

# 第一章 绪论



# 目录

- 数据结构的重要性
- 基本概念
  - 数据结构
  - 数据逻辑结构
  - 数据物理(存储)结构
  - 抽象数据类型(ADT)
- 算法
  - 算法设计
  - 算法评价标准
  - 算法时间复杂度
  - 算法空间复杂度

# 1. 数据结构的重要性

- 设计、开发一个计算机系统，**最核心**的工作是**编写程序**
- **算法和数据结构**是**程序设计方法学**的核心
- 数据结构服务于解决实际问题的算法的设计
  - 数据及其数据之间的关系如何存储，如何对其进行操作
- Niklaus Wirth (Turing Award Winner, 1984):  
Algorithm + Data Structures = Programs/1976
- **Donald Knuth** (Turing Award Winner, 1974): The Art of Computer Programming (4卷)
- Tony Hoare (Turing Award Winner, 1980): 算法的结构和选择在很大程度上依赖于作为基础的数据结构

## 2. 基本概念-1

- **数据(Data)** 对客观事物的符号表示。在计算机科学中是指所有能输入到计算机中并被计算机程序处理的符号的**总称**
  - 数值、文字、图形、图像、视频、声音
  - 例子：图像：矩阵
  - 例子：汉字：点阵，曲线  $B(t) = (1-t)^2P_0 + 2t(1-t)P_1 + t^2P_2$
- **数据元素(Data Element)** 是数据的**基本单位**，在程序中通常作为一个整体来进行考虑和处理
  - 一个数据元素可由若干个**数据项(Data Item)**组成。  
数据项是数据的不可分割的最小单位
- 例子：学生的学籍信息：
  - 学号、姓名、性别、出生日期、入学成绩

数据项

数据元素

# 基本概念-2

- **数据结构(Data Structure)**是指相互之间存在一定联系(**关系**)的数据元素的集合
- **数据元素之间的相互关系**称为(**逻辑**)结构,通常有下列4种基本结构:
  - **集合**: 结构中的数据元素除了“同属于一个集合”外,没有其它关系
  - **线性结构**: 结构中的数据元素之间存在一对一的关系
  - **树型结构**: 结构中的数据元素之间存在一对多的关系
  - **图状结构或网状结构**: 结构中的数据元素之间存在多对多的关系

# 数据结构实例：身份证号查询系统

- 设有一个身份证号簿，它记录了N个人的名字和其相应的身份证号，假定按如下形式安排：(a1, b1), (a2, b2), ...(an, bn)，其中ai, bi (i=1, 2...n) 分别表示某人的名字和身份证号
- 数据元素与数据元素构成简单的一对一的线性关系

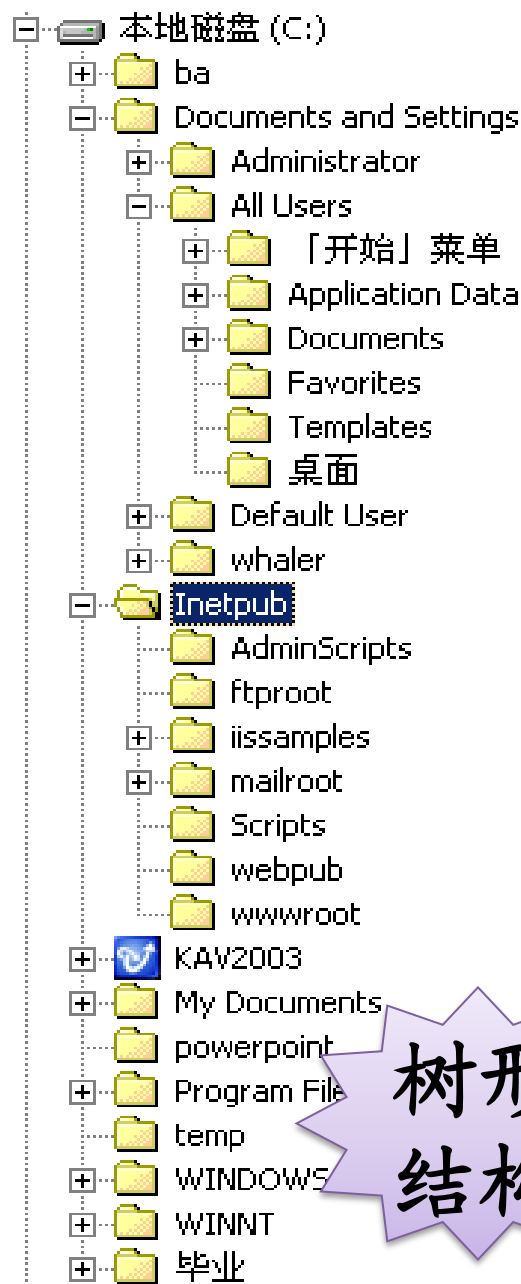
要求设计一个算法，当给定任何一个人的名字时，该算法能够打印出此人的身份证号，如果该身份证号簿中根本就没有这个人，则该算法能够报告没有这个人

姓名	身份证号
张三	530108 199001211220
李四	450102 202001211220
...	...

线性表  
结构

# 数据结构实例：磁盘目录文件系统

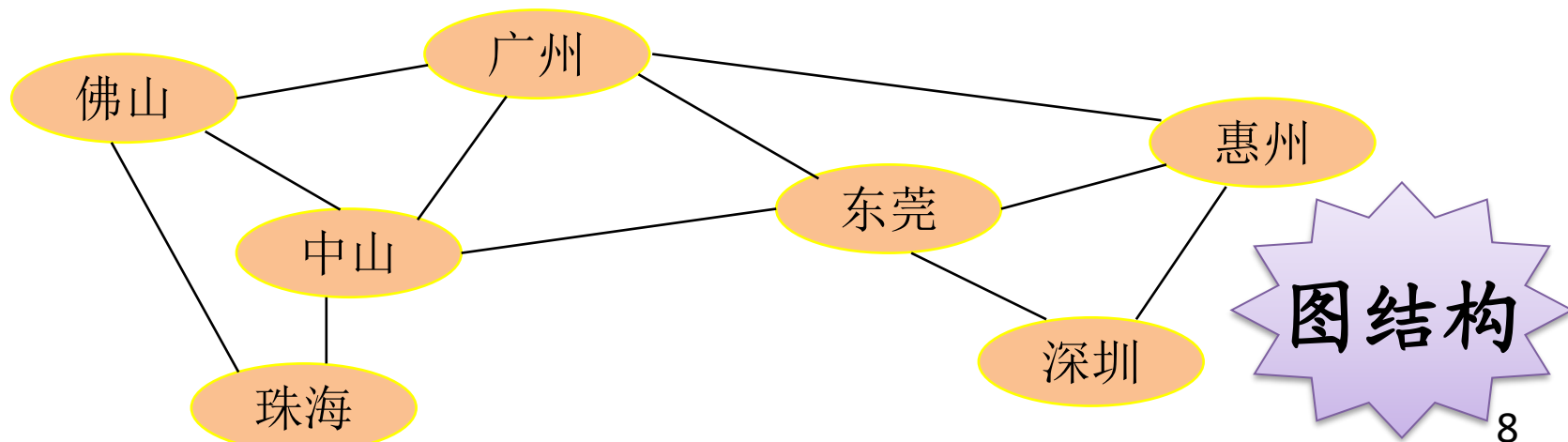
- 磁盘根目录下有很多子目录及文件，每个子目录里又可以包含多个子目录及文件，但每个子目录只有一个父目录，依此类推
- 数据元素与数据元素构成一对多的关系，是一种典型的非线性关系结构——树形结构



树形  
结构

# 数据结构实例：交通网络图

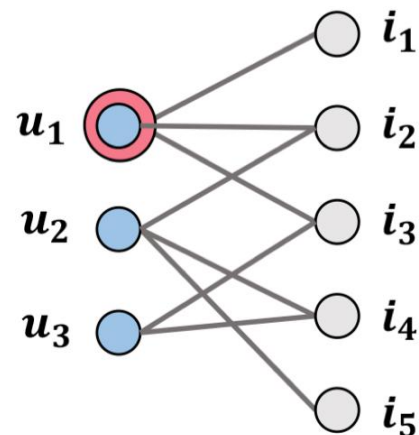
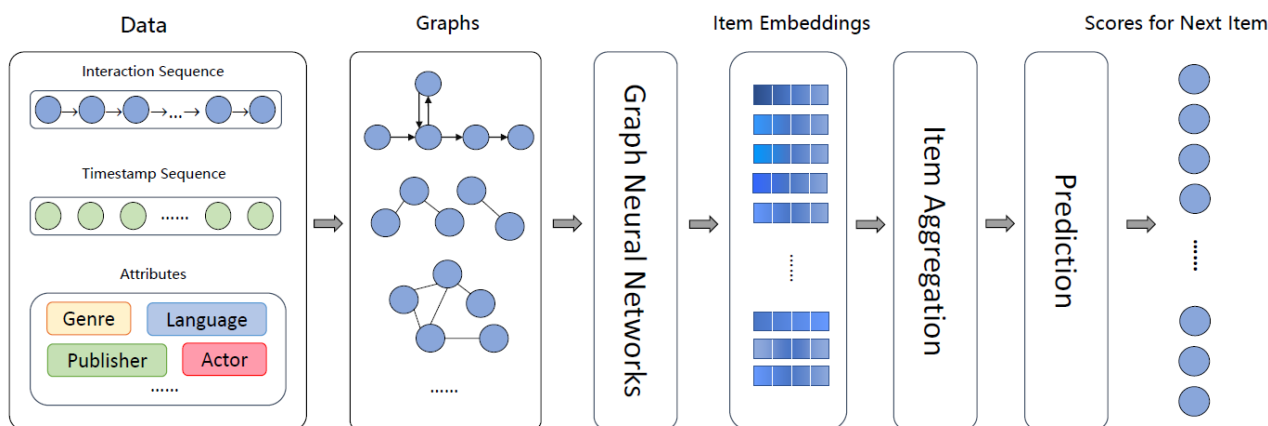
- 城市之间的交通状况：典型的网状结构问题，数据元素与数据元素构成多对多的关系，是一种非线性关系结构
- 寻找城市之间的**最短路径**：从一个地方到另外一个地方可以有 multiple 路径
- 修建能到达所有城市的高速公路：在图上构造**最小生成树**





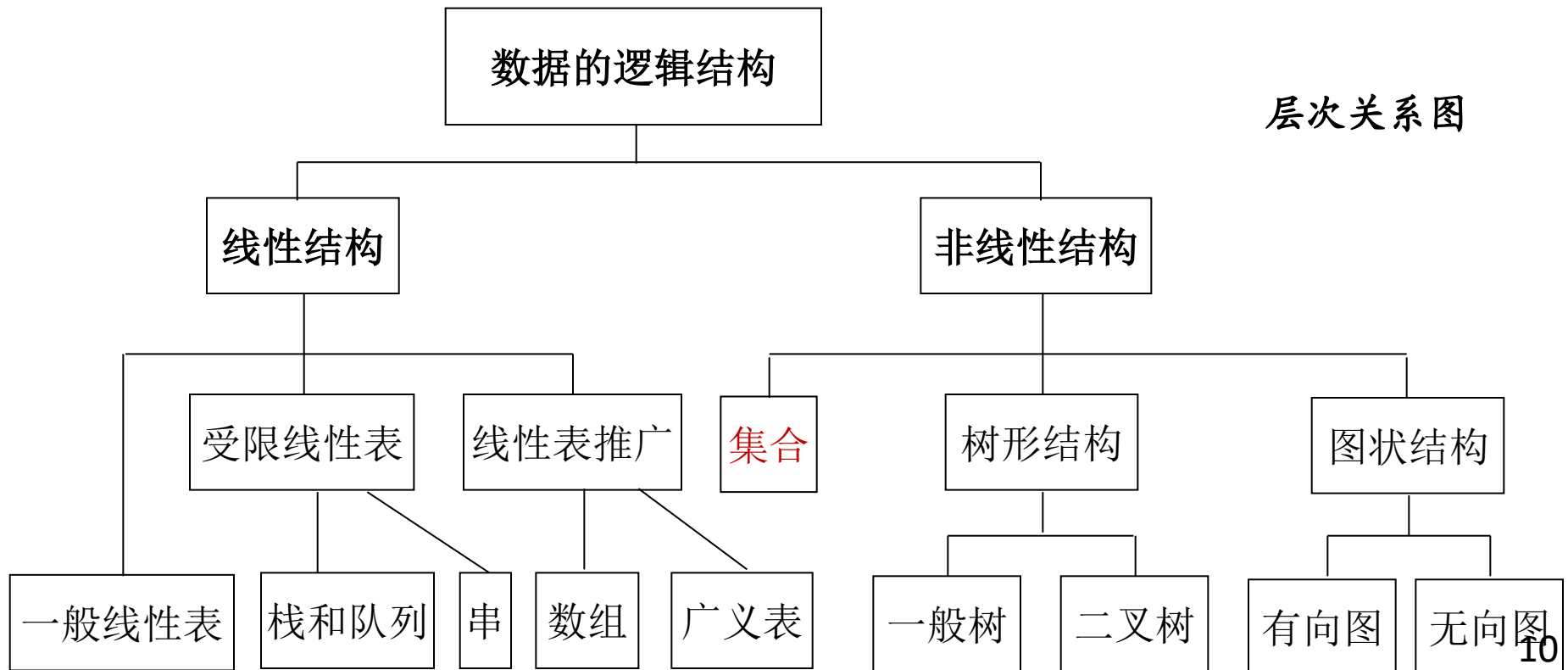
# 数据结构实例：用户-物品交互数据

- 用户-物品交互记录可以看作物品集合
- 用户-物品交互记录可以看作物品序列
- 用户-物品交互记录可以看作
  - 同构的物品图
  - 异构的用户-物品二分图(bipartite graph, 二部图)



# 数据结构的定义

- 数据结构包括“逻辑结构”和“物理结构”两个方面
  - 数据的逻辑结构给出了数据元素之间的逻辑关系的描述
  - 数据的物理结构(即存储方式)是其逻辑结构在计算机中的表示和实现

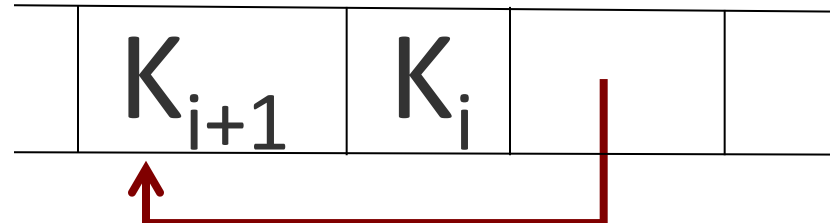
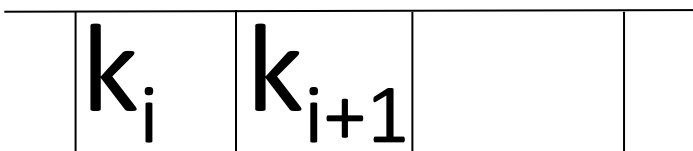


# 数据**逻辑结构**的存储方式(存储结构)

- 数据结构在计算机内存中的存储包括数据元素的存储和元素之间的关系的表示
- **数据元素之间的关系**在计算机中有两种不同的表示方法：顺序表示和非顺序表示
  - **顺序表示**：用数据元素在存储器中的**相对位置**来表示数据元素之间的逻辑关系
    - 在C语言中，用**一维数组**表示顺序存储结构
  - **非顺序表示**：用指示数据元素存储地址的**指针(pointer)**来表示数据元素之间的逻辑关系
    - 在C语言中，用**指针**表示链式存储结构
- 由此得出两种不同的存储结构：**顺序存储结构**和**链式存储结构**

# 例1 数据集合的存储 sizeof

- 设有数据集合  $A=\{3.0, 2.3, 5.0, -8.5, 11.0\}$ ，其数据元素可以有两种不同的存储结构：
  - 顺序结构：数据元素存放的地址是连续的或差恒定常量，令  $K_{i+1}$  的存储位置和  $K_i$  的存储位置之间差一个常量  $C$ ，而  $C$  是一个隐含值，那么，**整个存储结构中只含数据元素本身的信息**
  - 链式结构：数据元素存放的地址是否连续没有要求，需要用一個和  $K_i$  在一起的**附加信息**指示  $K_{i+1}$  的存储位置



## 例2 数据元素的表示

- 定义学生为:

```
typedef struct {
```

```
    char id[8];           // 学号
```

```
    char name[16];       // 姓名
```

```
    char gender;         // M/F
```

```
    DateType date_of_birth; // 出生日期
```

```
} Student;              // 学生类型
```

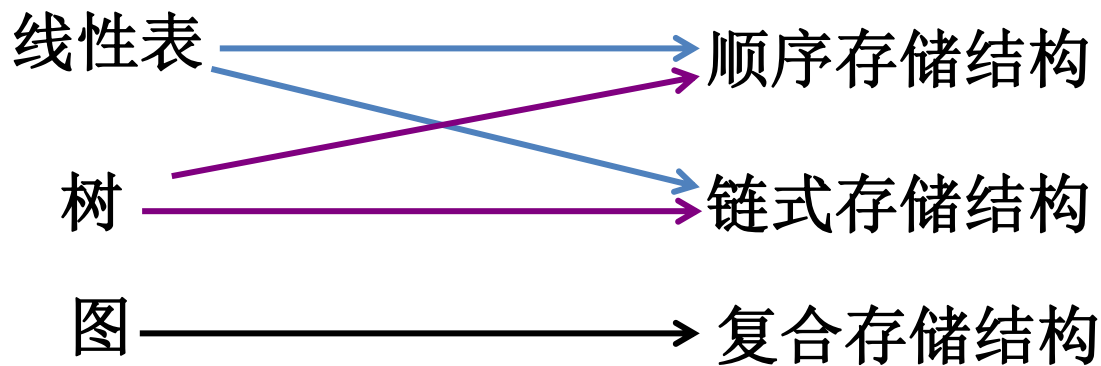
- 以三个带有次序关系的整数表示一个长整数时，可利用 C 语言中提供的整数数组类型，定义长整数为：

```
typedef int Long_int[3]
```

```
typedef struct {  
    int y;    // Year  
    int m;    // Month  
    int d;    // Day  
} DateType;  // 日期类型
```

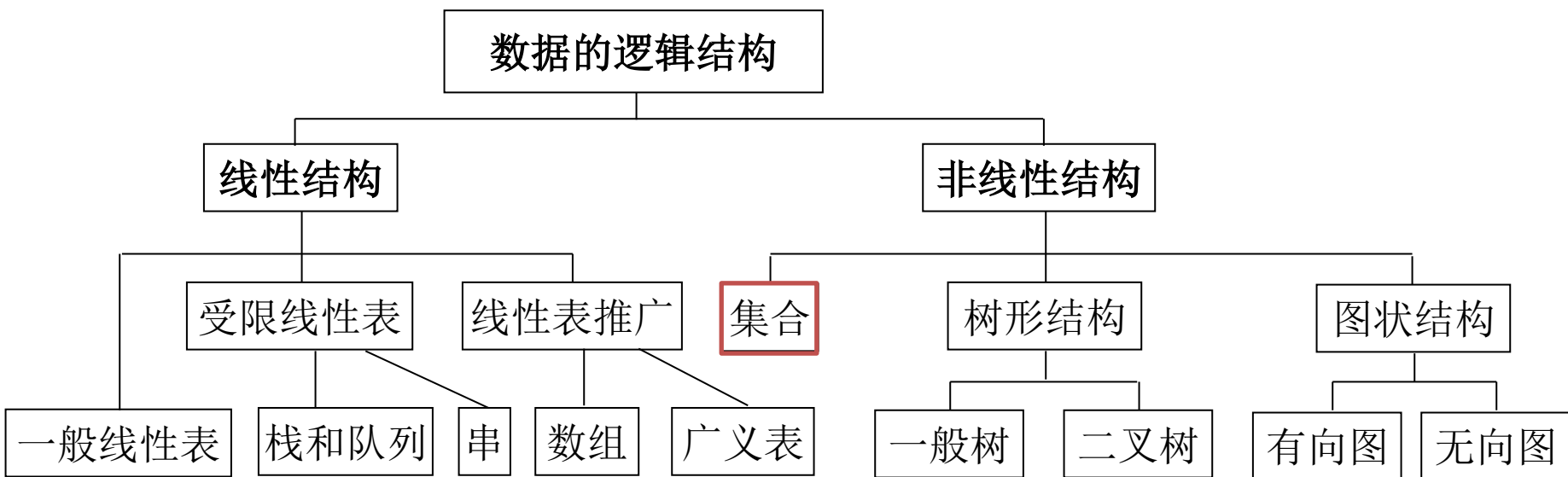
# 逻辑结构

# 物理结构





逻辑结构与所采用的存储结构

数据的逻辑结构和物理结构是密不可分的两个方面：一个**算法的设计**取决于所选定的**逻辑结构**，而**算法的实现**依赖于所采用的**存储结构**



数据的逻辑结构层次关系图

# 3. 算法

- **算法(Algorithm)** 是对特定问题求解步骤的一种描述，是指令的有限序列，其中每一条指令表示一个或多个操作
- 算法具有以下五个特性：
  - **有穷性**：一个算法必须总是在执行有穷步之后结束，且每一步都可在有穷时间内完成
    - 一个算法不收敛，是否违反了“有穷性”的要求？
  - **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。在任何条件下，只有唯一的一条执行路径。
    - 对于相同的输入，只能得到相同的输出 
  - **可行性**：一个算法是能行的，即算法描述的操作都可以通过已经实现的基本运算执行有限次来实现
  - **输入**：一个算法有**零个**或多个输入，这些输入取自于某个特定的对象集合
  - **输出**：一个算法有**一个**或多个输出，这些输出是同输入有着某些特定关系的量

# 算法

- 一个算法可以用多种方法描述，主要有：
  - 使用自然语言(包括中文和英文)描述
  - 使用形式语言(包括伪码)描述
  - 使用程序设计语言描述
    - 语言是思想的载体
  - 流程图：表示算法的一种方法
- 算法和程序是两个不同的概念
  - 一个计算机程序是对一个算法使用某种程序设计语言的具体实现
  - 算法必须可终止 意味着 不是所有的计算机程序都是算法



# 算法设计

- 用计算机解决实际问题的一般过程
  - 描述和分析问题: Problem Formulation /Modeling
  - 设计一个解决方案/算法去解决问题
  - 实现: 用计算机语言编程并调试
    - 计算Fibonacci 数列
    - 排序, 取Top-k个: 小规模, 大规模, 流
    - 选举(分布式选leader): 常规算法 vs. 无线动态环境

计算Fibonacci 数列： 1,1,2,3,5,8,13,21,34,55, ...

- $F[1]=1, F[2]=1, F[n]=F[n-1]+F[n-2] (n \geq 3)$ 
  - 从第3项开始，每一项都等于前两项之和

- 用递归实现

- 用递推实现:  $f0=0; f1=1;$

```
for(i=2; i<=n; i++){  
    fk=f0+f1;  
    f0=f1;f1=fk; }
```

- 用矩阵乘法实现

- 把  $[a_n, a_{n+1}]^T$  看成一个向量，把形成 Fibonacci 数列的递归公式  $a_{n+2} = a_n + a_{n+1}$  变成了如下所示的线性模型

$$\begin{bmatrix} a_{n+1} \\ a_{n+2} \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix} \cdot \begin{bmatrix} a_n \\ a_{n+1} \end{bmatrix}$$

- 于是有:  $\begin{bmatrix} a_{n+1} \\ a_{n+2} \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}^{n-1} \cdot \begin{bmatrix} a_1 \\ a_2 \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}^{n-1} \cdot \begin{bmatrix} 1 \\ 1 \end{bmatrix}_{29}$

# 算法设计

- 在用计算机解决实际问题的过程中，要考虑：
  - 问题所涉及的数据之间的关系及数据量的大小
  - 如何在计算机中存储数据及体现数据之间的关系
  - 处理问题时需要对数据作何种运算
  - 所编写的程序的性能是否良好

# 算法设计

- 基本的思路
  - **暴力法(Brute force method)**
    - 枚举/穷举(Enumeration): 图的遍历(DFS, BFS)
    - 回溯(Backtracking): 迷宫寻路, 八皇后
  - **分而治之法(Divide and conquer):** 分解成若干独立的子问题
    - 递归(分解后的所有子问题是一个小规模的原问题): Fibonacci 数列
    - 对广义表的操作, 快速排序, 归并排序
  - **动态规划法(Dynamic programming algorithm):** 分解成若干子问题, 但子问题之间不相互独立
    - Fibonacci数列, 背包问题
  - **贪心法(Greedy algorithm):** 根据当前已有信息做出最优选择:
    - 背包问题, 装箱(Bin packing)问题
    - Huffman编码, (图)最短路径, (图)最小生成树

# 算法的评价标准-1

- 评价一个好的算法有以下几个标准：
  - **正确性(Correctness)**: 算法应满足具体问题的需求; 对算法是否“正确”的理解可以有以下四个层次
    1. 程序中不含语法错误
    2. 程序对于几组输入数据能够得出满足要求的结果
    - 3. 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**
    4. 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 3 层意义的正确性作为衡量一个算法是否合格的标准

# 算法的评价标准-2

- **可读性(Readability)**: 算法应容易供人阅读和交流。可读性好的算法有助于对算法的理解和修改
- **健壮性(Robustness)**: 算法应具有容错处理。当输入非法或错误的数据时, 算法应能适当地作出反应或进行处理, 而不会产生莫名其妙的输出
  - 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
- **高效率(High efficiency)与低存储量(Low memory space)**需求: 效率指的是算法执行的时间; 存储量需求指算法执行过程中所需要的最大存储空间
  - 一般地, 这两者与问题的规模有关

# 算法效率的度量-1

- **算法的执行时间**需通过依据该算法编制的程序在计算机上运行所消耗的时间来度量
- 程序运行所消耗的时间与下列因素有关：
  - 问题的规模
  - 依据算法选用何种策略：决定算法中有哪些操作，这些操作执行了多少次
  - 程序设计的语言
  - 编译程序所产生的机器代码的质量
  - 机器执行指令的速度

# 算法效率的度量-2

- 方法通常有两种：

## – 事前分析法：

- 求出该算法的一个时间界限函数：撇开与软硬件等有关的因素，可以认为：算法的时间效率即一个特定算法“运行工作量”的大小只依赖于问题的规模(通常用 $n$ 表示)，或者说，它是问题规模的函数
- 工程上：Back-of-the-envelope calculation

## – 事后统计法：计算机内部进行执行时间和实际占用空间的统计

- 问题：必须先运行依据算法编制的程序；依赖软硬件环境，容易掩盖算法本身的优劣



# 算法的时间复杂度

- 算法的渐进分析(Asymptotic analysis): 关注算法的时间复杂度的总体变化趋势和增长速度
- 算法的渐近时间复杂度(Asymptotic Time Complexity, 简称时间复杂度):  $T(n)$
- 大O记号(Big-O Notation): 用于估计随输入增加, 算法所用的操作的次数
  - 定义: 若 $f(n)$ 是正整数 $n$ 的一个函数, 若 $\exists n_0, M \geq 0$ , 使得当 $n \geq n_0$ 时,  $|T(n)| \leq M|f(n)|$ , 那么 $T(n) = O(f(n))$
  - $f(n)$ : 在满足上述定义的一组函数中尽可能小的那个函数
  - 随着问题规模 $n$ 的增加, 算法执行时间的增长率和 $n$ 的某个函数 $f(n)$ 的增长率相同, 那么我们说该算法的时间量度 $T(n) = O(f(n))$

# 算法的时间复杂度

- 解决某个问题的算法的执行时间  $\sim$  与问题的规模  $n$  有关  $\sim$  以问题的规模  $n$  为变量的函数  $f(n)$ ,  $g(n)$ 
  - 问题规模  $n$ , 算法的执行时间  $T(n)$ , 分析  $T(n)$  的渐进上界
    - $T(n)=O(f(n))$ :  $n$  足够大之后,  $f(n)$  给出了  $T(n)$  的一个渐进上界
      - 对规模为  $n$  的输入, 算法的执行时长存在一个增长的上限
    - $T(n)=\Omega(g(n))$ :  $n$  足够大之后,  $g(n)$  给出了  $T(n)$  的一个渐进下界
      - 对规模为  $n$  的输入, 算法的时间不低于  $\Omega(g(n))$

# 时间复杂度-定理

- 对两个程序段，其时间复杂度分别为 $T_1(n)$ ,  $T_2(m)$

- 顺序执行上述两程序段，有：

$$T(n, m) = T_1(n) + T_2(m) = O(\max(f_1(n), f_2(m)))$$

- 将上述两程序段嵌套调用，有：

$$T(n, m) = T_1(n) \times T_2(m) = O(f_1(n) \times f_2(m))$$

- 定理：若 $T(n) = a_m \times n^m + a_{m-1} \times n^{m-1} + \dots + a_1 \times n + a_0$ 是一个 $m$ 次多项式，则 $T(n) = O(n^m)$

— 常系数可忽略：  $O(f(n)) = O(c \times f(n))$

— 低次项可忽略： 若 $a > b > 0$ ，那么 $O(n^a + n^b) = O(n^a)$

# 表示时间复杂度的阶

- 函数有如下的增长关系：

$$c < \log_2 n < n < n \log_2 n < n^2 < n^3 < 2^n < 3^n < n! < n^n$$

- 阶之间有如下的关系：

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

对数换底公式

$$\log_a b = \frac{\log_c b}{\log_c a}$$

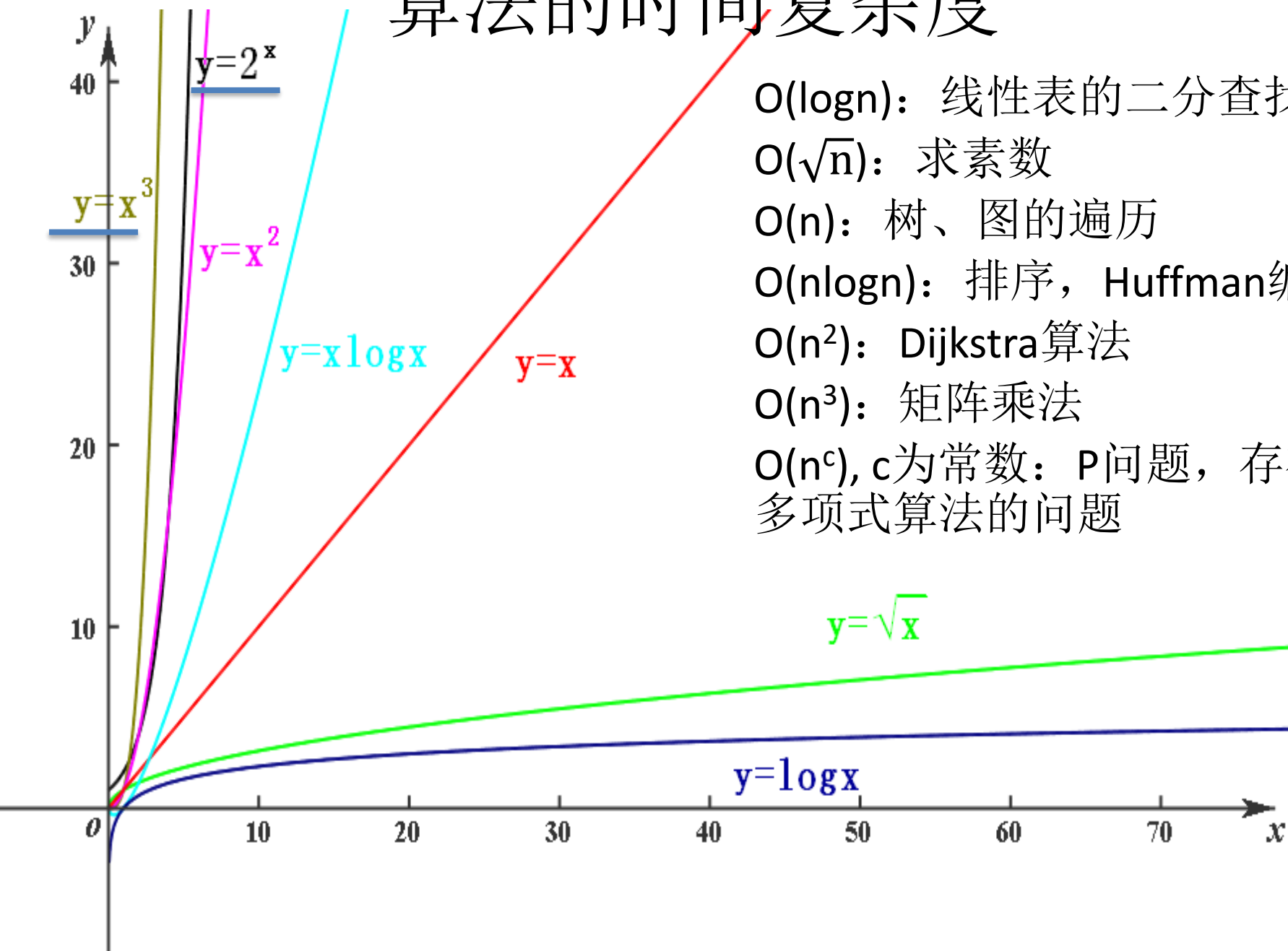
$O(1)$ : 常量阶       $O(\log n)$ : 对数阶

$O(n)$ : 线性阶       $O(n \log n)$ : 线性对数阶

$O(n^2)$ : 平方阶       $O(n^k)$ :  $k$ 次方阶( $k \geq 2$ )

$O(2^n)$ : 指数阶       $O(n!)$ : 阶乘阶

# 算法的时间复杂度



$O(\log n)$ : 线性表的二分查找

$O(\sqrt{n})$ : 求素数

$O(n)$ : 树、图的遍历

$O(n \log n)$ : 排序, Huffman编码

$O(n^2)$ : Dijkstra算法

$O(n^3)$ : 矩阵乘法

$O(n^c)$ ,  $c$ 为常数: P问题, 存在多项式算法的问题

# 如何分析算法复杂度-估算

- 算法由控制结构和**原操作(指固有数据类型的操作)**构成，算法时间取决于两者的综合效果
  - **算法的执行时间**与该**原操作执行次数之和**成正比
- 从算法中**选取**一种**对于所研究的问题来说**  
**是基本操作的原操作**，以**该基本操作**在算  
法中**重复执行的次数**作为算法运行时间的  
衡量准则

# 算法时间复杂度分析举例

例：两个**n阶方阵**的乘法

//以二维数组存储矩阵元素，c 为a 和b 的乘积

```
for(i=1; i<=n; ++i) ..... 2(n+1)
    for(j=1; j<=n; ++j) ..... 2n(n+1)
        { c[i][j]=0 ; ..... n2
            for(k=1; k<=n; ++k) ..... 2n2(n+1)
                c[i][j]+=a[i][k]*b[k][j] ; ..... n3
        }
```

所有语句的执行频度之和( $3n^3+5n^2+4n+2$ )是矩阵阶数 $n$ 的函数，  
则该算法的时间复杂度为  $T(n) = O(3n^3+5n^2+4n+2) = O(n^3)$

一般地，常用最深层循环内的语句中的原操作的执行频度  
(重复执行的次数)来表示

# 算法时间复杂度分析举例

例： `{++x; s+=x;}`

- 将`x`自增看成是基本操作，将`s+=x`也看成是基本操作，则整体语句频度为2，时间复杂度为 $O(1)$ ，即常量阶

例： `for(i=1; i<=n; ++i) { ++x; s+=x; }`

- 语句频度为 $2n$ ，其时间复杂度为 $O(n)$ ，即线性阶

例： `for(i=1; i<=n; ++i)`

`for(j=1; j<=n; ++j)`

`{ ++x; s+=x; }`

- 语句频度为 $2n^2$ ，其时间复杂度为 $O(n^2)$ ，即平方阶



# 算法时间复杂度分析举例

例：for(i=2;i<=n;++i)

for(j=1;j<=i-1;++j)

{++x; a[i][j]=x;}

- 语句频度为  $1+2+3+\dots+n-1=(1+n-1) \times (n-1)/2$   
 $= (n^2-n)/2$
- 上述程序的时间复杂度为  $O(n^2)$ ，即平方阶

例：for (i=1; i<=n; i\*=2)

for(j=1; j<=n; j++)

count++;

- 上述程序的时间复杂度为  $O(n \log_2 n)$ ，即线性对数阶

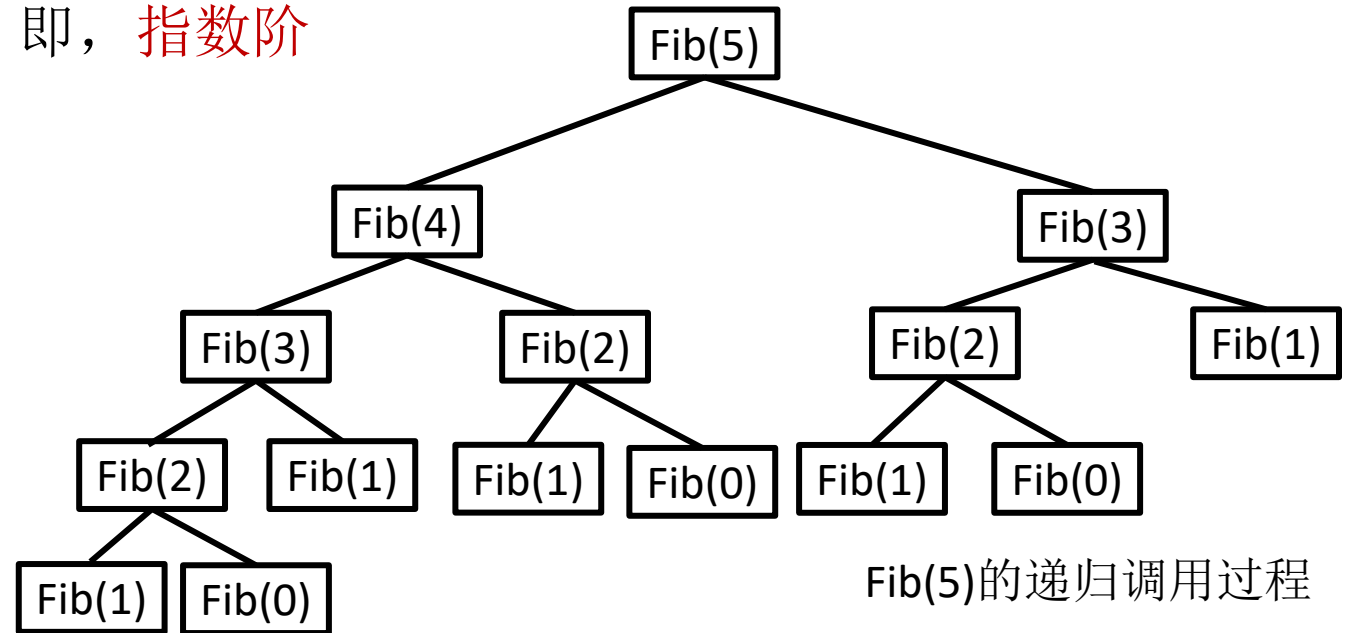
# 算法时间复杂度分析举例

- 例：计算Fibonacci 数列Fib(n)

$$\text{Fib}(n) = \text{Fib}(n-1) + \text{Fib}(n-2)$$

– 递归实现：

- 递归算法的时间复杂度  $\propto$  二叉树的结点个数
- 高度为n-1的满二叉树，其结点数为 $2^{n-1} - 1$
- $O(2^n)$ ，即，**指数阶**



– 递推实现：  $O(n)$

# 关于递归程序时间复杂度的Master定理

**定理：** 对于一个规模为 $n$ 的问题，通过分治将每个问题分解成 $a$ 个规模为 $\frac{n}{b}$ 的子问题，每次递归带来的额外运算为 $f(n^d)$ ，那么可得到它的时间复杂度关系式：

$$T(n) = a T\left(\frac{n}{b}\right) + f(n^d)$$

记 $\log(b,a)=\log_b a$ ，那么：

若 $\log(b,a) < d$ ，则 $T(n) = O(n^d)$

若 $\log(b,a) > d$ ，则 $T(n) = O(n^{\log(b,a)})$

若 $\log(b,a) = d$ ，则 $T(n) = O(n^d \log_b n)$

# 算法时间复杂度分析举例

例：for(i=1; i<n; i=1<<i) {x++;}

- 设 x++ 的执行次数为 y，那么
  - y 最多是 5 次，这时  $i = 2^{65536}$ ，时间复杂度为常数阶

i	i<n	语句执行次数y
i=1	满足条件	执行1次，y=1
i=1<<1=2	满足条件	执行1次，y=2
i=1<<2=2 <sup>2</sup> =4=2 <sup>2</sup>	满足条件	执行1次，y=3
i=1<<4=2 <sup>4</sup> =16=2 <sup>2<sup>2</sup></sup>	满足条件	执行1次，y=4
i=1<<16=2 <sup>16</sup> =65536=2 <sup>2<sup>2<sup>2</sup></sup></sup>		执行1次，y=5
i=1<<65536=2 <sup>65536</sup>		

$$y = \min\{\log^{(x)} n \leq 1; x > 0\}, \quad f^{(x)}(n) = \begin{cases} f(f^{(x-1)}(n)), & \text{如果 } x > 0 \\ n, & \text{如果 } x = 0 \end{cases}$$

# 算法时间复杂度分析举例

- 算法中基本操作重复执行的次数随输入数据集不同而不同
  - 最坏情况(Worse-case)下的时间复杂度
  - 最好情况(Best-case)下的时间复杂度
  - 平均情况(Average-case)下的时间复杂度：将规模为n的所有输入对应的算法执行时间进行加权平衡
    - 权重的设置取决于实际场景
    - 通常可以认为输入符合某种概率分布，由概率定权重

# 算法时间复杂度分析举例：求素数

例：素数的判断算法

```
void prime(int n) /* n是一个正整数 */
{
    int i=2;
    while ( (n%i)!=0 && i*1.0 < sqrt(n) ) i++;
    if (i*1.0>sqrt(n) )
        printf("&d 是一个素数\n", n);
    else
        printf("&d 不是一个素数\n", n);
}
```

- 时间复杂度与输入数据相关
- 嵌套的最深层语句是i++; 其频度由条件( (n% i)!=0 && i\*1.0< sqrt(n) ) 决定，主要是 i\*1.0< sqrt(n)
- 最坏情况下的时间复杂度是 $O(n^{1/2})$

# 算法时间复杂度分析举例：搜索

例：在 $a[0], \dots, a[n-1]$ 中搜索 $x$

```
int seqsearch ( int a[ ], int n, int x ) {
```

```
    int i = 0;
```

```
    while ( i < n && a[i] != x )
```

```
        i++;
```

```
    if ( i == n ) return -1;
```

```
    return i; }
```

- 最好情况： $O(1)$ 次；
- 最坏情况： $O(n)$ 次；
- 查找成功且输入符合均匀分布时的平均时间复杂度为： $O((1+2+3+\dots+n)/n) = O((n+1)/2) = O(n)$

# 算法时间复杂度分析举例：冒泡排序

- 依次比较每一对相邻元素，如有逆序，则交换之
- 经一轮扫描交换后，最大元素必然就位
- 经一轮扫描交换后，问题的规模缩减至 $n-1$
- 若整趟扫描都没有进行交换，则排序完成





# 算法时间复杂度分析举例：冒泡排序

```
void BubbleSort(int a[], int n) {
```

```
//将a中整数序列重新排列成自小到大的有序  
的整数序列
```

i 指示 比较的区间

```
for (i=n-1, change=1; i>=1 && change; --i)
```

//change: 前一趟没有任何交换时程序退出

```
for (j=0, change=0; j<i; ++j)
```

j 指向 进行比较的第一个元素

```
if (a[j]>a[j+1]) {
```

//两元素逆序时进行交换

```
y=a[j];
```

```
a[j]=a[j+1];
```

```
a[j+1]=y;
```

```
change= 1;
```

```
}
```

```
}
```

基本操作：元素的比较操作

最好情况/正序： $O(n)$  次

最坏情况/逆序：

$n-1+n-2+ \dots +2+1=n(n-1)/2= O(n^2)$

平均时间复杂度取决于  
各种数据出现的概率

# 算法的空间分析

- **空间复杂度(Space complexity)** 是指算法编写成程序后，在计算机中运行时所需存储空间大小的度量
  - $S(n)=O(f(n))$ ，其中：  $n$ 为问题的规模(或大小)
- 该存储空间一般包括三个方面：
  - 程序本身所占用的存储空间
  - 输入数据所占用的存储空间
  - 辅助(存储)空间
- 一般地，算法的空间复杂度指的是**辅助空间**
  - 一维数组 $a[n]$ ： 空间复杂度  $O(n)$
  - 二维数组 $a[n][m]$ ： 空间复杂度  $O(n*m)$
- 若所需**辅助空间**相对于输入数据量来说是**常数**，即，所占有的临时空间不随问题的规模而改变，则称此算法为**原地(in-place)算法**，其空间复杂度为 $O(1)$
- 若所需存储量依赖于特定的输入，则通常**按最坏情况考虑**

# 算法时空复杂度分析举例

- 将一维数组中的元素倒置存放

```
ReverseArray(int a[],int n){ int i,j,*b;  
    b= (int *)malloc(sizeof(int)*n);  
    for(i=0,j=n-1;i<n;i++,j--) b[j]=a[i];  
    for(i=j=0;i<n;i++,j++) a[i]=b[i];  
    free(b);  
}
```

基本语句是循环内的进行数据交换的赋值语句，共执行了 $2n$ 次， $T(n) = 2n = O(n)$

辅助空间是一个与问题规模同量级的一维数组空间，再加上两个控制变量 $i$ ， $j$ ，共 $n+2$ 个，故

$S(n) = n+2 = O(n)$

# 算法时空复杂度分析举例

- 将一维数组中的元素倒置存放

```
ReverseArray(int a[],int n){  
    int i,j,t;  
    for(i=0,j=n-1;i<j;i++,j--) {  
        t=a[i];  
        a[i]=a[j];  
        a[j]=t;}  
}
```

基本语句是循环内的交换语句，共执行了 $n/2$ 次， $T(n) = n/2 = O(n)$

辅助空间是3个临时变量， $S(n) = 3 = O(1)$

# 算法时空复杂度分析举例

- 将一维数组中的元素倒置存放

```
ReverseArray(int a[],int n){  
    int i,t;  
    For(i=0;i<n/2;i++) {  
        t= a[i];a[i]=a[n-i-1];a[n-i-1]=t;}  
}
```

基本语句是循环内的交换语句，共执行了 $n/2$ 次， $T(n) = n/2 = O(n)$

辅助空间是2个临时变量， $S(n) = 2 = O(1)$

# 递归程序的空间开销

- 递归程序的空间开销：不可忽视
- 递归工作栈：整个递归调用过程期间使用的数据存储区
  - 递归工作记录：每一层递归所需的信息(包括：实参、局部变量和上一层的返回地址)合成一个记录
    - 每进入一层递归，就产生一个新的工作记录压入栈顶；每退出一层递归，就从栈顶弹出一个工作记录
  - 当前活动记录：栈顶记录，指示当前层的执行情况
  - 当前环境指针：递归工作栈的栈顶指针

# 总结

- 熟悉数据结构的基本概念
- 理解算法基本特征的含义
- 熟悉算法分析的基本概念，掌握计算语句频度、估算算法时间复杂度和空间复杂度的方法
  - 算法的时空复杂度刻画了算法的效率，是算法描述的重要组成部分
  - 算法的两面：功能(**Effective**)和性能(**Efficient**)
    - 只有在算法功能正确的条件下，讨论算法的性能才有意义
      - Car Agent 快速地做出一个错误的动作，造成撞车
    - 不同场景下对算法性能的评价是不一样的
      - 机器学习模型分离线训练和在线推理两个阶段，对不同阶段的时空开销的容忍性是不一样的：以Go Agent为例
      - 早期 AlphaGo 要花几个月训练，达到职业围棋选手的水平
      - 2017年，AlphaGo Zero训练40天，战胜之前所有版本 AlphaGo，包括AlphaGo Master(2017击败了世界冠军柯洁)