递归详解

相关推荐：

- 特殊数据结构：单调队列
- 设计Twitter

首先说明一个问题，简单阐述一下递归，分治算法，动态规划，贪心算法这几个东西的区别和联系，心里有个印象就好。

递归是一种编程技巧，一种解决问题的思维方式；分治算法和动态规划很大程度上是递归思想基础上的（虽然动态规划的最终版本大都不是递归了，但解题思想还是离不开递归），解决更具体问题的两类算法思想；贪心算法是动态规划算法的一个子集，可以更高效解决一部分更特殊的问题。

分治算法将在这节讲解，以最经典的归并排序为例，它把待排序数组不断二分为规模更小的子问题处理，这就是 “分而治之” 这个词的由来。显然，排序问题分解出的子问题是不重复的，如果有的问题分解后的子问题有重复的（重叠子问题性质），那么就交给动态规划算法去解决！

递归详解

介绍分治之前，首先要弄清楚递归这个概念。

递归的基本思想是某个函数直接或者间接地调用自身，这样就把原问题的求解转换为许多性质相同但是规模更小的子问题。我们只需要关注如何把原问题划分成符合条件的子问题，而不需要去研究这个子问题是如何被解决的。递归和枚举的区别在于：枚举是横向地把问题划分，然后依次求解子问题，而递归是把问题逐级分解，是纵向的拆分。

以下会举例说明我对递归的一点理解，**如果你不想看下去了，请记住这几个问题怎么回答：**

1. 如何给一堆数字排序？ 答：分成两半，先排左半边再排右半边，最后合并就行了，至于怎么排左边和右边，请重新阅读这句话。
2. 孙悟空身上有多少根毛？ 答：一根毛加剩下的毛。
3. 你今年几岁？ 答：去年的岁数加一岁,1999 年我出生。

递归代码最重要的两个特征：结束条件和自我调用。自我调用是在解决子问题，而结束条件定义了最简子问题的答案。

```
int func(你今年几岁) {
    // 最简子问题，结束条件
    if (你1999年几岁) return 我0岁;
    // 自我调用，缩小规模
    return func(你去年几岁) + 1;
}
```

其实仔细想想，**递归运用最成功的是什么？我认为是数学归纳法。**我们高中都学过数学归纳法，使用场景大概是：我们推不出来某个求和公式，但是我们试了几个比较小的数，似乎发现了一点规律，然后编了一个公式，看起来应该是正确答案。但是数学是很严谨的，你哪怕穷举了一万个数都是正确的，但是第一万零一个数正确吗？这就要数学归纳法发挥神威了，可以假设我们编的这个公式在第 k 个数时成立，如果证明在第 k + 1 时也成立，那么我们编的这个公式就是正确的。

那么数学归纳法和递归有什么联系？我们刚才说了，递归代码必须要有结束条件，如果没有的话就会进入无穷无尽的自我调用，直到内存耗尽。而数学证明的难度在于，你可以尝试有穷种情况，但是难以将你的结论延伸到无穷大。这里就可以看出联系了 —— 无穷。

递归代码的精髓在于调用自己去解决规模更小的子问题，直到到达结束条件；而数学归纳法之所以有用，就在于不断把我们的猜测向上加一，扩大结论的规模，没有结束条件，从而把结论延伸到无穷无尽，也就完成了猜测正确性的证明。

为什么要写递归

首先为了训练逆向思考的能力。递推的思维是正常人的思维，总是看着眼前的问题思考对策，解决问题是将来时；递归的思维，逼迫我们倒着思考，看到问题的尽头，把解决问题的过程看做过去时。

第二，练习分析问题的结构，当问题可以被分解成相同结构的小问题时，你能敏锐发现这个特点，进而高效解决问题。

第三，跳出细节，从整体上看问题。再说说归并排序，其实可以不用递归来划分左右区域的，但是代价就是代码极其难以理解，大概看一下代码（归并排序在后面讲，这里大致看懂意思就行，体会递归的妙处）：

```
void sort(Comparable[] a){
    int N = a.length;
    // 这么复杂，是对排序的不尊重。我拒绝研究这样的代码。
    for (int sz = 1; sz < N; sz = sz + sz)
        for (int lo = 0; lo < N - sz; lo += sz + sz)
            merge(a, lo, lo + sz - 1, Math.min(lo + sz + sz - 1, N - 1));
}

/* 我还是选择递归，简单，漂亮 */
void sort(Comparable[] a, int lo, int hi) {
    if (lo >= hi) return;
    int mid = lo + (hi - lo) / 2;
    sort(a, lo, mid); // 排序左半边
    sort(a, mid + 1, hi); // 排序右半边
    merge(a, lo, mid, hi); // 合并两边
}
```

看起来简洁漂亮是一方面，关键是**可解释性很强**：把左半边排序，把右半边排序，最后合并两边。而非递归版本看起来不知所云，充斥着各种难以理解的边界计算细节，特别容易出 bug 且难以调试，人生苦短，我更倾向于递归版本。

显然有时候递归处理是高效的，比如归并排序，**有时候是低效的**，比如数孙悟空身上的毛，因为堆栈会消耗额外空间，而简单的递推不会消耗空间。比如这个例子，给一个链表头，计算它的长度：

```
/* 典型的递推遍历框架，需要额外空间 O(1) */
public int size(Node head) {
    int size = 0;
    for (Node p = head; p != null; p = p.next) size++;
    return size;
}
/* 我偏要递归，万物皆递归，需要额外空间 O(N) */
public int size(Node head) {
    if (head == null) return 0;
    return size(head.next) + 1;
}
```

写递归的技巧

我的一点心得是：**明白一个函数的作用并相信它能完成这个任务，千万不要试图跳进细节。**千万不要跳进这个函数里面企图探究更多细节，否则就会陷入无穷的细节无法自拔，人脑能压几个栈啊。

先举个最简单的例子：遍历二叉树。

```
void traverse(TreeNode* root) {
    if (root == nullptr) return;
    traverse(root->left);
    traverse(root->right);
}
```

这几行代码就足以扫荡任何一棵二叉树了。我想说的是，对于递归函数`traverse(root)`，我们只要相信：给它一个根节点`root`，它就能遍历这棵树，因为写这个函数不就是为了这个目的吗？所以我们只需要把这个节点的左右节点再甩给这个函数就行了，因为我相信它能完成任务的。那么遍历一棵N叉数呢？太简单了好吧，和二叉树一模一样啊。

```
void traverse(TreeNode* root) {
    if (root == nullptr) return;
    for (child : root->children)
        traverse(child);
}
```

至于遍历的什么前、中、后序，那都是显而易见的，对于N叉树，显然没有中序遍历。

以下**详解 LeetCode 的一道题来说明**：给一课二叉树，和一个目标值，节点上的值有正有负，返回树中和等于目标值的路径条数，让你编写 pathSum 函数：

```
/* 来源于 LeetCode PathSum III： https://leetcode.com/problems/path-sum-iii/ */
root = [10,5,-3,3,2,null,11,3,-2,null,1],
sum = 8

      10
     /  \
    5   -3
   / \    \
  3   2   11
 / \   \
3  -2   1

Return 3. The paths that sum to 8 are:

1.  5 -> 3
2.  5 -> 2 -> 1
3. -3 -> 11
```

```
/* 看不懂没关系，底下有更详细的分析版本，这里突出体现递归的简洁优美 */
int pathSum(TreeNode root, int sum) {
    if (root == null) return 0;
    return count(root, sum) +
        pathSum(root.left, sum) + pathSum(root.right, sum);
}
int count(TreeNode node, int sum) {
    if (node == null) return 0;
    return (node.val == sum) +
        count(node.left, sum - node.val) + count(node.right, sum - node.val);
}
```

题目看起来很复杂吧，不过代码却极其简洁，这就是递归的魅力。我来简单总结这个问题的**解决过程**：

首先明确，递归求解树的问题必然是要遍历整棵树的，所以**二叉树的遍历框架**（分别对左右孩子递归调用函数本身）必然要出现在主函数 pathSum 中。那么对于每个节点，他们应该干什么呢？他们应该看看，自己和脚底下的小弟们包含多少条符合条件的路径。好了，这道题就结束了。

按照前面说的技巧，根据刚才的分析来定义清楚每个递归函数应该做的事：

PathSum 函数：给他一个节点和一个目标值，他返回以这个节点为根的树中，和为目标值的路径总数。

count 函数：给他一个节点和一个目标值，他返回以这个节点为根的树中，能凑出几个以该节点为路径开头，和为目标值的路径总数。

```
/* 有了以上铺垫，详细注释一下代码 */
int pathSum(TreeNode root, int sum) {
    if (root == null) return 0;
```

```
    int pathImLeading = count(root, sum); // 自己为开头的路径数
    int leftPathSum = pathSum(root.left, sum); // 左边路径总数（相信他能算出来）
    int rightPathSum = pathSum(root.right, sum); // 右边路径总数（相信他能算出来）
    return leftPathSum + rightPathSum + pathImLeading;
}
int count(TreeNode node, int sum) {
    if (node == null) return 0;
    // 我自己能不能独当一面，作为一条单独的路径呢?
    int isMe = (node.val == sum) ? 1 : 0;
    // 左边的小老弟，你那边能凑几个 sum - node.val 呀?
    int leftBrother = count(node.left, sum - node.val);
    // 右边的小老弟，你那边能凑几个 sum - node.val 呀?
    int rightBrother = count(node.right, sum - node.val);
    return  isMe + leftBrother + rightBrother; // 我这能凑这么多个
}
```

还是那句话，明白每个函数能做的事，并相信他们能够完成。

总结下，PathSum 函数提供的二叉树遍历框架，在遍历中对每个节点调用 count 函数，看出先序遍历了吗（这道题什么序都是一样的）；count 函数也是一个二叉树遍历，用于寻找以该节点开头的目标值路径。好好体会吧！

分治算法

归并排序，典型的分治算法；分治，典型的递归结构。

分治算法可以分三步走：分解 -> 解决 -> 合并

1. 分解原问题为结构相同的子问题。
2. 分解到某个容易求解的边界之后，进行第归求解。
3. 将子问题的解合并成原问题的解。

归并排序，我们就叫这个函数`merge_sort`吧，按照我们上面说的，要明确该函数的职责，即**对传入的一个数组排序**。OK，那么这个问题能不能分解呢？当然可以！给一个数组排序，不就等于给该数组的两半分别排序，然后合并就完事了。

```
void merge_sort(一个数组) {
    if (可以很容易处理) return;
    merge_sort(左半个数组);
    merge_sort(右半个数组);
    merge(左半个数组, 右半个数组);
}
```

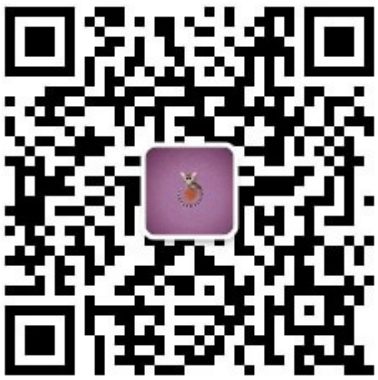
好了，这个算法也就这样了，完全没有任何难度。记住之前说的，相信函数的能力，传给他半个数组，那么这半个数组就已经被排好了。而且你会发现这不就是个二叉树遍历模板吗？为什么是后序遍历？因为我们分治算法的套路是 **分解 -> 解决（触底） -> 合并（回溯）** 啊，先左右分解，再处理合并，回溯就是在退栈，就相当于后序遍历了。至于`merge`函数，参考两个有序链表的合并，简直一模一样，下面直接贴代码吧。

下面参考《算法4》的 Java 代码，很漂亮。由此可见，不仅算法思想思想重要，编码技巧也是挺重要的吧！多思考，多模仿。

```
public class Merge {
    // 不要在 merge 函数里构造新数组了，因为 merge 函数会被多次调用，影响性能
    // 直接一次性构造一个足够大的数组，简洁，高效
    private static Comparable[] aux;

     public static void sort(Comparable[] a) {
        aux = new Comparable[a.length];
        sort(a, 0, a.length - 1);
    }

    private static void sort(Comparable[] a, int lo, int hi) {
        if (lo >= hi) return;
        int mid = lo + (hi - lo) / 2;
        sort(a, lo, mid);
        sort(a, mid + 1, hi);
        merge(a, lo, mid, hi);
    }

    private static void merge(Comparable[] a, int lo, int mid, int hi) {
        int i = lo, j = mid + 1;
        for (int k = lo; k <= hi; k++)
            aux[k] = a[k];
        for (int k = lo; k <= hi; k++) {
            if      (i > mid)              { a[k] = aux[j++]; }
            else if (j > hi)               { a[k] = aux[i++]; }
            else if (less(aux[j], aux[i])) { a[k] = aux[j++]; }
            else                           { a[k] = aux[i++]; }
        }
    }

    private static boolean less(Comparable v, Comparable w) {
        return v.compareTo(w) < 0;
    }
}
```

LeetCode 上有分治算法的专项练习，可复制到浏览器去做题：

https://leetcode.com/tag/divide-and-conquer/



==其他语言代码==