第一章 绪论

1.1 什么是数据结构

1.2 基本概念和术语

1.3 抽象数据类型的表现和实现

1.4 算法和算法分析

- 目前，计算机已深入到社会生活的各个领域，其应用已不再仅仅局限于科学计算，而更多的是用于控制、管理及数据处理等非数值计算领域。
- 随着应用问题的不断复杂，导致信息量剧增与信息范围的拓宽，使许多系统程序和应用程序的规模巨大，结构复杂。因此，必须分析待处理问题中的对象的特征及各对象之间存在的关系，这涉及到两个问题：
  - 信息的表示和组织
  - 信息的处理

  信息的表示和组织又直接关系到处理信息的程序的效率

[例1.1] 该如何摆放，才能让读者很方便地找到他所要找的书？

【分析】

[方法1] 随便放——任何时候有新书进来，哪里有空就把书插到哪里。

放书方便，但查找效率极低！

[方法2] 按照书名的拼音字母顺序排放。

查找方便，但插入新书很困难！

[方法3] 把书架划分成几块区域，每块区域指定摆放某种类别的图书；在每种类别内，按照书名的拼音字母顺序排放。

查找、插入方便，但每种类型的书不知道有多少本，有可能造成空间的浪费！
划分多少种类别？

[例1.2] 多项式的标准表达式可以写为：

$$f(x) = a_0 + a_1x + a_2x^2 + \ldots + a_nx^n$$

现给定一个多项式的阶数 n，并将全体系数存放在数组 a[ ] 里。请写程序计算这个多项式在给定点 x 处的值。

[方法2] 秦九韶法

$$f(x) = a_0 + x\,(a_1 + x\,(a_2 + x\,(a_3 + x\,(\ldots + x\,(a_n))))\ldots)$$

```
double f( int n, double a[], double x )
{   /* 计算阶数为n，系数为a[0]...a[n]的多项式在x点的值 */
        int i;
        double p = a[n];
        for (i=n; i>0; i--)
                p = a[i-1] + x*p;
        return p;
}
```

– 即使解决一个非常简单的问题，往往也有多种方法，且不同方法之间的效率可能相差甚远

  - 跟数据的组织方式有关（如例1.1）
  - 跟算法的策略有关（如例1.2）

1.1 数据结构

- 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等的学科。

“数据结构是计算机中组织、存储数据的方式。精心选择的数据结构可以带来最优效率的算法。”

1.2 基本概念和术语

一. 数据

- 是信息的载体，是描述客观事物的数、字符、以及所有能输入到计算机中，被计算机程序识别和处理的符号集合。
  - 数值性数据
  - 非数值性数据

1.2 基本概念和术语

二. 数据元素（Data Element）

- 数据的基本单位。在计算机程序中常作为一个整体进行考虑和处理。数据元素又称为元素、结点、记录
- 有时一个数据元素可以由若干数据项组成（此时数据元素被称为记录）

三. 数据项（Data Item）

- 具有独立含义的最小标识单位

学号	姓名	学院	专业

1.2 基本概念和术语

四、数据对象（Data Object）

- 具有相同性质的数据元素的集合，是数据的一个子集。

例：整数数据对象

N = {0, ±1, ±2, … }

例：字母字符数据对象

C={ ‘A’ , B’ , ‘C’ , … ‘F’ }

1.2 基本概念和术语

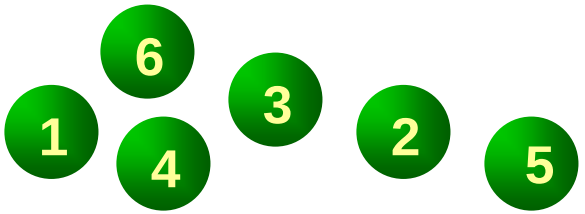
- 逻辑结构

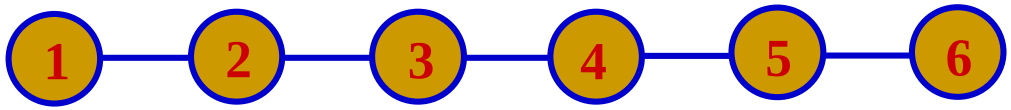
描述数据元素之间的逻辑关系（即关联方式或“邻接关系”，相互作用和依赖关系）

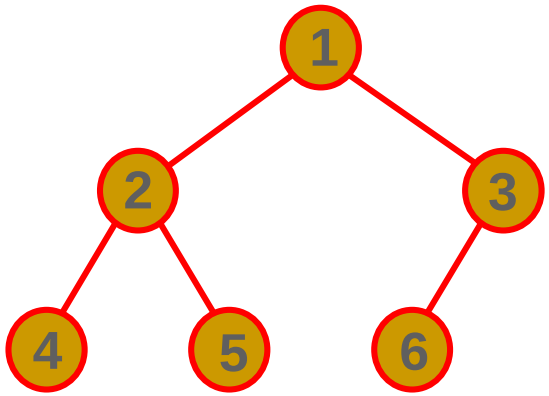
- 线性结构：线性表（表，栈，队列，串等）
- 非线性结构：树、图、集合

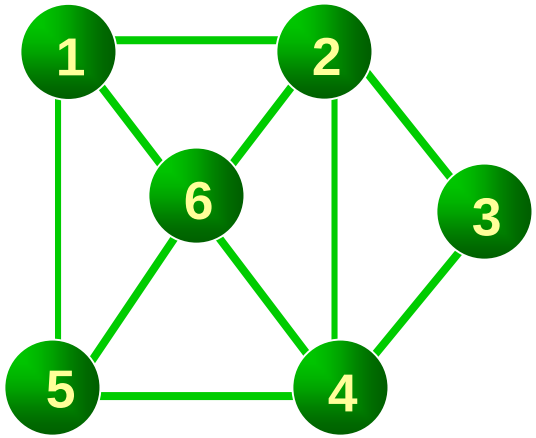
1.2 基本概念和术语

- 逻辑结构的四种基本形式
  - 集合
  - 线性结构
  - 树形结构
  - 图状结构









1.2 基本概念和术语

- 数据结构形式定义

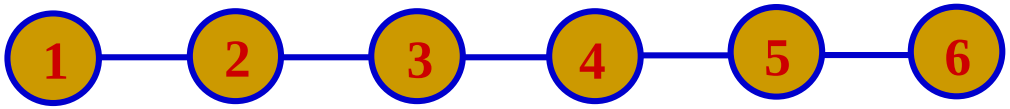
  数据结构是一个二元组，记为：

  Data_Structure = {D, S}

  其中，D是某一数据对象， S是该对象中所有数据成员之间的关系的有限集合。

1.2 基本概念和术语

- 线性数据结构举例



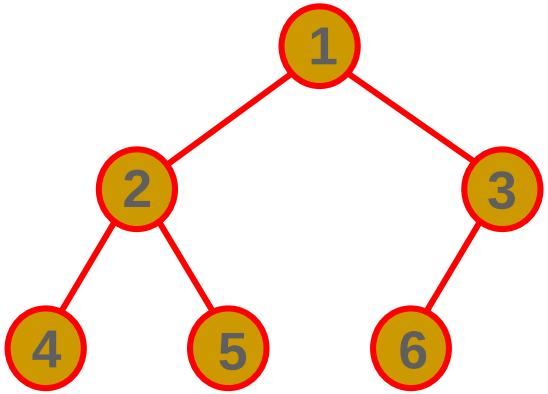
L = {K, R}

K = {1, 2, 3, 4, 5, 6}

R = {<1,2>, <2,3>, <3,4>, <4,5>, <5,6>}

1.2 基本概念和术语

- 树形数据结构举例



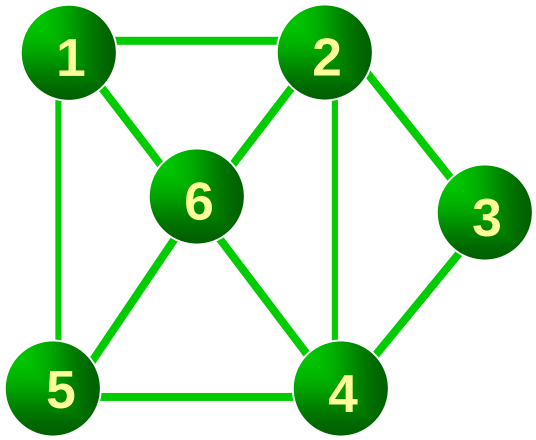
T = {K, R}

K = {1, 2, 3, 4, 5, 6}

R = {<1,2>, <1,3>, <2,4>, <2,5>, <3,6>}

1.2 基本概念和术语

- 图状结构举例



G = {K, R}

K = {1, 2, 3, 4, 5, 6}

R = {(1,2), (1,5), (1,6), (2,3), (2,4),
(2,6), (3,4), (4,5), (4,6), (5,6)}

1.2 基本概念和术语

数据结构应用举例

- 线性数据结构

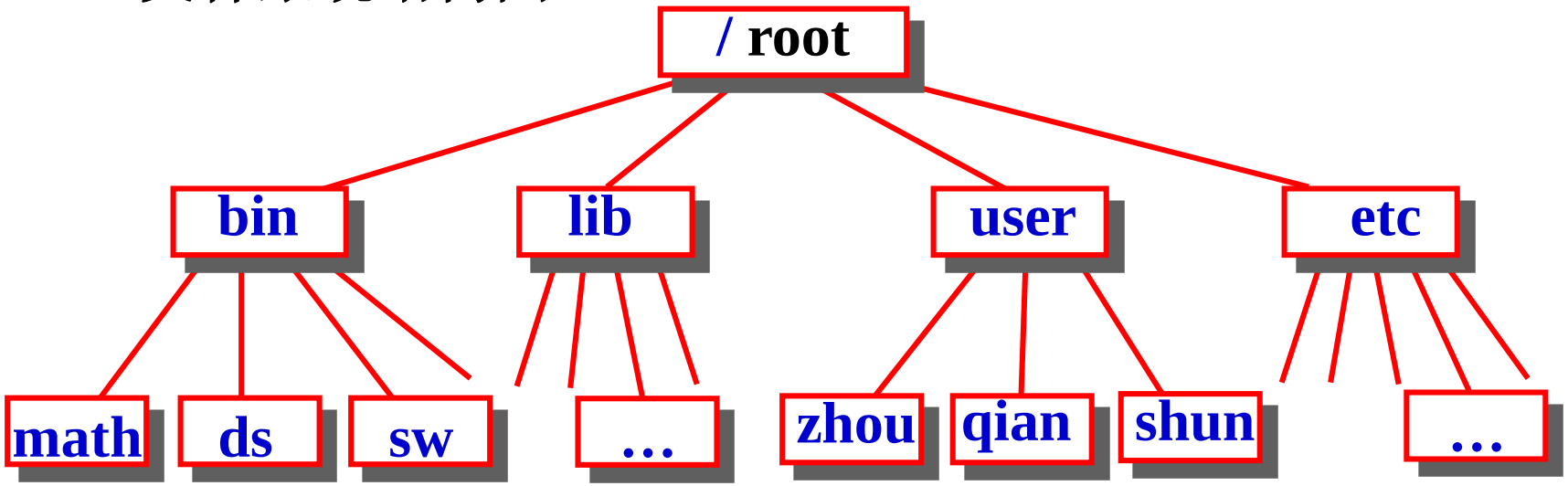
学生选课名单（部分）

姓名	学号	学院	专业
刘泰源	2006044060	计算机与软件学院	软件工程
温国乾	2006131006	计算机与软件学院	软件工程
强健	2006131043	计算机与软件学院	软件工程
杨海波	2006131047	计算机与软件学院	软件工程
李耀东	2006131051	计算机与软件学院	软件工程

1.2 基本概念和术语

- 树形数据结构

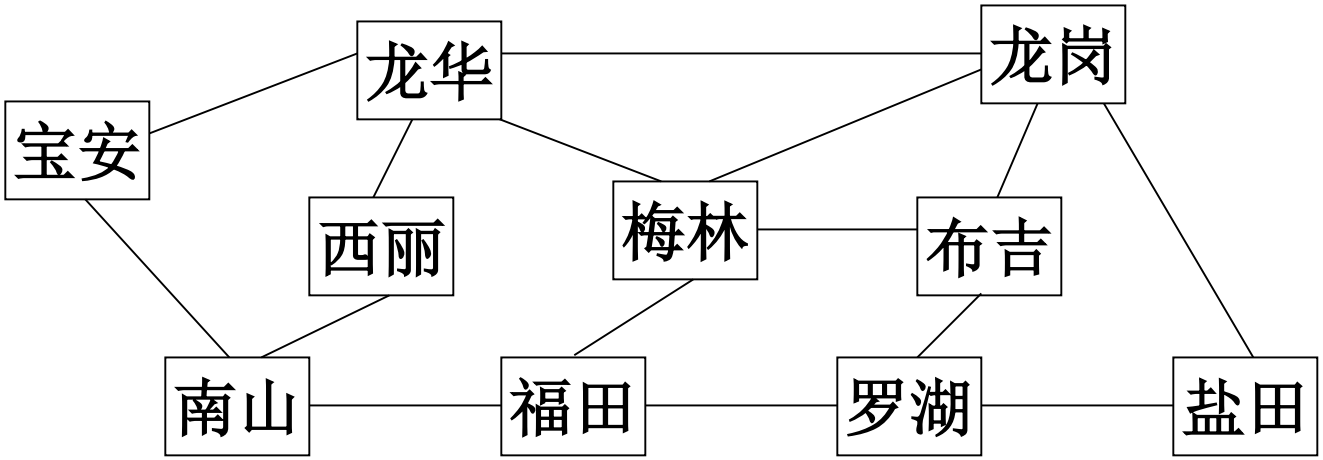
UNIX文件系统结构图



1.2 基本概念和术语

- 图状数据结构举例

深圳城市交通示意图



1.2 基本概念和术语

◆存储结构（物理结构）

是数据结构在计算机中的具体实现方式(又称映象)，包括数据的表示和关系的表示。

- 顺序存储表示(例如C语言中一维数组表示)
- 链式存储表示(例如C语言中的指针表示)

还有索引存储、散列存储等其它表示

1.2 基本概念和术语

◆顺序存储结构

逻辑上相邻的元素，存储空间也相邻。

- 所有元素占用一整块存储空间。
- 逻辑上相邻的元素，物理上也相邻。

1.2 基本概念和术语

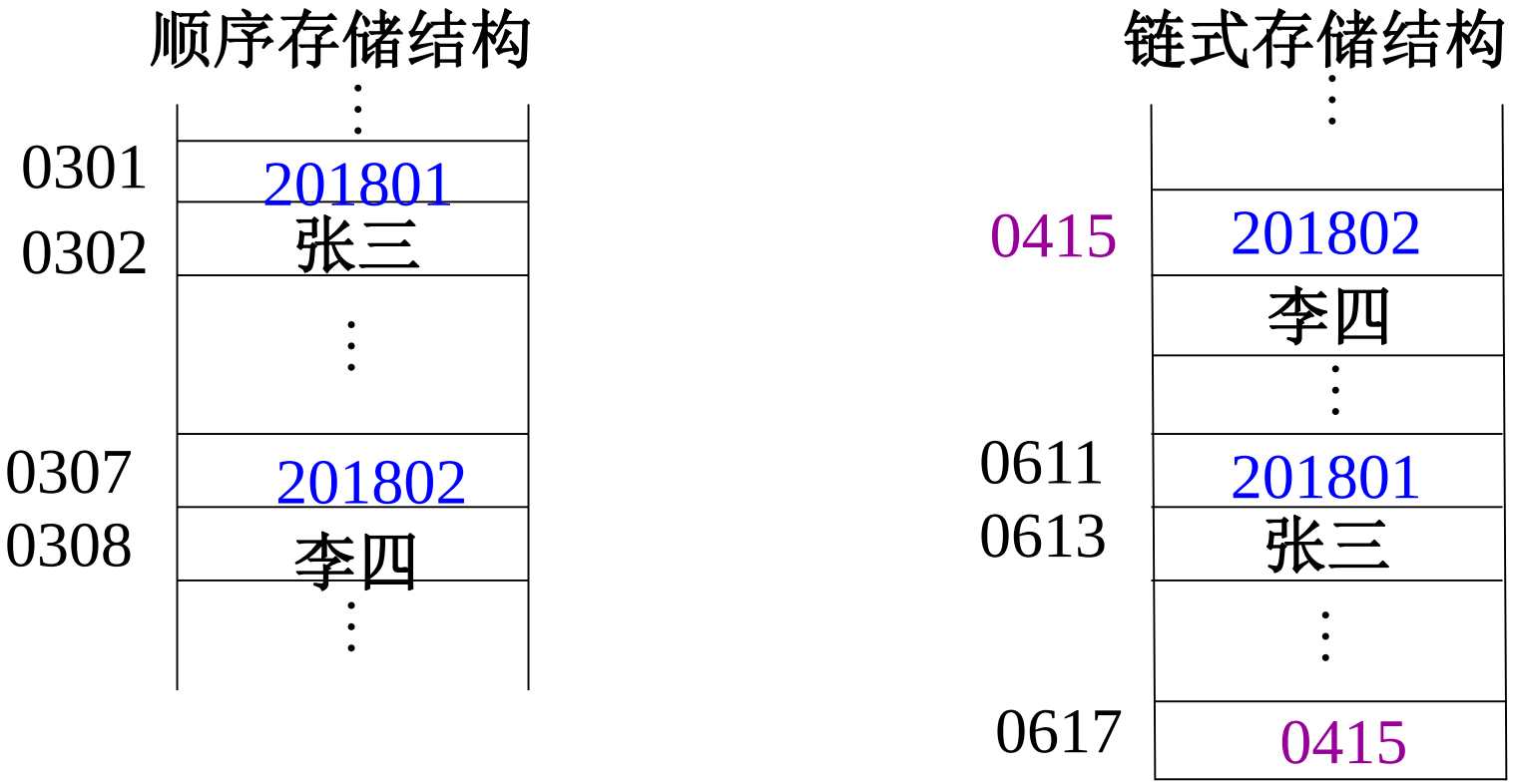
◆链式存储结构

逻辑上相邻的元素，存储空间不一定相邻。

- 一个逻辑元素用一个结点存储，每个结点单独分配，所有结点的存储地址不一定是连续的。
- 用指针来表示逻辑关系。

1.2 基本概念和术语

例：学生记录存储结构举例



1.2 基本概念和术语

◆逻辑结构与存储结构

- ❑ 同一种逻辑结构可以对应多种存储结构。
- ❑ 同样的运算，在不同的存储结构中，其实现过程是不同的。

※ 算法的设计取决于数据的逻辑结构，算法的实现依赖于采用的存储结构。

1.3 数据类型

一. 数据类型的概念

- 数据类型是一个值的集合和定义在这个值集上的一组操作的总称。

例如C语言中的整型数据(int)，其值集为某个区间上的整数，定义在其上的操作为+，－，×，/等

1.3 数据类型

- 按其值的不同特性，分为3种类型：
  - 原子数据类型

    值不可分解的数据类型，例如：C语言中的整型(int)，实型(float)，字符型(char)，指针类型(*)和空类型(void)等数据类型

  - 固定聚合类型

    由确定数目的成分按某种结构组成，例如：复数是由两个实数按确定次序组成

  - 可变聚合类型

    组成的成分数目不确定，例如：定义一个“有序整数序列”的抽象数据类型，其中序列的长度可变

※固定聚合类型和可变聚合类型又统称为结构类型

1.3 数据类型

二. 抽象数据类型（Abstract Data Type，简称ADT）

- 通常是由用户定义、用以表示应用问题的数学模型以及定义在该模型上的一组操作。其抽象性质使用与实现相分离，实行数据封装和信息隐蔽。
- 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？”的问题。
- 如何描述？

  数据结构 + 此数据结构上的一组操作

1.3 数据类型

二. 抽象数据类型

- 抽象数据类型的定义

  抽象数据类型可用（D，S，P）三元组表示

  - D是数据对象，S是D上的关系集， P是对D的基本操作集

```
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(函数)的定义〉
} ADT 抽象数据类型名
```

1.3 数据类型

二. 抽象数据类型

[例] 三元组抽象数据类型的定义：

ADT Triplet {

数据对象：D = {e1,e2,e3 | e1,e2,e3∈ElemSet}

数据关系：R = {<e1,e2>, <e2,e3>}

基本操作：Max(T, &e)

初始条件：三元组T已存在。

操作结果：用e返回T的3个元素中的最大值。

Min(T, &e)

初始条件：三元组T已存在。

操作结果：用e返回T的3个元素中的最小值。

} ADT Triplet

1.4 算法和算法分析

一. 算法（Algorithm）

- 算法是对特定问题求解步骤的一种描述，是一有限长的操作序列。

算法的描述可以不依赖于任何一种计算机语言以及具体的实现手段。可以用自然语言、流程图等方法来描述。但是，用某一种计算机语言进行伪码描述往往使算法容易被理解。

1.4 算法和算法分析

一. 算法（Algorithm）

- 重要特性：
  - 有穷性：在有穷步骤后能结束，每一步在有限时间内能完成。
  - 确定性：每一步定义都是确切的的，相同的输入有相同的输出。
  - 可行性：算法描述的操作都是可以通过已实现的基本运算经过有限次来实现的，每一步必须在计算机能处理的范围之内。
  - 有输入：有零个、一个或多个输入。
  - 有输出：至少产生一个输出。

1.4 算法和算法分析

一. 算法（Algorithm）

- 算法举例
  - 问题：递增排序
  - 策略：逐个选择最小数据，即选择排序
  - 算法描述：

```
for ( int i = 0; i < n-1; i++ ) {      //n-1次
        从a[i]检查到a[n-1]，找到最小数;若最小整数在a[k],
交换a[i]与a[k];
}
```

1.4 算法和算法分析

二. 算法要求

- 正确性：满足具体问题的需求
- 可读性：便于理解和修改
- 健壮性：当输入数据非法时，也能适当反应
- 效率高：执行时间少
- 空间省：执行中需要的最大存储空间

1.4 算法和算法分析

三. 时间复杂度

- 衡量算法的效率，主要依据算法执行所需要的时间，即时间复杂度。

和算法执行时间相关的因素：

1. 算法的策略
2. 问题的规模n
3. 编写程序的语言
4. 编译程序产生的机器代码的质量
5. 计算机执行指令的速度

1.4 算法和算法分析

- **事后统计法：**计算算法开始时间与完成时间差值

  缺点：必须执行程序；其它因素掩盖算法本质。

- **事前分析估算法**

  算法的执行时间是问题的规模的函数

1.4 算法和算法分析

三. 时间复杂度

- 一般地，用算法中的基本操作语句的重复执行次数（语句频度）表示算法的时间度量，它是问题规模n的某个函数，记为f(n)，f(n) 与 算法的执行时间成正比 。

```
#define MAX  20   //定义最大的方阶
void matrixadd(int n,int A[MAX][MAX],
     int B[MAX][MAX], int C[MAX][MAX])
 {        int i,j;
          for (i=0;i<n;i++)                    //①
               for (j=0;j<n;j++)               //②
                    C[i][j]=A[i][j]+B[i][j];   //③
 }
```

解：除变量定义语句外，该算法包括3个可执行语句①、②和③。

①：频度为$n+1$，循环体执行n次

②：频度为$n(n+1)$

③：频度为n^2

所有语句频度之和为：

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

1.4 算法和算法分析

三. 时间复杂度

- 语句频度的精确计算往往是困难的，因而人们设法估计算法时间复杂度的量级，只考虑宏观渐近性质，即当输入问题规模 n“充分大”时，观察算法的执行时间随着 n 变化的“增长趋势”：当n不断增加时，解决问题所需要的时间的增长变化特点。

1.4 算法和算法分析

三. 时间复杂度

- 渐近时间复杂度

算法的时间量度记作 **T(n) = O(f(n))**，称作算法的渐近时间复杂度，简称时间复杂度。它表示随问题规模**n**的增大，算法执行时间的增长率和**f(n)**的增长率相同。

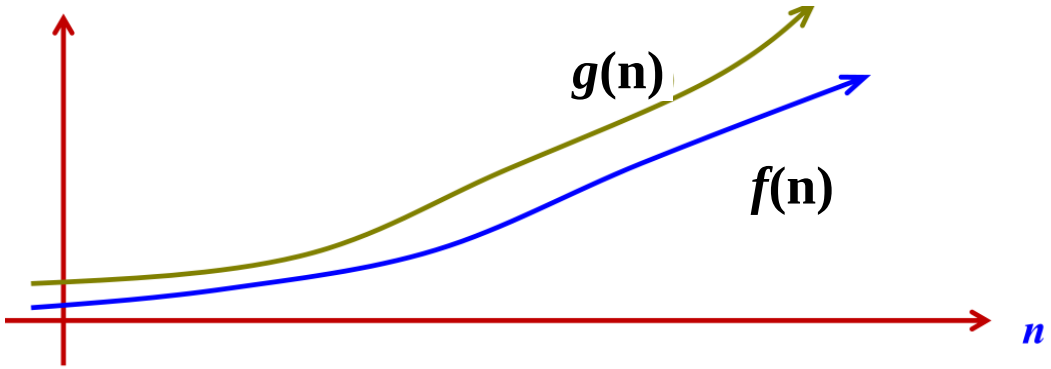
※ 渐近分析法不考虑算法的具体执行时间，只给出算法在问题规模**n**下的执行时间上届。

1.4 算法和算法分析

三. 时间复杂度

- 大O标记法：

如果存在正常数$c > 0$, 自然数$n_0 > 0$，使得当 $n \geq n_0$ 时都有 $f(n) \leq c\, g(n)$，则记作 $f(n) = O(g(n))$，g(n)是f (n)的一个上届，表示 f (n)的增长最多像 g(n)那么快。



1.4 算法和算法分析

三. 时间复杂度

a) {y *= x;}

时间复杂度O (1)，称为常数阶（即原操作语句的频度是常数c，与问题规模n无关）

a) for (i=1; i<=n; i++) { y *= x; }

时间复杂度O (n)，称为 线性阶

a) for (j=1; j<=n; j++)
  for (i=1; i<=n; i++}
    y *= x;

时间复杂度O (n^2)，称为 平方阶

1.4 算法和算法分析

三. 时间复杂度

d) for (i=1; i<=n; i*=2)

{ y *= x; }

e) for (i=n; i>0; i/=2)

{ y *= x; }

- d, e算法中，时间复杂度都是**O**(logN)，称为对数阶。

❖ 常用函数增长表

	输入规模n					
函数	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}

1.4 算法和算法分析

三. 时间复杂度

我们希望随着问题规模时间的增长复杂度是趋于稳定地上升,但上升幅度不能太大

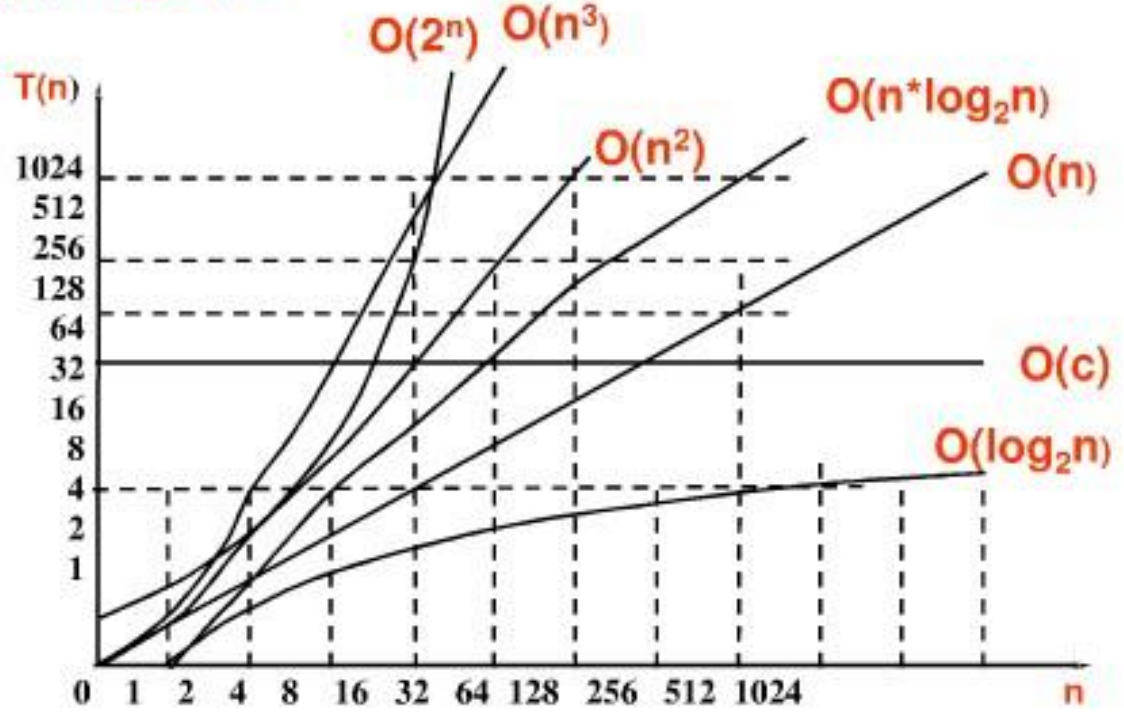


图1-7 常见的T(n)随n变化的增长率

1.4 算法和算法分析

三. 时间复杂度

- 按数量级递增排列，常见的时间复杂度有：
  - 多项式时间复杂度

$$O(1) < O(\log N) < O(N) < O(N\log N) < O(N^2) < O(N^3)$$

  - 指数阶时间复杂度

$$O(2^N) < O(N!) < O(N^N)$$

※通常不宜使用指数阶时间复杂度的算法

1.4 算法和算法分析

❖**求解算法时间复杂度的具体步骤是：**

(1) 找出算法中的基本语句

寻找频度最大的语句，通常是最内层循环的循环体。

(2) 计算基本语句的执行次数的数量级

只需计算基本语句执行次数的数量级，保证基本语句执行次数的函数中的最高次幂正确，忽略所有低次幂和常数系数。

(3) 用大O标记法表示

1.4 算法和算法分析

三. 时间复杂度

- 简化O(f(n)) 的一般过程是：

消除低阶项，只保留随n的增大对函数值增长影响最大的数据项，即最高阶，并忽略常数系数，从本质上讲，是用f(n)的最高数量级评价一个算法的时间性能。

- ※注意：在原操作语句频度不相同时，程序的时间复杂度有可能相同。

例如：

$f(n)=n^2+3n+4$ 与 $f'(n)=4n^2+2n+1$

它们的频度不同，但时间复杂度T(n)都为$O(n^2)$。

1.4 算法和算法分析

❖ 算法时间复杂度分析的一般法则：

(1) 若 $T(n)$是关于n的k阶多项式，那么$T(n) = O(n^k)$

(2)若两段算法分别有复杂度$T_1(n) = O(f_1(n))$ 和 $T_2(n) = O(f_2(n))$ ，

☆ 那么两段算法顺序串联在一起的复杂度:

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

☆ 那么两段算法嵌套在一起的复杂度:

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

(3) 一个循环算法的时间复杂度等于循环次数乘以循环体代码的复杂度。

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

时间复杂度是O(N)

1.4 算法和算法分析

(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$O(N*M)$:

```
for ( i=0; i<N; i++ )
    for ( j=0; j<M; 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支部分的复杂度。

```
if (P1)    /* P1的复杂度为 O(f1) */
    P2;    /* P2的复杂度为 O(f2) */
else
    P3;    /* P3的复杂度为 O(f3) */
```

总复杂度为 $O(f_1) + \max(O(f_2), O(f_3))$ 。

1.4 算法和算法分析

例如： 两个矩阵相乘

```
void mult(int a[], int b[], int& c[] ) {
  // 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)
      for (j=1; j<=n; ++j) {
         c[i,j] = 0;
         for (k=1; k<=n; ++k)
            c[i,j] += a[i,k]*b[k,j];
      } //for
} //mult
```

基本操作：乘法操作

时间复杂度： $O(n^3)$

1.4 算法和算法分析

三. 时间复杂度

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。

例: 在数值A[0..n-1]中查找给定值K的算法大致如下：

```
i=0;
while( i !=n &&  A[i]!=k  )
        i++;
return i;
```

该语句的频度不仅与问题规模n有关，还与输入实例A中的各元素取值及K的取值有关: ①若A中没有与K相等的元素，则该语句的频度是n； ②若A的第一个元素等于K,则该语句的频度是常数0。

1.4 算法和算法分析

三. 时间复杂度

- 平均复杂度： $T_{avg}(n)$
- 最坏情况复杂度： $T_{worst}(n)$

显然：$T_{avg}(n) \leq T_{worst}(n)$

对 $T_{worst}(n)$ 的分析往往比对 $T_{avg}(n)$的分析容易。

原因：最坏情况下的时间复杂度是算法在任何输入实例上运行时间的上界，这就保证了算法的运行时间不会比任何更长。

- 时间复杂度是衡量算法好坏的一个<u>最重要的标准</u>

1.4 算法和算法分析

例如： 冒泡排序

```
void bubble_sort(int& a[], int n) {
  // 将 a 中整数序列重新排列成自小至大有序的整数序列。
    for (i=n-1;  i>1;  --i)
    {
        for (j=0;  j<i;  ++j)
            if (a[j] > a[j+1])    a[j] ←→ a[j+1];
    } // 一趟起泡

} // bubble_sort
```

基本操作：赋值操作

最好情况：0次

最坏情况：n(n-1)/2

平均时间复杂度：$O(n^2)$

1.4 算法和算法分析

四. 空间复杂度

- 空间复杂度是指算法执行时所需要的存储空间的量度，它也是问题规模的函数，即：

$$S(n) = O(f(n))$$

1.4 算法和算法分析

四. 空间复杂度

算法的存储量包括:

1. 程序代码本身所占空间
2. 输入数据所占空间
3. 辅助变量所占空间

1.4 算法和算法分析

四. 空间复杂度

- 若输入数据所占空间只取决于问题本身，和算法无关，则只需要分析除输入和程序之外的辅助变量所占额外间。
- 若所需额外空间相对于输入数据量来说是常数，则称此算法为原地工作，即S(n) = 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。

1.4 算法和算法分析

- 对于一个算法，其时间复杂度和空间复杂度往往是相互影响的。
- 当设计一个算法（特别是大型算法）时，要综合考虑算法的各项性能：
  - ✓ 算法的使用频率
  - ✓ 算法处理的数据量的大小
  - ✓ 算法描述语言的特性
  - ✓ 算法运行的机器系统环境

复习

- 数据、数据元素、数据对象、数据结构的概念，P4
- 四种数据结构：集合、线性、树形、图状
- 数据的物理结构和逻辑结构，P6
- 数据类型和抽象数据类型，P7
  - 两者实质上是同一个概念
- 抽象数据类型用（D，S，P）三元组表示
  - 数据对象、数据关系、基本操作
- 算法是对特定问题求解步骤的一种描述，是一有限长的操作序列
- 算法特性：有穷性、确定性、可行性、输入和输出
- 时间复杂度和空间复杂度

练习

◆ 写出以下代码的时间复杂度。

1、

```
i=0; s=0;
while( s<n )
    {  i++;  s=s+i; }
```

2、

```
 m=0;
while( n>= m*m )
    m++;
```

练习

3、

```
i=1;  j=0;
while( i+j<=n )    {
        if ( i>j )   j++;
       else     i++;
  }
```

4、

```
int rec(int n)   {
        if (n==1)  return 1;
        else  return ( n*rec(n-1) );
  }
```

练习

5、求下列程序中语句K++的频度及程序的时间复杂度。

```
k=0;
for(i=1; i<=n; i++)
   for (j=i; j<=n; j++)
         k++;
```

练习

分析6.1，6.2两题中执行频度最大的语句及其频度。

6.1、

```
x=91; y=100;
while(y>0)
   if (x>10)   y--;
```

6.2、

```
 x=91;  y=100;
while(y>0){
        if(x>100)
        { x=x-10; y--; }
        else  x++;
}
```