<https://www.stdstring.com/2020/02/02/devide-binary-tree/>

## 二叉树中的分治法与遍历法

 发表于 2020-02-02   分类于 [算法](https://www.stdstring.com/categories/%E7%AE%97%E6%B3%95/)  阅读次数： 138

## 分治法 vs 遍历法

拿人口统计作为例子，遍历法与分治法的区别如下：

* 如果使用遍历法，做法如下：

人口普查员自己一个人带着一个本子，跑遍全中国挨家挨户的敲门查户口

* 而如果使用分治法，做法如下：

国家统计局的老板想要知道全国人口的总数，于是他找来全国各个省的统计局领导，下派人口普查任务给他们，让他们各自去统计自己省的人口总数。在老板  
这儿，他只需要最后将各个省汇报的人口总数结果累加起来，就得到了全国人口的数目。

然后每个省的领导，又找来省里各个市的领导，让各个市去做人口统计。

市找县，县找镇，镇找乡。最后乡里的干部挨家挨户敲门去查户口。

在这里，把全国的任务拆分为省级的任务的过程，就是分治法中分的这个步骤。把各个小任务派发给别人去完成的过程，就是分治法中  
治的这个步骤。但是事实上我们还有第三个步骤，就是将小任务的结果合并到一起的过程，合这个步骤。因此如果我来取名字的话，我会叫这个算法：分治合算法。

## 递归 vs 分治法 vs 遍历法

#### 联系

分治法与遍历法是两种常见的递归方法。

#### 分治法解决问题的思路

先让左右子树去解决同样的问题，然后得到结果之后，再整合为整棵树的结果。

#### 遍历法解决问题的思路

通过前序 / 中序 / 后序的某种遍历，游走整棵树，通过一个全局变量或者传递的参数来记录这个过程中所遇到的点和需要计算的结果。

#### 两种方法的区别

从程序实现角度分治法的递归函数，通常有一个返回值，遍历法通常没有

## 递归 vs 回溯 vs 搜索

#### 递归

与递归对应的是迭代。

递归可以从两个层面来理解：

1. 算法层面，是一种由大化小、由小化无的思想
2. 程序实现层面，是函数自己调用自己的一种方式

#### 搜索

搜索分为深度优先搜索 DFS 和宽度优先搜索 BFS，搜索是一种类似于枚举的算法，（当我们知道数组的范围和大小时）。

通常，BFS 不用递归，DFS 需要递归。

#### 回溯

回溯是深度优先搜索过程中的一个步骤。

<https://www.dazhuanlan.com/barneyzhu/topics/1175902>

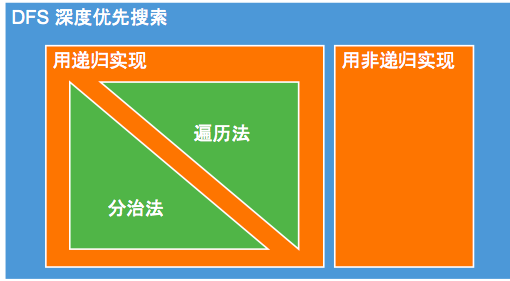
## Classic problems with Divide and Conquer

六、可使用分治法求解的一些经典问题

1. binary search
2. large integer mutiplication
3. streassen matrix multiplication
4. chess board coverage
5. merge sort
6. quick sort
7. linear time choice
8. Tower of Hanoi

这些经典问题的特性是: 将 complex problem break down 后都能形成一个 [树状] 的分解图。因而在此类问题使用 dfs 的 divide conquer 是合理的。

## Divide and Conquer vs Traverse

[[](https://s3.amazonaws.com/images.xuyiruan.com/2018-02-24-DandCvsTraverse.png)](https://s3.amazonaws.com/images.xuyiruan.com/2018-02-24-DandCvsTraverse.png)

**Traverse vs Divide Conquer**

* They are both Recursion Algorithm
* Result in parameter vs Result in return value
* Top down vs Bottom up
* Merge Sort / Quick Sort
* 90% Binary Tree Problems!

The difference of traverse and divide conquer in code can refer to:  
<http://www.jiuzhang.com/solutions/binary-tree-preorder-traversal/#tag-highlight>

Other thoughs help to understand their difference:

* “遍历” 是一个小人带着一本记事本游走整个二叉树。
* “分治” 是一个大王呆着两个小弟，让两个小弟分别去搞定左右子树的结果汇报给大王，大王做最后的整理和拍板。

From implementation prospective:

* “遍历” 使用 global var (大王的笔记本)
* “分治” 不使用 global var (结果在小弟们那，最后才收集起来)

<https://www.jiuzhang.com/qa/723/>

<https://www.jiuzhang.com/qa/1244/>

## Difference between iteration and recursion?

why manual stack is better than recursion (call stack):

非递归其实是模拟递归用的 stack

为什么自己模拟的可以, 调用计算机的就不行呢?

因为我们 new 出来的 stack 在 heap memory 里面, 能用的空间≈ memory size

stack memory ≈ process memory 是计算机分给每个程序的一个很小的独占的空间,所以递归的深度太深,就不够用了

[https://stomachache007.wordpress.com/2017/03/12/九章算法笔记-3-binary-tree-divide-conquer/](https://stomachache007.wordpress.com/2017/03/12/%E4%B9%9D%E7%AB%A0%E7%AE%97%E6%B3%95%E7%AC%94%E8%AE%B0-3-binary-tree-divide-conquer/)

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

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

题解： 非递归方式实现前序遍历时，首先存入当前节点值，然后先将右儿子压入栈中，再将左儿子压入栈中。对栈中元素遍历访问。

Version 0: Non-Recursion (Recommend)

/\*\*

\* Definition for binary tree

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

**public** class Solution {

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

Stack<TreeNode> stack = **new** Stack<TreeNode>();

List<Integer> preorder = **new** ArrayList<Integer>();

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

**return** preorder;

}

stack.push(root);

**while** (!stack.empty()) {

TreeNode node = stack.pop();

preorder.add(node.val);

**if** (node.right != null) {

stack.push(node.right);

}

**if** (node.left != null) {

stack.push(node.left);

}

}

**return** preorder;

}

}

//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;

}

}