学习分治算法的一些心得

原创 龙泽文 LOA算法学习笔记 2021-02-21 18:38

分治算法全称"分而治之"（devide and conquer），指的是把一个复杂的问题分成两个或更多的相同或相似的子问题，直到最后子问题可以简单的直接求解，原问题的解即子问题的解的合并。

让我们从生活中的一个常见例子说起。假设你有一个存钱罐，里面有数量不等的不同面额的纸币和硬币，有一天你决定数一数存钱罐里总共有多少钱。面对这么一大堆钱，我们可以直接去计算它的总额，但这些数据对你的大脑来说过于庞大了，而且很容易算错。于是我们可以把这些钱分成几个小份来计算，然后将总数加起来就可以得到这堆钱的总额了。当然如果这些小份对你来说还是大了一些，我们可以继续划分为更小的几堆钱分别计算然后合并。我们可以这样做的原因是：

计算每个小堆钱的方式和计算最大堆钱的方式是相同的(区别在于体量上)。

大堆钱总和其实就是小堆钱结果之和。

这样其实就是一种分治的思想，它把一个复杂的问题分为两个或更多相同或相似的子问题，再把子问题分为更小的子问题，直到最后得到的子问题可以简单地直接求解。分治法是很多高效算法的基础，比如排序算法（快速排序、归并排序）、傅里叶变换（快速傅里叶变换）等等。

将父问题分解为多个子问题并求解和递归的概念很吻合，所以分治算法通常以递归的方式实现。分治算法除了分、治，还有一个合的过程：

分（devide）：递归解决较小的问题，到终止层或可以解决的时候停止。

治（conquer）：递归求解，如果问题足够小直接求解。

合（combine）：将子问题的解合并得到父问题的解。

一般来说，分治算法在解决的时候会分解为两个以上的递归调用，并且子类问题是互不影响的。想使用分治算法需要满足以下条件：

1. 原问题规模比较大，不容易直接解决，但是问题缩小到一定程度就可以比较容易地解决。
2. 问题可以分解为若干规模较小、求解方式相似的子问题，并且子问题之间求解是独立的，互不影响。
3. 合并问题分解的子问题可以得到问题的解。

分治算法重要的是一种思想，注重的的问题分、治、合的过程。而递归是一种工具，这种工具通过自己调用自己形成一个来回的过程，而分治就是利用了多次这样的来回过程。

分治算法的时间复杂度通常这样计算：

在分治算法中的三个步骤中， 我们假设分解和合并过程所用的时间分别为D(n),C(n), 设T(n)为处理一个规模为n的序列所消耗的时间为子序列个数，每一个子序列是原序列的1/b，α为把每个问题分解成α个子问题,则所消耗的时间为：

如果n<=c(c是n中一个可以直接求解的规模)，T(n)= O（1）

否则 T(n)= αT(n/b)+ D(n) + C(n)。

在快速排序中,α 是为2的, b也为2, 则分解(就是取参照点,可以认为是1),合并(把数组合并,为n), 因此D(n) + C(n) 是一个线性时间O(n). 这样时间就变成了T(n) = 2T(n/2) +O(n).

在每个层上的时间复杂度为:在第一层上是cn(c为比较一次时所用的时间), 在第二层上时数组被分成了两部分, 每部分为n/2, 则在第二层上时间为 c * n/2 + c* n/2 =cn, 同样在第三层上, 被分成了四部分, 时间为c*n/4 +c*n/4 + c*n/4 + c*n/4 =cn. 层高一共是按刚才说的是Log2n层,每一层上都是cn, 所以共消耗时间cn * Log2n; 则总时间cn * Log2n +cn = cn(1+Log2n) 即O(nLog2n).

下面通过几道例题更具体地说明一下：

一、快速排序

快速排序的基本思想是通过一趟排序将待排记录分隔成独立的两部分，其中一部分记录的关键字均比另一部分的关键字小，则可分别对这两部分记录继续进行相同方式的排序，以达到整个序列有序。

整个数组被不断分割为更小的数组进行排序，最后整合起来，正是分治算法的思想。

```
1  void quick_sort(int s[], int l, int r){
2      if (l < r){
3          int i = l, j = r, x = s[l];
4          while (i < j)
```

```
        {
            while(i < j && s[j] >= x)//从右向左找第一个小于x的数
                j--;
            if(i < j)
                s[i++] = s[j];


            while(i < j && s[i] < x)//从左向右找第一个大于等于x的数
                i++;
            if(i < j)
                s[j--] = s[i];
        }
        s[i] = x;
        quick_sort(s, l, i - 1);//递归调用
        quick_sort(s, i + 1, r);
    }
}
```

二、汉诺塔

在汉诺塔游戏中，有三个分别命名为A、B、C的塔座，几个大小各不相同，从小到大依次编号的圆盘，每个圆盘中间有一个小孔。最初，所有的圆盘都在A塔座上，其中最大的圆盘在最下面，然后是第二大，以此类推. 游戏的目的是将所有的圆盘从塔座A移动到塔座B;塔座C用来放置临时圆盘。一次只能移动一个圆盘；任何时候都不能将一个较大的圆盘压在较小的圆盘上面；除了第二条限制，任何塔座的最上面的圆盘都可以移动到其他塔座上。

在解决汉诺塔问题时,事实上我们不是最关心圆盘1开始应该挪到哪个塔座上，而是最关心最下面的圆盘4.当然,我们不能直接移动圆盘4，但是圆盘4最终将从塔座A移动到塔座B.按照游戏规则，在移动圆盘4之前的情况一定为1、2、3按大小排列在C上，4单独在A上。

那么我们如何将前三个圆盘从A移动到C呢？应该是前两个圆盘从A移动到B，然后将圆盘3从A移动到C，最后将前两个圆盘从B移动到C的过程。实际上这里和上面的思路是一致的。

我们继续简化问题，最终我们将只需要处理一个圆盘从一个塔座移动到另一个塔座的问题. 当n=1时，也就是刚开始A石柱上仅仅摆放一个圆盘，那么直接将圆盘从A石柱上移动到B石柱上即可。当n=2时，从上往下按照大小顺序将圆盘编为1号和2号，那么要将圆盘全部从石柱A移动到石柱C，首先需要将1号圆盘移动到石柱B，再将2号圆盘移动到石柱C，最后将1号圆盘移动到石柱C。当n=3时，仍然从上往下按照大小顺序将圆盘编为1号、2号和3号，此时由于问题相对复杂，所以1号和2号圆盘看做一个圆盘，即1+2号圆盘，此时需要解决的就是将1+2号圆盘和3号圆盘移动到石柱C的问题，即先将1+2号圆盘移动到石柱B，再将3号圆盘移动到石柱C，最后将1+2号圆盘移动到石柱C即可。由于每次只能移动一个圆盘，那么如果要将1+2号圆盘移动到石柱B，需要将1+2号圆盘拆分为两个个体，看做将1号和2号圆盘移动到石柱B，同理将1+2号圆盘移动到石柱C。依此类推，那么到n=n时，将圆盘自上向下编为1号、2号、3号……n号，同理将1号到n-1号圆盘看做一个圆盘，即 ∑n1(n−1)∑1n(n−1)号圆盘，此时解决的就是将 ∑n1(n−1)∑1n(n−1)号圆盘和n号圆盘移动到石柱C的问题，即先将∑n1(n−1)∑1n(n−1)号圆盘移动到石柱B，再将n号圆盘移动到石柱C，最后将∑n1(n−1)∑1n(n−1)号圆盘移动到石柱C。因为将第n号圆盘移动到石柱C后，无论前n-1个圆盘怎么移动，都不需要再次移动第n号圆盘，即父问题与子问题相对独立且互不影响，因此可以将∑n1(n−1)∑1n(n−1)号圆盘的问题同理向下拆分移动。

下面是伪代码：

```
func tower(n,a,b,c)
    if(n==1)
        a->c//代表从a直接到c
    else
        tower(n-1,a,c,b)
        a->c//a上剩下的最大的盘子移动到c
```

```
7         tower(n-1,b,a,c)
```