

第1章 引论

- 数据结构研究什么
- 基本概念和术语
- 数据类型和抽象数据类型
- 算法和算法分析
- 用C语言描述数据结构与算法

■ 1.1 数据结构研究什么

■ 电子计算机的主要用途：

☞ 早期：

主要用于数值计算。

☞ 后来：

处理逐渐扩大到非数值计算领域（能处理多种复杂的具有一定结构关系的数据）。

→数值计算问题:

例1 鸡兔同笼，鸡脚 X 只，兔脚 Y 只，问鸡、兔各几只？

例2 预计人口增长情况，现有人口13亿，要在5年内控制在15亿以内，每年的增长率不能超过多少？

数学模型：数学方程

■ 1.1 数据结构研究什么

■ 电子计算机的主要用途：

☞ 早期：

主要用于数值计算。

☞ 后来：

处理逐渐扩大到非数值计算领域（能处理多种复杂的具有一定结构关系的数据）。

→ 非数值计算问题:

数据元素之间的相互关系一般无法用数学方程加以描述。

例1.1 电话号码查询问题:

- (1) 按顺序存储方式: 需遍历表
- (2) 按姓氏索引方式: 索引

要写出好的查找算法, 取决于这张表的结构及存储方式。

电话号码表的结构和存储方式决定了查找(算法)的效率。

→非数值计算问题:

例1.2 田径赛的时间安排问题（无向图的着色问题）：

设有六个比赛项目，规定每个选手至多可参加三个项目，有五人报名参加比赛（如下表所示）设计比赛日程表，使得在尽可能短的时间内完成比赛。

姓 名	项目 1	项目 2	项目 3
丁 一	跳高	跳 远	100 米
马 二	标 枪	铅 球	
张 三	标 抢	100 米	200 米
李 四	铅 球	200 米	跳 高
王 五	跳 远	200 米	

→非数值计算问题:

----田径赛的时间安排问题解法

(1) 设用如下六个不同的代号代表不同的项目：

跳高 跳远 标枪 铅球 100米 200米

A B C D E F

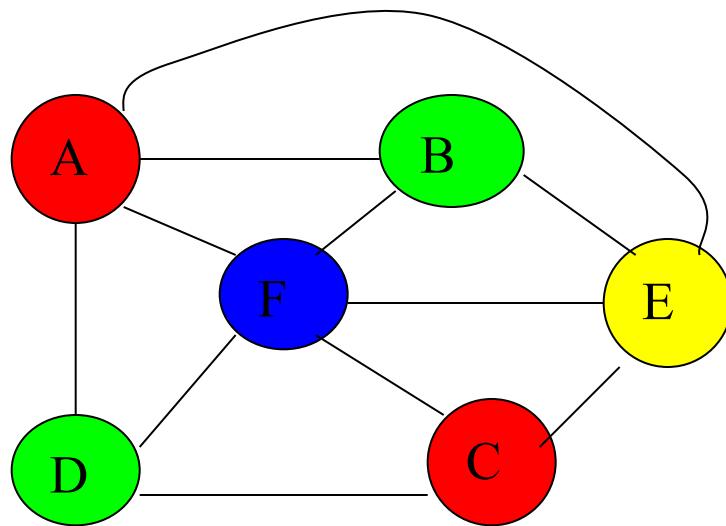
(2) 用顶点代表比赛项目

不能同时进行比赛的项目之间连上一条边。

(3) 某选手比赛的项目必定有边相连（不能同时比赛）。

姓名	项目1	项目2	项目3
丁一	A	B	E
马二	C	D	
张三	C	E	F
李四	D	F	A
王五	B	F	

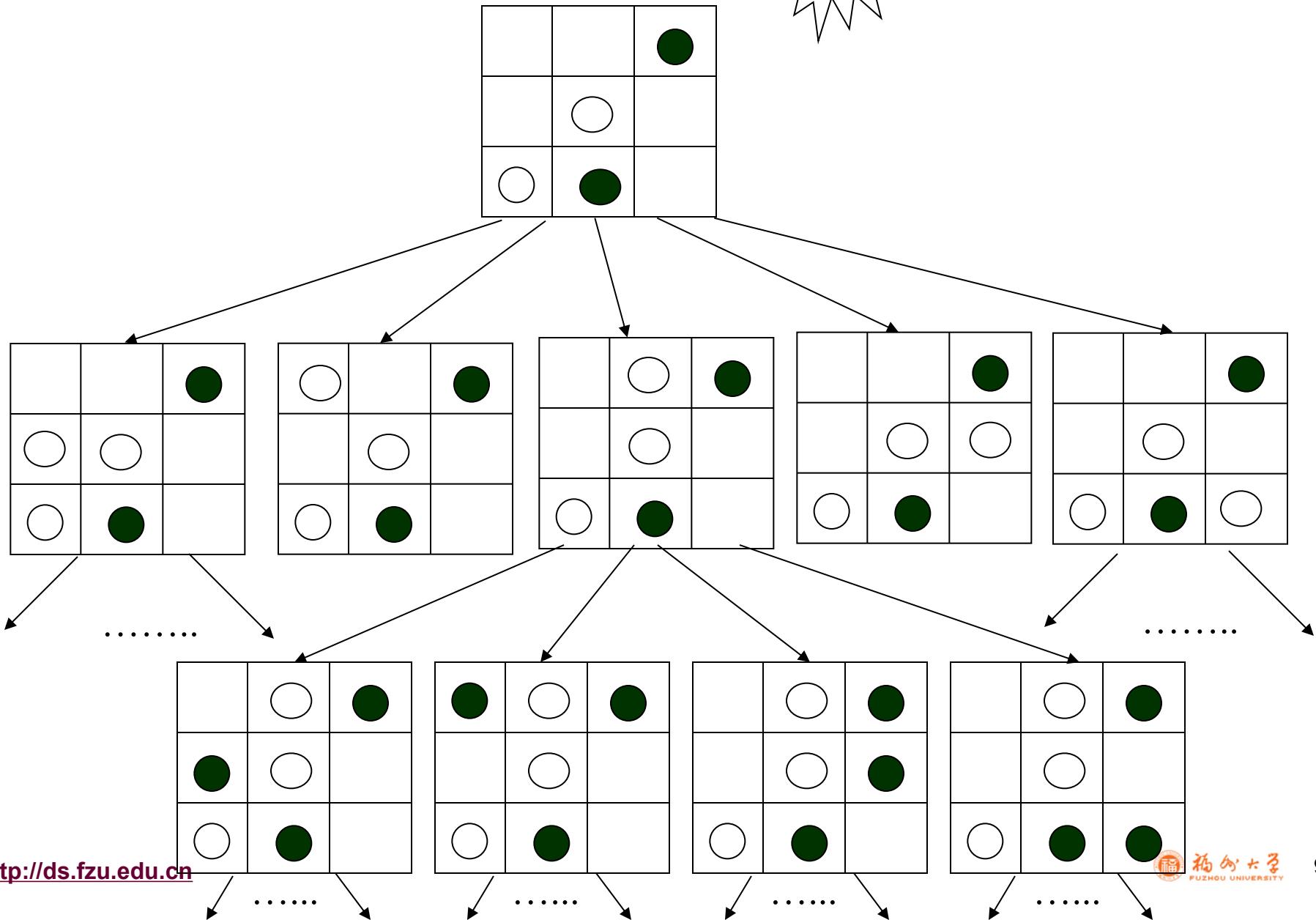
只需安排四个单位时间进行比赛



比赛时间	比赛项目
1	A, C
2	B, D
3	E
4	F

例1.3 人机对奕问题

树



→ 非数值计算问题：

求解非数值计算问题主要考虑的是设计出合适的数据结构及相应的算法。

即：首先要考虑对相关的各种信息如何表示、组织和存储？

因此，可以认为：数据结构是一门研究非数值计算的程序设计问题中计算机的操作对象以及它们之间的关系和操作的学科。

■ 1.2 基本概念和术语

数据结构的相关名词：

- ⊕ 数据 (Data)
- ⊕ 数据元素 (Data Element)
- ⊕ 数据对象 (Data Object)
- ⊕ 数据结构 (Data Structure)

数据 (Data)

- ◆ 定义：数据是所有能被输入计算机，且能被计算机程序加工处理的各种符号集合。
- ◆ 数据包含整型、实型、布尔型、图象、字符、声音等一切可以输入到计算机中的符号集合。

数据元素 (Data Element)

- ◆ 定义：数据元素是组成数据的基本单位，是计算机程序中加工处理的基本单位。例如：

学 号	姓 名	性 别	籍 贯	出生年月	住 址
101	赵虹玲	女	河北	1983. 11	北京
...

数据项 ↓

← 数据元素

数据对象（Data Object）

- ◆ 定义：数据对象是性质相同的数据元素的集合，是数据的一个子集。
- ◆ 例如：

整数集合： $N=\{0, \pm 1, \pm 2, \dots\}$ 无限集

字符集合： $C=\{'A', 'B', \dots, 'Z'\}$ 有限集

数据结构（Data Structure）

- ◆ 数据结构反映了数据的内部构成，即数据由哪些成分数据成，以什么方式构成，呈什么结构。数据结构是数据存在的形式。
- ◆ 简单地说，数据结构就是相互之间存在一种或多种特定关系的数据元素的集合。
- ◆ 定义：按某种逻辑关系组织起来的一批数据（或称带结构的数据元素的集合）应用计算机语言并按一定的存储表示方式把它们存储在计算机的存储器中，并在其上定义了一个运算的集合。

■ 数据结构的三个方面：

→ 逻辑结构

- ◆ 数据元素间抽象化的相互关系（简称为数据结构）。
- ◆ 与数据的存储无关，独立于计算机，它是从具体问题抽象出来的数学模型。

→ 存储结构（物理结构）

- ◆ 数据元素及其关系在计算机存储器中的存储方式。
- ◆ 是逻辑结构用计算机语言的实现，它依赖于计算机语言。

→ 运算（算法）

- ◆ 包括数据的检索、排序、插入、删除、修改等

→ 逻辑结构

(1) 线性结构

有且仅有一个开始和一个终端结点，并且所有结点都最多只有一个直接前趋和一个后继。

例如：线性表、栈、队列、串

(2) 非线性结构

一个结点可能有多个直接前趋和直接后继。

例如：树、图、集合

→ 存储结构

(1) 顺序存储: 借助元素在存储器中的相对位置来表示数据元素间的逻辑关系。

(2) 链式存储: 借助指示元素存储地址的指针表示数据元素间的逻辑关系

同一种逻辑结构可采用不同的存储方法，这主要考虑的是运算方便及算法的时空要求。



顺序存储



$$Loc(\text{元素}i) = L_o + (i-1)*m$$

Back

链式存储



Back

→ 逻辑结构存储结构小结

- (1) 数据的逻辑结构、存储结构和数据的运算（算法）构成了数据结构三个方面的含义。
- (2) 程序设计的实质是对实际问题选择一个好的数据结构，加之设计一个好的算法。而好的算法在很大程度上取决于描述实际问题的数据结构。

■ 1.3 数据类型和抽象数据类型

- ◆ 具有相同数据结构的数据属同一类。同一类数据的全体称为一个数据类型（Data Type）。
- ◆ 在高级程序设计语言中，数据类型用来说明数据在数据分类中的归属。它是数据的一种属性。
- ◆ 例 在C语言中

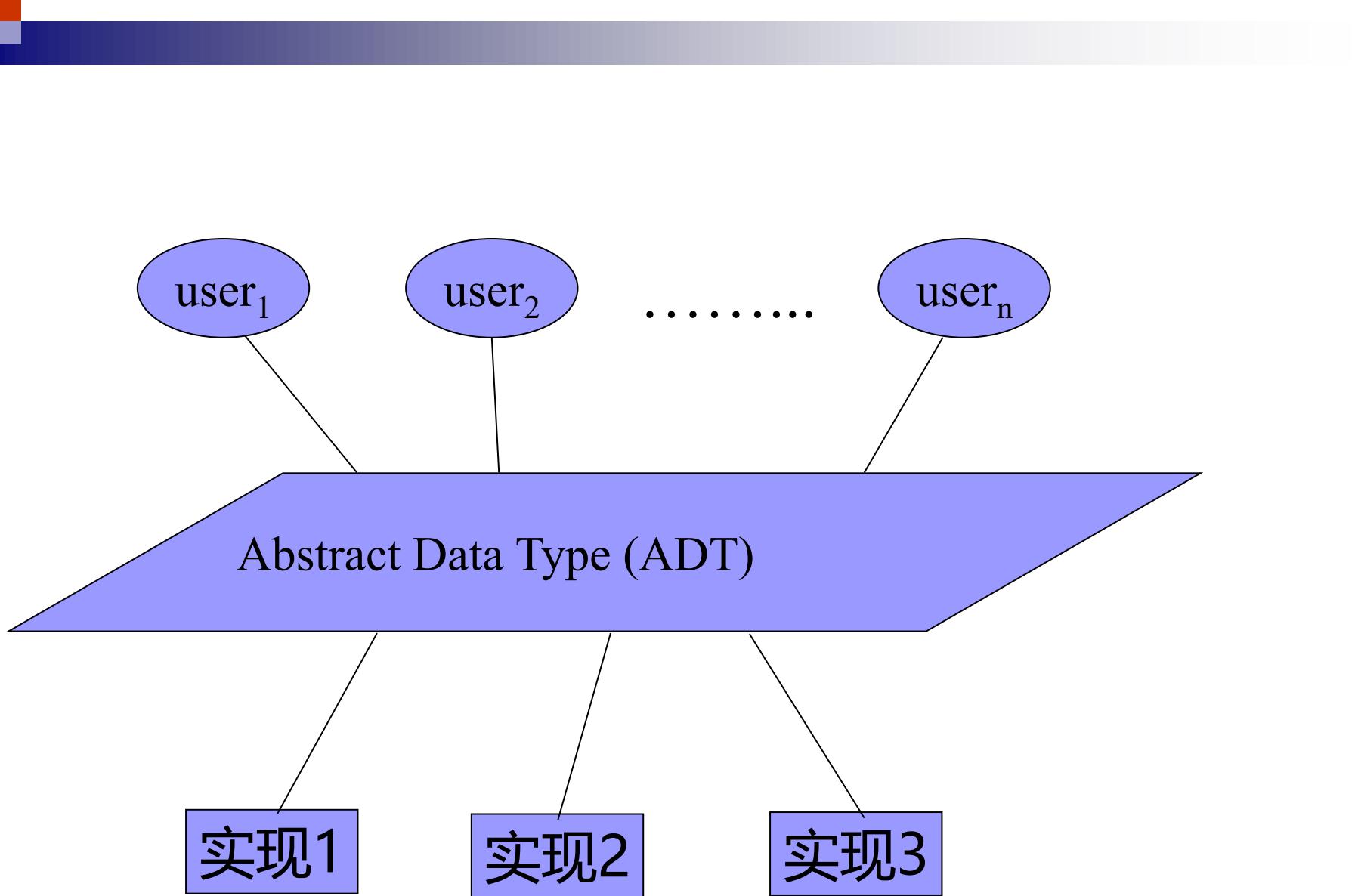
数据类型：基本类型和构造类型

基本类型：整型、浮点型、字符型

构造类型：数组、结构、联合、指针、枚举型、自定义

■ 1.3 数据类型和抽象数据类型

- ◆ 抽象数据类型(**Abstract Data Type**简称**ADT**)
 - ⇒ 抽象数据类型是用户在数据类型基础上新定义的数据类型
 - ⇒ 抽象数据类型定义包括数据组成和对数据的处理操作
 - ⇒ 从使用的角度看，一个抽象数据类型隐藏了所有使用者不关心的实现细节。





使用抽象数据类型观点，可以使程序模块的实现与使用分离，从而能够独立地考虑模块的外部界面，内部算法和数据结构的实现。这也可以使应用程序只要按抽象数据类型的观点统一其调用方式，不管其内部换用其他的任何数据表示方式和运算实现算法，对应用程序都没有影响。这个特征对系统的维护和修改非常有利。

在许多程序设计语言中预定义的类型，例如整数类型、浮点类型、指针类型等，都可以看作是简单的抽象数据类型。

从原则上讲，数据结构课程中讨论的各种表、栈、队列、二叉树等，也可以像整数、浮点数等一样定义为程序设计语言预定义的抽象数据类型。

■ 1.3 数据类型和抽象数据类型

抽象数据类型可以用以下的三元组来表示：

$$\text{ADT} = (\text{D}, \text{R}, \text{P})$$

数据对象 D上的关系集 D上的操作集

ADT
常用
定义
格式

ADT抽象数据类型名 {

数据对象：<数据对象的定义>

数据关系：<数据关系的定义>

基本操作：<基本操作的定义>

} ADT抽象数据类型名

例：抽象数据类型Circle

ADT **circle** {

 data:

float r ;

 operations:

void constructor()

 处理：构造一个圆

float area ()

 { return(3.14*r*r); }

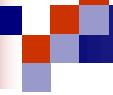
float circumference()

 { return(2*3.14*r); }

} ADT **circle**;

■ 1.4 算法和算法分析

- 算法（Algorithm）：解决某一特定问题的具体步骤的描述，是指令的有限序列。
- 算法的描述：类C语言（介于伪码和C语言之间）
- 一个算法必须满足以下几条性质：
 - (1) **有穷性**：一个算法必须总是在执行有穷步之后结束，且每一步都在有穷时间内完成。例
 - (2) **确定性**：算法中每一条指令必须有确切的含义。不存在二义性。且算法只有一个入口和一个出口。
 - (3) 输入：一个算法有零个或多个输入
 - (4) 输出：一个算法有至少一个输出



■ 例：一个不是算法的例子

(1)begin

(2) $n=0$

(3) $n=n+1$

(4)repeat (3)

(5)end

■ 例：一个不超过100次计数的算法

(1)begin

(2) $n=0$

(3) $n=n+1$

(4)if $n \geq 100$ do (5)

else repeat(3)

(5)output n

(6)end

■ 1.4 算法和算法分析

- 算法（Algorithm）：解决某一特定问题的具体步骤的描述，是指令的有限序列。
- 算法的描述：类C语言（介于伪码和C语言之间）
- 一个算法必须满足以下几条性质：
 - (1) **有穷性**：一个算法必须总是在执行有穷步之后结束，且每一步都在有穷时间内完成。例
 - (2) **确定性**：算法中每一条指令必须有确切的含义。不存在二义性。且算法只有一个入口和一个出口。
 - (3) 输入：一个算法有零个或多个输入
 - (4) 输出：一个算法有至少一个输出

■ 1.4 算法和算法分析

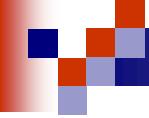
■ 算法与程序

- 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)。
- 例如操作系统，是一个在无限循环中执行的程序，因而不是一个算法。
- 操作系统的各种任务可看成是单独的问题，每一个问题由操作系统中的一个子程序通过特定的算法来实现。该子程序得到输出结果后便终止。

■ 1.4 算法和算法分析

■ 算法复杂性分析

- 评价一个算法主要是看运行算法所需要的计算机资源的量，即算法的复杂性。
- 计算机的资源，最重要的是时间和空间资源。需要的时间资源的量称为**时间复杂性 $T(n)$** ；需要的空间资源的量称**为空间复杂性 $S(n)$** 。其中n是问题的规模（输入大小）
- 算法中所有基本操作重复执行的次数之和即为时间复杂性 $T(n)$ ，**为方便计算**，假设算法中用到的所有不同的元运算各执行一次所需要的时间都是一个单位时间。



→ 算法的时间复杂性

- 通常考虑3种情况下的时间复杂性

(1) 最坏情况下的时间复杂性

$$T_{\max}(n) = \max\{ T(I) \mid \text{size}(I)=n \}$$

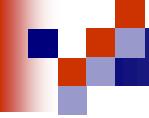
(2) 最好情况下的时间复杂性

$$T_{\min}(n) = \min\{ T(I) \mid \text{size}(I)=n \}$$

(3) 平均情况下的时间复杂性

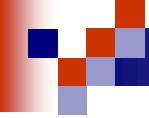
$$T_{\text{avg}}(n) = \sum_{\text{size}(I)=n} p(I)T(I)$$

其中 I 是问题的规模为 n 的实例， $p(I)$ 是实例 I 出现的概率。



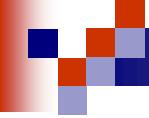
→ 算法的时间复杂性

- 以上3种情况下的时间复杂性各从某一个角度来反映算法的效率，各有各的局限性，也各有各的用处。实践表明，可操作性最好最有实际价值的是最坏情况下的时间复杂性。本书对算法时间复杂性的分析主要采用最坏情况下的时间复杂性分析。



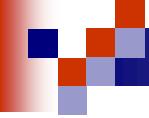
→ 算法渐近复杂性

- 设 $T(n)$ 是关于算法A的时间复杂性函数。
- 一般来说，当 $n \rightarrow \infty$ 时， $T(n) \rightarrow \infty$ 。对于 $T(n)$ ，若存在 $t(n)$ ，使得当 $n \rightarrow \infty$ 时， $(T(n) - t(n))/T(n) \rightarrow 0$ ，那么，就说 $t(n)$ 是 $T(n)$ 的渐近性态，为算法的渐近复杂性。
- 在数学上， $t(n)$ 是 $T(n)$ 的渐近表达式，是 $T(n)$ 略去低阶项留下的主项。它比 $T(n)$ 简单。
- 例如：当 $T(n)=2n^2+3n+2$ 时 $t(n)$ 的一个答案是 $2n^2$



→ 算法渐近复杂性

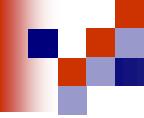
- 鉴于分析算法复杂性的目的在于比较求解同一问题的2个不同算法的效率。而当比较的2个算法的渐近复杂性的阶不相同时，只要能确定出各自的阶，就可以判定哪个算法的效率高。即此时的渐近复杂性分析**只需关心 $t(n)$ 的阶即可。**



→ 算法渐近复杂性

- 综上所述，已经给出了简化算法复杂性分析的方法，即只要考察当问题的规模充分大时，算法复杂性在渐近意义上的阶。为此，需引入以下渐近复杂性的记号。
- 设 $f(n)$ 和 $g(n)$ 是定义在正数集上的函数：

如果存在正的常数 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 $f(n)$ 当 n 充分大时上有界，且 $g(n)$ 是它的一个上界，记为 $f(n)=O(g(n))$



渐近分析的记号

在下面的讨论中，对所有 n , $f(n) \geq 0$, $g(n) \geq 0$ 。

⊕ (1) 渐近上界记号 O

$O(g(n)) = \{ f(n) \mid \text{存在正常数} c \text{ 和 } n_0 \text{ 使得对所有 } n \geq n_0$

有: $0 \leq f(n) \leq cg(n)$ }

⊕ (2) 渐近下界记号 Ω

$\Omega(g(n)) = \{ f(n) \mid \text{存在正常数} c \text{ 和 } n_0 \text{ 使得对所有 } n \geq n_0$

有: $0 \leq cg(n) \leq f(n)$ }



⊕ (3) 非紧上界记号o

$o(g(n)) = \{ f(n) \mid \text{对于任何正常数} c > 0, \text{ 存在正数和} n_0 > 0 \text{ 使得}$
 $\text{对所有} n \geq n_0 \text{ 有: } 0 \leq f(n) < cg(n) \}$

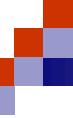
等价于 $f(n) / g(n) \rightarrow 0, \text{ as } n \rightarrow \infty.$

⊕ (4) 非紧下界记号ω

$\omega(g(n)) = \{ f(n) \mid \text{对于任何正常数} c > 0, \text{ 存在正数和} n_0 > 0 \text{ 使得}$
 $\text{对所有} n \geq n_0 \text{ 有: } 0 \leq cg(n) < f(n) \}$

等价于 $f(n) / g(n) \rightarrow \infty, \text{ as } n \rightarrow \infty.$

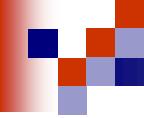
$$f(n) \in \omega(g(n)) \Leftrightarrow g(n) \in o(f(n))$$



⊕ (5) 紧渐近界记号 Θ

$\Theta(g(n)) = \{ f(n) \mid \text{存在正常数} c_1, c_2 \text{ 和 } n_0 \text{ 使得对所有 } n \geq n_0 \text{ 有:}$

$$c_1 g(n) \leq f(n) \leq c_2 g(n) \}$$



渐近分析记号在等式和不等式中的意义

- ⊕ $f(n) = \Theta(g(n))$ 的确切意义是： $f(n) \in \Theta(g(n))$ 。
- ⊕ 一般情况下，等式和不等式中的渐近记号 $\Theta(g(n))$ 表示 $\Theta(g(n))$ 中的某个函数。
 - ⊕ 例如： $2n^2 + 3n + 1 = 2n^2 + \Theta(n)$ 表示
 - ⊕ $2n^2 + 3n + 1 = 2n^2 + f(n)$, 其中 $f(n)$ 是 $\Theta(n)$ 中某个函数。
- ⊕ 等式和不等式中渐近记号 O, o, Ω 和 ω 的意义是类似的。



渐近分析中函数比较

- ⊕ $f(n) = O(g(n)) \approx a \leq b;$
- ⊕ $f(n) = \Omega(g(n)) \approx a \geq b;$
- ⊕ $f(n) = \Theta(g(n)) \approx a = b;$
- ⊕ $f(n) = o(g(n)) \approx a < b;$
- ⊕ $f(n) = \omega(g(n)) \approx a > b.$

算法分析中常见的复杂性函数

FUNCTION	NAME
c	Constant
$\log N$	Logarithmic
$\log^2 N$	Log-squared
N	Linear
$N \log N$	$N \log N$
N^2	Quadratic
N^3	Cubic
2^N	Exponential

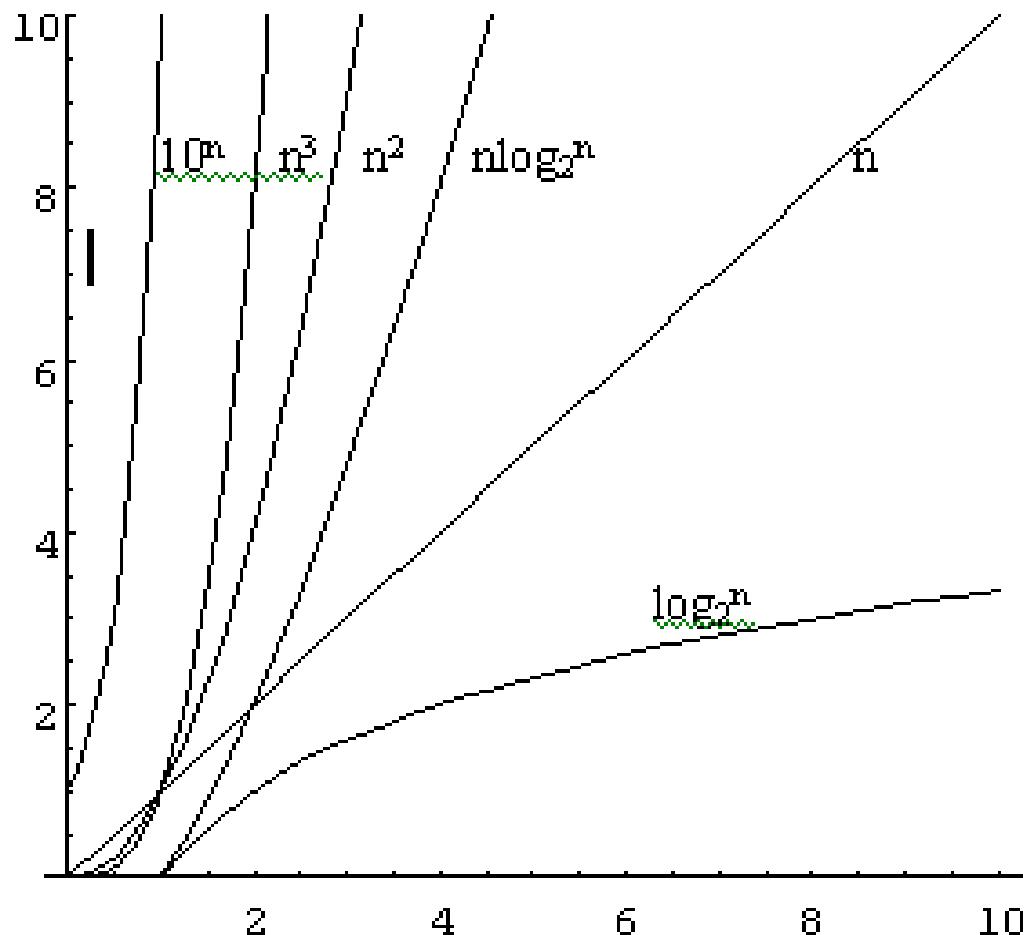
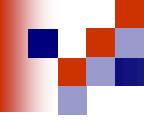


图 1.3 $f(n)$ 函数曲线变化速度的比较



常用的时间复杂度频率表：

$\log_2 n$	n	$n \log_2 n$	n^2	n^3	2^n
0	1	0	1	1	2
1	2	2	4	8	4
2	4	8	16	64	16
3	8	24	64	512	256
4	16	64	256	5096	65536
5	32	160	1024	32768	2147483648

算法分析的基本法则

非递归算法：

- (1) for / while 循环

循环体内计算时间*循环次数；

- (2) 嵌套循环

循环体内计算时间*所有循环次数；

- (3) 顺序语句

各语句计算时间相加；

- (4) if-else语句

if语句计算时间和else语句计算时间的较大者。

语句频度：是指该语句在算法中重复执行的次数。

例：{ ++x; }

语句频度为 1，即时间复杂度为 $O(1)$ 。

例：{ ++x;

s=0; }

语句频度为 2，其时间复杂度仍为 $O(1)$ ，即常量阶。

语句频度：是指该语句在算法中重复执行的次数。

例：两个n阶矩阵相乘

算法语句

算法语句	对应的语句频度
1 for (i=0; i< n;i++)	n
2 for (j=0; j<n;j++)	n^2
3 { c[i][j]=0;	n^2
4 for (k=0;k< n; k++)	n^3
5 c[i][j]=c[i][j]+a[i][k]*b[k][j];	n^3
}	

$$\text{总执行次数: } T_n = 2n^3 + 2n^2 + n$$

$$\therefore T(n) = O(n^3)$$

■ 例 求下列程序段的时间复杂度 (C)

x=n;

y=0;

while(x>=(y+1)*(y+1))

y++;

A.O(1)

B.O(n)

C.O($n^{1/2}$)

D.O(\log_2^n)

■ 1.5 用C语言描述数据结构与算法

- 变量和指针
- 函数与参数传递
- 结构
- 动态存储分配

→ 变量和指针

■ 变量具有以下属性：

- 变量名：标识变量的符号
- 地址：变量所占据的存储单元的地址
- 大小：变量所占据的存储空间的数量
- 类型：变量所取的值域及能执行的运算集
- 值：变量所占据的存储单元的内容
- 生命期：执行程序期间变量存在的时段
- 作用域：程序中变量被引用的语句范围

→ 变量和指针

- 指针变量就是地址变量，用于存放地址的变量。
- 例如：

```
int k,n,*p;
```

```
n=8;
```

```
p=&n;
```

```
k=*p;
```

→ 函数与参数传递

- 函数定义包括**4**个部分： 函数名、 形参、 返回类型和函数体。
- 函数使用者通过函数名来调用函数
- 函数的返回值通过函数体中的**return**语句返回， 不需要返回值的函数类型为**void**
- 调用函数时传递给形参的实参必须和形参在类型、 个数、 顺序上保持一致。
 - 值传递
 - 地址传递（需将形参声明为指针类型）

→ 结构

(1) 定义结构

- C语言的结构为自定义数据类型提供了灵活方便的方法
- 结构由结构名和结构的数据成员组成。

struct 结构名

{

 数据成员列表;

};

→ 结构

(2) 指向结构的指针

- 指向结构的指针值是相应的结构变量所占据的内存空间的首地址。
- 例如，如果已经定义了一个结构**st**，则语句

```
struct st *p; //定义一个指向结构st的指针
```

→ 结构

(3) 用**typedef**定义新数据类型

- 关键字**typedef**常与结构一起用于定义新数据类型。
- 例如，下面是用**typedef**和结构定义矩形数据类型**Rectangle**的例子。

```
typedef struct renode * Rectangle  
typedef struct renode  
{  
    int x,y,h,w; // (x,y)是矩形左下角点的坐标  
}Reinode;
```

→ 结构

(4) 访问结构变量的数据成员

- 对于结构类型的变量用圆点运算符“.”访问结构变量的数据成员。
- 定义为指向结构的指针类型的变量用箭头运算符“->”访问结构变量的数据成员。
- 例如

```
Recnode rr;
Rectangle R;
R=&rr;
rr.x=1;
rr.y=1;
rr.h=2;
rr.w=3;
printf("x=%d y=%d\n",R->x, R->y);
printf("Height=%d Width=%d\n",rr.h,rr.w);
```

→ 结构

(5) 新数据类型变量初始化

- 使用自定义数据类型变量前通常需要初始化操作。
- 如下面的函数用于说明一个**Rectangle**型变量并对其初始化。

```
Rectangle RecInit()
```

```
{
```

```
    Rectangle R=(Rectangle)malloc(sizeof *R);
```

```
    R->x=0; R->y=0; R->h=0; R->w=0;
```

```
    return R;
```

```
}
```

→ 动态存储分配

(1) 动态存储分配函数**malloc()**和**free()**

■ **malloc()**函数

- 其作用是在内存的动态存储区中分配一个长度为**size**的连续空间
- 此函数是一个指针型函数，返回的指针指向该分配域的开头位置
- 如果此函数未能成功地执行（如内存空间不足），则返回空指针(**NULL**)

→ 动态存储分配

(1) 动态存储分配函数**malloc()**和**free()**

■ **free()**函数

- 其作用是释放指针变量所指向的动态空间，使这部分空间能重新被其他变量使用。
- 如 **free(p);** //释放指针变量 p 所指向的已分配的动态空间
- **free**函数无返回值

→ 动态存储分配

(1) 动态存储分配函数**malloc()**和**free()**

```
char *str;  
if((str=(char *)malloc(10))==0)  
{  
    printf("内存不足\n");  
    exit(1); //退出  
}  
strcpy(str, "Hello");  
printf("String is %s\n",str);  
free(str);
```

→ 动态存储分配

(2) 动态数组

■ 为了在运行时创建一个大小可动态变化的一维浮点数组x，可先将x声明为一个**float**类型的指针。然后用函数**malloc()**为数组动态地分配存储空间。

■ 例如： **float *x=malloc(n*sizeof(float));**

创建一个大小为n的一维浮点数组，然后可用
x[0],x[1],...,x[n-1]访问每个数组元素。

→ 动态存储分配

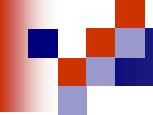
(2) 二维数组

- C语言提供了多种声明二维数组的机制。在许多情况下，当形式参数是一个二维数组时，必须指定其第二维的大小。
- 为了克服这种限制，可以使用动态分配的二维数组。
- 举例

- 例如，下面的函数创建一个int类型的动态工作数组，这个数组有r行和c列。

```
int **malloc2d(int r,int c)
{
    int i;
    int **t=malloc(r *sizeof(int *));
    for(i=0; i<r; i++)
        t[i]=malloc(c *sizeof(int));
    return t;
}
```

其他类型的二维动态数组可用类似方法创建。



本章小结

- 熟悉数据结构、数据类型、ADT等基本概念
- 理解数据结构的三个方面的内容
- 理解算法的概念及性质
- 掌握算法的算法复杂性概念及算法渐近复杂性的数学表述
- 掌握用C语言描述数据结构与算法的方法