算法时空复杂度分析实用指南

————

我以前的文章主要都是讲解算法的原理和解题的思维，对时间复杂度和空间复杂度的分析经常一笔带过，主要是基于以下两个原因：

1、对于偏小白的读者，我希望你集中精力理解算法原理。如果加入太多偏数学的内容，很容易把人劝退。

2、正确理解常用算法底层原理，是进行复杂度的分析的前提。尤其是递归相关的算法，只有你从树的角度进行思考和分析，才能正确分析其复杂度。

鉴于现在历史文章已经涵盖了所有常见算法的核心原理，所以我专门写一篇时空复杂度的分析指南，授人以鱼不如授人以渔，教给你一套通用的方法分析任何算法的时空复杂度。

本文会篇幅较长，会涵盖如下几点：

1、Big O 表示法的几个基本特点。

2、非递归算法中的时间复杂度分析。

3、数据结构 API 的效率衡量方法（摊还分析）。

4、递归算法的时间/空间复杂度的分析方法，这部分是重点，我会用动态规划和回溯算法举例。

废话不多说了，接下来一个个看。

Big O 表示法

首先看一下 Big O 记号的数学定义：

> `O(g(n))` = { `f(n)`：存在正常量 `c` 和 `n_0`，使得对所有 `n ≥ n_0`，有 `0 ≤ f(n) ≤ c*g(n)` }

我们常用的这个符号 `O` 其实代表一个函数的集合，比如 `O(n^2)` 代表着一个由 `g(n) = n^2` 派生出来的一个函数集合；我们说一个算法的时间复杂度为 `O(n^2)`，意思就是描述该算法的复杂度的函数属于这个函数集合之中。

理论上，你看明白这个抽象的数学定义，就可以解答你关于 Big O 表示法的一切疑问了。

但考虑到大部分人看到数学定义就头晕，我给你列举两个复杂度分析中会用到的特性，记住这两个就够用了。

1、只保留增长速率最快的项，其他的项可以省略。

首先，乘法和加法中的常数因子都可以忽略不计，比如下面的例子：

```
O(2N + 100) = O(N)
O(2^(N+1)) = O(2 * 2^N) = O(2^N)
O(M + 3N + 99) = O(M + N)
```

当然，不要见到常数就消，有的常数消不得：

```
O(2^(2N)) = O(4^N)
```

除了常数因子，增长速率慢的项在增长速率快的项面前也可以忽略不计：

```
O(N^3 + 999 * N^2 + 999 * N) = O(N^3)
O((N + 1) * 2^N) = O(N * 2^N + 2^N) = O(N * 2^N)
```

以上列举的都是最简单常见的例子，这些例子都可以被 Big O 记号的定义正确解释。如果你遇到更复杂的复杂度场景，也可以根据定义来判断自己的复杂度表达式是否正确。

2、Big O 记号表示复杂度的「上界」。

换句话说，只要你给出的是一个上界，用 Big O 记号表示就都是正确的。

比如如下代码：

```
for (int i = 0; i < N; i++) {
    print("hello world");
}
```

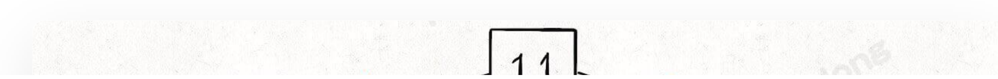
如果说这是一个算法，那么显然它的时间复杂度是 `O(N)`。但如果你非要说它的时间复杂度是 `O(N^2)`，严格意义上讲是可以的，因为 `O` 记号表示一个上界嘛，这个算法的时间复杂度确实不会超过 `N^2` 这个上界呀，虽然这个上界不够「紧」，但符合定义，所以没毛病。

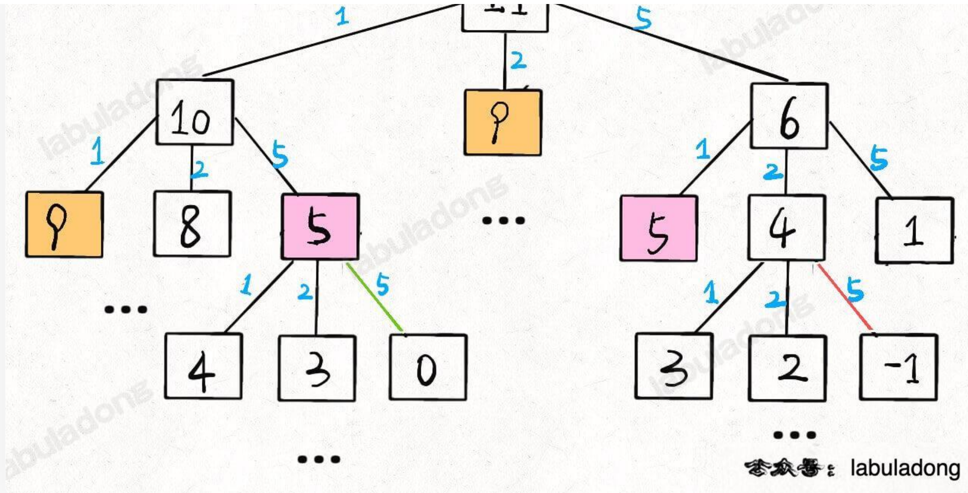
上述例子太简单，非要扩大它的时间复杂度上界显得没什么意义。但有些算法的复杂度会和算法的输入数据有关，没办法提前给出一个特别精确的时间复杂度，那么在这种情况下，用 Big O 记号扩大时间复杂度的上界就变得有意义了。

比如前文 动态规划核心框架 中讲到的凑零钱问题的暴力递归解法，核心代码框架如下：

```
// 定义：要凑出金额 n，至少要 dp(coins, n) 个硬币
int dp(int[] coins, int amount) {
    // base case
    if (amount <= 0) return;
    // 状态转移
    for (int coin : coins) {
        dp(coins, amount - coin);
    }
}
```

当 `amount = 11, coins = [1,2,5]` 时，算法的递归树就长这样：





后文会具体讲递归算法的时间复杂度计算方法，现在我们先求一下这棵递归树上的节点个数吧。

假设金额 `amount` 的值为 `N`，`coins` 列表中元素个数为 `K`，那么这棵递归树就是一棵 `K` 叉树。但这棵树的生长和 `coins` 列表中的硬币面额有直接的关系，所以这棵树的形状会很不规则，导致我们很难精确地求出树上节点的总数。

对于这种情况，比较简单的处理方式就是按最坏情况做近似处理：

这棵树的高度有多高？不知道，那就按最坏情况来处理，假设全都是面额为 1 的硬币，这种情况下树高为 `N`。

这棵树的结构是什么样的？不知道，那就按最坏情况来处理，假设它是一棵满 `K` 叉树好了。

那么，这棵树上共有多少节点？都按最坏情况来处理，高度为 `N` 的一棵满 `K` 叉树，其节点总数为等比数列求和公式 `(K^N - 1)/K`，用 Big O 表示就是 `O(K^N)`。

当然，我们知道这棵树上的节点数其实没有这么多，但用 `O(K^N)` 表示一个上界是没问题的。

所以，有时候你自己估算出来的时间复杂度和别人估算的复杂度不同，并不一定代表谁算错了，可能你俩都是对的，只是是估算的精度不同，一般来说只要数量级（线性/指数级/对数级/平方级等）能对上就没问题。

在算法领域，除了用 Big O 表示渐进上界，还有渐进下界、渐进紧确界等边界的表示方法，有兴趣的读者可以自行搜索。不过从实用的角度看，以上对 Big O 记号表示法的讲解就够用了。

－－－－－－－－－－－－－－

应合作方要求，本文不便在此发布，请扫码关注回复关键词「复杂度」或 点这里 查看：

共同维护高质量学习环境，评论礼仪见这里，违者直接拉黑不解释

0 Comments - *powered by utteranc.es*

Write Preview

Sign in to comment

Styling with Markdown is supported

Sign in with GitHub