树形 dp

2024 年 2 月 4 日

树形 dp

定义

树形 dp，即在树上进行的 dp，通常为利用 dp 解决树上的问题。

树形 dp

定义

树形 dp，即在树上进行的 dp，通常为利用 dp 解决树上的问题。

因为树的遍历需要用递归实现，因此树形 dp 的状态转移也是在递归函数中实现的。
并且，通常是树上儿子结点的某些状态转移到父亲结点的某些状态。
所以，容易发现，叶子结点的状态没有被其他状态转移到，即初始状态，需要进行初始化。

树形 dp

定义

树形 dp，即在树上进行的 dp，通常为利用 dp 解决树上的问题。

因为树的遍历需要用递归实现，因此树形 dp 的状态转移也是在递归函数中实现的。
并且，通常是树上儿子结点的某些状态转移到父亲结点的某些状态。
所以，容易发现，叶子结点的状态没有被其他状态转移到，即初始状态，需要进行初始化。

因此，通常情况下，树形 dp 的递归函数形态如下：

```
void DFS (int u) { // 递归转移以 u 为根的子树的所有结点的状态
    if u is the leaf // 如果 u 是叶子结点
        f[u] = 0 // 进行初始化
    int v is u's son: // 遍历 u 的所有儿子
        DFS (v) // 向下递归
        f[v] to f[u] // 将结点 v 的状态转移到结点 u 的状态
}
```

树形 dp

没有上司的舞会（Luogu P1352）

某大学有 n 个职员，编号为 $1 \sim n$。他们之间有从属关系，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。现在有个周年庆宴会，宴会每邀请来一个职员都会增加一定的快乐指数 a_i，但是呢，如果某个职员的直接上司来参加舞会了，那么这个职员就无论如何也不肯来参加舞会了。所以，请你编程计算，邀请哪些职员可以使快乐指数最大，求最大的快乐指数。

树形 dp

没有上司的舞会（Luogu P1352）

某大学有 n 个职员，编号为 $1 \sim n$。他们之间有从属关系，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。现在有个周年庆宴会，宴会每邀请来一个职员都会增加一定的快乐指数 a_i，但是呢，如果某个职员的直接上司来参加舞会了，那么这个职员就无论如何也不肯来参加舞会了。所以，请你编程计算，邀请哪些职员可以使快乐指数最大，求最大的快乐指数。

定义 $f(i, 0/1)$ 表示以 i 为根的子树，且 i 结点不参加/参加舞会，所能获得的最大的快乐指数。（第二维中记录了 i 结点的参加状态，0 表示不参加，1 表示参加）

树形 dp

没有上司的舞会（Luogu P1352）

某大学有 n 个职员，编号为 $1 \sim n$。他们之间有从属关系，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。现在有个周年庆宴会，宴会每邀请来一个职员都会增加一定的快乐指数 a_i，但是呢，如果某个职员的直接上司来参加舞会了，那么这个职员就无论如何也不肯来参加舞会了。所以，请你编程计算，邀请哪些职员可以使快乐指数最大，求最大的快乐指数。

定义 $f(i, 0/1)$ 表示以 i 为根的子树，且 i 结点不参加/参加舞会，所能获得的最大的快乐指数。（第二维中记录了 i 结点的参加状态，0 表示不参加，1 表示参加）

思考

为什么要开第二维状态？

树形 dp

没有上司的舞会（Luogu P1352）

某大学有 n 个职员，编号为 $1 \sim n$。他们之间有从属关系，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。现在有个周年庆宴会，宴会每邀请来一个职员都会增加一定的快乐指数 a_i，但是呢，如果某个职员的直接上司来参加舞会了，那么这个职员就无论如何也不肯来参加舞会了。所以，请你编程计算，邀请哪些职员可以使快乐指数最大，求最大的快乐指数。

定义 $f(i, 0/1)$ 表示以 i 为根的子树，且 i 结点不参加/参加舞会，所能获得的最大的快乐指数。（第二维中记录了 i 结点的参加状态，0 表示不参加，1 表示参加）

思考

为什么要开第二维状态？

题目中要求，当父亲结点参加时，儿子结点便不能参加。因此，当儿子结点的状态转移到父亲结点的状态时，如果儿子结点在它的状态中是参加的状态，那么在转移时，该状态便不能转移到父亲结点也参加的状态。所以，要把整个状态区分为 i 结点参加和不参加的两种状态，需要开第二维状态。

树形 dp

状态定义

$f(i,0/1)$ 表示以 i 为根的子树，且 i 结点不参加/参加舞会，所能获得的最大的快乐指数。（第二维中记录了 i 结点的参加状态，0 表示不参加，1 表示参加）

树形 dp

状态定义

$f(i,0/1)$ 表示以 i 为根的子树，且 i 结点不参加/参加舞会，所能获得的最大的快乐指数。（第二维中记录了 i 结点的参加状态，0 表示不参加，1 表示参加）

考虑状态转移：

树形 dp

状态定义

$f(i,0/1)$ 表示以 i 为根的子树，且 i 结点不参加/参加舞会，所能获得的最大的快乐指数。（第二维中记录了 i 结点的参加状态，0 表示不参加，1 表示参加）

考虑状态转移：

- 如果 i 结点参加，那么它的儿子便都不能参加，即转移方程为：

$$f(i,1)=\sum_{j\ is\ i's\ son} f(j,0)+a_i$$

树形 dp

状态定义

$f(i,0/1)$ 表示以 i 为根的子树，且 i 结点不参加/参加舞会，所能获得的最大的快乐指数。（第二维中记录了 i 结点的参加状态，0 表示不参加，1 表示参加）

考虑状态转移：

- 如果 i 结点参加，那么它的儿子便都不能参加，即转移方程为：

$$f(i,1)=\sum_{j\ is\ i's\ son} f(j,0)+a_i$$

- 如果 i 结点不参加，那么它的儿子可以参加也可以不参加，即转移方程为：

$$f(i,0)=\sum_{j\ is\ i's\ son} \max\{f(j,0),f(j,1)\}$$

树形 dp

状态定义

$f(i, 0/1)$ 表示以 i 为根的子树，且 i 结点不参加/参加舞会，所能获得的最大的快乐指数。（第二维中记录了 i 结点的参加状态，0 表示不参加，1 表示参加）

树形 dp

状态定义

$f(i,0/1)$ 表示以 i 为根的子树，且 i 结点不参加/参加舞会，所能获得的最大的快乐指数。（第二维中记录了 i 结点的参加状态，0 表示不参加，1 表示参加）

根据状态定义，便也容易得知叶子结点的初始化为：

$$f(i,0)=0, f(i,1)=a_i$$

树形 dp

状态定义

$f(i,0/1)$ 表示以 i 为根的子树，且 i 结点不参加/参加舞会，所能获得的最大的快乐指数。（第二维中记录了 i 结点的参加状态，0 表示不参加，1 表示参加）

根据状态定义，便也容易得知叶子结点的初始化为：

$$f(i,0)=0, f(i,1)=a_i$$

于是，根据最初提到的递归函数的形态，便可以实现本题的代码。

树形 dp

状态定义

$f(i,0/1)$ 表示以 i 为根的子树，且 i 结点不参加/参加舞会，所能获得的最大的快乐指数。（第二维中记录了 i 结点的参加状态，0 表示不参加，1 表示参加）

根据状态定义，便也容易得知叶子结点的初始化为：

$$f(i,0)=0, f(i,1)=a_i$$

于是，根据最初提到的递归函数的形态，便可以实现本题的代码。

时空复杂度

时间复杂度是 $\mathcal{O}(n)$ 的，空间复杂度也是 $\mathcal{O}(n)$ 的。

树形 dp

状态定义

$f(i,0/1)$ 表示以 i 为根的子树，且 i 结点不参加/参加舞会，所能获得的最大的快乐指数。（第二维中记录了 i 结点的参加状态，0 表示不参加，1 表示参加）

根据状态定义，便也容易得知叶子结点的初始化为：

$$f(i,0)=0, f(i,1)=a_i$$

于是，根据最初提到的递归函数的形态，便可以实现本题的代码。

时空复杂度

时间复杂度是 $\mathcal{O}(n)$ 的，空间复杂度也是 $\mathcal{O}(n)$ 的。

因为转移的时间复杂度是 $\mathcal{O}(1)$ 的，所以本质上只有一个树的遍历，因此时间复杂度和空间复杂度都是 $\mathcal{O}(n)$ 的。

树形 dp

代码实现：

```
void DFS (int u) {
    // vec[u] 用来存储 u 的所有儿子
    if (vec[u].size () == 0) { // u 是叶子结点
        f[u][0] = 0, f[u][1] = a[u]; // 初始化
        return;
    }
    f[u][0] = f[u][1] = 0;
    for (int i = 0; i < vec[u].size (); i ++) { // 枚举 u 的所有儿子
        int v = vec[u][i];
        DFS (v); // 先向儿子递归，递归返回后再进行状态转移
        f[u][0] += max (f[v][0], f[v][1]); // 状态转移方程
        f[u][1] += f[v][0]; // 状态转移方程
    }
    f[u][1] += a[u]; // 状态转移方程
}
```

树形 dp

选课（Luogu P2014）

现在有 n 门课程，第 i 门课程的学分为 a_i，每门课程有零门或一门先修课，有先修课的课程需要先学完其先修课，才能学习该课程。一位学生一共要学习 m 门课程，求其能获得的最多学分是多少。

数据范围：$n, m \leq 300$

树形 dp

选课（Luogu P2014）

现在有 n 门课程，第 i 门课程的学分为 a_i，每门课程有零门或一门先修课，有先修课的课程需要先学完其先修课，才能学习该课程。一位学生一共要学习 m 门课程，求其能获得的最多学分是多少。

数据范围：$n, m \leq 300$

如果将先修课这个限制去掉，同样每门课只能学一次，课程与课程之间没有先后顺序，便可以将它们放在一个线性序列上，前后顺序无所谓。容易发现，这就是一个简单的 01 背包问题（背包容积为 m，第 i 门课的体积是 1，价值是 a_i）。

树形 dp

选课（Luogu P2014）

现在有 n 门课程，第 i 门课程的学分为 a_i，每门课程有零门或一门先修课，有先修课的课程需要先学完其先修课，才能学习该课程。一位学生一共要学习 m 门课程，求其能获得的最多学分是多少。

数据范围：$n, m \leq 300$

如果将先修课这个限制去掉，同样每门课只能学一次，课程与课程之间没有先后顺序，便可以将它们放在一个线性序列上，前后顺序无所谓。容易发现，这就是一个简单的 01 背包问题（背包容积为 m，第 i 门课的体积是 1，价值是 a_i）。

那么加上先修课的限制，课程与课程之间就有了先后顺序。并且，题目中每门课程有零门或一门先修课的性质与森林结构的每个结点最多只有一个父亲的性质类似。于是，所有的课程便形成了一个森林结构。

树形 dp

选课（Luogu P2014）

现在有 n 门课程，第 i 门课程的学分为 a_i，每门课程有零门或一门先修课，有先修课的课程需要先学完其先修课，才能学习该课程。一位学生一共要学习 m 门课程，求其能获得的最多学分是多少。

数据范围：$n, m \leq 300$

如果将先修课这个限制去掉，同样每门课只能学一次，课程与课程之间没有先后顺序，便可以将它们放在一个线性序列上，前后顺序无所谓。容易发现，这就是一个简单的 01 背包问题（背包容积为 m，第 i 门课的体积是 1，价值是 a_i）。

那么加上先修课的限制，课程与课程之间就有了先后顺序。并且，题目中每门课程有零门或一门先修课的性质与森林结构的每个结点最多只有一个父亲的性质类似。于是，所有的课程便形成了一个森林结构。

森林结构太过散乱，于是不妨建立一个虚根，作为森林中每一棵树的根结点的父亲，便可以将整个森林转化为一棵树，背包问题也就搬到了树上。这便是树形 dp 中的经典问题——树上背包。

树形 dp

选课（Luogu P2014）

现在有 n 门课程，第 i 门课程的学分为 a_i，每门课程有零门或一门先修课，有先修课的课程需要先学完其先修课，才能学习该课程。一位学生一共要学习 m 门课程，求其能获得的最多学分是多少。

数据范围：$n, m \leq 300$

树形 dp

选课（Luogu P2014）

现在有 n 门课程，第 i 门课程的学分为 a_i，每门课程有零门或一门先修课，有先修课的课程需要先学完其先修课，才能学习该课程。一位学生一共要学习 m 门课程，求其能获得的最多学分是多少。

数据范围：$n, m \leq 300$

定义 $f(u, i, j)$ 表示以 u 为根的子树中，已经遍历了 u 结点的前 i 棵子树，选了 j 门课程，所能获得的最大学分。

树形 dp

选课（Luogu P2014）

现在有 n 门课程，第 i 门课程的学分为 a_i，每门课程有零门或一门先修课，有先修课的课程需要先学完其先修课，才能学习该课程。一位学生一共要学习 m 门课程，求其能获得的最多学分是多少。

数据范围：$n, m \leq 300$

定义 $f(u, i, j)$ 表示以 u 为根的子树中，已经遍历了 u 结点的前 i 棵子树，选了 j 门课程，所能获得的最大学分。

转移时，还是跟上题一样递归求解，只不过要加上背包 dp 的一些元素。即依次枚举 u 的儿子结点 v，再枚举 v 子树中选课个数，将其合并到 u 的状态中。便可得到状态转移方程：

树形 dp

选课（Luogu P2014）

现在有 n 门课程，第 i 门课程的学分为 a_i，每门课程有零门或一门先修课，有先修课的课程需要先学完其先修课，才能学习该课程。一位学生一共要学习 m 门课程，求其能获得的最多学分是多少。
数据范围：$n, m \leq 300$

定义 $f(u, i, j)$ 表示以 u 为根的子树中，已经遍历了 u 结点的前 i 棵子树，选了 j 门课程，所能获得的最大学分。
转移时，还是跟上题一样递归求解，只不过要加上背包 dp 的一些元素。即依次枚举 u 的儿子结点 v，再枚举 v 子树中选课个数，将其合并到 u 的状态中。便可得到状态转移方程：

$$f(u, i, j+k) = \max_{v \ is \ u's \ son \ \& \ j \leq sum_u \ \& \ k \leq siz_v} \{f(u, i-1, j) + f(v, num_v, k)\}$$

其中 sum_u 表示 u 前 $i-1$ 棵子树大小之和，siz_v 表示以 v 为根的子树大小，num_v 表示 v 的儿子个数。

树形 dp

状态定义及状态转移方程

$f(u,i,j)$ 表示以 u 为根的子树中，已经遍历了 u 结点的前 i 棵子树，选了 j 门课程，所能获得的最大学分。

$$f(u,i,j+k)=\max_{v\ is\ u's\ son\ \&\ j\le sum_u\ \&\ k\le siz_v}\{f(u,i-1,j)+f(v,num_v,k)\}$$

树形 dp

状态定义及状态转移方程

$f(u,i,j)$ 表示以 u 为根的子树中，已经遍历了 u 结点的前 i 棵子树，选了 j 门课程，所能获得的最大学分。

$$f(u,i,j+k)=\max_{v\ is\ u's\ son\ \&\ j\le sum_u\ \&\ k\le siz_v}\{f(u,i-1,j)+f(v,num_v,k)\}$$

第二维状态是可以压掉的，并且需要把 j 改为倒序枚举（有点类似于 01 背包中的压维）。

树形 dp

状态定义及状态转移方程

$f(u,i,j)$ 表示以 u 为根的子树中，已经遍历了 u 结点的前 i 棵子树，选了 j 门课程，所能获得的最大学分。

$$f(u,i,j+k)=\max_{v\ is\ u's\ son\ \&\ j\le sum_u\ \&\ k\le siz_v}\{f(u,i-1,j)+f(v,num_v,k)\}$$

第二维状态是可以压掉的，并且需要把 j 改为倒序枚举（有点类似于 01 背包中的压维）。也可以通过开中介数组的形式，来把第二维压掉，就不用太多的去考虑枚举顺序的问题了。

状态定义及状态转移方程

$f(u,i,j)$ 表示以 u 为根的子树中，已经遍历了 u 结点的前 i 棵子树，选了 j 门课程，所能获得的最大学分。

$$f(u,i,j+k)=\max_{v\ is\ u's\ son\ \&\ j\le sum_u\ \&\ k\le siz_v}\{f(u,i-1,j)+f(v,num_v,k)\}$$

第二维状态是可以压掉的，并且需要把 j 改为倒序枚举（有点类似于 01 背包中的压维）。也可以通过开中介数组的形式，来把第二维压掉，就不用太多的去考虑枚举顺序的问题了。

新的状态定义即为：$f(u,j)$ 表示以 u 为根的子树中，选了 j 门课程，所能获得的最大学分。只有当 u 的所有儿子递归遍历完，且都状态转移一一合并到了 $f(u,j)$ 中，$f(u,j)$ 就表示成了正确的值，用于 u 的父亲转移。对于 $f(u,j)$ 合并前的初始化即为：

$$f(u,1)=a_u$$

树形 dp

代码实现（倒序枚举）；

```
void DFS (int u) {
    siz[u] = 1, f[u][1] = a[u]; // 初始化
    for (int i = 0; i < vec[u].size (); i ++) {
        int v = vec[u][i]; // 枚举 u 的所有儿子
        DFS (v); // 同样是先递归儿子
        for (int j = min (siz[u], m + 1); j >= 1; j --)
            // m + 1 是因为虚根是必选的，比原来多选了一个
            for (int k = 0; k <= siz[v] && j + k <= m + 1; k ++)
                f[u][j + k] = max (f[u][j + k], f[u][j] + f[v][k]);
        siz[u] += siz[v];
        // 累加以 u 为根的子树大小，同时也作为 j 的枚举上界
    }
}
```

树形 dp

代码实现（中介数组）：

```
void DFS (int u) {
    siz[u] = 1, f[u][1] = a[u]; // 初始化
    for (int i = 0; i < vec[u].size (); i ++) {
        int v = vec[u][i]; // 枚举 u 的所有儿子
        DFS (v); // 同样是先递归儿子
        for (int j = 1; j <= siz[u] + siz[v] && j <= m + 1; j ++)
            g[j] = 0; // 先给中介数组清空
        for (int j = 1; j <= siz[u] && j <= m + 1; j ++)
            for (int k = 0; k <= siz[v] && j + k <= m + 1; k ++)
                g[j + k] = max (g[j + k], f[u][j] + f[v][k]);
        for (int j = 1; j <= siz[u] + siz[v] && j <= m + 1; j ++)
            f[u][j] = g[j]; // 中介数组赋值给原数组
        siz[u] += siz[v];
    }
}
```

树形 dp

时空复杂度

时间复杂度是 $\mathcal{O}(nm)$ 的，空间复杂度也是 $\mathcal{O}(nm)$ 的，其中 n 是树的大小，m 是背包容积。

树形 dp

时空复杂度

时间复杂度是 $\mathcal{O}(nm)$ 的，空间复杂度也是 $\mathcal{O}(nm)$ 的，其中 n 是树的大小，m 是背包容积。

一般情况下 $m \leq n$，所以时间复杂度的上界是 $\mathcal{O}(n^2)$ 的，这里简要证明一下时间复杂度的上界：

时空复杂度

时间复杂度是 $\mathcal{O}(nm)$ 的，空间复杂度也是 $\mathcal{O}(nm)$ 的，其中 n 是树的大小，m 是背包容积。

一般情况下 $m \leq n$，所以时间复杂度的上界是 $\mathcal{O}(n^2)$ 的，这里简要证明一下时间复杂度的上界：

容易发现，时间复杂度集中在 $f(u,j)$ 和 $f(v,k)$ 的合并过程，j 枚举的大小是已经遍历过的子树，k 枚举的大小是下一个将要合并的子树。可以看作分别枚举两个集合当中的点，这两个点形成一个点对。

树形 dp

时空复杂度

时间复杂度是 $\mathcal{O}(nm)$ 的，空间复杂度也是 $\mathcal{O}(nm)$ 的，其中 n 是树的大小，m 是背包容积。

一般情况下 $m \leq n$，所以时间复杂度的上界是 $\mathcal{O}(n^2)$ 的，这里简要证明一下时间复杂度的上界：

容易发现，时间复杂度集中在 $f(u,j)$ 和 $f(v,k)$ 的合并过程，j 枚举的大小是已经遍历过的子树，k 枚举的大小是下一个将要合并的子树。可以看作分别枚举两个集合当中的点，这两个点形成一个点对。

仔细思考一下便可以发现每个点对 (s,t) 只会在其最近公共祖先 $lca(s,t)$ 时枚举到一次，所以有多少个点对，就会一共枚举多少次，即时间复杂度的上界是 $\mathcal{O}(n^2)$。

树形 dp

时空复杂度

时间复杂度是 $\mathcal{O}(nm)$ 的，空间复杂度也是 $\mathcal{O}(nm)$ 的，其中 n 是树的大小，m 是背包容积。

一般情况下 $m \leq n$，所以时间复杂度的上界是 $\mathcal{O}(n^2)$ 的，这里简要证明一下时间复杂度的上界：
容易发现，时间复杂度集中在 $f(u,j)$ 和 $f(v,k)$ 的合并过程，j 枚举的大小是已经遍历过的子树，k 枚举的大小是下一个将要合并的子树。可以看作分别枚举两个集合当中的点，这两个点形成一个点对。
仔细思考一下便可以发现每个点对 (s,t) 只会在其最近公共祖先 $lca(s,t)$ 时枚举到一次，所以有多少个点对，就会一共枚举多少次，即时间复杂度的上界是 $\mathcal{O}(n^2)$。

树上背包中所用到的转移其实就是子树归并，时间复杂度就是子树归并的时间复杂度，所以后续算法用到子树归并时，时间复杂度也可以这么分析。

树形 dp

STA-Station（Luogu P3478）

给定一个 n 个结点的树，请求出一个结点，使得以这个结点为根时，所有结点的深度之和最大。

树形 dp

STA-Station（Luogu P3478）

给定一个 n 个结点的树，请求出一个结点，使得以这个结点为根时，所有结点的深度之和最大。

这是树形 dp 中一个很经典的问题——换根 dp，换根 dp 又被称为二次扫描，通常不会指定根结点，并且根结点的变化会对一些值，例如子结点深度和、点权和等产生影响。
通常需要两次遍历，第一次遍历时预处理诸如深度，点权和之类的信息，在第二次遍历时开始运行换根动态规划。

树形 dp

STA-Station（Luogu P3478）

给定一个 n 个结点的树，请求出一个结点，使得以这个结点为根时，所有结点的深度之和最大。

这是树形 dp 中一个很经典的问题——换根 dp，换根 dp 又被称为二次扫描，通常不会指定根结点，并且根结点的变化会对一些值，例如子结点深度和、点权和等产生影响。
通常需要两次遍历，第一次遍历时预处理诸如深度，点权和之类的信息，在第二次遍历时开始运行换根动态规划。
针对于本题，定义 $f(u)$ 表示以 u 为根时所有结点的深度之和。

树形 dp

STA-Station（Luogu P3478）

给定一个 n 个结点的树，请求出一个结点，使得以这个结点为根时，所有结点的深度之和最大。

这是树形 dp 中一个很经典的问题——换根 dp，换根 dp 又被称为二次扫描，通常不会指定根结点，并且根结点的变化会对一些值，例如子结点深度和、点权和等产生影响。
通常需要两次遍历，第一次遍历时预处理诸如深度，点权和之类的信息，在第二次遍历时开始运行换根动态规划。
针对于本题，定义 $f(u)$ 表示以 u 为根时所有结点的深度之和。
假设 $f(u)$ 已经求出，考虑将跟换到 u 的儿子节点 v 时，其余结点的变化：

树形 dp

STA-Station（Luogu P3478）

给定一个 n 个结点的树，请求出一个结点，使得以这个结点为根时，所有结点的深度之和最大。

这是树形 dp 中一个很经典的问题——换根 dp，换根 dp 又被称为二次扫描，通常不会指定根结点，并且根结点的变化会对一些值，例如子结点深度和、点权和等产生影响。
通常需要两次遍历，第一次遍历时预处理诸如深度，点权和之类的信息，在第二次遍历时开始运行换根动态规划。
针对于本题，定义 $f(u)$ 表示以 u 为根时所有结点的深度之和。
假设 $f(u)$ 已经求出，考虑将跟换到 u 的儿子节点 v 时，其余结点的变化：

- 所有在 v 的子树上的结点深度都减少了 1，那么总深度和就减少了 siz_v。
- 所有不在 v 的子树上的结点深度都增加了 1，那么总深度和就增加了 $n - siz_v$。

树形 dp

STA-Station（Luogu P3478）

给定一个 n 个结点的树，请求出一个结点，使得以这个结点为根时，所有结点的深度之和最大。

这是树形 dp 中一个很经典的问题——换根 dp，换根 dp 又被称为二次扫描，通常不会指定根结点，并且根结点的变化会对一些值，例如子结点深度和、点权和等产生影响。
通常需要两次遍历，第一次遍历时预处理诸如深度，点权和之类的信息，在第二次遍历时开始运行换根动态规划。
针对于本题，定义 $f(u)$ 表示以 u 为根时所有结点的深度之和。
假设 $f(u)$ 已经求出，考虑将跟换到 u 的儿子节点 v 时，其余结点的变化：

- 所有在 v 的子树上的结点深度都减少了 1，那么总深度和就减少了 siz_v。
- 所有不在 v 的子树上的结点深度都增加了 1，那么总深度和就增加了 $n - siz_v$。

便有 $f(v) = f(u) - siz_v + n - siz_v = f(u) + n - 2 \times siz_v$

树形 dp

STA-Station（Luogu P3478）

给定一个 n 个结点的树，请求出一个结点，使得以这个结点为根时，所有结点的深度之和最大。

树形 dp

STA-Station（Luogu P3478）

给定一个 n 个结点的树，请求出一个结点，使得以这个结点为根时，所有结点的深度之和最大。

于是，便可以通过以 1 为根进行第一次遍历，求得 $f(1)$。再以 1 为根进行第二次遍历，将其余结点的 f 值都求出。其中第二次遍历的代码实现如下：

```
void DFS (int u, int fa) {
    for (int i = head[u]; i; i = e[i].next) {
        // 因为是无根树，所以用前向星建边
        int v = e[i].to;
        if (v == fa) continue;
        f[v] = f[u] + n - siz[v] * 2; // 先转移
        DFS (v, u); // 后递归
    }
}
```

树形 dp

例题 1

战略游戏（Luogu P2016）

例题 2

重建道路（Luogu P1272）