



第1章 基本概念

内容

1.1 解决问题的方法的效率和什么相关？

1.2 如何获取程序的执行时间

1.3 什么是数据结构

1.4 什么是抽象数据类型

1.5 算法复杂度

1.6 空间复杂度分析

1.7 时间复杂度分析

1.8 算法复杂度的表示

1.9 常用的复杂度函数

1.10应用：最大子列和问题算法分析

附：算法的概念

1.1 解决问题的方法的效率和什么相关？

□ 图书的摆放问题

□ 随便摆放

- 怎么插入新书
- 怎么找书



□ 按字母索引摆放

- 怎么插入新书
- 怎么找书

□ 分区分类摆放

- 怎么插入新书
- 怎么找书



1.1 解决问题的方法的效率和什么相关？（续）

- 写程序实现一个函数**PrintN**，使得传入一个正整数为**N**的参数后，能顺序打印从**1**到**N**的全部正整数

循环实现

```
void PrintN ( int N )
{ int i;
  for ( i=1; i<=N; i++ )
    printf("%d\n", i );
  return;
}
```

递归实现

```
void PrintN ( int N )
{ if ( !N ) return;
  PrintN( N - 1 );
  printf("%d\n", N );
  return;
}
```

令N=100,1000,10000,100000,...

```
#include <stdio.h> void
PrintN ( int N );
int main () {
  int N;
  scanf("%d", &N);
  PrintN( N );
  return 0;
}
```

解决问题
方法的效
率跟**空间**
的利用率
有关

1.1 解决问题的方法的效率和什么相关？（续）

□ 写程序计算给定多项式在给定点 x 处的值

$$f(x) = a_0 + a_1x + a_2x^2 + \dots + a_{n-1}x^{n-1} + a_nx^n$$

法1——直接法

```
double f( int n, double a[], double x )
{
    int i;
    double p = a[0];
    for (i=1; i<=n; i++)
        p += (a[i]*pow(x,i));
    return p;
}
```

法2——秦九韶法

```
double f( int n, double a[], double x )
{
    int i;
    double p = a[n];
    for (i=n; i>0; i--)
        p = a[i-1] + x*p;
    return p;
}
```

解决问题
方法的效
率跟算法
的巧妙程
度有关

1.2如何得到程序的执行时间

- ❑ **clock()**：捕捉从程序开始运行到**clock()**被调用时所耗费的时间。
 - 这个时间单位是**clock tick**，即“时钟打点”
 - 常数**CLK_TCK**(或**CLOCKS_PER_SEC**)：机器时钟每秒所走的时钟打点数。

```
#include <stdio.h>
#include <time.h>
clock_t  start, stop; /* clock_t是clock()函数返回的变量类型 */
double  duration; /* 记录被测函数运行时间，以秒为单位 */
int main ()
{ /* 不在测试范围内的准备工作写在clock()调用之前*/
    start = clock(); /* 开始计时 */
    MyFunction(); /* 把被测函数加在这里 */
    stop = clock(); /* 停止计时 */
    duration = ((double)(stop - start))/CLK_TCK; /* 计算运行时间 */
    /* 其他不在测试范围的处理写在后面，例如输出duration的值 */
    return 0;
}
```

1.2如何得到程序的执行时间(续)

- 让被测函数重复运行充分多次，使得测出的总的时钟打点间隔充分长，最后计算被测函数平均每次运行的时间即可！

```
#include <stdio.h>
#include <time.h>
#include <math.h>

.....
#define MAXK 1e7 /* 被测函数最大重复调用次数*/
.....
int main ()
{
    .....
    start = clock();
    for ( i=0; i<MAXK; i++ ) /* 重复调用函数以获得充分多的时钟打点数*/
        f1(MAXN-1, a, 1.1);
    stop = clock();
    duration = ((double)(stop - start))/CLK_TCK/MAXK; /* 计算函数单次运行的时间*/
    printf("ticks1 = %f\n", (double)(stop - start));
    printf("duration1 = %6.2e\n", duration);
    .....
    return 0;
}
```

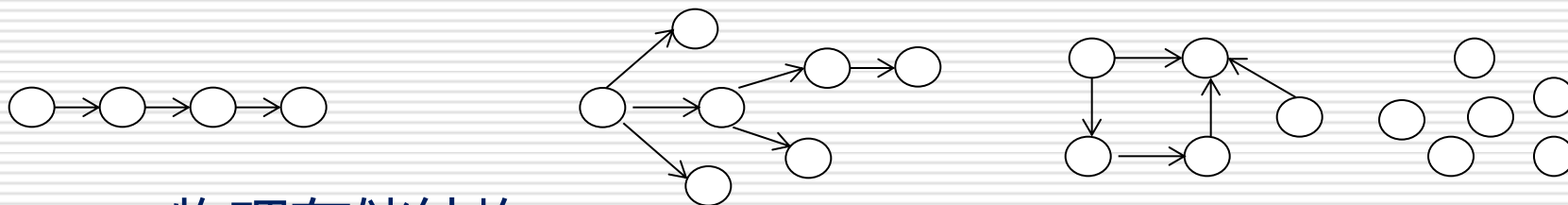
1.3什么是数据结构

□ **数据结构**是数据对象在计算机中的组织方式,形式定义为一个**二元组** $\text{Data_Structure}=(D,S)$, D 是数据元素的有限集即数据对象, S 是 D 上关系的有限集。

■ 逻辑结构

□ 数据对象的逻辑组织关系。分为线性、树、图、集合。

□ 如图书的随便摆放就是线性结构，分区摆放就是树形结构



■ 物理存储结构

□ 数据对象在计算机内存中的存储组织关系——逻辑结构在存储器中的映象。

■ 数据元素的映象：数据在内存中以**二进制**存储

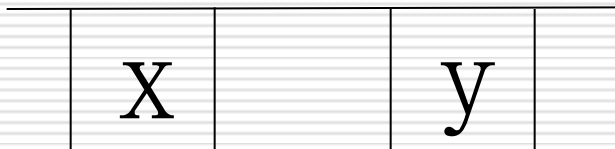
■ 关系的映象：**有序对** $\langle x,y \rangle$,所有关系都可表示为有序对的集合

□ 分为顺序存储和链式存储

1.3什么是数据结构（续）

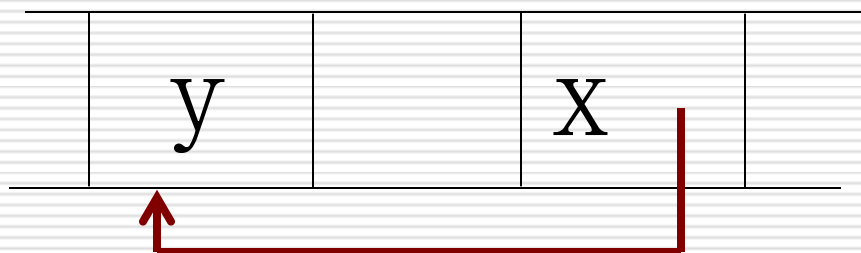
□ 有序对 $\langle x, y \rangle$ 的**顺序存储**：以**相对的存储位置**表示后继关系

- 例如：令 y 的存储位置和 x 的存储位置之间差一个常量 C ，而 C 是一个隐含值，**整个存储结构中只含数据元素本身的信息**



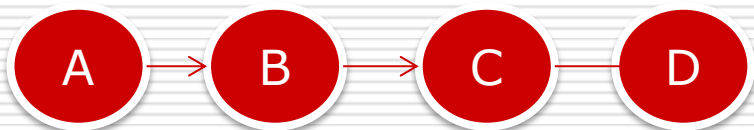
□ 有序对 $\langle x, y \rangle$ 的**链式存储**：以**附加信息(指针)**表示后继关系

- 需要用**一个和 x 在一起的附加信息**指示 y 的存储位置



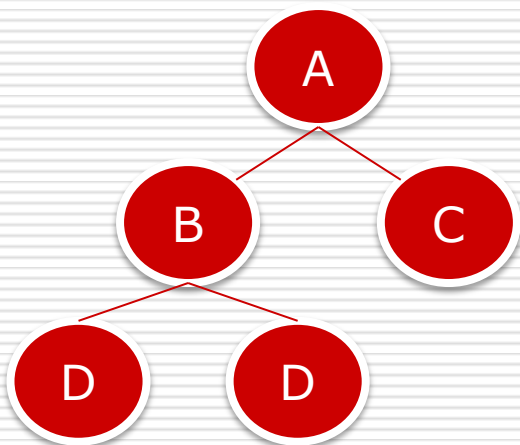
1.3什么是数据结构（续）

□ 用有序对表示线型结构



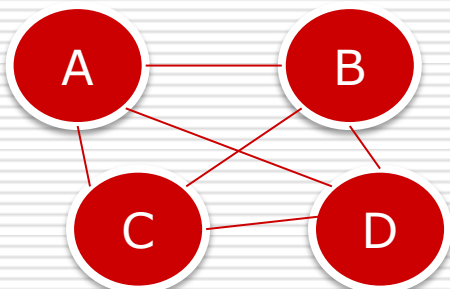
$\{ \langle A, B \rangle, \langle B, C \rangle, \langle C, D \rangle \}$

□ 用有序对表示树型结构



$\{ \langle A, B \rangle, \langle A, C \rangle, \langle B, D \rangle, \langle B, E \rangle \}$

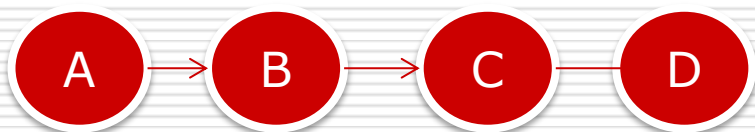
□ 用有序对表示图型结构



$\{ \langle A, B \rangle, \langle A, C \rangle, \langle A, D \rangle, \langle B, C \rangle, \langle B, D \rangle, \langle C, D \rangle \}$

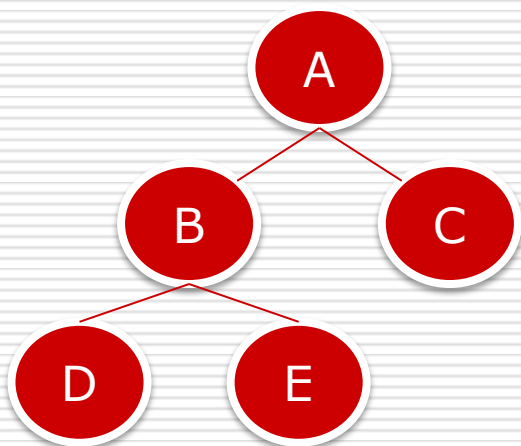
1.3什么是数据结构（续）

□ 用数组存储线型结构

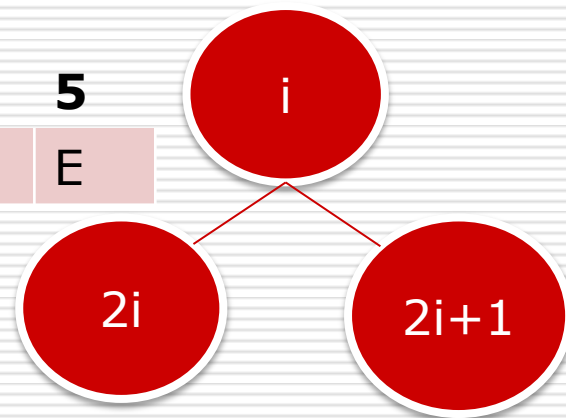


0	1	2	3
A	B	C	D

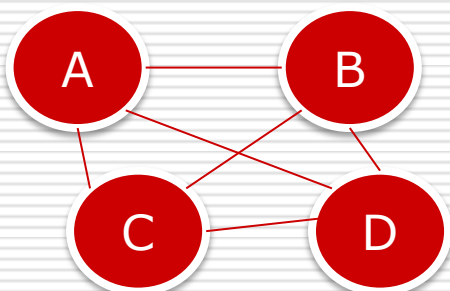
□ 用数组存储树型结构



0	1	2	3	4	5
	A	B	C	D	E



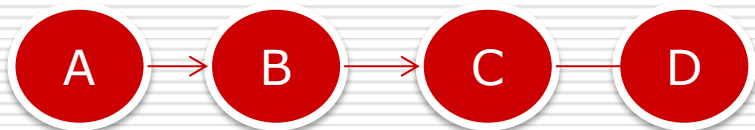
□ 用数组存储图型结构



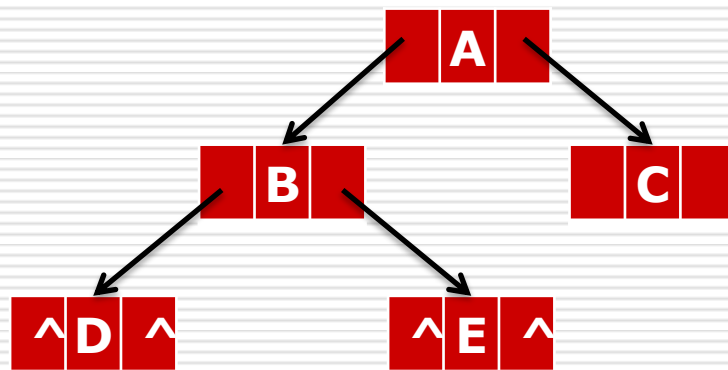
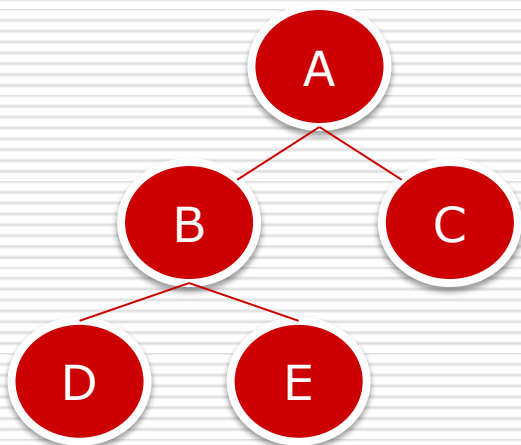
	0	1	2	3
0		1	1	1
1	1		1	1
2	1	1		1
3	1	1	1	

1.3什么是数据结构（续）

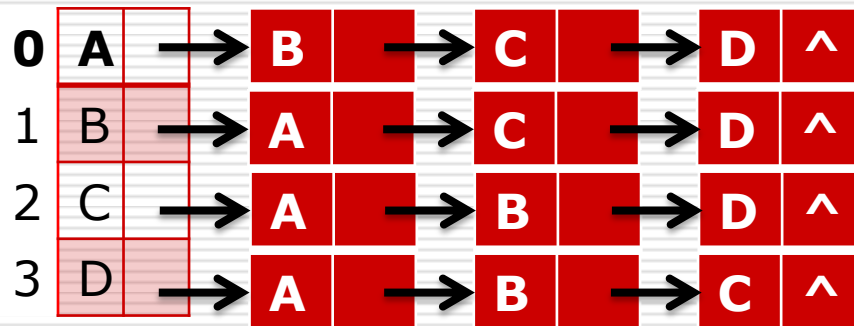
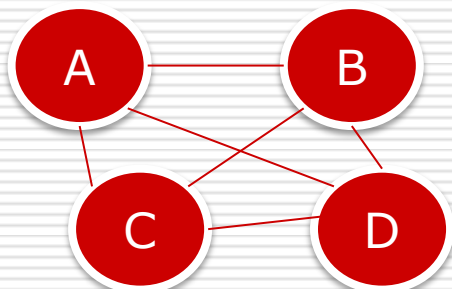
□ 用单链表存储线型结构



□ 用二叉链存储树型结构



□ 用邻接表存储图型结构



1.3什么是数据结构(续)


- 数据对象：实际问题中计算机要处理的事务。
 - 所有能被**输入**到计算机中，且能被计算机**处理的符号**的集合。
 - 是**计算机操作的对象**的总称。
 - 是计算机处理的**信息的某种特定的符号表示形式**。
 - 如图书摆放问题中的图书就是数据对象
- 数据元素
 - 是数据（集合）中的一个“**个体**”
 - 是数据结构中讨论的**基本单位**
- 数据项
 - 是数据结构中讨论的**最小单位**
 - **数据元素可以是数据项的集合**
 - 如：描述一个运动员的数据元素可以是

1.3什么是数据结构(续)

□ 数据项

- 是数据结构中讨论的**最小单位**
- **数据元素可以是数据项的集合**
- 如：描述一个运动员的数据元素可以是

姓名	俱乐部名称	出生日期	参加日期	职务	业绩
		年 月 日			



称之为组合项

1.3什么是数据结构(续)

□ 数据结构含义：带**结构**的数据元素的集合

例如：

3214,6587,9345 — **a1**(3214),**a2**(6587),**a3**(9345)

则在数据元素 a1、a2 和 a3 之间存在着 “**次序**” **关系** **<a1,a2>**、**<a2,a3>**

3214 , 6587 , 9345 **≠** 6587 , 3214 , 9345
a1 a2 a3 a2 a1 a3

1.3什么是数据结构(续)

□ 数据结构含义：或者说，数据结构是相互之间存在着某种逻辑关系的数据元素的集合。

又如：在2行3列的二维数组{a1, a2, a3, a4, a5, a6}

中六个元素之间存在两个关系：

a1	a2	a3
a4	a5	a6

row = {<a1,a2>, <a2,a3>, <a4,a5>, <a5,a6>}

col = {<a1,a4>, <a2,a5>, <a3,a6>}

a1 a3 a5

a1 a2 a3

a2 a4 a6

a4 a5 a6

≠

再如：在一维数组 {a1, a2, a3, a4, a5, a6} 的数据元

素之间存在如下的次序关系：{<a_i, a_{i+1}> | i=1, 2, 3, 4, 5}

可见，不同的“关系”构成不同的“结构”

1.4抽象数据类型

- 数据类型：数据对象的类型确定了其操作集和数据定义域
- 抽象数据类型：是指一个数学模型以及定义在此数学模型上的一组操作（D, S, P），突出抽象和封装，强调本质特征。数据对象+关系集+操作集
 - “抽象”的意思，是指我们描述数据类型的方法是不依赖于具体实现的
 - 即数据对象集和操作集的描述与存放数据的机器无关、与数据存储的物理结构无关、与实现操作的算法和编程语言均无关。
 - 简而言之，抽象数据类型只描述数据对象集和相关操作集“是什么”，并不涉及“如何做到”的问题。

1.4抽象数据类型（续）

ADT 抽象数据类型名 {

数据对象D：〈数据对象的定义〉

数据关系S：〈数据关系的定义〉

基本操作P：〈基本操作的定义〉

} ADT 抽象数据类型名

其中基本操作的定义格式为：

基本操作名（参数表）

初始条件：〈初始条件描述〉

操作结果：〈操作结果描述〉

1.4抽象数据类型（续）

赋值参数 只为操作提供输入值。

引用参数 以**&**打头，除可提供输入值外，还将返回操作结果。

初始条件 描述了操作执行之前数据结构和参数应满足的条件，若不满足，则操作失败，并返回相应出错信息。

操作结果 说明了操作正常完成之后，数据结构的变化状况和应返回的结果。若初始条件为空，则省略之。

1.4抽象数据类型（续）——矩阵的ADT举例

ADT Matrix{

数据对象： $D = \{a_{ij} \mid i=1, \dots, M; j=1, \dots, N\}$

数据关系：

$Rrow = \{ \langle a_{ij}, a_{i(j+1)} \rangle \mid i=1, \dots, M; j=1, \dots, N-1 \}$

$Rcol = \{ \langle a_{ij}, a_{(i+1)j} \rangle \mid i=1, \dots, M-1; j=1, \dots, N \}$

操作集：

Matrix Create(int M, int N)：返回一个**M×N**的空矩阵；

int GetMaxRow(Matrix A)：返回矩阵**A**的总行数；

int GetMaxCol(Matrix A)：返回矩阵**A**的总列数；

ElementType GetEntry(Matrix A, int i, int j)：返回**A_{ij}**

...

} ADT Matrix

1.4抽象数据类型（续）——复数的ADT举例

ADT Complex {

数据对象：

$$D = \{e1, e2 \mid e1, e2 \in \text{RealSet} \}$$

数据关系：

$$R1 = \{ \langle e1, e2 \rangle \mid \begin{array}{l} e1 \text{ 是复数的实数部分} \\ e2 \text{ 是复数的虚数部分} \end{array} \}$$

1.4抽象数据类型（续）——复数的ADT举例

基本操作：

AssignComplex(&Z, v1, v2)

操作结果：构造复数 Z,其实部和虚部分别被赋以参数 v1 和 v2 的值。

DestroyComplex(&Z)

操作结果：复数Z被销毁。

GetReal(Z, &realPart)

初始条件：复数已存在。

操作结果：用realPart返回复数Z的实部值。

1.4抽象数据类型（续）——复数的ADT举例

GetImag(Z, &ImagPart)

初始条件：复数已存在。

操作结果：用ImagPart返回复数Z的虚部值。

Add(z1,z2, &sum)

初始条件：z1, z2是复数。

操作结果：用sum返回两个复数z1, z2 的和值。

} ADT Complex

1.4抽象数据类型（续）——复数的ADT举例

假设: z_1 和 z_2 是上述定义的复数

则 $\text{Add}(z_1, z_2, z_3)$ 操作的结果

即为用户企求的结果

$$z_3 = z_1 + z_2$$

1.4抽象数据类型（续）——ADT的实现

抽象数据类型的表示和实现

抽象数据类型需要通过**固有数据类型**(高级编程语言中已实现的数据类型)来实现。

例如，对以上定义的复数。

1.4抽象数据类型（续）——复数的ADT实现

// -----存储结构的定义

```
typedef struct {  
    float realpart ;  
    float imagpart ;  
}complex ;
```

// -----基本操作的函数原型说明

```
void Assign( complex &Z,  
            float realval, float imagval ) ;  
// 构造复数 Z,其实部和虚部分别被赋以参数 //  
realval 和 imagval 的值
```

1.4抽象数据类型（续）——复数的ADT实现

```
float GetReal( cpmplex Z );  
// 返回复数 Z 的实部值
```

```
float Getimag( cpmplex Z );  
// 返回复数 Z 的虚部值
```

```
void add( complex z1, complex z2,  
          complex &sum );  
// 以 sum 返回两个复数 z1, z2 的和
```

1.4抽象数据类型（续）——复数的ADT实现

// -----基本操作的实现

```
void add( complex z1, complex z2,  
          complex &sum ) {  
    // 以 sum 返回两个复数 z1, z2 的和  
    sum.realpart = z1.realpart + z2.realpart;  
    sum.imagpart = z1.imagpart + z2.imagpart;  
}
```

{ 其它省略 }

1.5 算法复杂度定义

- 数据对象必定与一系列加在其上的操作相关联
 - 如查找、插入、删除、修改、求值、...
- 完成这些操作所用的方法就是算法
 - 通常一个算法用一个函数来实现。
 - 用空间复杂度和时间复杂度来评价算法
- 空间复杂度 $S(n)$ —— 根据算法写成的程序在执行时占用存储单元的长度。这个长度往往与输入数据的规模有关。空间复杂度过高的算法可能导致使用的内存超限，造成程序非正常中断
- 时间复杂度 $T(n)$ —— 根据算法写成的程序在执行时耗费时间的长度。这个长度往往也与输入数据的规模有关。时间复杂度过高的低效算法可能导致我们在有生之年都等不到运行结果

1.6空间复杂度分析

□ PrintN的递归算法的 $S(n)$ 太大： $S(n) = c \cdot n$

- n 是需要打印的整数的个数，是变量；
- c 是1个单位的内存空间占用存储单元的长度，为固定常数。

```
void PrintN ( int N )  
{  
    if ( !N ) return;  
    PrintN( N - 1 );  
    printf( "%d\n", N );  
    return;  
}
```

.....	100000	99999	99998	1
-------	--------	-------	-------	-------	---	-------

```
PrintN(100000)  
    PrintN(99999)  
        PrintN(99998)  
            PrintN(99997)  
                .....  
                    PrintN(0)
```

$$S(N) = C \cdot N$$

1.7时间复杂度分析

- 简单直接算法的 $T(n)$ 比较大： $T_2(n) = c_1n^2 + c_2n$ ，
 - 其中： n 是多项式的阶数，是变量；
 - c_1 是执行1/2次乘法需要的时间；
 - c_2 是执行1次加法和1/2次乘法需要的时间，都是固定常数。

```
double f( int n, double a[], double x )
{ int i;
  double p = a[0];
  for ( i=1; i<=n; i++ )           (1+2+.....+n)
    p += (a[i] * pow(x, i));       =(n2+n) / 2次乘法
  return p;
}
```

$T(n) = C_1n^2 + C_2n$

1.7时间复杂度分析（续）

- 秦九韶算法的 $T(n)$ 比较小： $T_1(n) = c \cdot n$
- 其中： n 是多项式的阶数，是变量；
 - c 是执行1次加法和乘法需要的时间，为固定常数。

```
double f( int n, double a[], double x )
{ int i;
  double p = a[n];
  for ( i=n; i>0; i-- )
    p = a[i-1] + x*p;
  return p;
}
```

n 次乘法！

$$T(n) = C \cdot n$$

1.8算法复杂度的渐进表示法

- 如何来“度量”一个算法的时间复杂度呢？
 - 首先，它应该与运行该算法的机器和编译器无关；
 - 其次，它应该与要解决的问题的规模 n 有关；（有时，描述一个问题的规模需要多个参数）
 - 再次，它应该与算法的“1步”执行需要的工作量无关！
 - 所以，在描述算法的时间性能时，人们只考虑宏观渐近性质，即当输入问题规模 n “充分大”时，观察算法复杂度随着 n 的“增长趋势”：当变量 n 不断增加时，解决问题所需要的时间的增长关系。
- 比如
 - 线性增长 $T(n) = c \cdot n$
 - 即问题规模 n 增长到2倍、3倍……时，解决问题所需要的时间 $T(n)$ 也是增长到2倍、3倍……（与 c 无关）
 - 平方增长： $T(n) = c \cdot n^2$
 - 即问题规模 n 增长到2倍、3倍……时，解决问题所需要的时间 $T(n)$ 增长到4倍、9倍……（与 c 无关）

1.8算法复杂度的渐进表示法（续）

□ [定义1.1 渐进上界]

- $T(n) = O(f(n))$ 表示存在常数 $c > 0, n_0 > 0$, 使得当 $n \geq n_0$ 时有 $T(n) \leq c f(n)$
 - 秦九韶算法的时间复杂度是 $O(n)$,
 - 而简单直接法的时间复杂度是 $O(n^2)$ 。

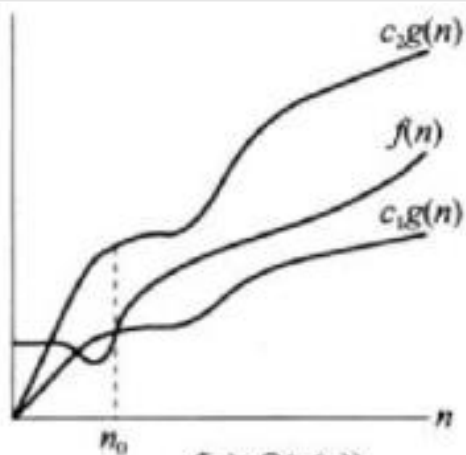
□ [定义1.2 渐进下界]

- $T(n) = \Omega(g(n))$ 表示存在常数 $c > 0, n_0 > 0$, 使得当 $n \geq n_0$ 时有 $T(n) \geq c g(n)$

□ [定义1.3 渐进确界]

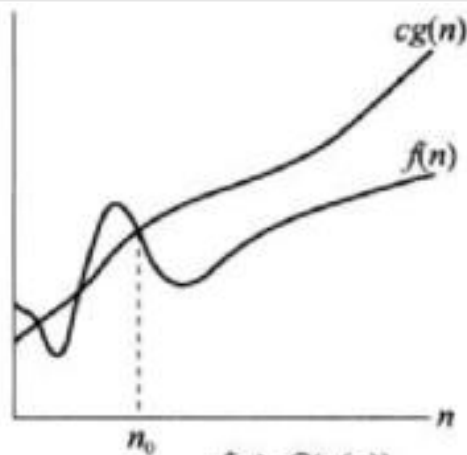
- $T(n) = \Theta(h(n))$ 表示
 $T(n) = O(h(n))$ 同时 $T(n) = \Omega(h(n))$

1.8 算法复杂度的渐进表示法 (续)



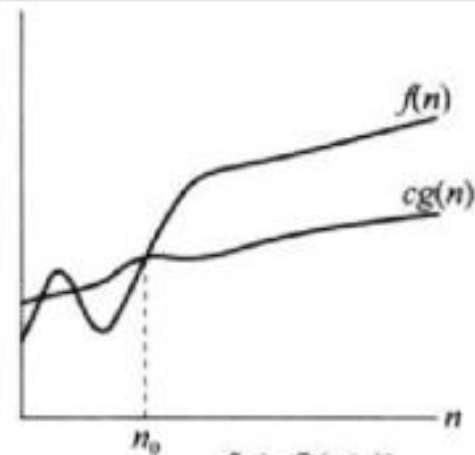
$$f(n) = \Theta(g(n))$$

(a)



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

(b)



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

(c)

1.8算法复杂度的渐进表示法（续）

□ 对给定的算法做渐进分析时，有几个小窍门：

(1) 若干层嵌套循环的时间复杂度等于各层循环次数的乘积再乘以循环体代码的复杂度。

例如下列2层嵌套循环的复杂度是 $O(N^2)$ ：

```
for ( i=0; i<N; i++ )  
    for ( j=0; j<N; j++ )  
        { x = y*x + z; k++; }
```

(2) if-else 结构的复杂度取决于if的条件判断复杂度和两个分枝部分的复杂度，总体复杂度取三者中最大。

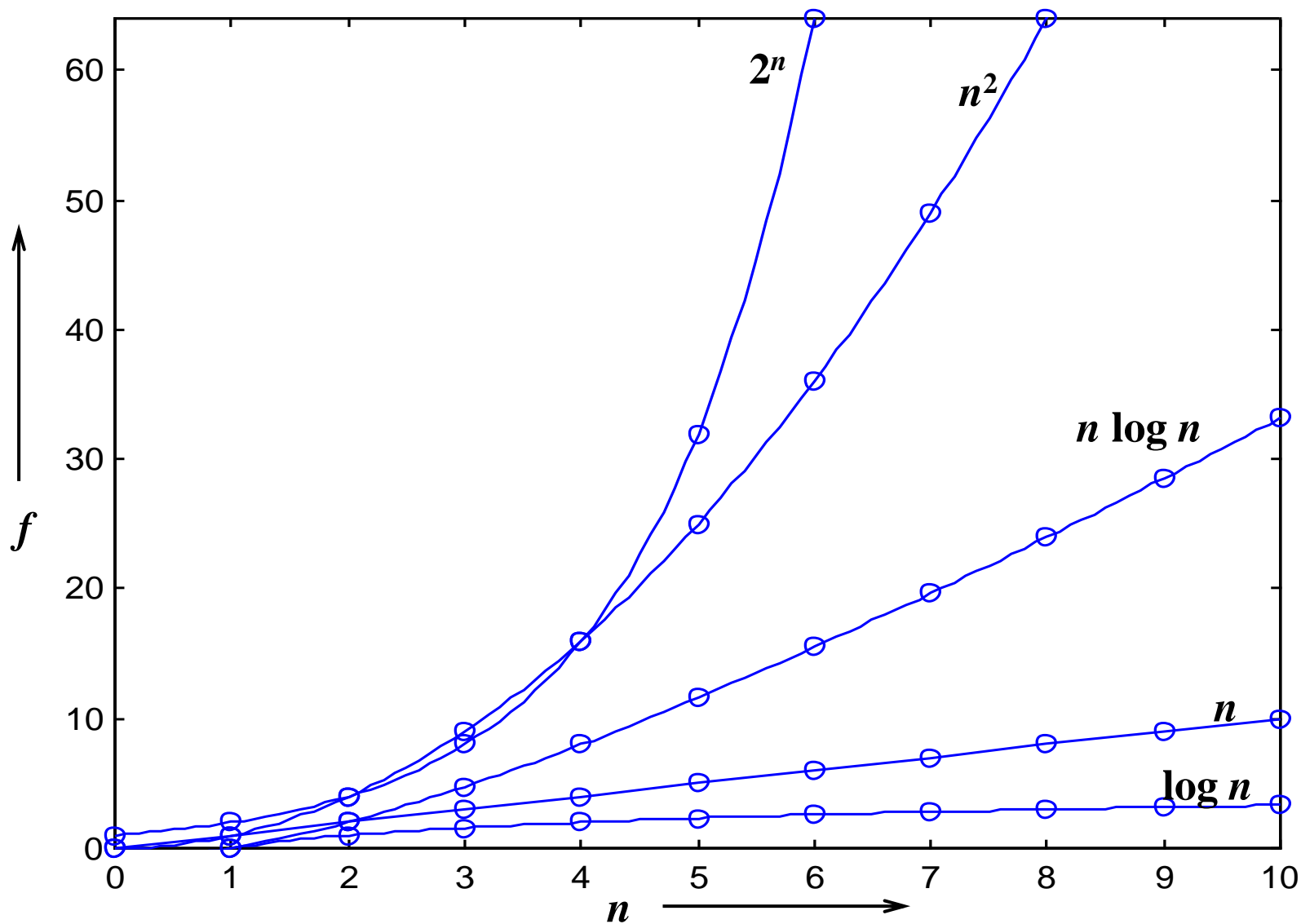
```
if (P1) /* P1的复杂度为  $O(f_1)$  */  
    P2; /* P2的复杂度为  $O(f_2)$  */  
else  
    P3; /* P3的复杂度为  $O(f_3)$  */  
总复杂度为  $\max( O(f_1), O(f_2), O(f_3) )$ 。
```

1.9常用复杂度函数

□ 常用函数增长表

函数	输入规模 n					
	1	2	4	8	16	32
1	1	1	1	1	1	1
$\log_2 n$	0	1	2	3	4	5
n	1	2	4	8	16	32
$n \log_2 n$	0	2	8	24	64	160
n^2	1	4	16	64	256	1024
n^3	1	8	64	512	4096	32768
2^n	2	4	16	256	65536	4294967296
$n!$	1	2	24	40326	2092278988000	26313×10^{33}

1.9常用复杂度函数（续）



1.9常用复杂度函数（续）

每秒10亿指令计算机的运行时间表							
n	$f(n)=n$	$n \log_2 n$	n^2	n^3	n^4	n^{10}	2^n
10	.01 μ s	.03 μ s	.1 μ s	1 μ s	10 μ s	10sec	1 μ s
20	.02 μ s	.09 μ s	.4 μ s	8 μ s	160 μ s	2.84hr	1ms
30	.03 μ s	.15 μ s	.9 μ s	27 μ s	810 μ s	6.83d	1sec
40	.04 μ s	.21 μ s	1.6 μ s	64 μ s	2.56ms	121.36d	18.3min
50	.05 μ s	.28 μ s	2.5 μ s	125 μ s	6.25ms	3.1yr	13d
100	.10 μ s	.66 μ s	10 μ s	1ms	100ms	3171yr	$4*10^{13}$ yr
1,000	1.00 μ s	9.96 μ s	1ms	1sec	16.67min	$3.17*10^{13}$ yr	$32*10^{283}$ yr
10,000	10 μ s	130.03 μ s	100ms	16.67min	115.7d	$3.17*10^{23}$ yr	
100,000	100 μ s	1.66ms	10sec	11.57d	3171yr	$3.17*10^{33}$ yr	
1,000,000	1.0ms	19.92ms	16.67min	31.71yr	$3.17*10^7$ yr	$3.17*10^{43}$ yr	

μ s = 微秒 = 10^{-6} 秒

ms = 毫秒 = 10^{-3} 秒

sec = 秒

min = 分

hr = 小时

yr = 年

d = 天

1.10最大子列和问题——算法1

- 给定 N 个整数的序列 $\{A_1, A_2, \dots, A_N\}$ ，求函数的最大值 $f(i, j) = \max\{0, \sum_{k=i}^j A_k\}$ ，若全部整数为负数，则最大子列和为0

```
int MaxSubseqSum1( int A[], int N )
{ int ThisSum, MaxSum = 0;
  int i, j, k;
  for( i = 0; i < N; i++ ) { /* i是子列左端位置 */
    for( j = i; j < N; j++ ) { /* j是子列右端位置 */
      ThisSum = 0; /* ThisSum是从A[i]到A[j]的子列和 */
      for( k = i; k <= j; k++ )
        ThisSum += A[k];
      if( ThisSum > MaxSum ) /* 如果刚得到的这个子列和更大 */
        MaxSum = ThisSum; /* 则更新结果 */
    } /* j循环结束 */
  } /* i循环结束 */
  return MaxSum;
}
```

$$T(N) = O(N^3)$$

1.10最大子列和问题——算法2

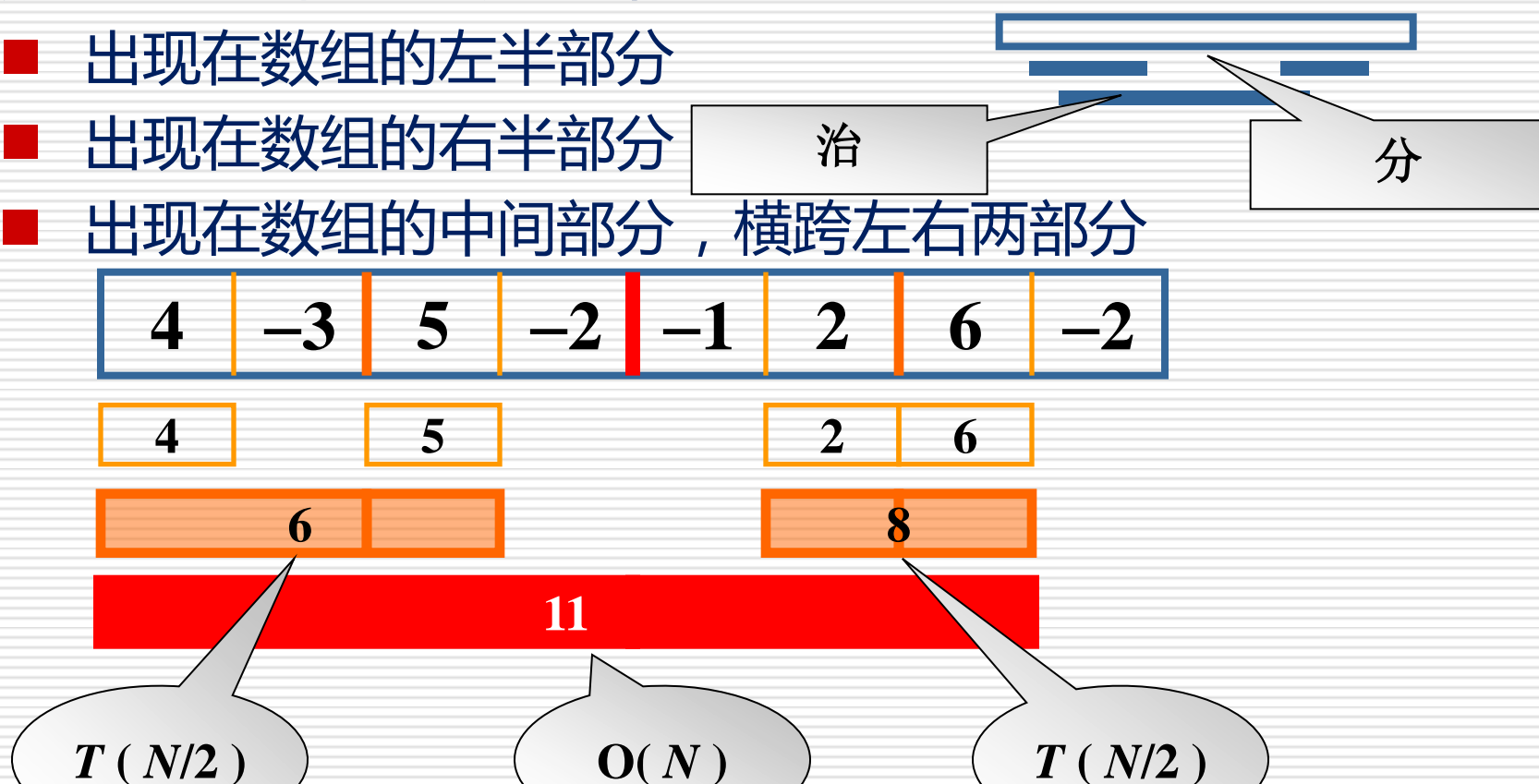
```
int MaxSubseqSum2( int A[], int N )
{ int ThisSum, MaxSum = 0;
  int i, j;
  for( i = 0; i < N; i++ ) { /* i是子列左端位置 */
    ThisSum = 0; /* ThisSum是从A[i]到A[j]的子列和 */
    for( j = i; j < N; j++ ) { /* j是子列右端位置 */
      ThisSum += A[j];
      /*对于相同的i, 不同的j, 只要在j-1次循环的基础上累加1项即可*/
      if( ThisSum > MaxSum ) /* 如果刚得到的这个子列和更大 */
        MaxSum = ThisSum; /* 则更新结果 */
    } /* j循环结束 */
  } /* i循环结束 */
  return MaxSum;
}
```

$$T(N) = O(N^2)$$

1.10最大子列和问题——算法3

□ 分治法：最大子序列和只有三种情况

- 出现在数组的左半部分
- 出现在数组的右半部分
- 出现在数组的中间部分，横跨左右两部分



$$\begin{aligned} T(N) &= 2T(N/2) + cN, & T(1) &= O(1) \\ &= 2[2T(N/2^2) + cN/2] + cN \\ &= 2^k O(1) + c k N & \text{此外 } N \text{ 不一定是 } 2^k \\ &= O(N \log N) \end{aligned}$$

结论对 $N \neq 2^k$
同样正确

1.10最大子列和问题——算法4

```
int MaxSubseqSum4( int A[], int N )
{ int ThisSum, MaxSum;
  int i;
  ThisSum = MaxSum = 0;
  for( i = 0; i < N; i++ ) {
    ThisSum += A[i]; /* 向右累加 */
    if( ThisSum > MaxSum )
      MaxSum = ThisSum; /* 发现更大和则更新当前结果 */
    else if( ThisSum < 0 ) /* 如果当前子列和为负 */
      ThisSum = 0; /* 则不可能使后面的部分和增大，抛弃之 */
  }
  return MaxSum;
}
```

-1	3	-2	4	-6	1	6	-1

$T(N) = O(N)$

该算法的核心思想是基于下面的事实：

如果整数序列 $\{a_1, a_2, \dots, a_n\}$ 的最大和子列是 $\{a_i, a_{i+1}, \dots, a_j\}$ ，那么必定有 $\sum_{k=i}^l a_k \geq 0$ 。对任意 $i \leq l \leq j$ 都成立。

因此，一旦发现当前子列和为负，则可重新开始考察一个新的子列。

1.10最大子列和问题——上述4种算法开销比较

(单位：秒)

算法		1	2	3	4
时间复杂性		$O(N^3)$	$O(N^2)$	$O(N \log N)$	$O(N)$
子列 大小	$N=10$	0.00103	0.00045	0.00066	0.00034
	$N=100$	0.47015	0.01112	0.00486	0.00063
	$N=1,000$	448.77	1.1233	0.05843	0.00333
	$N=10,000$	NA	111.13	0.68631	0.03042
	$N=100,000$	NA	NA	8.0113	0.29832

注：不包括输入子列的时间。

NA – Not Acceptable, 不可接受的时间

附、算法——概念和特性

算法是为了解决某类问题而规定的一个有限长的**操作序列**。一个算法必须满足以下**五个重要特性**：

- 1 . **有穷性**
- 2 . **确定性**
- 3 . **可行性**
- 4 . **有输入**
- 5 . **有输出**

1. 有穷性 对于任意一组合法输入值，在执行**有穷步骤**之后一定能结束，即：
算法中的每个步骤都能在**有限时间**内完成。

2. 确定性 对于**每种情况**下所应执行的操作，在算法中都有**确切**的规定，使算法的执行者或阅读者都能明确其含义及如何执行。**并且在任何条件下，算法都只有一条执行路径。**

3 . 可行性 算法中的所有操作都必须**足够基本**，都可以通过已经实现的基本操作运算有限次实现之。

4 . 有输入 作为算法加工对象的量值，通常体现为算法中的一组变量。有些输入量需要在算法执行过程中输入，而有的算法表面上可以没有输入，实际上已被嵌入算法之中。

5 . 有输出 它是一组与“输入”有确定关系的量值，是算法进行信息加工后得到的结果，这种确定关系即为算法的功能。

附、算法——设计原则

设计算法时，通常应考虑达到以下目标：

1. 正确性

2. 可读性

3. 健壮性

4. 高效率与低存储量需求

1. 正确性

首先，算法应当**满足**以特定的“规格说明”方式给出的需求。

其次，对算法是否“**正确**”的理解可以有以下四个层次：

- a. 程序中不含语法错误；
- b. 程序对于几组输入数据能够得出满足要求的结果
- c. 程序对于精心选择的、典型、苛刻且带有刁难性的几组输入数据能够得出满足要求的结果；**
- d. 程序对于一切合法的输入数据都能得出满足要求的结果；

通常以**第 c 层**意义的正确性作为衡量一个算法是否合格的标准。

2. 可读性

算法主要是为了人的**阅读与交流**，其次才是为计算机执行，因此算法应该**易于人的理解**；另一方面，晦涩难读的程序易于隐藏较多错误而难以调试。

3. 健壮性

当**输入的数据非法**时，算法应当恰当地作出反映或**进行相应处理**，而不是产生莫名奇妙的输出结果。并且，**处理出错的方法**不应是中断程序的执行，而应是**返回一个表示错误或错误性质的值**，以便在更高的抽象层次上进行处理。

4 . 高效率与低存储量需求

通常，效率指的是算法执行时间；

存储量指的是算法执行过程中所需的最大存储空间，
两者都与问题的规模有关。

Thank you very much

*Any comments and suggestions
are beyond welcome*