如何衡量算法的性能

> 本节部分内容截取自我的新书，完整内容请访问 https://leetcode-solution.cn/book-intro 进行试读

介绍

学习算法，首先要知道的就是如何判断一个算法的好坏。好的程序有很多的评判标准，包括但不限于可读性，扩展性，性能等。这里我们来看其中一项指标—— 性能 。 坏的程序性能不一定差，但是好的程序通常性能都比较好。那么如何分析一个算法的性能好坏呢？这就是我们要讲的复杂度分析，所有的数据结构教程都会把这个放在前面来讲，不仅是因为它们是基础，更因为它们真的非常重要。学会了复杂度分析，你才能够对算法进行分析，从而帮助你写出复杂度更优的算法。如果你对一种算法的复杂度推导很熟悉，那么我相信你已经掌握了这个算法。本章主要介绍时间复杂度，空间复杂度的分析方法也是类似，并且相对于时间复杂度，大多数情况下空间复杂度的分析更容易。如果你对复杂度不熟悉，或者看完本文还是不太理解其含义和用法，那么建议你搭配《算法》（第四版）中 1.4 算法分析 一起学习。

时间复杂度和空间复杂度分别衡量程序运行时间的长短和运行时所占空间的大小。如何衡量一个程序运行时间长短，占用内存大小呢？内存倒还好，但是运行时长，由于不同计算机的性能不同，执行时间也会不同，甚至有可能有数倍的差距，那么究竟应该如何衡量 程序运行时间的长短呢 ？

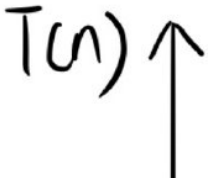
《计算机程序设计艺术》的作者高德纳（Donald Knuth）提出了一种方法，这种方法的核心思想很简单，就是 一个程序运行时间主要和两个因素有关，分别是1. 执行每条语句的耗时 2.执行每条语句的频率 。而前者取决于硬件，后者取决于算法本身和程序的输入。那么如何统计算法 执行每条语句的频率 呢？我们举个例子来说明。

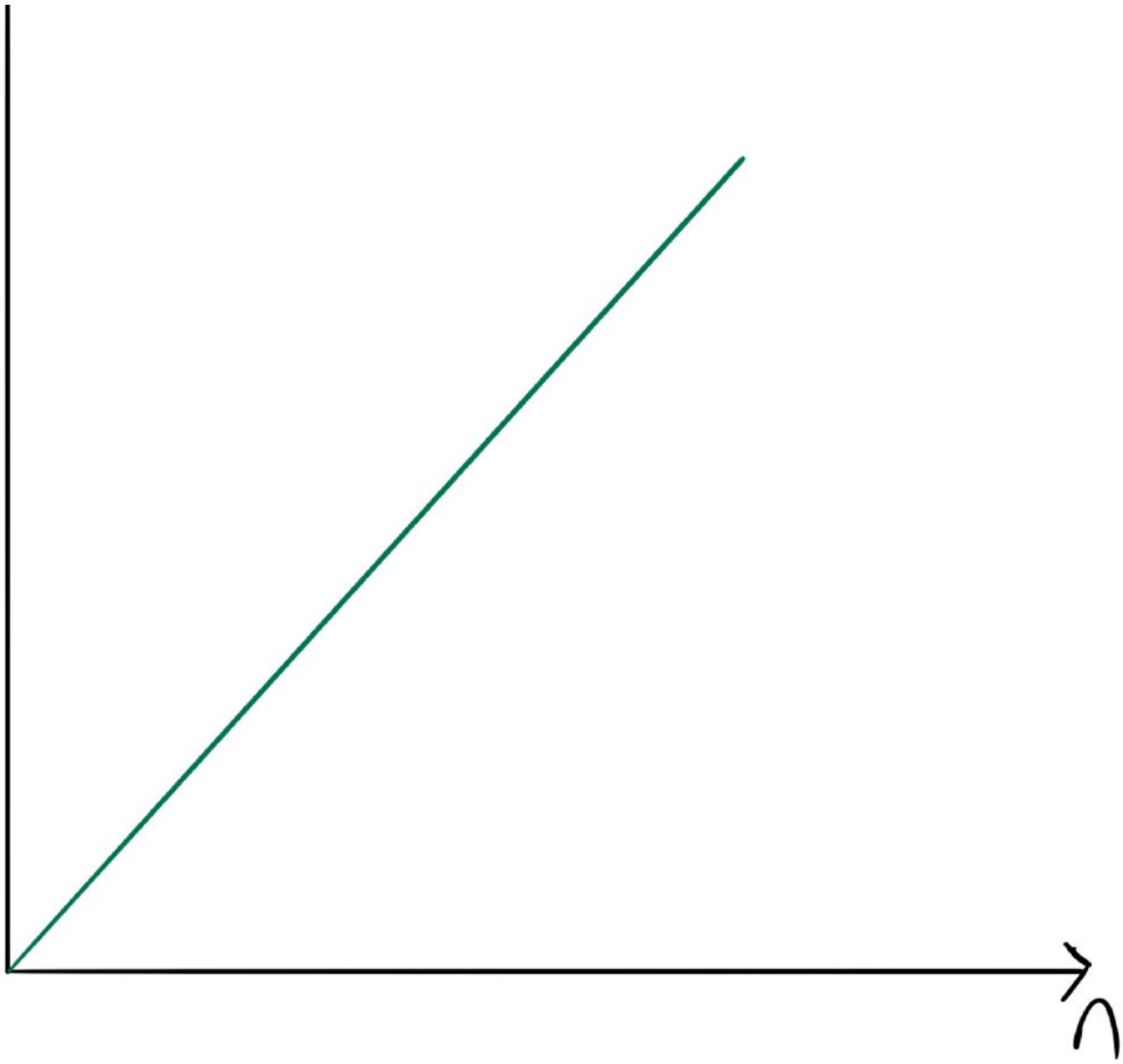
如下是一个从 1 累加到 n 的一个算法，这个算法用了一层循环，并且借助了一个变量 res 来完成。

```
def sum(n: int) -> int:
    res = 0
    for i in range(1, n + 1):
        res += i
    return res
```

（代码 1.3.1）

我们将这个方法从更加微观的角度来分析一下。上述代码会执行 N 次循环体的内容，假设每一次执行都是常数时间，不妨假设其执行时间是 x， `res = 0` 和 `return res` 的执行时间分别为 y 和 z。 那么总的时间就等于 $\mathbf{n} * \mathbf{x} + \mathbf{y} + \mathbf{z}$, 如果 粗略 地将 x，y 和 z 都看成一样的，那么可以得出总时间为 $(\mathbf{n} + \mathbf{2}) * \mathbf{x}$。 如果用图来表示的话就是这样的：





（图 数据规模和操作数的关系图）

换句话说算法的运行时间和数据的规模成正比。

在渐进意义上讲，我们常常忽略较小项，如上的 $2 * x$，而仅保留最大项，如上的 $n * x$，这样可以大大减少分析工作量，因此这种复杂度分析方法也被成为渐进复杂度分析。实际上这在现实中也很常见，即 程序运行时间往往取决于其中一小部分指令 。

大 O 表示法

以上正是一种叫做大 O 表示法的基本思想，它是一种描述算法性能的记法，这种描述和编译系统、机器结构、处理器的快慢等因素无关。 这种描述的参数是 N，表示数据的规模。 这里的 O 表示量级（order），比如说“二分查找的时间复杂度是 $O(logN)$”，就是说它需要“通过 $logN$ 量级的操作去查找一个规模为 N 的数据结构”。这种估测对算法的理论分析和大致比较是非常有价值的，我们可以很快地对算法进行一个大致的估算。

例如一个拥有较小常数项的 $O(N^2)$ 算法在规模 N 较小的情况下可能比一个高常数项的$O(N)$ 算法运行地更快。但是随着 N 足够大以后，具有较慢上升趋势的算法必然运行地更快，因此在采用大 O 表示复杂度的时候，可以忽略系数，这也是我们可以忽略不同性能计算机执行差异的原因， 因为你可以把不同性能计算机性能差异看成是系数的差异 。

除此之外，我们还应该区分算法的最好情况，最坏情况和平均情况，但是这不在本书的讨论范畴，本书的所有复杂度如不做特殊说明，均指的是**最坏复杂度。**

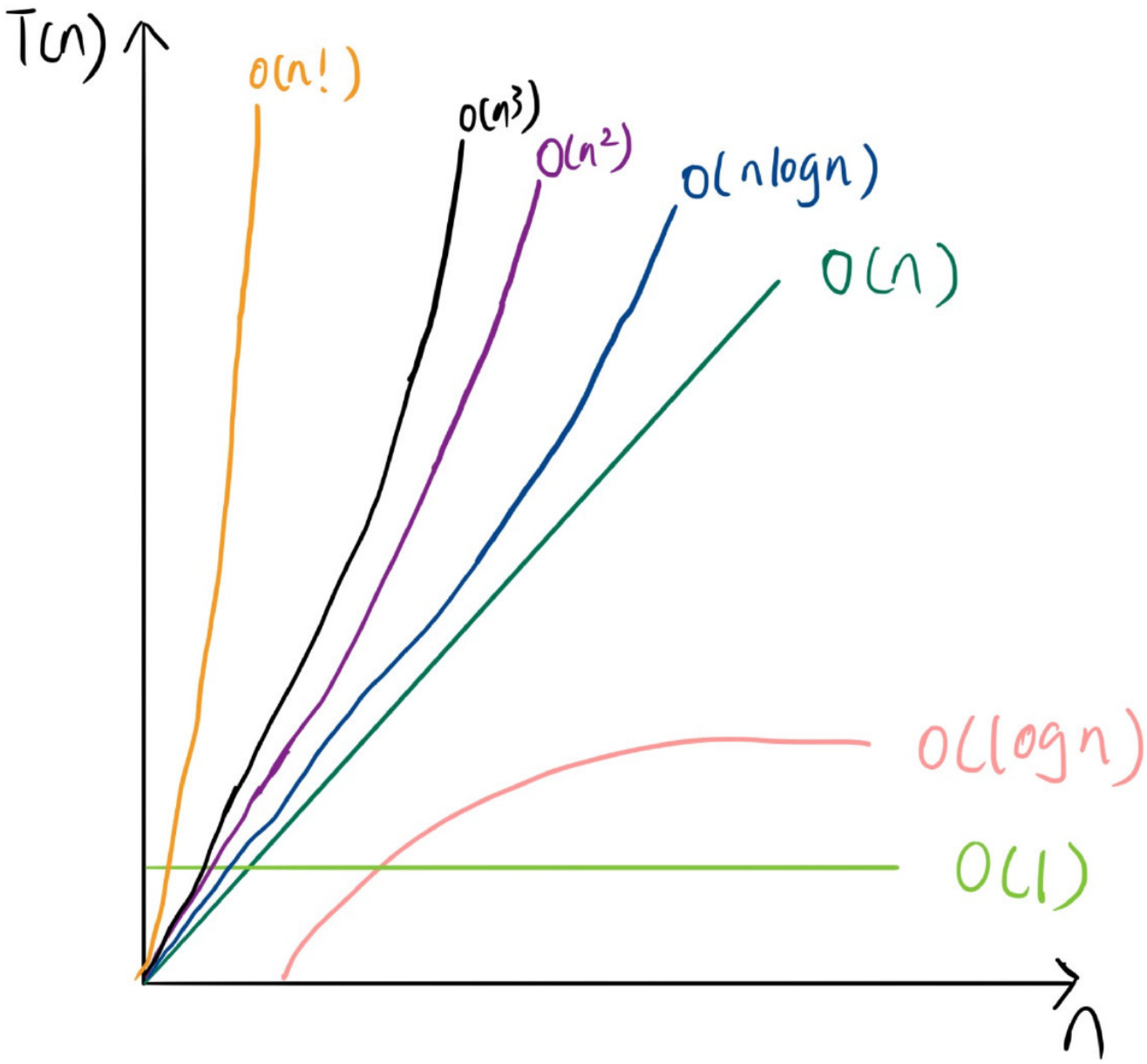
> 实际上大 O 表示就是来表示最坏复杂的的，而平均复杂度和最好复杂度不是这个符号，不过不需要大家掌握。

空间复杂度也是类似， 不过空间复杂度往往更好分析，因为很少空间复杂度会有诸如对数情况。

不过需要注意的是，递归的空间复杂度容易被大家忽略，也就是说每次开辟调用栈空间容易被大家忽略。我们可以将递归算法，看作是使用了栈的迭代算法。而其栈的空间就是递归中调用栈的空间。因此递归的空间复杂度需要额外加上开辟的栈空间。

值得注意的是本书全部内容的空间复杂度均指的是额外空间复杂度。比如题目给的入参，题目**要求的**的返回值是不计入空间复杂度的。我们只计算由于采用了这个算法，而不得不多用的**空间**。

最后我给出了这几种常见的时间复杂度的趋势图对比，大家可以直观地感受一下趋势变化。



（图 各种复杂度的渐进趋势对比）

复杂度分析的意义

如果你不会复杂度分析， 说明你很可能没有彻底明白这道题，从这个意义上将可以帮你找到算法短板。

复杂度分析可以帮助你快速排除不可能的算法。 比如你想到一个算法的时间复杂度是指数， 而题目的数组范围是 10^6， 这基本可以认为是不对的算法。因为出题人不太会出一个明显看起来不靠谱的题。

总结

复杂度分析是对算法执行效率的一个粗略评价，可以大致地锁定一个算法的性能。

我们研究算法的目的其实就是提高算法执行的效率，这种效率可以是时间上的，有可以是空间上的。 而实际业务中，时间上的优化更重要，因此我们也会讲解很多空间换时间的技巧。

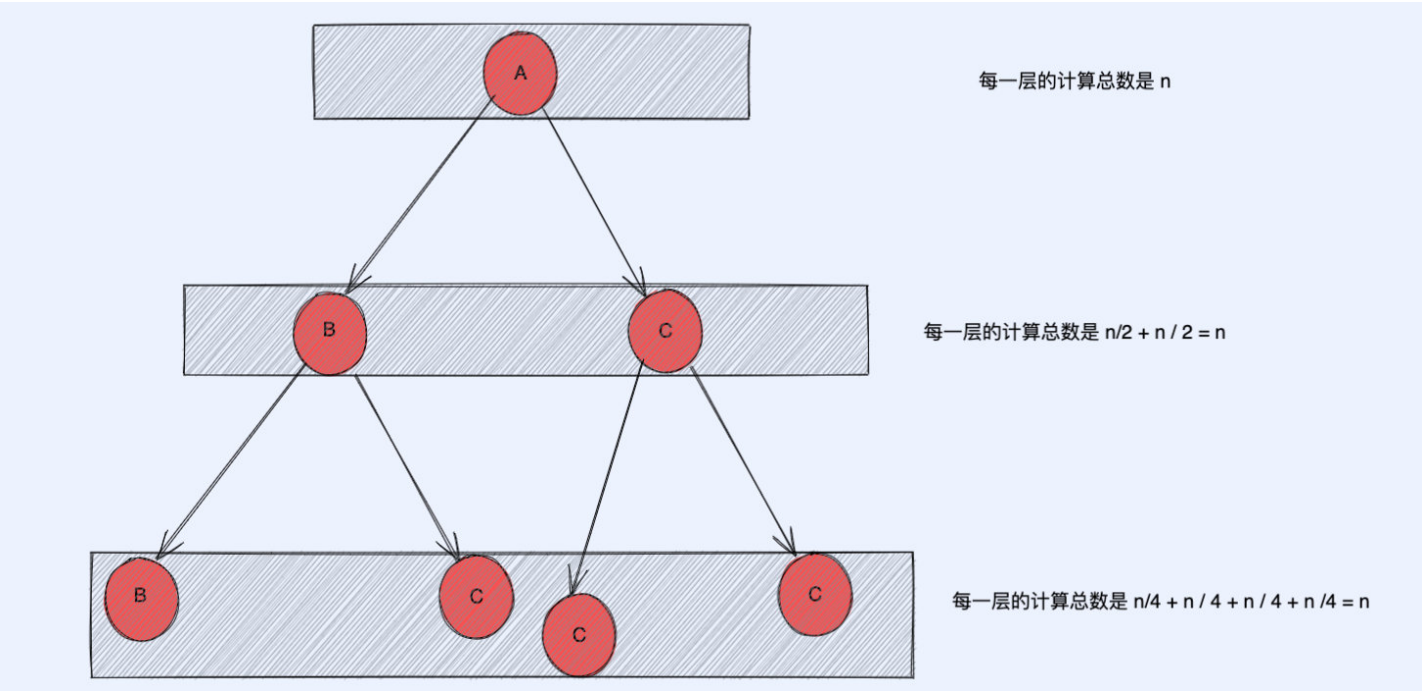
对于时间复杂度来说，我们只需要关注算法的基本操作的次数即可。给大家几个实操技巧是关注：

1. 如果有循环，则关注下最内层循环的执行次数。

> 不是两层循环就是 n ^ 2，而是看最内层执行了多少次

2. 如果有递归那就是：递归树的节点数 * 递归函数内部的基础操作数。而这句话的前提是所有递归函数内部的基本操作数是一样的，这样才能直接乘。

而如果递归函数的基本操作数不一样，可以看下递归树每一层的基础操作数是否一样，如果每一层的基础操作数一样，则可以通过**递归深度 * 每一层基础操作数**计算得出，比如归并排序的复杂度分析中就用到了这个技巧。这里我画个图方便大家理解：



每一层基础操作数相同的递归树分析

不过这种时间复杂度只能应付简单的情况。如果递归函数内部操作数不那么确定，则必须通过列出转移方程，然后根据方程推导出基础操作数的数量级，这样就得出了时间复杂度。这部分内容我已经写到了我的新书中了。

3. 如果用了一些 API。比如数组用了头部添加的 API，不要忽略了这是一个时间复杂度 $O(n)$ 的操作。

额外空间复杂度稍微简单一点。我们只需要看下算法用了什么样的集合数据结构就行了，而不需关心非集合数据结构，比如哈希表，数组等等，关心它是如何随着数据规模变化而变化就好了。比如我写动态规划，初始化了一个 $O(m * n)$ 的二维矩阵，那么此时空间复杂度就已经是 $O(m * n)$ 了。

而布尔值，number 值 等等不需要考虑。 上面的做法其实没有考虑递归，如果你用了递归，那么多考虑一个递归栈就行了，递归栈的开销就是递归树的深度。

复杂度分析很重要， 说其**最重要**都不为过。因此大家打卡的时候，尽量要给出复杂度的分析。

当你对某一个算法分析不出来的时候， 可以请教我们的导师，一来二去慢慢就会了。相信我，只要每道题你不仅做出来，还对复杂度分析透透的，很快算法你就上道了。

上一页 下一页