时间复杂度分析快速入门：题型分类法

原创 nettee 面向大象编程 2020-07-12

收录于话题
#编程 4773 #算法 4130 #数据结构 1265

最近，有不少读者留言或私信问我时间复杂度的分析方法。时间复杂度说难也不难，说简单也不简单，但它一定是我们学习算法的过程中过不去的一道坎。这篇文章就想给大家介绍一种快速分析时间复杂度的方法 —— 题型分类法。

网络上关于时间复杂度分析的文章，大部分都充斥着教材里的陈词滥调：先讲一大堆数学概念，什么增长速度、渐进复杂度，看得让人摸不着头脑。我今天的文章不想跟你讲这些虚的，只想跟你讲讲最实用的技巧，让你迅速搞定面试中的时间复杂度分析。

我们在面试中需要分析时间复杂度，无非是以下两种情况：

- 在写代码之前，预估自己代码的时间复杂度
- 写好代码之后，给面试官分析代码的时间复杂度

无论是哪种情况，你一定都清楚这道题的题型。那么只要记住不同题型的时间复杂度分析方法，二叉树套用二叉树的分析法、动态规划套用动态规划的分析法，就能做到快速分析时间复杂度了。

下面是我总结出的不同题型的时间复杂度分析方法：

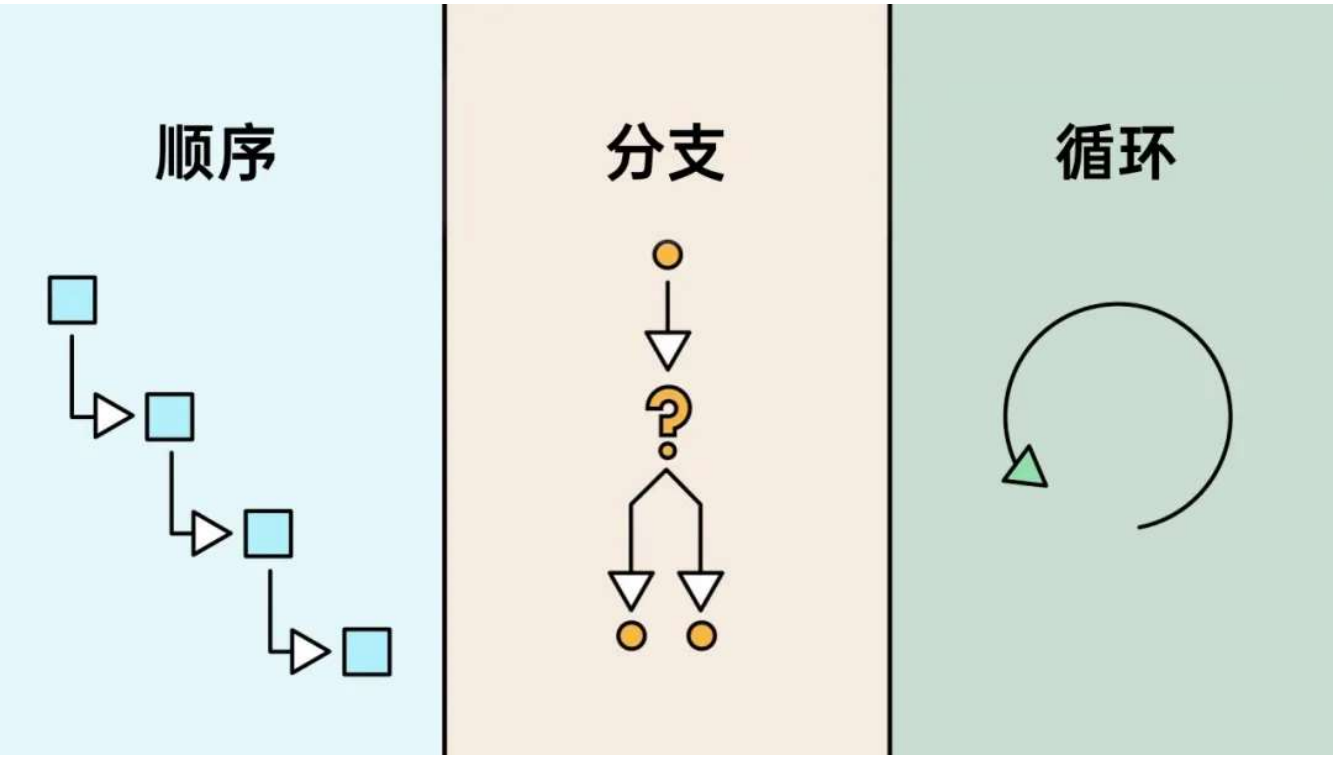
题型	时间复杂度 分析方法
数组 链表 动态规划	数循环法
二叉树 回溯法 DFS/BFS	整体法
分治法 二分查找	log n 分析法

本文将介绍「数循环法」与「整体法」两种方法、六类题型的全部分析技巧。针对这六类题型，文末有按不同题型分类的往期文章回顾，供大家参考！

一、数循环法

数循环法，就是看代码中有几个循环、每个循环会执行几次，来确定算法的时间复杂度。

我们必须要掌握的一个基础知识是：在程序的三大基本结构（顺序、分支、循环）中，只有循环能真正影响时间复杂度。一般情况下，我们可以直接忽略代码中的顺序结构、分支结构，直接看循环结构来判断时间复杂度。



程序的三大基本结构

数循环法典型题型：动态规划类题目

「数循环法」最典型的题型就是动态规划了。我们以一道经典的「最长公共子序列（LCS）」问题来看看数循环法是怎么发挥作用的。

（以下题解代码来自文章：**LeetCode 例题精讲 | 15 最长公共子序列：二维动态规划的解法**，不了解 LCS 问题或不清楚解法的同学，可以参考这篇文章。）

最长公共子序列（LCS）的题解代码如下：

```
public int longestCommonSubsequence(String s, String t) {
    if (s.isEmpty() || t.isEmpty()) {
        return 0;
    }
    int m = s.length();
    int n = t.length();
    int[][] dp = new int[m+1][n+1];
```

```
        int[][] dp = new int[m+1][n+1];
        for (int i = 0; i <= m; i++) {
            for (int j = 0; j <= n; j++) {
                if (i == 0 || j == 0) {
                    dp[i][j] = 0;
                } else {
                    if (s.charAt(i-1) == t.charAt(j-1)) {
                        dp[i][j] = dp[i-1][j-1] + 1;
                    } else {
                        dp[i][j] = Math.max(dp[i-1][j], dp[i][j-1]);
                    }
                }
            }
        }
        return dp[m][n];
    }
```

根据数循环法的基本思路，我们可以直接忽略掉代码中的各个细节，只看循环结构：

```
    public int longestCommonSubsequence(String s, String t) {
        ...
        for (int i = 0; i <= m; i++) {
            for (int j = 0; j <= n; j++) {
                ...
            }
        }
        ...
    }
```

数循环法要数两个东西：循环的层数、每个循环执行的次数。在这个例子中，代码有两个循环，都是简单的 for-i 循环：

- 外层循环，执行 $m+1$ 次，数量级是 m；
- 内层循环，执行 $n+1$ 次，数量级是 n。

那么，算法的时间复杂度就是内外层循环次数之积，也就是 $O(mn)$。

循环的不同层数：数组类题目

很多数组类题目也可以用数循环法来分析时间复杂度。根据题目和解法的不同，循环可能有 1~3 层。数循环法对于不同层数的循环都是有效的。

例如，经典的「最大子数组和」题目（**LeetCode 53. Maximum Subarray Sum**），从暴力的三重循环到动态规划的单层循环解法都有。

- 暴力法，三重循环：

```
public int maxSubArray(int[] nums) {
    int n = nums.length;
    int res = Integer.MIN_VALUE;
    for (int i = 0; i < n; i++) {
        for (int j = i; j < n; j++) {
            int sum = 0;
            for (int k = i; k <= j; k++) {
                sum += nums[k];
            }
            res = Math.max(res, sum);
        }
    }
    return res;
}
```

时间复杂度 $O(n^3)$。

- 暴力法改进，二重循环：

```
public int maxSubArray(int[] nums) {
    int n = nums.length;
    int res = Integer.MIN_VALUE;
    for (int i = 0; i < n; i++) {
        int sum = 0;
        for (int j = i; j < n; j++) {
            sum += nums[j];
            res = Math.max(res, sum);
        }
    }
    return res;
}
```

时间复杂度 $O(n^2)$。

- 动态规划解法，一层循环：

```
public int maxSubArray(int[] nums) {
    int n = nums.length;
```

```
        int[] dp = new int[n+1];
        dp[0] = 0;

        int res = Integer.MIN_VALUE;
        for (int k = 1; k <= n; k++) {
            dp[k] = Math.max(dp[k-1], 0) + nums[k-1];
            res = Math.max(res, dp[k]);
        }
        return res;
    }
```

时间复杂度 $O(n)$。

不理解问题的动态规划解法原理的同学，可以参考这篇文章：**LeetCode 例题精讲 | 16 最大子数组和：子数组类问题的动态规划技巧**

while 循环的处理方式：链表类题目

在上面的例子中，代码中的循环都是简单的 for-i 循环。有些情况下，代码中的循环是 while 循环，或是一些复杂的 for 循环。对于这种情况，我们就需要仔细分析循环的执行次数。

链表题型会经常用到 while 循环。我们用这道经典的「快慢指针」题目进行分析：**LeetCode 876 - Middle of the Linked List**。

（以下题解代码来自文章：**LeetCode 例题精讲 | 05 双指针×链表问题：快慢指针**，对题目或解法不了解的同学可以参考。）

```
    public ListNode middleNode(ListNode head) {
        ListNode fast = head;
        ListNode slow = head;
        while (fast != null && fast.next != null) {
            // fast 一次前进两个元素，slow 一次前进一个元素
            fast = fast.next.next;
            slow = slow.next;
        }
        // 链表元素为奇数个时，slow 指向链表的中点
        // 链表元素为偶数个时，slow 指向链表两个中点的右边一个
        return slow;
    }
```

可以看到，while 循环在 `fast` 指针到达链表尾部时结束。而 `fast` 指针初始位于链表的头部，一次前进两个元素。设链表的长度为 n，那么这个 while 循环会执行 $n/2$ 次，时间复杂度

是 $O(n/2)$（也可以写成 $O(n)$，数量级是一样的）。

二、整体法

整体法一般涉及到递归和数据结构，不能直接用循环次数判定时间复杂度，而是要看代码中处理了几个元素，来确定算法的时间复杂度。

使用整体法进行分析的题型有：二叉树、回溯法、DFS、BFS 等。下面我们分别举例介绍。

二叉树：看结点的个数

《LeetCode 例题精讲》系列中写过好几次二叉树的解题技巧。这些二叉树问题的解法都有一个共性：

- 使用递归遍历二叉树
- 二叉树的每个结点都遍历一次

实际上，大部分二叉树问题都可以使用子问题方法与「三步走」套路来求解。我专门写过一篇文章总结二叉树问题的套路：二叉树问题太复杂？「三步走」方法解决它！

二叉树问题既然都是用递归求解，就没有循环来让我们使用「数循环法」。这时候，我们就需要使用整体法，通过遍历结点的个数来判断时间复杂度。

二叉树问题的时间复杂度离不开一个变量 N，也就是二叉树结点的数量。对于大部分简单的二叉树问题，代码都是只遍历每个结点一次。这种情况下，算法的时间复杂度就是 $O(N)$。

不过对于稍难一些的题目，也可能出现「重复遍历」的情况。例如 **LeetCode 437. Path Sum III** 这道题：

> 给定一个二叉树，它的每个结点都存放着一个整数值。找出路径和等于给定数值的路径总数。
>
> 路径不需要从根结点开始，也不需要在叶结点结束，但是路径方向必须是向下的（只能从父结点到子结点）。

由于题目中的路径可能位于二叉树的中部，不少人会写出这样的题解代码：

```
public int pathSum(TreeNode root, int sum) {
    if (root == null) {
        return 0;
```

```
            return 0;
        }
        return rootPathSum(root, sum)
            + pathSum(root.left, sum)
            + pathSum(root.right, sum);
    }

    int rootPathSum(TreeNode root, int sum) {
        if (root == null) {
            return 0;
        }
        int target = sum - root.val;
        return (target == 0 ? 1 : 0)
            + rootPathSum(root.left, target)
            + rootPathSum(root.right, target);
    }
```

这是一个典型的「双递归」代码，存在 `pathSum` 和 `rootPathSum` 两个递归函数，它们会相互调用。这样，一个结点会不止遍历一次，算法的时间复杂度就不能简单地用 $O(N)$ 来表示。

也就是说，如果想在二叉树问题中写出 $O(N)$ 时间复杂度的代码，我们需要保证一个结点只会被遍历一次。

回溯法：看结果的个数

回溯法类题目有一个非常简单的时间复杂度分析指南：有多少个结果，时间复杂度就是多少。例如：

- **LeetCode 78. Subsets**，求数组所有可能的子集。对于大小为 n 的数组，所有可能的子集的个数为 2^n，那么回溯算法的时间复杂度就是 $O(2^n)$。
- **LeetCode 46. Permutations**，求数组所有可能的全排列。对于大小为 n 的数组，所有可能的全排列的个数为 $n!$（阶乘），那么回溯算法的时间复杂度就是 $O(n!)$。

为什么是这样分析的呢？这是因为回溯法的原理就是不断尝试可能的结果，遇到走不通的地方再撤回（回溯）尝试下一个方案。那么，回溯法尝试的次数和结果的次数是相对应的。我们可以根据结果的数量级来判断回溯法的时间复杂度。

需要注意的时，回溯算法的时间复杂度一般都很大，如 $O(2^n)$、$O(n!)$、$O(n^n)$ 之类的。这是回溯法的特点决定的，一般也不需要做进一步的优化。

DFS/BFS：看元素的个数

很多同学还没有掌握 DFS/BFS 类题目分析的窍门。其实这类题目的分析方法很简单：不论是 DFS 还是 BFS，都是用来遍历所有元素的，那么算法的时间复杂度就是元素的个数！例如：

- 如果是遍历二叉树，设二叉树的结点个数为 N，那么 DFS/BFS 的时间复杂度就是 $O(N)$；
- 如果是遍历图，设图中的结点个数为 N，那么 DFS/BFS 的时间复杂度就是 $O(N)$；
- 如果是遍历网格结构，设二维网格的边长是 n，那么 DFS/BFS 的时间复杂度就是 $O(n^2)$。

DFS 代码一般是使用递归，只能使用整体法。而 BFS 代码中一般会有 1~2 层的循环。有的同学可能会疑惑于究竟使用数循环法还是整体法。例如 BFS 层序遍历：

```
// 二叉树的层序遍历
void bfs(TreeNode root) {
    Queue<TreeNode> queue = new ArrayDeque<>();
    queue.add(root);
    while (!queue.isEmpty()) {
        int n = queue.size();
        for (int i = 0; i < n; i++) {
            // 变量 i 无实际意义，只是为了循环 n 次
            TreeNode node = queue.poll();
            if (node.left != null) {
                queue.add(node.left);
            }
            if (node.right != null) {
                queue.add(node.right);
            }
        }
    }
}
```

（关于 BFS 层序遍历的详细讲解，参见文章：**LeetCode 例题精讲 | 13 BFS 的使用场景：层序遍历、最短路径问题**）

在这段代码中，使用了两层循环，乍一看是平方级别的时间复杂度。然而，内层循环的次数 n 是非常有限的，并不能简单地将这段代码的时间复杂度分析为平方级别。

如果用整体法来分析的话。考虑队列 `queue`，二叉树的每个结点都会进队一次、出队一次。那么循环的次数恰好也是二叉树的结点数。设二叉树的结点个数是 N，则算法的时间复杂度是 $O(N)$。

因此，我们在分析 BFS 算法的时间复杂度时，要使用整体法，而不要关注具体的循环。

总结

本文介绍了两种时间复杂度分析方法：「数循环法」与「整体法」，用于分析六类题型：动态规划、数组、链表、二叉树、回溯法、DFS/BFS。基本上已经涵盖了常见的算法题目种类。

准备面试的时间总是有限的。在准备算法题时，我们的主要精力还是要放在题目的解法上。时间复杂度分析的学习不应追求面面俱到，而是先做到能上手分析，后续再不断补充学习。希望本文列举的时间复杂度分析方法能够对你有所帮助。

关于分治法、二分搜索的时间复杂度，由于分析方法比较复杂，在后面会有专门的文章详细讲解，敬请期待。

以下是与六类题型相关的往期文章分类回顾：

动态规划：

- **LeetCode** 例题精讲 | **14** 打家劫舍问题：动态规划的解题四步骤
- **LeetCode** 例题精讲 | **15** 最长公共子序列：二维动态规划的解法
- 经典动态规划：编辑距离
- **LeetCode** 例题精讲 | **16** 最大子数组和：子数组类问题的动态规划技巧
- **LeetCode** 例题精讲 | **17** 动态规划如何拆分子问题，简化思路
- 经典动态规划：「换硬币」系列三道问题详解

数组：

- **LeetCode** 例题精讲 | **18** 前缀和：空间换时间的技巧

链表：

- **LeetCode** 例题精讲 | **01** 反转链表：如何轻松重构链表
- **LeetCode** 例题精讲 | **05** 双指针×链表问题：快慢指针

二叉树：

- **LeetCode** 例题精讲 | **02 Path Sum**：二叉树的子问题划分
- **LeetCode** 例题精讲 | **10** 二叉树直径：二叉树遍历中的全局变量
- **LeetCode** 例题精讲 | **11** 二叉树转化为链表：二叉树遍历中的相邻结点
- 二叉树问题太复杂？「三步走」方法解决它！

回溯法：

- **LeetCode** 例题精讲 | **03** 从二叉树遍历到回溯算法
- **LeetCode** 例题精讲 | **08** 排列组合问题：回溯法的候选集合
- **LeetCode** 例题精讲 | **09** 排列组合问题再探：回溯法的去重策略

- 一套代码解决 **Combination Sum** 系列问题（**LeetCode 39/40/216**）

DFS/BFS：

- **LeetCode** 例题精讲 **| 12** 岛屿问题：网格结构中的 **DFS**
- **LeetCode** 例题精讲 **| 13 BFS** 的使用场景：层序遍历、最短路径问题

我是 nettee，致力于分享面试算法的解题套路，让你真正掌握解题技巧，做到举一反三。我的《LeetCode 例题精讲》系列文章正在写作中，关注我的公众号，获取最新文章。

原创不易，欢迎分享、点赞和「在看」⇩