第一部分 基础知识

这一部分将引导读者开始思考算法的设计和分析问题，简单介绍算法的表达方法、将在本书中用到的一些设计策略，以及算法分析中用到的许多基本思想。本书后面的内容都是建立在这些基础知识之上的。

第 1 章是对算法及其在现代计算系统中地位的一个综述。本章给出了算法的定义和一些算法的例子。此外，本章还说明了算法是一项技术，就像快速的硬件、图形用户界面、面向对象系统和网络一样。

在第 2 章中，我们给出了书中的第一批算法，它们解决的是对 n 个数进行排序的问题。这些算法是用 Java 代码形式给出的，尽管 Java 代码比较冗长，但是足够清晰地表达了算法的结构。我们分析的排序算法是插入排序，它采用了一种增量式的做法；另外还分析了归并排序，它采用了一种递归技术，称为“分治法”。尽管这两种算法所需的运行时间都随 n 的值而增长，但增长的速度是不同的。我们在第 2 章分析了这两种算法的运行时间，并给出了一种有用的表示方法来表达这些运行时间。

第 3 章给出了这种表示法的准确定义，称为渐近表示。在第 3 章的一开始，首先定义几种渐近符号，它们主要用于表示算法运行时间的上界和下界。第 3 章余下的部分主要给出了一些数学表示方法。这一部分的作用更多的是为了确保读者所用的记号能与本书的记号体系相匹配，而不是教授新的数学概念。

第 4 章更深入地讨论了第 2 章引入的分治法，给出了更多分治法的例子，包括用于两方阵相乘的 Strassen 方法。第 4 章包含了求解递归式的方法。递归式用于描述递归算法的运行时间。“主方法”是一种功能很强的技术，通常用于解决分治算法中出现的递归式。

第一章 算法在计算中的作用

什么是算法？为什么算法值得研究？相对于计算机中使用的其他技术来说算法的作用是什么？本章我们将回答这些问题。

1.1 算法

非形式化地说，**算法** (algorithm) 就是任何良定义的计算过程，该过程取某个值或值的集合作为**输入**并产生某个值或值的集合作为**输出**。这样算法就是把输入转换成输出的计算步骤的一个序列。

我们也可以把算法看成是用于求解良说明的**计算问题**的工具。一般来说，问题陈述说明了期望的输入/输出关系。算法则描述一个特定的计算过程来实现该输入/输出关系。

例如，我们可能需要把一个数列排成非递减序。实际上，这个问题经常出现，并且为引入许多标准的设计技术和分析工具提供了足够的理由。下面是我们关于**排序问题**的形式化定义。

输入：n 个数的一个序列 $\langle a_0, a_1, \ldots, a_{n-1} \rangle$ 。

输出：输入序列的一个排列 $\langle a'_0, a'_1, \ldots, a'_{n-1} \rangle$，满足 $a'_0 \le a'_1 \le \cdots \le a'_{n-1}$ 。

例如，给定输入序列 $\langle 31, 41, 59, 26, 41, 58 \rangle$，排序算法将返回序列 $\langle 26, 31, 41, 41, 58, 59 \rangle$ 作为输出。这样的输入序列称为排序问题的一个实例 （instance） 。一般来说，问题实例由计算该问题解所必需的（满足问题陈述中强加的各种约束的）输入组成。

因为许多程序使用排序作为一个中间步，所以排序是计算机科学中的一个基本操作。因此，已有许多好的排序算法供我们任意使用。对于给定应用，哪个算法最好依赖于以下因素：将被排序的项数、这些项已被稍微排序的程度、关于项值的可能限制、计算机的体系结构，以及将使用的存储设备的种类（主存、磁盘或者磁带）。

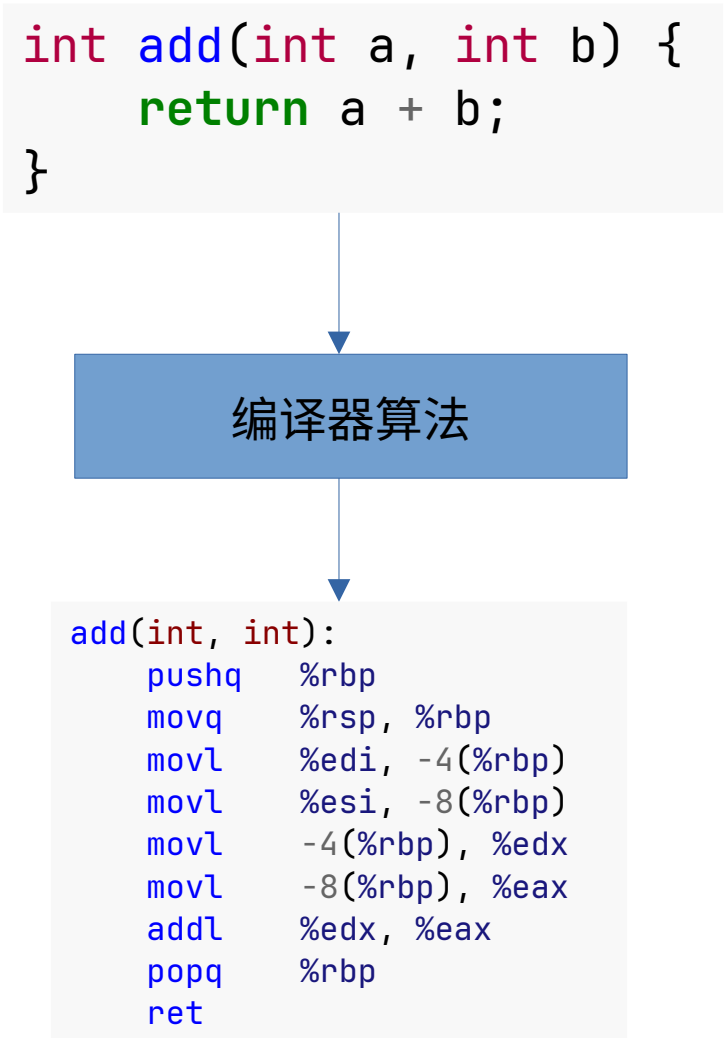
若对每个输入实例算法都以正确的输出停机，则称该算法是正确的，并称正确的算法解决了给定的计算问题。不正确的算法对某些输入实例可能根本不停机，也可能以不正确的回答停机。与人们期望的相反，不正确的算法只要其错误率可控有时可能是有用的。例如布隆过滤器。但是通常我们只关心正确的算法。

算法可以用自然语言说明，也可以说明成计算机程序，甚至说明成硬件设计。唯一的要求是这个说明必须精确描述所要遵循的计算过程。

算法解决哪种问题

排序绝不是已开发算法的唯一计算问题（当看到本教程的厚度时，你可能觉得算法也同样多）。算法的实际应用无处不在，包括以下例子：

- 浏览器：当浏览器的窗口大小变化时，文本需要自适应地跟着变化。那么文本在哪个位置换行？这就是断行算法。
- 编译器：例如 C 语言编译器是将 C 程序编译成汇编语言。如下图所示：



编译器中几乎用到了我们将要学习的所有数据结构与算法：树形结构，有限状态机，图论算法等等。例如 JVM 的垃圾收集算法（标记-清除算法，Mark And Sweep）实际上就是图论中的深度优先搜索算法。

- GPT 模型：GPT 模型也可以认为是一种算法（Transformer 模型），模型的输入是我们的提示（Prompt），输出是回答（Response）。模型中用到了很多深度学习算法。例如反向传播算法。如果我们学习了本课程，在学习神经网络算法时，应该可以知道神经网络算法的核心算法：反向传播算法，其实就是：动态规划 + 链式求导。
- 密码学：RSA 算法。非对称加密。《算法导论》简称 “CLRS”，由四位作者的名字的首字母组成。其中的 “R” 就是 RSA 的 R。
- 数据库：MySQL 数据库的索引使用 **B+树** 数据结构。数据库执行计划的优化使用了很多图论算法。
- TCP/IP 协议：三次握手的建模使用了有限状态机。流量控制和乱序重排使用了滑动窗口算法。
- 分布式系统
    - Paxos 算法使用在 Zookeeper 以及 etcd 中。etcd 是 K8S 的核心组件。用来做分布式一致性。
    - 比特币为了实现存在恶意节点情况下的分布式一致性，使用了“工作量证明”算法。

所以我们发现，几乎所有计算机的领域都离不开算法，**一名优秀的工程师一定精通他所在领域里常用的算法**。所以想要成为一名优秀的工程师，无论是前端、后端还是大数据工程师，都必须认真学习数据结构与算法。数据结构与算法是所有计算机领域的基础。这个学好了，才谈得上深入底层学习别的东西。

数据结构

本教程也包含了常见的数据结构的讲解。数据结构是一种存储和组织数据的方式,旨在便于访问和修改。没有一种单一的数据结构对所有用途均有效，所以重要的是知道几种数据结构的优势和局限。

算法设计技术

虽然可以把本书当做一本有关算法的“菜谱”来使用，但是也许在某一天你会遇到一个问题，一时无法很快找到一个已有的算法来解决它。本教程将教你一些算法设计与分析的技术，以便你能自行设计算法、证明其正确性和理解其效率。例如：分治策略，动态规划和贪心算法等等。

难以求解的问题

本教程大部分讨论有效算法。我们关于效率的一般量度是速度，即一个算法花多长时间产生结果。然而有一些问题，目前还不知道有效的解法。后面我们会研究这些问题的一个有趣的子集，其中的问题被称为 NP 完全的。

为什么 NP 完全问题有趣呢？第一，虽然迄今为止不曾找到对一个 NP 完全问题的有效算法，但是也没有人能证明 NP 完全问题确实不存在有效算法。换句话说，对于 NP 完全问题，是否存在有效算法是未知的。第二，NP 完全问题集具有一个非凡的性质：如果任何一个 NP 完全问题存在有效算法，那么所有 NP 完全问题都存在有效算法。 NP 完全问题之间的这种关系使得有效解的缺乏更加诱人。第三，有几个 NP 完全问题类似于（但又不完全同于）一些有着已知有效算法的问题。计算机科学家迷恋于如何通过对问题陈述的一个小小的改变来很大地改变其已知最佳算法的效率。

你应该了解 NP 完全问题，因为有些 NP 完全问题会时不时地在实际应用中冒出来。如果要求你找出某一 NP 完全问题的有效算法，那么你可能花费许多时间在毫无结果的探寻中。如果你能证明这个问题是 NP 完全的，那么你可以把时间花在开发一个有效的算法，该算法给出一个好的解，但不一定是最好的可能解。

作为一个具体的例子，考虑一家具有一个中心仓库的投递公司。每天在中心仓库为每辆投递车装货并发送出去，以将货物投递到几个地址。每天结束时每辆货车必须最终回到仓库，以便准备好为第二天装货。为了减少成本，公司希望选择投递站的一个序，按此序产生每辆货车行驶的最短总距离。这个问题就是著名的“旅行商问题”，并且它是 NP 完全的。它没有已知的有效算法。然而，在某些假设条件下，我们知道一些有效算法，它们给出一个离最小可能解不太远的总距离。后面的内容将讨论这样的“近似算法”。

并行性

我们或许可以指望处理器时钟速度能以某个持续的比率增加多年。然而物理的限制对不断提高的时钟速度给出了一个基本的路障：因为功率密度随时钟速度超线性地增加，一旦时钟速度变得足够快，芯片将有熔化的危险。所以，为了每秒执行更多计算，芯片被设计成包含不止一个而是几个处理”核”。我们可以把这些多核计算机比拟为在单一芯片上的几台顺序计算机；换句话说，它们是一类“并行计算机”。为了从多核计算机获得最佳的性能，设计算法时必须考虑并行性。后面的内容给出了充分利用多核的“多线程”算法的一个模型。从理论的角度来看，该模型具有一些优点，它形成了几个成功的计算机程序的基础，包括一个国际象棋博弈程序。

1.2 作为一种技术的算法

假设计算机是无限快的并且计算机存储器是免费的，你还有什么理由来研究算法吗？即使只是因为你还想证明你的解法会终止并以正确的答案终止，那么回答也是肯定的。

如果计算机无限快，那么用于求解某个问题的任何正确的方法都行。也许你希望你的实现在好的软件工程实践的范围内（例如，你的实现应该具有良好的设计与文档），但是你最常使用的是最容易实现的方法。

当然，计算机也许是快的，但它们不是无限快。存储器也许是廉价的，但不是免费的。所以计算时间是一种有限资源，存储器中的空间也一样。你应该明智地使用这些资源，在时间或空间方面有效的算法将帮助你这样使用资源。

效率

为求解相同问题而设计的不同算法在效率方面常常具有显著的差别。这些差别可能比由于硬件和软件造成的差别要重要得多。

作为一个例子，第 2 章将介绍两个用于排序的算法。第一个称为插入排序，为了排序 n 个项，该算法所花时间大致等于 $c_1 n^2$，其中 c_1 是一个不依赖于 n 的常数。也就是说，该算法所花时间大致与 n^2 成正比。第二个称为归并排序，为了排序 n 个项，该算法所花时间大致等于 $c_2 n \log n$，其中 $\log n$ 的底是 2（在计算机科学中，不专门说明，对数的底一般是 2），且 c_2 是另一个不依赖于 n 的常数。与归并排序相比，插入排序通常具有一个较小的常数因子，所以 $c_1 < c_2$。我们将看到就运行时间来说，常数因子可能远没有对输入规模 n 的依赖性重要。把插入排序的运行时间写成 $c_1 n \cdot n$ 并把归并排序的运行时间写成 $c_2 n \cdot \log n$。这时就运行时间来说，插入排序有一个因子 n 的地方归并排序有一个因子 $\log n$，后者

要小得多。（例如，当 $n=1000$ 时，$\log n$ 大致为 10，当 n 等于 100 万时，$\log n$ 大致仅为 20 。）虽然对于小的输入规模，插入排序通常比归并排序要快，但是一旦输入规模 n 变得足够大，归并排序 log n 对 n 的优点将足以补偿常数因子的差别。不管 c_1 比 c_2 小多少，总会存在一个交叉点，超出这个点，归并排序更快。

作为一个具体的例子，我们让运行插入排序的一台较快的计算机（计算机 A） 与运行归并排序的一台较慢的计算机（计算机 B）竞争。每台计算机必须排序一个具有 1000 万个数的数组。（虽然 1000 万个数似乎很多，但是，如果这些数是 8 字节的整数，那么输入将占用大致 80MB，即使一台便宜的便携式计算机的存储器也能多次装入这么多数。）假设计算机 A 每秒执行百亿条指令（快于写本书时的任何单台串行计算机），而计算机 B 每秒仅执行 1 000 万条指令，结果计算机 A 就纯计算能力来说比计算机 B 快 1 000 倍。为使差别更具戏剧性，假设世上最巧妙的程序员为计算机 A 用机器语言编码插入排序，并且为了排序 n 个数，结果代码需要 2n2 条指令。进一步假设仅由一位水平一般的程序员使用某种带有一个低效编译器的高级语言来实现归并排序，结果代码需要 50n lgn 条指令。为了排序 1 000 万个数，计算机 A 需要