第 2 章　算法

启示

算 法:

算法是解决特定问题求解步骤的描述，在计算机中表现为指令的有限序列，并且每条指令表示一个或多个操作。

2.1 开 场 白

各位同学大家好。

上次上完课后，有同学对我说，老师，我听了你的课，感觉数据结构没什么的，你也太夸大它的难度了。

是呀，我好像是强调了数据结构比较搞脑子，而上次课，其实还没拿出复杂的东西来说道。不是不想，是没必要，第一次课就把你们糊弄晕，那以后还玩什么，逃课的不就更多了吗？你们看，今天来的人数和第一次差不多，而且暂时还没有睡觉的。

今天我们介绍的内容在难度上就有所增加了，做好准备了吗？

2.2 数据结构与算法关系

我们这门课程叫数据结构，但很多时候我们会讲到算法，以及它们之间的关系。市场上也有不少书叫“数据结构与算法分析”这样的名字。

有人可能就要问了，那你到底是只讲数据结构呢，还是和算法一起讲？它们之间是什么关系呢？干吗要放在一起？

这问题怎么回答。打个比方吧，今天是你女友生日，你打算请女友去看爱情音乐剧，到了戏院，抬头一看——《梁山伯》18：00 开演。嗯，怎么会是这样？一问才知，今天饰演祝英台的演员生病，所以梁山伯唱独角戏。真是搞笑了，这还有什么看头。于是你们打算去看爱情电影。到了电影院，一看海报——《罗密欧》，是不是名字写错了，问了才知，原来饰演朱丽叶的演员因为嫌弃演出费用太低，中途退演了。制片方考虑到已经开拍，于是就把电影名字定为《罗密欧》，主要讲男主角的心路旅程。哎，这电影还怎么看啊？

事实上，数据结构和算法也是类似的关系。只谈数据结构，当然是可以，我们可以在很短的时间就把几种重要的数据结构介绍完。听完后，很可能你没什么感觉，不知道这些数据结构有何用处。但如果我们再把相应的算法也拿来讲一讲，你就会发现，甚至开始感慨：哦，计算机界的前辈们，的确是一些很牛很牛的人，他们使得很多看似很难解决或者没法解决的问题，变得如此美妙和神奇。

也许从这以后，慢慢地你们中的一部分会开始把你们的崇拜对象，从帅哥美女、

什么“哥”什么“姐”们，转移到这些大胡子或者秃顶的老头身上，那我就非常欣慰了。而且，这显然是一种成熟的表现，我期待你们中多一点这样的人，这样我们国家的软件行业，也许就有得救了。

不过话说回来，现在好多大学里，通常都是把“算法”分出一门课单独讲的，也就是说，在《数据结构》课程中，就算谈到算法，也是为了帮助理解好数据结构，并不会详细谈及算法的方方面面。我们的课程也是按这样的原则来展开的。

2.3 两种算法的比较

大家都已经学过一门计算机语言，不管学的是哪一种，学得好不好，好歹是可以写点小程序了。现在我要求你写一个求 1+2+3+……+100 结果的程序，你应该怎么写呢?

大多数人会马上写出下面的 C 语言代码（或者其他语言的代码)：

```
int i, sum = 0, n = 100;
for（i = 1; i < = n; i++）
{
    sum = sum + i;
}
printf（" %d ", sum）;
```

这是最简单的计算机程序之一，它就是一种算法，我不去解释这代码的含义了。问题在于，你的第一直觉是这样写的，但这样是不是真的很好？是不是最高效?

此时，我不得不把伟大数学家高斯的童年故事拿来说一遍，也许你们都早已经听过，但不妨再感受一下，天才当年是如何展现天分和才华的。

据说 18 世纪生于德国小村庄的高斯，上小学的一天，课堂很乱，就像我们现在下面那些窃窃私语或者拿着手机不停摆弄的同学一样，老师非常生气，后果自然也很严重。于是老师在放学时，就要求每个学生都计算 1+2+…+100 的结果，谁先算出来谁先回家。

天才当然不会被这样的问题难倒，高斯很快就得出了答案，是 5050。老师非常惊讶，因为他自己想必也是通过 1+2=3，3+3=6，6+4=10，……，4950+100=5050 这样算出来的，也算了很久很久。说不定为了怕错，还算了两三遍。可眼前这个少年，为何可以这么快地得出结果?

高斯解释道：

$$
\begin{aligned}
\text{sum} &= 1 + 2 + 3 + \ldots + 99 + 100 \\
\text{sum} &= 100 + 99 + 98 + \ldots + 2 + 1 \\
2\times\text{sum} &= \underbrace{101 + 101 + 101 + \ldots + 101 + 101}_{\text{共 } 100 \text{ 个}}
\end{aligned}
$$

所以 sum=5050

用程序来实现如下：

```
int i, sum = 0,n = 100;
sum = （1 + n） * n / 2;
printf（"%d", sum）;
```

神童就是神童，他用的方法相当于另一种求等差数列的算法，不仅仅可以用于 1 加到 100，就是加到一千、一万、一亿（需要更改整型变量类型为长整型，否则会溢出），也就是瞬间之事。但如果用刚才的程序，显然计算机要循环一千、一万、一亿次的加法运算。人脑比电脑算得快，似乎成为了现实。

2.4 算法定义

什么是算法呢？算法是描述解决问题的方法。算法（Algorithm）这个单词最早出现在波斯数学家阿勒·花剌子密在公元 825 年（相当于我们中国的唐朝时期）所写的《印度数字算术》中。如今普遍认可的对算法的定义是：

> **算法是解决特定问题求解步骤的描述，在计算机中表现为指令的有限序列，并且每条指令表示一个或多个操作。**

刚才的例子我们也看到，对于给定的问题，是可以有多种算法来解决的。

那我就要问问你们，有没有通用的算法呀？这个问题其实很弱智，就像问有没有可以包治百病的药呀！

现实世界中的问题千奇百怪，算法当然也就千变万化，没有通用的算法可以解决所有的问题。甚至解决一个小问题，很优秀的算法却不一定适合它。

算法定义中，提到了指令，指令能被人或机器等计算装置执行。它可以是计算机指令，也可以是我们平时的语言文字。

为了解决某个或某类问题，需要把指令表示成一定的操作序列，操作序列包括一组操作，每一个操作都完成特定的功能，这就是算法了。

2.5 算法的特性

算法具有五个基本特性：输入、输出、有穷性、确定性和可行性。

2.5.1 输入输出

输入和输出特性比较容易理解，**算法具有零个或多个输入。**尽管对于绝大多数算法来说，输入参数都是必要的，但对于个别情况，如打印“hello world!”这样的代码，不需要任何输入参数，因此算法的输入可以是零个。**算法至少有一个或多个输出，**算法是一定需要输出的，不需要输出，你用这个算法干吗？输出的形式可以是打印输出，也可以是返回一个或多个值等。

2.5.2 有穷性

有穷性：指算法在执行有限的步骤之后，自动结束而不会出现无限循环，并且每一个步骤在可接受的时间内完成。现实中经常会写出死循环的代码，这就是不满足有穷性。当然这里有穷的概念并不是纯数学意义的，而是在实际应用当中合理的、可以接受的“有边界”。你说你写一个算法，计算机需要算上个二十年，一定会结束，它在数学意义上是有穷了，可是媳妇都熬成婆了，算法的意义也不就大了。

2.5.3 确定性

确定性：算法的每一步骤都具有确定的含义，不会出现二义性。算法在一定条件下，只有一条执行路径，相同的输入只能有唯一的输出结果。算法的每个步骤被精确定义而无歧义。

2.5.4 可行性

可行性：算法的每一步都必须是可行的，也就是说，每一步都能够通过执行有限次数完成。可行性意味着算法可以转换为程序上机运行，并得到正确的结果。尽管在目前计算机界也存在那种没有实现的极为复杂的算法，不是说理论上不能实现，而是

因为过于复杂，我们当前的编程方法、工具和大脑限制了这个工作，不过这都是理论研究领域的问题，不属于我们现在要考虑的范围。

2.6 算法设计的要求

刚才我们谈到了，算法不是唯一的。也就是说，同一个问题，可以有多种解决问题的算法。这可能让那些常年只做有标准答案题目的同学失望了，他们多么希望存在标准答案，只有一个是正确的，把它背下来，需要的时候套用就可以了。不过话说回来，尽管算法不唯一，相对好的算法还是存在的。掌握好的算法，对我们解决问题很有帮助，否则前人的智慧我们不能利用，就都得自己从头研究了。那么什么才叫好的算法呢?

嗯，没错，有同学说，好的算法，起码要是正确的，连正确都谈不上，还谈什么别的要求?

2.6.1 正确性

正确性：算法的正确性是指算法至少应该具有输入、输出和加工处理无歧义性、能正确反映问题的需求、能够得到问题的正确答案。

但是算法的“正确”通常在用法上有很大的差别，大体分为以下四个层次。

1．算法程序没有语法错误。

2．算法程序对于合法的输入数据能够产生满足要求的输出结果。

3．算法程序对于非法的输入数据能够得出满足规格说明的结果。

4．算法程序对于精心选择的，甚至刁难的测试数据都有满足要求的输出结果。

对于这四层含义，层次 1 要求最低，但是仅仅没有语法错误实在谈不上是好算法。这就如同仅仅解决温饱，不能算是生活幸福一样。而层次 4 是最困难的，我们几乎不可能逐一验证所有的输入都得到正确的结果。

因此算法的正确性在大部分情况下都不可能用程序来证明，而是用数学方法证明的。证明一个复杂算法在所有层次上都是正确的，代价非常昂贵。所以一般情况下，我们把层次 3 作为一个算法是否正确的标准。

好算法还有什么特征呢?

很好，我听到了说算法容易理解。没错，就是它。

2.6.2 可读性

可读性：算法设计的另一目的是为了便于阅读、理解和交流。

可读性高有助于人们理解算法，晦涩难懂的算法往往隐含错误，不易被发现，并且难于调试和修改。

我在很久以前曾经看到过一个网友写的代码，他号称这程序是“用史上最少代码实现俄罗斯方块”。因为我自己也写过类似的小游戏程序，所以想研究一下他是如何写的。由于他追求的是“最少代码”这样的极致，使得他的代码真的不好理解。也许除了计算机和他自己，绝大多数人是看不懂他的代码的。

我们写代码的目的，一方面是为了让计算机执行，但还有一个重要的目的是为了便于他人阅读，让人理解和交流，自己将来也可能阅读，如果可读性不好，时间长了自己都不知道写了些什么。可读性是算法（也包括实现它的代码）好坏很重要的标志。

2.6.3 健壮性

一个好的算法还应该能对输入数据不合法的情况做合适的处理。比如输入的时间或者距离不应该是负数等。

健壮性：当输入数据不合法时，算法也能做出相关处理，而不是产生异常或莫名其妙的结果。

2.6.4 时间效率高和存储量低

最后，好的算法还应该具备时间效率高和存储量低的特点。

时间效率指的是算法的执行时间，对于同一个问题，如果有多个算法能够解决，执行时间短的算法效率高，执行时间长的效率低。存储量需求指的是算法在执行过程中需要的最大存储空间，主要指算法程序运行时所占用的内存或外部硬盘存储空间。**设计算法应该尽量满足时间效率高和存储量低的需求。**在生活中，人们都希望花最少的钱，用最短的时间，办最大的事，算法也是一样的思想，最好用最少的存储空间，花最少的时间，办成同样的事就是好的算法。求 100 个人的高考成绩平均分，与求全省的所有考生的成绩平均分在占用时间和内存存储上是有非常大的差异的，我们自然

是追求可以高效率和低存储量的算法来解决问题。

综上，好的算法，应该具有正确性、可读性、健壮性、高效率和低存储量的特征。

2.7 算法效率的度量方法

刚才我们提到设计算法要提高效率。这里效率大都指算法的执行时间。那么我们如何度量一个算法的执行时间呢？

正所谓“是骡子是马，拉出来遛遛”。比较容易想到的方法就是，我们通过对算法的数据测试，利用计算机的计时功能，来计算不同算法的效率是高还是低。

2.7.1 事后统计方法

事后统计方法：这种方法主要是通过设计好的测试程序和数据，利用计算机计时器对不同算法编制的程序的运行时间进行比较，从而确定算法效率的高低。

但这种方法显然是有很大缺陷的：

- 必须依据算法事先编制好程序，这通常需要花费大量的时间和精力。如果编制出来发现它根本是很糟糕的算法，不是竹篮打水一场空吗？
- 时间的比较依赖计算机硬件和软件等环境因素，有时会掩盖算法本身的优劣。要知道，现在的一台四核处理器的计算机，跟当年 286、386、486 等老爷爷辈的机器相比，在处理算法的运算速度上，是不能相提并论的；而所用的操作系统、编译器、运行框架等软件的不同，也可以影响它们的结果；就算是同一台机器， CPU 使用率和内存占用情况不一样，也会造成细微的差异。
- 算法的测试数据设计困难，并且程序的运行时间往往还与测试数据的规模有很大关系，效率高的算法在小的测试数据面前往往得不到体现。比如 10 个数字的排序，不管用什么算法，差异几乎是零。而如果有一百万个随机数字排序，那不同算法的差异就非常大了。那么我们为了比较算法，到底用多少数据来测试，这是很难判断的问题。

基于事后统计方法有这样那样的缺陷，我们考虑不予采纳。

2.7.2 事前分析估算方法

我们的计算机前辈们，为了对算法的评判更科学，研究出了一种叫做事前分析估算的方法。

事前分析估算方法：在计算机程序编制前，依据统计方法对算法进行估算。

经过分析，我们发现，一个用高级程序语言编写的程序在计算机上运行时所消耗的时间取决于下列因素：

1．算法采用的策略、方法。

2．编译产生的代码质量。

3．问题的输入规模。

4．机器执行指令的速度。

第 1 条当然是算法好坏的根本，第 2 条要由软件来支持，第 4 条要看硬件性能。也就是说，抛开这些与计算机硬件、软件有关的因素，**一个程序的运行时间，依赖于算法的好坏和问题的输入规模。**所谓**问题输入规模是指输入量的多少。**

我们来看看今天刚上课时举的例子，两种求和的算法：

第一种算法：

```
int i, sum = 0,n = 100;          /*执行 1 次*/
for（i = 1; i < = n; i++）       /*执行了 n+1 次*/
{
    sum = sum + i;               /*执行 n 次*/
}
printf（"%d", sum）;             /*执行 1 次*/
```

第二种算法：

```
int sum = 0,n = 100;             /*执行一次*/
sum = （1 + n） * n/2;           /*执行一次*/
printf（"%d", sum）;             /*执行一次*/
```

显然，第一种算法，执行了 1+（n+1）+n+1 次=2n+3 次；而第二种算法，是 1+1+1=3 次。事实上两个算法的第一条和最后一条语句是一样的，所以我们关注的代码其实是中间的那部分，我们把循环看作一个整体，忽略头尾循环判断的开销，那么这两个算法其实就是 n 次与 1 次的差距。算法好坏显而易见。

我们再来延伸一下上面这个例子：

```
int i, j, x = 0,sum = 0,n = 100;    /*执行一次*/
for（i = 1; i < = n; i++）
{
    for （j = 1; j < = n; j++）
    {
        x++;                        /*执行 n×n 次*/
        sum = sum + x;
    }
}
printf（"%d", sum）;                /*执行一次*/
```

这个例子中，i从 1 到 100，每次都要让j循环 100 次，而当中的x++和sum = sum + x；其实就是 1+2+3+…+10000，也就是 100^2次，所以这个算法当中，循环部分的代码整体需要执行n^2（忽略循环体头尾的开销）次。显然这个算法的执行次数对于同样的输入规模n = 100，要多于前面两种算法，这个算法的执行时间随着n的增加也将远远多于前面两个。

此时你会看到，测定运行时间最可靠的方法就是计算对运行时间有消耗的基本操作的执行次数。运行时间与这个计数成正比。

我们不关心编写程序所用的程序设计语言是什么，也不关心这些程序将跑在什么样的计算机中，我们只关心它所实现的算法。这样，不计那些循环索引的递增和循环终止条件、变量声明、打印结果等操作，**最终，在分析程序的运行时间时，最重要的是把程序看成是独立于程序设计语言的算法或一系列步骤。**

可以从问题描述中得到启示，同样问题的输入规模是n，求和算法的第一种，求1+2+…+n需要一段代码运行n次。那么这个问题的输入规模使得操作数量是f（n）= n，显然运行 100 次的同一段代码规模是运算 10 次的 10 倍。而第二种，无论n为多少，运行次数都为 1，即f（n）= 1；第三种，运算 100 次是运算 10 次的 100 倍。因为它是f（n）= n^2。

我们在分析一个算法的运行时间时，重要的是把基本操作的数量与输入规模关联起来，即基本操作的数量必须表示成输入规模的函数（如图 2-7-1 所示）。

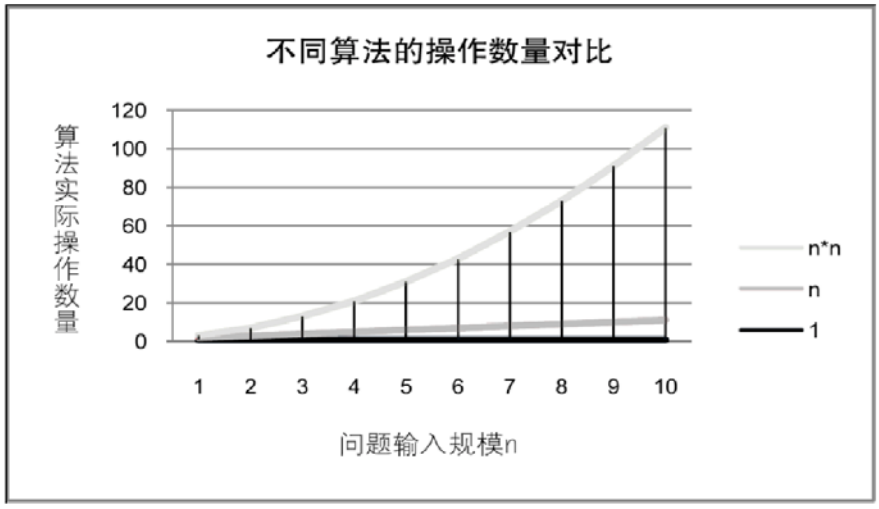

图 2-7-1

我们可以这样认为，随着 n 值的越来越大，它们在时间效率上的差异也就越来越大。好比你们当中有些人每天都在学习，我指有用的学习，而不是只为考试的死读书，每天都在进步，而另一些人，打打游戏，睡睡大觉。入校时大家都一样，但毕业时结果可能就大不一样，前者名企争抢着要，后者求职无门。

2.8 函数的渐近增长

我们现在来判断一下，两个算法 A 和 B 哪个更好。假设两个算法的输入规模都是 n，算法 A 要做 2n + 3 次操作，你可以理解为先有一个 n 次的循环，执行完成后，再有一个 n 次循环，最后有三次赋值或运算，共 2n + 3 次操作。算法 B 要做 3n + 1 次操作。你觉得它们谁更快呢?

准确说来，答案是不一定的（如表 2-8-1 所示）。

表 2-8-1

次数	算法 A（2n + 3）	算法 A'（2n）	算法 B（3n + 1）	算法 B'（3n）
n = 1	5	2	4	3
n = 2	7	4	7	6
n = 3	9	6	10	9
n = 10	23	20	31	30
n = 100	203	200	301	300

当 n = 1 时，算法 A 效率不如算法 B（次数比算法 B 要多一次）。而当 n = 2 时，两者效率相同；当 n > 2 时，算法 A 就开始优于算法 B 了，随着 n 的增加，算法 A 要

越来越好过算法 B 了（执行的次数比 B 要少）。于是我们可以得出结论，算法 A 总体上要好过算法 B。

此时我们给出这样的定义，输入规模 n 在没有限制的情况下，只要超过一个数值 N，这个函数就总是大于另一个函数，我们称函数是渐近增长的。

函数的渐近增长：给定两个函数 f（n）和 g（n），如果存在一个整数 N，使得对于所有的 n > N，f（n）总是比 g（n）大，那么，我们说 f（n）的增长渐近快于 g（n）。

从中我们发现，随着 n 的增大，后面的 +3 还是 +1 其实是不影响最终的算法变化的，例如算法 A′ 与算法 B′ ，所以，**我们可以忽略这些加法常数**。后面的例子，这样的常数被忽略的意义可能会更加明显。

我们来看第二个例子，算法C是 $4n + 8$，算法D是 $2n^2 + 1$（如表 2-8-2 所示）。

表 2-8-2

次数	算法 C（4n+8）	算法 C'（n）	算法D（$2n^2$+1）	算法D'（n^2）
n = 1	12	1	3	1
n = 2	16	2	9	4
n = 3	20	3	19	9
n = 10	48	10	201	100
n = 100	408	100	20 001	10 000
n = 1000	4 008	1 000	2 000 001	1 000 000

当 $n \leqslant 3$ 的时候，算法 C 要差于算法 D（因为算法 C 次数比较多），但当 $n > 3$ 后，算法 C 的优势就越来越优于算法 D 了，到后来更是远远胜过。而当后面的常数去掉后，我们发现其实结果没有发生改变。甚至我们再观察发现，哪怕去掉与 n 相乘的常数，这样的结果也没发生改变，算法 C′ 的次数随着 n 的增长，还是远小于算法 D′ 。也就是说，**与最高次项相乘的常数并不重要。**

我们再来看第三个例子。算法E是 $2n^2 + 3n + 1$，算法F是 $2n^3 + 3n + 1$（如表 2-8-3 所示）。

表 2-8-3

次数	算法E（$2n^2$+3n+1）	算法E'（n^2）	算法F（$2n^3$+3n+1）	算法F'（n^3）
n = 1	6	1	6	1
n = 2	15	4	23	8
n = 3	28	9	64	27
n = 10	231	100	2 031	1 000
n = 100	20 301	10 000	2 000 301	1 000 000

当 n = 1 的时候，算法 E 与算法 F 结果相同，但当 n > 1 后，算法 E 的优势就要开始优于算法 F，随着 n 的增大，差异非常明显。通过观察发现，**最高次项的指数大的，函数随着 n 的增长，结果也会变得增长特别快。**

我们来看最后一个例子。算法G是 $2n^2$，算法H是 $3n + 1$，算法I是 $2n^2 + 3n + 1$（如表 2-8-4 所示）。

表 2-8-4

次数	算法G（$2n^2$）	算法 H（3n+1）	算法I（$2n^2$+3n+1）
n = 1	2	4	6
n = 2	8	7	15
n = 5	50	16	66
n = 10	200	31	231
n = 100	20 000	301	20 301
n = 1,000	2 000 000	3 001	2 003 001
n = 10,000	200 000 000	30 001	200 030 001
n = 100,000	20 000 000 000	300 001	20 000 300 001
n = 1,000,000	2 000 000 000 000	3 000 001	200 000 3000 001

这组数据应该就看得很清楚。当n的值越来越大时，你会发现，3n+1 已经没法和 $2n^2$的结果相比较，最终几乎可以忽略不计。也就是说，随着n值变得非常大以后，算法G其实已经很趋近于算法I。于是我们可以得到这样一个结论，**判断一个算法的效率时，函数中的常数和其他次要项常常可以忽略，而更应该关注主项（最高阶项）的阶数**。

判断一个算法好不好，我们只通过少量的数据是不能做出准确判断的。根据刚才的几个样例，我们发现，如果我们可以对比这几个算法的关键执行次数函数的渐近增长性，基本就可以分析出：**某个算法，随着 n 的增大，它会越来越优于另一算法，或者越来越差于另一算法**。这其实就是事前估算方法的理论依据，通过算法时间复杂度来估算算法时间效率。

2.9 算法时间复杂度

2.9.1 算法时间复杂度定义

在进行算法分析时，语句总的执行次数 T（n）是关于问题规模 n 的函数，进而分析 T（n）随 n 的变化情况并确定 T（n）的数量

级。算法的时间复杂度，也就是算法的时间量度，记作：T（n）= O(f(n))。它表示随问题规模 n 的增大，算法执行时间的增长率和 f（n）的增长率相同，称作算法的渐近时间复杂度，简称为时间复杂度。其中 f（n）是问题规模 n 的某个函数。

这样用大写 O( )来体现算法时间复杂度的记法，我们称之为**大 O 记法**。

一般情况下，随着 n 的增大，T(n)增长最慢的算法为最优算法。

显然，由此算法时间复杂度的定义可知，我们的三个求和算法的时间复杂度分别为$O(n)$，$O(1)$，$O(n^2)$。我们分别给它们取了非官方的名称，$O(1)$叫常数阶、$O(n)$叫线性阶、$O(n^2)$叫平方阶，当然，还有其他的一些阶，我们之后会介绍。

2.9.2 推导大 O 阶方法

那么如何分析一个算法的时间复杂度呢？即如何推导大 O 阶呢？我们给出了下面的推导方法，基本上，这也就是总结前面我们举的例子。

推导大 O 阶：

1.用常数 1 取代运行时间中的所有加法常数。

2.在修改后的运行次数函数中，只保留最高阶项。

3.如果最高阶项存在且不是 1，则去除与这个项相乘的常数。

得到的结果就是大 O 阶。

哈，仿佛是得到了游戏攻略一样，我们好像已经得到了一个推导算法时间复杂度的万能公式。可事实上，分析一个算法的时间复杂度，没有这么简单，我们还需要多看几个例子。

2.9.3 常数阶

首先顺序结构的时间复杂度。下面这个算法，也就是刚才的第二种算法（高斯算法），为什么时间复杂度不是 O(3)，而是 O(1)。

```
int sum = 0,n = 100;     /*执行一次*/
sum = （1+n）*n/2;        /*执行一次*/
printf（"%d", sum）;      /*执行一次*/
```

这个算法的运行次数函数是 f（n）=3。根据我们推导大 O 阶的方法，第一步就是把常数项 3 改为 1。在保留最高阶项时发现，它根本没有最高阶项，所以这个算法的

时间复杂度为 O(1)。

另外，我们试想一下，如果这个算法当中的语句 sum=（1+n）*n/2 有 10 句，即：

```
int sum = 0, n = 100;    /*执行 1 次*/
sum = （1+n）*n/2;        /*执行第 1 次*/
sum = （1+n）*n/2;        /*执行第 2 次*/
sum = （1+n）*n/2;        /*执行第 3 次*/
sum = （1+n）*n/2;        /*执行第 4 次*/
sum = （1+n）*n/2;        /*执行第 5 次*/
sum = （1+n）*n/2;        /*执行第 6 次*/
sum = （1+n）*n/2;        /*执行第 7 次*/
sum = （1+n）*n/2;        /*执行第 8 次*/
sum = （1+n）*n/2;        /*执行第 9 次*/
sum = （1+n）*n/2;        /*执行第 10 次*/
printf（"%d",sum）;       /*执行 1 次*/
```

事实上无论 n 为多少，上面的两段代码就是 3 次和 12 次执行的差异。这种与问题的大小无关（n 的多少），执行时间恒定的算法，我们称之为具有 O(1)的时间复杂度，又叫常数阶。

注意：不管这个常数是多少，我们都记作 O(1)，而不能是 O(3)、O(12)等其他任何数字，这是初学者常常犯的错误。

对于分支结构而言，无论是真，还是假，执行的次数都是恒定的，不会随着 n 的变大而发生变化，所以单纯的分支结构（不包含在循环结构中），其时间复杂度也是 O(1)。

2.9.4 线性阶

线性阶的循环结构会复杂很多。要确定某个算法的阶次，我们常常需要确定某个特定语句或某个语句集运行的次数。因此，我们要**分析算法的复杂度，关键就是要分析循环结构的运行情况。**

下面这段代码，它的循环的时间复杂度为 O(n)，因为循环体中的代码须要执行 n 次。

```
int i;
for（i = 0; i < n; i++）
```

```
{
    /*时间复杂度为O(1)的程序步骤序列*/
}
```

2.9.5 对数阶

下面的这段代码，时间复杂度又是多少呢?

```
int count = 1;
while （count < n）
{
    count = count * 2;
    /*时间复杂度为O(1)的程序步骤序列*/
}
```

由于每次count乘以 2 之后，就距离n更近了一分。也就是说，有多少个 2 相乘后大于n，则会退出循环。由 $2^x=n$得到$x=\log_2 n$。所以这个循环的时间复杂度为$O(\log n)$。

2.9.6 平方阶

下面例子是一个循环嵌套，它的内循环刚才我们已经分析过，时间复杂度为$O(n)$。

```
int i,j;
for（i = 0; i < n; i++）
{
    for （j = 0; j < n; j++）
    {
        /*时间复杂度为O(1)的程序步骤序列*/
    }
}
```

而对于外层的循环，不过是内部这个时间复杂度为$O(n)$的语句，再循环n次。所以这段代码的时间复杂度为$O(n^2)$。

如果外循环的循环次数改为了 m，时间复杂度就变为 $O(m \times n)$。

```
int i,j;
for（i = 0; i < m; i++）
{
    for （j = 0; j < n; j++）
```

```
    {
        /*时间复杂度为 O(1)的程序步骤序列*/
    }
}
```

所以我们可以总结得出，循环的时间复杂度等于循环体的复杂度乘以该循环运行的次数。

那么下面这个循环嵌套，它的时间复杂度是多少呢?

```
int i,j;
for（i = 0; i < n; i++）
{
    for （j = i; j < n; j++）  /*注意 int j = i 而不是 0*/
    {
        /*时间复杂度为 O(1)的程序步骤序列*/
    }
}
```

由于当 i = 0 时，内循环执行了 n 次，当 i = 1 时，执行了 n－1 次，……当 i = n－1 时，内循环执行了 1 次。所以总的执行次数为

$$n+(n-1)+(n-2)+\cdots+1=\frac{n(n+1)}{2}=\frac{n^2}{2}+\frac{n}{2}$$ 。

用我们推导大O阶的方法，第一条，没有加法常数不予考虑；第二条，只保留最高阶项，因此保留$n^2/2$；第三条，去除这个项相乘的常数，也就是去除 1/2，最终这段代码的时间复杂度为$O(n^2)$。

从这个例子，我们也可以得到一个经验，其实**理解大 O 推导不算难，难的是对数列的一些相关运算，这更多的是考察你的数学知识和能力，**所以想考研的朋友，要想在求算法时间复杂度这里不失分，可能需要强化你的数学，特别是数列方面的知识和解题能力。

我们继续看例子，对于方法调用的时间复杂度又如何分析。

```
int i,j;
for（i = 0; i < n; i++）
{
    function（i）;
}
```

上面这段代码调用一个函数 function。

```
void function(int count)
{
    print(count);
}
```

函数体是打印这个参数。其实这很好理解，function 函数的时间复杂度是 O(1)。所以整体的时间复杂度为 O(n)。

假如 function 是下面这样的：

```
void function(int count)
{
    int j;
    for (j = count; j < n; j++)
    {
        /*时间复杂度为O(1)的程序步骤序列*/
    }
}
```

事实上，这和刚才举的例子是一样的，只不过把嵌套内循环放到了函数中，所以最终的时间复杂度为O(n^2)。

下面这段相对复杂的语句：

```
n++;                          /*执行次数为1*/
function(n);                  /*执行次数为n*/
int i,j;
for(i = 0; i < n; i++)        /*执行次数为n²*/
{
    function (i);
}
for(i = 0; i < n; i++)        /*执行次数为n(n + 1)/2*/
{
    for (j = i;j < n; j++)
    {
        /*时间复杂度为O(1)的程序步骤序列*/
    }
}
```

它的执行次数 $f(n)=1+n+n^2+\frac{n(n+1)}{2}+\frac{3}{2}n^2+\frac{3}{2}n+1$，根据推导大O阶的方法，最终

这段代码的时间复杂度也是$O(n^2)$。

2.10 常见的时间复杂度

常见的时间复杂度如表 2-10-1 所示。

表 2-10-1

执行次数函数	阶	非正式术语
12	$O(1)$	常数阶
$2n+3$	$O(n)$	线性阶
$3n^2+2n+1$	$O(n^2)$	平方阶
$5\log_2 n+20$	$O(\log n)$	对数阶
$2n+3n\log_2 n+19$	$O(n\log n)$	$n\log_2 n$阶
$6n^3+2n^2+3n+4$	$O(n^3)$	立方阶
2^n	$O(2^n)$	指数阶

常用的时间复杂度所耗费的时间从小到大依次是：

$$O(1)<O(\log n)<O(n)<O(n\log n)<O(n^2)<O(n^3)<O(2^n)<O(n!)<O(n^n)$$

我们前面已经谈到了$O(1)$常数阶、$O(\log n)$对数阶、$O(n)$线性阶、$O(n^2)$平方阶等，至于$O(n\log n)$我们将会在今后的课程中介绍，而像$O(n^3)$，过大的n都会使得结果变得不现实。同样指数阶$O(2^n)$和阶乘阶$O(n!)$等除非是很小的n值，否则哪怕n只是 100，都是噩梦般的运行时间。所以这种不切实际的算法时间复杂度，一般我们都不去讨论它。

2.11 最坏情况与平均情况

你早晨上班出门后突然想起来，手机忘记带了，这年头，钥匙、钱包、手机三大件，出门哪样也不能少呀。于是回家找。打开门一看，手机就在门口玄关的台子上，原来是出门穿鞋时忘记拿了。这当然是比较好，基本没花什么时间寻找。可如果不是放在那里，你就得进去到处找，找完客厅找卧室、找完卧室找厨房、找完厨房找卫生间，就是找不到，时间一分一秒的过去，你突然想起来，可以用家里座机打一下手机，听着手机铃声来找呀，真是笨。终于找到了，在床上枕头下面。你再去上班，迟到。见鬼，这一年的全勤奖，就因为找手机给黄了。

找东西有运气好的时候，也有怎么也找不到的情况。但在现实中，通常我们碰到的绝大多数既不是最好的也不是最坏的，所以算下来是平均情况居多。

算法的分析也是类似，我们查找一个有 n 个随机数字数组中的某个数字，最好的情况是第一个数字就是，那么算法的时间复杂度为 $O(1)$，但也有可能这个数字就在最后一个位置上待着，那么算法的时间复杂度就是 $O(n)$，这是最坏的一种情况了。

最坏情况运行时间是一种保证，那就是运行时间将不会再坏了。在应用中，这是一种最重要的需求，**通常，除非特别指定，我们提到的运行时间都是最坏情况的运行时间。**

而平均运行时间也就是从概率的角度看，这个数字在每一个位置的可能性是相同的，所以平均的查找时间为 n/2 次后发现这个目标元素。

平均运行时间是所有情况中最有意义的，因为它是期望的运行时间。也就是说，我们运行一段程序代码时，是希望看到平均运行时间的。可现实中，平均运行时间很难通过分析得到，一般都是通过运行一定数量的实验数据后估算出来的。

对算法的分析，一种方法是计算所有情况的平均值，这种时间复杂度的计算方法称为平均时间复杂度。另一种方法是计算最坏情况下的时间复杂度，这种方法称为最坏时间复杂度。**一般在没有特殊说明的情况下，都是指最坏时间复杂度。**

2.12 算法空间复杂度

我们在写代码时，完全可以用空间来换取时间，比如说，要判断某某年是不是闰年，你可能会花一点心思写了一个算法，而且由于是一个算法，也就意味着，每次给一个年份，都是要通过计算得到是否是闰年的结果。还有另一个办法就是，事先建立一个有 2 050 个元素的数组（年数略比现实多一点），然后把所有的年份按下标的数字对应，如果是闰年，此数组项的值就是 1，如果不是值为 0。这样，所谓的判断某一年是否是闰年，就变成了查找这个数组的某一项的值是多少的问题。此时，我们的运算是最小化了，但是硬盘上或者内存中需要存储这 2050 个 0 和 1。

这是通过一笔空间上的开销来换取计算时间的小技巧。到底哪一个好，其实要看你用在什么地方。

算法的空间复杂度通过计算算法所需的存储空间实现，算法空间复杂度的计算公式记作：$S(n)= O(f(n))$，其中，n 为问题的规模，$f(n)$为语句关于 n 所占存储空间的函

数。

一般情况下，一个程序在机器上执行时，除了需要存储程序本身的指令、常数、变量和输入数据外，还需要存储对数据操作的存储单元。若输入数据所占空间只取决于问题本身，和算法无关，这样只需要分析该算法在实现时所需的辅助单元即可。若算法执行时所需的辅助空间相对于输入数据量而言是个常数，则称此算法为原地工作，空间复杂度为 O(1)。

通常，我们都使用“时间复杂度”来指运行时间的需求，使用“空间复杂度”指空间需求。当不用限定词地使用“复杂度”时，通常都是指时间复杂度。显然我们这本书重点要讲的还是算法的时间复杂度的问题。

2.13 总结回顾

不容易，终于又到了总结的时间。

我们这一章主要谈了算法的一些基本概念。谈到了数据结构与算法的关系是相互依赖不可分割的。

算法的定义：算法是解决特定问题求解步骤的描述，在计算机中为指令的有限序列，并且每条指令表示一个或多个操作。

算法的特性：有穷性、确定性、可行性、输入、输出。

算法设计的要求：正确性、可读性、健壮性、高效率和低存储量需求。

算法特性与算法设计容易混，需要对比记忆。

算法的度量方法：事后统计方法（不科学、不准确）、事前分析估算方法。

在讲解如何用事前分析估算方法之前，我们先给出了函数渐近增长的定义。

函数的渐近增长：给定两个函数 f（n）和 g（n），如果存在一个整数 N，使得对于所有的 n > N，f（n）总是比 g（n）大，那么，我们说 f（n）的增长渐近快于 g（n）。于是我们可以得出一个结论，判断一个算法好不好，我们只通过少量的数据是不能做出准确判断的，如果我们可以对比算法的关键执行次数函数的渐近增长性，基本就可以分析出：某个算法，随着 n 的变大，它会越来越优于另一算法，或者越来越差于另一算法。

然后给出了算法时间复杂度的定义和推导大 O 阶的步骤。

推导大 O 阶：

- 用常数 1 取代运行时间中的所有加法常数。
- 在修改后的运行次数函数中，只保留最高阶项。
- 如果最高阶项存在且不是 1，则去除与这个项相乘的常数。

得到的结果就是大 O 阶。

通过这个步骤，我们可以在得到算法的运行次数表达式后，很快得到它的时间复杂度，即大 O 阶。同时我也提醒了大家，其实推导大 O 阶很容易，但如何得到运行次数的表达式却是需要数学功底的。

接着我们给出了常见的时间复杂度所耗时间的大小排列：

$$O(1)<O(\log n)<O(n)<O(n\log n)<O(n^2)<O(n^3)<O(2^n)<O(n!)<O(n^n)$$

最后，我们给出了关于算法最坏情况和平均情况的概念，以及空间复杂度的概念。

2.14 结 尾 语

很多学生，学了四年计算机专业，很多程序员，做了很长时间的编程工作，却始终都弄不明白算法的时间复杂度的估算，这是很可悲的一件事。因为弄不清楚，所以也就从不深究自己写的代码是否效率低下，是不是可以通过优化让计算机更加快速高效。

他们通常的借口是，现在 CPU 越来越快，根本不用考虑算法的优劣，实现功能即可，用户感觉不到算法好坏造成的快慢。可事实真是这样吗？还是让我们用数据来说话吧。

假设CPU在短短几年间，速度提高了 100 倍，这其实已经很夸张了。而我们的某个算法本可以写出时间复杂度是$O(n)$的程序，却写出了$O(n^2)$的程序，仅仅因为容易想到，也容易写。即在$O(n^2)$的时间复杂度算法程序下，速度其实只提高了 10 倍（$\sqrt{100}=10$），而对于$O(n)$时间复杂度的算法来说，那才是真的 100 倍。

也就是说，一台老式CPU的计算机运行$O(n)$的程序和一台速度提高 100 倍新式CPU运行$O(n^2)$的程序。最终效率高的胜利方却是老式CPU的计算机，原因就在于算法的优劣直接决定了程序运行的效率。

也许你就可以深刻的感受到，愚公移山固然可敬，但发明炸药和推土机，可能更加实在和聪明（如图 2-14-1 所示）。

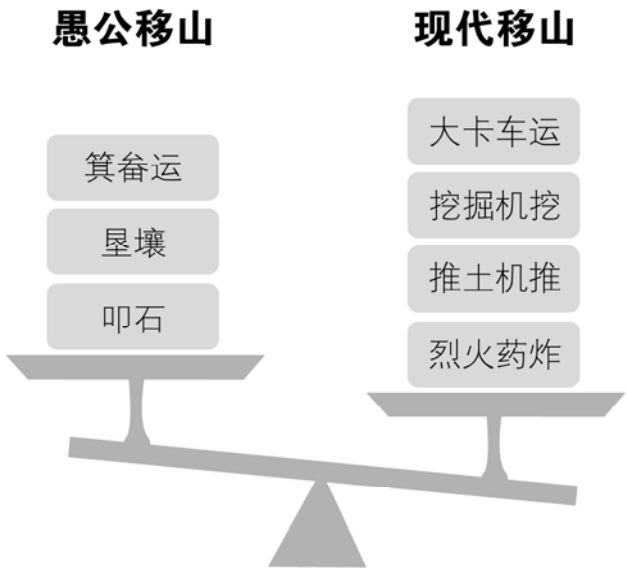

图 2-14-1

希望大家在今后的学习中，好好利用算法分析的工具，改进自己的代码，让计算机轻松一点，这样你就更加胜人一筹。

读书笔记