算法设计与分析

林 海

Lin.hai@whu.edu.cn

教材和考核

- 算法设计技巧与分析，M. H. Alsuwaiyel 著, 电子工业出版社，吴伟昶等译
- 算法导论（原书第3版）机械工业出版社
- 平时（作业，到课率） 30%
- 期末 70%

课程目标

- 达到目标
  - 学习各种经典算法
  - 设计算法解决问题
  - 学会思考
  - ~~并非做编程、调试代码~~

课程意义

- 意义
  - 实际应用
    - 人工智能 AlphaGo，Google疾病预测
  - 1966年至2011年的图灵奖获得者中有19人直接或间接地与算法相关

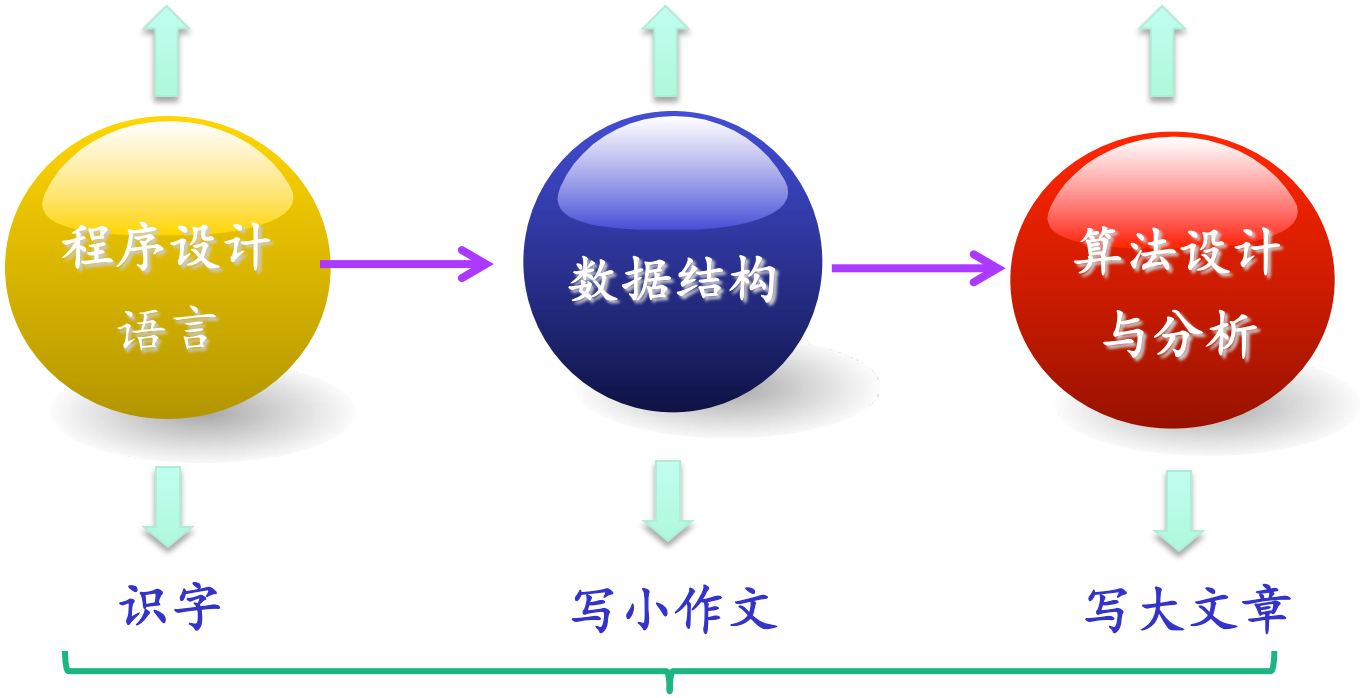
1985年	理查德·卡普	算法理论，尤其是NP-完全性理论
1986年	约翰·霍普克洛夫特	算法和数据结构的设计与分析
	罗伯特·塔扬	

  - 姚期智：Yao's min-max principle
  - 算法在计算机科学中的地位
    - 核心基础课程

基本编程

以数据结构为中心的算法设计—基本算法设计方法

通用算法设计—算法设计方法学



与语文学习过程类比

第1章 算法分析基本概念

- 算法：1. 为解决某一问题定义的计算过程；2. 是通过一个有限的指令序列集合对特定问题进行求解的一种计算执行描述
  - 输 入：有零个或多个外部量作为算法的输入。
  - 输 出：算法产生至少一个量作为输出。
- 如将一个数列进行排序
  - 输入<5,2,4,6,1,3>
  - 输出<1,2,3,4,5,6>
- 算法能解决什么？
  - DNA测对
  - 互联网：路由、搜索、加解密
  - 大数据应用

算法的五个重要的特性

（1） 有穷性：在有穷步之后结束，算法能够停机。

（2） 确定性：无二义性。

（3） 可行性：可通过基本运算有限次执行来实现，也就是算法中每一个动作能够被机械地执行。

（4） 有输入

（5） 有输出

} 表示存在数据处理

【例】考虑下列两段描述，这两段描述均不能满足算法的特性，试问它们违反了哪些特性？

```
void exam1()
{   int  n=2;
    while (n%2==0)
        n=n+2;
    printf("%d\n", n);
}
```

其中有一个死循环，违反了算法的有穷性特性。

（1）描述一

（2）描述二

```
void exam2()
{    int x，y;
     y=0;
     x=5/y;
     printf("%d，%d\n"，x，y);
}
```

其中包含除零错误，违反了算法的可行性特性

1.1算法概念：算法与程序的区别

- 算法的概念与程序十分相似,但实际上有很大不同。程序并不都满足算法所要求的上述特征, 例如有限性特征。算法代表了对特定问题的求解,而程序则是算法在计算机上的实现。因此算法也常常称为是一个能行过程。

- 如,操作系统是一种程序而不是算法

1.3 简单的算法：搜索

- 顺序搜索（线性搜索）
  - 依次搜索所有的元素

1. $j \leftarrow 1$
2. **while** $(j < n)$ **and** $(x \neq A[j])$
3. $\quad j \leftarrow j + 1$
4. **end while**
5. **if** $x = A[j]$ **then return** j **else return** 0

如果元素已经排序好了，顺序搜索效率低下

1.3 搜索：二分搜索

- 二分搜索（已经排序好）
  - 对序列进行二等分，然后判断被搜索数据在哪个部分中，相应的部分被保留，另外部分被丢弃。
  - 重复以上步骤

例如：在数组A中搜索22

$A[1\cdots14]=$ | 1 | 4 | 5 | 7 | 8 | 9 | 10 | 12 | 15 | 22 | 23 | 27 | 32 | 35 |

二分： $A[\lfloor(1+14)/2\rfloor]=A[7]=10$

22>10，所以被搜索数据在后一部分： $A[8\cdots14]=$ | 12 | 15 | 22 | 23 | 27 | 32 | 35 |

继续在此部分二分： $A[\lfloor(8+14)/2\rfloor]=A[11]=23$

22<23，所以被搜索数据在前一部分： $A[8\cdots10]=$ | 12 | 15 | 22 |

重复以上步骤

1.3 二分搜索

- 二分搜索
  - 对序列进行二等分，然后判断被搜索数据在哪个部分中，相应的部分被保留，另外部分被丢弃。
  - 重复以上步骤

输入：非降序排列的数组A[1...n]和元素x
输出：如果x=A[j], 1 ≤j ≤n,则输出j,否则输出0.

```
1.  low←1; high ←n; j←0
2.  while(low ≤high)  and  (j=0)
3.      mid ← ⌊( low+high)/2⌋
4.      if x=A[mid] then  j ←mid
5.      else if x<A[mid]  then  high ←mid-1
6.      else low ←mid+1
7.  end while
8.  return j
```

1.3 二分搜索算法分析

- 最好情形：比较1次
- 最坏情形：比较$\lfloor \log n \rfloor+1$次
  - 每次循环都要抛弃一些元素，例如第二次循环时，剩余元素为A[1...mid-1]或A[mid+1...n]，不妨设为A[mid+1...n]，则剩余的元素个数是$\lfloor n/2 \rfloor$
  - 第j次while循环时，剩余元素的个数是$\lfloor n/2^{j-1} \rfloor$
  - 或者找到x，或者程序在子序列长度达到1时终止搜索，此时$\lfloor n/2^{j-1} \rfloor=1 \Leftrightarrow 1 \le n/2^{j-1} <2 \Leftrightarrow 2^{j-1} \le n< 2^j$
    $\Leftrightarrow \log n<j\le \log n+1 \Leftrightarrow j=\lfloor \log n \rfloor+1$

1.4 排序：合并排序

- 已知：两个已按升序排列好的子数组A[p…q]和A[q+1…r]
- 输出：A[p…r]合并以上两个子数组，且也按照升序排序
- 算法:
  - 两个指针s和t分别指向两数组的起始位置，并设置一个空数组B[p…r]
  - 比较指针指向的元素，将小的元素添加到B中，并移动相应的指针
  - 重复步骤2直到其中一个指针指向末尾位置，此时将另外一个数组的元素全部拷贝到B

1.4 排序：合并排序

算法 1.3　MERGE

```
1. comment: B[p…r]是个辅助数组
2. s ← p; t ← q+1; k ← p
3. while s≤q and t≤r
4.     if A[s]≤A[t] then
5.         B[k] ← A[s]
6.         s ← s+1
7.     else
8.         B[k] ← A[t]
9.         t ← t+1
10.    end if
11.    k ← k+1
12. end while
13. if s = q+1 then B[k…r] ← A[t…r]
14. else B[k…r] ← A[s…q]
15. end if
16. A[p…r] ← B[p…r]
```

1.4 合并排序分析

假设合并两个数组分别为n_1和n_2，且$n=n_1+n_2$

- 最好情形：比较$\min(n_1, n_2)$次 

- 最坏情形：比较n-1次 

- 如果两数组大小分别为⌊n/2⌋和⌈n/2⌉，则比较的次数是⌊n/2⌋到n-1之间

1.5 排序：选择排序

- 将一个数组进行排序
- 算法:
  - 找到最小的元素，将其和第一个元素（A[1]）交换
  - 在剩余的元素中重复以上步骤

采用简单选择排序方法对(2，1，3，4，5) 进行排序

初始关键字	2	1	3	4	5
i=0	1	2	3	4	5
i=1	1	2	3	4	5
i=2	1	2	3	4	5
i=3	1	2	3	4	5

没有记录移动

任何情况下：都要做n−1趟

1.5 排序：选择排序

算法 1.4 SELECTIONSORT

输入： n 个元素的数组 $A[1\cdots n]$。

输出： 按非降序排列的数组 $A[1\cdots n]$。

```
1. for i ← 1 to n − 1
2.     k ← i
3.     for j ← i + 1 to n        {查找第 i 小的元素}
4.         if A[j] < A[k] then k ← j
5.     end for
6.     if k ≠ i then   交换 A[i]与 A[k]
7. end for
```

1.5 选择排序分析

算法比较次数:

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

无论原数组的顺序如何，选择排序的比较次数是固定的

1.6 插入排序

- 算法（升序排序）：
  - 排序数组的第一个元素A[1]
  - 从未排序的数组中取第一个元素，将其插入到已经排序好的数组中
    - 依次（从后往前）和已经排序好的数组进行比较
    - 如果元素小于数组元素，则交换，直到大于数组元素，停止比较
  - 重复以上步骤

1.6 插入排序

算法 1.5　INSERTIONSORT

输入： n 个元素的数组 $A[1\cdots n]$。

输出： 按非降序排列的数组 $A[1\cdots n]$。

```
1. for i ← 2 to n
2.     x ← A[i]
3.     j ← i − 1
4.     while (j > 0) and (A[j] > x)
5.         A[j + 1] ← A[j]
6.         j ← j − 1
7.     end while
8.     A[j + 1] ← x
9. end for
```

【例10-1】 设待排序的表有10个元素，其关键字分别为（9，8，7，6，5，4，3，2，1，0）。说明采用直接插入排序方法进行排序的过程。

初始：（9，8，7，6，5，4，3，2，1，0）

i=1：（8，9，7，6，5，4，3，2，1，0）

i=2：（7，8，9，6，5，4，3，2，1，0）

i=3：（6，7，8，9，5，4，3，2，1，0）

i=4：（5，6，7，8，9，4，3，2，1，0）

i=5：（4，5，6，7，8，9，3，2，1，0）

i=6：（3，4，5，6，7，8，9，2，1，0）

i=7：（2，3，4，5，6，7，8，9，1，0）

i=8：（1，2，3，4，5，6，7，8，9，0）

i=9：（0，1，2，3，4，5，6，7，8，9）

1.6 插入排序分析

- 最好情形（数组已经排序好）：比较n-1次
- 最坏情形：

$$\sum_{i=2}^{n} i - 1 = \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}$$

比较次数和n^2成正比

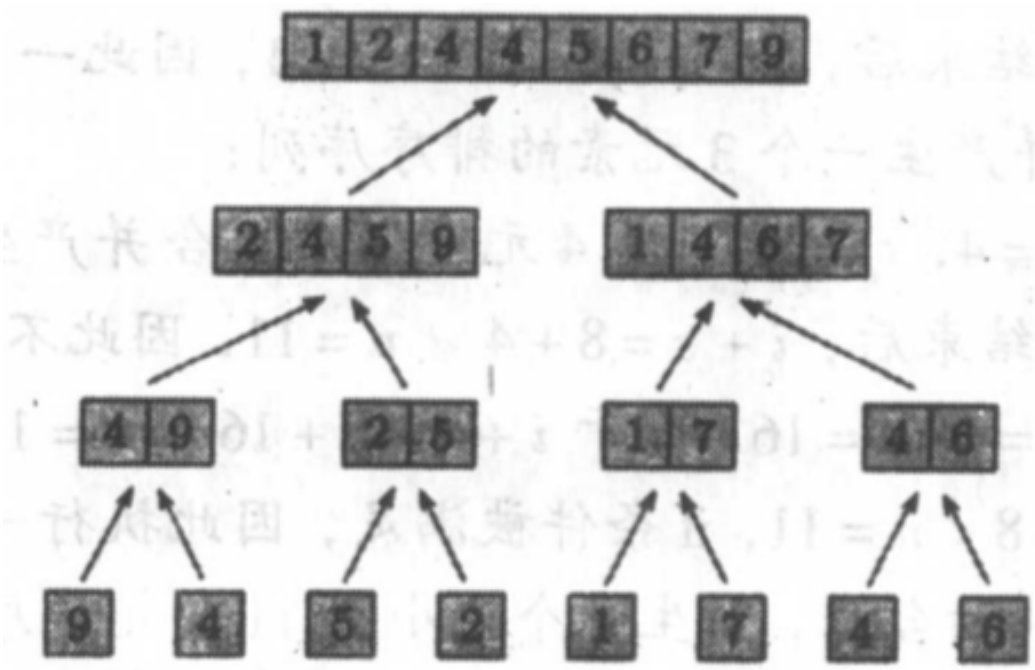
1.7 排序：自底向上合并排序

- 算法（升序排序）：
  - 将数组的每个元素看成一个独立数组
  - 将这些数组进行两组合并，形成新的独立数组
    - 通过前面MERGE算法
  - 重复以上步骤

1.7 排序：自底向上合并排序

- 如

9	4	5	2	1	7	4	6



1.7 排序：自底向上合并排序

算法 1.6　BOTTOMUPSORT

输入： n 个元素的数组 $A[1\cdots n]$。

输出： 按非降序排列的数组 $A[1\cdots n]$。

```
1. t ← 1
2. while t < n
3.     s ← t;   t ← 2s;   i ← 0
4.     while i + t ≤ n
5.         MERGE (A, i + 1, i + s, i + t)
6.         i ← i + t
7. end while
8.     if i + s < n then MERGE (A, i + 1, i + s, n)
9. end while
```

1.7 排序：合并排序算法分析

- 设元素个数n为2的幂（满二叉树）
  - 在第1次迭代中（最低层），对2^0个元素的数组（共n个），进行了$n/2^1$次合并，每次合并比较1次
  - 在第2次迭代中（第2层），对2^1个元素的数组（共n/2个），进行了$n/2^2$次合并，每次合并比较次数在2和(4-1)次之间
  - 在第j次迭代中（第j层），对2^{j-1}个元素的数组（共$n/2^{j-1}$个），进行了$n/2^j$次合并，每次合并比较次数在 2^{j-1}和(2^j-1)次之间
- 另k=log n，那么最好的情形

$$\sum_{j=1}^{k}\left(\frac{n}{2^j}\right)2^{j-1} = \sum_{j=1}^{k}\frac{n}{2} = \frac{kn}{2} = \frac{n\ \log\ n}{2}$$

- 最坏的情形

$$\begin{aligned}\sum_{j=1}^{k}\frac{n}{2^j}(2^j-1) &= \sum_{j=1}^{k}\left(n-\frac{n}{2^j}\right)\\ &= kn - n\sum_{j=1}^{k}\frac{1}{2^j}\\ &= kn - n\left(1-\frac{1}{2^k}\right)\\ &= kn - n\left(1-\frac{1}{n}\right)\\ &= n\ \log\ n - n + 1\end{aligned}$$

1.8 时间复杂性

- 插入排序每条语句执行的次数

	代价	次数
1. **for** $i \leftarrow 2$ **to** n	c_1	n
2. $x \leftarrow A[i]$	c_2	$n-1$
3. $j \leftarrow i-1$	c_4	$n-1$
4. **while** $(j>0)$ **and** $(A[j]>x)$	c_5	$\sum_{j=2}^{n} t_j$
5. $A[j+1] \leftarrow A[j]$	c_6	$\sum_{j=2}^{n} (t_j-1)$
6. $j \leftarrow j-1$	c_7	$\sum_{j=2}^{n} (t_j-1)$
7. **end while**		
8. $A[j+1] \leftarrow x$	c_8	$n-1$
9. **end for**		

总代价 $$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7 \sum_{j=2}^{n} (t_j - 1) + c_8(n-1)$$

1.8 时间复杂性

最佳情况下（已经排序好）总代价

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1)$$
$$= (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8)$$

$$an + b$$

最坏情况下（反向排序）总代价

$$\sum_{j=2}^{n} j = \frac{n(n+1)}{2} - 1$$

$$\sum_{j=2}^{n} (j-1) = \frac{n(n-1)}{2}$$

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5\left(\frac{n(n+1)}{2} - 1\right)$$
$$+ c_6\left(\frac{n(n-1)}{2}\right) + c_7\left(\frac{n(n-1)}{2}\right) + c_8(n-1)$$
$$= \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}\right)n^2 + \left(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8\right)n$$
$$- (c_2 + c_4 + c_5 + c_8)$$

$$an^2 + bn + c$$

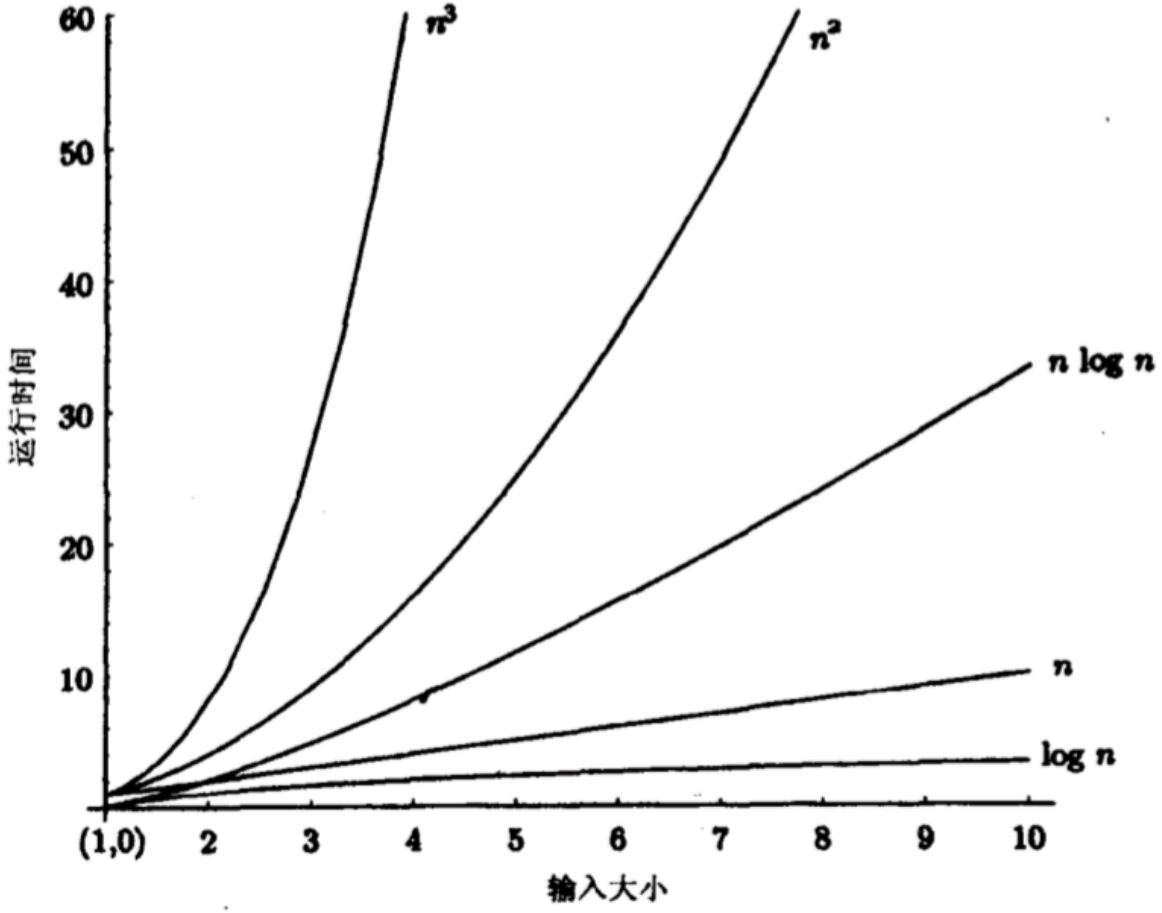
1.8 时间复杂性

考察: an^2+bn+c

- 当n很大时，a不起作用
  - 如将an^2和dn^3比较，当n很大时，系数a和d不起作用
- 当n很大时，低阶项（bn+c）不起作用
- 所以上述算法的运行时间主要是n^2其作用，称插入排序的运行时间（时间复杂度）为“n^2阶”

1.8 时间复杂性

- 几种典型的时间复杂度函数



1.8.2 O符号

定义 1.2　令 $f(n)$和 $g(n)$是从自然数集到非负实数集的两个函数，如果存在一个自然数 n_0 和一个常数 $c>0$，使得

$$\forall n \geqslant n_0, f(n) \leqslant cg(n)$$

则称 $f(n)$为 $O(g(n))$。

- g(n) 是f (n)的一个上界
- f (n) 的阶不高于g(n) 的阶

因此，如果 $\lim_{n\to\infty} f(n)/g(n)$存在，那么

$$\lim_{n\to\infty}\frac{f(n)}{g(n)} \neq \infty \text{ 蕴含着 } f(n)=O(g(n))$$

1.8.2 O符号

- 例： $f(n)=n^2$，$g(n)=n^3$。因为：存在$n_0=1$，$C=1$，当$n\geq n_0$时，有$n^2\leq Cn^3$，所以：$n^2=O(n^3)$
- 例： $f(n)=n^2$，$g(n)=n^2$。因为：存在$n_0=1$，$C=1$，$n\geq n_0$时，有$n^2\leq Cn^2$，所以：$n^2=O(n^2)$
- 例： $f(n)=n^2+n\log(n)$，$g(n)=n^2$。同样有$n^2+n\log(n)=O(n^2)$
- 例： $f(n)=an^2+n\log(n)$，$g(n)=n^2$。同样有$an^2+n\log(n)=O(n^2)$
- 例： $f(n)=an^2+n\log(n)+c$，$g(n)=n^2$。同样有$an^2+n\log(n)+c=O(n^2)$

小结

- 在进行阶的运算时，常系数、低的阶以及常数项可以忽略。
- 根据O的定义，得到的是在问题规模充分大时，算法复杂度的一个上界。上界的阶越低则评估越有价值。
- 运算规则
  - $O(f) + O(g) = O(\max(f, g))$;
  - $O(f) \cdot O(g) = O(f \cdot g)$;
  - $O(C \cdot f(n)) = O(f(n))$;
  - $f = O(f)$;

1.8.3 Ω符号

定义 1.3 设 $f(n)$和 $g(n)$是从自然数集到非负实数集的两个函数，如果存在一个自然数 n_0 和一个常数 $c>0$，使得

$$\forall n \geqslant n_0, f(n) \geqslant cg(n)$$

则称 $f(n)$为 $\Omega(g(n))$。

- g(n) 是f (n)的一个下界
- f (n) 的阶不低于g(n) 的阶

因此，如果 $\lim_{n\to\infty} f(n)/g(n)$存在，那么

$$\lim_{n\to\infty}\frac{f(n)}{g(n)} \neq 0 \text{ 蕴含着 } f(n)=\Omega(g(n))$$

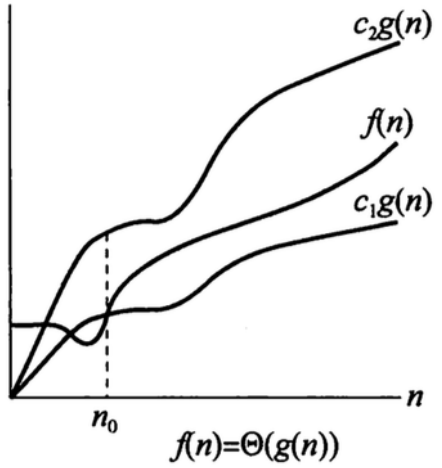
1.8.4 Θ符号

定义 1.4　设 $f(n)$和 $g(n)$是从自然数集到非负实数集的两个函数，如果存在一个自然数 n_0 和两个正常数 c_1 和 c_2，使得

$$\forall n \geqslant n_0, c_1 g(n) \leqslant f(n) \leqslant c_2 g(n)$$

则称 $f(n)$是 $\Theta(g(n))$的。

- f (n) 和g(n) 同阶



1.8.4 Θ符号

因此，如果 $\lim_{n\to\infty} f(n)/g(n)$存在，那么

$$\lim_{n\to\infty}\frac{f(n)}{g(n)}=c \text{ 蕴含} f(n)=\Theta(g(n))$$

其中 c 必须是一个大于0的常数。

$f(n)=\Theta(g(n))$，当且仅当 $f(n)=O(g(n))$并且 $f(n)=\Omega(g(n))$

1.8.6 o符号

定义 1.5 令 $f(n)$和 $g(n)$是从自然数集到非负实数集的两个函数，如果对每一个常数 $c>0$，存在一个正整数 n_0，使得对于所有的 $n \geqslant n_0$，都有 $f(n)<cg(n)$成立，则称 $f(n)$是 $o(g(n))$的。

O记号与o记号的定义类似。主要的区别是在 $f(n)=O(g(n))$中，界 $0 \leqslant f(n) \leqslant cg(n)$对某个常量 $c>0$ 成立，但在 $f(n)=o(g(n))$中，界 $0 \leqslant f(n)<cg(n)$对所有常量 $c>0$ 成立。直观上，在o记号中，当 n 趋于无穷时，函数 $f(n)$相对于 $g(n)$来说变得微不足道了，即

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0 \tag{3.1}$$

$f(n)=o(g(n))$，当且仅当 $f(n)=O(g(n))$，但 $g(n) \neq O(f(n))$

算法复杂性分析：性质

传递性：

$f(n) = \Theta(g(n))$ 且 $g(n) = \Theta(h(n))$ 蕴涵 $f(n) = \Theta(h(n))$

$f(n) = O(g(n))$ 且 $g(n) = O(h(n))$ 蕴涵 $f(n) = O(h(n))$

$f(n) = \Omega(g(n))$ 且 $g(n) = \Omega(h(n))$ 蕴涵 $f(n) = \Omega(h(n))$

$f(n) = o(g(n))$ 且 $g(n) = o(h(n))$ 蕴涵 $f(n) = o(h(n))$

自反性：

$$f(n) = \Theta(f(n))$$

$$f(n) = O(f(n))$$

$$f(n) = \Omega(f(n))$$

对称性：

$f(n) = \Theta(g(n))$ 当且仅当 $g(n) = \Theta(f(n))$

转置对称性：

$f(n) = O(g(n))$ 当且仅当 $g(n) = \Omega(f(n))$

1.8.5 例子

证明$\frac{1}{2}n^2-3n=\Theta(n^2)$

确定正常量 c_1、c_2 和 n_0，使得对所有 $n\geqslant n_0$，有：

$$c_1n^2\leqslant\frac{1}{2}n^2-3n\leqslant c_2n^2$$

用 n^2 除上式得：

$$c_1\leqslant\frac{1}{2}-\frac{3}{n}\leqslant c_2$$

选择

$$c_2\geqslant 1/2 \qquad c_1\leqslant 1/14 \qquad n_0=7$$

可以证明$\frac{1}{2}n^2-3n=\Theta(n^2)$

1.8.5 例子

假设 $f(n)$与 $g(n)$都是渐近非负函数。使用 Θ 记号的基本定义来证明 $\max(f(n), g(n)) = \Theta(f(n)+g(n))$。

C1，C2, n0使得当n>n0时，

C1（f(n)+g(n)）<=max(f(n),g(n))<= C2（f(n)+g(n)）

取C1=0.5 C2=1，等式成立

练习

解释为什么“算法 A 的运行时间至少是 $O(n^2)$”这一表述是无意义的。

因为O(n^2)可为任意（小于等于）二阶函数，如f(n)=0
f(n)=O(n^2)　注意可写成f(n)∈ O(n^2)
即算法A的运行时间至少为f（n）（至少为0）

1.8.5 例子

证明：$\sum_{j=1}^{n} \log j = \Theta(n \log n)$

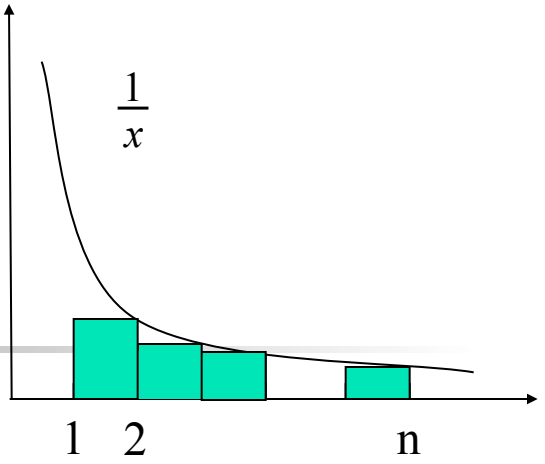
$$\sum_{j=1}^{n} \log j \leqslant \sum_{j=1}^{n} \log n \quad \Rightarrow \quad \sum_{j=1}^{n} \log j = O(n \log n)$$

$$\sum_{j=1}^{n} \log j \geqslant \sum_{j=1}^{\lfloor n/2 \rfloor} \log\left(\frac{n}{2}\right) = \lfloor n/2 \rfloor \log\left(\frac{n}{2}\right) = \lfloor n/2 \rfloor \log n - \lfloor n/2 \rfloor$$

$$\Rightarrow \quad \sum_{j=1}^{n} \log j = \Omega(n \log n)$$

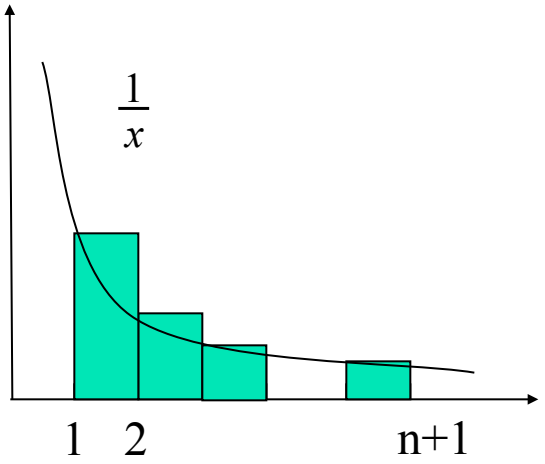
所以：$\sum_{j=1}^{n} \log j = \Theta(n \log n)$

1.8.5 例子



证明调和级数 $H_n = \sum_{j=1}^{n} \frac{1}{j} = \Theta(\log n)$

$$\sum_{j=1}^{n} \frac{1}{j} = 1 + \sum_{j=2}^{n} \frac{1}{j} \leqslant 1 + \int_1^n \frac{dx}{x} = 1 + \ln n$$



$$\sum_{j=1}^{n} \frac{1}{j} \geqslant \int_1^{n+1} \frac{dx}{x} = \ln(n+1)$$

$$\Rightarrow \quad \ln(n+1) \leqslant \sum_{j=1}^{n} \frac{1}{j} \leqslant \ln n + 1$$

$$\Rightarrow \quad \frac{\log(n+1)}{\log e} \leqslant \sum_{j=1}^{n} \frac{1}{j} \leqslant \frac{\log n}{\log e} + 1$$

$$\Rightarrow \quad \sum_{j=1}^{n} \frac{1}{j} = O(\log n) \text{ 且 } \sum_{j=1}^{n} \frac{1}{j} = \Omega(\log n)$$

1.9 空间复杂性

- 空间复杂度
  - 为了求解问题的实例而执行的计算步骤所需要的内存空间数目
- 空间复杂度不可能超过时间复杂度
  $S(n)=O(T(n))$
- 例子
  - 选择排序的空间复杂度为$\Theta(1)$
  - 合并排序的空间复杂度为$\Theta(n)$

例子

请对某歌曲库里的歌曲播放设计一个算法，算法需要能够实现每首歌曲的随机播放概率应该正比于它的评分（评分为 10 分制，精确到小数点后一位），例如评分 9.1 的歌曲和评分 7.9 的歌曲，播放的次数大概是 91：79。要求给出算法的思路（举例说明更好）、伪代码和复杂度（时间复杂度和空间复杂度）。注意：要求歌曲的播放是随机的。（本题评分除了考虑算法的正确性外，也考虑算法的复杂度）

例子

算法1：将所有的歌曲按照评分复制其编号，如歌曲1的评分为5.5，就将1复制55，歌曲10评分为6.6，则将10复制66。然后随机的从这些编号中选取一个编号，选到的编号即为播放曲目。
此算法的时间复杂度为O(1)，空间复杂度为n*E[歌曲的评分]。

算法2：将所有的歌曲按照评分排列，并根据评分生成随机区间，然后总区间的一个随机值，这个值落在哪个区间，即播放相应的歌曲。如有4首歌其评分为[1,1.5,2,2]，则生成区间[0-1, 1-2.5, 2.5-4.5, 4.5-6.5]4个区间，然后在[0-6.5]取一个随机数，随机数落在哪个区间，播放相应的歌曲。
此算法的时间复杂度为O(logn)，空间复杂度为n*2。

1.11估计算法的时间复杂度

- 计算迭代次数
- 频度分析
- 使用递归方程

1.11 估计算法的运行时间

- 计算迭代次数
  - 通常，程序的运行时间和程序的迭代次数成比例。因此计算程序的迭代次数就可以作为算法运行时间的指示器。这在很多算法中都可以得到应用，如查找、排序等等

1.11 估计算法的运行时间

输入： n ($n=2^k$， k 为某一正整数)

输出：count （迭代次数）

```
1. count ← 0
2. while n ≥ 1
3.   for j ← 1 to n
4.     count ← count+1          //执行一次耗费时间设为a
5.   end for
6.   n ← n/2                    //执行一次耗费时间设为d
7. end while
8. return count
```

分析：while 迭代的次数是k +1次(因为n≥1 可以写成$n\geq 2^0$, 运行过程 $n=2^k \rightarrow 2^0$)，$k = \log n$ 。在每次while 循环里面for 依次执行n，n/2,n/4,…,1 次，所以，算法的时间复杂度为：

$$T(n)=\sum_{j=0}^{k} a(n/2^j)=an\sum_{j=0}^{k}2^{-j}=an(2-1/2^k)=2an-a=\Theta(n)$$

1.11 估计算法的运行时间

- 为什么在上面计算算法的时间复杂度时不考虑step 6 ,而只是考虑step 4呢？
- 如果同时考虑 step 4和step 6，我们有：

$$T(n) = (\sum_{j=0}^{k} n/2^j)a + (k+1)d$$

$$= an\sum_{j=0}^{k} 2^j + (k+1)d = na(2 - 1/2^k) + (k+1)d$$

$$= 2an - a + d + d\log n = \Theta(n)$$

小结：使用计算迭代次数的技术来分析算法的时间复杂度时，只需要考虑最深层次的迭代。

1.11 估计算法的运行时间

输入：正整数n

输出：step 5 的执行次数（复杂度）

1. count ← 0
2. for i ← 1 to n
3. m← ⌊n/i⌋
4. for j ← 1 to m
5. count ← count+1
6. end for
7. end for
8. return count

分析：Step5 的执行次数依次为：
⌊n/1⌋ , ⌊n/2⌋, ⌊n/3⌋, …, ⌊n/n⌋

$$T(n)=\sum_{i=1}^{n}\left\lfloor\frac{n}{i}\right\rfloor$$

因为 $\sum_{i=1}^{n}(\frac{n}{i}-1)\le\sum_{i=1}^{n}\left\lfloor\frac{n}{i}\right\rfloor\le\sum_{i=1}^{n}\frac{n}{i}$

所以 $T(n)=\Theta(n\log n)$

1.11 估计算法的运行时间

- 频度分析
  - 对于有些算法，计算迭代次数非常麻烦，有时甚至是不可能的。这时候，可以使用频度分析。

定义 1.6 如果算法中的一个元运算具有最高频度，所有其他元运算频度均在它的频度的常数倍内，则称这个元运算为基本运算。

在MEGE算法中，赋值运算具有最大频度
- 将A的每个元素移到B （语句5和8）
- 又将B的每个元素移到A （语句16）

所以算法复杂度为2n

1.11 估计算法的运行时间

使用递归关系（二分搜索）

输入：非降序排列的数组A[1…n]和元素x
输出：如果x=A[j], 1 ≤j ≤n,则输出j,否则输出0.

```
1.  low←1; high ←n; j←0
2.  while(low ≤high)  and  (j=0)
3.      mid ← ⌊( low+high)/2⌋
4.      if x=A[mid] then  j ←mid
5.      else if x<A[mid]  then  high ←mid-1
6.      else low ←mid+1
7.  end while
8.  return j
```

$$T(n)=\begin{cases}1 & if\ n=1\\ T(n/2)+1 & if\ n\geq 2\end{cases}$$

$$\Rightarrow T(n)=T(n/2)+1=T(n/2^2)+1+1$$

$$=T(n/2^{\log n})+\log n=1+\log n=\Theta(\log n)$$

详见递归章节

1.12最坏情况和平均情况分析

- 算法运行时间与输入规模有关，同时也与输入的元素的性质有关，
- 如对于InsertSort算法在n个元素的序列中，当序列是降序、升序和随即机排列时其元素比较次数分别是：$n^2/2$，n，$n^2/4$.

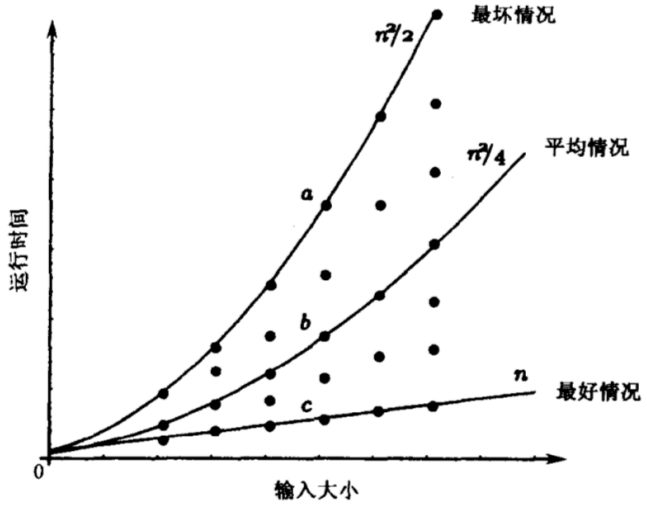


图 1.6　算法 INSERTIONSORT 的性能：最坏情况、平均情况和最好情况

1.12最坏情况和平均情况分析

- 最坏情况分析是对输入中选择代价最大的分析；
- 最坏情况下许多算法的上下界合一，因此可以说算法在最坏情况下以**θ(f(n))** 运行。**如算法LinerSearch是O(n)**和Ω(1)，在最坏情况下该算法的运行时间既是O(n)也是Ω(n)，也就是**θ(n).**
  - 并不是最坏情况下，上下界总是重合的。

如：**if n 是奇数 then k:=BinarySearch(A,x)**
else k:=LinerSearch(A,x)
最坏情况是**O(n)**和Ω(logn)

1.12最坏情况和平均情况分析

- **平均情况分析**：通常考虑匀分布情况下的复杂度
- 如插入排序中，将第i个元素插入到前面已经排序好的数组中
  - 插入到第1个位置，比较i-1次
  - 插入到其他位置（位置j），比较i-j+1此
  - 平均比较次数为：

$$\frac{i-1}{i}+\sum_{j=2}^{i}\frac{i-j+1}{i}=\frac{i-1}{i}+\sum_{j=1}^{i-1}\frac{j}{i}=1-\frac{1}{i}+\frac{i-1}{2}=\frac{i}{2}-\frac{1}{i}+\frac{1}{2}$$

1.12最坏情况和平均情况分析

- 插入排序总的平均比较次数为：

$$\sum_{i=2}^{n}\left(\frac{i}{2}-\frac{1}{i}+\frac{1}{2}\right)=\frac{n(n+1)}{4}-\frac{1}{2}-\sum_{i=2}^{n}\frac{1}{i}+\frac{n-1}{2}=\frac{n^2}{4}+\frac{3n}{4}-\sum_{i=1}^{n}\frac{1}{i}$$