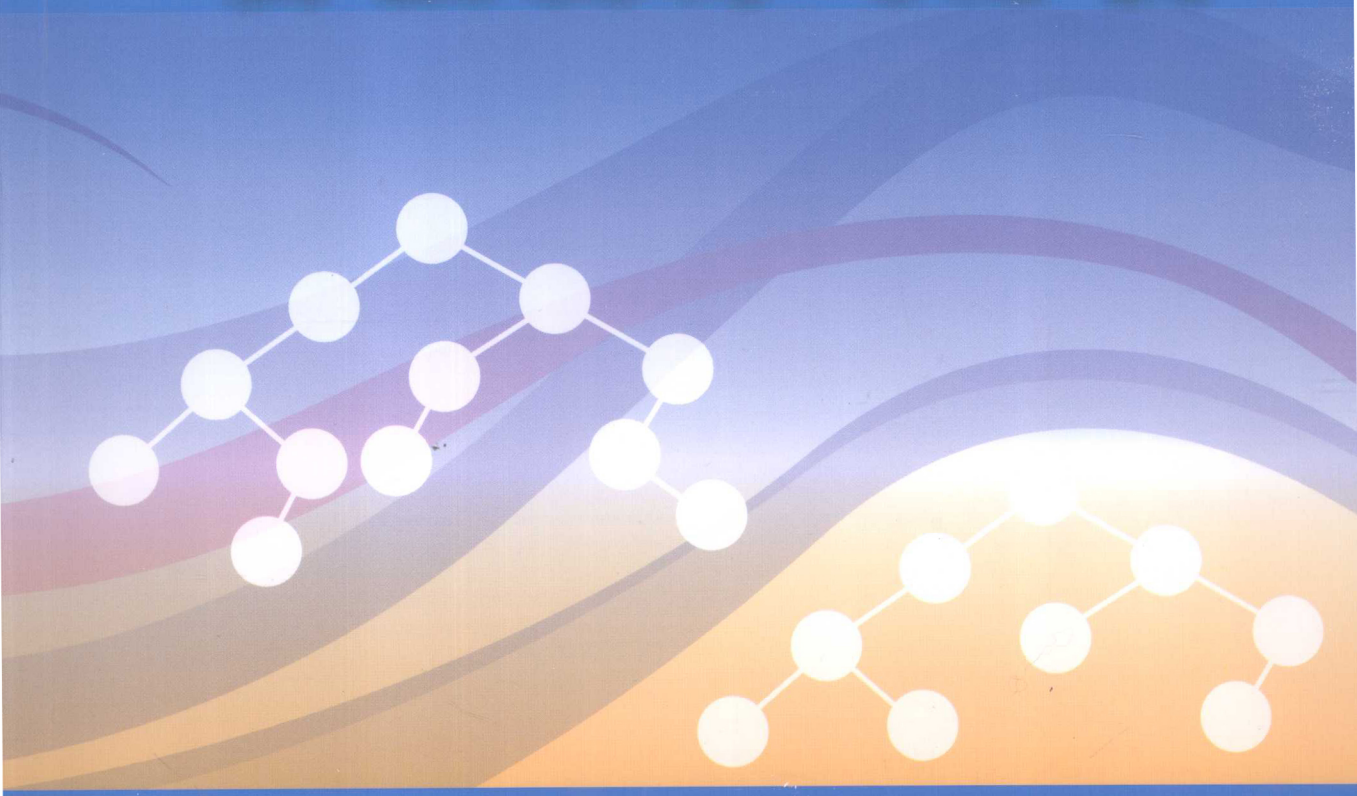
高等学校计算机专业规划教材

Algorithm Design and Analysis

算法设计与分析



徐义春 万书振 解德祥 编著

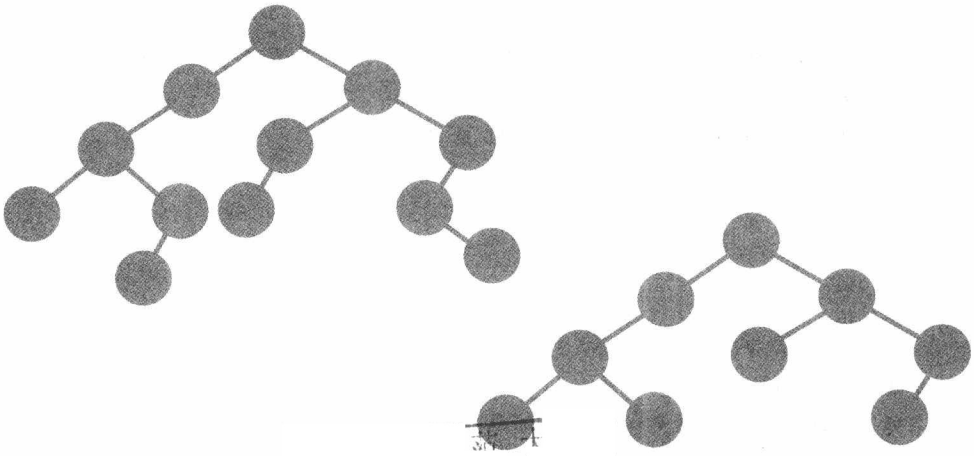
清华大学出版社

高等学校计算机专业规划教材

Algorithm Design and Analysis

算法设计与分析

徐义春 万书振 解德祥 编著



清华大学出版社
北京

内 容 简 介

本书涵盖了常见计算机算法设计和分析的思路和方法，内容包括算法概论、递推与递归、分治法、动态规划法、搜索方法、近似算法、随机算法等，最后提供一些高级数据结构的介绍，以帮助实现效率更高的算法。本书重视算法思路的总结以及方法的正确性证明，以深入浅出的方式引导学生学习教材内容，既具有严谨性，又具有简明性。全书为绝大多数算法提供了可以直接验证的 C/C++ 代码。

本书适合作为高等院校计算机相关专业的教材，也可作为编程竞赛的辅导用书。

图书在版编目(CIP)数据

算法设计与分析/徐义春，万书振，解德祥编著. —北京：清华大学出版社，2016
高等学校计算机专业规划教材
ISBN 978-7-302-43789-5

Ⅰ. ①算… Ⅱ. ①徐… ②万… ③解… Ⅲ. ①电子计算机－算法设计－高等学校－教材 ②电子计算机－算法分析－高等学校－教材 Ⅳ. ①TP301.6

中国版本图书馆 CIP 数据核字(2016)第 100188 号

责任编辑：龙启铭
封面设计：何凤霞
责任校对：梁　毅
责任印制：李红英

出版发行：清华大学出版社
网　　址：http://www.tup.com.cn，http://www.wqbook.com
地　　址：北京清华大学学研大厦 A 座　　**邮　　编**：100084
社 总 机：010-62770175　　**邮　　购**：010-62786544
投稿与读者服务：010-62776969，c-service@tup.tsinghua.edu.cn
质量反馈：010-62772015，zhiliang@tup.tsinghua.edu.cn
课件下载：http://www.tup.com.cn，010-62795954
印 刷 者：北京富博印刷有限公司
装 订 者：北京市密云县京文制本装订厂
经　　销：全国新华书店
开　　本：185mm×260mm　　**印　　张**：11.75　　**字　　数**：295 千字
版　　次：2016 年 8 月第 1 版　　**印　　次**：2016 年 8 月第 1 次印刷
印　　数：1～2000
定　　价：29.00 元

产品编号：069377-01

前言

21世纪以来，计算机技术日益进入人们的普通生活，硬件不断升级，软件功能越来越强大。算法作为软件技术的核心，在人们关心的应用中的重要性越来越突出。当前最热门的词汇应该算是互联网以及大数据，源于网络大量数据的处理需求，需要有快速有效的处理算法，这凸显了算法研究的重要性。

人工智能技术也取得了跨越式发展，继1997年IBM公司的深蓝计算机战胜人类国际象棋冠军卡斯帕罗夫后，2015年年底，Google公司的Deepmind团队研发的AlphaGo围棋程序，以五连胜的战绩，赢了欧洲围棋冠军樊麾。2016年3月，AlphaGo又以4∶1的战绩战胜世界冠军李世石。这些标志性的成果都源于计算机算法的显著进步。

人们已经认识到计算机算法的重要性。各大计算机技术企业在员工招聘中对算法人才极其青睐，微软、Google、华为、腾讯、阿里等国内外巨头纷纷赞助或者举办以算法为主的各类编程竞赛，中国计算机学会常年进行青少年信息学奥赛人才的选拔和培养，而ACM大学生程序设计竞赛更是吸引了全国各大高校参与。

在以上背景下，我们总结多年来在高校计算机算法教学中的经验，出版了这本教材。本教材涵盖了分治法、贪心法、动态规划法、深度优先与宽度优先搜索、近似方法与随机算法等经典算法，阐明了每一种算法的本质特点，简洁但严谨地证明了算法的正确性，并提供了典型的应用以及完整的C/C++程序实现。

学生在使用本教材时，应预先掌握“数据结构”课程的基础内容，包括线性结构、树结构和图结构的基本概念与应用。高性能的算法往往与数据结构密切相关，本教材在第10和第11章提供了若干关于高级数据结构的介绍，供读者参考。为了读者能顺利验证，本教材中提供的绝大多数的算法代码都能直接编译运行。

本教材第1～9章由徐义春撰写，第10章由解德祥撰写，第11章由万书振撰写。由于作者的水平有限，书中一定存在不少缺点，热忱欢迎各同行及读者批评指正。

目录

第 1 章 算法与性能

1.1 算法的概念

在计算机程序设计中，为了实现程序的特定功能，需要设计一组指令操作序列，而算法就是准确定义这组操作序列的规则集合。

例如，对于等差数列$\{a_1, a_2, \cdots, a_n\}$的求和问题，已知数列首元素为 a_1，满足 $a_i = a_{i-1} + d$，则前 n 个元素的和为

$$S(n) = na_1 + \frac{n(n-1)}{2}d$$

可以给出以下计算机算法：

(1) 输入初始值 a_1，差值 d，以及元素个数 n；

(2) 计算 $S = na_1 + \frac{n(n-1)}{2}d$；

(3) 输出 S 的值。

数据是算法的第一个重要构成元素。首先要从输入设备上读取数据，最后要将计算结果在输出设备上输出，而中间则要存储必要的数据以待处理。算法中的数据组织方式，通常称为“数据结构”。

操作的序列是算法的第二个重要构成元素。为完成算法的功能，需要设计特定操作序列，对数据进行变换。这些特定的操作序列，称为“控制流程”。对同一个计算问题，使用不同的数据结构和控制流程，可以得到不同的算法。

算法还有一个重要特征是经过有限步计算后停止计算并输出结果。一个程序可以始终不停机，例如操作系统可以始终运行，但不能称为一个算法。算法完成计算所需要的步数或者时间，是评价算法性能的一个指标。

1.2 算法的表达

算法的设计和交流必须使用某种语言工具。目前常用的算法表达工具有自然语言、结构化图形工具以及计算机高级语言等。

1.2.1 自然语言

算法可以直接使用人类交流的自然语言进行描述。例如求解两个整数 a 和 b 的最大

公约数(Greatest Common Divisor)的欧几里得算法 GCD(a, b),基于以下原理:

如果 $a>b$,那么 GCD(a, b) = GCD($a-b$, b),也就是把其中较大的一个数变为两数之差后,最大公约数不变。例如 GCD(36, 14) = GCD (36−14,14) = GCD(22, 14)。

将上述原理反复使用,最后就能得到最大公约数。用结构化的自然语言描述欧几里得算法如下:

算法 1.1 求解正整数 a, b 的最大公约数的欧几里得算法

1. 检查 a, b 两个数,如果 $a<b$ 则将其值交换。此步骤保证 $a\geqslant b$;
2. 当 a 不能被 b 整除时,重复执行:
   2.1 令 $a=a-b$;
   2.2 检查 a、b 两个数,如果 $a<b$ 则将其值交换;
3. 返回 b 的值,算法停止。

在算法 1.1 的第 2 步,反复利用欧几里得原理,对 a、b 的值进行变换,同时保证最大公约数不变。可以观察到 b 在不断减小,但 b 最小可能是 1,因此经过有限次变换后,算法必然中止。

假设在算法 1.1 对 a、b 进行变换时,同时观察算法运行情况,则在对 $a=36$,$b=14$ 的最大公约数求解过程中,可观察到如下的值:

(36,14), (22, 14),(14, 8),(8, 6),(6, 2)

在最后 2 能整除 6,因此算法停止第 2 步的循环,通过第 3 步返回最大公约数 2。

由于自然语言并不严格,使用者容易产生歧义,因此用自然语言描述算法,也可能造成理解上的错误。

1.2.2 结构化图形工具

表达算法的结构化图形设计工具有很多,比如流程图、N-S 图等。结构化设计工具将算法的控制流程分为三种:串行结构(图 1.1(a))、选择结构(图 1.1(b))和循环结构(包括当型循环(图 1.1(c))和直到型循环(图 1.1(d)))。

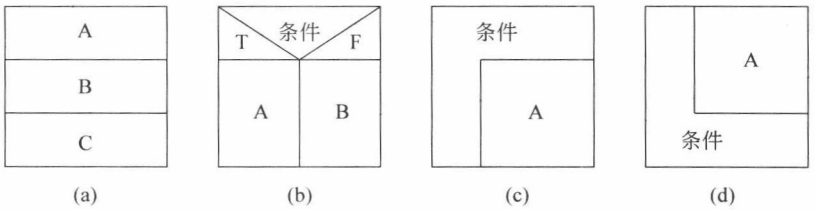


图 1.1 N-S 图的控制结构图:(a) 串行结构;(b) 选择结构;(c) 当型循环;(d) 直到型循环

在图 1.1(a)所示的串行结构的控制流程规定为自顶向下依次执行操作 A→B→C。图 1.1(b)所示的选择结构根据条件的逻辑值决定执行哪个操作,当条件为真(T)时执行 A,条件为假(F)时执行 B。图 1.1(c)所示的当型循环是反复测试条件,当条件满足时执行 A,否则退出循环。图 1.1(d)与图 1.1(c)的区别在于,图 1.1(d)先执行 A,然后测试条件是否满足,条件为真时继续重复执行 A。

图 1.2 是算法 1.1 的 N-S 图表达方法。可以看出,N-S 图在表达各种控制流程时,更

为直观和清晰。

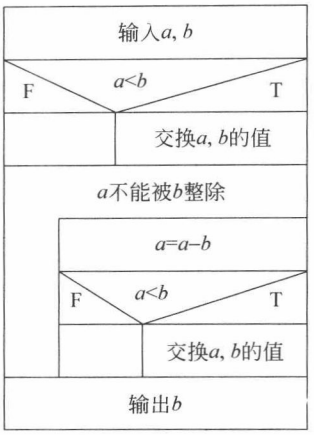


图 1.2　表达算法 1.1 的 N-S 图

1.2.3　计算机高级语言

计算机高级语言定义了严格的语法规则，不会产生二义性，也经常用来表达算法。例如 PASCAL 就是一种常用的描述算法的语言。使用高级语言描述算法的好处之一是，读者可以方便地在计算机上运行并验证算法。但是使用高级语言表达算法，需要算法作者和读者较熟悉该门语言。本书中的大部分算法都使用 C/C++ 语言描述。

算法 1.2　用 C 语言的描述方式为：

```
int GCD (int a,int b){
//输入 a,b 为正整数,输出 a,b 的最大公约数
   int x;
   if (a<b){ x=a; a=b; b=x; }
   while (a%b !=0){
     a=a-b;
     if (a<b){ x=a; a=b; b=x; }
   }
   return b;
}
```

1.3　算法的评价

19 世纪 30 年代，理论计算机科学家(如图灵、冯·诺依曼等)研究了一个计算问题是否可以用算法来解决，建立了可计算理论，奠定了计算机技术发展的基础。

但是，一个计算问题即使理论上存在求解算法，在实践上还是远远不够。例如国际象棋，每个棋子可以选择的位置是有限的，在规则约束下，经过有限步一定会有一个结果。完全可以设计一个算法，考虑己方的所有可能下法，以及对手的所有可能应着，从而选择

一个最佳下法。但是，随着考虑步数的深入，各种下法的可能性急剧增加（有人粗略估计将超过 10^{50}），如果要用计算机检测这些可能性要花几千年的时间。因此，通过检测所有的下法来获得最优下法的算法是不实用的。

以下从三个方面对算法进行评价，即算法的正确性、算法的空间复杂性、算法的时间复杂性。

1.3.1 算法的正确性

算法是用于求解问题的，因此首先必须保证算法能得到正确的计算结果，达到了算法设计的目标。一个输出错误结果的算法是没有意义的。

1.3.1.1 算法正确性证明

算法的正确性证明包括主要思想上的证明和实现上的证明。主要的算法思想必须有严格的逻辑支持，在数学上必须是正确的。实现上的证明则是保证在计算机语言实现算法时，不能引进错误。

要保证一个算法实现的正确性，也可以从理论上进行证明。算法的每一个操作指令，将使得数据间产生一些逻辑关系。例如“x=a;”执行之后，将满足“x == a”这个逻辑关系。如果算法最后得到了类似于“计算结果 == 预期结果”的逻辑关系，则算法的正确性得到了证明。

简单的操作指令产生的效果比较容易确认，重点需要关注循环结构中产生的效果。对循环结构效果的确认技术是寻找并证明循环中的不变量。**循环不变量**是指在循环开始和循环中每一次迭代时恒为真的逻辑关系。

在算法 1.1 中，假设 a，b 初始值的最大公约数是 g，即满足 g == GCD(a, b)。经过第 1 步，满足逻辑关系 R：g == GCD(a, b) and $a \geqslant b$。在第 2 步中，第 2.1 步根据欧几里得原理有 g == GCD(a, b)，而第 2.2 步保证了 $a \geqslant b$，因此在每一轮测试循环条件时，R 始终是满足的，因此 R 是循环不变量。

最后，在第 2 步的循环结束时，满足逻辑关系 g == GCD(a, b) and $a \geqslant b$ and b 整除 a，从而第 3 步输出的 b 是最大公约数，算法的正确性得到保证。

一个简单的算法实现的正确性可以通过上述证明完成，但一个较大的程序往往包含许多算法的组合，由人来进行证明负担较重，而且人的证明也往往会出错。一种解决思路是设计一个能证明算法正确的程序，但是在理论上这是不可行的。因为目前已经证明无法设计一个程序判断其他的程序是否停机，而最终停机是算法的基本要求。

1.3.1.2 算法的验证测试

实用的算法正确性验证方法是通过测试来进行。给定一组输入的数据，按照算法设计的目标，它应该输出一组正确的输出结果。如果输出结果不对，则表明算法设计有错。如果通过了这组数据测试，则算法正确的可能性就增加了一些。

例如算法 1.1，可以设计以下几组数据进行测试：

(1) a=20，b=16，最大公约数 4；

(2) a=12，b=18，最大公约数 6；

(3) $a=12$，$b=7$，最大公约数1。

一组测试输入数据和相应的输出数据称为测试用例。可以通过设计完备的测试用例集合，遍历所有的可能输入数据，来证明算法是正确的。但在现实中，往往输入数据的可能性太多，或者本来就有无穷多种组合，使得测试全部的输入组合是不可能完成的工作，因此只能通过设计一些有代表性的测试数据来测试算法。

由于测试数据的不完备性，通过了测试不能保证算法是正确的，只能说是提高了算法正确的可能性。如何设计有限的测试用例以高效地检验算法的正确性，是软件测试理论的研究内容，读者可以参考软件工程相关书籍。

1.3.2　算法的空间复杂性

算法的空间复杂性衡量的是算法设计时声明所需存储空间的大小，例如在算法1.2的C语言描述中，所定义的中间变量只有x，因此它的空间复杂性为常数1。

如果算法设计中需要保持的中间变量太多，需要占用过大的空间，操作系统会频繁地调度内外存，或者直接拒绝算法的运行，从而影响算法的运行性能。所以，空间资源的占用是评价一个算法好坏的性能指标之一。

1.3.3　算法的时间复杂性

算法的时间复杂性是对算法的时间耗费(或效率)的一种估计，是评价算法性能的重要指标。计算时间对有些计算机系统至关重要，例如在控制系统中，往往需要在较短的时间内完成解算。如果一个算法需要占用较长的运行时间，超出了设计的时间限制，即使是正确的，它也无法完成控制任务。

但是在进行算法分析的时候，具体的时钟时间却不是效率的一个好指标。首先，算法的具体运行时间与硬件平台有关系，同样的一个算法在不同的硬件平台上运行时间是不同的。其次，运行时间还涉及个人的编程风格以及所使用计算机语言的效率，例如一个算法用C语言实现相比用Java语言的程序，可能运行速度更快。

因此在算法理论中，是通过统计语句的执行次数来衡量算法的时间复杂性。在后面的叙述中，我们也称执行次数为执行时间。

下面对冒泡排序算法进行具体分析。

算法1.3　冒泡排序

```
int BubbleSort(int A[],int n){
//输入：数组A,元素数目n
//输出：数组A中元素完成从小到大排序
    int i,j,x;
1   for (i=0;i<n-1;i++)
2     for (j=0;j<n-i-1;j++)
3         if (A[j]>A[j+1]){
4             x=A[j]; A[j]=A[j+1]; A[j+1]=x ;
           }
    return 1;
  }
```

我们需要统计基本语句的执行次数。**基本语句**指一组语句(可能只包括一条语句),它们的执行时间是常数,可视为单位时间。如果一组语句的执行时间跟输入数据的规模相关,则不能作为基本语句,在估计时间复杂性时应考虑该组语句本身的复杂性。

在算法 1.3 中,语句 3 和 4 的执行时间是常数,可作为基本语句。它们的执行次数由两重循环语句 1 和 2 控制,外层循环 i 分别要取值 0,1,…,$n-2$,每个 i 值下,内存循环分别要执行 $n-i-1$ 次,因此基本语句总的执行次数为

$$\sum_{i=0}^{n-2}(n-i-1)=(n-1)+(n-2)+\cdots+1=n(n-1)/2 \tag{1.1}$$

如果算法中包含了多个循环段,则算法的时间复杂性为每个循环段的复杂性求和。如果算法语句中含有子函数或者子过程,则应先了解子函数或子过程的时间复杂性。例如下面一段代码对 $m\times n$ 的二维数组 A 的每一行排序,最后得到每行的最小元素和:

算法 1.4 矩阵最小元素和

```
int MinSum(int A[],int m,int n){
//输入:A 是二维数组的首地址,m 是行数,n 是列数
//输出:每行的最小元素和
  int row, sum;
1  sum=0;
2  for(row=0;row<m;row++) {
3     BubbleSort(&(A[row * n]), n);   //对每行排序
4     sum +=A[row * n];               //对每行的最小值求和
  }
  return sum;
}
```

在 MinSum 的算法中,语句 1 和 4 是基本语句,它们的执行次数分别是 1 和 m。由于语句 3 的执行时间是 $n(n-1)/2$,它与输入数据的规模 n 有关,因此不是基本语句。综合考虑,MinSum 的时间复杂性估计为

$$1+m+mn(n-1)/2 \tag{1.2}$$

1.4 最差时间复杂性和平均时间复杂性

1.3 节分析说明,冒泡排序算法 1.3 的执行时间为 $n(n-1)/2$,与输入数组的元素个数即输入数据的规模相关。

除了输入的规模外,具体的输入数据值也会影响算法的效率,例如一个简单查询算法:

算法 1.5 简单查询算法

```
int search(int A[], int n, int a){
//输入:数组 A,元素个数 n, 一个待查的值 a,
//输出:如果 a 不在表中输出-1,否则输出 a 在 A 中的位置
    int i;
```

```
1    for( i=0;i<n;i++)
2       if (a==A[i])return i;
    return -1;
}
```

在算法 1.5 中，初看起来循环语句 1 的执行次数是 n，语句 2 为基本操作。但仔细分析会发现，如果 a 在 A 中，则语句 2 的条件($a==A[i]$)为真时，算法会中途停止循环。如果 $A[0]=a$，第一次执行语句 2 算法就结束。只有在当 a 不在 A 中时，循环才会执行 n 次。

最差时间复杂性指在所有可能的输入中，算法的最长执行时间。I 表示一个规模为 n 的输入，以 $T(I)$ 表示分析出的执行时间，D 是所有规模为 n 的输入数据集合，则定义算法的最差时间复杂性为

$$w(n) = \max(T(I) \mid I \in D) \tag{1.3}$$

算法 1.4 的最差复杂性为 $w(n)=n$。

如果给出了算法的输入数据 I 的分布特征，即 I 的出现概率为 $p(I)$，则可以计算出算法的**平均时间复杂性**为

$$A(n) = \sum_{I \in D} T(I) \cdot p(I) \tag{1.4}$$

在算法 1.4 中，假定要求被查询数组 A 的元素互异，搜索失败的概率是 0.2，而每个元素被搜索的概率都是相同的，则算法 1.4 的平均时间复杂性为

$$A(n) = 0.2n + 0.8/n(1 + 2 + \cdots + n) = 0.6n + 0.4$$

最差时间复杂性和平均时间复杂性都是评价算法的指标，在不同的应用中可能起不同的作用。例如，个人机交互系统中的算法，平均时间复杂性可以反映用户的体验。而在一个实时系统中，如果算法的超时会带来灾害性后果，则应使用最差复杂性进行算法评价。

1.5 函数的阶与渐进性分析

1.5.1 复杂性函数的阶

一般情况下，算法的输入规模用正整数 n 表示，算法的复杂性是一个关于 n 的递增函数 $T(n)$。

在算法分析理论中，我们更关注**复杂性函数的阶**。例如一个问题有两个算法 A 和 B，其时间复杂性函数分别为 $T_A(n)=n$，$T_B(n)=2n$，观察两个算法的运行时间之比：

$$r = \frac{T_A(n) \cdot t_A}{T_B(n) \cdot t_B} = \frac{nt_A}{2nt_B} = \frac{t_A}{2t_B}$$

其中，t_A 和 t_B 是算法 A、B 的基本操作所需时间。上式表明两个算法的运行时间之比是一个常数，与输入无关。如果算法 A 执行更快，则可以通过更换硬件等方式，改变基本语句的执行时间，使算法 B 的执行速度赶上或超过算法 A。我们认为这两个算法的时间复杂性具有相同的阶。