408 补充讲义

Bai_Yu

May 2024

目录

1 写在前面

本补充讲义是针对“王道计算机考研 2025”的四本教材：《数据结构》、《计算机组成原理》、《操作系统》、《计算机网络》做的补充讲义，用于补充书上没有提及，但是实际上已经出现在题目中的内容。

对于书上已经出现的内容，如果其对于理解补充内容有重要意义，我会将其补充在书中；如果无意义，我并不会补充。也就是说，本补充讲义所面向的读者是正在使用该教材的人士。在阅读某一块内容之前，比如数据结构的内容，我们希望读者已经通读原教材中“数据结构”的相关知识。

由于本人学识浅薄，所整理的内容可能存在缺漏或错误。欢迎读者批评指正。

2 数据结构

2.1 算法

算法，顾名思义，就是计算的方法。此计算方法不同于计算机中的“计算方法”学科，那个学科探索的是计算机在做计算时怎样提高解精度的问题，而我们常说的“算法”可以看作一种广义的映射。当带着数据信息的数据结构通过算法之后，就会被映射到对应的解。一般地，计算机学科的“算法”有这样的广为接受的定义：

𝔇𝔢𝔣 2.1.1 (算法)：一个有穷的指令集，当这些指令为解决某一特定任务规定了一个运算序列时，称作一个**算法**。

由于计算机学科是一个偏实践的学科，其中的定义大都不要求背诵。所以上边这个定义看看就得了。接下来我们看一看算法有什么性质：

2.1.2 算法的性质：算法具有**确切性**、**有穷性**、**有效性**。

确切性，说的是算法的每一步指令不能有歧义；**有穷性**，说的是算法应在执行有穷步后结束；**有效性**，说的是每一个步骤都可用计算机指令实现。算法有三大性质，正如集合有三大性质（确定性、互异性、无序性）。

有了算法的定义，我们不难联想：当我们要解决某个特定问题时，我们可能会设计出不同而正确的多种算法。这就引出了两个问题：一是如何设计出算法？二是如何评价这些算法的好坏？这两个问题我们在下面两节里重点地聊一聊。

2.1.1 算法的设计

算法的设计，是计算机专业看家的一门重要学科。但是为了引入数据结构，我在这里还是不得不提一下这方面的内容，希望可以起到一个抛砖引玉的作用。以下的说法假定读者对 C/C++ 程序设计有一定的基础。

2.2.1 暴力算法：毫不使用技巧，而是直接靠暴力枚举或暴力搜索解决问题的算法。

这一算法无论是在理解上还是代码书写上都是简单的。我们以排序问题为例：

排序问题：**有 n 个数，请把这些数按照由小到大的顺序排列**。

这个问题显然可以使用暴力搜索求解，例如，我们可以找到序列中最小的数置于顶端，再找到次小的数置于第二位，以此类推。我们称这样的暴力搜索算法叫**插入排序**，其正确性是显然的。接下来，我们再介绍一种经典的暴力算法：依次对相邻两个元素的值进行两两比较，若发现逆序则交换，使值较大的元素逐渐从前移向后部。你会发现，这样一来，值较小的元素就像气泡浮出水面一样“浮”到了序列的顶端。因此这一算法有一个形象的名字——**冒泡排序**。其正确性可以使用**归纳法**证明。读者可以自行证明并完成代码实现。

聪明的你肯定会思考起来，是否存在其他的排序方法呢？答案是肯定的。例如，我们可以将待排序元素分成大小大致相同的两个子集合，分别对两个子集合进行排序，最终将排好序的子集合合并成为所要求的排好序的集合。那么这就涉及到三个过程：分（Divide）、治（Conquer）、合（Combine），这种排序算法叫**归并排序**。另外，我们把使用了分、治、合思想的算法统称为**分治算法**。其正确性依然可以使用**归纳法**证明。

2.2.2 分治算法：把规模较大的大问题分为数个规模较小的子问题，然后通过求解子问题一步步求出大问题的解的算法。

容易发现的是，当我们使用分治算法的时候，我们可能需要把 $n \times k$ 规模的问题压缩为 n 个规模为 k 的问题。为了方便理解，我们暂且机械地把求解该问题的算法定义为函数 $f(x)$ ，其中自变量 x 代表问题的规模。那么当我们想要求解 $f(n \times k)$ 的时候，我们就要调用 n 次 $f(k)$。像这种直接或间接地调用自身的算法称为**递归算法**。如果你是计算机专业的学生，相信你一定写过使用递归求**整数阶乘**和使用递归求解 **Hanoi 塔问题**的算法。下面是一个使用递归求**第 n 个 Fibonacci 数**的示例代码:

```
int F(int n)
{
 if(n == 0)
 return 0;
 else if(n == 1)
 return 1;
 else
 return F(n - 1) + F(n - 2);
}
```

我们再来思考这样的问题：分治算法对于子问题彼此独立的情况具有良好的求解效果。但是当子问题存在重叠部分的时候，相同子问题将被重复计算多次，浪费计算资源。例如上面的递归求解斐波那契数列的函数就存在这样的问题。对于这种问题的解决方案，相信每位读者都有自己的理解。我们在本章的习题中再做讨论。

2.1.2 算法复杂度

在前面的学习中，我们看到：对于同一个问题，可能有多个算法能够求出正确的解。那么如何来评价这些算法的优劣程度呢？定性的研究显然是不恰当的。我们应当定义一个或多个**定量**的标准来比较同一问题的不同算法的效率。为此，我们思考以下问题：

对于一个问题，有两个不同的算法 A 和 B。当给定的数据规模为 n 时，算法 A 需要占用系统内规模为 $\log_2 n$ 的存储空间，而算法 B 需要占用系统内规模为 n 的存储空间。那么显然算法 A **在内存占用上**是优于算法 B 的。

对于一个问题，有两个不同的算法 A 和 B。当给定的数据规模为 n 时，算法 A 需要计算 n 次，而算法 B 需要计算 n^2 次。那么显然算法 A **在计算次数上**是优于算法 B 的。

为了定量描述上述内容，我们引入算法复杂度的概念。

$\mathfrak{Def}$ 2.3.1 (空间复杂度)：算法中定义的所有量所占的存储空间之和为算法的**空间复杂度**。

$\mathfrak{Def}$ 2.3.2 (时间复杂度)：算法中的所有语句运行次数之和为算法的**时间复杂度**。

时间复杂度一般为数据规模 n 的函数 $T(n)$。我们知道，在数学中，我们会使用**渐进表达式** $T(n) = O(f(n))$ 来表示 $T(n)$ 的上界为 $f(n)$, 即：$n \to \infty$, $\exists k \in \mathbb{N}_+$, $s.t. T(n) \le kf(n)$。因此，当我们能够求出数据规模为 n 的算法具有上界 $f(n)$ 时，称**算法的时间复杂度上界为** $O(f(n))$。例如，冒泡排序算法的算法复杂度上界为 $O(n^2)$ 。一般地，当我们提到算法复杂度的时候，说的都是时间复杂度。

类似地, 我们还可以定义时间复杂度的下界。即: 对于函数 $f(n)$, 若 $n \to \infty$, $\exists k \in \mathbb{N}_+$, $s.t. T(n) \ge kf(n)$, 记作 $T(n) = \Omega(f(n))$, 称**算法的时间复杂度下界为** $\Omega(f(n))$。例如，冒泡排序算法的算法复

杂度下界为 $\Omega(n^2)$ 。

对于像冒泡排序这样的算法, 它具有一个相等的上界和下界, 都是 $f(n)=n^2$, 这时我们就可以称**算法的时间复杂度为** $\Theta(f(n))$。即: 存在这样的函数 $f(n)$, 使得:

$$n\to\infty,\ \exists k_1,k_2\in\mathbb{N}_+,\ s.t.\ k_1f(n)\le T(n)\le k_2f(n).$$

在实际的问题中，我们往往要处理规模很大的数据。因此，考察时间复杂度的下界对于算法优劣程度意义不大。一般地，我们只通过考察复杂度的上界来考察算法的优劣程度。不难理解的是，对于一个算法, 其上界函数 $f(n)$ 的无穷大量阶数越高, 其算法的劣势越明显。例如，$O(n)$ 的算法就要比 $O(n^2)$ 的算法要好。

而对于空间复杂度的情况，有以下结论：假定现在有一个解决问题 K 的算法，在这一算法中，我们需要建立 k 个规模为 n 的数组。那么该算法的空间复杂度就是 $O(kn)$，当 k 为常数时，我们可以认为其复杂度为 $O(n)$。类似地，如果我们需要建立 k 个规模为 n^2 的二维数组，空间复杂度就是 $O(kn^2)$，当 k 为常数时，我们可以认为其复杂度为 $O(n^2)$。

接下来我们讨论一些算法复杂度的表示问题。

2.3.3 复杂度的性质：根据数学中上界的定义，算法复杂度具有以下性质:

性质 1： $O(kf(n))=O(f(n)),\ k\in\mathbb{N}_+,$

性质 2：$O(f(n))+O(g(n))=O(f(n)+g(n))=max\{f(n),g(n)\}.$

性质 1 是显然成立的。下面证明性质 2：设函数 $T(n)$ 的两个上界函数分别为 $f(n)$ $g(n)$，即:

$$n\to\infty,\ \exists k_1\in\mathbb{N}_+,\ s.t.T(n)\le k_1f(n)$$

$$n\to\infty,\ \exists k_2\in\mathbb{N}_+,\ s.t.T(n)\le k_2g(n)$$

则有：$T(n) = O(f(n) + g(n))$，其中 k_1, k_2 分别为 k, k' 中的最大值。

1.3.4 算法复杂度分析：在算法设计中，我们常常需要估计算法的复杂度，以判断是否有更好的改进方法。常见的算法复杂度包括：$O(n!)$，$O(x^n)$（其中 $x > 0$），$O(n^\alpha \log n)$（其中 $\alpha > 0$），$O(n^\alpha)$（其中 $\alpha > 0$），$O(\log n)$，$O(1)$。这里的对数一般以 2 为底。

下面，我们来分析分治算法的复杂度。为了计算分治算法的复杂度，我们需要考虑分治算法进行了多少次运算。

以分治法递归求解 Fibonacci 数列第 n 项的算法为例：我们可以画出递归树, 其中树的高度为 n。递归的出口 (叶子节点) 是 $n = 0$ 或 $n = 1$，这些步骤的复杂度是 $O(1)$；对于非叶节点，其值为两个孩子的值加和，复杂度也是 $O(1)$。因此，整个递归算法的复杂度等于整棵递归树中的节点个数，即 $O(2^n - 1)$，也就是 $O(2^n)$。

如果你对树、二叉树以及叶子节点的概念不熟悉，可以参考第四章的相关内容。另外，对于分治算法的复杂度计算，我们有以下结论：

假设一个分治算法的每一步能够把一个规模为 n 的大问题分割成 k 个规模为 $\frac{n}{m}$ 的子问题，且分割和合并的总复杂度为 $O(n^i)$，则整体分治算法的复杂度可表示为：

$$T(n) = kT(\frac{n}{m}) + O(n^i)$$

可以解得：

$$T(n) = \begin{cases} \Theta(n^{\log_m k}) & \log_m k > i \\ \Theta(n^i \log n) & \log_m k = i \\ \Theta(n^i) & \log_m k < i \end{cases}$$

上述结论称为求解分治算法复杂度的**主方法（Master Method）**。

其正确性的证明并不是必须的，但如果你感兴趣，可以自行探索或查找相关材料加以理解。

举例来说