第一章 绪论

2022-02-26　　14:45

一、数据结构研究的内容

★ 通常用计算机解题的步骤：

【一、具体问题抽象为数学模型】

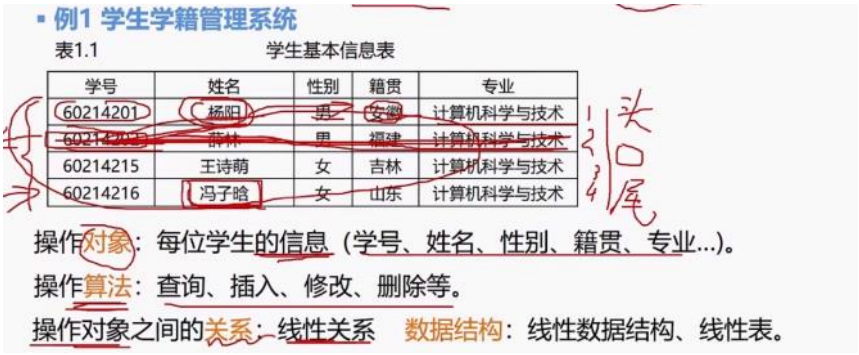
实质：分析问题—>提取操作对象—>找出操作对象之间的关系—>用数学语言描述（数据结构）

【二、设计算法】

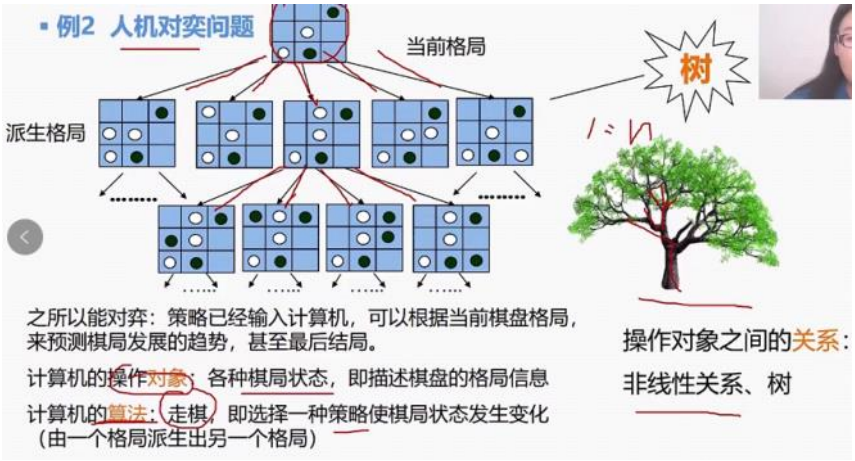
【三、编程、调试、运行】

★ 应用举例：

例一、学生学籍管理系统



例二、人机对弈



例三、地图导航—求最短路径（最快路径）

★ 小结：

- 这些问题的共性都是无法用数学的公式或方程来描述，是一些“非数值计算”的程序设计问题。
- 描述非数值计算问题的数学模型不是数学方程，而是诸如表、树和图之类的具有逻辑关系的数据。
- 数据结构是一门研究非数值计算的程序设计中计算机的操作对象以及它们之间的关系和操作的学科。

二、基本概念和术语

★ 基本概念

数据：

是能输入计算机且能被计算机处理的各种符号的集合

信息的载体

是对客观事物符号化的表示

能够被计算机识别、存储和加工

包括：

数值型的数据：整型、实数等

非数值型的数据：文字、图像、图形、声音等

数据元素：

是数据的基本单位，在计算机程序中通常作为一个整体进行考虑和处理。

也简称为元素，或称为记录、结点或顶点

一个数据元素可由若干个数据项组成

数据项：

构成数据元素的不可分割的最小单位

学籍表

学 号	姓 名	性 别	出生日期	政治面貌
0001	陆 宇	男	986/09/02	团员
0002	李 明	男	1985/12/25	党员
0003	汤晓影	女	1986/03/26	团员

数据项

数据元素

数据、数据元素、数据项三者之间的关系：

数据>数据元素>数据项

例：学生表>个人纪录>学号、姓名...

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：

整数数据对象是集合N={0，±1，±2，…，}

字母字符数据对象是集合C={‘A’，‘B’，…，‘Z’}

学籍表也可看作一个数据对象

数据元素与数据对象

数据元素——组成数据的基本单位

与数据的关系：是集合的个体

数据对象——性质相同的数据元素的集合

与数据的关系：集合的子集

★ 数据结构：

概念：

数据元素不是孤立存在的，它们之间存在着某种关系，数据元素相互之间的关系称为结构

是指相互之间存在一种或多种特定关系的数据元素集合

或者说，数据结构是带结构的数据元素的集合。

数据结构包括以下三个方面的内容：

1.数据元素之间的逻辑关系，也称逻辑结构

2.数据元素及其关系在计算机内存中的表示（又称为印像），称为数据的物理结构或数据的存储结构。

3.数据的运算和实现，即对数据元素可以施加的操作以及这些操作在相应的存储结构上的实现。

数据结构的两个层次

逻辑结构：

描述数据元素之间的逻辑关系

与数据的存储无关，独立于计算机

是从具体问题抽象出来的数学模型

物理结构（存储结构）：

数据元素及其关系在计算机存储器中的结构（存储方式）

是数据结构在计算机中的表示

逻辑结构与存储结构的关系：

存储结构是逻辑关系的映像与元素本身的映像

逻辑结构是数据结构的抽象，存储结构是数据结构的实现

两者综合起来建立了数据元素之间的结构关系。

逻辑结构的种类：

划分方法一：

（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直接后继

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

划分方法二：

（1）集合结构：结构中的数据元素之间除了之间除了同属于一个集合的关系外，无其他任何关系

（2）线性结构：结构中的数据元素之间存在着一对一的线性关系。

（3）树形结构：结构中的数据元素之间存在着一对多的层次关系。

（4）图形结构或网状结构：结构中的数据元素之间存在着多对多的任意关系。

物理结构的种类：

顺序存储结构

用一组连续的存储单元依次存储数据元素，数据元素之间的逻辑关系由元素的存储位置来表示。

链式存储结构

用一组任意的存储单元存储数据元素，数据元素之间的逻辑关系用指针来表示。

索引存储结构

散列存储结构

三、抽象数据类型的表示与实现

在使用高级程序设计语言编写程序时，必须对程序中出现的每个变量、常量或表达式，明确说明它们所属的数据类型。如C语言中char，float类型，一些最基本数据结构可以用数据类型来实现，如数值、字符串等，二另一些常用的数据结构，如栈、队列、树、图等，不能直接用数据类型来表示。高级语言中的数据类型明显或隐含地规定了在程序执行期间变量和表达的所有可能的取值范围。

由此可以看出数据类型的作用：1.约束变量或常量的取值范围 2.约束变量或常量的操作

★ 定义：

数据类型：是一组性质相同的值的集合以及定义于这个值集合上的一组操作的总称

数据类型=值的集合+值集合上的一组操作

抽象数据类型：是指一个数学模型以及定义在次数学模型上的一组操作。（ADT）

由用户定义，从问题抽象出数据模型（逻辑结构）

还包括定义在数据模型上的一组抽象运算（相关操作）

不考虑计算机内的具体存储结构于运算的具体实现算法

★ 使用：

抽象数据类型的形式定义：

抽象数据类型可用（D，S，P）三元组表示

其中：D是数据对象

S是D上的关系集

P是对D的基本操作集

基本操作定义格式说明：

参数表：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。

若初始条件为空，则省略之。

操作结果：说明操作正常完成之后，数据结构的变化状况和应返回的结果。

抽象数据类型定义举例：Circle的定义

ADT 抽象数据类型名	ADT Circle
{	{
Data	数据对象：D={r,x,y| r,x,y 均为实数}
数据对象的定义	数据关系：R={< r,x,y >| r是半径,<x,y>是圆心坐标}
数据元素之间逻辑关系的定义	基本操作：
Operation	Circle(&C, r, x ,y)
操作 1	操作结果：构造一个圆。
初识条件	double Area©
操作结果描述	初识条件：圆已存在。
操作 2	操作结果：计算面积。
….	double Circumference©
操作 n	初始条件：圆已存在。
….	操作结果：计算周长。
}ADT 抽象数据类型名	}ATD Circle

★ 抽象数据类型如何实现：

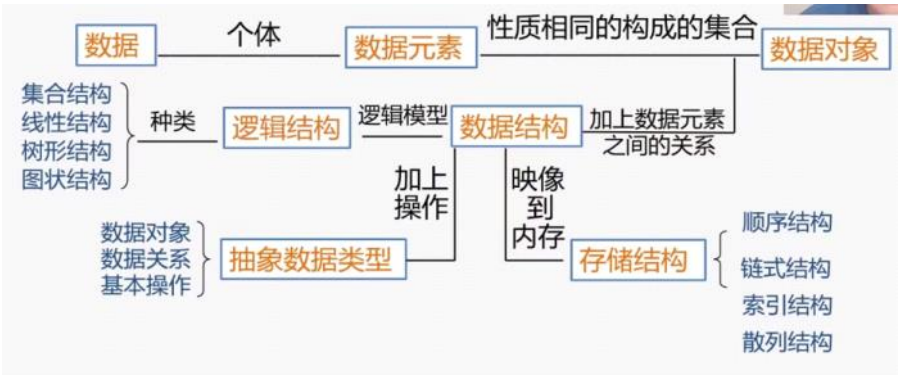
C语言实现抽象数据类型：用已有数据类型定义描述它的存储结构；用函数定义描述它的操作

抽象数据类型可以通过固有的数据类型（如整型、实型、字符型等）来表示和实现。

```
Void  assign (Complex  *A,  float real ,  float imag){
        A->realpart = real;                    /* 实部赋值 */
        A->imagpart= imag                    /* 虚部赋值 */
}                                           /* End of assign()*/

void add (Complex  *c, Complex  A, Complex  B) {     /* c = A + B */
        c->realpart = A.realpart +B.realpart;          /* 实部相加*/
        c->imagpart= A.imagpar+B.imagpart;             /* 虚部相加*/
}    /* End of Add() */
.....
```

★ 概念小结：



四、算法和算法分析

★ 算法：

算法的定义

对特定问题求解方法和步骤的一种描述，它是指令的有限序列。其中每个指令表示一个或多个操作。

简而言之，算法就是解决问题的方法和步骤。

算法的描述

自然语言：英语、中文

流程图：传统流程图、NS流程图

伪代码：类语言（类C语言）

程序代码：C语言程序、Java程序

算法与程序

算法是解决问题的一种方法或一个过程，考虑如何将输入转换成输出，一个问题可以用多种算法。

程序是用某种程序设计语言对算法的具体实现。

程序=数据结构+算法

数据结构通过算法实现操作

算法根据数据结构设计程序

算法的特性：

有穷性、确定性、可行性、输入、输出

算法设计的要求：

正确性、可读性、健壮性、高效性

★ 算法分析

算法在满足前面的特性的前提下，主要考虑算法的效率，通过算法的效率高低来评判不同算法的优劣程度。算法效率主要从以下两个方面考虑：1.时间效率（时间复杂度）2.空间效率（空间复杂度）

但这两个方面有时候是矛盾的。

算法时间效率的度量

算法时间效率可以用依据该算法编制的程序在计算机上执行所消耗的时间来度量。度量的方法有两种

一种为事后统计，一种为事前分析，但事前分析需要编写程序来实现，会耗费更多的时间与精力

并且所得结果依赖于计算机的软硬件等环境因素，掩盖算法本身的优劣。

因此我们在判断一个算法的时间效率时，通过事前分析相对来说更好。

事前分析判断算法时间效率：

一个算法的运行时间是指一个算法在计算机上运行所耗费的时间大致可以等于计算机执行一种简单操作(如赋值、比较、移动等）所需要的时间与算法中进行的简单操作次数乘积

算法运行时间=一个简单操作所需要的时间×简单操作次数

=Σ每条语句的执行次数×该语句执行一次所需要的时间

一般情况下我们假设执行每条语句所需要的时间均为单位时间，此时对算法的运行时间的讨论就可转化为讨论该算法中所有语句的执行次数，即频度之和。

例一：两个n×n矩阵相乘的算法可描述为：

```
1  for(i=1;i<=n;i++)                          // n+1次
2      for(j=1;j<=n;j++){                     // n(n+1)次
3              c[i][j]=0;                     // n*n次
4              for(k=0;k<n;k++)               // n*n*(n+1)次
5                  c[i][j]=c[i][j]+a[i][k]*b[k][j];  // n*n*n次
6      }
```

- 我们把算法所耗费的时间定义为**该算法中每条语句的频度之和**，则上述算法的时间消耗T(n)为：

$$T(n)=2n^3+3n^2+2n+1$$

这是一个关于n的函数

为了便于比较不同算法的时间效率，我们仅比较它们的数量级即可

若有某个辅助函数f(n)，使得当n趋近于无穷大时，T(n)/f(n)的极限值为不等于零的常数，则称f(n)是T(n)的同数量级函数。记作T(n)=O(f(n))，称O(f(n))为算法的渐近时间复杂度，简称时间复杂度。

对于上述问题，T(n)=2n^3+3n^2+2n+1。当n—>∞时，T(n)/n^3—>2，这表示n充分大时，T(n)与n^3是同阶或同数量级，引入大"O"记号，则T(n)=O(n^3)。

且一般情况下，不必计算所有操作的执行次数，而只考虑算法中基本操作执行的次数。

算法时间复杂度定义

算法中基本语句重复执行的次数是问题规模n的某个函数f(n)，算法时间量度记作：T(n)=O(f(n))

- 算法中重复执行的次数和算法的执行时间成正比的语句
- 对算法运行时间的贡献最大
- 执行次数最多

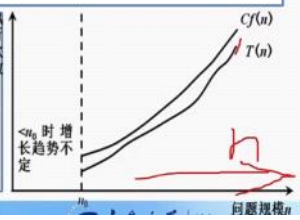
n越大算法执行时间越长

- 排序：n为记录数
- 矩阵：n为矩阵的阶数
- 多项式：n为多项式的项数
- 集合：n为元素个数
- 数：n为树的结点个数
- 图：n为图的顶点数或边数

它表示随着n的增大，算法执行的时间的增长率和f(n)的增长率相同，称渐近时间复杂度。

数学符号“O”的定义为：

若$T(n)$和$f(n)$是定义在正整数集合上的两个函数，则$T(n)=O(f(n))$表示存在正的常数C和n_0，使得当$n\geq n_0$时都满足$0\leq T(n)\leq Cf(n)$。



分析算法时间复杂度的基本方法

定理1.1

忽略所有低次幂和最高次幂系数，体现出增长率的含义。

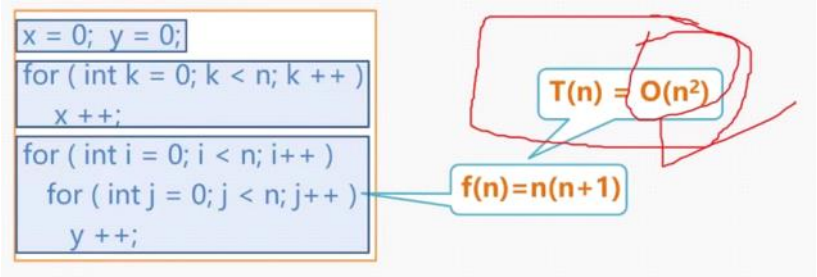
方法：

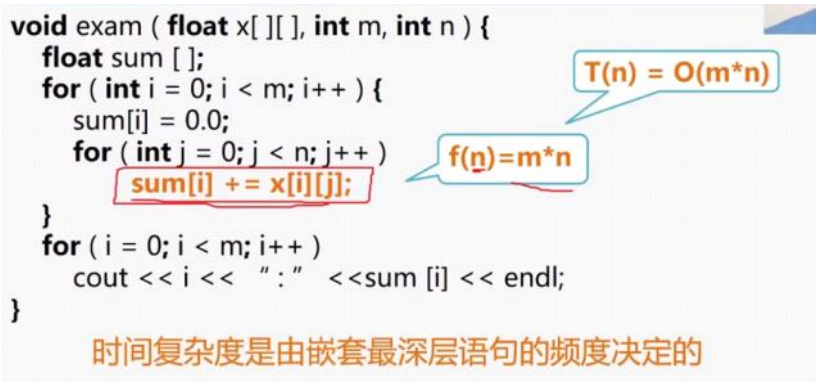
1.找出语句频度最大的那条语句作为基本语句

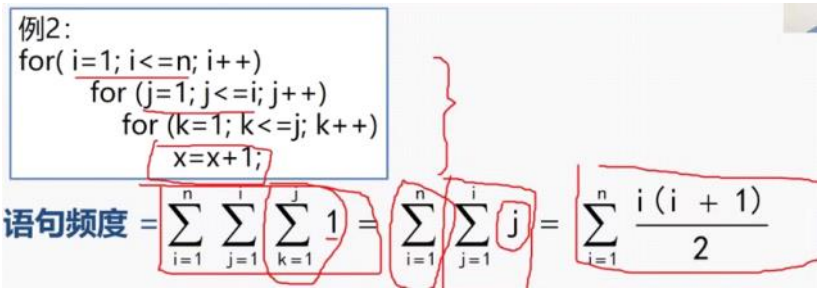
2.计算基本语句的频度得到问题规模n的某个函数f(n)

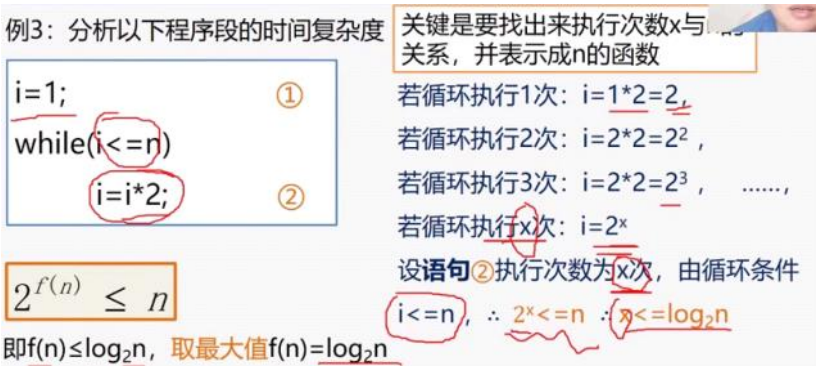
3.取其数量级用符号"O"表示

例题：



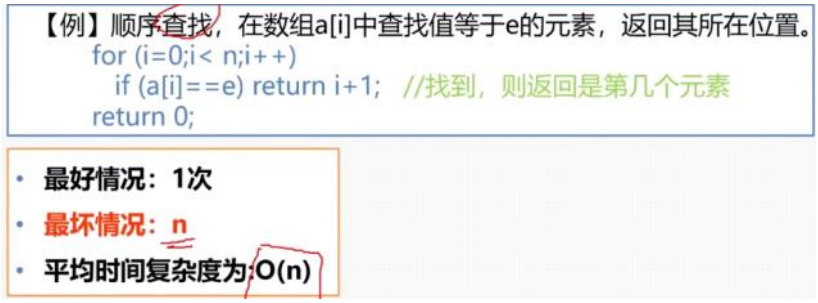






注意：

在有的情况下，算法中基本操作重复执行的次数还随问题的输入数据集不同而不同。



一般法则：

1）for循环——一次for循环运行时间至多是该for循环内语句（包括测试）的运行时间乘以迭代次数

2）嵌套的for循环——该语句的运行时间乘以该组所有的for循环的大小的乘积

3）顺序语句——各个运行时间求和即可，其中最大值就是所得的运行时间

4）if/else语句——其从不超过判断的时间加上S1和S2中运行时间较长者的总的运行时间

5）while语句——对数

几个新概念：

最坏时间复杂度：指在最坏的情况下，算法的时间复杂度。—>一般考虑它

平均时间复杂度：指在所有可能输入实例在等概率出现的情况下，算法的期望运行时间。

最好时间复杂度：指在最好情况下，算法的时间复杂度。

对于复杂的算法，可以将它分成几个容易估算的部分，然后利用大O加法法则和乘法法则。

a）加法法则：$T(n)=T1(n)+T2(n)=O(f(n))+O(g(n))=O(max(f(n),g(n)))$

b）乘法法则：$T(n)=T1(n)\times T2(n)=O(f(n))\times O(g(n))=O(f(n)\times g(n))$

渐近空间复杂度：

算法所需存储空间的度量，记作$S(n)=O(f(n))$

算法要占据的空间：

1.算法本身要占据的空间，输入/输出，指令，常数，变量等

2.算法要使用的辅助空间。

★ 总结：

设计好算法的过程