1.2 算法的复杂性分析

●**算法的复杂性:指执行算法所消耗或占用的资源的数量**

●**一个算法所需资源越多,我们就说它的复杂性越高,反之则低.**

算法的复杂性
- 空间复杂性
- 时间复杂性

考虑空间复杂性的理由

- 在多用户系统中运行时，需指明分给该程序的内存大小；
- 需预先知道计算机系统是否有足够的内存来运行该程序；
- 一个问题可能有若干个不同的内存解决方案，从中择取；
- 用空间复杂性估计一个程序可能解决的问题的最大规模。

考虑时间复杂性的理由

- 某些计算机用户需要提供程序运行时间的上限（用户可接受的）；
- 所开发的程序需要提供一个满意的实时反应。

算法分析:(渐进算法分析):**对执行算法所消耗或占用的资源进行估算,给出算法耗费的限界函数.**

需解决两个问题:

- 如何度量复杂性?
- 如何分析复杂性?

1.复杂性的计量

算法的复杂性: 与算法求解**问题的规模n**,算法的**输入数I** 及**算法本身**有关.

问题的规模n:指问题的输入数据或初始数据的量.

在不同的问题中, n有不同的表现形式:

例如	在数组中找分量c,	n:数组中分量的个数
	两个矩阵相乘,	n:矩阵的维数
	表中排序,	n:数组中分量的个数
	遍历一棵树,	n:树中节点数

5*5的矩阵相乘与10*10矩阵的矩阵相乘所需时间空间均不相同；

<问题规模不同>

找c在数组A中的位置,c=A(1),与c=A(100),所需时间显然也不同

<输入数不同>

顺序查找还是折半查找速度也是不一样的.

<算法不同>

算法效率与计算机的性能有关，但此因素对所有算法的影响相同。

令 n: 问题规模 I: 输入数据 A: 算法本身 C: 算法的复杂性, 则

$$C=f(n, I, A)$$

将时间复杂性和空间复杂性分别考虑,并用T和S表示. 则有:

$$T=T(n, I, A) \qquad S=S(n, I, A)$$

将算法A 隐含在函数名中,不同函数名代表不同算法, 则简化为

$$T=T(n, I) \qquad S=S(n, I)$$

- **仅以时间复杂性为例将复杂性函数具体化.**

设一台抽象计算机提供的元运算有k种, 记作 $O_1,\ldots,O_k$;

设这些元运算每执行一次所需时间分别为 $t_1,\ldots,t_k$;

设在算法A中用到O_i的次数为 e_i, $i=1,\ldots,k$, 则 $e_i = e_i(n, I)$

$$T=T(n,I)=\sum_{i=1}^{k} t_i \cdot e_i(n, I)$$

- **当问题的规模 n和算法确定后,T是输入变元 i 的函数.**

说明：

我们不可能对规模为n的每一种合法输入I都计算e_i次，因为输入可能是无穷集合，我们只能对规模为n的问题的某类具有代表性的合法输入统计相应的e_i.

最好情况：$T_{min}(n) = \min_{I \in D_n} T(n,I) = \min_{I \in D_n} \sum_{i=1}^{k} t_i \cdot e_i(n,I)$

$= \sum_{i=1}^{k} t_i \cdot e_i(n,\tilde{I}) = T(n,\tilde{I})$

D_n中达到$T_{min}(n)$的一个输入.

最坏情况：$T_{max}(n) = \max_{I \in D_n} T(n,I) = \max_{I \in D_n} \sum_{i=1}^{k} t_i \cdot e_i(n,I)$

$= \sum_{i=1}^{k} t_i \cdot e_i(n,I^*) = T(n,I^*)$

D_n中达到$T_{max}(n)$的一个输入.

平均情况：$T_{avg}(n) = \sum_{I \in D_N} P(I) \cdot T(n,I) = \sum_{I \in D_n} P(I) \cdot \sum_{i=1}^{k} t_i \cdot e_i(n,I)$

$P(I)$出现输入为I的概率

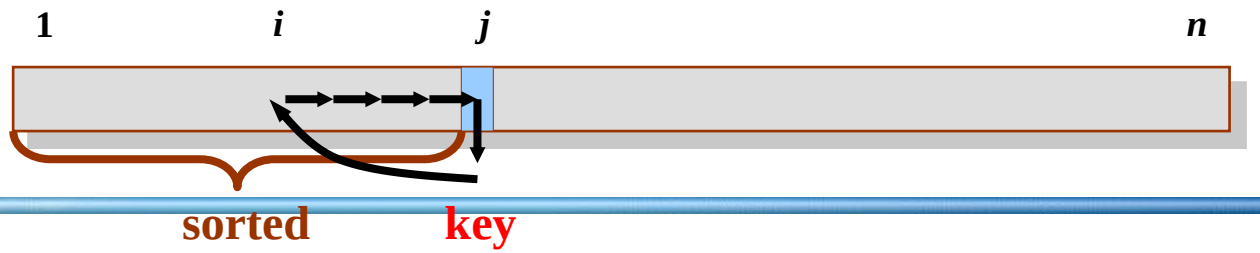
其中

实践表明，最有使用价值的且操作性好的是最坏时间复杂性.

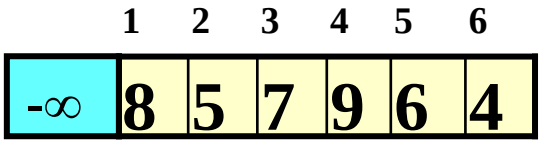
例 题 1-1 插入排序.长度为*n*的一个待排序数组A[1..n],算法在数组A中重排这些数,其中使用常数个外部存储.

- 输入: n个数的一个序列 $<a_1,a_2,\ldots,a_n>$
- 输出: 输入序列的一个排列 $<a'_1,a'_2,\ldots,a'_n>$,满足 $a'_1 \le a'_2 \le \ldots \le a'_n$
- 实例
  --输入: <8,5,7,9,6,4>
  --输出: <4,5,6,7,8,9>

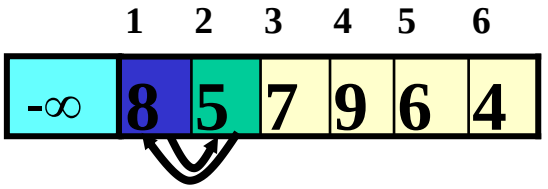
```
Insertion-Sort(A)                    //插入排序算法
1   for j=2 to A.length
2   key=A[j]
3   //将A[j]插入到数组A[1..j-1]中

4   i=j-1
5   while i>0 and A[i]>key
6       A[i+1]=A[i]
7       i=i-1
8    A[i+1]=key
```



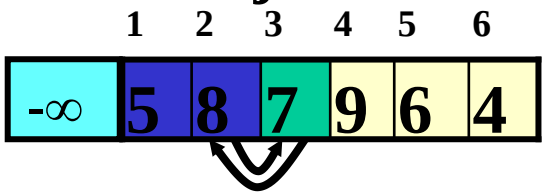
初始



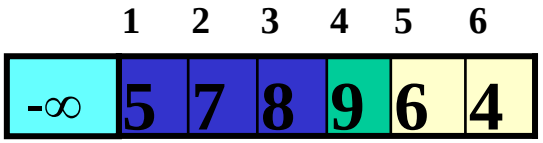
j=2



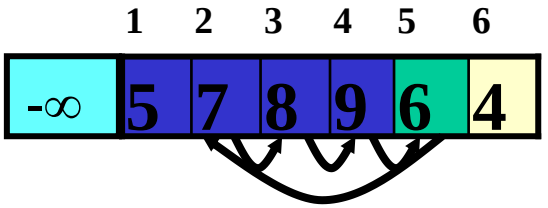
j=3



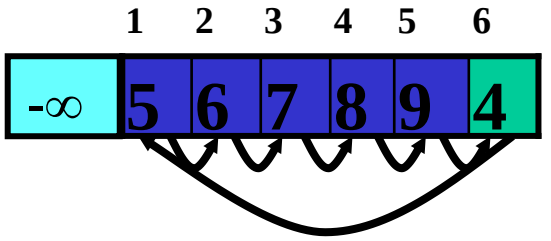
j=4



j=5



j=6



t 表示每次执行的时间，
e_j 表示对应值 j 执行的次数

Insertion-Sort(A)	代价	次数
1 **for** j=2 **to** $A.length$	t_1	n
2 key=A[j]	t_2	$n-1$
3 将A[j]插入到数组A[1..$j-1$]中	0	$n-1$
4 i=$j-1$	t_4	$n-1$
5 **while** i>0 and A[i]>key	t_5	$\sum_{j=2}^{n} e_j$
6 A[i+1]=A[i]	t_6	$\sum_{j=2}^{n} (e_j - 1)$
7 i=i-1	t_7	$\sum_{j=2}^{n} (e_j - 1)$
	t_8	$n-1$

循环语句退出时，执行测试的次数比执行循环体的次数多1

插入排序.运行时间 $T(n)$,得到

$$T(n)=t_1n+t_2(n-1)+t_4(n-1)+t_5\sum_{j=2}^{n}e_j+t_6\sum_{j=2}^{n}(e_j-1)+t_7\sum_{j=2}^{n}(e_j-1)+t_8(n-1)$$

如果输入数组已排序，则出现最佳情况运行时间 $T(n)$,得到：

$$T(n)=t_1n+t_2(n-1)+t_4(n-1)+t_5(n-1)+t_8(n-1)$$

$$=(t_1+t_2+t_4+t_5+t_8)n-(t_2+t_4+t_5+t_8)$$

我们把该运行时间表示为$an+b$，其中常量a和b依赖于语句代价 t_i

如果输入数组已反向排序，必须将每个元素$A[j]$与整个已排序子数组$A[1..j-1]$比较，有$e_j=j$,则出现最坏情况运行时间 $T(n)$,得到：

$$T(n)=t_1n+t_2(n-1)+t_4(n-1)+t_5(\frac{n(n+1)}{2}-1)$$

$$+t_6(\frac{n(n-1)}{2}-1)+t_7(\frac{n(n-1)}{2}-1)+t_8(n-1)$$

$$=(\frac{t_5}{2}+\frac{t_6}{2}+\frac{t_7}{2})n^2+\left(t_1+t_2+t_4+\frac{t_5}{2}-\frac{t_6}{2}-\frac{t_7}{2}+t_8\right)n$$

$$-(t_2+t_4+t_5+t_8)$$

我们把该运行时间表示为an^2+bn+c，其中常量a，b，c 依赖于语句代价 t_i

平均情况:所有数据的出现概率相等$e_j=(j+1)/2$，

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

运行时间 $T(n)$,得到：

$$\begin{aligned} T(n) &= t_1 n + t_2(n-1) + t_4(n-1) + t_5(n^2+3n-4)/4 \\ &\quad + t_6(n^2-3n+2)/4 + t_7(n^2-3n+2)/4 + t_8(n-1) \\ &= (t_5/4 + t_6/4 + t_7/4)n^2 + (t_1 + t_2 + t_4 + 3t_5/4 - 3t_6/4 \\ &\quad - 3t_7/4 + t_8)n - (t_2 + t_4 + t_5 - t_6/2 - t_7/2 + t_8) \end{aligned}$$

我们把该运行时间表示为an^2+bn+c，其中常量a，b，c 依赖于语句代价 t_i

2.渐进性态的阶

设f(n)和 $g(n)$ 是定义在正整数集上的正函数,

(1)**大O表示法**（算法运行时间的上限 ）

若∃正常数c和 自然数N_0 使得当 $n \geq N_0$ 时,有$f(n) \leq cg(n)$
则称函数 $f(n)$在n充分大时有**上界**, 且 $g(n)$是它一个上界.
记为 $f(n) = O(g(n))$, 也称 $f(n)$ **的阶不高于**$g(n)$ **的阶**.

例如 $3n=O(n)$, $n+1024=O(n)$, $2n^2+11n-10=O(n^2)$

$n^2=O(n^3)$? √ $n^3=O(n^2)$? ≠

三点注意:

1. **用来作比较的函数 g(n)应该尽量接近 f(n).**

   例如 $3n+2=O(n^2)$ 松散的界限；$3n+2=O(n)$ 较好的界限

2. **不要产生错误界限。**

   例如 $n^2+100n+6$，当 $n<3$ 时，$n^2+100n+6<106n$，由此就认为 $n^2+100n+6 = O(n)$.

3. **f(n)=O(g(n))不能写成 g(n)=O(f(n))**

   因为两者并不等价。实际上，这里的等号并不是通常相等的含义。

运算规则

1. $O(f)+O(g)=O(\max(f, g))$
2. $O(f)+O(g)=O(f+g)$
3. $O(f)\cdot O(g)=O(f\cdot g)$
4. 如果 $g(n)=O(f(n))$,则 $O(f)+O(g)=O(f)$
5. $f=O(f)$
6. $O(cf(n))=O(f(n))$

证明: O(*f*)+O(g)=O(max(*f*, g))

设$F(N)=O(f)$.根据O的定义,存在正常数C_1和自然数N_1,使得对所有$N\geq N_1$,有$F(N)\leq C_1 f(N)$.类似$G(N)=O(g)$.根据O的定义,存在正常数C_2和自然数N_2,使得对所有$N\geq N_2$,有$G(N)\leq C_2 g(N)$.

令$C_3=max\{C_1, C_2\}$, $N_3=max\{N_1, N_2\}$, $h(N)=max\{f, g\}$,则对所有$N\geq N_3$, 有

$$F(N)\leq C_1 f(N) \leq C_1 h(N) \leq C_3 h(N).$$

类似 $$G(N)\leq C_2 g(N) \leq C_2 h(N) \leq C_3 h(N).$$

$$O(f)+O(g)= F(N) + G(N)$$

$$\leq C_3 h(N)+C_3 h(N)=2C_3 h(N)= O(h)=O(\max(f, g))$$

例 估计如下二重循环的$T_{max}(n)$的阶.

```
for i:= 1 to n do
    for j:=1 to i do
     { s1, s2, s3, s4 } ;
```

s1, s2, s3, s4为单一赋值语句

分析:内循环体只需O(1)时间,故

2. $O(f)+O(g)=O(f+g)$

内循环共需 $\sum_{j=1}^{i} O(1) = O(\sum_{j=1}^{i} 1) = O(i)$

外循环共需 $\sum_{i=1}^{n} O(i) = O(\sum_{i=1}^{n} i) = O(\frac{n(n+1)}{2}) = O(n^2)$

(2) 大Ω表示法（算法运行时间的下限）

若∃正常数c和自然数N_0使得当 $n \geq N_0$ 时,有$f(n) \geq c\ g(n)$ 则称函数$f(n)$在n充分大时有下限, 且 $g(n)$是它的一个下限,记为$f(n) = \Omega(g(n))$ 也称$f(n)$的阶不低于 $g(n)$的阶

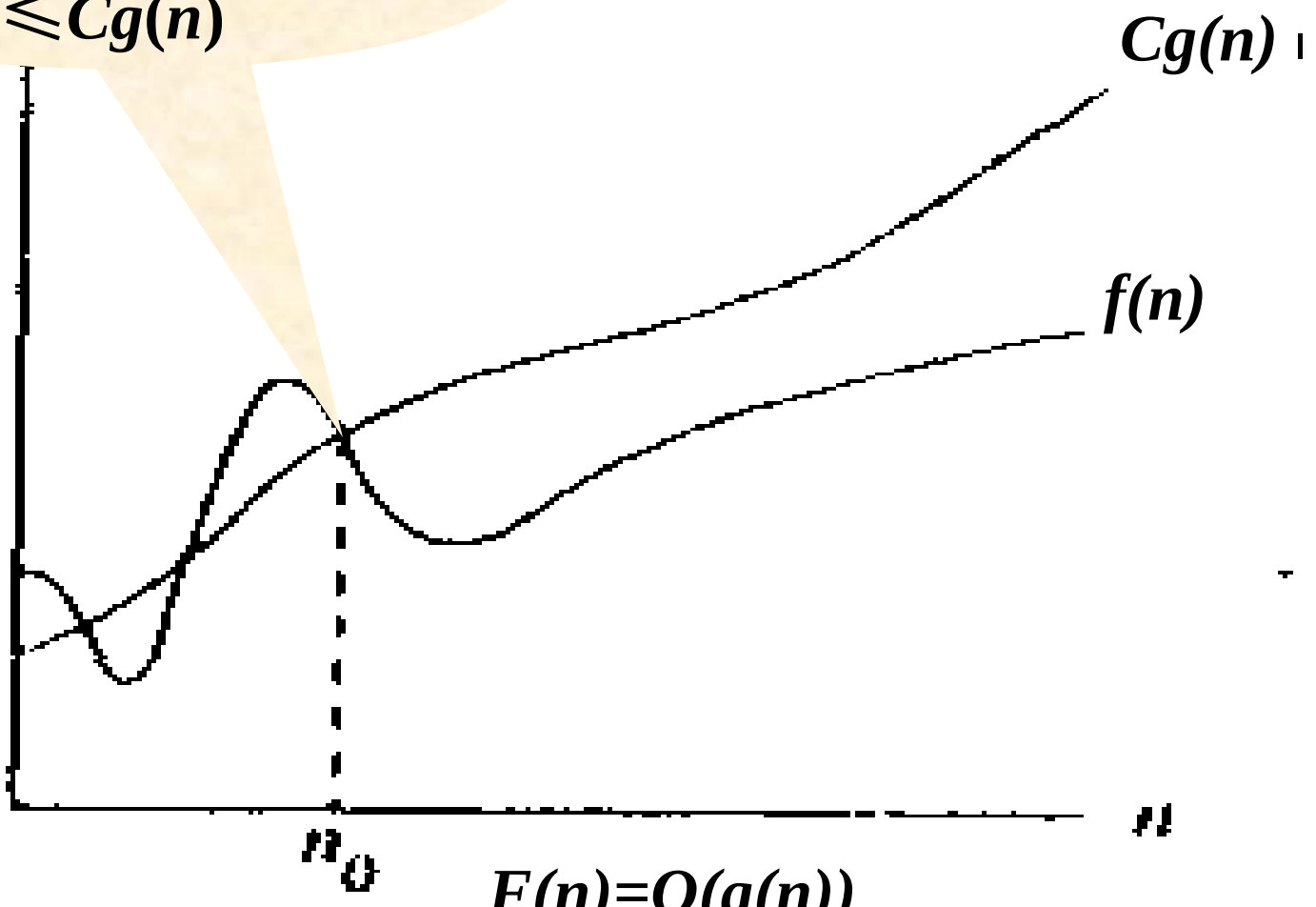
例 $T(n)=c_1n^2+c_2n$, **则** $T(n)= \Omega(n^2)$,

(3) θ表示法

$f(n)=\theta(g(n))$等价于 $f(n)=O(g(n))$ 且 $f(n)=\Omega(g(n))$称函数f(n)与$g(n)$ 同阶.

例 $T(n)=c_1n^2+c_2n$ ， 则 $T(n)=\theta(n^2)$

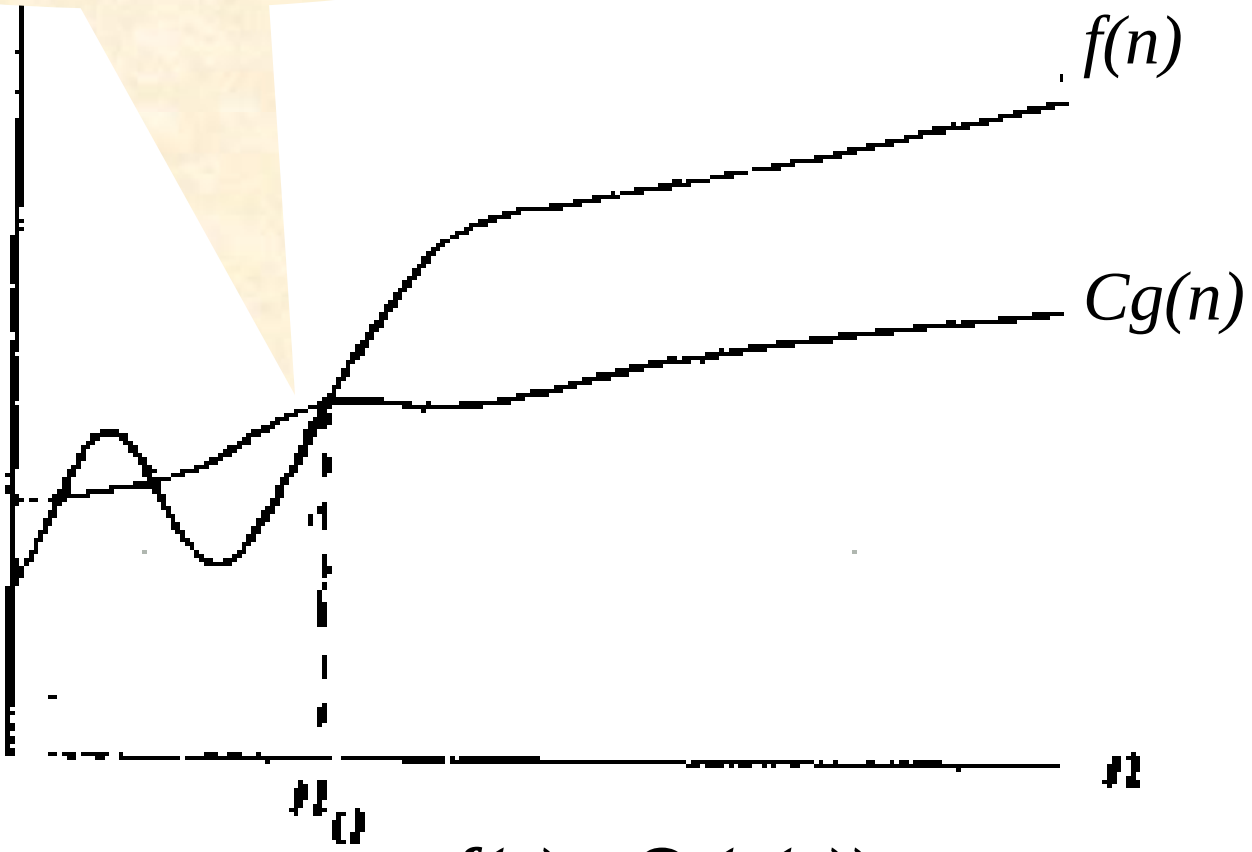
对于所有$n \geqslant n_0$，
$f(n) \leqslant Cg(n)$



$F(n)=O(g(n))$

对于所有$n \geqslant n_0$,

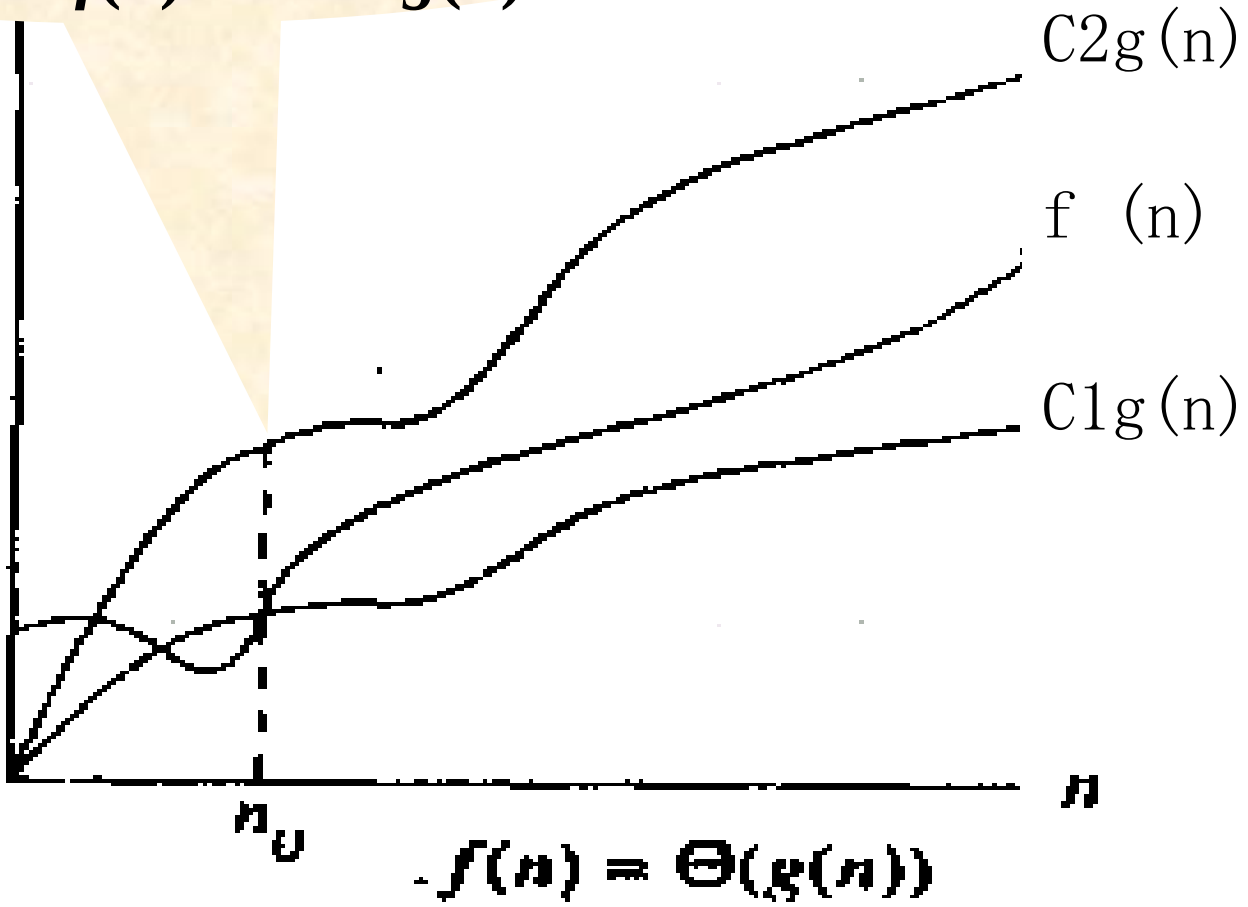
$f(n) \geqslant Cg(n)$



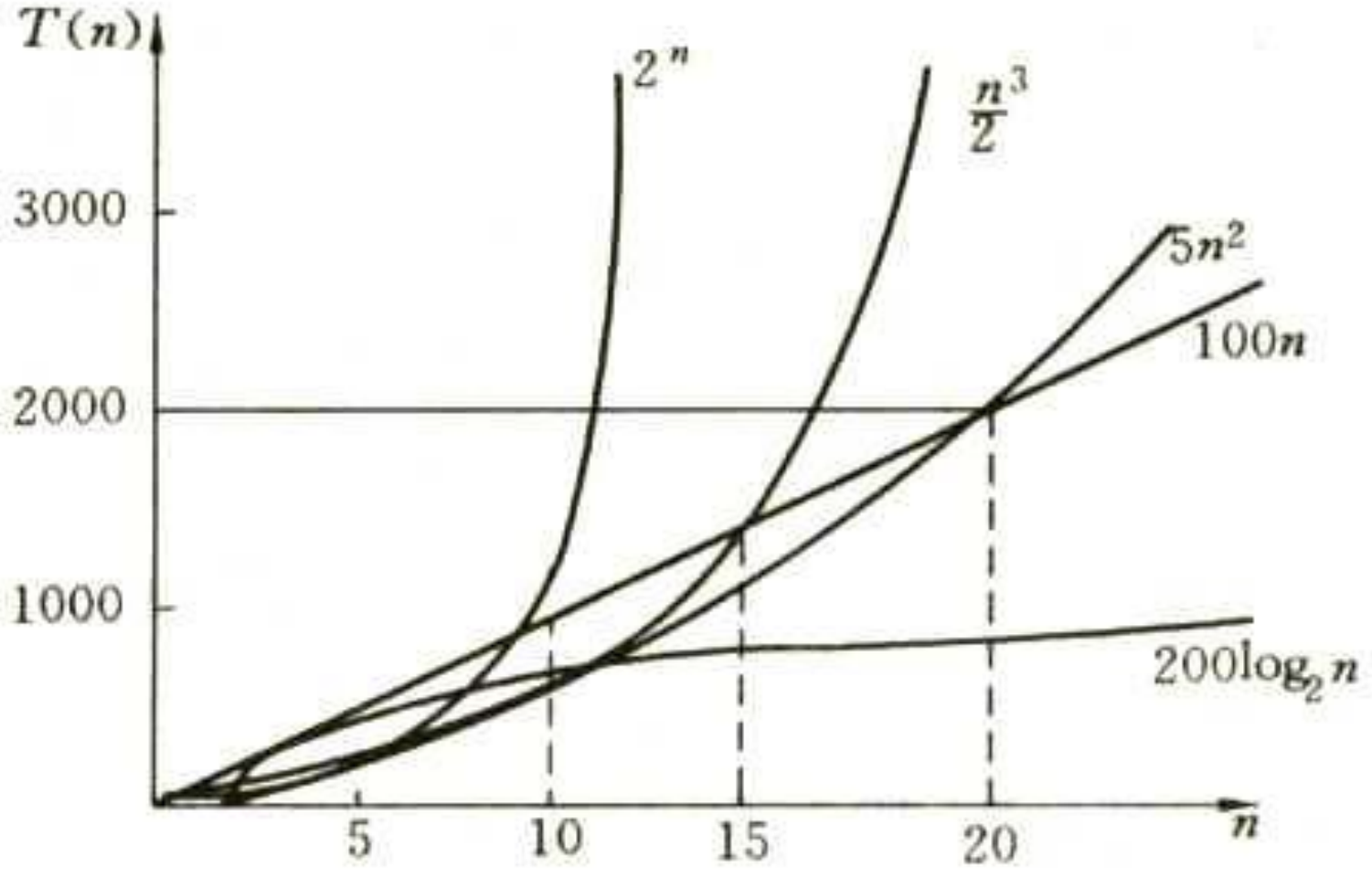
$f(n) = \Omega(g(n))$

对于所有$n \geqslant n_0$,

$C2g(n) \geqslant f(n) \geqslant C1g(n)$



不同时间函数的增长率



不同时间复杂性函数的对比

T(n)微秒	logn	n	nlogn	n^2	n^3	n^5	2^n	3^n	n!
n=10	3.3	10	33	100	1 毫秒	0.1 秒	1 毫秒	59 毫秒	3.6 秒
n=40	5.3	40	213	1600	64 毫秒	1.7 分	12.7 天	3855 世纪	10^3 世纪
n=60	5.9	60	354	3600	216 毫秒	1.3 分	366 世纪	1.3×10^{13} 世纪	10^{66} 世纪

可见，不同T(n)的算法当n增长时运算时间增长的快慢很不同。T(n)为指数形式的算法当n较大时实际上是无法应用的。有些算法T(n)与n！成正比，它随n的加大比指数函数增长还要快，这种算法更是没有实用价值。凡是T(n)为n的对数函数、线性函数或多项式的(幂函数也是多项式的特例)，称为“有效算法”。

算法复杂性分类:

多项式阶算法(有效算法):若算法的最坏情形时间复杂度$T(n)=O(n^k)$;

指数阶算法:若算法的最坏情形时间复杂度$T(n)=\Omega(a^n)$, $a>1$. 这类算法可认为计算上不可行的算法.

常见的多项式阶有:

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

常见的指数阶有: $O(2^n) < O(n!) < O(n^n)$

一般当n很大时,在计算机上运行比O(logn)复杂性更高的算法已经很困难了.

最优算法: 时间复杂性达到其下界的算法.

提高计算速度对不同时间复杂性函数的影响对比

时间复杂性	用现在的计算机	用快 100 倍的计算机	用快 1000 倍的计算机
n	N_1	$100N_1$	$1000N_1$
n^2	N_2	$10N_2$	$3.16N_2$
n^3	N_3	$4.64N_3$	$10N_3$
n^5	N_4	$2.5N_4$	$3.98N_4$
2^n	N_5	$N_5+6.64$	$N_5+9.97$
3^n	N_6	$N_6+4.19$	$N_6+6.29$

两点说明:

- **对规模较小的问题,决定算法工作效率的可能是算法的简单性而不是算法执行的时间.**
- **当比较两个算法的效率时,若两个算法是同阶的,必须进一步考察阶的常数因子才能辨别优劣.**

时间复杂度并不是表示一个程序解决问题需要花多少时间，而是当问题规模扩大后，程序需要的时间长度增长得有多快。