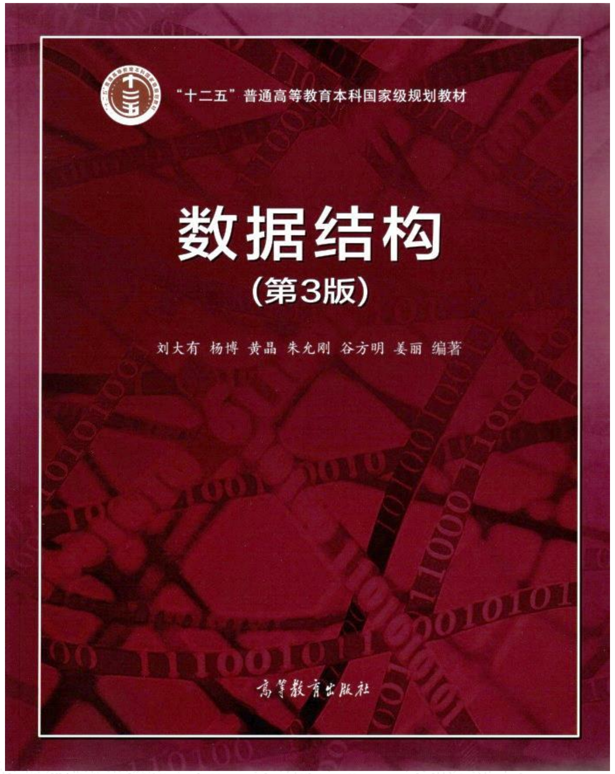
课堂实录

算法概述



- 算法基础和ADL回顾
- 时间复杂度及渐近表示法
- 典型例题
- 均摊时间复杂度

数据之法
结构之美
算法之道

Last updated on 2023.8

zhuyungang@jlu.edu.cn

代码是什么？

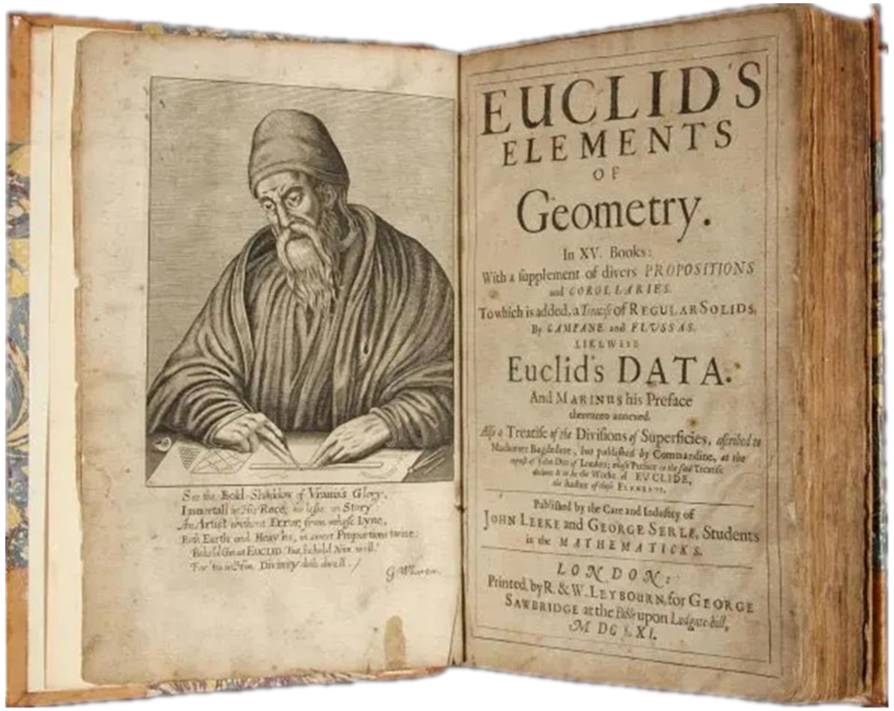
慕课自学内容

- 算法及其特性
- 算法的描述（ADL）
- 算法的评价准则

世界上第一个算法

辗转相除法（亦称欧几里得算法）

- 计算两个整数的最大公约数
- 公元前3世纪由欧几里得（Euclid）提出

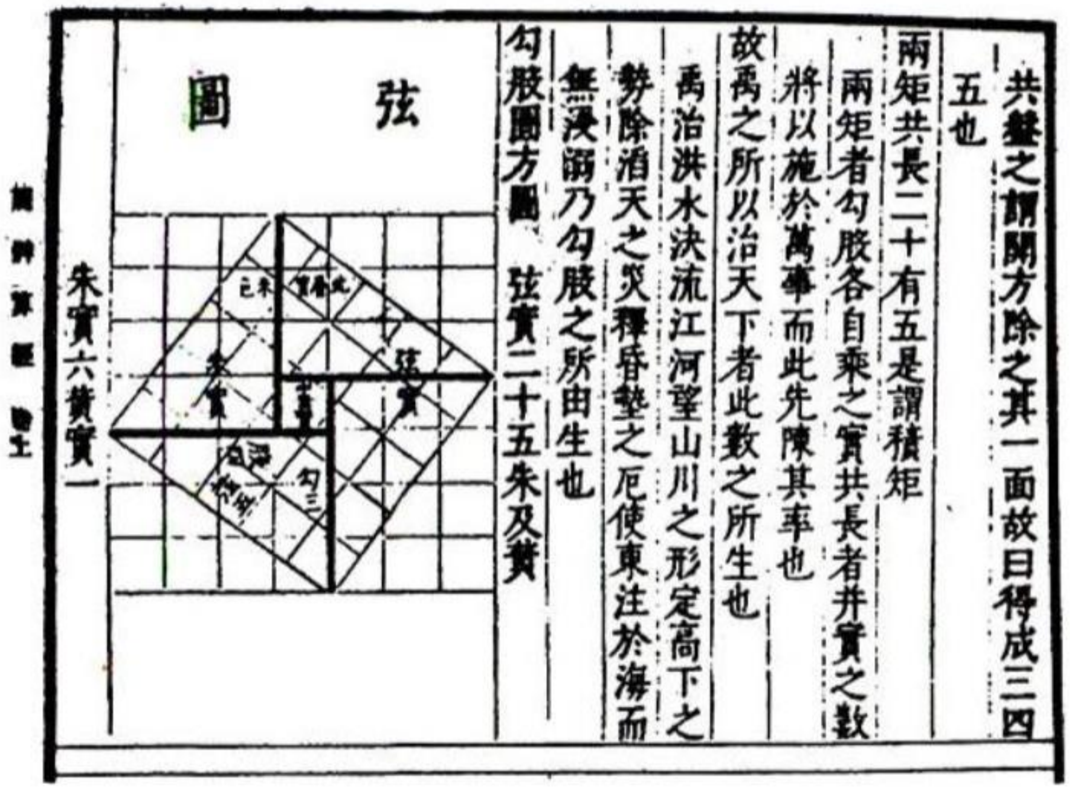


*Algorithm*一词的由来

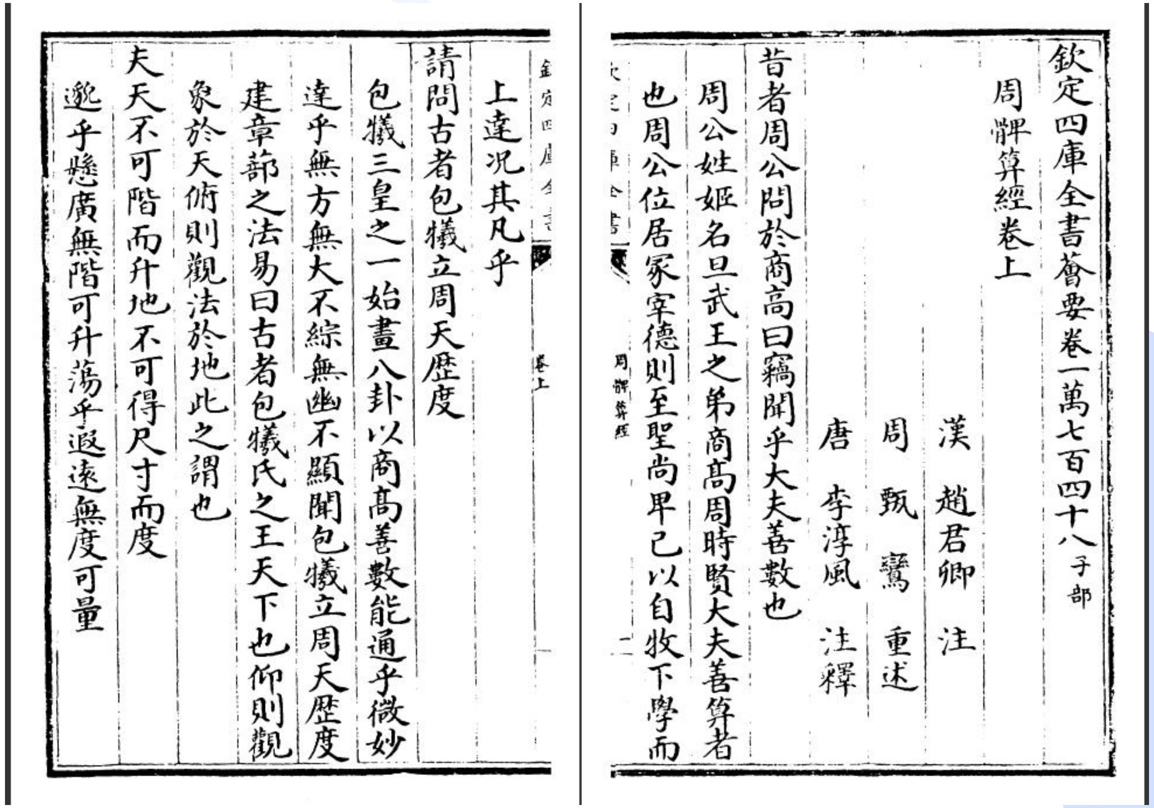
- 公元9世纪，著名波斯数学家阿尔·花拉子米写成《印度数字算术》一书。
- 随后该书被译成拉丁语《*Algoritmi de Numero Indorum*》传入欧洲，对阿拉伯数字在西方世界的传播起到至关重要的作用。*Algoritmi* 即为花拉子米的拉丁文译名。
- *Algorithm*一词由阿尔·花拉子米的拉丁文译名演变而来。

中文“算法”一词的由来

- ➢公元前1世纪《周髀算经》。



共蓋之謂開方除之其一面故曰得成三四
五也
兩矩共長二十有五是謂積矩
兩矩者勾股各自乘之實共長者并實之數
將以施於萬事而此先陳其率也
故禹之所以治天下者此數之所生也
禹治洪水決流江河望山川之形定高下之
勢除滔天之災釋昏墊之厄使東注於海而
無浸溺乃勾股之所由生也
勾股圓方圖 弦實二十五朱及黃



欽定四庫全書薈要卷一萬七百四十八 子部
周髀算經卷上
漢 趙君卿 注
周 甄鸞 重述
唐 李淳風 注釋
昔者周公問於商高曰竊聞乎大夫善數也
周公姓姬名旦武王之弟商高周時賢大夫善筭者
也周公位居冢宰德則至聖尚卑己以自牧下學而
上達況其凡乎
請問古者包犧立周天歷度
包犧三皇之一始畫八卦以商高善數能通乎微妙
達乎無方無大不綜無幽不顯聞包犧立周天歷度
建章蔀之法易曰古者包犧氏之王天下也仰則觀
象於天俯則觀法於地此之謂也
夫天不可階而升地不可得尺寸而度
邈乎懸廣無階可升蕩乎遐遠無度可量

算法（Algorithm）

在计算机科学中，算法泛指解决问题时所采用的方法或步骤。具体的，算法由有限条指令构成，规定了解决特定问题的一系列操作。

算法的特性

（1）有限性：算法必须能在执行有限个步骤之后终止；

（2）确定性：每条指令有明确的含义，无二义性，相同的输入得到相同结果；

（3）输　入：具有0个或多个由外界提供的量；

（4）输　出：产生1个或多个结果；

（5）可行性：算法中描述的操作都是可以通过已经实现的基本运算执行有限次来完成的，原则上人们用纸和笔都可在有穷时间内完成它们。

算法 VS 程序

- 程序是算法用某种程序设计语言的具体实现
- 程序可以不满足算法的性质（1）有限性。
- while(1) {

  }

算法的评价准则

- 正确性
- 时间复杂性
- 空间复杂性
- 可读性
- 健壮性

1. 正确性

对于一切合法的输入数据，该算法经过有限时间的执行都能产生正确（或者说满足规格说明要求）的结果。

2. 可读性

可读性好的算法使得证明或测试其正确性比较容易，同时便于阅读、实现、调试和维护。添加注释有利于提高程序的可读性。

3. 健壮性（鲁棒性-Robust）

尽可能充分地应对、处理各种极端输入。

4. 时间复杂性

如何度量算法的时间效率：实际运行时间？

- 算法的实际执行时间依赖于机器。同一个算法在不同机器上的执行时间不一定相同。
- 算法的实际执行时间还依赖于编写算法的程序设计语言及实现细节，一个用 C 语言编写的程序比 Java、Python 快，printf()和 scanf() **VS** cout和cin

4. 时间复杂性

- 度量算法时间效率的标准：

（1）能表示算法所采用的方法的时间效率；

（2）与算法描述语言及设计风格无关；

（3）与算法的许多细节无关；

（4）足够精确和具有一般性。

- 基本运算（关键运算）

算法中起主要作用且费时最多的操作。

- 时间复杂性（度）

一个算法的时间复杂性（度）是指该算法的基本运算次数。

5. 空间复杂性

- 算法在运行过程中所占用的存储空间的大小被定义为算法的**空间复杂性**。

算法描述语言

- 教材上主要采用伪代码（ADL语言）描述算法。
- ADL语言不要求一定会写，但至少应能看懂，平时作业和各类考试，可以使用C/C++、JAVA、Python、ADL等任意语言，只要能让人看懂就行。

C/C++:

```
if (i= =5)
{   i=i+1;
    j=j+1;
}
else
{   i=i-1;
    j=j-1;
}
```

ADL:

```
IF i = 5 THEN
(  i ← i+1.
   j ← j+1.
)
ELSE
(  i ← i-1.
   j ← j-1.
)
```

C/C++:

```
switch (i)
{ case 1: j=j+1; break;
  case 2: j=j+2; break;
  case 3: j=j+3; break;
}
```

ADL:

```
CASE DO
(  i=1:  j ← j+1.
   i=2:  j ← j+2.
   i=3:  j ← j+3.
)
```

C/C++:

```
while (i < 5)
{         i=i+1;
          j=j+1;
}
```

ADL:

```
WHILE i < 5 DO
(  i ← i+1.
   j ← j+1.
)
```

C/C++:

```
for (i=1;i <= 5; i++)
{         i=i+1;
          j=j+1;
}
```

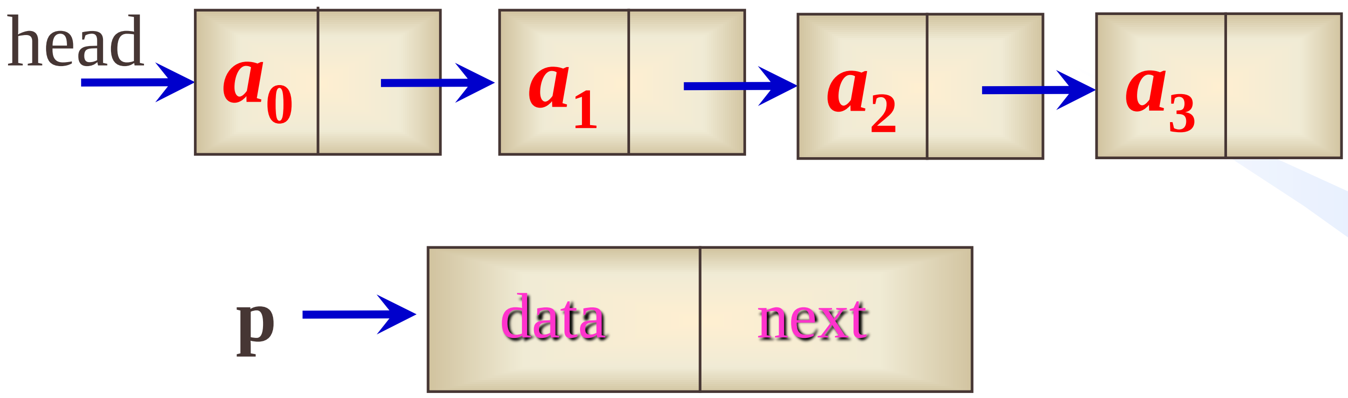
ADL:

```
FOR i = 1 TO 5 STEP 1 DO
(  i ← i+1.
   j ← j+1.
)
```

ADL:

```
FOR ∀ i ∈ U DO
    i ← i+1.
```

- 每个算法都用“▌”表示算法书写完毕
- 需要注释的语句由“//”或者“/*”引导
- 输入或输出语句使用 **READ(x)** 和 **PRINT(x)**



C/C++:

p->data
p->next

ADL:

data(p)
next(p)

例 A 是一个包含n个不同元素的实数数组(下标从1开始），求 A 的最大和最小元素。

算法SelectMaxMin(A , n . max , min)

```
max ← min ← A[1].
FOR i = 2 TO n DO     //求最大和最小元素
(   IF A[i] > max  THEN  max ← A[i].
    IF A[i] < min  THEN  min ← A[i].
) ▌
```

ADL 的特点

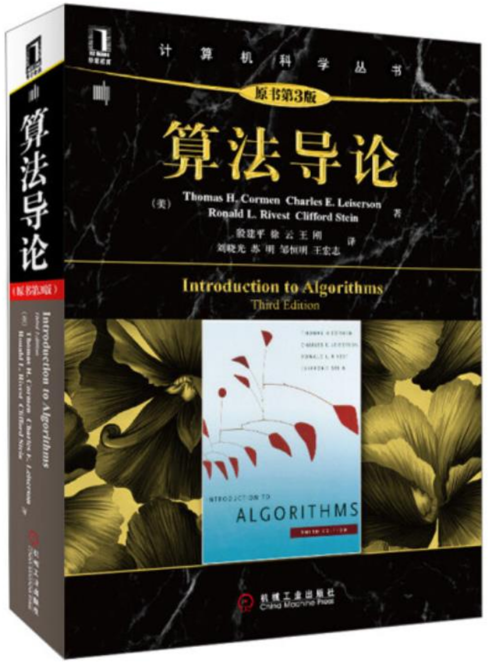
- 书写简便
  - 不必考虑过多程序设计语言的细节
  - 例: 交换 a 和 b 的值
- 易于理解
  - 便于不同编程习惯的人交流
  - 不必考虑与算法无关的内容(运行环境、编译状态等)
- 要点：不拘小节，只要把算法描述清楚了就行

ADL	C++

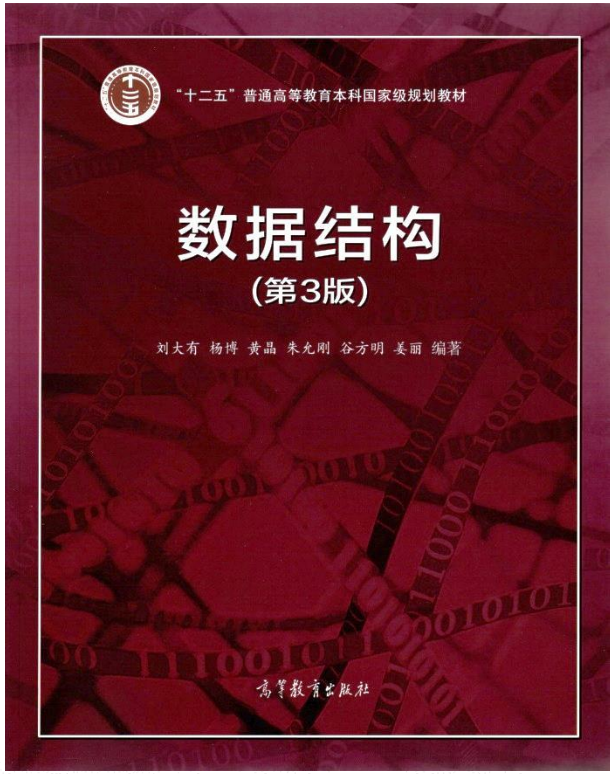
a ↔ b.	int temp;
	temp = a;
	a = b;
	b = temp;

伪代码的意义

- 数据结构与算法具有通用性，独立于具体程序设计语言。
- 科研论文往往使用伪代码描述算法。
- 两本编程圣经TAOCP和CLRS均使用伪代码描述算法。
- 教育部101计划《数据结构》教材，也将使用伪代码。



课堂实录



算法概述

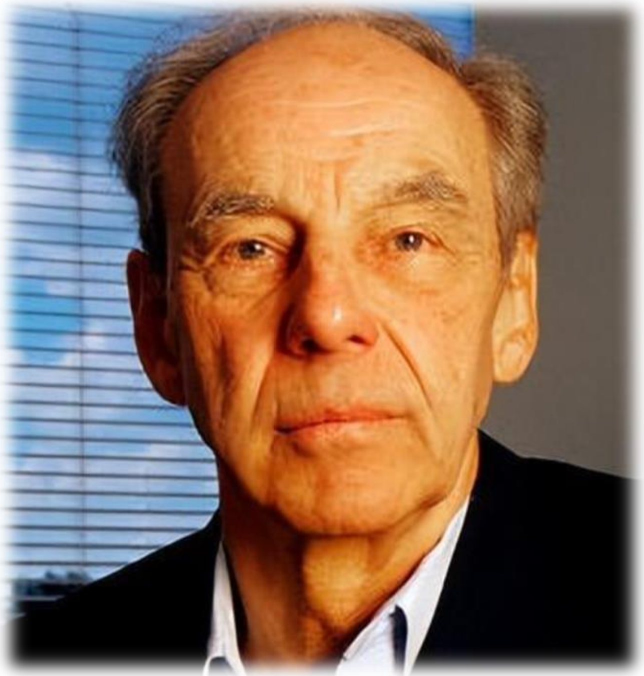
- 算法基础和ADL回顾
- 时间复杂性及渐近表示法
- 典型例题
- 均摊时间复杂度

数据之法
结构之美
算法之道

Last updated on 2023.8

zhuyungang@jlu.edu.cn

算法复杂性



Juris Hartmanis
图灵奖获得者
康奈尔大学教授
美国工程院院士

Richard Stearns
图灵奖获得者
纽约州立大学教授

Juris Hartmanis, Richard Stearns. On the Computational Complexity of Algorithms. Transactions of the American Mathematical Society, 177 (1965), 285-306.

关键运算与时间复杂性

- 关键运算（基本运算、关键操作、基本操作）

  算法中起主要作用且费时最多的操作。

  例：矩阵运算中的“*”，“+”，

  排序算法中的“比较”

  数组插入/删除操作的“赋值/移动”

- 时间复杂性（度）

  算法中基本运算的次数

  往往是问题规模的函数

例 **FOR** i=1 **TO** n **DO**

$x \leftarrow x*y.$

$T(n) = n$

例 **FOR** i=1 **TO** n **DO**

FOR j=1 **TO** n **DO**

$x \leftarrow x*y.$

$T(n) = n^2$

算法SelectMaxMin(A , n . max , min)

```
max ← min ← A[1].
FOR i = 2 TO n DO      //求最大和最小元素
(    IF A[i] > max  THEN  max ← A[i].
     IF A[i] < min   THEN  min ← A[i].
) ▌
```

$$T(n) = 2(n-1)$$

例 数组R由n个元素组成（下标从1开始），给定一个数K，试确定K是否是R的元素。

算法Find(R, n, K)

  $i \leftarrow 1$.

  WHILE $i \le n$ **DO**

  (**IF** $R[i]=K$ **THEN** **RETURN** i.

   $i \leftarrow i+1$.

).

  RETURN -1. ▌

若返回值∈[1, n]，则 K 在 R 中
若返回-1，则K 不是 R 的元素。

基本运算：R 中元素与 K 的比较

- K等于R中第1个元素，基本运算次数为1
- K等于R中第n个元素或K不在R中，基本运算次数n

同一个算法的不同输入可能导致不同的基本运算次数

算法的平均、最好和最坏时间复杂性

定义 设一个领域问题的输入的规模为n，
D_n是该领域问题的所有输入的集合，对任一输入$I \in D_n$，
$P(I)$是I出现（发生）的概率，
$T(I)$是算法在输入I下所执行的基本运算次数。

一个算法的不同输入可能产生不同的运算次数

该算法的平均时间复杂性定义为：

$$E(n)=\sum_{I \in D_n}\{P(I)\times T(I)\}$$

另一种理解：按照假定的概率分布，对算法各种输入情况下的基本运算次数做加权求和。权值：每种输入发生的概率

该算法的最好时间复杂性为：$\min\{T(I)\}$
该算法的最坏时间复杂性为：$\max\{T(I)\}$

例 数组R由n个元素组成（下标从1开始），给定一个数K，试确定K是否是R的元素。

算法Find(R, n, K)

$i \leftarrow 1$.

WHILE $i \le n$ **DO**

(**IF** $R[i]=K$ **THEN** **RETURN** i.

$i \leftarrow i+1$.

).

RETURN -1. ▌

基本运算：R 中元素与 K 的比较

- 最好时间复杂度：1【最少比较次数】
- 最坏时间复杂度：n【最大比较次数】

假定：设 K 是R中元素的概率为 $q\ (0 \le q \le 1)$，
K不在R中的概率$1\text{-}q$

R[1]	R[2]	R[3]	R[4]	R[5]	R[6]	R[7]	R[8]
5	20	12	7	30	40	25	16

$\underbrace{\qquad\qquad\qquad\qquad}_{q}$

假定 K 等于 R 的每个元素的概率相同

$$\begin{cases} K = \text{某个}R[\,i\,]\text{的概率 } q/n & 1 \le i \le n \\ K \text{ 不等于}R\text{中元素的概率} 1\text{-}\, q & \end{cases}$$

有多少种可能的输入?

输入种类	输入内容	输入发生概率	基本运算次数
第1种	K 等于 R 中第1个元素 $R[1]$	q/n	1

算法Find的平均复杂性为

$$E(n) = \frac{q}{n} \times 1 +$$

有多少种可能的输入？

输入种类	输入内容	输入发生概率	基本运算次数
第1种	K 等于 R 中第1个元素 $R[1]$	q/n	1
第2种	K 等于 R 中第2个元素 $R[2]$	q/n	2

算法Find的平均复杂性为

$$\mathrm{E}(n) = \frac{q}{n} \times 1 + \frac{q}{n} \times 2 +$$

有多少种可能的输入？

输入种类	输入内容	输入发生概率	基本运算次数
第1种	K 等于 R 中第1个元素 $R[1]$	q/n	1
第2种	K 等于 R 中第2个元素 $R[2]$	q/n	2
	……		

算法Find的平均复杂性为

$$E(n) = \frac{q}{n} \times 1 + \frac{q}{n} \times 2 + \frac{q}{n} \times 3 + \cdots$$

有多少种可能的输入？

输入种类	输入内容	输入发生概率	基本运算次数
第1种	K 等于 R 中第1个元素 $R[1]$	q/n	1
第2种	K 等于 R 中第2个元素 $R[2]$	q/n	2
	……		
第n种	K 等于 R 中第n个元素 $R[n]$	q/n	n

算法Find的平均复杂性为

$$\mathrm{E}(n) = \frac{q}{n} \times 1 + \frac{q}{n} \times 2 + \frac{q}{n} \times 3 + \cdots + \frac{q}{n} \times n$$

有多少种可能的输入?

输入种类	输入内容	输入发生概率	基本运算次数
第1种	K 等于 R 中第1个元素 $R[1]$	q/n	1
第2种	K 等于 R 中第2个元素 $R[2]$	q/n	2
……			
第n种	K 等于 R 中第n个元素 $R[n]$	q/n	n
第n+1种	K不等于 R 中任何一个元素	$1-q$	n

算法Find的平均复杂性为

$$\mathrm{E}(n)=\frac{q}{n}\times 1+\frac{q}{n}\times 2+\frac{q}{n}\times 3+\cdots+\frac{q}{n}\times n+(1-q)\times n$$

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

算法Find的平均复杂性为

$$E(n) = q(n+1)/2+(1-q)n$$

如果已知 *K* 在 *R* 中，即*q*=1，则有

$$E(n) = (n+1)/2$$

例 A是一个含有n个不同元素的实数数组，求A之最大和最小元素的算法

算法SelectMaxMin(A , n . max , min)

```
max ← min ← A[1].
FOR i = 2 TO n DO      //求最大和最小元素
(    IF A[i] > max  THEN  max ← A[i].
     IF A[i] < min   THEN  min ← A[i].
) ▌
```

$$T(n) = 2(n-1)$$

例 算法SelectMaxMin的改进算法BinarySelect

```
算法BinarySelect (A, i, j. max, min) // i和j为数组A的起止元素下标，i≤j
  IF i = j THEN ( max←min←A[i].  RETURN. ) //递归出口，有1个元素
  IF i = j - 1 THEN          //递归出口，有2个元素
  (    IF A[i] < A[j]  THEN  ( max←A[j].   min←A[i]. )
       ELSE ( max←A[i].  min←A[j]. )
       RETURN.
  )
  mid ← ⌊(i+j)/2⌋.   //取中值
  BinarySelect ( A, i, mid. Lmax, Lmin ) .    //在左侧子数组递归查找
  BinarySelect ( A, mid+1, j. Rmax, Rmin ) . //在右侧子数组递归查找
  IF  Lmax > Rmax  THEN max ← Lmax.   //合并
  ELSE max ← Rmax.
  IF  Lmin < Rmin  THEN min ← Lmin.
  ELSE min ← Rmin. ▌
```

i			*mid*				*j*

有无问题？

分治法

- 算法**BinarySelect**的基本运算为元素的比较
- 算法**对不同的输入**A[i]...A[j]，**都有相同的基本运算次数**。
- 设T(n)表示其基本运算次数，则根据算法的递归过程，有：

$$T(n)=\begin{cases}0 & n=1\\ 1 & n=2\\ T(\lfloor n/2\rfloor)+T(\lceil n/2\rceil)+2 & n>2\end{cases}$$

当 n 是 2 的幂时（即存在正整数 k，使得 $n=2^k$）有

$$T(n)=\begin{cases}0 & n=1\\ 1 & n=2\\ T(\lfloor n/2\rfloor)+T(\lceil n/2\rceil)+2 & n>2\end{cases}$$

$$\begin{aligned}T(n) &= 2T(n/2)+2\\ &= 2(2T(n/4)+2)+2\\ &= 4T(n/4)+4+2\\ &= 4(2T(n/8)+2)+4+2\\ &= 8T(n/8)+8+4+2=\cdots\\ &= 2^{k-1}T(n/2^{k-1})+2^{k-1}+2^{k-2}+\cdots+2\\ &= 2^{k-1}T(2)+\sum_{i=1}^{k-1}2^i \; = 2^{k-1}+2^k-2=\frac{3}{2}n-2\end{aligned}$$

例 A是一个含有n个不同元素的实数数组，求A之最大和最小元素的算法

```
算法SelectMaxMin(A , n .  max , min)
   max ← min ← A[1].
   FOR i = 2 TO n DO      //求最大和最小元素
   (    IF A[i] > max  THEN  max ← A[i].
        IF A[i] < min  THEN  min ← A[i].
   ) ▌
```

$$T(n) = 2(n-1)=2n-2$$

BinarySelect与SelectMaxMin的比较

SelectMaxMin：$2n$-2

BinarySelect：$1.5n$-2

就计算时间而言，算法BinarySelect优于算法SelectMaxMin。

然而算法BinarySelect是递归算法，因此它的实现需要额外的辅助空间栈。

时间复杂性的渐近表示方法

- 在很多情况下，特别当输入规模 n 较大时，确定一个算法的基本运算次数T，得到 T 和 n 之间的精确的函数关系比较困难。
- 很多时候，我们并不需要知道$T(n)$的精确表达式，只需知道$T(n)$数量级，即：随输入规模的增长，算法执行时间$T(n)$的变化趋势。

能不能用一个简单的函数表示$T(n)$？

时间复杂性的渐近表示方法

$$T(n)=n^2+100n+\log_{10}n+1000$$

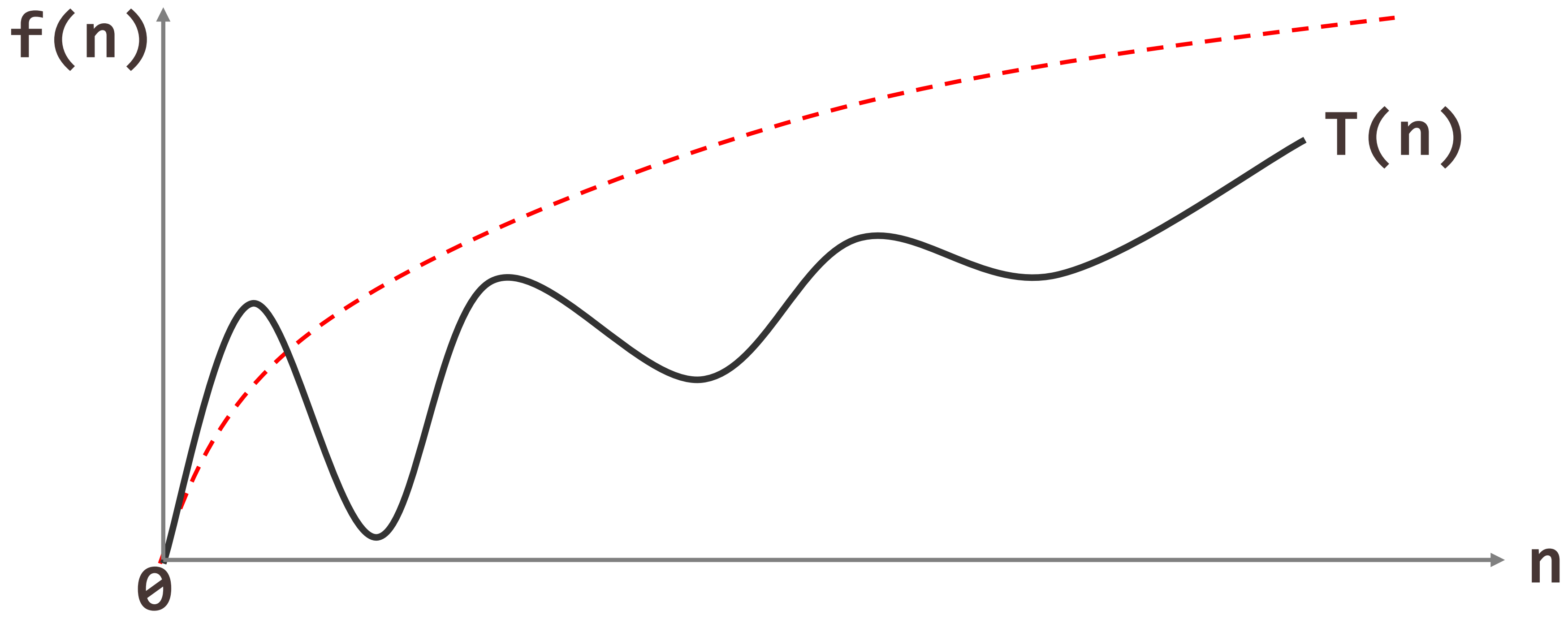
*n*取很小值时，例如为1时，最后的常数项对函数值贡献最大，随着*n*的增大后3项对函数值的贡献越来越小，当*n*很大时，函数值主要依赖于第一项。

随着*n*增大，第一项决定了*T*(*n*)的变化趋势

$$g(n)=n^2$$

时间复杂性的渐近表示方法

算法分析中通常采用大 *O*、大 *Ω*、大*Θ* 渐近表示算法的基本运算次数，从形式上化简*T*(*n*)的表示。



O 表示法

Ω 表示法

Θ表示法

(1) O表示法

设T(n)和g(n)是正整数集到正实数集上的函数。

称T(n) = O(g(n))，当且仅当存在一个正常数C 和 n_0，使得对任意的 $n \geq n_0$，有T(n) ≤ C g(n)。

在某一条件下，在大O的意义下，将T(n)表示为一个更简单的函数g(n)。

充要条件：存在C，当n足够大，T(n)不会超过g(n)的C倍。

好读书，不求甚解

———陶渊明

(1) O表示法

设T(n)和g(n)是正整数集到正实数集上的函数。

称$T(n) = O(g(n))$，当且仅当存在一个正常数C 和 n_0，使得对任意的 $n \geq n_0$，有$T(n) \leq C\, g(n)$。

n 是算法输入的规模，如数组的长度，图的顶点数等；

一个算法时间复杂性是$O(g(n))$，称其**时间复杂性的阶**为$g(n)$；

$$T(n) = O(g(n)) \Leftrightarrow \lim_{n\to\infty} \frac{T(n)}{g(n)} \leq C \quad (C \geq 0)$$

例 1 $T(n)=3n-2$， $g(n)=n$

存在 $C = 3, n_0 = 1$, 当 $n \geq n_0$ 时 有

$$3n-2 \leq 3n$$

于是有：$T(n)=O(n)$

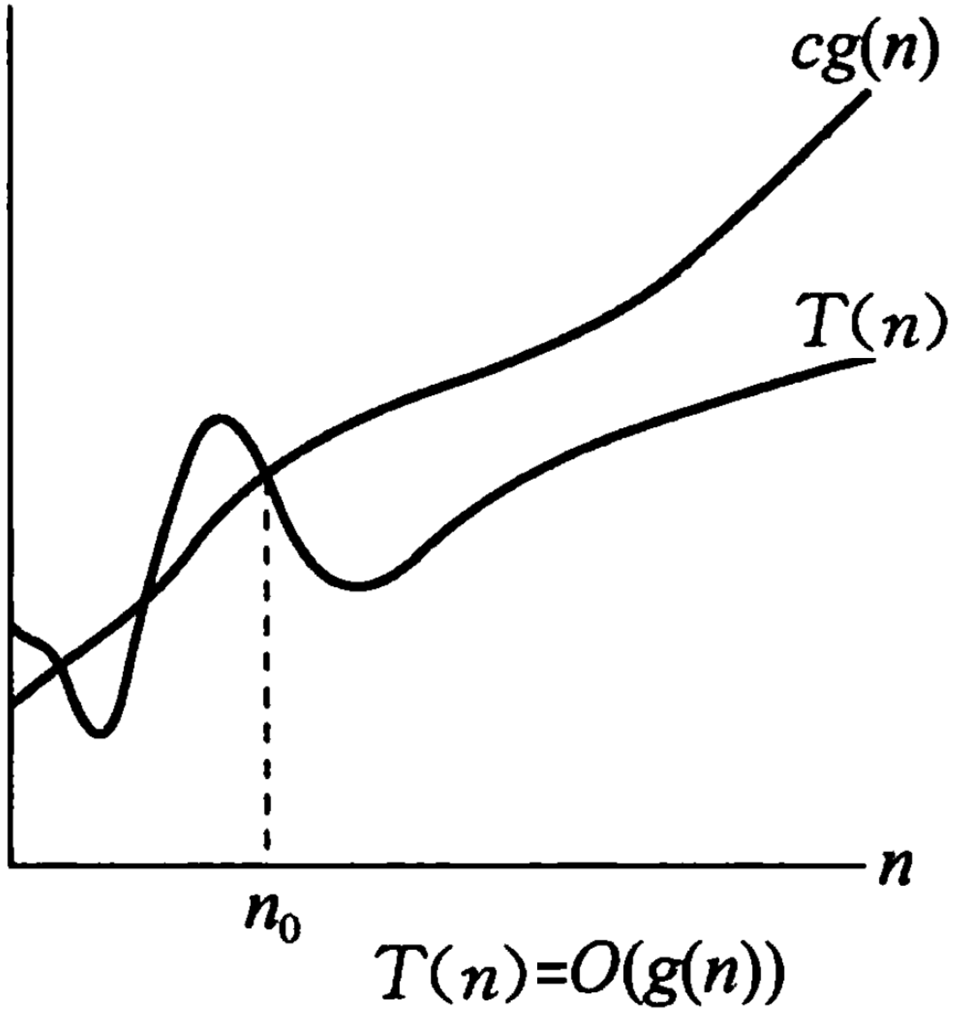
例2 $T(n)=(n+1)^2$ ， $g(n) = n^2$

存在 $n_0=1, C=4$， 当 $n \geq n_0$ 时有：

$$(n+1)^2 \leq (2n)^2=4n^2,$$

于是有：$T(n)=O(n^2)$

与T(*n*)相比，g(*n*)更为简洁，但依然反应前者的增长趋势

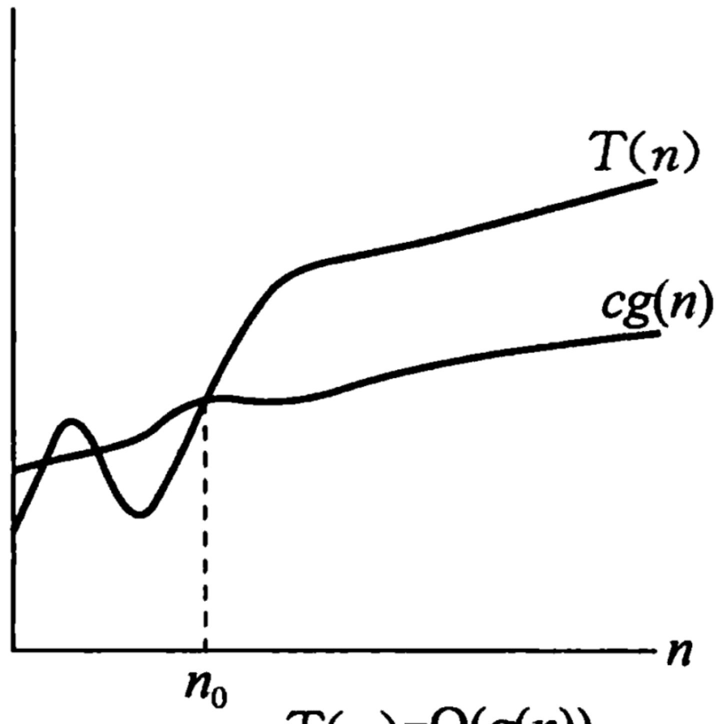


大 O 符号定义了函数 T(*n*) 的一个上限，算法的运行 时间（基本运算次数）至多是g(*n*)的一个常数倍，即：

T(*n*)的增长速度至多与g(*n*)的增长速度一样快。

(2) Ω表示法

称$T(n) = \Omega(g(n))$，当且仅当存在一个正的常数C和n_0，使得对任意的$n \geq n_0$，有$T(n) \geq C\, g(n)$。



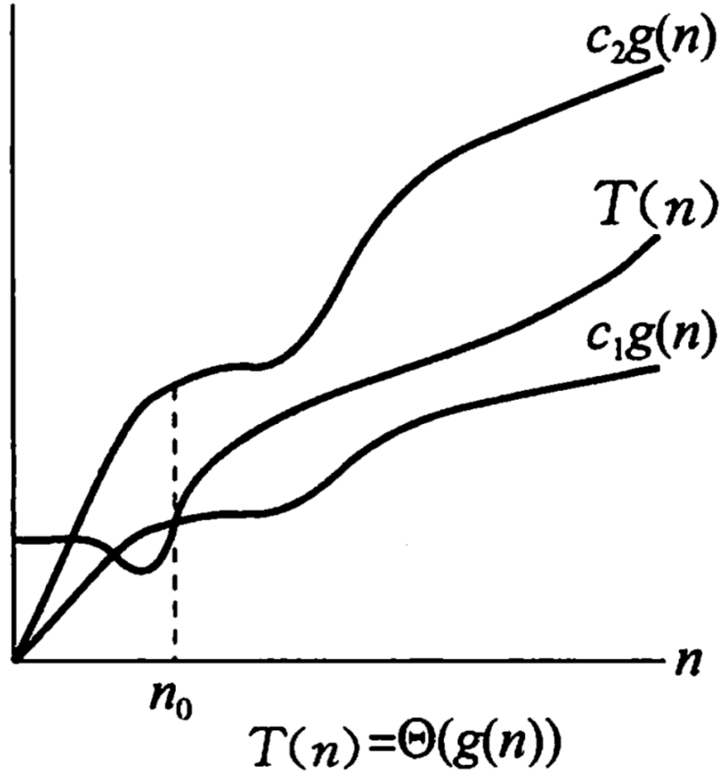
$T(n)=\Omega(g(n))$

大 Ω 符号定义了函数 T(n)的一个下限，算法的运行时间至少是g(n)的一个常数倍。即：

T(n)的增长速度不低于g(n)的增长速度。

(3) 大Θ表示法

称T(n) = Θ (g(n))， 当且仅当存在正常数C_1、C_2和n_0，使得对任意的$n \geq n_0$，有 $C_1 g(n) \leq T(n) \leq C_2 g(n)$，



大Θ符号定义了函数T(n)的上限和下限。

T(n)的增长速度与g(n)的增长速度相同。

例 3 $T(n)=3\log n+\log\log n$，$g(n)=\log n$

存在 $C = 4, n_0 = 2$ 当 $n \geq n_0$ 时 有

$$3\log n+\log\log n \leq 4\log n$$

于是有：$T(n)=O(\log n)$

存在 $C = 3, n_0 = 2$ 当 $n \geq n_0$ 时 有

$$3\log n+\log\log n \geq 3\log n$$

于是有：$T(n)= \Omega(\log n)$

进而存在 $C_1 = 3$，$C_2 = 4$，$n_0 = 2$，当 $n \geq n_0$ 时 有

$$3\log n \leq 3\log n+\log\log n \leq 4\log n$$

于是有：$T(n)= \Theta(\log n)$

回顾：大O 表示法

定义： 称$\mathrm{T}(n)$是$O(g(n))$，当且仅当存在正常数C和n_0，使得对任意的$n \geq n_0$，有$T(n) \leq Cg(n)$.

理解：

在某一条件下，在大O的意义下，将$\mathrm{T}(n)$表示为一个更简单的函数$\mathrm{g}(n)$。

充要条件：存在C，当n足够大，$\mathrm{T}(n)$不会超过$\mathrm{g}(n)$的C倍。

$$\mathrm{T}(n) = O(g(n))$$

例：$n^2+100n+\log_{10}n+1000 = \mathrm{O}(n^2)$

常见的时间复杂度

$O(1)$ 表示算法的时间复杂性为一常数. $O(\log n)$、$O(n)$、$O(n^2)$、$O(n^3)$、$O(n^m)$ 和 $O(2^n)$ 分别表示算法时间复杂性为对数、线性、平方、立方、多项式和指数阶（也称为级）的，其中常数 $m \geq 1$.

$O(1)$,	$O(\log n)$,	$O(n)$,	$O(n^2)$,	$O(n^3)$,	$O(n^m)$,	$O(2^n)$
常数级	对数级	线性级	平方级	立方级	多项式级	指数级

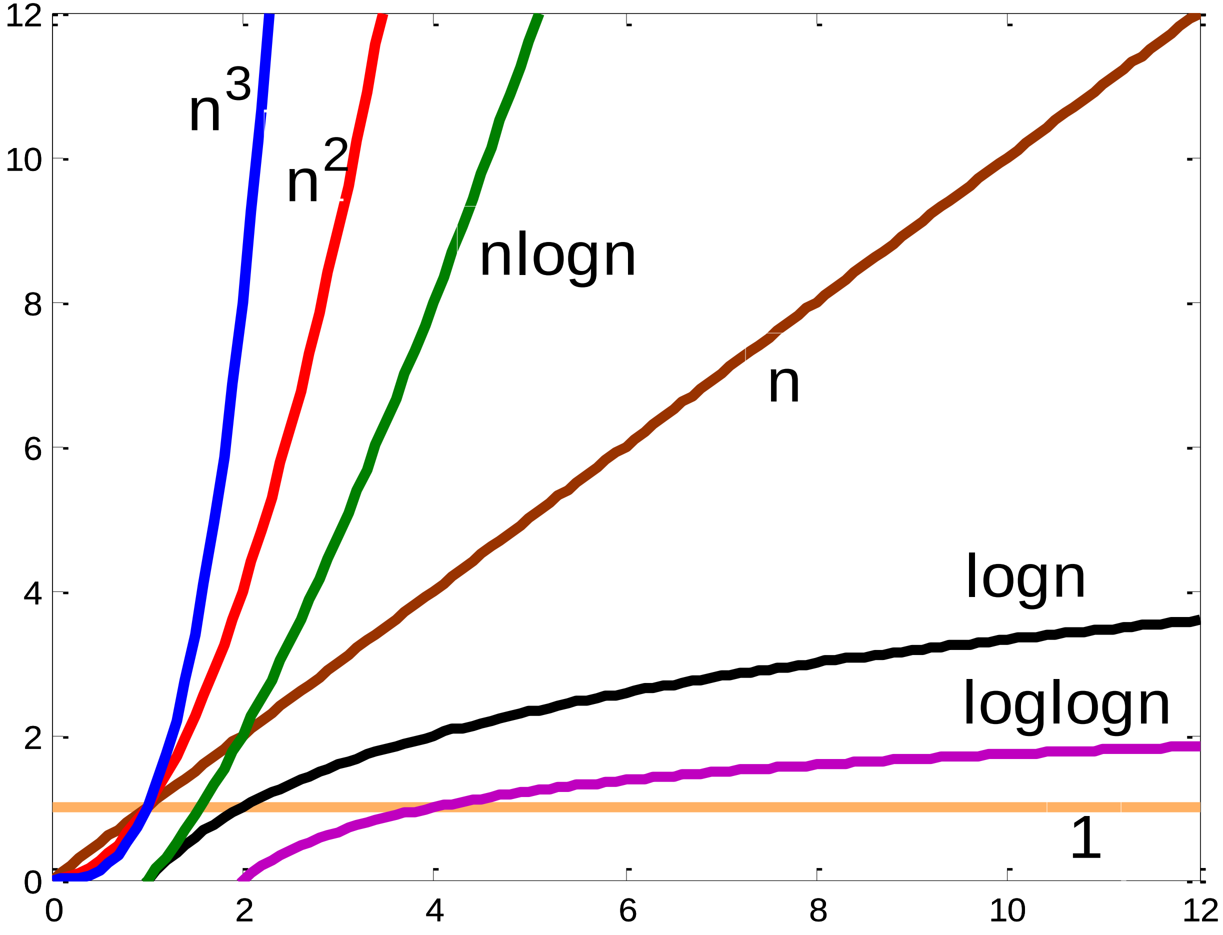
当n很大时，有如下关系成立

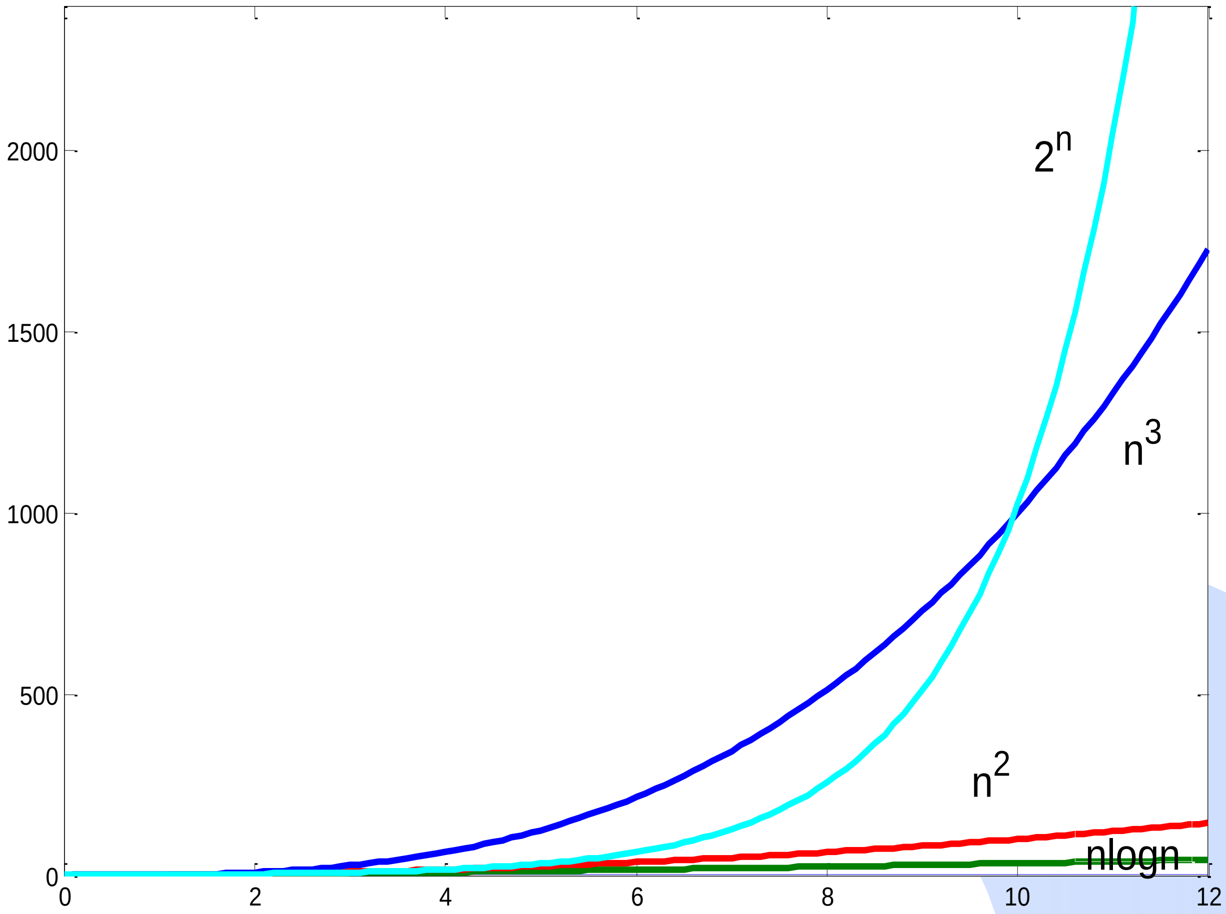
$$1 < \log\log n < \log n < n < n\log n < n^2 < n^3 < 2^n$$

因此，有

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

常见算法时间复杂性函数的序关系





常见算法的时间复杂度与问题规模的关系

问题规模 / 时间复杂度	$n = 10$	$n = 30$	$n = 60$	$n = 100$
$O(n)$	0.01ms	0.03ms	0.06ms	0.1ms
$O(n^2)$	0.1ms	0.9ms	3.6ms	10ms
$O(n^5)$	0.1s	24.3s	13min	167min
$O(2^n)$	1.0ms	17.9min	366世纪	4×10^{16}年

在每秒处理100万次基本操作的计算机上的执行时间

神威·太湖之光超级计算机

- 超级计算机，被称为“国之重器”，是世界各国竞相角逐的科技制高点，也是一个国家科技实力的重要标志。
- 美国2015年宣布对中国禁售高性能处理器。
- 神威·太湖之光超级计算机是由我国自主研制，2016年至今位列全球超级计算机Top5，连续两年位居世界第1。

使用“神威·太湖之光”超级计算机（浮点运算速度9.3亿亿次/秒）

问题规模 / 时间复杂度	$n = 10$	$n = 30$	$n = 60$	$n = 100$
$O(n)$				
$O(n^2)$				
$O(n^5)$				
$O(2^n)$				43万年

仅提高硬件计算速度是不够的，关键是从理论上降低时间复杂性

大O标记的常用性质与结果

- **常系数可忽略：**

$$O(d\cdot f(n)) = O(f(n))$$

- **低次项可忽略：**

$$O(n^a+n^b) = O(n^a)\ ,\quad a > b > 0$$

$$\begin{aligned} & n^a+n^b \\ \le\ & n^a+n^a \\ \le\ & 2n^a \\ =\ & O(n^a) \end{aligned}$$

大O标记的常用性质与结果

- **对数底可忽略**

$$\log_a n = \log_a b \times \log_b n \qquad a, b>0$$

$$O(\log_a n) = O(\log_b n)$$

$$O(\log n)$$

大O标记的常用性质与结果

➢ **对数常数次幂可忽略**

$$\log n^c = c \times \log n = O(\log n)$$

大O标记的常用性质与结果

函数的阶取决于阶最高的项，与其系数和其它较低阶项无关。

例：

$$n+\log n = O(n)$$

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

$$a_m n^m+\ldots+a_1 n+a = O(n^m)$$

$$(n+1)(3n^2+2)= O(\quad\quad)$$

例

基本运算次数	渐近时间复杂度
$\frac{1}{10} \cdot n^2 + 100$	$O(n^2)$
$0.063 \cdot n^2 - 5n + 12.7$	$O(n^2)$
$100 \cdot n^{1.5} - 10^{10000}\sqrt{n}$	$O(n^{1.5})$
$11 \cdot n\log n + 1$	$O(n\log n)$

大O标记的常用性质与结果

级数：

$$1+2+\ldots+n = n(n+1)/2 = O(n^2)$$

$$1^2+2^2+\ldots+n^2 = n(n+1)(2n+1)/6 = O(n^3)$$

$$1^3+2^3+\ldots+n^3 = O(\quad)$$

$$1^4+2^4+\ldots+n^4 = O(\quad)$$

$$\sum_{k=1}^{n} k^d \approx \int_0^n x^d dx = \frac{n^{d+1}}{d+1} = O(n^{d+1})$$

大O标记的常用性质与结果

调和级数： $1+\frac{1}{2}+\frac{1}{3}+\frac{1}{4}+\ldots+\frac{1}{n}$

$$=\sum_{k=1}^{n}\frac{1}{k} \approx \int_{1}^{n}\frac{1}{x}dx$$

$$= \ln n$$

$$= O(\log n)$$

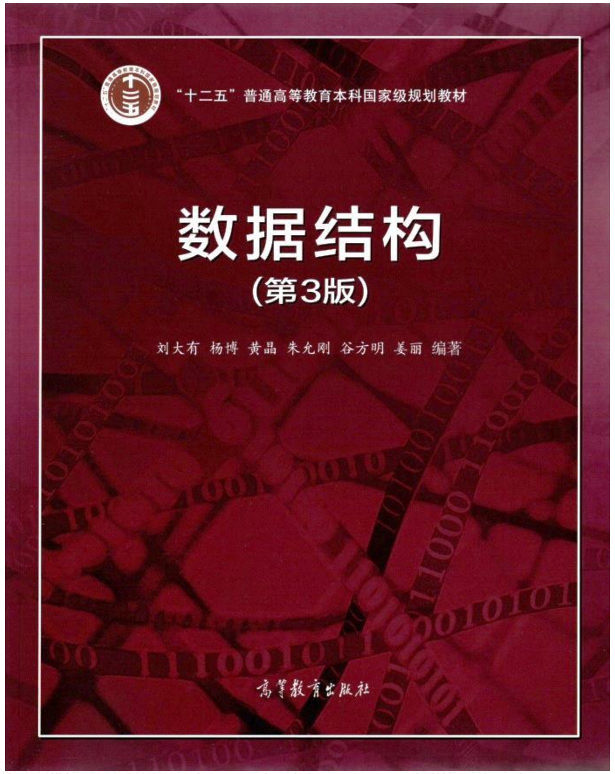
课下思考题

判断题：

（1）$200+\sin(n) = \Theta(1)$ 【剑桥大学练习题】

（2）$n^{\log\log\log n} = O(\lfloor \log n \rfloor !)$ 【清华大学考研题】

课堂实录



算法概述

- 算法基础和ADL回顾
- 时间复杂性及渐近表示法
- **典型例题**
- 均摊时间复杂度

数据之法
结构之美
算法之道

zhuyungang@jlu.edu.cn

典型例题

写出以下程序段的时间复杂性（用大O表示法）

```
count=0;
for(i=0;i<n;i++)
    count++;
```

O(*n*)

```
count=0;
for(i=0;i<n;i++)
    for(j=1;j<n;j++)
        count++;
```

$\mathbf{O}(\boldsymbol{n}^2)$

```
count=0;
for(i=0;i<n;i++)
    for(j=1;j<=n;j++)
        for(k=1;k<=n;k++)
            count++;
```

$O(n^3)$

```
count=0;
for(i=0;i<n;i++)
    for(j=0;j<9999;j++)
        count++;
```

O(*n*)

2019年考研题（全国卷）

```
x=0;
while(n>=(x+1)*(x+1))
    x=x+1;
```

O(　　)

```
count=0;
for(i=1; i<=n; i++)
    for(j=1; j<=i; j++)
        count++;
```

$$\sum_{i=1}^{n}\sum_{j=1}^{i}1$$

$$=\sum_{i=1}^{n}i$$

$$=\frac{n(n+1)}{2}=O(n^2)$$

```
count=0;
for(i=1; i<n; i*=2)
    count++;
```

O(log*n*)

i的值

第1次迭代后： 2^1

第2次迭代后： 2^2

……

第k次迭代后： 2^k

```
count=0;
for(i=n; i>=1; i/=2)
    count++;
```

O(log*n*)

2014年考研题（全国卷）

```
count=0;
for(k=1; k<=n; k*=2)
    for(j=1; j<=n; j++)
        count++;
```

O(*nlogn*)

考研题

```
void f(int n) {
    int i, j;
    for(i=0; i<n; i++) {
        j=1;
        while(j<n) j=j*2;
    }
}
```

O(*nlogn*)

吉大考研题

$n \geq 3$

```
for(i=0; i<n-1; i++)
    for(j=1; j<n; j=j+n/3)
        for(k=1; k<n; k=k*2)
            x=x*5;
```

O(*nlogn*)

几个稍复杂的例子

```
count=0;
for( i=1; i<=n; i++)
    for(j=0; j<n; j+=i)
        count++;
```

i=1时内循环执行n次

i=2时内循环执行$n/2$次

i=3内循环执行$n/3$次

$$T(n) = n + \frac{n}{2} + \frac{n}{3} + \frac{n}{4} + \ldots + 1$$

$$= n(1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \ldots + \frac{1}{n})$$

$$= O(nlogn)$$

大O标记的常用性质与结果

调和级数： $1+\frac{1}{2}+\frac{1}{3}+\frac{1}{4}+\ldots+\frac{1}{n}$

$$=\sum_{k=1}^{n}\frac{1}{k} \approx \int_{1}^{n}\frac{1}{x}dx$$

$$= \ln n$$

$$= O(\log n)$$

2017年考研题（全国卷）

```
int func(int n){
    int i=0, sum=0;
    while(sum<n)
        sum += ++i;
    return i;
}
```

O(   )

	i	*sum*
循环体第1次执行后：	1	1
循环体第2次执行后：	2	1+2
循环体第3次执行后：	3	1+2+3
…		
循环体第*k*次执行后：	*k*	1+2+…+*k*

若规定sum++为基本运算，请使用大O表示法给出下面算法尽可能精确的时间复杂度上界。**【2022年考研题全国卷、吉林大学2020级期末考试题】**

```
sum=0;
for( i=1; i<n; i*=2 )
    for( j=0; j<i; j++ )
        sum++;
```

sum++执行次数:
$1+2^1+2^2+2^3+\ldots+2^{k-1}$
$= 2^k-1$
$= O(n)$

外循环第1次执行，i=1，sum++执行1次
外循环第2次执行，i=2，sum++执行2^1次
外循环第3次执行，i=2^2，sum++执行2^2次
…
外循环第k次执行，i= 2^{k-1}，sum++执行2^{k-1}次

空间复杂度的分析方法

```c
int f(int a[], int n){
    int i, sum = 0;
    for(i = 0; i < n; i++)
        sum += a[i];
    return sum;
}
```

存储空间不会随数据规模*n*的变化而变化，空间复杂度O(1)

空间复杂度的分析方法

```
void f (int a[], int n){
    int *temp = new int [n];
    for(int i = 0; i < n; i++)
        temp[i] = a[n-i-1];
    for(int i = 0; i < n; i++)
        a[i] = temp[i];
    delete[] temp;
}
```

空间复杂度O(n)

设计高效算法

例：给定n,设计算法求$1^2+2^2+3^2+\ldots+n^2$的值。

- 算法1

```
sum = 0;
for(int i=1; i<=n; i++)
    sum += i*i;
```

时间复杂度O(n)

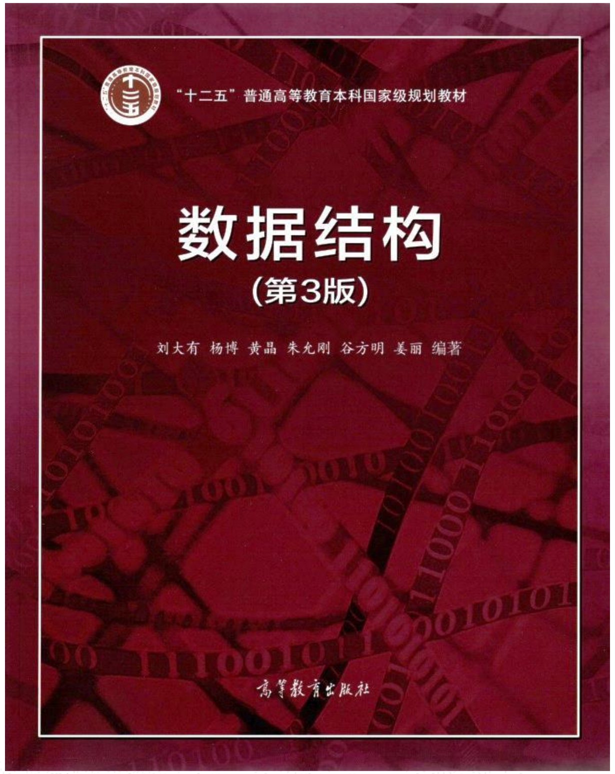
- 算法2

```
sum = n*(n+1)*(2*n+1)/6;
```

时间复杂度O(1)

很多问题设计低效的算法往往很容易，但设计出高效的算法却很不容易。

课堂实录



算法概述

- 算法基础和ADL回顾
- 时间复杂性及渐近表示法
- 典型例题
- **均摊时间复杂度**

数据之法
结构之美
算法之道

Last updated on 2023.9

zhuyungang@jlu.edu.cn

均摊时间复杂度
Amortized Complexity

- n位二进制加计数器

A							

```
void AddOne(int A[],int n){
   int i=0;
   while(i<n && A[i]==1){
     A[i]=0;
     i++;
   }
   if(i<n) A[i]=1;
}
```

- **最坏情况下，所有位都需要改变，最坏时间复杂度为O(n)**
- **算法连续执行n次：O(n^2)?**
- **计数器在不断加1的过程中，不可能遇到连续若干次最坏情况**
- **此时分析一个操作序列的总时间，而不是单个操作的时间，更可靠，这就称为均摊分析。**

均摊时间复杂度

Robert Tarjan

图灵奖获得者

普林斯顿大学教授

美国科学院院士

美国工程院院士

SIAM J. ALG. DISC. METH.
Vol. 6, No. 2, April 1985

016

AMORTIZED COMPUTATIONAL COMPLEXITY*

ROBERT ENDRE TARJAN†

Abstract. A powerful technique in the complexity analysis of data structures is *amortization*, or averaging over time. Amortized running time is a realistic but robust complexity measure for which we can obtain surprisingly tight upper and lower bounds on a variety of algorithms. By following the principle of designing algorithms whose amortized complexity is low, we obtain "self-adjusting" data structures that are simple, flexible and efficient. This paper surveys recent work by several researchers on amortized complexity.

均摊时间复杂度

Counter Value	$A[7]$	$A[6]$	$A[5]$	$A[4]$	$A[3]$	$A[2]$	$A[1]$	$A[0]$
0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	1
2	0	0	0	0	0	0	1	0
3	0	0	0	0	0	0	1	1
4	0	0	0	0	0	1	0	0
5	0	0	0	0	0	1	0	1
6	0	0	0	0	0	1	1	0
7	0	0	0	0	0	1	1	1
8	0	0	0	0	1	0	0	0
9	0	0	0	0	1	0	0	1
10	0	0	0	0	1	0	1	0
11	0	0	0	0	1	0	1	1
12	0	0	0	0	1	1	0	0

A[0]:每次调用算法，都会翻转，*n*次调用翻转*n*次

A[1]:每2次调用算法，翻转1次，*n*次调用翻转*n*/2次

A[2]:每4次调用算法，翻转1次，*n*次调用翻转*n*/4次

……

A[*i*]:每2^i次调用算法，翻转1次，*n*次调用翻转*n*/2^i次

均摊时间复杂度

- 初值为0的计数器，连续执行n次加1操作，总代价（翻转次数，基本运算次数）

$$\sum_{i=0}^{n-1}\left\lfloor \frac{n}{2^i} \right\rfloor < \sum_{i=0}^{n-1}\frac{n}{2^i} = n\sum_{i=0}^{n-1}\frac{1}{2^i} = n\frac{1-\left(\frac{1}{2}\right)^n}{1-\frac{1}{2}} = 2n\left(1-\frac{1}{2^n}\right) = O(n)$$

- 每个操作的平均代价，即AddOne算法的均摊时间复杂度 $O(n)/n=O(1)$

均摊时间复杂度

- 很多情况下，数据结构涉及的是操作序列而不是单个操作，每个操作的执行可能会影响下一个操作的执行时间（操作之间不是独立的，存在关联）。
- 在评估整个操作序列最坏时间复杂度时，把每个操作的最坏情况时间复杂度加起来，有时候会过于悲观，忽略了操作之间的联系/关联。

$$C(op_1,\ldots,op_n)=C_{worst}(op_1)+\ldots+C_{worst}(op_n)$$

均摊时间复杂度

- 确定n个操作的总代价上界T(n)，每个操作的平均代价T(n)/n称为每个操作的均摊时间复杂度。
- n次操作的总代价分摊至各操作，均摊时间复杂度考虑了操作间的关联。
- 一般反映最坏情况下每个操作的平均性能。

均摊时间复杂度VS 平均时间复杂度

	输入情况	考察对象	计算方法	特点
平均时间复杂度	假定各种输入发生的概率分布	单个操作	按照假定的概率分布，对各种输入情况下所需时间做加权平均	将各操作作为独立事件分别考察，不考虑操作之间的相关性和连贯性
均摊时间复杂度	不对输入的分布做任何假设	真实可行的、足够长的连续操作序列	操作序列的总时间分摊至单次操作	对一系列操作做整体的考量

均摊分析的应用场景

- 对一个数据结构进行一组连续操作；
- 大部分情况时间复杂度都很低，只有个别情况时间复杂度比较高，且这些操作之间存在前后连贯的时序关系；
- 这时可以看能否把时间复杂度高的操作的执行时间分摊到其他低时间复杂度的操作上。