《算法设计与分析》
第1章 算法及基础知识(1)
—定义与分析基础

山东师范大学信息科学与工程学院
段会川
2014年9月

目录

教材—简介

- 《算法设计与分析》
- 王秋芬、吕聪颖、周春光编著
  - 南阳理工学院
- 21世纪高等学校规划教材—计算机科学与技术
- 清华大学出版社
- 2011年8月
- ￥33.00, 315页
- 清华大学文泉书局提供电子版，￥23.10

教材—目录

上课效果保证

- 教材
- 笔
- 幻灯片讲义
- 在幻灯片讲义和教材上做笔记

算法的重要性

有两种思想，像珠宝商放在天鹅绒上的宝石一样熠熠生辉，一个是微积分；另一个就是算法。微积分以及在微积分基础上建立起来的数学分析体系造就了现代科学，而算法则造就了现代世界。

——David Berlinski 之《THE ADVENT OF THE ALGORITHM》

算法的重要性

- 算法是计算机科学与技术的根基之一
- 在计算机专业教育中占有很重要的地位
- 对算法的学习和研究主要包括：
  - 算法设计、算法描述、正确性证明、分析和验证

另外，算法的设计与分析是计算机专业教育的核心问题，掌握算法的设计策略和算法分析的基本方法是对一个软件工作者的基本要求，为此，本书主要对这两大方面进行研究。设计策略是指面对一个问题，如何设计一个正确有效的算法；算法分析是指对于一个已设计的算法，如何评价其优劣。二者相互依存，设计出的算法需要进行分析和评价，对算法的分析和评价反过来又将推动算法设计的改进。

1.1.1 学习算法的重要性

1. 算法与日常生活息息相关
2. 算法是程序设计的根基
3. 学习算法能够提高分析问题的能力
4. 算法是推动计算机科学和产业发展的关键
5. 研究算法是件快乐的事情

1.1.2 算法的定义及特性

推动算法传播的是生活在美索不达米亚的 Al Khwarizmi，他于 9 世纪以阿拉姆语著述了一本教科书。该书列举了加、减、乘、除、求平方根和计算圆周率数值的方法。这些计算方法的特点是：简单、没有歧义、机械、有效和正确，这就是算法。注意，这个定义加上了“正确”一词。几百年后，当十进制记数法在欧洲被广泛使用时，“算法”(Algorithm)这个单词被人们创造出来以纪念 Al Khwarizmi 先生。当代著名计算机科学家 D. E. Knuth 在他撰写的《THE ART OF COMPUTER PROGRAMMING》一书中写到：“一个算法，就是一个有穷规则的集合，其中之规则规定了一个解决某一特定类型的问题的运算序列。”

1.1.2 算法的定义及特性

- 算法是求解某一特定问题的一组有穷规则的集合
  - 有穷性：执行有限条指令后结束
  - 确定性：指令无歧义，相同的输入总能得到相同的结果
  - 输入：0或1个
  - 输出：1或多个
  - 可行性：每个运算均可由人用笔和纸在有限时间内完成

唐纳德•克努特
Donald Knuth
(1938--)
《计算机程序设计的艺术》
《The Art of Computer Programming》
1974年图灵奖

The Art of Computer Programming

1.1.3 算法的描述方式—概述

算法的设计者在构思和设计一个算法之后，必须清楚准确地将所设计的求解步骤记录下来，这就是算法描述。

算法可以使用各种不同的方式来描述，常用的描述方式有自然语言、图形、程序设计语言和伪代码等。下面以欧几里得算法为例逐一介绍它们的具体操作方法。

欧几里得算法的功能是：求两个非负整数 a 和 b 的最大公约数。通常采用辗转相除法来实现该功能，其过程为：当 b 不为 0 时，辗转用操作：$r=a\%b, a=b, b=r$ 消去相同的因子，直到 b 为 0 时，a 的值即为所求的解。

1.1.3 算法的描述方式—GCD算法

- 欧几里德最大公约数算法
  - Greatest Common Divisor (GCD)
  - gcd(a,0) = a
  - gcd(a,b) = gcd(b, a MOD b)
  - 系少数几个古老而直到现代都很有用的算法

- 欧几里德(Euclid, 325-265 BC)
  - 古希腊数学家，几何学之父
- 欧几里德算法的复杂度: $O(n^3)$
  - 数字有关的算法输入规模是数字的二进制位数
    - n位二进制数的范围: $0 \sim 2^n-1$

1.1.3 GCD算法—自然语言描述

自然语言也就是人们日常进行交流的语言，如汉语、英语等。其最大的优点是简单、通俗易懂，缺点是不够严谨、烦琐且不能被计算机直接执行。

欧几里得算法用自然语言描述如下：

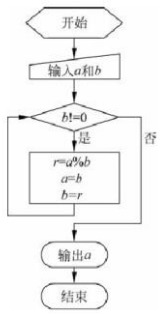
(1) 输入 a 和 b。

(2) 判断 b 是否为 0，如果不为 0，转步骤(3)；否则转步骤(4)。

(3) a 对 b 取余，其结果赋值给 r，b 赋值给 a，r 赋值给 b，转步骤(2)。

(4) 输出 a，算法结束。

1.1.3 GCD算法—流程图描述



1.1.3 GCD算法—程序设计语言描述

```
class OJLD{
    int a,b;                    //a 和 b 是私有数据成员
  public:
    OJLD(int m,int n)           //OJLD 是带两个整型参数的构造函数
      {a=m;b=n;}
    void zzf( )                 //zzf 是类的成员函数,用于计算 a 和 b 的最大公约数
        {
        int r;
        while(b!=0)
          {r=a%b;a=b;b=r;}
        printf("%d\n",a);
        }
  };                            //类 OJLD 的定义结束
```


1.1.3 GCD算法—伪代码描述

为了解决理解与执行这两者之间的矛盾，人们常常使用一种称为伪代码语言的描述方式来对算法进行描述。伪代码是介于自然语言与程序设计语言之间的一种用文字和符号结合的算法描述工具，它忽略了程序设计语言中一些严格的语法规则与描述细节，因此它比程序设计语言更容易描述和被人理解；它比自然语言更接近程序设计语言，因而较容易转换为能被计算机直接执行的程序。为此，对于计算机专业的初学者或非计算机人士来说，使用伪代码来描述算法倒是一个不错的选择。

欧几里得算法用伪代码描述如下：

```
Begin{算法开始}
   Step1: input a,b;
   Step2: if (b 不等于 0) 执行 Step3,否则执行 Step4;
   Step3: r=a%b,a=b,b=r,转 Step2;
   Step4: print a;
End{算法结束}
```


1.1.3 算法的伪代码描述标准

- 说明中文和简化的英文名称，必要时标上序号
- 说明输入和输出
- 顺序标记行号
- 赋值语句：使用=或左箭头进行变量赋值
- 变量可以带有下标，如：x_1, y_i
- 使用方括号表示数组和元素的下标
- 变量可以使用数学符号，如：$\alpha, \emptyset$
- 可以有数学公式表示的运算，如：$x = \frac{b}{a^2}$
- 可用简化的自然语言描述操作，如：交换a和b的值
- 可以使用直观可理解的符号，如：$a \leftrightarrow b$
- 可以使用直观可理解的数学函数，如：$\sin\theta$

1.1.3 算法的伪代码描述标准

- 使用\\书写注释，如：swap(a,b) \\交换*a*和*b*的值
- 使用return结束算法，必要时返回结果
- 使用if语句表示分支，可以带有then, else和else if
- for循环：for i=1 to n step 2 \\step为1时可以省略
- while循环
- do … while循环
- repeat … until循环
- 复合语句
  - 分支或循环体有一个语句时，可以不用复合语句标识
  - 可以使用{ … }作为复合语句界定符
  - 可以使用begin … end作为复合语句界定符
  - 可以使用end if, end for, end while作为复合语句界定符
  - 使用缩格表示层次关系

1.1.3 欧几里得GCD算法的伪码描述

- 算法1-1：欧几里得最大公约数算法(GCD)
- 输入：整数a, b
- 输出：a, b的GCD
- 1: while b≠0
- 2: c = a%b, a=b, b=c
- 3: end while
- 4: print a

1.2 算法设计的一般过程

1. 充分理解要解决的问题
2. 数学模型拟制
3. 算法详细设计
4. 算法描述
5. 算法思路的正确性验证
6. 算法分析
7. 算法的程序实现和测试
8. 文档资料的编制

目录

1.3.1 算法分析的概念

算法的复杂性指的是算法在运行过程中所需要的计算机资源的量，算法分析就是对该量的多少进行分析。所需资源的量越多，表明该算法的复杂性越高，反之，算法的复杂性越低。计算机的资源最重要的是运行算法时所需的时间，存储程序和数据所需的空间。因而，算法分析是对时间复杂性和空间复杂性进行分析。

算法分析对算法的设计、选用和改进有着重要的指导意义和实用价值：①对于任意给定的问题，设计出复杂性尽可能低的算法是在设计时考虑的一个重要目标；②当给定的问题已有多种算法时，选择复杂性最低者是在选用算法时应遵循的一个重要准则；③算法分析有助于对算法进行改进。

1.3.2 时间复杂性—事后统计法

因为很多计算机内部都有计时功能，有的甚至可以精确到毫秒，所以采用不同算法设计出的程序可通过一组或若干组相同的统计数据以分辨优劣。

但该方法有两种缺陷：一是必须先运行依据算法编写的程序；二是所得时间的统计量依赖于计算机的硬件、软件等环境因素，有时会掩盖算法本身的优劣。因此，人们常常不采用该方法进行时间复杂性分析。

1.3.2 时间复杂性—事前分析估算法

与算法的运行时间相关的因素通常有问题的规模、算法的输入序列、算法选用的设计策略、编写程序的语言、编译程序产生的机器代码的质量及计算机执行指令的速度等。

显然，在各种因素都不能确定的情况下，将很难估算出算法的运行时间，可见使用运行算法的绝对时间来衡量算法的效率是不现实的。如果撇开这些与计算机硬、软件有关的因素，一个特定算法的运行时间依赖于问题的规模（通常用正整数 n 表示）和它的输入序列 I。因此，算法的运行时间可表示为二者的函数，记为 $T(n,I)$。

1.3.2 时间复杂性—事前分析估算法

通常情况下，人们只考虑三种情况下的时间复杂性，即最坏情况、最好情况和平均情况，并分别记为 $T_{max}(n)$、$T_{min}(n)$ 和 $T_{avg}(n)$。设 D_n 是问题规模为 n 的算法的所有合法输入序列集合；I^0 是使算法的时间效率达到最差的合法输入序列集合；I^1 是使算法的时间效率达到最好的合法输入序列集合；$P(I)$ 是算法在应用中出现输入序列 I 的概率。在数学上有

$$T_{max}(n) = \max_{I\in D_n} T(n,I) = T(n,I^0)$$

$$T_{min}(n) = \min_{I\in D_n} T(n,I) = T(n,I^1)$$

$$T_{avg}(n) = \sum_{I\in D_n} T(n,I)P(I)$$

由此，针对特定的输入序列，算法的时间复杂性只与问题的规模 n 有关。一个不争的事实是：几乎所有的算法，规模越大所需的运行时间就越长。当 n 不断变化时，运行时间也会不断变化，故人们通常将算法的运行时间记为 $T(n)$。

1.3.2 时间复杂性—线性查找示例

- 算法1-2：线性查找(linearsearch)
- 输入：n个元素的数组a和带查找的元素x
- 输出：x在a中的位置，0表示未找到
- 1: for i=1 to n
- 2: if x=a[i] then
- 3: return i
- 4: return 0

1.3.3 算法的空间复杂性

- 由于把数据写入到每一个存储单元，至少需要一个特定的时间间隔，因此，算法的空间复杂性不会超过其时间复杂性，即：
  - $S(n) = O(T(n))$
  - 如果不特别指明，说到算法的复杂性都是指算法的时间复杂性
  - 如果同一个问题的不同算法时间复杂性相同，这时空间复杂性就非常重要了

目录

1.3.4 渐近复杂性态—引入

假设算法 A 的运行时间表达式 $T_1(n)$ 为：

$$T_1(n) = 30n^4 + 20n^3 + 40n^2 + 46n + 100 \quad (1\text{-}1)$$

算法 B 的运行时间表达式 $T_2(n)$ 为：

$$T_2(n) = 1000n^3 + 50n^2 + 78n + 10 \quad (1\text{-}2)$$

显然，当问题的规模足够大的时候，例如 $n=100$ 万，算法的运行时间将主要取决于时间表达式的第一项，其他项的执行时间只有它的几十万分之一，可以忽略不计。第一项的常数，随着 n 的增大，对算法的执行时间也变得不重要了。

于是，算法 A 的运行时间可以记为：$T_1^*(n)\approx n^4$，称 n^4 为 $T_1^*(n)$ 的阶。

同理，算法 B 的运行时间可以记为：$T_2^*(n)\approx n^3$，称 n^3 为 $T_2^*(n)$ 的阶。

1.3.4 渐近复杂性态—引入

由上述分析可以得出一个结论：随着问题规模的增大，算法的时间复杂性主要取决于运行时间表达式的阶。如果要比较两个算法的效率，只需比较它们的阶就可以了。

定义 1 设算法的运行时间为 $T(n)$，如果存在 $T^*(n)$，使得

$$\lim_{n\to\infty}\frac{T(n)-T^*(n)}{T(n)}=0$$

就称 $T^*(n)$为算法的渐进性态或渐进时间复杂性。

可见，问题规模充分大时，$T(n)$和 $T^*(n)$近似相等。因此，在算法分析中，对算法的时间复杂性和算法的渐进时间复杂性往往不加区分，并常用后者来对一个算法的时间复杂性进行衡量，从而简化了大规模问题的时间复杂性分析。

1.3.4 渐近复杂性态—渐近上界O

(1) 渐近上界记号：O(big-oh)。

定义 2 若存在两个正常数 c 和 n_0，使得当 $n\geqslant n_0$ 时，都有 $T(n)\leqslant cf(n)$，则称 $T(n)=O(f(n))$，即 $f(n)$是 $T(n)$的上界。换句话说，在 n 满足一定条件的范围内，函数 $T(n)$的阶不高于函数 $f(n)$的阶。

这就是说当$n>n_0$时，f增长的速度不超过g的某个常数倍

【例 1-1】 用 O 表示 $T(n)=10n+4$ 的阶。

存在 $c=11,n_0=4$，使得当 $n\geqslant n_0$ 都有：

$$T(n)=10n+4\leqslant 10n+n=11n$$

令 $f(n)=n$，可得

$$T(n)\leqslant cf(n)$$

即 $T(n)=O(f(n))=O(n)$。

应该指出，根据符号 O 的定义，用它评估算法的复杂性得到的只是问题规模充分大时的一个上界。这个上界的阶越低，则评估就越精确，结果就越有价值。如果有一个新的算法，其运行时间的上界低于以往解同一问题的所有其他算法的上界，就认为建立了一个解该问题所需时间的新上界。

1.3.4 渐近复杂性态—常见的复杂度类

常见的几类时间复杂性有：

$O(1)$：常数阶时间复杂性。它的基本运算执行的次数是固定的，总的时间由一个常数来限界，此类时间复杂性的算法运行时间效率最高。

$O(n)$、$O(n^2)$、$O(n^3)$、…：多项式阶时间复杂性。大部分算法的时间复杂性是多项阶的，通常称这类算法为多项式时间算法。$O(n)$称为 1 阶时间复杂性，$O(n^2)$称为 2 阶时间复杂性，$O(n^3)$称为 3 阶时间复杂性…。

$O(2^n)$、$O(n!)$和 $O(n^n)$：指数阶时间复杂性。这类算法的运行效率最低，这种复杂性的算法根本不实用。如果一个算法的时间复杂性是指数阶的，通常称这个算法为指数时间算法。

$O(n\log n)$和 $O(\log n)$：对数阶时间复杂性。除常数阶时间复杂性以外，它的效率最高。

以上几种复杂性的关系为：

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

1.3.4 渐近复杂性态—运算规则

另外，按照 O 的定义，容易证明如下运算规则成立，这些规则对后面的算法分析是非常有用的。

① $O(f)+O(g)=O(\max(f,g))$。

② $O(f)+O(g)=O(f+g)$。

③ $O(f)O(g)=O(fg)$。

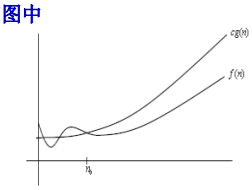
④ 如果 $g(n)=O(f(n))$，则 $O(f)+O(g)=O(f)$。

⑤ $O(Cf(n))=O(f(n))$，其中 C 是一个正的常数。

⑥ $f=O(f)$。

1.3.4 渐近复杂性态—渐近上界举例

- $f=O(g(n))$可以形象地表示在下图中
- $10n=O(n)$
  - 取$n_0=1$，$c=11$，则$n>n_0$时，有$10n\leq c\cdot n$
- $0.1n=O(n)$
  - 取$n_0=1$，$c=1$，则$n>n_0$时，有$0.1n\leq c\cdot n$
- $n+1000=O(n)$
  - 取$n_0=1000$，$c=2$，则$n>n_0$时，有$n+1000\leq c\cdot n$
- $3n^2+100n=O(n^2)$
  - 取$n_0=33$，$c=6$，则$n>n_0$时，有$3n^2+100n\leq c\cdot n^2$



1.3.4 渐近复杂性态—大O极限判断准则

- 根据$f(n)=O(g(n))$的定义，可以推得如下的极限判断准则：
  - 当$n\to\infty$时，$f(n)=O(g(n))$，当且仅当$\lim\limits_{n\to\infty}\frac{f(n)}{g(n)}<\infty$
  - $\lim\limits_{n\to\infty}\frac{10n}{n}=10<\infty\Rightarrow 10n=O(n)$
  - $\lim\limits_{n\to\infty}\frac{0.1n}{n}=0.1<\infty\Rightarrow 0.1n=O(n)$
  - $\lim\limits_{n\to\infty}\frac{n+1000}{n}=1<\infty\Rightarrow n+1000=O(n)$
  - $\lim\limits_{n\to\infty}\frac{3n^2+100n}{n^2}=3<\infty\Rightarrow 3n^2+100n=O(n^2)$

1.3.4 渐近复杂性态—大O的渐近紧界

- 显然
  - $3n = O(n)$、$3n = O(n^2)$、$3n = O(n^3)$
- 而$3n = O(n)$显然比后两者更有意义
  - 因此*f(n)=O(g(n))*中的*g(n)*通常取对*f(n)*最紧的界
  - 后两者的情况在数学上称为是平凡的(trivial)
- 非平凡(non-trivial)的g(n)一般满足如下极限
  - $0 < \lim_{n\to\infty} \frac{f(n)}{g(n)} < \infty$
  - 但有时找不到这样的$g(n)$
- 平凡的g(n)一般满足如下极限
  - $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$

1.3.4 渐近复杂性态—复杂度类

- $f(n) = O(g(n))$中的“=”号与一般数学中“=”号的意义是不同的
  - 此处“=”号可理解为“is”：*f(n)*是*O(g(n))*的
  - 因此，更易理解的表示法是：$f(n) \in O(g(n))$
  - 也就是说$g(n)$是一个函数类，更确切地说是一个复杂度类
  - 因而，我们可以说，对于任意的实数*a*、*b*且*a*>0，任何的$f(n) = an + b$都是属于$O(n)$(类)的，或者说都是复杂度为$O(n)$的
  - 更广泛地，我们有$\sum_{i=0}^{k} a_i n^k = O(n^k)$，当$a_k > 0$时

1.3.4 渐近复杂性态—上界与阶

- $f(n) = O(g(n))$还可做如下理解
  - 由于可能有$g(n) < f(n)$，直接将$g(n)$理解为$f(n)$的“上界”有违常理，较合理的理解为：$g(n)$是$f(n)$在某个常数倍意义上的“渐近上界”
  - 也可以将$g(n)$理解为“阶”的概念，即$f(n) = O(g(n))$表示复杂度$f(n)$是不超过$g(n)$阶的。
    非正式地，也将其说成是：$f(n)$是阶为$g(n)$的复杂度

1.3.4 渐近复杂性态—渐近下界Ω

(2) 渐近下界记号：Ω(big-omege)。

定义 3 若存在两个正常数 c 和 n_0，使得当 $n \geqslant n_0$ 时，都有 $T(n) \geqslant cf(n)$，则称 $T(n) = \Omega(f(n))$，即 $f(n)$ 是 $T(n)$ 的下界。换句话说，在 n 满足一定条件的范围内，函数 $T(n)$ 的阶不低于函数 $f(n)$ 的阶。它的概念与 O 的概念是相对的。

【例 1-2】 用 Ω 表示 $T(n) = 30n^4 + 20n^3 + 40n^2 + 46n + 100$ 的阶。

存在 $c = 30, n_0 = 1$，使得当 $n \geqslant n_0$ 都有

$$T(n) \geqslant 30n^4$$

令 $f(n) = n^4$，可得

$$T(n) \geqslant cf(n)$$

即 $T(n) = \Omega(f(n)) = \Omega(n^4)$。

1.3.4 渐近复杂性态—渐近精确界Θ

(3) 渐近精确界记号：Θ(big-theta)。

定义 4 若存在三个正常数 c_1、c_2 和 n_0，使得当 $n \geqslant n_0$ 时，都有 $c_2 f(n) \leqslant T(n) \leqslant c_1 f(n)$，则称 $T(n) = \Theta f(n)$。Θ 意味着在 n 满足一定条件的范围内，函数 $T(n)$ 和 $f(n)$ 的阶相同。由此可见，Θ 用来表示算法的精确阶。

【例 1-3】 用 Θ 表示 $T(n) = 20n^2 + 8n + 10$ 的阶。

定理 1 若 $T(n) = a_m n^m + a_{m-1} n^{m-1} + \cdots + a_1 n + a_0 (a_i > 0, 0 \leqslant i \leqslant m)$ 是关于 n 的一个 m 次多项式，则 $T(n) = O(n^m)$，且 $T(n) = \Omega(n^m)$，因此有 $T(n) = \Theta(n^m)$。

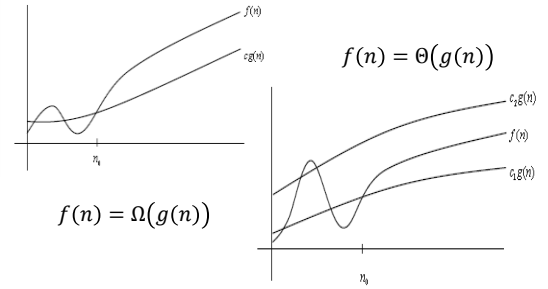
1.3.4 渐近复杂性态—大Ω及其极限定义法

- 大Ω也有相应的极限定义法：
  - $f(n) = \Omega(g(n))$，当且仅当 $\lim_{n\to\infty} \frac{f(n)}{g(n)} \neq 0$
  - 如：$3n + 2 = \Omega(n)$
  - $3n^2 + 100n = \Omega(n^2)$
  - $3n^2 + 100n = \Omega(n)$

1.3.4 渐近复杂性态—大Θ及其极限定义法

- 大Θ的极限定义法
  - $f(n)=\Theta(g(n))$，当且仅当$0<\lim_{n\to\infty}\frac{f(n)}{g(n)}=C<\infty$
  - 如：$3n+2=\Theta(n)$
  - $3n^2+100n=\Theta(n^2)$
- 实际上，
  - 当同时满足$f(n)=O(g(n))$和$f(n)=\Omega(g(n))$时，$f(n)$与$g(n)$具有相同的阶，
  - 即$f(n)=\Theta(g(n))$.

1.3.4 渐近复杂性态—大Ω与大Θ



1.3.4 渐近复杂性态—大O记法的来历

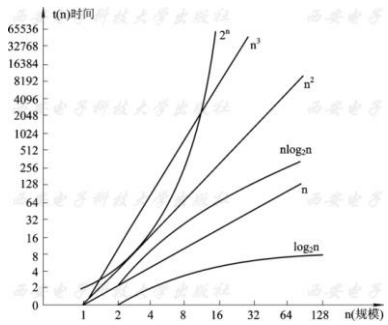
- 大O记法是由德国数论学家保罗 巴赫曼（Paul Bachmann，1837—1920）在其1892年的著作《解析数论》（Analytische Zahlentheorie）首先引入的。
- 另一位德国数论学家艾德蒙 朗道（Edmund Landau，1877—1938）在其著作中对大O记法进行了推广。
- O取自于“order of ...”（……阶），最初是一个大写的希腊字母'O'（Omicron）。
- 大O 、大Ω与大Θ是朗道—巴赫曼记号集中的主要成员

1.3.4 渐近复杂性态—常见的渐近函数阶

符号	名称
$O(1)$	常数（阶，下同）
$O(\log^* n)$	迭代对数
$O(\log n)$	对数
$O[(\log n)^c]$	多对数
$O(n)$	线性，次线性
$O(n\log n)$	线性对数，或对数线性、拟线性、超线性
$O(n^2)$	平方
$O(n^c), \text{Integer}(c>1)$	多项式，有时叫作“代数”（阶）
$O(c^n)$	指数，有时叫作“几何”（阶）
$O(n!)$	阶乘，有时叫做“组合”（阶）

$\log^2 n=\log(\log n)=\log\log n$

1.3.4 渐近复杂性态—常见的渐近函数曲线



1.3.4 渐近复杂性态—常见的复杂性类

- 常数类：$C=\Theta(1)$
- 线性类：$an+b=\Theta(n), a>0$
- 二次类：$an^2+bn+c=\Theta(n^2), a>0$
- 多项式类：$\sum_{i=0}^{k}a_in^k=O(n^k), a_k>0$
- 指数阶：$\Theta(2^n)$
- 对数阶：$\log n^k=\Theta(\log n)$
- 阶乘的对数：$\log n!=\Theta(n\log n)$
  - 如无特别说明，复杂性类中的log函数都是以2为底的

1.3.4 渐近复杂性态—$T(n)$建立的依据

如1.3.2节所述可知，要想精确地表示出算法的运行时间是很困难的。考虑到算法分析的主要目的是比较求解同一个问题的不同算法的效率。因此，在算法分析中只是对算法的运行时间进行粗略估计，得出其增长趋势即可，而不必精确计算出具体的运行时间。

(1) 非递归算法中 $T(n)$建立的依据。

为了求出算法的时间复杂性，通常需要遵循以下步骤：

① 选择某种能够用来衡量算法运行时间的依据。

② 依照该依据求出运行时间 $T(n)$的表达式。

③ 采用渐进符号表示 $T(n)$。

④ 获得算法的渐进时间复杂性，进行进一步的比较和分析。

其中，步骤①是最关键的，它是其他步骤能够进行的前提。通常衡量算法运行时间的依据是基本语句，所谓基本语句是指对算法的运行时间贡献最大的原操作语句。

1.3.4 渐近复杂性态—$T(n)$建立的依据示例

当算法的时间复杂性只依赖问题规模时，基本语句选择的标准是：必须能够明显地反映出该语句操作随着问题规模的增大而变化的情况，其重复执行的次数与算法的运行时间成正比，多数情况下是算法最深层循环内的语句中的原操作；对算法的运行时间贡献最大，在解决问题时占支配地位。这时，就可以采用该基本语句的执行次数来作为运行时间 $T(n)$建立的依据，即用其执行次数来对运行时间 $T(n)$进行度量。

【例 1-4】 求出一个整型数组中元素的最大值。

算法描述如下：

```
int arrayMax(int a[n])
{
  int max = a[0];
  for(int i = 1;i < n;i++)
    if (a[i]> max)
          max = a[i];
  return max;
}
```


1.3.4 渐近复杂性态—$T(n)$建立的依据示例

当算法的时间复杂性既依赖问题规模又依赖输入序列时，例如插入、排序、查找等算法，如果要合理全面地对这类算法的复杂性进行分析，则要从最好、最坏和平均情况三个方面进行讨论。

【例 1-5】 在一个整型数组中顺序查找与给定整数值 K 相等的元素(假设数组中当且仅当只有一个元素的值为 K)。

算法描述如下：

```
int find(int a[n],int K)
{
    for(int i = 0;i <n;i++)
        if(a[i] = = K)
            break;
    return i;
}
```

在该算法中，问题的规模由数组中的元素个数决定。显然，对算法的运行时间贡献最大的语句是if($a[i]==K$)，因此将该语句作为基本语句。但是该语句的执行次数不但依赖问题的规模，还依赖于输入数据的初始状态。

目录

习题

1. 写出Knuth的算法定义
2. 给出Knuth关于算法特性的描述
3. 给出一般伪代码描述原则
4. 给出欧几里得GCD算法的伪码描述
5. 给出线性搜索算法的伪码描述
6. 给出求数组中最大值算法的伪码描述
7. 给出渐近上界大O的定义及极限判断准则
8. 给出渐近下界大Ω的定义及极限判断准则
9. 给出渐近精确界大Θ的定义及极限判断准则
10. 按复杂度由低到高排列下列复杂度类
    - $O(\log n), O(n!), O(n^n), O(n^2), O(2^n), O(1), O(n\log n), O(\log n), O(n^3), O(n)$