小结：树形 DP 的解题思路

首先要辨识题目描述中的树结构。描述有很多种，实际上都是在传达“这是一个树结构”的信息。

- 这是一个树结构(最直白了)
- 任意两点间只有一条路径
- N 个结点相互连通，但只有 N-1 条边/N-1 对点相邻
- 家族关系/上司下属关系(也许有些时候不是树，需要仔细确认)

其次是建立树结构。如果已经告诉结点间的父子关系，则树可以用静态的结构体数组存储；否则，需要动态的结构体链表存储。如果树是以普通图的形式给出（无根树），则需要遍历一次确定父子关系。

重点与难点是定义状态。状态中几乎可以肯定有一维表示的是以 i 为根的子树，而且大多数题目会有第二个维度的存在。

- 在双约束问题中，第二个维度可以是“某个约束的值为 xx 时”，此时往往还需要多叉转二叉，以便分配约束条件。也可以不转二叉，直接在这个结点上进行背包。
- 在选择/覆盖问题中，第二个维度可以是“是否选择”、“是否被覆盖”等
- 在路径问题中，第二个维度可以是“是否返回”

接着是建立状态转移方程。树形 DP 的子状态一般都是儿子结点，比较容易建立方程。对于边界状态，叶结点是一种边界状态。第二个维度的边界值也是一种边界状态。

最后是实现，树形 DP 用记忆化搜索，树太深的时候要注意爆栈的问题。

2.树形动态规划的经典模型

2.1 双约束类型

在前文如何定义状态中已经提到，树形 DP 大致有 3 种类型。例 1、2 属于双约束问题。

基本思路是把约束条件在左、右子树之间分配。如果是多叉树，则需要先转为二叉树，再做DP。我们再看用背包的思想如何解决双约束类型。

例 3 有线电视网(bzoj1289 pku1155)

【题目描述】

某收费有线电视网计划转播一场重要的足球比赛。他们的转播网和用户终端构成一棵树状结构，这棵树的根结点位于足球比赛的现场，树叶为各个用户终端，其他中转站为该树的内部结点。从转播站到转播站以及从转播站到所有用户终端的信号传输费用都是已知的，一场转播的总费用等于传输信号的费用总和。

现在每个用户都准备了一笔费用想观看这场精彩的足球比赛，有线电视网有权决定给哪些用户提供信号而不给哪些用户提供信号。

写一个程序找出一个方案使得有线电视网在不亏本的情况下使观看转播的用户尽可能多。

【输入数据】

第 1 行：2 个整数 N 和 M，其中 2≤N≤3000，1≤M≤N-1，N 为整个有线电视网的结点总数，M 为用户终端的数量。 第一个转播站即树的根结点编号为 1，其他的转播站编号为 2 到 N-M，用户终端编号为 N-M+1 到 N。

接下来的 N-M 行每行表示一个转播站的数据，第 i+1 行表示第 i 个转播站的数据，其格式如下： K A1 C1 A2 C2 … Ak Ck K 表示该转播站下接 K 个结点(转播站或用户)，每个结点对应一对整数 A 与 C，A 表示结点编号，C 表示从当前转播站传输信号到结点 A 的费用。

最后一行依次表示所有用户为观看比赛而准备支付的钱数。

【输出数据】

仅一行，包含一个整数，表示上述问题所要求的最大用户数。

【样例输入】

```
5 3
2 2 2 5 3
2 3 2 4 3
3 4 2
```

【样例输出】

```
2
```

【问题分析】

本题的树结构是直接告诉的，而且是求最优解，DP 的倾向非常明显。但是如果定义状态却不那么容易。除结点 i 为根作为一个维度外，第二个维度是什么呢？题目涉及两个因素：转播的收益和用户数。如果把收益作为第二个维度，定义状态 F[i][j]表示在 i 为根的子树，收益为 j 时最多可转播的用户数。继续往下分析，似乎最优子结构也是存在的。但这样定义状态的问题是：j 的值域是什么，似乎不好确定。

如果把 j 当作用户数来充当第二个维度，然后去求为转播给 j 个用户的最大收益。这样

定义也是有最优子结构的。而且最大用户数是题目给出的，便于枚举。因此定义状态为：F[i][j]表示以 i 为根的子树恰好转播给 j 个用户的最大收益。目标状态为 F[root][x]，从小到大枚举 x 的值，对每个 x，做一次 DP。由于是记忆化搜索，DP 的次数虽然有多次，但不会有重复的计算。当遇到某个 x 收益为负时，动态规划结束。

这样定义状态后，就是一道典型的双约束树形 DP 了。树的结构是多叉的，我们尝试用背包的思想来建立状态转移方程。

如果结点 i 有若干个子树，总共要转播给 j 个用户，即得到 j 个叶结点。把 j 看成背包，顺序地考虑 i 的各个子树，当前的儿子结点为 k 的子树有 son[k]个叶子结点。我们考虑在 k 的子树上选取 x 个叶结点（1<=x<=son[k]），则有状态转移方程：

F[i][j] = max{F[i][j], F[i][j-x] + F[k][x] + tree[k].cost | j<=x <=son[k]}

tree[k].cost 表示结点 k 的转播费用，值为负。而所有叶结点的转播费用其实代表正的收益。

在实现时，用一个数组 son[i]记录以 i 为根的子树所包含的叶结点数量。初始时，所有叶结点 son[leaf] = 1，而所有内结点 i 的 son[i] = 0。DP 时，由于是一个 DFS 的过程，可以递归地处理完一个子树，就对这个子树做一次背包，并把这个子树包含的叶结点数量累加到背包变量 j 中。根据输入数据的特点，整棵树建成一个父亲指向儿子的有向图，这样可以在 DFS 时省掉加标记。

```
void dfs(int u)        //有向无环图，不用 visit[]
{
    int i, j , k, v;
    int tmp[MAXN];
    for(node *p = adj[u]; p != NULL; p = p->next)
    {
        v = p->v;
        dfs(v);
        for(i = 0; i<=son[u]; i++)              //把不包含子树 v 的状态值转存到 tmp 数组中
            tmp[i] = f[u][i];
        //对子树 v 作背包 DP，向前递推
        for(i = 0; i <= son[u]; i++)            //枚举之前子树上取的叶结点数
            for(j = 1; j <= son[v]; j++)        //枚举子树 v 上取的叶结点数
                f[u][i+j] = max(f[u][i+j], tmp[i]+f[v][k]+p->cost);
        son[u] += son[v];
    }
}
```

上述代码对子树 v 作背包 DP 之前，把状态值 f[u][i]转存到临时数组 tmp[i]中是必须的，因为 f[u][i]的值一定是在不是 v 的树上选 i 个叶结点。如果直接引用 f[u][i]，就有可能用到了之前的 i+j 还比较小的包含 v 子树上结点的状态值。

tmp[]数组定义在递归函数中，稍不注意就引起爆栈，所以最好想办法去掉。如果内循环从大到小枚举 j 是否可以不用 tmp[]数组呢？

小结：动态规划本质是一种搜索。像本题，目标状态不能事先确定，需要去查找 DP 后的表格，搜索满足条件的目标状态。这种方式的 DP 不常见，但仍非常重要。希望能认真体会其思想。这种树形 DP 套背包 DP 的方法，是解决多叉树双约束问题的经典套路。个人觉得，比多叉转二叉的思维上要直观一些，更容易理解和掌握。

类似的题目还是 BZOJ1288 重建道路。

树型 DP。首先需要有一个感性认识，只有这个 P 是不大于粮仓数的一个正整数，这个子树是肯定存在的。将某个叶子剪掉，子树的粮仓数目就减少了 1，如此将某个非叶子节点的所有叶子剪掉之后，这个节点就变成了叶子节点，如此递归下去可减少子树的节点数。由此可知，这个摧毁的道路数，最大值就是 N-P。当然，最小值的做法就是像样例那样，将整个子树与总树的连接边剪掉，这样就能一次过减少几个节点。这里就需要树型 DP，通过遍历树，来“分配”这些剪掉的边，将其分至各子树中。通过 DP[i][j]表示以 i 为父节点，得出节点数为 j 的子树，至少需要在该子树内摧毁几条边。首先，对于叶子节点，自然就是 0 了。而对于非叶子节点，在 dfs 遍历其所有节点后，对于每个子节点 x，dp[i][j]=min{dp[i][k]+dp[x][j-k],0<=k<=j}，实质就是枚举 j 分到 x 的子树的分法。最后需要注意，有可能这个子树的根不是原来的根，所以对于其他子树，最后是需要+1，表示摧毁该子树根与其父节点的连边。

```
#include <iostream>
#define MAX 152
#define INF 0x3fffffff
using namespace std;

int dp[MAX][MAX];
int son[MAX], bla[MAX], root, n, p;
bool hf[MAX];



void dfs(int s)
{
    int i, j, k, temp;
    for(i = 0; i <= p; i++)
        dp[s][i] = INF;
    dp[s][1] = 0;
    k = son[s];
    while(k){
        dfs(k);
        for(i = p; i >= 0; i--){
            temp = dp[s][i]+1;
            for(j = 0; j <= i; j++){
                if(dp[k][i-j] + dp[s][j] < temp)
                    temp = dp[k][i-j] + dp[s][j];
            }
            dp[s][i] = temp;
        }
        k = bla[k];
    }
}

int slove(int root)
```

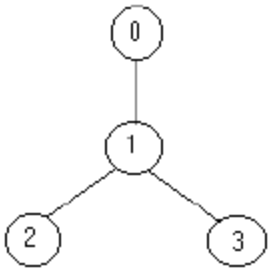
```
{
    dfs(root);
    int i, ans;
    ans = dp[root][p];
    for(i = 1; i <= n; i++){
        if(dp[i][p] < ans)
            ans = dp[i][p] + 1;
    }
    return ans;
}

int main()
{
    int i, s, t;
    while(cin >> n >> p){
        memset(son, 0, sizeof(son));
        for(i = 1; i < n; i++){
            cin >> s >> t;
            bla[t] = son[s];
            son[s] = t;
            hf[t] = true;
        }
        for(i = 1; i <= n; i++){
            if(!hf[i])
                root = i;
        }
        cout << slove(root) << endl;
    }
    return 0;
}
```

2.2 覆盖类型

例 4 战略游戏(bzoj1377)

【题目描述】

Bob 喜欢玩电脑游戏，特别是战略游戏。但是他经常无法找到快速玩过游戏的办法。现在他有个问题。他要建立一个古城堡，城堡中的路形成一棵树。他要在这棵树的结点上放置最少数目的士兵，使得这些士兵能了望到所有的路。 注意，某个士兵在一个结点上时，与该结点相连的所有边将都可以被了望到。



只需要选结点 1 放一个士兵

请你编一程序，给定一树，帮 Bob 计算出他需要放置最少的士兵。

【输入格式】

第 1 行:1 个整数 N(0＜N≤1500)，表示树中结点的数目。

第 2 行..第 N+1 行: 每行描述每个结点信息，依次为：该结点标号 i，k(后面有 k 条边与结点 i 相连)，接下来 k 个数，分别是每条边的另一个结点标号 r1，r2，...，rk。

【输出格式】

输出仅包含一个数，为所求的最少的士兵数目。

【样例输入】

```
4
0 1 1
1 2 2 3
2 0
3 0
```

【样例输出】

```
1
```

【分析】

本题实质是在一棵树上，选最少的结点覆盖所有的边（所有边满足两端中至少有一个结点被选中），被称为最小点覆盖集问题，可以用树形 DP 解决。

如何定义状态？有一维必然表示以 i 为根的子树，也可以知道状态的值表示覆盖所有的树边所需的最少结点数。仅这一维能刻画出结点 i 的不同状态吗？显然不够！

每条边被覆盖，至少 i 与它的父亲中的一个结点被选中，因此需要第二维表示要么 i 被选中，要么 i 的父亲被选中。定义 F[i][0]表示当 i 的父亲被选中时，覆盖子树所有边需要的最少结点数；F[i][1]表示当 i 被选中时，覆盖子树所有边需要的最少结点数。

当 i 的父亲被选中时，i 可以被选，也可以不被选，若 i 被选，对应的状态值即为 F[i][1]。若 i 不选，为覆盖 i 与所有儿子之间的树边，i 的每个儿子必须被选，因此有：

F[i][0] = min{F[i][1], sum(F[i.son][1])}

当 i 被选时，i 与父亲的边就由 i 覆盖了。i 的每个儿子选不选暂不管，但其状态正巧可表示为 F[i.son][0]（因为 i 被选了，i 是它各个儿子的父亲）。因此有：

F[i][1] = 1 + sum(F[i.son][0])

边界状态如何确定呢？

当 i 是叶结点时，没有儿子了，因此 F[i][1] = 1。而当叶结点的父亲被选了，它自己就可

以不被选了，因此 F[i][0] = 0。

目标状态是什么呢？稍作思考即可想到：F[root][0]

至于实现，就留作练习了。

另外，本题还可以构建二分图模型，然后用最大流算法求解。

例 5 电话网络（USACO JAN08 GOLD bzoj1700）

【题目描述】

Farmer John 决定为他的所有奶牛都配备手机，以此鼓励她们互相交流。不过，为此 FJ 必须在奶牛们居住的 N(1 <= N <= 10,000)块草地中选一些建上无线电通讯塔，来保证任意两块草地间都存在手机信号。所有的 N 块草地按 1..N 顺次编号。

所有草地中只有 N-1 对是相邻的，不过对任意两块草地 A 和 B(1 <= A <= N; 1 <= B <= N; A != B)，都可以找到一个以 A 开头以 B 结尾的草地序列，并且序列中相邻的编号所代表的草地相邻。无线电通讯塔只能建在草地上，一座塔的服务范围为它所在的那块草地，以及与那块草地相邻的所有草地。

请你帮 FJ 计算一下，为了建立能覆盖到所有草地的通信系统，他最少要建多少座无线电通讯塔。

【输入格式】

第 1 行:1 个整数，N

第 2..N 行: 每行为 2 个用空格隔开的整数 A、B，为两块相邻草地的编号

【输出格式】

第 1 行: 输出 1 个整数，即 FJ 最少建立无线电通讯塔的数目

【输入样例】

```
5
1 3
5 2
4 3
3 5
```

【输出样例】

```
2
```

【分析】

这个题实质在在一棵树上，选中一个结点，可以覆盖自身与相邻结点。求用最少的结点覆盖所有的结点。这被称为树的最小支配集问题，

分析发现，一个结点可能处在未被覆盖的状态，也可能在上边建塔，也可能被覆盖。而被覆盖又可分为被父结点覆盖和被子代结点覆盖。我们限制状态为是否被父结点覆盖，更容易厘清转移关系，建立方程。

令 F[i][j]表示以 i 为根的子树，结点 i 处于以下 3 个值（0、1、2）时，最少建的塔数：

0：i 被父结点覆盖，i 自身建不建塔无所谓；

1：i 未被父结点覆盖，i 自身需要建塔或者被子代结点覆盖

2：i 自身建塔

再看状态转移方程。当 i 被父结点覆盖时，若 i 自身建塔，状态为 F[i][2]；若 i 不建塔，则它的儿子的状态处于不受 i 覆盖的状态，即：

F[i][0] = min{ F[i][2], sum(F[i.son][1]) }

当 i 不被父结点覆盖时，若 i 建塔，状态为 F[i][2]；若 i 不建塔，则某个儿子必须建塔，否则 i 就无法被覆盖。到底哪个儿子建塔呢？比较了才知道：sum(F[i.son][1])表示所有儿子处于不受 i 覆盖的状态时共需要建多少塔。若第 j 个儿子要建塔，则它的状态为 F[j][2]，而不再为 F[j][1]，故要从 sum(F[i.son][1])中减去。即某个儿子建塔时的状态表示为：

sum(F[i.son][1] + min{F[j][2]-F[j][1]}

从而：F[i][1] = min{ F[i][2], sum(F[i.son][1] + min{F[j][2]-F[j][1]}}

当 i 自身建塔时，它的儿子将处在受 i 覆盖的状态下，因此有：

F[i][2] = 1 + sum(F[i.son][0])

边界状态的处理比较简单，就留作实现时自行处理了。

本题是棵无根树，在实现时要先 DFS 建树后，再做 DP。

本题另有一个非常巧妙的贪心算法。可参看某英文版的 FLASH 视频。

例 6 没有上司的晚会(bzoj1378)

【题目描述】

Ural 大学有 N 个职员，编号为 1~N。他们有从属关系，也就是说他们的关系就像一棵以校长为根的树，父结点就是子结点的直接上司。每个职员有一个快乐指数。现在有个周年庆宴会，要求与会职员的快乐指数最大。但是，没有职员愿和直接上司一起参加宴会。

【输入格式】

第一行一个整数 N。(1≤N≤6000)

接下来 N 行，第 i+1 行表示 i 号职员的快乐指数 Ri。(-128≤Ri≤127)

接下来 N-1 行，每行输入一对整数 L,K。表示 K 是 L 的直接上司。

最后一行输入 0,0。

【输出格式】

第 1 行：输出最大的快乐指数。

【样例输入】

7

1

1

1

1

1

1

1

1 3

2 3
6 4
7 4
4 5
3 5
0 0

【样例输出】
5

【分析】

本题实质是选出尽量多的结点，使得任何两个结点均不相邻，并且结点的欢乐值之和最大。这个问题是树的最大独立集的一个变形(参见刘汝佳教材)。

有了前面两道覆盖类型题目的分析经验。本题的状态定义与转移方程都比较简单。

定义 F[i][0]表示 i 的父亲没有被选的最大欢乐值，F[i][1]表示 i 的父亲被选的最大欢迎值。如果 i 的父亲没有被选，则 i 可选，也可不选。若 i 被选时，状态值由它的欢乐值与所有儿子子树的欢迎乐组成，表示为 happy[i] + sum(F[i.son][1])。若 i 不选时，i 的儿子们处于不受 i 影响的状态下，状态值表示为 sum(F[i.son][0])。因此有：

F[i][0] = max{ happy[i] + sum(F[i.son][1]), sum(F[i.son][0]) }

当 i 的父亲要选时，则 i 一定不可选，因此有：

F[i][1] = sum(F[i.son][0])

以这两个转移方程，再加上边界状态，就可以求出整个问题的最优解了。

也可以定义状态 F[i][0]表示结点 i 自身未被选，F[i][1]表示结点 i 自身被选，然后再写出状态转移方程。同样可以解决本题。

因此，对于动态规划问题，状态定义的含义不同，转移方程也就可能不同，但最终可以做到殊路同归。但同学们在平时的训练时，要体会经典问题状态定义的精髓，总结出规律性的东西。竞赛时就能胸有成竹，以不变应万变了。

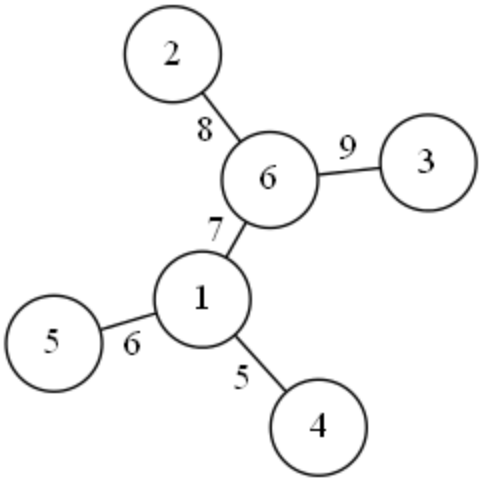
2.3 路径类型

例 7 北极地区的路（PKU2631）

【题目描述】

寒冷、人烟稀少的北极地区修建与维护道路时非常昂贵的。因此，修路时只会在任意两个村子之间形成一条唯一的路径，并且中途不会经过其它村子两次。给出一些村子之间的道路信息，使得任意一个村子都能够与任意其它村子连通。

你的任务是找出相距最远的两个村子之间的距离。例如，下图中 3 与 5 两个村子相距最远，距离为 22。



最多有 10,000 个村子，村子从 1 开始，顺序编号。

【输入格式】
有若干行，每行 3 个整数 i, j, k，i 和 j 是两个村子的编号，k 表示它们之间道路的长度。道路是双向的。

【输出格式】
第 1 行：1 个整数，表示相距最远的两个村子的距离。

【输入样例】
5 1 6
1 4 5
6 3 9
2 6 8
6 1 7

【输出样例】
22

【分析】

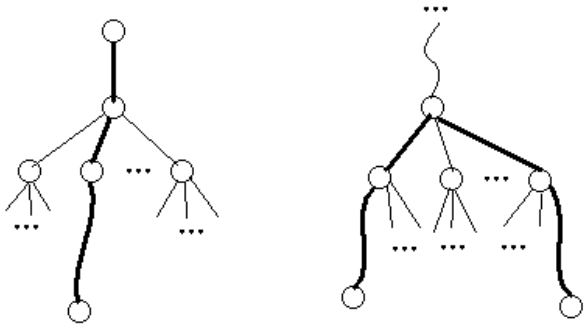
这是一个经典的求树的最长链问题。树的最长链，也称树的直径。寻找树的直径，可以用树形 DP 求解。其关键思想是：用 DFS 遍历树。对于必然存在的树的直径，遍历时必然会遇到直径上的结点（遍历是对树上每个结点作一次访问）。这时会存在两种情况：

1. DFS 的起点恰好是直径的一个端点
2. DFS 的起点不是直径的任何一个端点

对于情况 1，树的直径可表示为伸向子树的最大“深度”。对于情况 2，复杂一些。设想，从某个非直径端点的结点出发，进行 DFS。当第一次访问到直径上的某个结点时 u 时，后续的过程会怎样？

因为 u 是直径上第一个被访问的结点，DFS 又会遍历整棵树的每个结点，所以从 u 出发递归时，必然会访问到直径的一个端点 v 和另一个端点 w。换言之，u 将是包含树的直径的那棵子树的根，u-v 和 u-w 是在 DFS 时形成的两条以 u 为根的路径。u-v 和 u-w 是所有以 u 为根的路径中的其中两条。它们要成为树的直径的两段，在长度上应该满足什么条件呢？

显然 u-v 和 u-w 应该是所有以 u 为根的路径中最长的和次长的两条。



左图指示了从直径的一个端点 DFS 的情况。右图指示了直径由最长与次长的两段相加的情况。下面考虑状态定义：

DEP[i]表示以 i 为根的子树的最大深度；

F[i]表示以 i 为根的子树，包含结点 i 的最长链的长度。

考虑转移方程：

DEP[i] = max {DEP[j] + w[i][j]}　j 是深搜时 i 的子结点，w[i][j]是边(i,j)的长度。

F[i] = max {DEP[i], DEP[j] + w[i][j] + DEP[k] + w[i][k] | j≠k}

边界状态非常简单：如果 i 是叶结点，则 DEP[i] = 0，F[i] = 0。

在实现时，只做一遍 DFS 遍历，在回朔的时候分别计算 DEP 和 F 的值。对于 F 值的计算，由于节点 j 和 k 之间没有关联，所以我们只需要用两个局部变量 max1, max2 分别保存 i 的儿子结点中(dep(j)+w(i,j))的最大值与次大值，最后累加即可。不过要注意，最大值与次大值所对应的儿子节点一定要不同才行。

本题还有一个用两次 BFS 求树的最长链的方法，留为思考。

例 8 铲雪车问题

【题目描述】

大雪覆盖了整座城市，市政府要求冬季服务部门尽快将一些街道（列在一份清单中）的积雪清除掉以恢复交通。整个城市由许多交叉路口和街道构成，当然任意两个交叉路口都是直接或间接连通的。清单给出了最少的街道，使得这些街道的积雪清除后任意两个交叉路口之间有且仅有一条通路。冬季服务部门只有一辆铲雪车和一名司机，这辆铲雪车的出发点位于某个交叉路口。无论街道上有没有积雪，铲雪车每前进一米都要消耗一升燃料。冬季服务部门要求司机在铲除清单上的所有街道的积雪的前提下，消耗燃料最少，铲完后车可以停在任意交叉路口。

【输入格式】

第 1 行：2 个整数 N,S。（1 < =N < = 100000,1 < = S < =N），N 为交叉路口总数，S 为铲雪车出发的路口序号。路口的编号为 1～N。

接下来的 N-1 行为清单上的街道，每一行包含三个用空格隔开的整数 A,B,C，表示一条从交叉路口 A 到交叉路口 B 的街道，C 为该街道的长度。单位为米，1 < = C < =1000

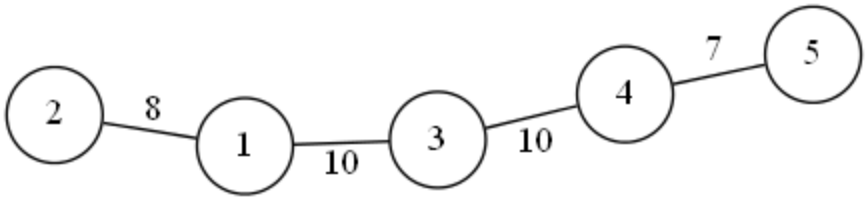
【输出格式】

第 1 行：一个整数，表示铲掉所有积雪所需的最少燃料。

【样例输入】

5 1
1 2 8
1 3 10
3 4 10
4 5 7

【样例输出】
43
【分析】



对于样例数据，画出图形。首先根据题目描述应该明白，如果把路口视为顶点，街道视为边，则整个图是一棵树。

模拟铲雪车行走过程，会发现，由于每条边都必须走，为了尽量节约油耗，最好能不走或少走同一条边两次。这能办到吗？

思考后发现，由于铲雪车从起点出发，最好能一次性铲除一条到叶结点路径上的所有边，然后再返回到某个位置，否则这条路径还要走第二遍，那该返回到什么位置呢？

显然应该返回到一个有另一条未铲雪路径的结点处。当把这条路径铲完后，再次返回此处，如果仍有其它路径，照此办理。若没有，则沿父路径回溯。

总结这个铲雪的过程会发现一个性质：每条边都一定走两次，一次深入，一次返回。那题目所要求的尽可能节油，体现在何处呢？

由于结束全部工作后，铲雪车可停留在任意位置。这样，当它铲完最后一条路径的雪后，就不必再返回。那它该选择停留在哪条路径的末端呢？

由于所有树边权*2 是个定值，要想最节油，必然减掉长度最大的一条路径，即从起点算起的一条最长链。由此，本问题得到解决。

【拓展问题 1】

如果在同一起点，有两台铲雪车，其它条件都不变，问题该如何解决呢？

我们考虑用树形 DP 加以解决。定义以下 3 种状态：

F[i][1]：表示派出一台铲雪车，扫完以 i 为根的子树，并且返回结点 i 的费用。

F[i][2]：表示派出一台铲雪车，扫完以 i 为根的子树，但是不返回结点 i 的费用。

F[i][3]：表示派出二台铲雪车，扫完以 i 为根的子树，但是不返回结点 i 的费用。

考虑状态转移方程。

若派出一台铲雪车，从结点 i 出发，扫完子树上所有边，再返回结点 i，必须每条边要走 2 遍，写方程时，可考虑先返回到 i 的儿子结点，再走一条父子边，因此：

F[i][1] = sum{F[j][1] + 2*w(i, j) | j 是 i 的儿子}

若派出一台铲雪车，扫完子树所有边上的雪后，不返回结点 i，则考虑它停留在 i 的不同的儿子子树内。设它停留在儿子 j 的子树内，首先意识到，若 i 有多个儿子，一定要先走

完其它儿子子树后，最后再进入要停留的子树 j。这样，少走的路程如何表示？

我们在分析树形 DP 时，递归的思维一定要随时随地自如运用。所以，我们先考虑从结点 i 到 j 这一段，少走多少路程？答案是：w(i,j)。因为停留在子树 j 中，所以也不用返回 j（贪心地想，这时铲雪车一定要停留在叶结点上）。再用整体与部分的关系，可把状态表示为：

F[i][2] = min{F[i][1] – F[j][1] – w(i,j) + F[j][2] | j 是 i 的儿子}

若派出二台铲雪车，可以证明，这两台车要么分别沿 F[i][2]的最小值 fi 和次小值 si 方向，要么一起沿 fi 方向。因此

F[i][3] = min{ F[i][1] - F[fi][1] - w(i,fi) + F[fi][2] - F[si][1] - w(i,si) + F[si][2] ,
F[i][1] - F[fi][1] + F[fi][3] }

最终的解 ans = min { F[root][2], F[root][3] }

可以在 POJ.ORG 上提交 1849 即为此题。

本题也可用贪心的思想求解。关键是看穿本题的本质仍然是求一条树的最长链，ans = sum(边权) * 2 – sum(树的直径)。即若起点是树的直径的一个端点，则用一台铲雪车工作即可。否则，两台铲雪车分别工作，铲完其它路径上的雪，最后分别走直径上的一段，分别停在直径的两个端点上。

【拓展问题 2】

若有 K 台铲雪车，又该如何解决呢？

定义 F[i][k_in][k_out]表示扫除子树 i 所有雪的情况下，进入 k_in 辆车，返回 k_out 辆车，此时的最少油耗。状态转移方程留作思考。

2.4 进阶问题

例 9 河流(IOI2005)

【问题描述】

几乎整个 Byteland 王国都被森林和河流所覆盖。小点的河汇聚到一起，形成了稍大点的河。就这样，所有的河水都汇聚并流进了一条大河，最后这条大河流进了大海。这条大河的入海口处有一个村庄——名叫 Bytetown

在 Byteland 国，有 n 个伐木的村庄，这些村庄都座落在河边。目前在 Bytetown，有一个巨大的伐木场，它处理着全国砍下的所有木料。木料被砍下后，顺着河流而被运到 Bytetown 的伐木场。Byteland 的国王决定，为了减少运输木料的费用，再额外地建造 k 个伐木场。这 k 个伐木场将被建在其他村庄里。这些伐木场建造后，木料就不用都被送到 Bytetown 了，它们可以在 运输过程中第一个碰到的新伐木场被处理。显然，如果伐木场座落的那个村子就不用再付运送木料的费用了。它们可以直接被本村的伐木场处理。

注意：所有的河流都不会分叉，也就是说，每一个村子，顺流而下都只有一条路——到 bytetown。

国王的大臣计算出了每个村子每年要产多少木料，你的任务是决定在哪些村子建设伐木场能获得最小的运费。其中运费的计算方法为：每一块木料每千米 1 分钱。

编一个程序：

1．从文件读入村子的个数，另外要建设的伐木场的数目，每年每个村子产的木料的块数以及河流的描述。

2．计算最小的运费并输出。

【输入格式】

第一行 包括两个数 n（2<=n<=100），k（1<=k<=50,且 k<=n）。n 为村庄数，k 为要建的伐木场的数目。除了 bytetown 外，每个村子依次被命名为 1，2，3……n，bytetown 被命名为 0。接下来 n 行，每行包涵 3 个整数

wi——每年 i 村子产的木料的块数 （0<=wi<=10000）

vi——离 i 村子下游最近的村子（或 bytetown）（0<=vi<=n）

di——vi 到 i 的距离(km)。（1<=di<=10000）

保证每年所有的木料流到 bytetown 的运费不超过 2000,000,000 分

50％的数据中 n 不超过 20。

【输出格式】

输出最小花费，精确到分。

【样例输入】

4 2

1 0 1

1 1 10

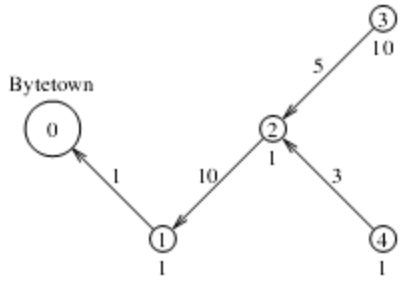
10 2 5

1 2 3

【样例输出】

4

【样例说明】



圈内数字为村庄编号，下方数字为存储木料块数，边权为河的长度。伐木场应建在村庄 2 和 3。

【分析】

这是一个综合性的树形 DP 题。比较明显的，是要建的伐木场的数量是一个约束条件，因此这是一个双约束类型问题，要在当前结点的各个儿子之间分配要求建的伐木场。对于此，前文已经详细提供了两种解决方案：多叉转二叉或者背包。随着分析的深入，会发现如果在当前结点不建伐木场，那么当前结点与子树中的一部分结点的木材需要沿着父路径运送到最近的一个伐木场。也就是说，在状态中需要有一维来指示当前结点父路径上最近的伐木场的位置。因此，定义状态为：

F[i][j][k] 表示以 i 为根的子树建 j 个伐木场，离 i 最近的往父路径上的伐木场是 k。如果用多叉转二叉的方式来存储树结构，那么状态方程为：

若 i 建伐木场，则 f[i][j][k] = min{f[i.lch][x][i]+f[rch][j-x-1][i]}

若 i 不建伐木场，则 f[i][j][k] = min{f[lson][x][k]+f[rson][j-x][k]}+dist[i][k]*wood[i]
其中 dist[i][k]表示结点 i 与 k 的路径长度，wood[i]表示在村子 i 的木材数量。
以此分析为基础，实现本题不再困难。

例 10 贪吃的九头龙(NOI2002)

【问题描述】

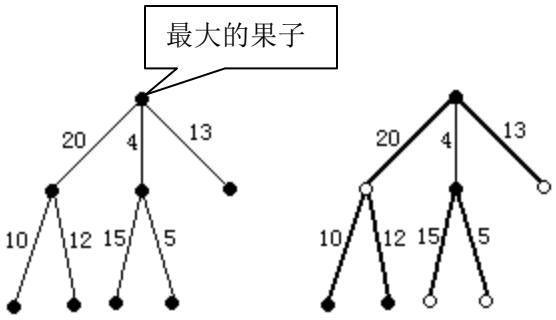
传说中的九头龙是一种特别贪吃的动物。虽然名字叫“九头龙”，但这只是说它出生的时候有九个头，而在成长的过程中，它有时会长出很多的新头，头的总数会远大于九，当然也会有旧头因衰老而自己脱落。

有一天，有 M 个脑袋的九头龙看到一棵长有 N 个果子的果树，喜出望外，恨不得一口把它全部吃掉。可是必须照顾到每个头，因此它需要把 N 个果子分成 M 组，每组至少有一个果子，让每个头吃一组。

这 M 个脑袋中有一个最大，称为“大头”，是众头之首，它要吃掉恰好 K 个果子，而且 K 个果子中理所当然地应该包括唯一的一个最大的果子。果子由 N-1 根树枝连接起来，由于果树是一个整体，因此可以从任意一个果子出发沿着树枝“走到”任何一个其他的果子。对于每段树枝，如果它所连接的两个果子需要由不同的头来吃掉，那么两个头会共同把树枝弄断而把果子分开；如果这两个果子是由同一个头来吃掉，那么这个头会懒得把它弄断而直接把果子连同树枝一起吃掉。当然，吃树枝并不是很舒服的，因此每段树枝都有一个吃下去的“难受值”，而九头龙的难受值就是所有头吃掉的树枝的“难受值”之和。

九头龙希望它的“难受值”尽量小，你能帮它算算吗？

例如图 1 所示的例子中，果树包含 8 个果子，7 段树枝，各段树枝的“难受值”标记在了树枝的旁边。九头龙有两个脑袋，大头需要吃掉 4 个果子，其中必须包含最大的果子。即 N=8， M=2， K=4：



图一

大头吃 4 个果子，用实心点标识；
小头吃 4 个果子，用空心点标识；
九头龙的难受值为 4，因为图中用细边标记的树枝被大头吃掉了。

图二

图一描述了果树的形态，图二描述了最优策略。

【输入格式】

第 1 行包含三个整数 N (1<=N<=300)，M (2<=M<=N)，K (1<=K<=N)。N 个果子依次编号 1,2,…,N，且最大的果子的编号总是 1。

第 2 行到第 N 行描述了果树的形态，每行包含三个整数 a (1<=a<=N)， b (1<=b<=N)，c (0<=c<=105)，表示存在一段难受值为 c 的树枝连接果子 a 和果子 b。

【输出格式】
仅有一行，包含一个整数，表示在满足“大头”的要求的前提下，九头龙的难受值的最小值。如果无法满足要求，输出-1。

【样例输入】
8 2 4
1 2 20
1 3 4
1 4 13
2 5 10
2 6 12
3 7 15
3 8 5

【样例输出】
4

练习题

1. PKU 1383 Labyrinth 迷宫
2. PKU 1935 Journey
3. PKU 3513Let's Go to the Movies
4. PKU 1947 Rebuilding Roads
5. PKU 1770 Special Experiment
6. PKU 1192 最优连通子集
   大意：
   平面上四领域点组成的树形结构
   每个点有个权值
   求一颗子树使得子树各点权值和最大

   限制：
   点数<=1000
   -10^6<=坐标<=10^6
   -100<=权值<=100
7. PKU 1694 An Old Stone Game
   大意：
   树形结构的棋盘上，规则是这样的：
   一开始篮子里有些石子
   每次可以在空的叶子节点上放一颗石子
   当某个节点的所有直接子节点都有石子时，可以移除这些石子并在这个节点上放上一颗石子
   移除的石子放回篮子

求最少要多少石子才能让根节点上有石子

限制：

节点数<=200

8. PKU 1741 Tree

大意：

求带边权的树中有多少对节点间距离不超过 k

限制：

节点数<=10,000

0<边权<1001

9. PKU 1848 Tree

大意：

在树上添加最少数量条边使得每个节点位于正好一个环上

限制：

节点数<=100

10. PKU 3585 Accumulation Degree

大意：

带边容量限制的树

一个节点的 accumulation degree 是以它为源，其他叶节点为汇的最大流

一棵树的 accumulation degree 是它所有节点的 accumulation degree 的最大值

求树的 accumulation degree

限制：

0<=容量<=200,000

节点数<=200,000

11. SGU 149 Computer Network

大意：求树上的每个节点能到达的最远距离。