<https://doodlejack.gitbooks.io/coding-notes/content/problem-sets/tree/dfs-vs-traverse.html>

source: <https://www.jiuzhang.com/qa/1497/>

Q: 递归（搜索）时，什么时候需要有 return type?

A: 递归分两种，分治(divide and conquer)和遍历(traversal)

我们先不管return type的问题。递归的程序有三个部分：  
1.base case  
2.递归成为更小的问题  
3.进行当前层的处理计算

其中分治算法的顺序是123，遍历算法是132.（强烈建议自己拿个小一点的树或者其他结构模拟一下）。也就是说，分治是先“分”再”治“，而遍历是先“治”再“分”

这样导致的结果就是分治的方向是从下到上完成(和从下到上的DP有点相似)，遍历的方向是从上到下完成。

现在回答我们最初的问题，什么情况下是void，什么情况下是别的return type

由于遍历的算法是从上到下，这个时候要return value完全没有意义。因为算到了一个中间结果的时候对最终结果不会产生本质的影响。我举个栗子，比如说你在刷算法题，这个时候有朋友问你，刷的怎么样了。你回答他，刷了3个月，这个答案不算很好，为什么呢？因为你刷了3个月，很可能还有3个礼拜，3个月或者3年，除非有一天你刷完了（走到了二叉树的底层），否则别人无法从这个回答中得到最终状态的关系，那么这个结果就没有什么价值。但是如果你回答，根据我的计划，还有2个礼拜就能刷完，这个答案就很有意义了，因为这个结果对最终的结果有一个很好的联系。

以上都是从哲学的角度来帮助我们理解这个问题。换到刷题的时候，只需要记住，遍历的都是void型，分治都是有具体的return type的（但是有一种情况例外，就是题目本身就是要写一个mutator（void型），参见翻转二叉树）

一般对数组和String做DFS的时候，都是用遍历的方式，思路简单易懂，参见subset和permutation，以及动态规划的各种暴力算法，这种情况都是void型的（就是我们俗称的“带着结果走一圈”）。但是二叉树的递归通常喜欢用分治，因为大多数情况都是从底层向上做处理的，这些情况都是有具体的return type（就是我们俗称的“老大给小弟布置任务”）

遍历算法在解二叉树的时候会遇到很多困难（甚至有可能做不出来，比如Balanced Binary Tree这类需要自定义class的题型）。特别推荐valid binary search tree这道题，可以试试分别用分治和遍历做一遍，你就能理解他们的差异啦~

九章算法班的顺序是先讲subset的递归版本（用的是遍历）。对于这些结构不明显的数组来说，理解起来可能稍有难度。等你上完二叉树了之后说不定就会对两种算法有更深刻的体会啦~

A: 简单来说就是  
1、如果你的递归是在原数组上修改，或者有一个传递引用的变量results来接收结果，那么函数是不需要返回值的，那我们定义成void。  
2、如果函数需要返回值，并且返回值是基础类型int，string等等，那么就直接把返回类型定义成这些就可以了。  
3、如果你的函数需要返回多个值，比如有些题你既需要返回MAX又需要返回MIN，当有多个值返回的时候，你需要用一个Result Type进行封装，定义成一个Class。

所以函数返回什么是根据算法的需求来看的。当然你需要返回的参数也可以用引用的方式传回来。

e.g

**二叉树的前序遍历 · Binary Tree Preorder Traversal**

<https://www.jiuzhang.com/solutions/binary-tree-preorder-traversal/#tag-highlight>

//Version 1: Traverse

1.base case  
3.进行当前层的处理计算  
2.递归成为更小的问题

void return

有一个传递引用的变量results来接收结果，那么函数是不需要返回值的，那我们定义成void

**public** class Solution {

**public** ArrayList<Integer> preorderTraversal(TreeNode root) {

ArrayList<Integer> result = **new** ArrayList<Integer>();

traverse(root, result);

**return** result;

}

// 把root为跟的preorder加入result里面

**private** **void** traverse(TreeNode root, ArrayList<Integer> result) {

**if** (root == null) {

**return**;

}

result.add(root.val);

traverse(root.left, result);

traverse(root.right, result);

}

}

//Version 2: Divide & Conquer

1.base case  
2.递归成为更小的问题  
3.进行当前层的处理计算

return type

**public** class Solution {

**public** ArrayList<Integer> preorderTraversal(TreeNode root) {

ArrayList<Integer> result = **new** ArrayList<Integer>();

// null or leaf

**if** (root == null) {

**return** result;

}

// Divide

ArrayList<Integer> left = preorderTraversal(root.left);

ArrayList<Integer> right = preorderTraversal(root.right);

// Conquer

result.add(root.val);

result.addAll(left);

result.addAll(right);

**return** result;

}

}