树形dp初步

以下内容中dp与动态规划会混用
同时以下文章也为本人理解，能力有限，如果有问题本人愿意尽最大努力回答。

树形dp，顾名思义就是dp以树的形式来呈现，说得再明白点，**就是在树上进行动态规划**。而像很多传统的动态规划都是在数组上进行的，只不过数据的存储形式发生了改变。

那么，树形dp跟其它dp有什么相同的地方呢?
先来看看动态规划的简单理解(本人理解):

```
//从某一项开始
dp[n]=(操作){dp[0],dp[1],...,dp[n-1]}
```

就是说，从某项开始，**第 `n` 个位置的状态可以由前面 `n-1` 个位置的状态推出**。那么在树形dp里面，**对于节点 `node(node!=null)` 的状态就是可以由 `node.left` 的状态和 `node.right` 的状态推出**。
即:

```
dp[node]=(操作){dp[node.left],dp[node.right]}
```

那又有什么不同呢？这个从所用的数据结构就可以知道。**数组是顺式结构，所以在考虑第 `n` 个位置的时候可以跑到前面 `n-1` 个位置去推**，但是，**树是链式结构，上面说对于节点 `node` 的状态可以由 `node.left` 的状态和 `node.right` 的状态推出，也仅仅只能由 `node.left` 和 `node.right` 的状态推出**。这点就决定了， `dp[root]` 只能返回经过 `root` 或者根 `root` 有关的状态，如果考虑不经过 `root` 的话，就要定义一个全局的变量去取相应的值了，现在说这个有点抽象，下面会根据题目具体分析。

看到这里，你应该可以猜出树形dp应该用什么算法实现了吧，没错，思想就是树的深度优先遍历里面的**后序遍历**。

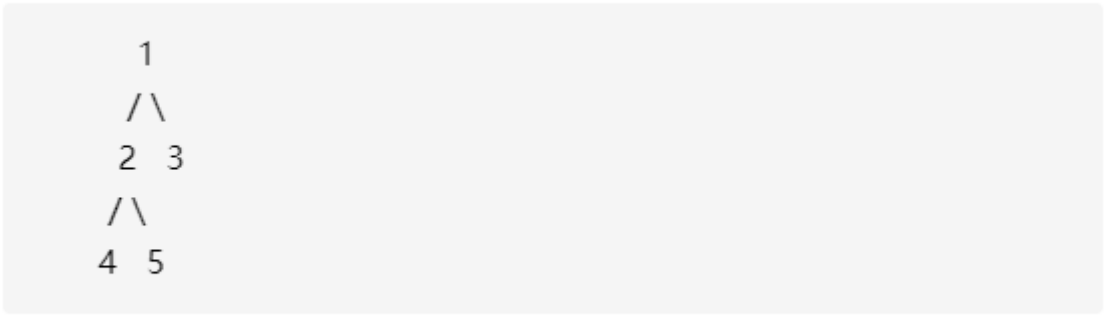
当然，学方法是为了解决问题的。解决不了问题就是纸上谈兵，那么，就先放个简单的题目感受一下吧:

543. 二叉树的直径

难度 简单 493

给定一棵二叉树，你需要计算它的直径长度。一棵二叉树的直径长度是任意两个结点路径长度中的最大值。这条路径可能穿过也可能不穿过根结点。

示例：
给定二叉树



返回 **3**, 它的长度是路径 [4,2,1,3] 或者 [5,2,1,3]。

二叉树的直径

刚开始看到这道题时，我思维就开始混乱了。只知道如果树是空结点和叶子节点肯定是0，那除了这两种情况，对于每个节点(除了根节点)都要考虑左右方向还有往父亲节点的方向，然后还要进行比较。这样下去复杂度分分钟爆炸啊。看到难度，呃，简单题😭。leetcode难度审查真的也太看得起我了。

那么了解到树形dp后呢，确实是简单题😂，只能说之前我太菜了。那么不着急，先一步步来吧😐

首先，当然是要建立动态规划的模型啦，就是要理清楚 `dp[node]` 代表什么东西？最长路径的位置吗？如果是这样的话那可以在不同的位置，得到不同的答案。可以看到，无论最长的路径有多长，**以 `node` 为开始节点的最长路径长度就是一定的**。
比如说在本题中，以根节点为开始的最长路径长度就是2(`1->2->4`或者`1->2->5`)，如果是以节点值为2的节点开始的最长路径就是 `1(2->4`或者`2->5)` 。所以， **`dp[node]` 表示的就是以 `node` 为开始节点的最长路径的长度或节点数，考虑到长度只是节点数-1，而且节点数更好理解，所以 `dp[node]` 表示的就是以 `node` 为开始节点的节点数**。

那么，第二步，当然是看看初始状态是啥。虽然说到后面可以推出来，但是初始状态还是必须是一个确定的，已经知道的数。我们知道，在树里面递归很舒服，但也是有条件的，比如说出口就是空节点(甚至有些时候是叶子节点)。那么在这道题里面，**初始状态就是 `dp[null]=0`** 。如果连起点都没有，那么何谈有路？

第三步，建立状态转移方程。听着高大上是吧？实际上就是看如何
从 dp[node.left] 和 dp[node.right] 推出 dp[node] 。在回到题目中的例子，就是说加入要求 dp[1] ，那么如何从 dp[2] 和 dp[3] 推出来。如果是拿节点值为1的节点作为起点，那么下一步就是要选择节点2还是节点3。这句话听着...不是一句废话吗？那把这句话换种说法呢，如果是拿节点值为1的节点作为起点，那么下一步就是要选择以节点2作为起点的路还是以节点3作为起点的路？
如果要以节点值为1的起点的路最长，那下一步肯定选择节点2作为起点的路还是以节点3作为起点的路里面最长的那条啊。
然后，状态转移方程就出来了 dp[node]=max{dp[node.left],dp[node.right]}+1

好了，那么就可以把模板写出来了，以后续遍历的方法:

```
/**************************************************
 *@param:node 节点为node
 *@return: 这里返回的就是以node为开始节点的最大路径的节点数，就是上面说到dp[node]
 */
int dfs(TreeNode node){
    //dp[null]=0;
    if(node==null)
        return 0;
    //dp[node.left]
    int left=dfs(node.left);
    //dp[node.right]
    int right=dfs(node.right);
    //dp[node]=max{dp[node.left],dp[node.right]}+1
    return Math.max(left,right)+1;
}
```

写到这里，已经完成90%，那剩下的10%是什么呢？题目要求的是**整棵树里面的最长路径**,也就是说，如果 node 作为起点的话，不仅仅可以往左右子树跑，而且可以往父亲节点跑！!但是，题目只要我们求出长度就行，所以 node 往父亲节点跑的长度和父亲节点往 node 跑的长度是一样的。也就是说最大路径结点数就是 max{dp[node.left]+dp[node.right]+1|(node为树的全部节点)}
那么在遍历的过程中引入全局变量 maxNum 记录最大路径的节点数。
那么代码就是:

```
maxNum=Math.max(left+right+1,maxNum);
```

合起来:

```
//定义全局变量来找最大路径节点数
int maxNum=0;
/**************************************************
 *@param:node 节点为node
 *@return: 这里返回的就是以node为开始节点的最大路径的节点数，就是上面说到dp[node]
 */
int dfs(TreeNode node){
    //dp[null]=0;
    if(node==null)
        return 0;
    //dp[node.left]
    int left=dfs(node.left);
    //dp[node.right]
    int right=dfs(node.right);
    //max{dp[node.left]+dp[node.right]+1|(node为树的全部节点)}
    maxNum=Math.max(left+right+1,maxNum);
    //dp[node]=max{dp[node.left],dp[node.right]}+1
    return Math.max(left,right)+1;
}
```

完整代码：

```java
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
class Solution {
    //定义全局变量来找最大路径节点数
    int maxNum=0;
    /**************************************************
    *@param:node 节点为node
    *@return: 这里返回的就是以node为开始节点的最大路径的节点数，就是上面说到dp[node]
    */
    int dfs(TreeNode node){
        //dp[null]=0;
        if(node==null)
            return 0;
        //dp[node.left]
        int left=dfs(node.left);
        //dp[node.right]
        int right=dfs(node.right);
        //max{dp[node.left]+dp[node.right]+1|(node为树的全部节点)}
        maxNum=Math.max(left+right+1,maxNum);
        //dp[node]=max{dp[node.left],dp[node.right]}+1
        return Math.max(left,right)+1;
    }
    public int diameterOfBinaryTree(TreeNode root) {
        //空结点和叶子节点当然没有路径可言
        if(root==null||(root.left==null&&root.right==null))
            return 0;
        dfs(root);
        //注意：maxNUum是最大路径的节点数，如果是长度还要-1
        return maxNum-1;
    }
}
```

虽然是简单题，但是思想很重要，后面的题大多数也是这个模板。

再来看看另外一道题：

124. 二叉树中的最大路径和

难度 困难 713

给定一个**非空**二叉树，返回其最大路径和。

本题中，路径被定义为一条从树中任意节点出发，沿父节点-子节点连接，达到任意节点的序列。该路径**至少包含一个**节点，且不一定经过根节点。

示例 1：

输入： [1,2,3]

```
  1
 / \
2   3
```

输出： 6

示例 2：

输入： [-10,9,20,null,null,15,7]

```
  -10
  / \
 9  20
   /  \
  15   7
```

二叉树中的最大路径和

从简单到困难，往往只需要增加一点条件。

回到这道题:

`dp[node]` 是什么？**当然是以 `node` 为开始节点的最大路径和。**

初始条件是什么？**`dp[null]` =0。**

上面两个问题估计如果上一简单道题充分理解的话就可以脱口而出了。但是状态转移方程就有变化了

看看上面的题状态转移方程？

```
dp[node]=max{dp[node.left],dp[node.right]}+1
```

那么可能很多人就盲目照搬:

```
dp[node]=max{dp[node.left],dp[node.right]}+node.val;
```

可以看到，不管怎么说 `dp[node]` 都是大于等于0的。但是放在这道题里面，**`dp[node.left]` 和 `dp[node.right]` 可能都<0**。如果两个都小于0的话，还要选最大的那个？就比如说在你面前两份生意，你无论做那一份都会亏，只不过亏多亏少罢了，你怎么选择？肯定没有人选亏大的那个，但是也建议不要选亏少的那个，有句古话嘛，亏本的买卖咱不做。所以，放到这里面，如果**`dp[node.left]` 和 `dp[node.right]` 都<0，那么当然都不选!**
那么加的代码就是

```
int left=dfs(node.left);
int right=dfs(node.right);
//如果小于0还不如不选
left=Math.max(left,0);
right=Math.max(right,0);
```

同时，跟第一道简单题一样，也要定义全局变量。
所以，完整代码就是:

```
class Solution {
    int res=Integer.MIN_VALUE;
    public int dfs(TreeNode node){
      if(node==null)
        return 0;
      int left=Math.max(dfs(node.left),0);
      int right=Math.max(dfs(node.right),0);
      res=Math.max(left+right+node.val,res);
      return Math.max(left,right)+node.val;
    }
    public int maxPathSum(TreeNode root) {
      dfs(root);
      return res;
    }
}
```

复杂的题也就这样，但是方法真的经典。

再看看一道假的简单题:

687. 最长同值路径

难度 简单 361

给定一个二叉树，找到最长的路径，这个路径中的每个节点具有相同值。
这条路径可以经过也可以不经过根节点。

注意：两个节点之间的路径长度由它们之间的边数表示。

示例 1:

输入:

```
      5
     /\
    4  5
   /\  \
  1  1  5
```

输出:

```
2
```

最长同值路径

估计很多人被中等，难题摧残到自闭后想做简单题，但是不小心切到这道。然后又开始怀疑人生...其实这个也是简单题，但是是在学了方法以后(这确实是废话)。那么话不多说，还是问那三个问题：

`dp[node]` 是什么？以 `node` 为开始节点的最长同值路劲的长度！！！

初始条件是什么？ `dp[null]=0`

好！ 很有精神！！！

那么状态转移方程呢?

这就是唯一的难点了，也是跟第一道题有不同的地方。对于第一道题来说是这样的：

```
dp[node]=max{dp[node.left],dp[node.right]}+1
```

但是这道题，想想，如果 **所选路径的开头的节点的值和第二个节点的值都不相同，那么哪来的同值路径，肯定不能选啊。**

也就是说 `node` 能选择以 `node.left(node.right)` 作为开头的路径，必须保

证 node.val 与 node.left.val(node.right.val) 相同
所以加的代码就如下：

```
int left=dfs(node.left);
int right=dfs(node.right);
int val=node.val;
//不相同肯定不选
left=node.left!=null&&node.left.val==val?left:0;
right=node.right!=null&&node.right.val==val?right:0;
```

那么完整代码如下：

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
class Solution {
    private int res=0;
    public int dfs(TreeNode node){
      if(node==null)
        return 0;
      int left=dfs(node.left);
      int right=dfs(node.right);
      //增加的判断条件
      int val=node.val;
      left=node.left!=null&&node.left.val==val?left:0;
      right=node.right!=null&&node.right.val==val?right:0;
      //
      res=Math.max(left+right+1,res);
      return Math.max(left,right)+1;
    }
    public int longestUnivaluePath(TreeNode root) {
      if(root==null)
        return 0;
      dfs(root);
      return res-1;
    }
}
```

还有一道题跟这个很类似的：
250.统计同值子树
不过用到的方法就是记忆化了，不是动态规划。

再看看也是从节点值就开始选择的dp题目吧：

549. 二叉树中最长的连续序列

难度 中等 46

给定一个二叉树，你需要找出二叉树中最长的连续序列路径的长度。

请注意，该路径可以是递增的或者是递减。例如，[1,2,3,4] 和 [4,3,2,1] 都被认为是合法的，而路径 [1,2,4,3] 则不合法。另一方面，路径可以是 子-父-子 顺序，并不一定是 父-子 顺序。

示例 1:

输入:

```
    1
   / \
  2   3
```

输出: 2

解释: 最长的连续路径是 [1, 2] 或者 [2, 1]。

示例 2:

输入:

```
    2
   / \
  1   3
```

输出: 3

解释: 最长的连续路径是 [1, 2, 3] 或者 [3, 2, 1]。

二叉树中的最长连续序列

跟上面一道题类似(呃，我也不知道说了多少遍了)。但是题目说了哦，**路径可以是递减，也可以是递增**。所以对于开始的节点来说，**可以是路径的最大值，也可以是路径的最小值**。那咋办？还能怎么办，就将 `dp[node]` 的维数扩大咯，成为 `dp[node][2]` ，其中 `dp[node][0]` 代表**以 node 为开始节点，最长递减路径的节点数**， `dp[node][1]` 代表**以 node 为开始节点，最长递增路径的节点数**。

好的，接下来看看初始条件：如果是空节点，那么无论递增还是递减，都是0。

即： `dp[null]=new int[]{0,0}`

最后，就是状态转移方程了，回到题目里面，如果以1这个节点，要选递增路线，就只能选择以

2(2=1+1)作为开头节点的递增或者空节点点的路线。那么同样，以3为节点值的节点，如果要选递减路线，也只能选以2(2=3-1)作为开头的递减或者空节点路线。
就是如下操作：

```
dp[node.left][0]=(node.left!=null&&node.left.val==val-1)?dp[node.left][0]:0;
dp[node.left][1]=(node.left!=null&&node.left.val==val+1)?dp[node.left][1]:0;
dp[node.right][0]=(node.right!=null&&node.right.val==val-1)?dp[node.right[0]]:0;
dp[node.right][1]=(node.right!=null&&node.right.val==val+1)?dp[node.right][1]:0;
```

而对于全局变量来说，由于递增递减都OK，所以要比较两次：

```
res=Math.max(dp[node.left][0]+1+dp[node.right][1],res);
res=Math.max(dp[node.left][1]+1+dp[node.right][0],res);
```

好了，看看完整代码吧：

```java
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
class Solution {
    int res=0;
    public int[] dfs(TreeNode node){
        //初始条件
        if(node==null)
            return new int[]{0,0};
        int val=node.val;
        int[] left=dfs(node.left);
        int[] right=dfs(node.right);
        //dp[node.left][0]=(node.left!=null&&node.left.val==val-1)?dp[node.left][0]:0;
        int left_0=(node.left!=null&&node.left.val==val-1)?left[0]:0;
        //dp[node.left][1]=(node.left!=null&&node.left.val==val+1)?dp[node.left][1]:0;
        int left_1=(node.left!=null&&node.left.val==val+1)?left[1]:0;
        //dp[node.right][0]=(node.right!=null&&node.right.val==val-1)?dp[node.right[0]]:0;
        int right_0=(node.right!=null&&node.right.val==val-1)?right[0]:0;
        //dp[node.right][1]=(node.right!=null&&node.right.val==val+1)?dp[node.right][1]:0;
        int right_1=(node.right!=null&&node.right.val==val+1)?right[1]:0;
        //比较两次
        res=Math.max(left_0+1+right_1,res);
        res=Math.max(left_1+1+right_0,res);
        //
        int d=Math.max(left_0,right_0)+1;
        int u=Math.max(left_1,right_1)+1;
        return new int[]{d,u};
    }
    public int longestConsecutive(TreeNode root) {
        if(root==null)
            return 0;
        dfs(root);
        return res;
    }
}
```

判断条件也烦了，来点简单的放松一下吧：

1372. 二叉树中的最长交错路径

难度 中等 30 收藏 分享 切换为英文 接收动态 反馈

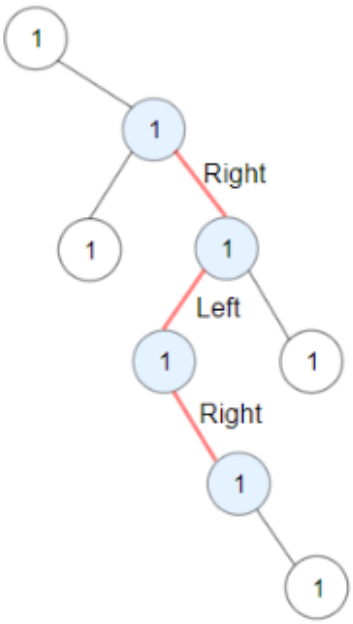
给你一棵以 root 为根的二叉树，二叉树中的交错路径定义如下：

- 选择二叉树中 **任意** 节点和一个方向（左或者右）。
- 如果前进方向为右，那么移动到当前节点的的右子节点，否则移动到它的左子节点。
- 改变前进方向：左变右或者右变左。
- 重复第二步和第三步，直到你在树中无法继续移动。

交错路径的长度定义为：**访问过的节点数目 - 1**（单个节点的路径长度为 0 ）。

请你返回给定树中最长 **交错路径** 的长度。

示例 1：



输入： root = [1,null,1,1,1,null,null,1,1,null,1,null,null,null,1,null,1]
输出： 3
解释： 蓝色节点为树中最长交错路径（右 -> 左 -> 右）。

二叉树中的最长交错路径

跟上面的题有啥不同呢？不就是选了左(右)子树然后到了新节点后就不能选右(左)边嘛！
还是老规矩，用 `dp[node][2]` 。其中 **`dp[node][0]` 就是以node开始，然后选左子树的最长交错路径节点数**， **`dp[node][1]` 就是以node开始，然后选右子树的最长交错路径节点数**；`dp[null]=new int[2]` ；
然后就是状态转移方程了

```
dp[node][1]=dp[node.left][1]+1;
dp[node][0]=dp[node.right][0]+1;
```

而这里的全局变量就是：

```
res=max{dp[node][0],dp[node][1],res}
```

对，要同时比较左右两边。

好了，那代码就可以轻松写出来了：

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
class Solution {
    private int maxx=0;
    public int []dfs(TreeNode node){
      if(node==null)
        return new int[2];
      int[] left=dfs(node.left);
      int[] right=dfs(node.right);
      int[] res=new int[]{left[1]+1,right[0]+1};
      maxx=Math.max(maxx,Math.max(res[0],res[1]));
      return res;
    }
    public int longestZigZag(TreeNode root) {
      if(root==null)
        return 0;
      dfs(root);
      return maxx-1;
    }
}
```

好，估计看到这也累了，放心，下面的就是最后一道题了。

337. 打家劫舍 III

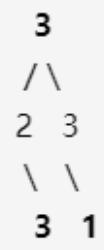
难度 中等 592 收藏 分享 切换为英文 接收动态 反馈

在上次打劫完一条街道之后和一圈房屋后，小偷又发现了一个新的可行窃的地区。这个地区只有一个入口，我们称之为“根”。 除了“根”之外，每栋房子有且只有一个“父”房子与之相连。一番侦察之后，聪明的小偷意识到“这个地方的所有房屋的排列类似于一棵二叉树”。 如果两个直接相连的房子在同一天晚上被打劫，房屋将自动报警。

计算在不触动警报的情况下，小偷一晚能够盗取的最高金额。

示例 1:

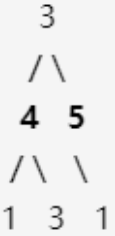
输入: [3,2,3,null,3,null,1]



输出: 7
解释: 小偷一晚能够盗取的最高金额 = 3 + 3 + 1 = **7**.

示例 2:

输入: [3,4,5,1,3,null,1]



输出: 9
解释: 小偷一晚能够盗取的最高金额 = **4** + **5** = **9**.

打家劫舍III

又是这个水平不咋地的小偷。
按照题目的意思，就是说偷了 node 以后就不能再偷 node.left 或 node.right 了。
那么就还是像上面一样，定义 dp[node][2] ，其中 **dp[node][0] 表示以node作为根节点的树，偷了node能得到的最多的钱。** **dp[node][1] 表示以node作为根节点的树，没有偷node能得到的最多的钱。**
由于小偷要偷的就是整棵树，所以返回dp[root][0]和dp[root][1]的最大值就行。

同样，如果到的地方没房子，那就啥也没有： dp[null]=new int[2]

那么，再来看看:假如说偷了node，那么node.left和node.right就动不得了。
即：

```
dp[node][0]=dp[node.left][1]+dp[node.right][1]+node.val;
```

那如果没有偷node呢？是不是一定要偷node.left和node.right？那可真不一定，万一node.left.left比node.left的值还多，偷了node.left那不是血亏？所以还是要比一下哪个更大。
即：

```
dp[node][1]=Math.max(dp[node.left][0],dp[node.left][1])+Math.max(dp[node.right][0],dp[node.right
```

好了，那么代码也要出来了：

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
class Solution {
    public int[] getDp(TreeNode node){
      if(node==null)
        return new int[]{0,0};
      int[] left=getDp(node.left);
      int[] right=getDp(node.right);
      int robNode=left[1]+right[1]+node.val;
      int notRobNode=Math.max(left[0],left[1])+Math.max(right[0],right[1]);
      return new int[]{robNode,notRobNode};
    }
    public int rob(TreeNode root) {
      int[] dp=getDp(root);
      return Math.max(dp[0],dp[1]);
    }
}
```

曾经，这些题都是让笔者绞尽脑汁的存在，甚至有些题，笔者提交了差不多10遍。但是经典的方法总是要勤加练习，所以今天笔者才可以将这篇文章写出来，以帮助需要灵感与方法的刷题者。对于经典的题目，目的性地重复训练总是有好处的，每次重复都会有更深的理解。

$Peace$!