第一章 算法复杂性分析初步

柳银萍

ypliu@cs.ecnu.edu.cn

数统楼127室

算法与程序

算法：是满足下述性质的指令序列。

- 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。
- 确定性：组成算法的每条指令清晰、无歧义。
- 有限性：算法中每条指令的执行次数有限，执行每条指令的时间也有限。
- 可行性：算法描述的操作可通过已经实现的基本操作执行有限次来实现。

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(4)即有限性。

一个好的算法除满足以上五个重要特性，还需具备以下特性：

- 正确性：对任何合法的输入，都会得出正确的结果；
- 健壮性：对错误的输入，算法应能识别并作出处 理，而不是产生错误动作或陷入瘫痪；
- 可理解性：算法容易理解和实现；
- 抽象分级：算法步骤太多的话，抽象分级处理；
- 高效性：算法的时间效率和空间效率。

复杂性分析初步

程序性能 (program performance) 是指运行一个程序所需的内存大小和时间多少。所以，程序的性能一般指程序的空间复杂性和时间复杂性。性能评估主要包含两方面, 即性能分析 (performance analysis)与性能测量 (performance measurement), 前者采用分析的方法，后者采用试验的方法。

考虑空间复杂性的理由

✓想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需求解决方案,从中择取;

✓用空间复杂性来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。

选取方案的规则：如果对于解决一个问题有多种可选的方案，那么方案的选取要基于这些方案之间的性能差异。对于各种方案的时间及空间的复杂性，最好采取加权的方式进行评价。但是随着计算机技术的迅速发展，对空间的要求往往不如对时间的要求那样强烈。因此我们这里的分析主要强调时间复杂性的分析。

§1 空间复杂性

程序所需要的空间:

- **指令空间**---用来存储经过编译之后的程序指令。程序所需的指令空间的大小取决于如下因素:
  - ✓ 把程序编译成机器代码的编译器;
  - ✓ 编译时实际采用的编译器选项;
  - ✓ 目标计算机。
- **数据空间**---用来存储所有常量和变量的值。分成两部分:

  a.) 存储常量和简单变量;

  b.) 存储复合变量。

计算方法：结构变量所占空间等于各个成员所占空间的累加；数组变量所占空间等于数组大小乘以单个数组元素所占的空间。 例如：

double a[100]; 所需空间为： **100 × 8=800**

int matrix[r][c]; 所需空间为 **2×r×c**

- 环境栈空间---保存函数调用返回时恢复运行所需要的信息。当一个函数被调用时，下面数据将被保存在环境栈中：
  - ✓ 返回地址；
  - ✓所有局部变量的值、递归函数的传值形式参数的值；
  - ✓所有引用参数以及常量引用参数的定义。

在分析空间复杂性中，实例特征的概念特别重要。所谓实例特征是指决定问题规模的那些因素。如，输入和输出的数量或相关数的大小，如对 n 个元素进行排序、n×n矩阵的加法等。都可以 n 作为实例特征， 两个 m×n 矩阵的加法应该以n 和 m 两个数作为实例特征。

一个程序所需要的空间可分为两部分：

- 固定部分，它独立于实例特征。主要包括指令空间、简单变量以及定义复合变量所占用的空间、常量所占用的空间。

- 可变部分，主要包括复合变量所需的空间（其大小依赖于所解决的具体问题）。动态分配的空间（依赖于实例特征）、递归栈所需的空间（依赖于实例特征）。

令 S(P)表示程序 P 所需的空间，则有

$$S(P) = c + S_p(\text{实例特征})$$

其中 c 表示固定部分所需要的空间，是一个常数；S_p 表示可变部分所需的空间。在分析程序的空间复杂性时，一般着重于估算S_p(实例特征)。

实例特征的选择一般受到相关数的数量以及程序输入和输出规模的制约。

注：一个精确的分析还应当包括编译期间所产生的临时变量所需的空间，这种空间与编译器直接相关联，在递归程序中除了依赖于递归函数外，还依赖于实例特征。但是，在考虑空间复杂性时，一般都被忽略。

例子：

例1 顺序搜索

```
template <class T>
int SequentialSearch(T a[],const T &x, int n)
{ // 在 a[0:n-1]中搜索x，若找到则回所在的位置，否则返回-1
  int i;
  for(i=0; i<n && a[i]!=x; i++);
  if (i==n) return –1;
   return i;
}
```

在例1中，假定采用被查数组的长度 n 作为实例特征，并取T为 int 类型。数组中每个元素需要 2 个字节，实参x 需要2 个字节，传值形式参数 n 需要2 个字节，局部变量 i 需要2 个字节，整数常量 –1 需要2 个字节，所需要的总的数据空间为 10 个字节，其独立于 n, 所以 $S_{\text{顺序查找}}(n)=0$. 这里我们并没有把数组a 所需的空间计算进来，因为该数组所需的空间已在定义实际参数(对应于 a) 的函数中分配，不能再加到函数 SequentialSearch 所需的空间上去。

§2 时间复杂性

- **时间复杂性的构成**

✓ 一个程序所占时间 T(P)=编译时间+运行时间；

✓ 编译时间与实例特征无关，而且，一个编译过的程序可以运行若干次，人们主要关注的是运行时间，记作 t_p(实例特征）；

✓ 如果了解所用编译器的特征，就可以确定代码 P 进行加、减、乘、除、比较、读、写等所需的时间，从而得到计算 t_p 的公式。令 n 代表实例特征（这里主要是问题的规模），则有如下的计算公式：

$$t_p(n)=c_aADD(n)+c_sSUB(n)+c_mMUL(n)+c_dDIV(n)+c_cCMP(n)+\ldots$$

其中，c_a, c_s, c_m, c_d, c_c 分别表示一次加、减、乘、除及比较操作所需的时间，函数 ADD, SUB, MUL, DIV, CMP 分别表示代码 P 中所使用的加、减、乘、除及比较操作的次数；

✓ 一个算术操作所需的时间取决于操作数的类型(int, float, double 等等），所以，有必要对操作按照数据类型进行分类；

✓ 在构思一个程序时，影响 t_p 的许多因素还是未知的，所以，在多数情况下仅仅是对 t_p 进行估计。

✓估算运行时间的方法：

A. 找一个或多个关键操作，确定这些关键操作所需要的执行时间（对于前面所列举的四种运算及比较操作，一般被看作是基本操作，并约定所用的时间都是一个单位）；

~~B. 确定程序总的执行步数。~~

- 操作计数

首先选择一种或多种操作（如加、乘和比较等），然后计算这些操作分别执行了多少次。关键操作对时间复杂性影响最大。

例2 寻找最大元素

```
template<class T>
int Max(T a[],int n)
{ //寻找 a[0:n-1]中的最大元素
  int pos=0;
  for(i=1; i<n; i++)
    if(a[pos]<a[i])
        pos=i;
  return pos;
}
```

这里关键操作是比较。For 循环中共进行了n-1 次比较。Max 还执行了其它比较，如 for 循环每次执行之前都要比较一下i和n.此外，Max 还进行了其它的操作，如初始化 pos、循环控制变量i的增量。这些一般都不包含在估算中，若纳入计数，则操作计数将增加一个常量。

例3 n 次多项式求值

```
template<class T>
T PolyEval(T coeff[], int n, const T& x)
{ // 计算 n 次多项式的值， coeff[0:n] 为多项式的系数
   T y=1, value=coeff[0];
   for(i=1;i<=n;i++)  //n 循环
   { // 累加下一项
      y*=x;  //一次乘法
      value+=y*coeff[i]; //一次加法和一次乘法
    }
   return value;
}      // 3n 次基本操作
```

例4

```
template<class T>
void Rank(T a[],int n,int r[])
{ //计算 a[0:n-1] 中元素的排名
 for(int i=1; i<n; i++)
  r[i]=0;   //初始化
 // 逐对比较所有的元素
 for(int i=1;i<n;i++)
    for (j=0;j<i;j++)
      if (a[j]<a[i]) r[i]++;
      else   r[j]++;
}
```

例5

```
template<class T>
void SelectionSort(T a[],int n)
{// 对数组 a[0:n-1] 中元素排序
 for(int size=n;size>1;size--)
  {  j=Max(a, size);
        Swap(a[j],a[size-1]);
   }
}
// 其中函数 Max 定义如例3
//函数Swap 由下述给出
template<class T>
inline void Swap(T &a, T &b)
{  T temp=a; a=b; b=temp; }
```

分析的种类

最坏情况: (usually)

$T(n)$ = maximum time of algorithm on any input of size n.

平均情况: (sometimes)

$T(n)$ = expected time of algorithm over all inputs of size n. Need assumption of statistical distribution of inputs.

最好情况: (bogus)

Cheat with a slow algorithm that works fast on some input.

§3 渐近符号

确定程序的操作计数和执行步数的目的是为了比较两个完成同一功能的程序的时间复杂性，预测程序的运行时间随着实例特征变化的变化量。设 $T(n)$ 是算法A 的复杂性函数。一般说来，当n单调增加趋于∞ 时，$T(n)$也将单调增加趋于∞ 。如果存在函数$\tilde{T}(n)$，使得当$n\to\infty$时有$(T(n)-\tilde{T}(n))/T(n)\to 0$，则称$\tilde{T}(n)$ 是$T(n)$当$n\to\infty$时的渐近复杂性。$\tilde{T}(n)$是$T(n)$中略去低阶项所留下的主项，所以它无疑比$T(n)$来得简单（举一个例子）。

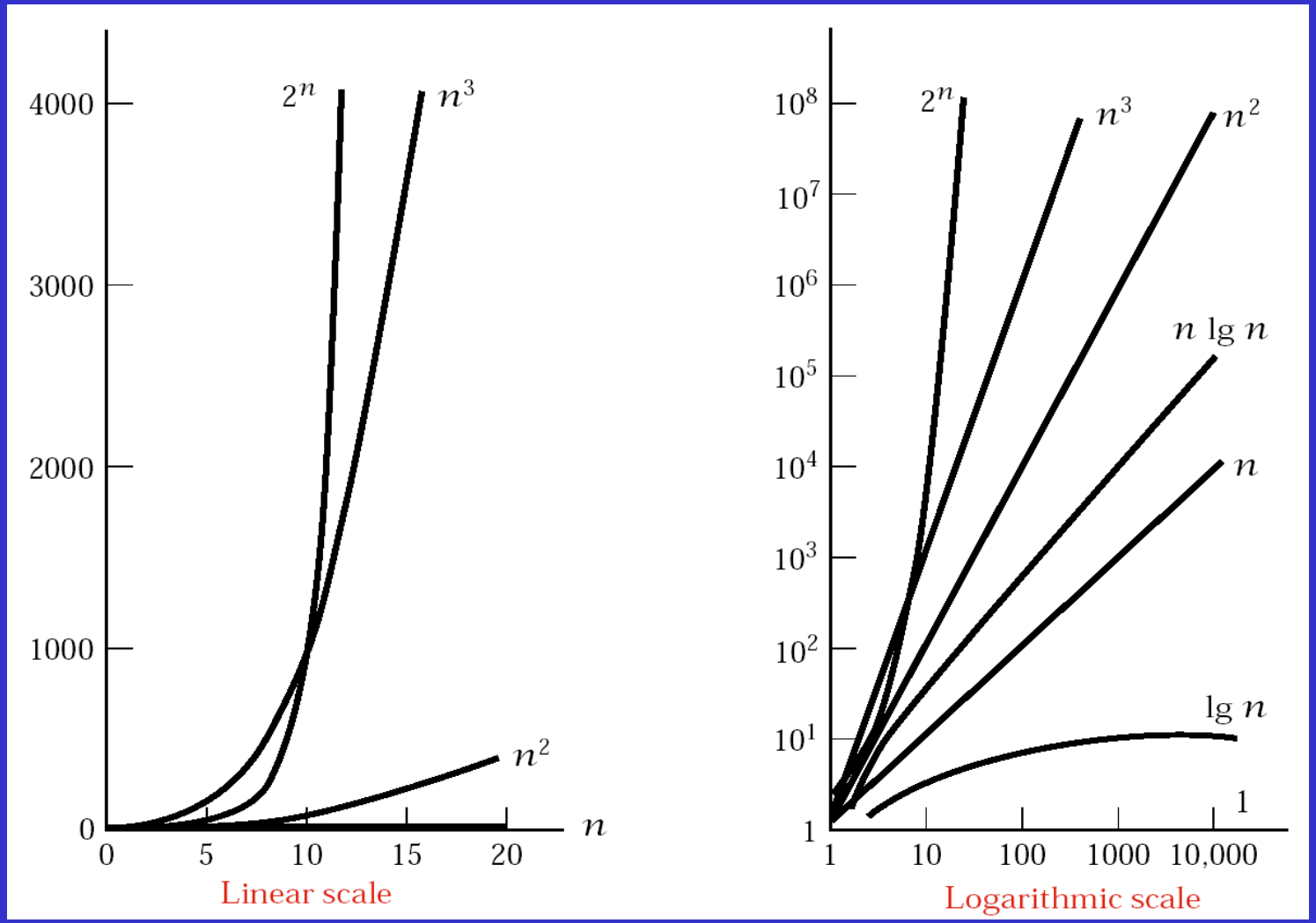
进一步分析可知，两个算法的渐近复杂性的阶不相同时，只要确定出各自的阶，就可以知道哪一个算法的效率高。换句话说，渐近复杂性分析只要关心$\tilde{T}(n)$的阶就够了，不必关心包含在$\tilde{T}(n)$中的常数因子。所以，可对$\tilde{T}(n)$的分析进一步简化，即假设算法中用到的所有不同的基本运算各执行一次所需要的时间都是一个单位时间。

综上分析，我们已经简化算法复杂性为其在渐近意义下的阶。为此引入渐近符号，首先给出常用的渐近函数。

表 1　常用的渐近函数

函数	名称	函数	名称
1	常数	n^2	平方
logn	对数	n^3	立方
n	线性	2^n	指数
nlogn	n倍logn	n!	阶乘

常用阶的增长率


4000
3000
2000
1000
0
2^n
n^3
n^2
n
0
5
10
15
20
Linear scale
10^8
10^7
10^6
10^5
10^4
10^3
10^2
10^1
1
2^n
n^3
n^2
$n \lg n$
n
$\lg n$
1
1
10
100
1000
10,000
Logarithmic scale

常用阶的比较

n	1	$\lg n$	n	$n \lg n$	n^2	n^3	2^n
1	1	0.00	1	0	1	1	2
10	1	3.32	10	33	100	1000	1024
100	1	6.64	100	664	10,000	1,000,000	1.27×10^{30}
1000	1	9.97	1000	9970	1,000,000	10^9	1.07×10^{301}

用f(n)表示一个程序的时间或空间复杂性，它是实例特征n的函数。假定函数f(n)≥0，且可假定n≥0。

渐近符号O的定义：

f(n)=O(g(n))当且仅当存在正的常数c 和n_0，使得对于所有的n>=n_0，有f(n)<=cg(n)。此时，称g(n)是f(n)的一个上界。

$C_0=O(1)$

$3n+2=O(n)$: 取 $c=4$，$n_0=2$。

$100n+6=O(n)$: 取 $c=101$，$n_0=6$。

$10n^2+4n+3=O(n^2)$: 取 $c=11$, $n_0=6$

$6\times 2^n+n^2=O(2^n)$: 取 $c=7$，$n_0=4$.

$3\times \log n + 2\times n + n^2 =O(n^2)$.

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

$3n+2=O(n^2)$.

注意事项

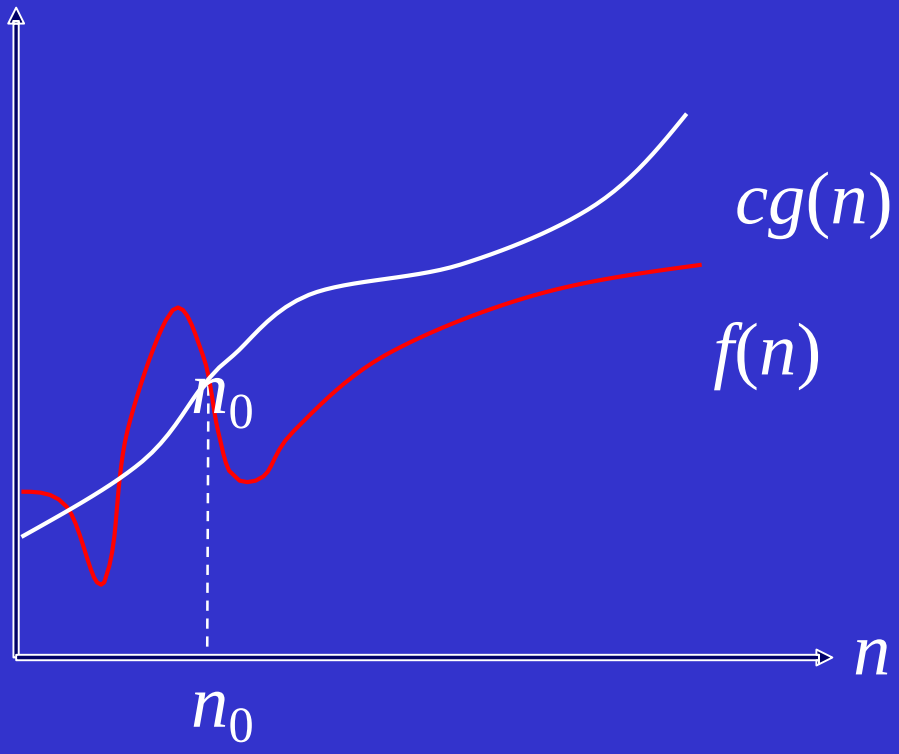
1. 用来作比较的函数g(n)应该尽量接近所考虑的函数f(n).

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

2. f(n)=O(g(n))不能写成g(n)=O(f(n))，因为两者并不等价。

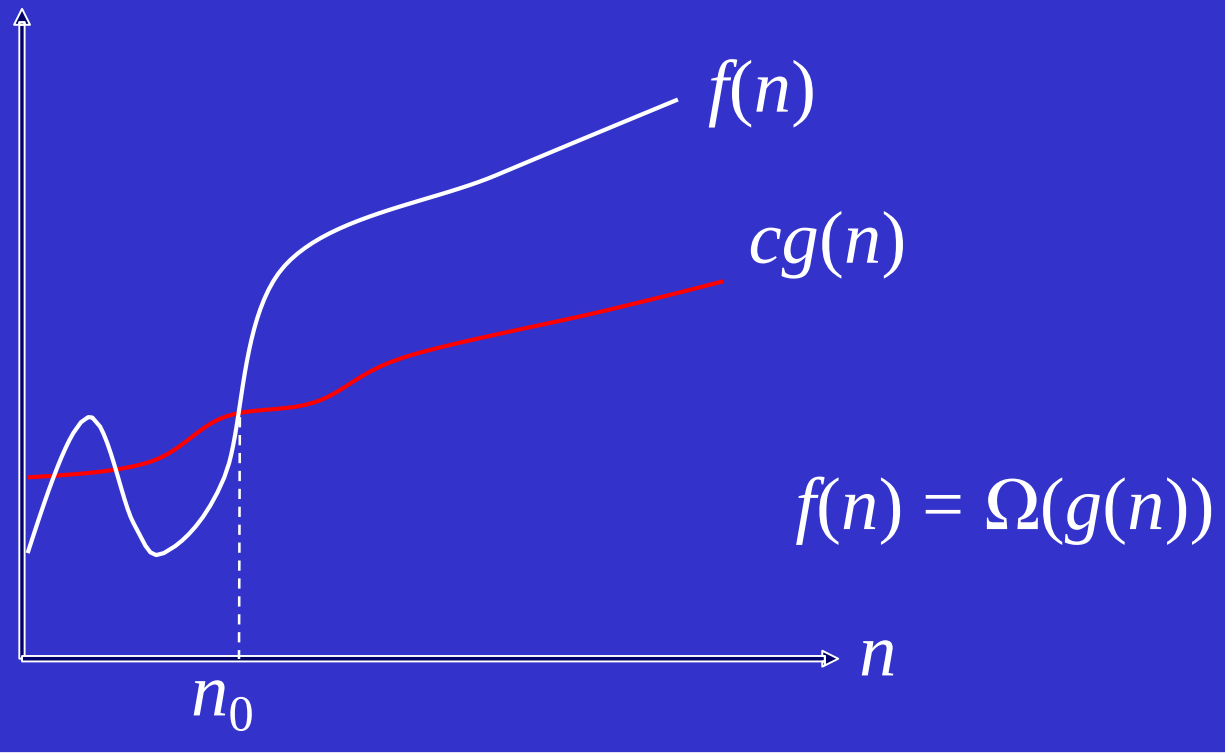
这里的等号并不是通常相等的含义。

$O(g(n)) = \{ f(n)$：存在正常数 c和 n_0，使得当$n \geq n_0$ 时，都有$0 \leq f(n) \leq cg(n)\}$

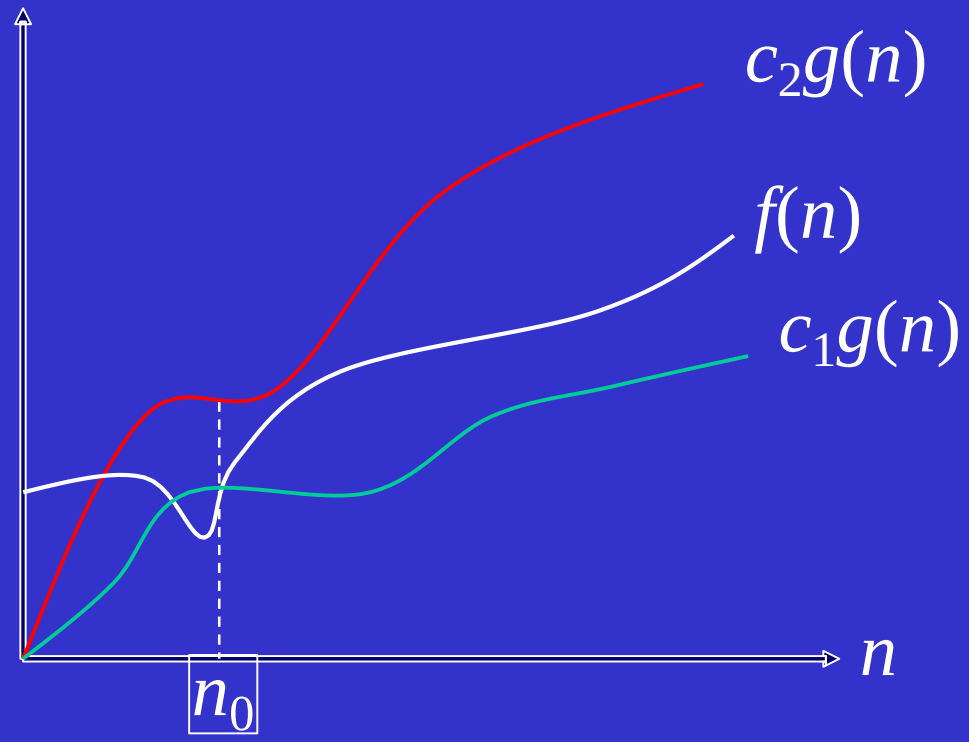


$f(n) = O(g(n))$

$\Omega(g(n)) = \{ f(n)$: 存在一个正整数c及 n_0 使得当$n \geq n_0$ 时，有 $0 \leq cg(n) \leq f(n)\}$



$\Theta(g(n)) = \{ f(n)$：存在正的常数$c_1, c_2,$ 和 n_0 使得当$n \geq n_0$ 时，都有$0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)\}$



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

定理 对于多项式函数 $a_m n^m + a_{m-1} n^{m-1} + \cdots + a_1 n + a_0$. 如果$a_m>0$, 则

$$f(n) = O(n^m), \quad f(n) = \Omega(n^m), \quad f(n) = \Theta(n^m).$$

小o 符号定义： **f(n)= o (g(n))** 当且仅当**f(n)= O(g(n))** 和**g(n)≠ O(f(n)).**

例7 折半搜索

```
Template <class T>
Int BinarySearch(T a[ ], const T & x, int n)
{// 在a[0]<=a[1]<=…<=a[n-1] 中搜索 x
  // 如果找到，则返回所在位置，否则返回 -1
  int left=0; int right=n-1;
   while(left<=right)  {
      int middle=(left+right)/2;
      if (x==a[middle]) return middle;
      if (x>a[middle]) left=middle+1;
      else   right=middle-1;
   }
Return -1 ;  // 未找到 x
}
```

该循环在最坏的情况下需要执行$\Theta(\log n)$次. 每次循环耗时$\Theta(1)$, 在最坏情况下，总的时间复杂性为$\Theta(\log n)$.