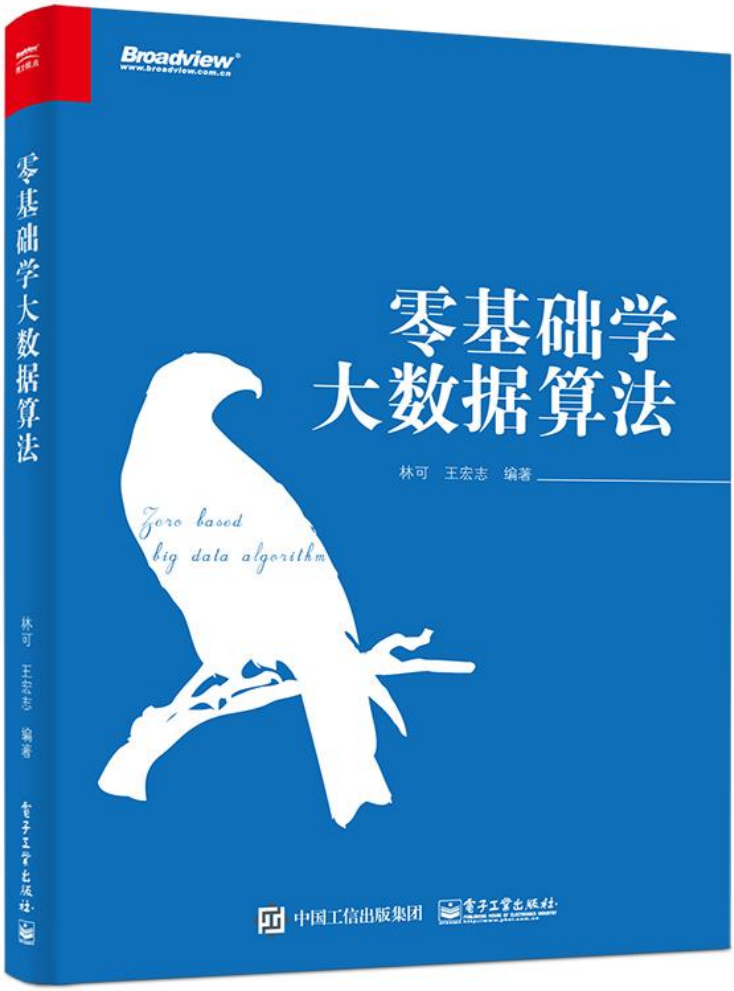

Broadview
www.broadview.com.cn
零基础学大数据算法
林可 王宏志 编著
Zero based
big data algorithm
中国工信出版集团
电子工业出版社

了解更多

前　言

这是一个互联网的时代，也是一个大数据的时代。经常有朋友问起：什么是大数据？大数据是做什么用的？我们为什么要研究大数据？应该怎么研究大数据？在寻找这些问题的答案时，许多朋友找到的内容常常是专业的概念、复杂的公式和难懂的“算法”，这让他们望而却步。很多计算机专业的新生或低年级学生在听到大数据的概念后对其非常好奇，却因没有足够扎实的专业基础知识而无法认识和理解大数据问题，更无法对大数据问题给出很好的解决办法。于是，笔者决定编写一本新生乃至非专业人士也能读懂的大数据算法教程。

本书以一个计算机专业新生小可的口吻，将他内心对大数据的好奇一一询问学识渊博的 Mr. 王。虽然书中的他不懂数据结构，也不懂经典算法的设计与分析，却在 Mr. 王的耐心教导下一一突破了大数据背景下的亚线性算法、磁盘算法、并行算法、众包算法，了解了数据挖掘和推荐算法的基本思想，更是在 Mr. 王的指导下完成了各种大数据算法的实现。在所有大数据算法的讲授中，本书无处不渗透着对各种经典算法的回顾，学过的读者可以进行充分的复习，没有学过的读者更是可以借此机会提前掌握各种经典数据结构和经典算法，使得其在今后的学习中事半功倍。这些贯穿全书的前置知识，也使得新生甚至是非专业人士能够通过本书读懂大数据，读懂大数据算法。相信非专业人士也能通过大数据算法的思想，重新认识大数据，并获得一些启迪。

本书前半部分主要以理论知识为主，文中涉及的伪代码、算法等均以简单易懂的自然语言进行了步骤描述和解释，同时给出了小规模运行的例子，使得读者可以轻松地理解。同时，笔者也深知理论与实践相结合的重要性。后半部分包含一些让读者进行实践的实验和程序。这需要具有一些基础程序设计能力，如果读者觉得不能很好地理解它们也不要心急，可以待学会这些语言后，再来尝试。即使并不完全理解这几种语言的语法，也可以按照书中详尽的步骤进行实验，体会成果出现在屏幕上的喜悦，其实尝试之后就会发现，阅读它们并不困难。

虽然本书气氛轻松、语言活泼，但讲授的知识和内容却是非常“专业”的，“算法设计与分析”是计算机学科的核心主题之一，计算机科学中所有问题的解决，都离不开算法设计与分析。本书虽讲解大数据，但无处不紧扣算法设计与分析这一要点，这也让本书带上了浓厚的计算机学科的味道，让读者能够在学习和认识大数据的过程中，学会算法设计与分析，为其他领域知识的学习打下基础。

本书亦算是大数据算法领域的“敲门砖”，本书可以引导读者形成设计大数据算法的思维。在阅读本书之后，读者可以带着设计大数据算法的基本思想去阅读更加深入的专著或论文，进一步的阅读必对大数据算法的学习大有裨益。

本书成书时间仓促，笔者水平亦有限，书中内容、表述、推理等方面的各种不当之处在所难免，敬请各位读者在阅读过程中不吝提出宝贵意见。

目　录

第 1 篇　背景篇

第 2 篇　理论篇

第 3 篇　应用篇

第 4 篇　实践篇

第 1 章　何谓大数据

1.1　身边的大数据

小可：王老师，那什么是大数据呢？

Mr. 王：你还真是一下就问了个很复杂的问题。其实大数据是一个很模糊的概念，很多学者和学术组织都对其提出过自己的定义，但是至今还没有公认的定义。我们先不谈确切的定义，先来举几个例子说明吧。你平常用社交网络吗？

小可：嗯，是的。

Mr. 王：你有很多好友吧？他们是不是每天都会发很多的状态和消息？

小可：是的，甚至有很多新闻我都是首先通过社交网络知道的。社交网络传递信息的速度真的很快，朋友们每天发布的状态我都看不完，而且不仅有原创的内容，还有很多来自他们好友的转载内容。

Mr. 王：其实社交网络上的这些信息就是一种典型的大数据。

小可惊讶地说：原来这就已经是大数据了？我一直以为大数据都在实验室里面呢。

Mr. 王：此言差矣，其实大数据就在我们身边。我们常用的社交网络上就有着非常巨大的信息量，虽然一个人发布的状态非常有限，但由于使用的人数众多，加之转载和评论，巨大的数据规模就使得社交网络信息无法在短时间内由人工或者由少量的几台计算机存储和管理。站在社交网络之外看待它，就会发现里面有很多且杂乱无章的信息和内容，同时其规模非常大。这就是大数据的一个典型例子。

小可恍然大悟地说道：哦，原来这就是大数据啊，那其实我每天都在接触大数据。

Mr. 王笑道：的确，大数据就在我们每个人的身边，随着信息时代的到来，我们每个人每天接触到的数据量都是非常大的。但你在查看这些消息的时候，有没有看到除字面内容以外的东西呢？

小可想了一下，说：好像没有什么，我关注的只是消息本身。

Mr. 王：我们研究大数据不只是能知道它的数据量很大，或者说仅仅研究如何把它们存储起来，我们还要发掘在大数据中隐藏的知识和有价值的信息。

小可：哦？大数据中隐藏着知识？

Mr. 王：是的，从表面上看，大数据可能只是一些简单的文本、杂乱的符号或者是一些数字的序列或者集合，但是从这些文本或者数字的背后，我们可以发掘其作为一个群体所具有的一些性质，从而发现一些对我们有意义、有价值的信息，所以我们才要研究大数据。

小可：大数据不是很大很大吗？那么我们研究它不就会变得很困难吗？

Mr. 王：不错，大数据的量很大很大，我们单单是把其中的信息逐个地访问一遍都很困难，所以发掘其中的知识就更加困难了，这就是研究大数据要解决的重要问题，也就需要我们这些研究大数据的人、热爱大数据的人加倍地努力了。

小可思考片刻后，说：那在超市里面，每年都会有很多人去买东西，他们的购物单上又会包含着很多内容，对超市来说，这些购物的记录就是“大数据”吧？而通过分析这些购物单，发现顾客更喜欢买哪些商品，这算不算一种通过大数据分析出的知识呢？

Mr. 王：很聪明嘛，你举了一个很好的例子。商业数据也是大数据的一个重要体现，超市购物的明细记录、公司运营的详细账目这些数据量都是很大的，处理起来非常费时费力，而其中又包含着有价值的信息，通过这些信息不仅可以分析出本年度公司的运营情况，同时可以指导下一年度公司的营销战略，这些数据对公司来说可谓是价值连城。

小可：那么大数据在别的方面又有哪些体现呢？

Mr. 王：你应该对生物遗传有所了解吧。

小可点点头道：是的，人体通过 DNA 携带遗传信息。

Mr. 王：在医疗和生物计算领域中，每次对 DNA 序列的分析都会产生大量的数据，这个数据量已经不是用 GB 可以衡量的了，甚至要达到 PB 级别或者更大。而这么大的数据，不仅计算机的内存装不下，而且一般计算机的硬盘都已经存不下了。即使是扫描一遍，在上面发现一个小序列都需要一些时间，在这些数据上面做分析将是一件更困难的事情。这也是一种大数据。

不仅在生物学中如此，而且在很多科学仪器的使用过程中也都会产生大量的数据，比如天文观测、显微观测，现在逐渐应用的传感器和传感器网络在使用过程中都会记录下大量的数据。这些仪器不停地记录下的数据，都涉及如何存储、如何分析研究的问题，这些都是大数据。

生活中的大数据

小可：嗯。

Mr. 王：那我们就给大数据下个定义吧。

定义 1：所涉及的数据量规模巨大到无法通过人工，在合理时间内达到截取、管理、处理并整理成为人类所能解读的信息。（Dan Kusnetzky, What is "Big Data"？）

定义 2：不用随机分析法（抽样调查）这样的捷径，而采用所有数据的方法。（维克托·迈尔 - 舍恩伯格、肯尼斯·库克耶，"大数据时代"）

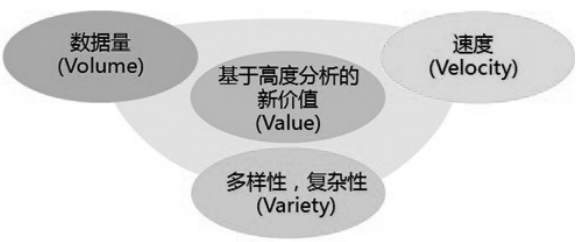
定义 3："大数据"是需要新处理模式才能具有更强的决策力、洞察发现力和流程优化能力的海量、高增长率和多样化的信息资产。（"大数据"（Big Data）研究机构 Gartner）

有了前面的那些例子，这些定义是不是相对好理解一些呢?

小可：嗯，我懂了。

1.2 大数据的特点和应用

Mr. 王：大数据具有较大的数据量，和一般的数据相比，其具有如下一些特点。



- 在数据量上，大数据是通过各种设备产生的海量数据，其数据规模极为庞大，远大于目前互联网上的信息流量，PB 级别将是大数据的常态。
- 在多样性上，大数据种类繁多，在编码方式、数据格式、应用特征等多个方面存在差异性，多信息源并发形成大量的异构数据。
- 在速度上，涉及感知、传输、决策、控制开放式循环的大数据，对数据实时处理有着极高的要求，通过传统数据库查询方式得到的“当前结果”很可能已经没有价值。
- 在价值上，数据持续到达，并且只有在特定时间和空间中才有意义。

Mr. 王：我们分析大数据、研究大数据，是希望能够利用它们获得我们需要的知识。我们可以利用大数据进行：

- 预测
- 推荐
- 商业情报分析
- 科学研究

等发现大数据中的价值，使用大数据、利用大数据的过程。由此可知，对大数据的研究还是非常重要而有意义的。

小可：有种大数据中有黄金的感觉啊。

Mr. 王：正是如此，从大数据中挖掘出来的价值，真是难以估量啊。今天时间不早了，你先回去吧，下节课咱们讨论一下关于算法的问题，要讨论大数据算法，必须先了解算法的相关知识。

小可：谢谢老师，那我下次再来。

第2章　何谓算法

2.1　算法的定义

小可：王老师您好，了解了什么是大数据，今天我来听听关于算法的内容。

Mr. 王：你好，想要学懂大数据算法，算法的基础知识是一定要了解的，你必须要知道如何设计和分析算法，我们才能谈及如何在大数据集合上研究算法。

小可：我经常会听到“算法”这个词，计算机专业的同学总会提到，那么到底什么是算法呢？

Mr. 王：至今为止，算法都没有一个准确的定义。每个计算机科学家和工程师都在设计算法、使用算法、分析算法、实现算法，但对于算法的定义依然是众说纷纭，很多书籍都曾经给出自己的定义，与其说那是一个定义，不如说都是对算法的一种诠释。的确，算法是一个很模糊、抽象的概念。

小可：那么我该如何搞懂什么是算法呢？

Mr. 王：在解释“算法”这个概念之前，我们首先来谈谈，一个计算机科学家是如何解决问题的。我先问问你，计算机是用来做什么的呢？

小可：计算、办公、游戏、影音娱乐，还有上网。

Mr. 王：不错，宽泛地谈起计算机的用途真是数不胜数。不过总结起来，其实计算机做的事情就一个——解决问题。

小可：解决问题？

Mr. 王：对，生活中有很多问题，其中有些问题人工解决起来很费时费力，于是我们发明了许多工具，在这个层面上，其实计算机也是一种工具，本质上它就是解决问题的一种工具。

比如：

- 升空卫星的轨道是怎样的?
- 从哈尔滨到深圳，走怎样的一条路线最短、最省路费?
- 模拟一次比赛的结果，分析到底谁的胜算更大?
- 我们希望知道，某个游戏或者博弈中是不是有必胜的策略?

小可：嗯，其中有些问题人工解决起来确实很费劲。那么计算机科学家又是如何解决这些问题的呢?

Mr. 王：首先，如果希望计算机能真正地解决一个实际问题，我们先要将现实世界中的事物转化为模型，这个模型可以被计算机理解和处理，它可以表示成数据和指令等。这个过程我们称之为**建立模型**。在此过程中，我们需要把一个实际问题抽象成计算机可以理解的语言，或者说计算机可以理解的问题，才可以用计算机求解。

小可：哦，这就是所谓的“建模”吧。

Mr. 王：其次，我们要知道这个问题是不是可计算的。计算机可以解决很多问题，也有很多问题解决不了。那么，由此诞生的，研究一个问题是不是计算机可计算的、可解的计算机科学分支叫作**可计算理论**。

小可：还有计算机解决不了的问题吗?

Mr. 王：当然，比如著名的“停机问题”。在这方面做出卓越贡献的科学家是非常著名的阿兰·图灵。图灵曾经提出过很多对计算机科学产生深远影响的理论，直到现在，我们使用的电

子计算机在模型上依然可以称之为“图灵机”。停机问题在很多资料中也称作“图灵停机问题”。图灵已经进行了证明，停机问题是不可计算的。

小可：那是不是说，我的计算机内存太小、CPU 太慢，有些特别大型的问题在我这里就“计算不出来”，这就是一个不可计算的问题呢？

Mr. 王：不，这是不对的，不可计算的问题并不是出于 CPU 速度和内存大小等资源的限制而无法在一定的时间内完成，而是不论给计算机多大的内存、给它多快的 CPU 都是无法求解的。如果你的计算机 CPU 太慢、内存太小，在一台内存更大、CPU 更快的机器上，还是能够求解的，那么这样的问题不是一个不可计算的问题。

不过这里有一个问题：某个问题虽然是可计算的，但是对于这个问题我们急着要的结果要算几年时间，那么这个问题恐怕也“相当于”解决不了，或者说交给计算机解决已经没有什么意义了。所以我们必须要知道的一件事情是，这个问题能不能用计算机在我们可以接受的时间或者空间界限内解决。研究某个问题被计算机解决的时空下界限的计算机科学分支称为**计算复杂性理论**。计算复杂性理论主要研究的是某个问题可以被计算机求解的时间和空间下界限。它研究给出的问题的时空下界限，是任何算法都无法突破的，研究比下界限更快的算法是没有意义的，当发现某一个算法已经足够快到可以和计算复杂性理论中得出的下界限是同一个级别时，我们就不必再去提升它的效率了。同时，研究这种时空下界限有另一方面的目的，就是可以用来评价我们现在设计出来的算法与这个问题可以被解决的极限时间空间界限还有多远的距离，很多时候我们设计出来的算法还不足以达到这个极限界限，但有了这个界限，可以给我们接下来的研究指明一个方向。

总结起来，可计算理论和计算复杂性理论都是一个研究“问题”的范畴。

研究过问题之后，我们要考虑的就是如何去解决问题。这也是计算机作为一种工具的重要属性。想要解决问题，就需要我们设计算法。

小可：那么到底什么是算法呢？

Mr. 王：这里我们还是举个生活化的例子吧。比如，我们现在想要煮一锅汤，这就是一个问题。根据生活经验，我们认为它是可解的（可计算分析），也是理论上可以在我们接受的时间范围内解决的（计算复杂性分析）。这时，需要我们设计一个解决它的方法或者说一系列步骤。

小可：我想想，煮汤我们可以想到的步骤就是洗菜、切菜、烧水、煮汤、出锅。哈哈。

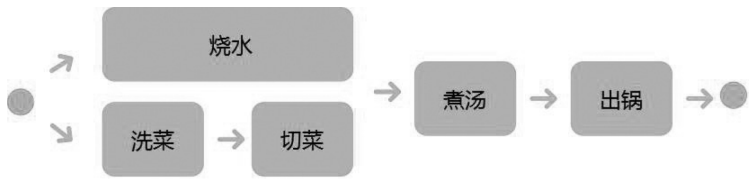
煮汤的“算法”

Mr. 王：很好，你已经在设计一个算法了。

小可吃惊地说：啊？！这就是一个算法了？

Mr. 王：从广义上讲，这种解决一个问题的一系列准确完整的步骤，就是算法。

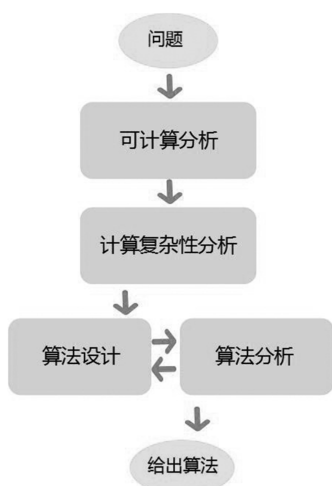
小可：哦，原来这已经是一个算法了？等等，王老师，我觉得我设计的这个步骤有些缺陷，如果按照这个步骤，在烧水的过程中，人一直是闲着的。如果先把水烧上，再去洗菜、切菜，是不是能更快地把汤煮完呢？也就是说，一个更好的步骤是，烧水、洗菜、切菜、煮汤、出锅。



煮汤的“算法”改进版

Mr. 王：不错嘛，你刚刚很好地分析了自己设计的算法。你注意到了一个算法具有的缺陷，并且分析它、改进它，提出了更好的新算法。但是计算机科学中的算法分析要比这个略复杂些，一会儿我会给你讲解，为了理解大数据算法，必须要了解如何分析算法。

在计算机科学中，研究算法的设计和评价算法“好坏”的分支，称为**算法设计与分析理论**。它研究如何去设计解决问题的算法，同时给出一个对算法在计算机中执行的时间和空间效率，评价这个算法是不是足够快、占用的空间足够小。到目前为止，高速的 CPU 和高速大容量的寄存器、缓存和内存依然是很昂贵的计算资源。另外，CPU 的运算速度和内存容量相对目前的大数据来说依然是不够的。所以设计高效率的算法，一方面是为了节约时间；另一方面也是为了节省金钱。从另一个角度讲，如果计算机的速度非常快、内存非常大，而且程度可以逼近无穷的话，恐怕我们就不再需要对算法进行分析和改进了，只要结果正确就可以了。不过从目前的数据量、研究的问题以及计算机的运算存储能力来看，还是有很大差距的。



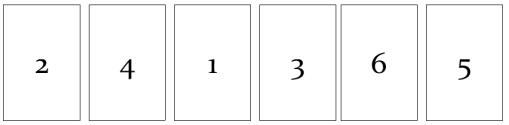
计算机科学解决问题的办法

小可：嗯，不仅仅是设计算法，看来分析算法也是非常有必要的，设计一个好的算法可以事半功倍。

Mr. 王：算法设计与分析这一部分也是计算机科学解决问题最重要的部分，是计算机科学的核心所在。在科学的发展史上，也正是因为算法的出现，才使得计算机科学得以独立成为一门学科。在计算机科学领域，算法有着举足轻重的地位。

小可：我明白了，计算复杂性理论研究的是问题能解决的时空下界限，研究的是“问题”，而算法分析研究的是某一个方法的时间界限，研究的是“算法”。

Mr. 王从抽屉里拿出一组卡片，打乱顺序放在桌上，说：完全正确。为了让你了解计算机中的算法，我们举个机器能够实际运行的算法例子。我在这里举一个比较简单的问题。



排序

比如现在有一组数字，我们希望将它们从小到大排序。这是算法设计中一类很基础也是很重要的问题，叫作“排序”。当我们要设计一个算法时，首先要分析它的输入输出。如果将算法视作一个机器的话，我们要将所需要处理的数据当作“原料”放进机器，然后经过机器处理将“成品”从机器中取出来。放进机器里的“原料”就是算法的输入，而取出来的“成品”就是输出。

在这个问题中输入输出应该是什么呢？

小可：输入是一组数字，输出是由这组数字构成的从小到大的序列。

Mr. 王：嗯。算法取一个或一组值作为输入，并产生一个或一组值作为输出，算法是一个定义良好的计算过程，或者说是一系列的计算步骤，它可以将输入数据转化为输出结果。这就是算法在计算机中的定义。

小可：原来这就是算法啊。

Mr. 王：没错。好了，咱们回到排序的问题上，你想到什么方法了吗？

小可：这应该有很多方法吧，我可以大概看一看，动一动就可以排出来了。

Mr. 王：排序的方法确有很多，但是这个“大概看一看”，计算机可做不到，这违背了算法的确定性，定义不明确，计算机如何执行呢？你去想一种每一步都可以确定执行的方法。

小可一边思索着，一边摆弄着桌子上的卡片：嗯，可以这样，第一次，我找出整个集合里面最小的数，放在第一位；第二次，我找出第二小的数，放在第二位。依此类推，直到所有的数都被放进序列中。

Mr. 王：假设计算机每次只能比较两个数的大小，那么应如何发现一组数里的最小值呢？

小可想了想，看着桌子上的卡片：这的确是个问题……可以这样做，我们先假设第一个数就是最小值，然后用后面的数与之比较，一旦发现有比它小的数，那么这个数就可以作为新的假设最小值，直到扫描了整个序列。

Mr. 王：不错，这样算法的步骤就被有效地具体化了。我们每一轮都执行选取最小值这个工作，这样第 n 步将第 n 小的数放在了第 n 个位置上，当 n 等于集合的大小时，就成功排列了。这种实现排序的算法叫作“选择排序”。我们用伪代码描述就是下面的程序：

```
Selection-Sort(A,n)
begin
    i ← 1;
    while i<n
    do
        min_position ← Find the minimal from A[i] to A[n];
        swap (A,i, min_position);
        i ← i + 1;
    end while
end
```

小可：伪代码是什么？

Mr. 王：伪代码是用来描述算法的一种语言，不是真正的计算机程序，但算法被描述成伪代码后，程序员就可以很容易地翻译成任何一种计算机语言了，这一步骤叫作算法的**实现**。而在算法被计算机语言实现后，可以形成相应的**软件系统**。我们今后的讨论也将常常用到伪代码来描述算法。伪代码并没有唯一的标准，很多书籍上使用的伪代码都是不同的，有时甚至就是英语或者中文句子。但是多数伪代码都是非常容易读懂的，具有一点编程基础的人都能够将它们实现为真正的计算机程序。

Mr. 王：我们来读读这段伪代码，看看它具体是怎么做的。

在伪代码中，我们常用“←”来表示赋值，它相当于很多高级语言中的等号“=”，它的意思就是把右边的值赋给左边。

就像我们之前描述的那样，每一轮，我们处理的对象都是还没有被排序的部分，在伪代码中体现的就是不断增加的 i。第一轮，从第 1 个到第 n 个；第二轮，从第 2 个到第 n 个。这是由在大多数高级语言中都有的 while 语句实现的。

在每一轮中，我们都进行两个操作。假设现在执行的是第 i 轮，第一个操作是从未排序的部分中选出一个最小值；第二个操作是将这个值与第 i 个位置进行交换，也就做到了第 i 轮将第 i 小的数放到第 i 个位置上。

如果希望具体一点的话，则可以将求最小值的方法也写成伪代码：

```
findMin(A,start,end)
begin
    i ← start
    min_pos ← i
```

```
        while i <= end
        do
            if A[i] < A[min_pos] then
            min_pos ← i
            end if
        end while
        return min_pos
    end
```

我们首先假设第一个值是最小值，然后扫描整个数组，一旦发现有比它还小的值，那么这个值就假设为最小值。这里始终更新的不是最小值，而是最小值的所在位置，然后通过这个位置来访问最小值。如果访问最小值的函数希望返回最小值的话，那么只需要稍作修改即可，这个就留给你回去修改了。

需要注意的一点是，我这里使用的伪代码中的数组下标是从 1 开始的。而像 C 语言这样的很多高级语言都是从 0 开始的，不过相信聪明的你一定能够在实现它的时候注意到这个问题并进行相应的调整。

swap 这个函数非常简单，相信有一点编程基础的人都可以实现。你一定没问题吧?

小可：没问题，我觉得像这样写就可以了：

```
swap (A,i,j)
begin
    temp ← A[i]
    A[i] ← A [j]
    A[j] ← temp
end
```


Mr. 王：至此，我们就完成了一个非常简单的算法——选择排序的设计。

小可：哦，那么我设计的这个算法怎么样呢?

Mr. 王：其实，这并不是一个很好的算法，它的时间复杂度是 $O(n^2)$，而计算复杂性理论已经成功地证明了，基于比较的排序方法，最低的时间复杂度是 $O(n\log n)$。

小可：听不懂了。

Mr. 王：嗯，下一部分课我们就来讲讲具体如何评价一个算法。不论是大数据算法还是经典算法，算法的分析都是非常重要的，评价一个算法有助于考虑是否在某个问题的求解、工程的实现、系统的设计上使用该算法，同时也非常有助于通过改进而派生出新的算法。

2.2 算法的分析

小可：嗯，我觉得评价一个算法的最基本方式就是看它运行得快不快。

Mr. 王：嗯，这是重要的考量标准之一。研究算法运行得快不快的指标叫作**时间复杂度**。

而在评价算法的同时还要考察其对内存空间的占用情况，也就是**空间复杂度**。这两个指标对于评价算法来说是最重要的。

小可：时间和空间都是计算机的资源，好的算法应该较少地占用资源而得出结果。

Mr. 王：对。我们先从时间复杂度开始探讨，空间复杂度与之类似，只不过是面向内存空间的。

小可：那时间复杂度是不是就是指一个算法运行的时间长短呢？

Mr. 王：不，这是一个常见的误解，算法的时间复杂度并不是指一个算法实际运行的时间。举个简单的例子，要访问一个集合中的每个数据，这在计算机科学中称为**遍历**。可以考虑一下，对一个只有 10 个数据的集合，和对有 10000 个同类型数据的集合使用同一种算法进行遍历，时间显然是不一样的。一个坏的算法在小的数据集合上，很可能比一个好的算法在大的数据集合上运行得要快。另外，现在计算机的发展速度很快，将同样的算法和同样的数据在不同的计算机上运行，得出结果的速度往往也不同。所以从这个角度来看，只用得出结果的时间这一指标来衡量，是不够准确和恰当的。

小可：对，算法的执行时间还跟数据量大小有关，也跟所使用的计算系统有关。

Mr. 王：的确，所以我们必须要考虑输入的**数据规模**。在算法分析中，这个数据规模常常用 n 来表示。对于不同的问题，n 的单位也是不同的。在研究刚才的排序时，n 代表的是有 n 个数；在研究数据库的查询时，n 就表示有 n 条记录等。我们可以将关于数量级 n 的运行时间复杂度记为 $T(n)$。

我们就用选择排序法来做个例子吧。这个算法可以分为两个部分：第一，我们要找出在未排序部分里的最小值并放在该放的位置上；第二，持续执行第一部分，直到所有的数据都已经排好序。假设进行比较需要消耗一个常数 c 的时间，中间更新假设最小值的时间和进行交换的时间我们暂时不考虑，那么进行一次扫描需要多久？

小可：那就是 n 和 c 的乘积 cn。

Mr. 王：很好，但是每一次进行选择排序时，集合的大小都会缩小 1，假设每次都发生了交换，你说说看，整个算法的复杂性可以如何衡量？

小可：这样的扫描需要进行 $n-1$ 次，因为只剩下一个数时就不用比了。而每次进行比较的元素次数分别是 $1, 2, 3,\cdots, n-1$。这是一个等差数列。结合前面的式子，可以求得为 $T(n)=(n-1+1)\times(n-1)/2\times c=cn(n-1)/2$。

Mr. 王：于是，这个算法的复杂度为 $T(n)=O(n^2)$。

小可：这要怎么解释呢？

Mr. 王：假设 n 是一个很大的数，比如 10^{10}。那么常数 c 和 n 相比如何呢？

小可：常数 c 已经小到可以忽略不计了。

Mr. 王：同理，n 和 n^2 相比呢？

小可：那 n 也可以小到忽略不计了。

Mr. 王：在进行时间复杂度分析时，我们只保留多项式中的最高阶项。因为相比最高阶项而言，低阶项可以被忽略。同时，忽略其中的所有常数项系数。因为算法复杂度分析关注的是其数量级，而非具体的数值，当 n 是一个非常大的数时，n 的幂值和 c 相比已经非常悬殊了，c 的值小到对 n 的幂值不会产生太大的影响。另外，在计算复杂性理论中，有一个与这个问题相关的图灵机线性加速定理，这个定理证明了 cn^a 和 n^a 在复杂度分析上没有区别，感兴趣的话可以去查阅一下相关的资料。不管从哪个角度来看，都忽略在时间复杂度中所有的常数项系数。

回到我们的例子中，我们估计选择排序的运行时间是 $T(n)=cn(n-1)/2$，转化成多项式的形式就是 $T(n)=\frac{c}{2}\cdot n^2+\frac{c}{2}\cdot n$。根据前面的约定，忽略多项式中的低阶项，只保留最高阶项，就是 $\frac{c}{2}\cdot n^2$；还要忽略常数项系数，就是 n^2，所以 $T(n)$ 的数量级就是 $O(n^2)$。这里如果考虑对数组中元素进行交换时间的话，不难发现，当元素是顺序时交换次数最少，为 0 次；当元素是逆序时交换次数最多，为 $n-1$ 次；平均情况是一个关于 n 的线性函数，是 n^2 的低阶项，可以被忽略，没有影响到我们的分析结果。

小可：那么前面的大 O 表示什么呢?

Mr. 王：嗯，这里需要说明一下。很多时候当 n 不够大时，时间多项式中的低阶部分确实没有高阶部分大。比如对于常数较大的 n^2+c，当 n 比较小的时候，c 可能会比 n^2 还大，这就不符合 c 和关于 n 高阶项相比小到可以忽略这个要求。而我们在研究算法的复杂性时，并不关心 n 比较小的时候算法的运行时间，因为这时一些处理时间为常数项或者低阶项的操作对算法运行时间的影响是非常大的，此时的运行时间不足以真正地代表一个算法的性能。所以引入了一系列记号对复杂度进行定义，要对 n “较大”这个条件进行限制。

$O(g(n))$ 表示这样的一系列函数 $f(n)$，存在正常数 c 和 n_0，对于所有的 n 大于 n_0，有 $0 \leqslant f(n) \leqslant c(g(n))$。大 O 记号的含义是 $f(n)$ 比 $g(n)$ 低阶。换句话说，$g(n)$ 表示的是 $f(n)$ 的上界。

n_0 的存在保障了我们研究的范围是 n 足够大时，它使得高阶项可以充分地大于低阶项。

小可：原来 $T(n)=O(g(n))$ 就表示当 n 足够大时，$T(n)$ 要比 $g(n)$ 小啊。

Mr. 王：通俗地讲是这样的。如果一个算法的运行时间 $T(n)=O(g(n))$ 的话，则说明其运行时间的渐进上界是 $g(n)$。也就是说，对于足够大的输入规模 n，这个算法的运行时间不会超过 $g(n)$。举个例子，选择排序的时间复杂度为 $O(n^2)$，如果输入规模是 10^{10}，那么它的运行时间应该在 10^{20} 这个数量级。

小可：这样看来，选择排序的运行时间应该会很久吧?

Mr. 王：是的，$O(n^2)$ 这个数量级实际是比较大的，这说明选择排序并不是一个很好的排序算法。

另外还要注意一点，我们在实际做算法分析时，如果使用大 O 记号来表示一个算法的复杂度，所确定的 $g(n)$ 一定要足够紧确，这样才能最好地代表一个算法的复杂性如何；因为我们去找一个很大很大的 $g(n)$ 也能满足 $T(n)=O(g(n))$，不过这样对于分析这个算法来说，就没什么

意义了。

小可：比如一个算法的运行时间是 $T(n)=2n+3$，那么它就是一个 $O(n)$ 的算法，就定义而言，$T(n)=O(n^2)$ 也是成立的。但是后者就不够紧确，这种不够紧确的时间界限不能真正地反映一个算法运行的时间界限。

Mr. 王：不错，这种 $T(n)O(n)$ 的算法，也叫**线性算法**，说明它可以在与输入同规模的时间界限之内得出结论，与输入规模呈线性关系。如果一个算法的复杂度比线性算法还低，它就可以称为**亚线性算法**。比如 $O(\log n)$、$O(\log\log n)$ 以及 $O(1)$。$O(1)$ 也可以称作常数时间算法，我们注意到 $\log n$ 可以不写对数的底数，这同样也是对常数的一种忽略。这样忽略的原因其实非常简单，因为有对数换底公式，比如 $10\log_2 n=2\log_{10} n$，在复杂度分析时，忽略前面的常数项系数，所以 $O(\log_2 n)$ 与 $O(\log_{10} n)$ 同阶。在常用的写法中，我们会忽略对数的底数，在复杂度中的 log 与 ln 都是同阶的。另外，如果某个算法的时间复杂度 $T(n)$ 可以表示为一个多项式，那么这个算法可以叫作**多项式算法**。对于一个不太大的数据规模，或者说在经典的计算理论中，我们认为这是一个易解问题；如果找不到多项式算法，比如找到的算法都是 $O(2^n)$ 级别的，我们会认为这是一个难解问题。

小可：嗯，当 n 比较大时，2 的 n 次幂比关于 n 的多项式要大得太多了。如果一个 $O(2^n)$ 的算法真的有 10^{10} 这样的输入规模，那可真是天文数字了。可是现实中真的有这样的算法吗？

Mr. 王：其实这种需要 $O(2^n)$ 的算法，还真的广泛存在。比如我要枚举一个集合的所有子集，你看看它的复杂度有多大。

小可：如果一个集合有 n 个元素，那么它的子集就有 2^n 个！

Mr. 王：于是，子集的枚举就非常容易产生 $O(2^n)$ 这种高阶的复杂度。相比之下，比较高阶的复杂度还有 $O(n!)$，指数函数和阶乘的增长速度都是非常快的，不难看出，在一个比较小的 n 值下，指数函数和阶乘都可以达到一个非常高的数量级。也就是说，一个拥有如此高阶复杂度的算法在同样的数据规模下运行时间往往是非常长的，一般来讲，这样的算法不是好的算法。就目前的知识看来，如果一个问题需要超过多项式时间界限的算法来解决，我们一般认为这个问题是一个难解问题。

小可：那计算机科学有没有对易解和难解问题进行一个相对严格的界定呢？

Mr. 王：有的，这里既然提到了多项式算法和易解难解问题，那么我们就简单来谈一谈 NP 完全性的问题，这有助于对后面一些问题的理解。真正的 NP 完全性讨论和复杂度归约是比较复杂的主题，一般要到硕士生阶段才会接触，这里我们只简单谈谈。

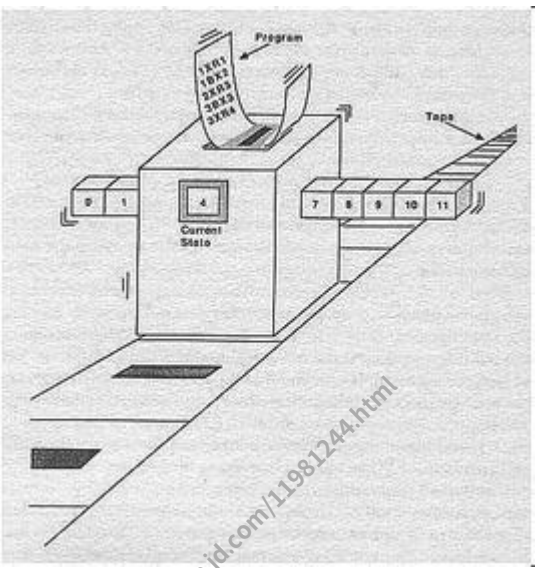
提到 NP 完全性，我们先要了解前面提到过的“图灵机”。

小可：我也很好奇，这个“图灵机”究竟是什么呢？

Mr. 王：想要理解图灵机，需要具有一定的自动机理论基础，最好先了解一下有穷自动机和下推自动机。这里我力图用浅显易懂的语言来解释它，所涉及的一些数学定义和形式化定义

就暂且不提了。

简单来说，图灵机是由一个读写头和一条两端无限延伸的纸带构成的。当然，也可以是多个读写头和多条纸带，科学家们已经证明，单带图灵机和多带图灵机是等价的。这里我们先谈谈单带图灵机。



其中读写头可以在纸带上左右移动，可以在纸带上写下符号，也可以将纸带上的符号擦去（或者说写下“空”这个符号），还可以读取纸带上的符号。在读写头的内部，有一些“状态”，在读取不同的符号或者移动时，读写头的内部可以进行状态转换，在下一个阶段时，读写头可以根据状态和读取到的符号决定如何移动、读、写。

小可：这还是有一点抽象。

Mr. 王：这样吧，我们来设计一个图灵机，来看看图灵机是如何工作的。

比如，想让图灵机解决 2+3 这个算术题，我们就去编一个加法计算器的图灵机程序。

对于图灵机来说，它的一切都是可以定义的。为了方便起见，我们不使用十进制或者二进制形式，而是用纸带上的 1 个“1”表示数字 1，用 2 个“1”表示数字 2，依此类推。

小可：这也就是说，只要在纸带上画几个“1”就可以表示几了。

Mr. 王：另外，我们还要用一个符号来表示加号。这里其实选用任何符号都可以，但在计算机中常常使用二进制形式，那么除数字 1 以外就会用到数字 0。为了方便起见，我们用数字 0 来表示加号。那么这个算式可以表示成什么？

小可：我们要计算的算式是 2+3，那就是“110111”。

Mr. 王：很好，我们将它写在纸带上。为了让纸带上的空白区域更加方便地被图灵机识别，

我们用字母 B 表示空白（blank）。

小可：由于纸带是无限长的，那么纸带上留下的就是……BBBB110111BBBB……

Mr. 王：光有纸带还是不够的，还要对图灵机进行编程。我们先不去看图灵机程序，而是想一想，这应该怎么做？

小可：如果“11”表示 2，“111”表示 3，那么 2+3 的结果 5 就是“11111”。

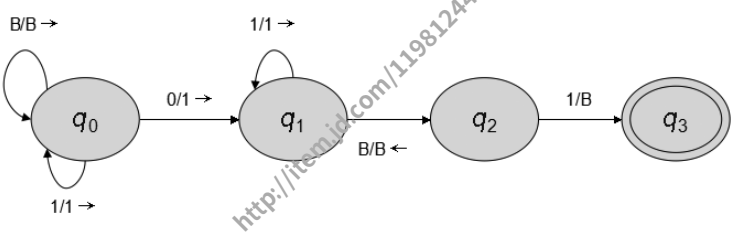
Mr. 王：很好，这个思路是对的，我们需要做的其实就是将纸带上的 110111 变成 11111。当然，一个加法计算器不能只对 2+3 可用，它也应该可以把 1011 变成 111，表示 1+2=3；可以把 11101111 变成 1111111，表示 3+4=7。

小可：其实这好像也不难，只要把中间的 0 去掉，然后把右边所有的 1 都往左挪就可以了。

Mr. 王：很好，其实图灵机就是这样做的。不过还可以做一个小小的改进，让图灵机执行起来更加方便，当图灵机遇到 0 时，我们把它变成 1，然后再把最后一个 1 抹掉，是不是也可以啊？

小可：嗯，这样和前面那种效果是一样的。

Mr. 王：好，我们用状态图来表示一下这个图灵机程序。



状态转换图

小可：这个好复杂啊。

Mr. 王：其实图灵机程序本质上就是读写头遇到了一个什么符号、如何转换状态、如何修改纸带上的符号、什么时候停机这样的一个组合，箭头连接符号用来表示状态的转换。上面的 X/Y →表示读到符号 X 时，将其改写为符号 Y，并且读写头右移（相应的“←”就表示左移）。

我们来看看它的执行结果。

刚开始，图灵机读到 B，根据转换图读写头向右移动：

BBBB110111BBBB　　状态：q_0

↑

遇到 1，读写头右移，又遇到 1，读写头继续右移：

BBBB110111BBBB　　状态：q_0

↑

读写头遇到 0 时，根据我们的设计，要将它改为 1，然后右移：

BBBB110111BBBB　　状态：q_0

↑

同时，状态变为 q_1：

BBBB111111BBBB　　状态：q_0->q_1

↑

右面的 1 和左面的一样，继续让读写头右移跳过它们即可，然后继续右移：

BBBB1111111BBBB　　状态：q_0

↑

继续右移，遇到了 B，这时我们知道右边已经没有 1 了，但是前面还有一个 1 我们没有将其抹掉，也就是此时纸带上留下的计算结果比正确的结果要多 1，根据程序继续执行，将其去掉。注意，此时状态要发生转换。想想看，如果没有一种状态记录读写头已经经过了扫描全部的 1 那个步骤的话，就会出现读写头返回去抹掉多余的 1 时遇到了 1，根据状态 q_1 上面的转换，又继续往右走，然后遇到 B 又返回的死循环情况。

BBBB1111111BBBB　　状态：q_1->q_2

↑

BBBB1111111BBBB　　状态：q_2

↑

小可：哦，状态在这里起到了让读写头知道现在应该执行什么操作的作用。

Mr. 王：是的，根据转换图，我们可以看出 q_2 在遇到 1 时将其改写为 B。

BBBB11111BBBBB　　状态：q_2->q_3

↑

图灵机停机：

BBBB11111BBBBB　　状态：q_3

↑

小可：结果是对的，纸带上留下了 5 个 1！

Mr. 王：还可以多用几个算式执行一下这个图灵机，来验证我们设计的程序还是不错的。这是一个很简单的图灵机例子，不过可以很有效地说明图灵机是如何定义和工作的。

小可：嗯，我懂了。可是您前面说现在的计算机在模型上都可以称作图灵机，这个要如何理解呢？

Mr. 王：你能思考这个问题是非常好的。其实现在电子计算机可以解决的所有问题，都可以用图灵机解决，就用 2+3 这个例子，我们一开始将“算式”写在纸带上，相当于“输入”；图灵机的执行过程相当于计算机对问题进行处理；留在纸带上的结果相当于“输出”；状态转换图，相当于计算机程序；纸带在执行过程中相当于内存，读写头一部分是 CPU，同时也是读

写内存的设备。

小可恍然大悟，说：这么一说，好像确实和计算机挺像的。

Mr. 王：好，既然你初步理解了什么是图灵机，我们就来说说什么是易解问题和难解问题。

前面我们提到过多项式时间。如果一个问题可以用确定状态图灵机（DTM）在多项式时间界限内解决的话，我们称之为 P 问题；如果一个问题可以用不确定状态图灵机（NTM）在多项式时间界限内解决的话，我们称之为 NP 问题。

在这里，所谓的确定状态就是说如果在每一种状态下接收到一个特定的输入，它都会执行固定的状态转换和动作，这就是确定状态图灵机；反之，如果在某一种或多种状态下，对于某一个输入，它可能产生多种不同的状态转换，这就是不确定状态图灵机。而计算机只能表达确定状态图灵机，无法表达不确定状态图灵机，所以我们用计算机去解决 NP 问题的时间界限往往是指数的。

有这样一类问题，首先它是 NP 问题，其次所有的 NP 问题都可以归约为它，我们称之为 NP 完全问题（NP-complete）。

小可：什么是归约呢？

Mr. 王：简单来说，如果找到了解决问题 A 的办法，那么问题 B 也就得到了解决，而且正可以用解决问题 A 的那种办法。那么我们说，B 可以归约为 A。从这里可以看出，B 可以归约为 A，说明 A 要比 B 难以解决，或者说 A 比 B 难。

小可：哦，原来是这样，那么 NP 完全问题就是 NP 问题中最难的那些问题了？

Mr. 王：你说得对。还有一类问题，所有的 NP 问题都可以归约为它，但我们还无法判定它是不是 NP 问题。我们将这类问题称为 NP 难问题（NP-hard）。

小可：也就是说，我们还确定不了不确定状态图灵机能不能在多项式时间界限内解决它，那说明它的难度有可能比 NP 完全问题更高吧。那是不是说，如果我们恰好证明了一个 NP 难问题是 NP 问题的话，那么它就是 NP 完全问题了？

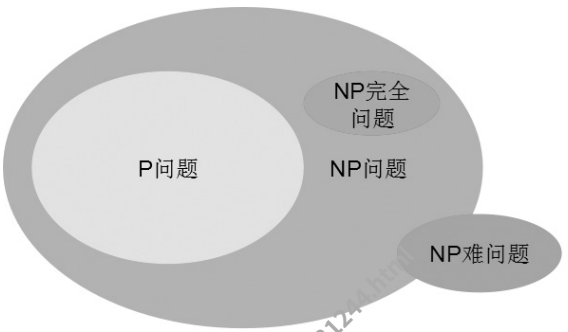
Mr. 王：是的，这说明你对 NP 问题的定义已经有了很好的了解。了解了这几类问题，我们不难发现有 $P \subseteq NP$，也就是说，P 问题都是 NP 问题。这是因为我们可以把确定状态图灵机看作不确定状态图灵机的一种特殊形式，只不过它的所有转换都是确定的，也就是说，所有的转换概率都是 1。

小可：嗯，的确是这样，P 问题都是 NP 问题。

Mr. 王：现在困扰计算机科学界最大的问题就是 P 是否等于 NP，这个问题被提出了多年，影响着很多问题的研究，至今未能得到解决。显然有 P 包含于 NP，但到目前为止 NP 是不是包含于 P 还不能确定。如果这个问题得到了证明，就说明 P=NP。在计算机科学界里，大量的疑团都指向 P 是否等于 NP。如果 P=NP，那么现在很多科学研究的结论将会被改写，很多新的结论也会被提出。但是证明这个问题是非常非常难的，当然，也有可能最后证明的结果是 P

不等于 NP。要证明 P=NP，就需要计算机科学界的学者们多多努力了。

好了，现在我们来给问题的难度排个序。首先有 P 包含于 NP，所以在难度上 P ⩽ NP。由于 NP 完全问题是 NP 问题中最难解决的，故 NP 完全问题会难于一般的 NP 问题，所以有 P ⩽ NP ⩽ NPcomplete。由于 NP-hard 和 NP-complete 同属的所有 NP 类都可以归约为它们的这种问题，而 NP-hard 还不能确定是不是 NP 问题，所以它应该更难一些，所以有 P ⩽ NP ⩽ NPcomplete ⩽ NPhard。我们一般认为 P 问题是易解问题，而 NP-complete 以上的就是难解问题。



P–NP 问题的关系

小可：嗯，我懂了。

Mr. 王：不过进入了大数据时代以后，易解和难解问题又相应地发生了一些变化，当数据规模并没有那么大的时候，多项式算法在求解很多问题时，算法的实际运行时间或许我们还可以接受，我们认为多项式算法还算是好的算法，能用多项式算法解决的问题还算是易解问题；但当数据量真的大到可以称之为“大数据”的时候，多项式算法的实际运行时间也会变得非常长，变得我们难以接受，这样多项式算法就已经不能满足我们对于很多大数据规模的问题求解。有时一个问题虽然是 P 问题，但是由于数据规模太大，也变得很难以解决，甚至当输入规模特别大的时候，在很多情况下就连线性算法也难以满足需求了。有些时候，我们就不得不去设计一些后面要讲的亚线性算法来匹配这些非常大的数据集合，以满足我们对它的速度要求。

小可：那有没有更快的算法？比如其运行时间与输入的数量级完全无关，如就是个常数项 c 呢？也就是说，这个算法不论输入 n 多大，它的运行时间都是一个特定的常数 t 呢？

Mr. 王：这样的时间界限记为 O(1)，我们称之为常数时间算法，这样的算法一般来说是最快的，因为它与输入规模完全无关，不论输入规模 n 多么大，我们都可以用一个与输入规模 n 无关的常数时间得出结论，相比于巨大的 n 来说，这个常数在数量级上已经微乎其微了。

小可：哦，这也体现了只关心数量级，而不关心具体数值的思想。

Mr. 王：另外，与大 O 记号类似，常用的记号还有 Θ，$\Theta(g(n))$ 表示函数 $f(n)$ 构成的集合，存在 n_0, c_1, c_2，当 $n \geqslant n_0$ 时，$0 \leqslant c_1 g(n) \leqslant f(n) \leqslant c_2 g(n)$。这样保证了当 n 足够大时，$f(n)$ 在一个

常数因子范围内与 $g(n)$ 是相等的，$g(n)$ 是 $f(n)$ 的一个渐进确界。比如 $T(n)=2n+3$，同样也符合 $\Theta(n)$。此时我们称 $f(n)$ 和 $g(n)$ 是同阶的函数。

相应的，还有 Ω 记号，Ω 记号表示函数 $f(n)$ 构成的集合，存在 n_0,c_1,c_2，当 $n\geqslant n_0$ 时，$0\leqslant cg(n)\leqslant f(n)$。换句话说，$g(n)$ 是 $f(n)$ 的低阶函数，或者说 $g(n)$ 是 $f(n)$ 的下界。

如果希望大小关系不包含等于，则还有 ω 和 o 两种记号。其中 $\omega(g(n))$ 表示存在 n_0,c_1,c_2，当 $n\geqslant n_0$ 时，$0\leqslant cg(n)<f(n)$；而 $o(g(n))$ 表示存在 n_0,c_1,c_2，当 $n\geqslant n_0$ 时，$0\leqslant f(n)<cg(n)$。它们与大 O 记号和 Ω 记号类似，只是在大小关系上不包含等于。

小可：嗯，听到这里，我理解了如何进行算法的分析和几种记号表示的含义了。

Mr. 王：另外，很多时候，算法的运行时间并不是稳定的，在算法分析的过程中，我们还要考虑算法运行的最好情况、最坏情况和平均情况。很多算法的最好情况非常好，但平均情况不够理想；而有的算法运行时间的最坏情况复杂度非常高，但平均情况却不错。

这里我们举个例子来说吧。对于一个存放了 100 个整数元素的数组，而且这些整数是无序的，我们要从中找到一个数字 50。这就存在最好情况和最坏情况。

小可：我明白了，如果使用逐个访问数组中每个元素的算法，最好情况是一比较发现第一个元素也就是 $A[0]$ 恰好是 50，那就不用再往后比较了，只进行一次比较就找到了 50；最坏情况就是逐个比较下去，发现最后一个元素是 50，或者最后一个元素也不是 50，则说明数组中不存在 50 这个元素。这时候要对每个元素都进行一次比较，这里有 100 个元素，就进行了 100 次比较。

Mr. 王：那算法的最好和最坏情况复杂度如何呢？

小可：如果有 n 个元素，在最好情况下，可以以常数时间找到我们所要找的元素，也就是 $O(1)$；在最坏情况下，我们要和最后一个元素进行比较才能得出结论，就是要进行和数据规模 n 相关的次数比较，也就是 $O(n)$。

Mr. 王：嗯，对于很多算法来说，用最好和最坏情况都不能够最佳地代表一个算法的复杂度。就拿这个例子来说，用最好情况来代表算法的复杂度显然是不恰当的，因为我们要找的元素往往不能总是第一个元素，如果这些元素是随机分布的，只有 $1/n$ 的概率让其出现在第一个位置上。同理，我们要找的元素恰好出现在最后一个位置上的概率也是 $1/n$，所以说它的运行时间就是访问 n 个元素的运行时间显然也不够公平。因此，很有必要给出一种复杂度，叫作平均复杂度，顾名思义，平均复杂度就是算法运行的平均情况的时间复杂度，既不是最好的，也不是最坏的，而是所有情况的平均值，或者说是在所有情况下复杂度的数学期望，很多时候，平均复杂度能最好地概括一个算法的运行情况。那么，从数组中逐个搜索一个元素的算法的平均情况如何呢？

小可：如果元素是随机分布的，元素出现在数组中每一个位置上的概率就是均等的，所以期望的运行时间应该是访问 $n/2$ 个元素的时间，也就是 $O(n/2)$。不过，这个值好像也是 $O(n)$ 啊。

Mr. 王：嗯，对于这个算法来说，平均情况和最坏情况的复杂度是同数量级的；但对于一些算法来说，最坏情况的复杂度却要比平均情况高一个数量级，用最坏情况去衡量它的复杂度就会将其评价为不够快的算法，这也不够公平。所以对于很多算法来说，我们要考虑它的最好、最坏和平均情况，以便更好地估计一个算法运行的真正时间。

2.3 基础数据结构——线性表

Mr. 王：为了以后的知识描述方便，这里简单介绍一下数据结构的概念。数据结构是一个广泛存在于计算机科学中的概念。曾经有一位计算机界的大师说："数据结构 + 算法 = 程序"。随着计算机科学的发展，虽然现在这个理论被认为不够全面，但也足以说明数据结构的重要性。

小可：这么说，数据结构拥有和算法同样重要的地位了！那么数据结构究竟是什么呢？

Mr. 王：在客观世界中，信息是多种多样的，有数字、颜色、图形、文本、声音等。但是这些信息"本身"并不能直接存储在计算机中，而是要以数据的形式存储在计算机中。比如要描述一个颜色，就可以用显示器输出的红色、绿色、蓝色的值（RGB 值）来表示；描述一张图片，就是用多个这样的 RGB 值或者标识颜色的数据标签来记录的。不论何种类型的信息都要以数据的形式在计算机中进行表示。

不过，这些数据不能杂乱无章地存放在计算机中；否则，不仅损失了信息之间应有的逻辑关系，而且查找起来效率也比较低。所以我们就要试图去发现这些数据之间的一些相互关系，并且利用这些关系将数据组织成一种逻辑结构。这种结构就是数据结构，它研究的就是如何依照这些数据之间的逻辑关系将数据表示在计算机中。对于一组数据，我们不仅要关注它们的逻辑结构，也就是数据之间在逻辑上的相互关系，在实际使用时，还要关注它们的存储结构，也就是说，这些数据在内存（磁盘）中是如何存储的。这都是数据结构研究的范畴。

小可：既然有人说，程序等于数据结构加算法，那么数据结构和算法之间又有怎么样的关系呢？

Mr. 王：这是个很好的问题，计算机中的算法都要对数据进行处理，而既然要处理数据，就要涉及数据如何存储在计算机中，如何能够高效地访问和处理数据，这就需要用到数据结构，所以说算法离不开数据结构。而当我们需要使用、访问、添加、删除、修改存储在数据结构之中的数据时，也要依照数据结构的特点进行操作，而增加、删除、修改、查找这些操作本身就是一种算法，所以说数据结构也离不开算法。

小可：原来是这样，数据结构和算法之间的联系还真是紧密啊。

Mr. 王：下面我们就来介绍一下最基础的数据结构——线性表。

小可：线性表是不是就是这些数据一个挨着一个地线性存储的结构呢？

Mr. 王：是的，线性表是由相同类型的数据按照一定的顺序排成的序列。这是一种非常常见的基础数据结构，使用非常广泛。

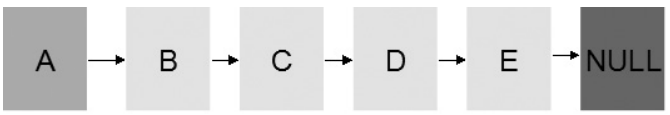
小可：具体的线性表都有哪些呢？

Mr. 王：常见的线性表有链表、数组线性表、栈和队列。

小可：数组我知道！ C 语言和 Java 中就有，将同一类型的数据存储在连续的内存空间里，通过首地址和数组下标可以访问到数组中的每一个元素。嗯，数组果然是符合线性表的定义的。

Mr. 王：是的，数组就是一种典型的线性表。我们学过了算法分析，可以考虑一下，数组只要知道首地址和下标就可以找到一个元素，这意味着能够以 O(1) 的复杂度访问其中的每一个元素，从这个角度来看，数组也是具有其优势的。但是增加和删除其中的元素却比较麻烦，因为如果要在数组中间增加一个元素，需要将后面的元素全都向后移动一个位置，平均情况需要 O(n) 次移动数据，是非常耗时的。前面我们也讨论过，如果数组内容是无序存储的，查找其中某个特定的值就会比较困难，要逐个地去访问比较，需要 O(n) 的时间复杂度。从空间角度来讲，如果我们确定知道元素的数量，那么用数组会比较节省空间；但是对于那些不知道有多少个元素、需要频繁添加和删除的元素集合来说，就需要在定义数组时让它稍大一些，这就造成了空间浪费。

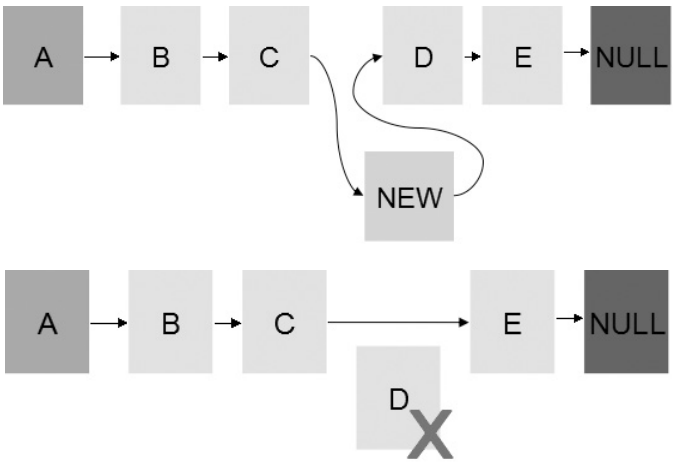
另外一种比较常见的线性结构是链表，它的构建是将元素一个一个地链接起来。访问链表时有一个首地址，我们可以通过这个首地址找到链表的第一个元素，这和数组是一样的；但和数组不一样的地方是，它的元素可能不连续地存储在空间中，而是每个元素的后面都带着一个地址，这个地址指向下一个元素的位置。我们想要逐个访问元素时，就要先找到第一个元素，通过第一个元素后面的地址找到第二个元素，然后再通过第二个元素后面的地址找到第三个元素，依此类推。



链表的例子

小可：那访问一个元素的时间复杂度就不是 O(1) 了，从平均情况来讲，访问一个元素的时间复杂度就是 O(n)。

Mr. 王：很好，但是链表也并不是没有优点。链表的优点是，在一个特定的位置添加元素，并不需要移动任何元素的位置，只需要将其前面元素后面的地址指向新元素，再将新元素后面的地址指向前面元素原来的后面地址就可以了。删除也是一样，只需要将待删除元素前面元素的指针指向删除后面的元素，然后释放掉被删除元素的内存即可。



链表的删除操作

小可：这个时间复杂度是 O(1)。

Mr. 王：这说明链表适合用于那些需要频繁进行添加、删除的元素集合。另外，从空间角度来讲，链表的长度相对比较灵活，它不像数组，即使数组中某一个位置是空的，没有元素，也要占用预先申请的内存空间；而链表每有一个元素，就占用一个元素和它后继节点地址的存储空间，但相对于元素数量变化幅度比较大的元素集合来说，链表的存储更加高效。

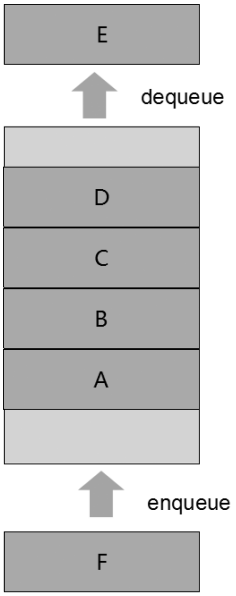
小可：嗯，果然是互补的两种结构，要根据实际情况选择适合的结构来存储。

Mr. 王：这里还有两种非常重要的线性结构要介绍，这两种结构非常简单，但却是很多算法的基础和组成部分，它们就是栈和队列。

小可：那什么是栈和队列呢?

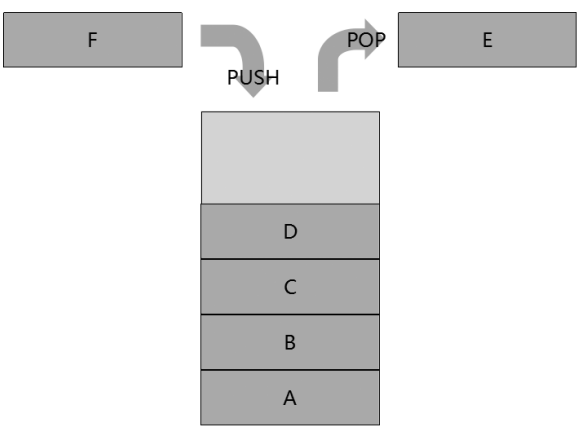
Mr. 王：栈和队列都是特殊的线性表，不论用数组还是链表来实现它们都是可以的。但是它们除了是线性表以外，还有自己的特殊性质。其实栈和队列这两种结构在生活中都可以见到。

先说队列吧。平时我们在各种公共场所排队时，如果将每一个人都看作一个元素的话，那么组成的就是一个队列。队列的特点就是，每一个新加入的元素，都要放在整个队列的最后面；而当一个元素要离开队列时，总是从队列的最前端离开。这就是说，先进入队列的元素先离开队列，后进入队列的元素后离开队列，总结起来就是 FIFO 策略（先进先出策略）。队列的常见操作有入队列（enqueue，让一个元素进入队列）、取队首（front，获得队列最前端元素的值）和出队列（dequeue，让队首的元素离开队列）等，这些算法都非常简单。



小可：嗯，队列还是很直观的。那什么是栈呢？

Mr. 王：栈的特点和队列正好相反。用生活中的例子来说，就像是一个书箱子，箱子只有一个朝上的开口，我们可以把书从上面放进箱子，但是想要取书时，也只能从箱子的顶端取走，不能直接从箱子的中间和底部取书。所以，先进入箱子的书就会最后离开箱子。栈只有一个访问端，元素的出入都只能通过这一个端，这样我们也可以分析出，先进入栈的元素会被放在栈的最底端，也就是说，它会最后离开栈。总结起来就是 LIFO 策略（后进先出策略）。对栈的常见操作有入栈（push，将一个新元素放在栈顶的位置）、出栈（pop，将栈顶的元素弹出栈，露出下一个元素作为栈顶）和取栈顶元素（top，获得栈顶元素的值）。



栈的示意图

小可：嗯，我懂了。可是栈在计算机中有什么用呢？

Mr. 王：看起来队列在生活中似乎更加常见一些，但在计算机中，栈的应用却是非常广泛的，我们可以通过一个非常经典的例子来介绍计算机是如何利用栈这个结构的。

2.4 递归——以阶乘为例

Mr. 王：我们介绍一个在计算机算法设计和程序设计中都非常常见的概念——递归。

小可：什么是递归呢？

Mr. 王：从程序设计的角度来说，递归就是一个函数，在它的定义中调用了它本身。从算法的角度来说，递归就是一个算法对于一个输入的求解需要对这个算法在更小输入上求解的情况。

小可：这个说法听起来有点复杂啊。

Mr. 王：我们举个例子来说明吧。你一定听说过有一个数学概念叫作阶乘。

小可：我知道，阶乘就是把一个正整数一直乘以它的值减 1，直到乘数为 1，比如 5!=5×4×3×2×1。推广到 n 的情况就是 $n!=n\times(n-1)\times(n-2)\times\cdots\times3\times2\times1$（特殊的，0!=1）。

Mr. 王：在计算机中求解一个数的阶乘，就可以利用递归。因为阶乘具有一个很有意思的特征，就是：$n!=n\times(n-1)!$。假如我们把阶乘定义为 $f(n)$ 的话（也就是 $f(n)=n!$），就有 $f(n)=n\times f(n-1)$。

小可：哦，从阶乘的定义来看，就是我们想知道 $f(n)$，就要知道 $f(n-1)$ 是多少，推广下去，想知道 $f(n-1)$ 就要知道 $f(n-2)$，一直到 $f(1)$。

Mr. 王：从递归的定义来看，求阶乘这个算法是不是正好符合求对于一个输入 n 的解，需要求取这个算法在一个更小的输入 $n-1$ 上的解，而对于 $n-1$ 的解需要知道去求取 $n-2$ 的解。

小可：嗯，从这个角度来看，这种求阶乘的算法确实是一个递归算法。

Mr. 王：如果要设计一个程序，我们也可以书写一个求递归的函数的伪代码：

```
int f(int n)
{
    if (n==1 || n==0)
    {
        return 1;
    }
    else
        return f(n-1);
}
```

小可：原来函数还可以这样定义啊。

Mr. 王：是的，C/C++ 语言是非常典型的支持递归的语言。一些早期的语言不支持递归，

不过现在很多程序设计语言都支持递归算法的设计。虽然所有的递归算法都可以设计成非递归的版本，比如阶乘，我们可以用一个循环来实现：

```
int f(int n)
{
    if (n==1 || n==0)
    {
        return 1;
    }
    else
    {
        for (int i=1;i <= n;i++)
        {
            int ans=1;
            ans *=i;
        }
        return ans;
    }
}
```

但是递归往往可以更加直观地表达算法的思路，这是非常有利于算法实现和程序设计的。不过有一点需要注意，设计不好的递归算法是非常容易出现无限循环的，在设计递归算法时，一定要设计递归的终点。比如在阶乘中，我们必须指定递归最终会达到的结果 $f(1)=1$；否则程序就会一直执行下去，直到内存溢出。

小可：嗯，我懂什么是递归了，但是这和栈有什么关系呢？在递归算法中也没有发现栈的存在啊？

Mr. 王：递归算法和栈的联系非常紧密，虽然在递归程序中我们并没有直接定义出一个栈，但程序运行的内部却会帮我们生成一个栈，这对于递归算法的运行是必要的。现在我们就以阶乘为例来剖析递归算法是如何运行的。

比如我们要求 5 的阶乘，也就是 $f(5)$。这时程序内的一个栈空间会开始工作，这个空间叫作函数调用栈。程序会将 $f(5)$ 压栈：

```
Call stack：[top= f(5)]
```

求解 $f(5)$ 时，程序发现需要知道 $f(4)$，就把 $f(4)$ 压栈：

```
Call stack：[top= f(4)][f(5)]
```

求解 $f(4)$ 时，程序发现需要知道 $f(3)$，就把 $f(3)$ 压栈：

```
Call stack：[top= f(3)][f(4)][f(5)]
```

依此类推，最后栈中会形成这样一种情况：

```
Call stack：[top= f(1)][f(2)][f(3)][f(4)][f(5)]
```

此时，程序发现 $f(1)$ 的值我们知道了，$f(1)=1$。所以我们得到了 $f(1)$ 的解，$f(1)$ 这个函数返回 1，已经解决的问题或者说已经返回的函数就会弹出调用栈：

```
Call stack：[top=f(2)][f(3)][f(4)][f(5)]
```

然后，程序发现 $f(1)$ 我们知道了，$f(2)$ 也就知道了，$f(2)=2\times f(1)$。$f(2)$ 返回了值 2，$f(2)$ 得到解决之后再将 $f(2)$ 移出栈：

```
Call stack：[top=f(3)][f(4)][f(5)]
```

依此类推，程序发现 $f(2)$ 我们知道了，$f(3)$ 也就知道了，$f(3)=3\times f(2)$。$f(3)$ 返回了值 6，相应地，$f(3)$ 得到解决之后再将 $f(3)$ 移出栈：

```
Call stack：[top=f(4)][f(5)]
```

不断地执行下去，就能够得出 $f(5)$ 的值为 120，此时栈空，程序结束。

不难看出，在运行递归程序时，栈一直在工作。因为我们调用函数的嵌套关系恰好满足先到的问题后得到结果、先调用的函数最后返回这样的关系，所以语言的设计者们就利用这一点，用栈结构来表示函数的调用关系。

小可：原来是这样，虽然看不见，但栈一直存在于我们设计的递归和函数调用程序之中。

Mr. 王：是的，栈这种看似简单的数据结构，其实应用是非常广泛的。

这里再谈谈以递归实现算法的缺点。递归程序虽然能够非常有效地表达程序的思路，使得程序的书写变得非常简洁，易于理解，但它的运行速度和执行同样工作的非递归版本相比往往是比较慢的，如果对程序的执行效率有要求，则可以将递归版本重写为非递归的。另外，递归程序在实际的执行过程中执行了多少层递归是不容易预测的。我们知道，前面提到的调用栈也是在计算机的内存空间中，如果递归的层次非常深，就会导致调用栈占用的内存空间被占满，无法继续下一层递归的运行，这就是很多人说的栈溢出或者说“爆栈”，栈溢出会导致程序运行崩溃，所以递归也并不是十全十美的。还是一定要对程序的运行环境进行评估，选择设计递归或者非递归版本的程序。

第3章　何谓大数据算法

Mr. 王：下面我们就来谈谈大数据算法与一般算法的区别和联系。

小可：好。

Mr. 王：前面我们讲了如何评价一个算法，在相对比较小的数据规模下，我们往往可以接受多项式时间算法。但是当数据量很大时，很多小数据量上我们能够在可以接受的时间内解决问题的方法，也都变得不再可以接受。虽然有些算法是多项式算法，但是它的高阶项指数却是非常大的，导致当数据规模大起来时，它的增长速度会变得非常快。对于较大的数据量，资源约束和时间约束都变得相对很苛刻，我们要对可以接受的时间界限进行重新思考。

小可：那在大数据上比较好的算法是什么样的呢？

Mr. 王：**大数据算法**是在给定的资源约束下，以大数据为输入，在给定的时间约束内可以生成满足给定约束结果的算法。

对于大数据而言，访问全部数据是很费时的，所以大数据算法有时需要采取读取部分数据的办法，也就是设计时间亚线性算法。而且数据往往在内存中也存不下，数据要存储在磁盘上，所以要考虑设计外存算法；或者是采取读取部分数据的办法，设计空间亚线性算法。

小可：亚线性算法有一种抽样的感觉，不访问全部数据，而是尝试选择部分数据来代表全部数据。

Mr. 王：没错。如果单台计算机不能保存所有的数据，或者一台计算机的计算资源不足以在给定的时间内解决问题，则还要引入多台计算机进行并行处理，让它们的 CPU、内存、磁盘都参与到问题的解决之中，这时候还要设计并行算法。甚至当计算机的能力不足以对某些问题进行处理，或者对某些问题的处理不够好，而这些问题的某一部分恰好是人类非常擅长做的工作时，还可以引入人工参与到问题的解决之中。

小可：引入人工来帮忙，这倒真是神奇啊！我还以为计算机算法都是一定要计算机来执行的呢，大数据算法还可以不是完全由计算机执行的啊？

Mr. 王：嗯，不仅如此，大数据算法还可以不是内存算法，很多时候需要磁盘参与到海量数据的存储之中；可以不是精确算法，很多时候得出精确解的代价过大，大数据算法就以得出一个足够让我们满意的近似解来谋求更高的计算效率；可以不是串行算法，在很多常见的大数据问题求解中，引入多台计算机参与到其中，发挥它们的各种计算资源的作用以提升问题的解决速度；甚至可以不是仅仅由计算机来执行的算法，在某些特定的情况下，有很多问题由人工来解决会比由机器来解决更容易、更准确或者更高效一些。从这些方面来看，大数据算法的设计和分析与传统的经典算法有着很大的区别。

另外，在大数据算法中常用的算法设计技术有：精确算法设计方法、并行算法、近似算法、随机算法、在线算法 / 数据流算法、外存算法、面向新型体系结构的算法、现代优化算法等。在大数据算法设计之后，与经典算法一样，依然要对算法进行分析。但在分析大数据算法时，我们需要研究的也不仅仅限于时间、空间复杂度的分析和优化。对于一些不能得出精确解的算法，还要对结果质量进行分析，看看在我们可以接受的时间范围内得出的近似结论是不是足够达到要求；对于要借助磁盘存储的算法，还要考虑磁盘的 IO 复杂性。有时候大数据算法会运行在比如无线传感器节点这样的对电池电力有较强限制的终端上，我们还要分析算法运行消耗的能量是不是很大，这时还要进行能量复杂度分析；如果我们使用的是一个分布式系统，整个系统架构在网络上，要依靠各个节点的频繁通信来实现，那么还要考虑系统的通信复杂度。这些特点也使得大数据算法的分析变得相对复杂一些。

这些概念现在听起来也是一头雾水吧？

小可：嗯，的确。

Mr. 王：在以后的课程中，我会把大数据算法所涉及的内容讲解给你。以后我会给你讲讲大数据算法中的亚线性算法、外存算法、并行算法、众包算法，这些都是大数据算法中的核心算法。时间不早了，我们先下课吧。

小可：那太好了。那就明天再见了，老师。

第 2 篇　理论篇

第4章　窥一斑而见全豹
——亚线性算法

4.1　亚线性算法的定义

Mr. 王：从今天开始，我们正式讲解大数据算法的内容。首先谈谈关于亚线性算法的问题。

小可：我记得前面提到过亚线性算法，就是复杂度低于输入规模的算法。

Mr. 王：我们给出一个严格的定义，还是设输入规模为 n，那么亚线性算法就是指时间、空间、通讯、能量等复杂度为 $O(n)$ 的算法。

小可若有所思，说：如果输入规模为 n，而算法的复杂度还要低于 n，这是不是说我们不能保存所有的数据，或者不能访问所有的数据呢？

Mr. 王：是的。只有这样才能实现亚线性的要求。

小可：可是，如果访问不到所有的数据，对于很多问题我们是得不到正确答案的啊。比如有一组规模比较大的数据，我们要求它们的中位数，如果不访问所有的数据，得到的结果就有可能是错误的啊。

Mr. 王：对于这样的答案我们不能简单地称之为正确解和错误解，而是称之为**精确解**和**近似解**，在大数据算法中解决问题的重要思路就是**近似**。由于在规定的时间内和计算条件下，我们得到精确解的时间太久，所以采用近似的方法来得到一个“差不多”的答案。这样的答案的误差在我们可以容忍的范围内，能够满足应用的需求就可以了。在小的数据集合上，我们时常需要的是精确解，而当数据量很大时，得到精确解的开销也会变得大到不能接受。因此，我们

求近似解以换取更高的效率。近似是亚线性算法的核心思想。

小可：原来是这样。

Mr. 王：亚线性算法也可以分为空间亚线性算法、时间亚线性计算算法和时间亚线性判定算法。下面我就这几类问题分别举个典型的例子来看看亚线性算法是如何解决问题的。

4.2　空间亚线性算法

4.2.1　水库抽样

Mr. 王：首先我们看一个经典问题：水库抽样。这是一个典型的空间亚线性算法。考虑这样一个问题：很多时候我们要在大宗数据中进行一个均匀的抽样，这在实际的生活中是非常常见的，但是在实际情况下，有时候我们要处理的数据量太大，以至于只能让这些数据从我们的面前“流过”一次。这就好比数据从一个生产线或者流水线上源源不断地到来，当来到计算系统中时，我们可以对其进行处理，但是对其进行过处理之后，或者由于能存储这些数据的能力有限，我们不得不将其从内存中清除出去，这样就再也不能访问它了。

水库抽样问题的要求是，每一刻所取到的样本，就是前面已经“流过”的全部数据的均匀抽样。你来根据我们前面提过的分析问题的方法，说一说问题定义。

小可想了想，说：

输入：一组数据，其大小未知。

输出：这组数据的 k 个均匀抽样。

要求：

a．仅扫描数据一次；

b．空间复杂性为 $O(k)$；

c．扫描到前 n（$n>k$）个数据时，保存当前已扫描数据的 k 个均匀抽样。

Mr. 王：说得不错，对于这个问题，我们给出下面的算法：

（1）申请一个长度为 k 的数组 A 保存抽样（此处的数组下标以 $[1,k]$ 表示，对于 $[0,k\text{-}1]$ 的实际操作情况，可以非常容易地进行替换）。

（2）保存首先接收到的 k 个元素。

（3）当接收到第 i 个新元素 t 时，生成 $[1,i]$ 间随机数 j，若 $j \leqslant k$，则以 t 替换 $A[j]$。

算法的关键在于第 3 步：每当新到来一个元素 t 时，都生成一个随机数，一旦随机数落在了数组范围之内，就用新元素把它替换掉。

小可：这个算法看起来倒是挺简单的，用一个随机数来决定是不是进行抽样替换，可是它

真的能够满足均匀抽样的需求吗?

Mr. 王：嗯，接下来我们就来证明这个算法的正确性。你先说一说，均匀采样应该满足什么条件?

小可：在本题目的条件中，对于任意一个元素 i，它被选入样本的概率均为 k/n。

Mr. 王：好，那么我们只需要证明该算法满足这个要求就可以了。首先，对于每一个新到来的元素 i，它是以 k/i 的概率被收入抽样集合的，这是因为生成的随机数范围是 $[1,i]$，而当数字小于等于 k 时，它会被替换进数组 $\boldsymbol{A}$。

当第 i+1 个元素到来时，i+1 被替换进来的概率就是 k/i+1，而此时，前一个元素 i 被从中替换出来的概率是 $1/k$。这两个值的乘积就是当第 i+1 个元素到来时前面的 i 被替换出来的概率，其值为 $1/(i+1)$，那么 $1\text{-}1/(i+1)$ 就是 i 没有在 i+1 到来时被替换出去的概率。当 i+2、i+3 等这些元素到来时，其计算过程和 i+1 是同理的。

如果元素 i 被选入集合中，并且在后面所有的替换过程中，每一次替换都没有被替换出去时，它就是我们选出来的样本，那么元素 i 在样本中的概率应该是多少呢?

小可：

$$\frac{k}{i}\times\left(1-\frac{1}{i+1}\right)\times\left(1-\frac{1}{i+2}\right)\times\cdots\times\left(1-\frac{1}{n}\right)=\frac{k}{n}$$

Mr. 王：这就是说，对于任意元素 i，其被选入样本的概率均为 k/n。也就是说，它符合随机抽样。

小可：原来随机决定了替换的结果，还真的能保证抽样的均匀性。

Mr. 王：讨论过算法的正确性，可以确定这个算法的执行结果是正确的，但是我们希望它是一个空间亚线性算法，所以还要分析它的空间复杂度是不是满足亚线性这一要求。你来分析一下，这个算法的空间复杂度如何?

小可：不论“流动”来了多少个数据，我们只需要保存 k 个数据作为样本就可以了，其余的计算空间都是常数开销，那就应该是 $O(k)$。

Mr. 王：显然，k 是小于 n 的。也就是说，我们的这个算法在正确的前提下，对于输入规模 n，做到了 $O(k)$ 的空间复杂度，而 $O(k) \in o(n)$，也就表明，它是一个空间亚线性算法。

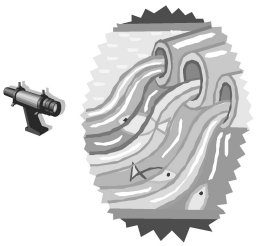
小可：我懂了。

Mr. 王：这里也为我们设计空间亚线性算法提供了一个思路，就是不能去尝试保存全部的数据。我们要进行亚线性的均匀抽样，不能把所有的数据都保存下来，再去做均匀抽样，这样势必会造成 $O(n)$ 的空间复杂度。我们要考虑，能不能在只保存大数据中的一个小集合的情况下，来达到问题的要求。在水库抽样问题中，虽然我们采用的是空间亚线性算法，但是得到的依然是精确解。

4.2.2　数据流中的频繁元素

Mr. 王：我们再来讲一个例子，数据流中的频繁元素。我们先来说说大数据的数据流模型。

小可：数据流，是流动的数据的意思吗？和我们前面说的水库抽样是不是很像？



数据流

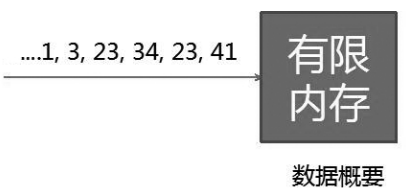
Mr. 王点点头，说：嗯，其实水库抽样也是以数据流的思想来处理的。顾名思义，数据流意味着数据在流动，数据会不断地到达计算系统进行处理，这意味着一个数据只能被扫描一次，一旦处理过或者在内存中被放弃，就不能再访问了。你想想看，这在复杂度上意味着什么？

小可想了想，说：超过 O(n) 的算法肯定是不行的，只能寻找亚线性算法了。

Mr. 王：没错，而且数据流模型的数据是源源不断到来的，比如我们使用一个传感器进行感知，它就会按照一定的采样频率进行数据采集。其实传感器获得的数据就是一个典型的数据流，如果传感器一直在工作，我们就会得到源源不断的数据。但是不管是传感器使用的内存还是其他存储介质都不是无限的，这是符合客观现实的。所以对数据流进行处理，要求我们所使用的内存必须是亚线性的，最好是和数据量无关的。

小可：就像水库抽样一样吧，内存中随时保存着的都是对前面数据流的一个均匀抽样，而且所使用的内存有限，不论来了多少数据，都只保存 k 个，也是与数据量无关的。

Mr. 王：很好，这样用于概括数据的数据结构叫作数据概要，我们用有限的、与数据量无关的内存空间来维护这个数据概要，就是要对相关性质的当前状态做出一个有效估计。



数据流模型

我们说数据流模型是适用于大数据的，因为它仅顺序扫描数据一次，而且它的内存是亚线性的。数据流通常是来自某个域中元素的序列，$<x_1, x_2, x_3, x_4, ...>$。

小可：这个序列里为什么没有末项 x_n？

Mr. 王：因为数据是源源不断的，所以没有末项。好了，我们来总结一下数据流模型的特点：

（1）数据流通常是来自某个域中元素的序列，$<x_1, x_2, x_3, x_4, ...>$。

（2）数据量是远大于内存容量的，这意味着无法将所有的数据都放进内存中。内存的规模一般为 $O(\log^k n)$ 或者 $O(n^\alpha)$，显然 $\alpha<1$。

（3）要快速处理每一个数据，因为数据会快速地源源不断地到来，这也是很必要的一点。

小可：嗯，那我们将大数据视为数据流模型，可以拿来计算什么呢？

Mr. 王：应用就有很多了，有 sum（求和）、max（最大值）、min（最小值）、count（计数）、avg（平均数）这样的基本统计量，还有比如中位数、频繁项、分析、挖掘、预警等。

我们还是举个具体的例子吧，这里有一组数字：32,12,14,32,7,12,32,7,6,12,4。

Mr. 王：你说说看，前面提到的统计量哪些是比较容易计算的？

小可：前面讲解选择排序时，讲过求 min 的方法，我觉得那个方法同样适用于数据流，只要拿出一个变量来保存当前的最小值，发现比它更小的就将其替换掉；max 与之同理。求和 sum 也是很容易的，只要用一个变量来保存当前的加和，每到来一个数据，就把它加进这个变量里。至于 count，初始化为 0，然后每到来一个数据，就把它加 1。如果要求平均数 avg，就同时维护 sum 和 count 作商。

Mr. 王：很好，可以看出，这样的数据概要是单个值，同时它也是一个可合并的值。在这里，“可合并”就是指两部分数据流的数据概要可以直接通过诸如累加和比较这样的操作合并成整个数据流的概要。

小可：嗯，比如我们要求 200 个数据的最大值，前 100 个数据的最大值和后 100 个数据的最大值的较大者，就是 200 个数据的最大值。

Mr. 王：现在我们来处理一个复杂一点的问题——频繁元素。

为了研究方便，我们用 n 来表示不同元素的个数，用 m 来表示元素的总个数。

你先来想一想，如果要求出一个精确解的话，可以用什么方法？

小可：可以用这样的方法：

（1）为每一个单独元素设置一个计数器。

（2）当处理一个元素时，增加相应的计数器。

Mr. 王：这样做有什么问题吗？

小可：如果每个元素都有自己的计数器，就需要有 n 个计数器，这样没有满足 $o(n)$ 的要求，相当于内存需要存储所有的数据，这对于数据流来说是不能实现的。

Mr. 王：嗯，所以我们提出如下的方法：

（1）处理一个新到来的元素 x 时

（2）If 已经为其分配了计数器，增加之

（3）Else If 没有相应的计数器，但计数器个数少于 k，为 x 分配计数器 k，并设为 1

（4）Else 所有的计数器值减 1，删除值为 0 的计数器

这个算法称为 Misra Gries（MG）算法。第一种情况，如果内存中已经有新到来元素的计数器，则只需要将其值加 1 即可；第二种情况，如果还没有为新到来的元素提供计数器，并且内存没有被填满时，则可以为这个元素的计数器开辟新的空间；第三种情况，当新到来的元素没有被分配计数器，同时内存中的计数器个数已经达到了 k 个，也就是分配的内存空间已经被填满时，则将所有的计数器值减 1，删除值为 0 的计数器，此时内存中就重新有位置了，我们再为这个新到达的元素分配一个计数器即可。当然，别忘了要将其置为 1。

我们用前面的这组数字举个例子：32,12,14,32,7,12,32,7,6。

假设内存中有 3 个存放计数的空间。

前 3 个数据进入内存时，都符合情况二，将它们加入内存中。

抵达数据：32　　内存：[32:1]

抵达数据：12　　内存：[32:1][12:1]

抵达数据：14　　内存：[32:1][12:1][14:1]

第 4 个数据 32 到来时，将 32 的计数值加 1。

抵达数据：32　　内存：[32:2][12:1][14:1]

当第 5 个数据 7 抵达时，符合情况三，也就是频繁元素统计的大数据处理的关键，我们将所有的计数器值减 1，并删除那些值为 0 的计数器，然后为新到来的 7 建立计数器，并置为 1。

抵达数据：7　　内存：[32:1][12:0][14:0]

内存：[32:1]

内存：[32:1][7:1]

然后是 12、32 和 7，分别属于情况二和情况一。

抵达数据：12　　内存：[32:1][7:1][12:1]

抵达数据：32　　内存：[32:2][7:1][12:1]

抵达数据：7　　内存：[32:2][7:2][12:1]

当 6 到达时，内存已经被填满，我们执行算法的第三种情况。

抵达数据：6　　内存：[32:1][7:1][12:0]

内存：[32:1][7:1]

内存：[32:1][7:1][6:1]

使用该算法求出的几个频繁元素就是 32、7 和 6。就原数据来说，最频繁的 3 个元素分别是 32、7、12。我们成功地找出了 32 和 7，虽然没有找出最频繁的 3 个元素，但整体来说已经是一个不错的结果了。如果只需要最频繁的元素，那么该算法已经在这组数字中找出了 32 这个最频繁的元素。不过在最后对频繁元素的计数值一般是不准确的，所以还要对它的计数进行分析，估计它所记录的数值误差如何。当我们使用近似手段处理问题时，一定要对近似解进行误差分析，研究所得到的近似解是否在我们可以接受的误差范围之内，误差太大的近似解也同

样达不到解决问题的目的。

你说说看，这个近似算法是高估了频繁元素的数量还是低估了？

小可：内存中的频繁元素的计数器不断地由于内存被占满而被削减，显然是低估了。

Mr. 王：没错，我们不能仅仅知道结果是低估了准确值，最好还要能根据算法分析出究竟低估了多少。想要知道低估了多少，我们首先要考虑的就是一个计数器被减小了几次。这就需要我们考虑到在整个算法的执行过程中，执行过多少个减少计数器的步骤。假如把整个结构的权重（也就是计数器的和）记作 m'，整个数据流的权重（全部元素的数量）记作 m。每当计数器需要降低时，由于内存中有 k 个计数器，我们也就减少了 k 个计数，但是这时新到来的元素 x 并未计入内存中的计数器，它的到来只是标志着该削减计数器了，所以我们少加了 k+1 个计数器。因此，最多有$\frac{m-m'}{k+1}$个减少步骤。也就是说，估计和真实值最多相差$\frac{m-m'}{k+1}$。如果数据流的元素总量远大于$\frac{m-m'}{k+1}$值，我们可以得到一个好的频繁元素的估计。

可以看出，错误的界限是与 k 成反比的。你说说看，这说明什么？

小可：k 是内存中计数器的个数，也就是程序使用内存的大小，这说明内存越大，结果就越准确。

Mr. 王：嗯，这也是符合客观规律的。而且我们可以利用数据概要来计算错误的界限，只需要记录 m、m' 和 k 就可以了。

不过不难看出，如果数据集合中每个元素的数量都相差不多的话，这个算法求出的结果会具有很大的随机性，好在我们一般需要处理的数据都满足 Zipf 法则。

Zipf 法则：典型的频率分布是高度偏斜的，只有少数频繁元素，最多 10% 的元素占元素总个数的 90%。这个定律说明，只有少数的元素是大量重复出现的，而绝大多数元素的出现是不频繁的。

根据 Zipf 法则我们知道，频繁元素的种类只有少数，而其数量往往是非常大的，在算法执行的过程中，不断地削减内存中的计数器对于频繁元素最终被保留在内存里不会有太大程度的影响。

4.3 时间亚线性计算算法

4.3.1 图论基础回顾

Mr. 王：我们再讲一个时间亚线性算法——平面图直径的求解。平面图是图论中的一个概念，在大数据算法的很多地方都会涉及图的相关内容，所以这里我们还是要回顾一下图论的知识。