分治算法详解：运算优先级

读完本文，你不仅学会了算法套路，还可以顺便解决如下题目：

牛客	LeetCode	力扣	难度
-	241. Different Ways to Add Parentheses	241. 为运算表达式设计优先级	🟠

————

我们号已经写了 动态规划算法，回溯（DFS）算法，BFS 算法，贪心算法，双指针算法，滑动窗口算法，现在就差个分治算法没写了，今天来写一下。

其实，我觉得回溯、分治和动态规划算法可以划为一类，因为它们都会涉及递归。

回溯算法就一种简单粗暴的算法技巧，说白了就是一个暴力穷举算法，比如让你用回溯算法求子集、全排列、组合，你就穷举呗，就考你会不会漏掉或者多算某些情况。

动态规划是一类算法问题，肯定是让你求最值的。因为动态规划问题拥有 最优子结构，可以通过状态转移方程从小规模的子问题最优解推导出大规模问题的最优解。

分治算法呢，可以认为是一种算法思想，通过将原问题分解成小规模的子问题，然后根据子问题的结果构造出原问题的答案。这里有点类似动态规划，所以说运用分治算法也需要满足一些条件，你的原问题结果应该可以通过合并子问题结果来计算。

其实这几个算法之间界定并没有那么清晰，有时候回溯算法加个备忘录似乎就成动态规划了，而分治算法有时候也可以加备忘录进行剪枝。

我觉得吧，没必要过分纠结每个算法的定义，定义这东西无非文学词汇而已，反正能把题做出来你说这是啥算法都行，**所以大家还是得多刷题，刷出感觉，各种算法都手到擒来**。

最典型的分治算法就是归并排序了，核心逻辑如下：

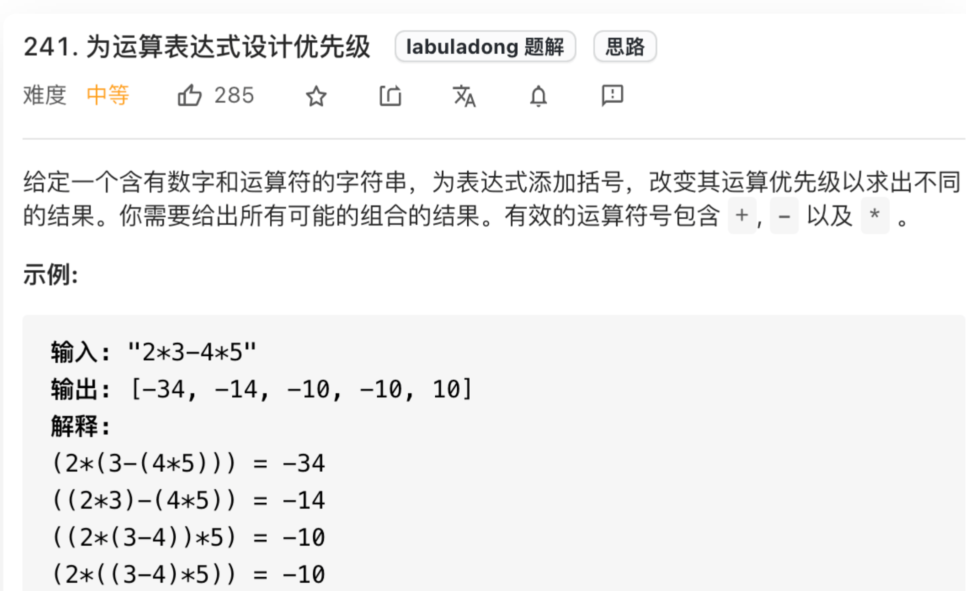
```java
void sort(int[] nums, int lo, int hi) {
    int mid = (lo + hi) / 2;
    /****** 分 ******/
    // 对数组的两部分分别排序
    sort(nums, lo, mid);
    sort(nums, mid + 1, hi);
    /****** 治 ******/
    // 合并两个排好序的子数组
    merge(nums, lo, mid, hi);
}
```

「对数组排序」是一个可以运用分治思想的算法问题，只要我先把数组的左半部分排序，再把右半部分排序，最后把两部分合并，不就是对整个数组排序了吗？

下面来看一道具体的算法题。

添加括号的所有方式

我来借力扣第 241 题「为运算表达式设计优先级」来讲讲什么是分治算法，先看看题目：



241. 为运算表达式设计优先级 labuladong 题解 思路

难度 中等 285

给定一个含有数字和运算符的字符串，为表达式添加括号，改变其运算优先级以求出不同的结果。你需要给出所有可能的组合的结果。有效的运算符号包含 +, - 以及 * 。

示例:

```
输入: "2*3-4*5"
输出: [-34, -14, -10, -10, 10]
解释:
(2*(3-(4*5))) = -34
((2*3)-(4*5)) = -14
((2*(3-4))*5) = -10
(2*((3-4)*5)) = -10
```

```
(((2*3)-4)*5) = 10
```

简单说，就是给你输入一个算式，你可以给它随意加括号，**请你穷举出所有可能的加括号方式，并计算出对应的结果**。

函数签名如下：

```
// 计算所有加括号的结果
List<Integer> diffWaysToCompute(String input);
```

看到这道题的第一感觉肯定是复杂，我要穷举出所有可能的加括号方式，是不是还要考虑括号的合法性？是不是还要考虑计算的优先级？

是的，这些都要考虑，但是不需要我们来考虑。利用分治思想和递归函数，算法会帮我们考虑一切细节，也许这就是算法的魅力吧，哈哈哈。

废话不多说，解决本题的关键有两点：

1、不要思考整体，而是把目光聚焦局部，只看一个运算符。

这一点我们前文经常提及，比如 手把手刷二叉树第一期 就告诉你解决二叉树系列问题只要思考每个节点需要做什么，而不要思考整棵树需要做什么。

说白了，解决递归相关的算法问题，就是一个化整为零的过程，你必须瞄准一个小的突破口，然后把问题拆解，大而化小，利用递归函数来解决。

2、明确递归函数的定义是什么，相信并且利用好函数的定义。

这也是前文经常提到的一个点，因为递归函数要自己调用自己，你必须搞清楚函数到底能干嘛，才能正确进行递归调用。

下面来具体解释下这两个关键点怎么理解。

我们先举个例子，比如我给你输入这样一个算式：

`1 + 2 * 3 - 4 * 5`

请问，这个算式有几种加括号的方式？请在一秒之内回答我。

估计你回答不出来，因为括号可以嵌套，要穷举出来肯定得费点功夫。

不过呢，嵌套这个事情吧，我们人类来看是很头疼的，但对于算法来说嵌套括号不要太简单，一次递归就可以嵌套一层，一次搞不定大不了多递归几次。

所以，作为写算法的人类，我们只需要思考，如果不让括号嵌套（即只加一层括号），有几种加括号的方式？

还是上面的例子，显然我们有四种加括号方式：

`(1) + (2 * 3 - 4 * 5)`

`(1 + 2) * (3 - 4 * 5)`

`(1 + 2 * 3) - (4 * 5)`

`(1 + 2 * 3 - 4) * (5)`

发现规律了么？**其实就是按照运算符进行分割，给每个运算符的左右两部分加括号**，这就是之前说的第一个关键点，不要考虑整体，而是聚焦每个运算符。

现在单独说上面的第三种情况：

`(1 + 2 * 3) - (4 * 5)`

我们用减号 `-` 作为分隔，把原算式分解成两个算式 `1 + 2 * 3` 和 `4 * 5`。

分治分治，分而治之，**这一步就是把原问题进行了「分」，我们现在要开始「治」了。**

`1 + 2 * 3` 可以有两种加括号的方式，分别是：

`(1) + (2 * 3) = 7`

`(1 + 2) * (3) = 9`

或者我们可以写成这种形式：

`1 + 2 * 3 = [9, 7]`

而 `4 * 5` 当然只有一种加括号方式，就是 `4 * 5 = [20]`。

然后呢，你能不能通过上述结果推导出 `(1 + 2 * 3) - (4 * 5)` 有几种加括号方式，或者说有几种不同的结果？

显然，可以推导出来 `(1 + 2 * 3) - (4 * 5)` 有两种结果，分别是：

`9 - 20 = -11`

`7 - 20 = -13`

那你可能要问了， `1 + 2 * 3 = [9, 7]` 的结果是我们自己看出来的，如何让算法计算出来这个结果呢？

这个简单啊，再回头看下题目给出的函数签名：

```java
// 定义：计算算式 input 所有可能的运算结果
List<Integer> diffWaysToCompute(String input);
```

这个函数不就是干这个事儿的吗？**这就是我们之前说的第二个关键点，明确函数的定义，相信并且利用这个函数定义**。

你甭管这个函数怎么做到的，你相信它能做到，然后用就行了，最后它就真的能做到了。

那么，对于 `(1 + 2 * 3) - (4 * 5)` 这个例子，我们的计算逻辑其实就是这段代码：

```java
List<Integer> diffWaysToCompute("(1 + 2 * 3) - (4 * 5)") {
    List<Integer> res = new LinkedList<>();
    /****** 分 ******/
    List<Integer> left = diffWaysToCompute("1 + 2 * 3");
    List<Integer> right = diffWaysToCompute("4 * 5");
    /****** 治 ******/
    for (int a : left)
        for (int b : right)
            res.add(a - b);

    return res;
}
```

好，现在 `(1 + 2 * 3) - (4 * 5)` 这个例子是如何计算的，你应该完全理解了吧，那么回来看我们的原始问题。

原问题 `1 + 2 * 3 - 4 * 5` 是不是只有 `(1 + 2 * 3) - (4 * 5)` 这一种情况？是不是只能从减

号 `-` 进行分割？

不是，每个运算符都可以把原问题分割成两个子问题，刚才已经列出了所有可能的分割方式：

`(1) + (2 * 3 - 4 * 5)`

`(1 + 2) * (3 - 4 * 5)`

`(1 + 2 * 3) - (4 * 5)`

`(1 + 2 * 3 - 4) * (5)`

所以，我们需要穷举上述的每一种情况，可以进一步细化一下解法代码：

```java
List<Integer> diffWaysToCompute(String input) {
    List<Integer> res = new LinkedList<>();
    for (int i = 0; i < input.length(); i++) {
        char c = input.charAt(i);
        // 扫描算式 input 中的运算符
        if (c == '-' || c == '*' || c == '+') {
            /****** 分 ******/
            // 以运算符为中心，分割成两个字符串，分别递归计算
            List<Integer>
                left = diffWaysToCompute(input.substring(0, i));
            List<Integer>
                right = diffWaysToCompute(input.substring(i + 1));
            /****** 治 ******/
            // 通过子问题的结果，合成原问题的结果
            for (int a : left)
                for (int b : right)
                    if (c == '+')
                        res.add(a + b);
                    else if (c == '-')
                        res.add(a - b);
                    else if (c == '*')
                        res.add(a * b);
        }
    }
    // base case
    // 如果 res 为空，说明算式是一个数字，没有运算符
    if (res.isEmpty()) {
        res.add(Integer.parseInt(input));
    }
    return res;
}
```

有了刚才的铺垫，这段代码应该很好理解了吧，就是扫描输入的算式 `input`，每当遇到运算符就进行分割，递归计算出结果后，根据运算符来合并结果。

这就是典型的分治思路，先「分」后「治」，先按照运算符将原问题拆解成多个子问题，然后通过子问题的结果来合成原问题的结果。

当然，一个重点在这段代码：

```
// base case
// 如果 res 为空，说明算式是一个数字，没有运算符
if (res.isEmpty()) {
    res.add(Integer.parseInt(input));
}
```

递归函数必须有个 base case 用来结束递归，其实这段代码就是我们分治算法的 base case，代表着你「分」到什么时候可以开始「治」。

我们是按照运算符进行「分」的，一直这么分下去，什么时候是个头？显然，当算式中不存在运算符的时候就可以结束了。

那为什么以 `res.isEmpty()` 作为判断条件？因为当算式中不存在运算符的时候，就不会触发 if 语句，也就不会给 `res` 中添加任何元素。

至此，这道题的解法代码就写出来了，但是时间复杂度是多少呢？

如果单看代码，真的很难通过 for 循环的次数看出复杂度是多少，所以我们需要改变思路，本题在求所有可能的计算结果，不就**相当于在求算式 `input` 的所有合法括号组合**吗？

那么，对于一个算式，有多少种合法的括号组合呢？这就是著名的「卡特兰数」了，最终结果是一个组合数，推导过程稍有些复杂，我这里就不写了，有兴趣的读者可以自行搜索了解一下。

其实本题还有一个小的优化，可以进行递归剪枝，减少一些重复计算，比如说输入的算式如下：

`1 + 1 + 1 + 1 + 1`

那么按照算法逻辑，按照运算符进行分割，一定存在下面两种分割情况：

`(1 + 1) + (1 + 1 + 1)`

`(1 + 1 + 1) + (1 + 1)`

算法会依次递归每一种情况，其实就是冗余计算嘛，所以我们可以对解法代码稍作修改，加一个备忘录来避免这种重复计算：

```java
// 备忘录
HashMap<String, List<Integer>> memo = new HashMap<>();

List<Integer> diffWaysToCompute(String input) {
    // 避免重复计算
    if (memo.containsKey(input)) {
        return memo.get(input);
    }
    /****** 其他都不变 ******/
    List<Integer> res = new LinkedList<>();
    for (int i = 0; i < input.length(); i++) {
        // ...
    }
    if (res.isEmpty()) {
        res.add(Integer.parseInt(input));
    }
    /***********************/

    // 将结果添加进备忘录
    memo.put(input, res);
    return res;
}
```

当然，这个优化没有改变原始的复杂度，只是对一些特殊情况做了剪枝，提升了效率。

最后总结

解决上述算法题利用了分治思想，以每个运算符作为分割点，把复杂问题分解成小的子问题，递归求解子问题，然后再通过子问题的结果计算出原问题的结果。

把大规模的问题分解成小规模的问题递归求解，应该是计算机思维的精髓了吧，建议大家多练，如果本文对你有帮助，记得分享给你的朋友哦~

接下来可阅读：

- 回溯算法和动态规划到底谁是谁爹