计算机领域本科教育教学改革试点
工作计划（“101计划”）研究成果

数据结构

张羽丰

湖南大学

第 1 章

绪论

张羽丰

湖南大学

提纲

1.1 问题引入

1.2 问题求解

1.3 数据结构定义

1.4 算法分析及优化

1.5 应用场景

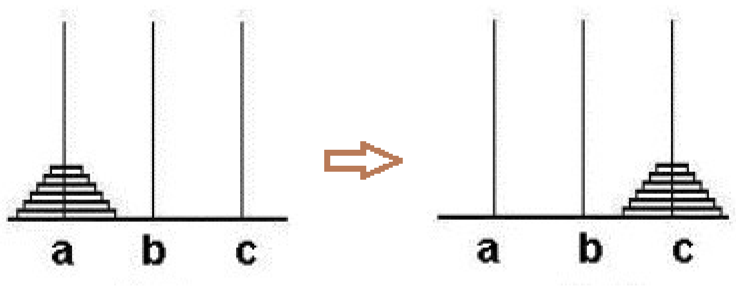
1.1 问题引入：汉诺塔（ Hanoi Tower ）问题

问题：大梵天创造世界的时候做了三根金刚石柱子，在一根柱子上从下往上按照大小顺序摞着64片黄金圆盘。

大梵天命令婆罗门把圆盘从下面开始按大小顺序重新摆放在另一根柱子上。并且规定

1. 任何时候，在小圆盘上都不能放大圆盘
2. 在三根柱子之间一次只能移动一个圆盘

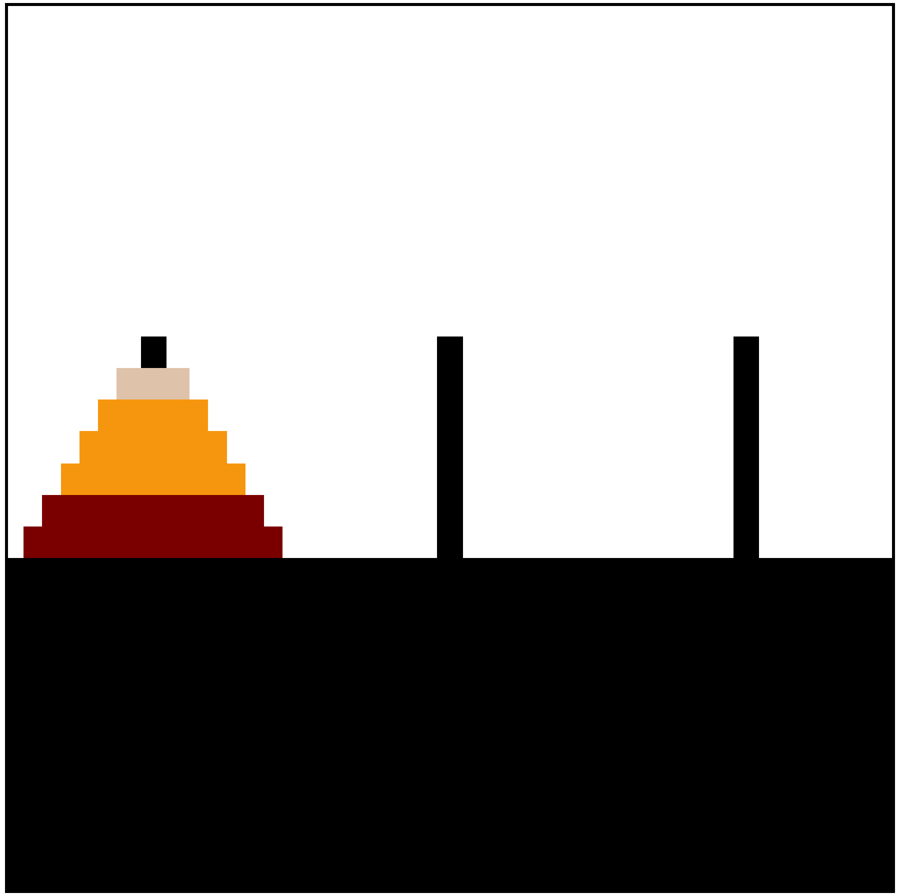
请问应该如何操作?



1.1 问题引入：汉诺塔（Hanoi Tower）问题

大小为6的汉诺塔问题解法

学完数据结构课程，你也可以完成汉诺塔问题~

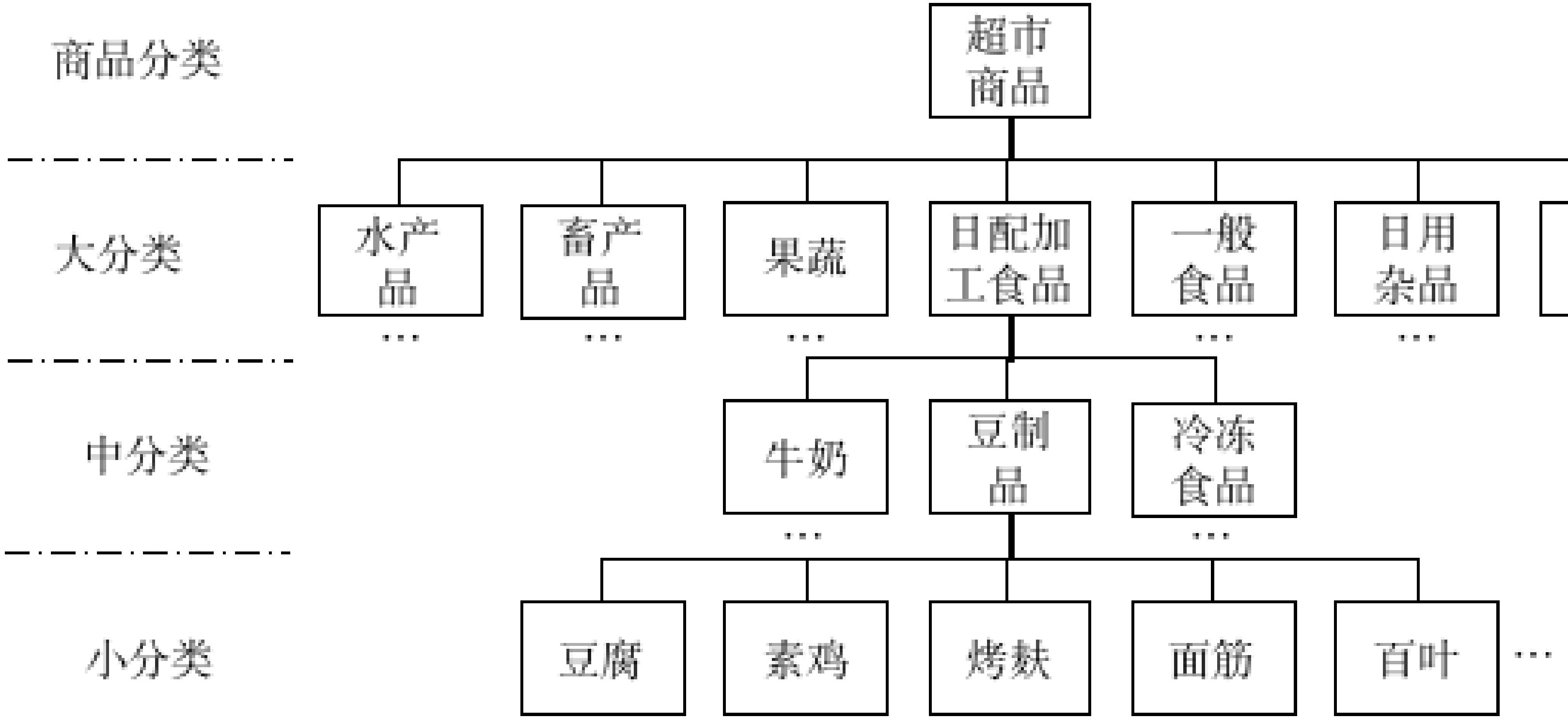


1.1 问题引入：大型超市

问题：顾客如何能快速找到想要购买的商品，超市又是如何实现方便补货呢?

关键：如何陈列商品

商品分类：



1.1 问题引入：大型超市

商品陈列：

商品分类陈列原则：按照商品的分类层次，大区域 → 中区域 → 小区域

价格按序排列原则：由上至下、由左向右，价格由低到高陈列

先进先出陈列原则：对于同一种商品，先摆放的，客户先取到

特价区：无序（乱放）

问题：

如何对商品信息（数据）进行合理的组织（商品分类）、存储（商品陈列）、以及提供必须的操作（商品补架、下架及查找商品）？

↓

如何管理数据以及管理数据的时间和空间的有效性?
(数据结构课程需要研究的两个重要问题)

1.2 问题求解

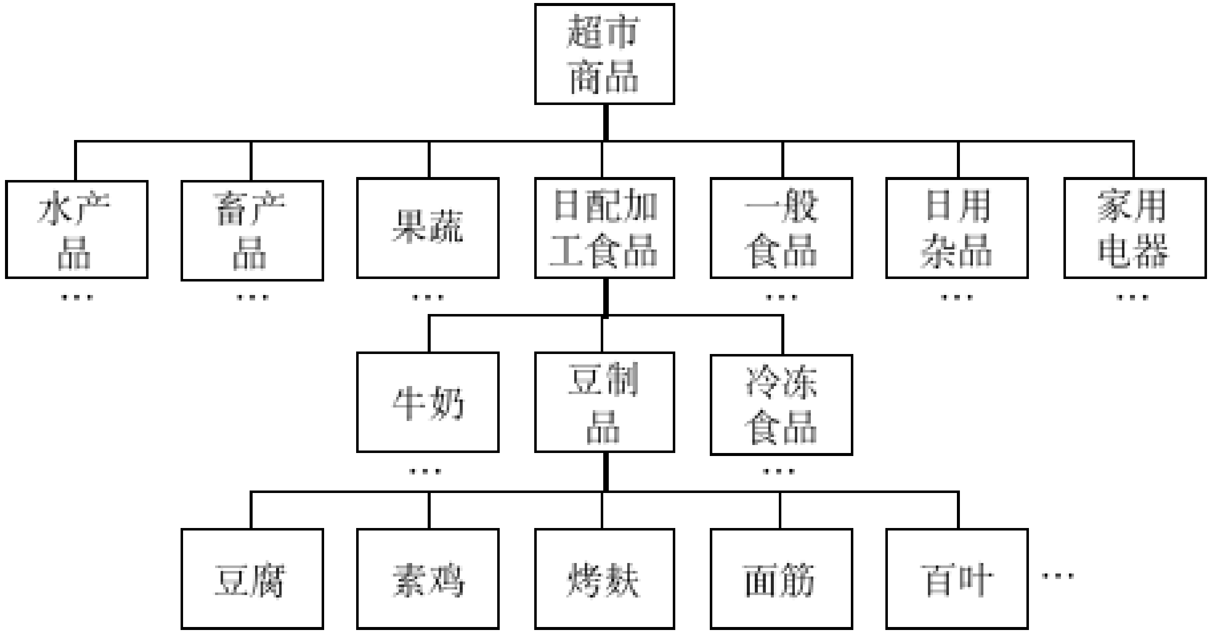
问题分析：为超市寻求一个合适的商品存放方法和所需的对商品的标准操作

商品分类 ➡ 商品编码 ➡ 库存&展示管理

查找商品：通过商品编码检索到该商品对应的商品信息

商品信息：商品产地，商品价格，出厂日期，保质日期等

对商品的抽象



存储结构

超市里几乎所有的管理都不仅与商品有关，还与超市的空间布局有关

必须把商品的数据信息与超市的空间布局进行组合，使商品与其展示位置一一对应

假设某超市将商品划分为A、B、C、D、E、F、G、H八个区域，每个区有9个货架，每个货架有6层。则可设计一个代表物理位置的三位编码：

$$(a_1\ a_2\ a_3)$$

其中， a_1、a_2、a_3分别代表商品的区域、货架及货架层次，其取值范围分别可以为1~8、1~9、1~6。

如：3 5 4表示商品放在区域C、第5个货架的第6层上。

思考：商品编码取值、取值范围是否还有其他方案?

算法设计

算法设计：针对超市商品的操作及实现的问题。例如，在超市中，最常见的操作为商品的补架和下架等。

商品补架

假设当超市货架上某商品已售出20%左右时，该商品需补架。流程如下：

1）如果在货架上某商品已售出20%，则根据该商品的编码，从库存取出该商品满架的20%件数，同时库存减少相应的件数；

2）如商品件数不够，需通知采购补货；

3）将取出的商品放置在指定的货架和层架上，使其满架。

商品下架

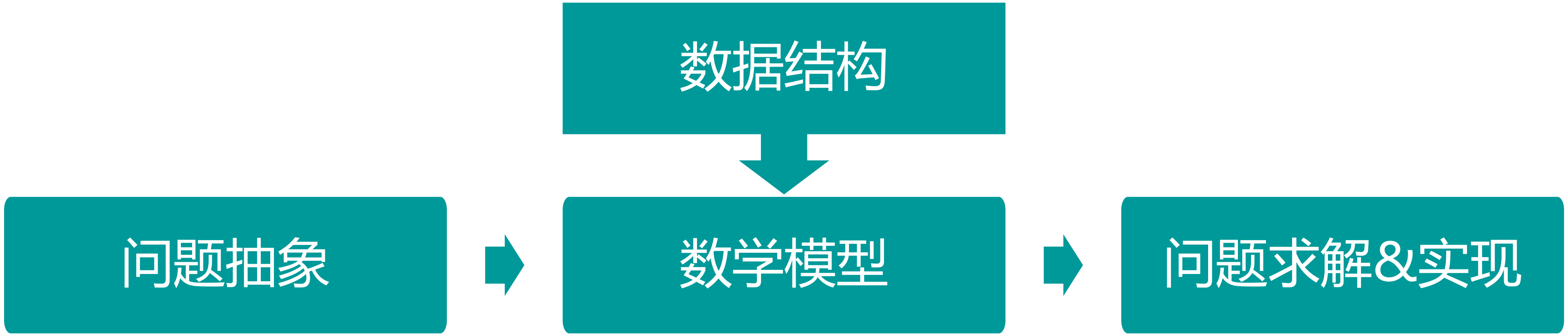
假设当超市货架上某商品已临近有效日期或长时间几乎无售出等情况时，该商品将下架。流程如下：

1）将该商品在货架上的剩余件数全部取下，使货架为空；

2）将取下的商品放回库存，并增加相应的库存量；

3）对该商品库存作相应处置，使其编码失效。

1.3 数据结构定义



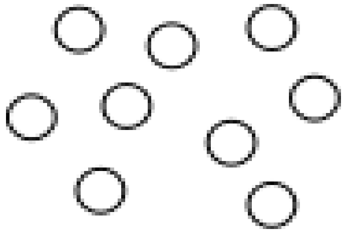
数据结构：一组具有特定关系的同类数据元素的集合。它包括三个要素：**数据的逻辑结构**、**数据的存储结构**及其**操作定义与实现**。

在超市的例子中，商品就是数据元素，商品的编码表示商品的存储结构，商品的上架、下架和补架都是对商品的操作定义与实现。

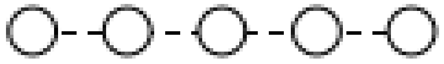
数据的逻辑结构

逻辑上，数据元素之间的关系只有4种：无关系、一对一关系、一对多关系、多对多关系，这4种逻辑关系总称为**数据的逻辑结构。**

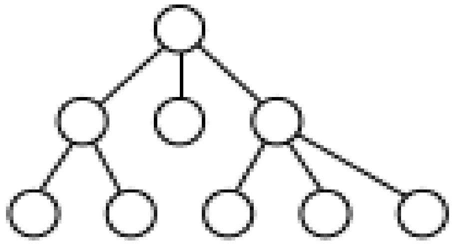
集合：包含的所有数据元素之间无关系，即数据元素之间的次序是任意的。



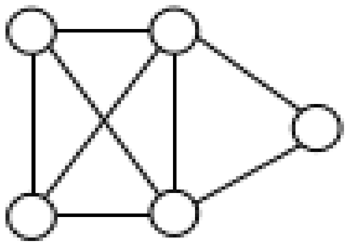
线性结构：包含的数据元素之间存在一对一的关系，即数据元素之间构成一个有序序列。



树形结构：包含的数据元素之间存在一对多的关系，即数据元素之间形成一个层次关系。



图形结构：包含的数据元素（结点）之间存在多对多的关系，即图中每个数据元素的前驱和后继数目都不限。



抽象数据类型

如：两个数的相加，可以是两个整数的相加，也可以是两个浮点数的相加。这时需要针对两个不同的类型的数据元素定义并实现两个加法操作。

抽象数据类型（abstract data structure）：一个与“数据元素及在数据元素之上的实现“无关的数据类型，它们对使用者来说无需知道数据元素的类型，只需知道数据元素之间的逻辑关系，也不用关心是怎么实现的。

ADT 抽象数据类型名 {
数据元素：<数据元素的定义>
数据关系：<数据关系的定义>
基本操作：<基本操作的定义>
}

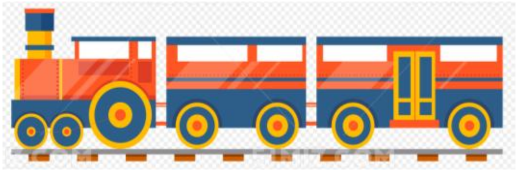
数据的存储结构

数据的存储结构（即**数据的物理结构**）：数据的逻辑结构在计算机内的存储方式。

顺序存储：将所有的数据元素存放在一段连续的存储空间中，数据元素的存储位置反应了它们之间的逻辑关系

座位

链式存储：逻辑上相邻的数据元素不需要在物理位置上也相邻，也就是说数据元素的存储位置可以是任意的



列车

索引存储：在存储数据元素的同时还增加了一个索引表。索引表中的每一项包括关键字和地址，关键字是能够唯一标识一个数据元素的数据项，地址是指向数据元素的存储地址

目录

散列存储（即**哈希存储**）：将数据元素存储在一个连续区域，每一个数据元素的具体存储位置是根据其关键字的值，并通过散列（哈希）函数直接计算出来的

邮政编码

数据的操作实现

数据的操作（也称**运算**或**算法**）：包括操作的定义和实现。

操作定义：对现实问题的抽象，它独立于计算机。

操作实现：建立在数据的存储结构之上完成的，它依赖于计算机和具体的程序设计语言。

例：超市里的商品补架和商品下架的描述就是商品（数据）的操作实现。

重要说明：

本书将不涉及具体程序设计语言，所有的操作（运算）和算法（即问题求解步骤的有限集合）都用伪代码描写，以便读者阅读与理解。

课程要求：学生需要用具体程序设计语言实现所有数据结构

1.4 算法分析与优化

算法的基本概念：

- **正确性**：能够按照预定功能产生正确的输出
- **易读性**：逻辑清楚、结构清晰，算法易于阅读、理解、维护
- **鲁棒性**：对于边界条件输入、不频繁出现的输入，能够产生正确的输出；对于非法输入，算法能够输出相应提示，不会发生崩溃
- **高效率**：在时间和空间上高效，需要较少的运行时间和存储空间

算法0-0：求两个非负整数的最大公约数GCD(x, y)

输入：x, y ∈非负整数集
输出：x, y 的最大公约数

```
if x < y then                  // 判断x与y的大小
| x ↔ y                        // 如x < y，则交换x与y
end
while x mod y ≠ 0 do           // x不能整除y执行循环
| r ← x mod y                  // 计算x除以y的余数r
| x ← y                        // 用y重新赋值x值
| y ← r                        // 用r重新赋值y值
end
return y                       // x整除y，y即为最大公约数
```

时间复杂性的度量

通常情况下，一段程序代码执行的时间性能一般与以下几种因素相关：

- **计算机的硬件性能**，如CPU、GPU的核心数和频率决定了机器的性能

vs

- **编程语言和生成代码的质量**，如Python、C++等不同语言及编译器，所生成的可执行代码效率不同

vs

- **问题和数据的规模**，如在10本书和100本书中寻找所需要的书籍，处理的方式和效率是不一样的

vs

- **算法设计效率**，如针对相同规模大小为N的输入，算法需要消耗线性的时间还是二次幂的时间

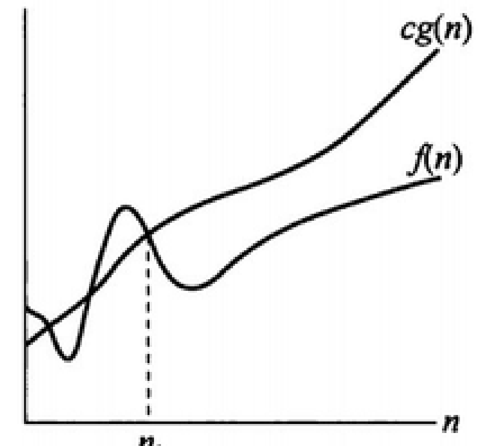
vs

渐近时间复杂度：四种常见表示法O, Ω,Θ, o

大O表示法：若存在一个正数c>0和正整数$n_0 > 0$，使得对所有$n \geq n_0$， 满足T(n) ≤ c·f(n)，则称T(n)=O(f(n))

大O表示法表示T(n)的数量级小于等于f(n)的数量级，表示**小于等于**。

例：$T(n)=3n^2+100n$，求在大O表示法下的时间复杂度。
解：T(n)的时间复杂度是$O(n^2)$。当n≥100时，设c=4，对于所有的n， $T(n) \leq c \cdot n^2$。
据此可以推导出，最高次幂为k的多项式，则其时间复杂度为$O(n^k)$。

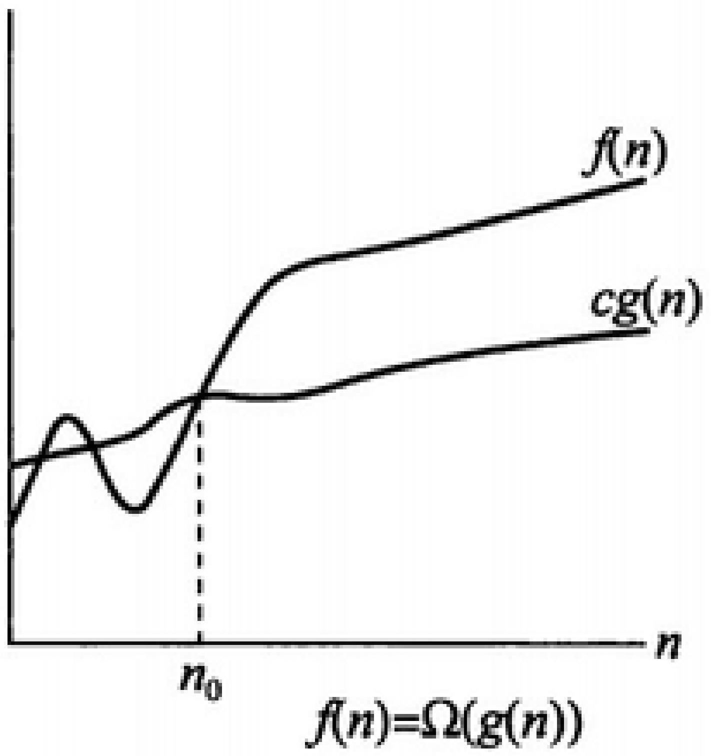


f(n)=O(g(n))

渐近时间复杂度

大Ω(Omega)表示法：若存在一个正数c>0和正整数$n_0 > 0$，使得对所有$n \geq n_0$，满足$T(n) \geq c \cdot f(n)$，则称$T(n)=\Omega(f(n))$

用来表示**复杂度的下界**

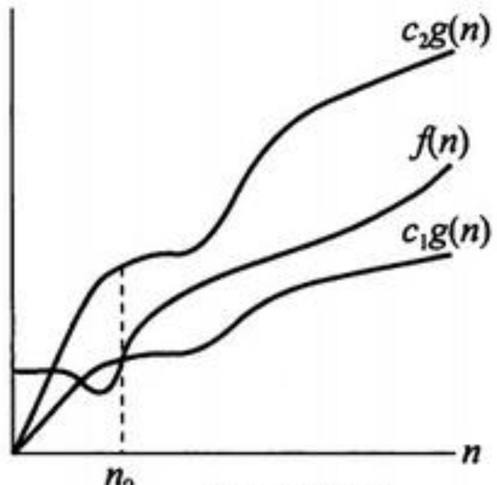


渐近时间复杂度

大Θ(Theta)表示法：若存在一组正数c_1，$c_2 > 0$和正整数$n_0 > 0$，使得对所有$n \geq n_0$，满足$c_1 \cdot f(n) \leq T(n) \leq c_2 \cdot f(n)$，则称$T(n)=\Theta(f(n))$

小o表示法：若对所有正数$c>0$，存在一个正整数$n_0 > 0$，使得对所有$n \geq n_0$，满足$T(n) < c \cdot f(n)$，则称$T(n)=o(f(n))$

小o表示法代表“**严格小于**”，即T(n)的数量级小于f(n)的数量级。

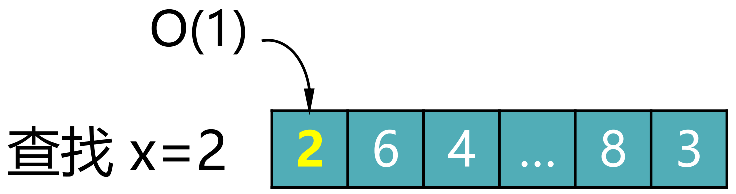


$f(n)=\Theta(g(n))$

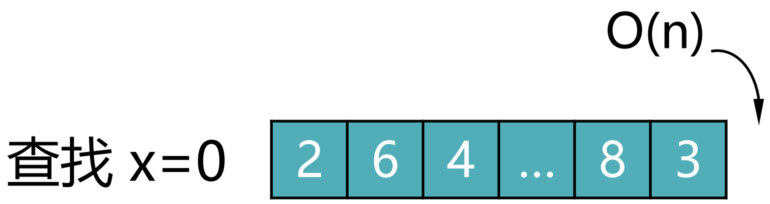
最好、最坏、平均情况时间复杂度

例：假设现有一函数F，其功能是在一个无序的数组A中查找变量 x 出现的位置。如果找到则停止，并返回x在数组A中的下标；若没有找到，则返回-1。

最好情况复杂度：在最理想的情况下，算法所能达到的最高效率。如要查找的变量 x 正好是数组的第一个元素，对应最好情况时间复杂度O(1)



最坏情况复杂度：算法可能遇到的最糟糕情况的效率，算法耗时最长。如果数组中没有要查找的变量 x，需要把整个数组都遍历一遍，对应最坏情况时间复杂度O(n)



平均情况复杂度：算法在所有可能输入的平均效率。通常假设所有输入出现的概率符合特定的分布（最简单的为均匀分布）。对于函数F，其平均时间复杂度为O(n)

$$\sum_{i=1}^{n} i\cdot\frac{1}{n}=\frac{n(n+1)}{2n}=O(n)$$

空间复杂性的度量

算法执行时的空间消耗：包括程序代码本身所占的空间、存储数据所占的空间和中间过程使用的辅助空间等。

注意：

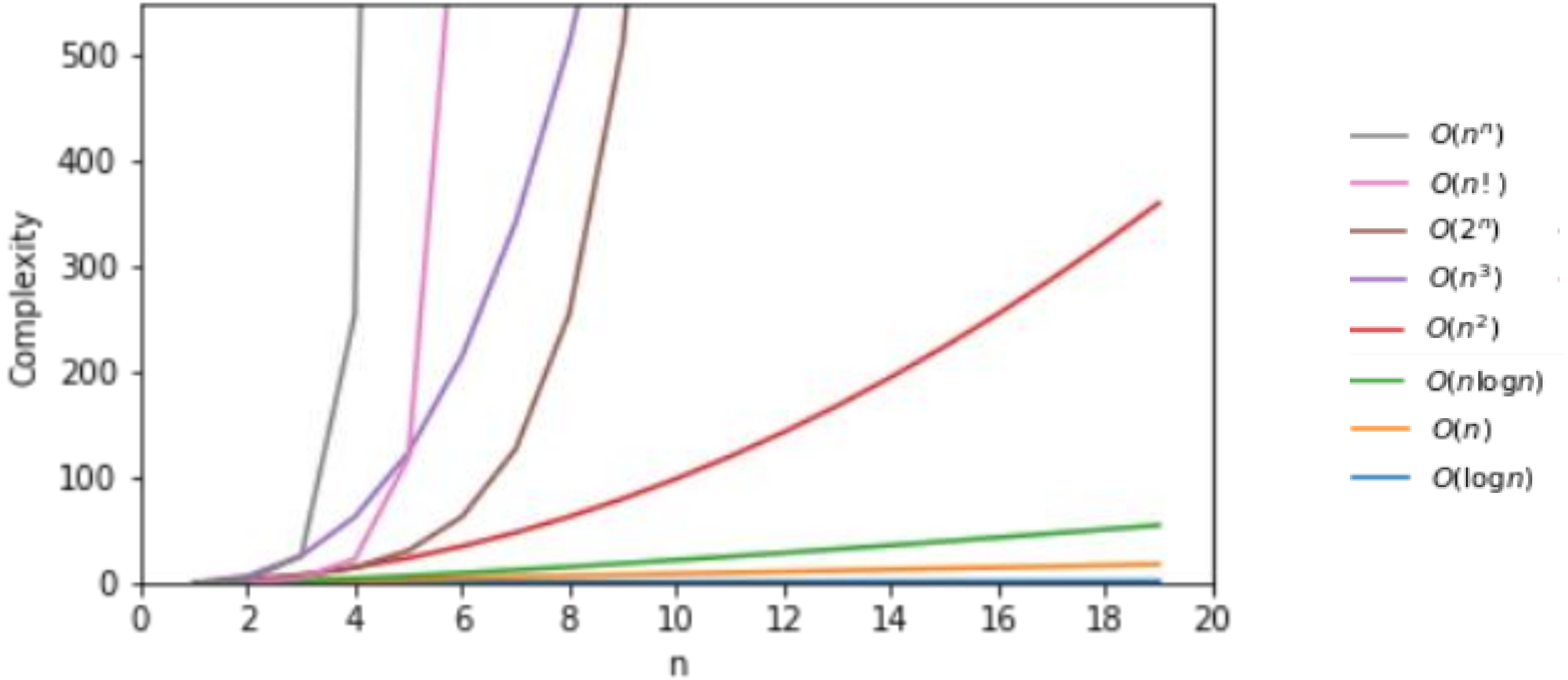
- 算法运行的瓶颈在于内存空间（以TB为单位）与外存空间（以TB为单位）的较大差异引起的
- 每一段程序代码在执行时，都需要将其代码和所需的数据装入内存。若所需空间大于现有内存，则会出现内存溢出、宕机等情况

算法的空间复杂性：度量算法所使用的辅助空间大小和数据规模n之间的关系。空间复杂性的表示也常使用渐近复杂度来表达，定义方法与时间复杂性相似。

常用复杂度函数

通常采用以下几种常见的时间复杂度函数。如图所示，当N逐渐增大时，它们的时间复杂度由左到右依次增大：

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



渐近表示法的计算

求和定理：假设两个已知程序片段的时间复杂度分别为$T_1(n)=O(f(n))$和$T_2(n)=O(g(n))$，那么顺序组合两个程序片段得到的程序的时间复杂度为：

$$T1(n)+T2(n)= O(Max(f(n), g(n)))$$

用途：适用于顺序语句/程序片段

求积定理：假设两个已知程序片段的时间复杂度分别为T1(n)=O(f(n))和T2(n)=O(g(n))，那么交叉乘法组合两个程序片段得到的程序时间复杂度为：

$$T1(n)\cdot T2(n)= O(f(n)\cdot g(n))$$

用途：适用于嵌套/多层嵌套循环语句

算法优化 – 时间复杂度

例：连续子序列最大和问题。该问题关注一个序列s，其元素值存储在一维整数数组s.array, 数组大小为s.n，希望从s.array中找出一个连续子序列，该子序列各元素的和最大。如果序列元素都是负数，计算结果返回0。

$O(n^3)$算法：

1. 枚举所有子序列
2. 找出和最大的子序列

运用求积定理，三层for循环：

$$\sum_{i=1}^{n}\sum_{j=i}^{n}\sum_{k=i}^{j} 1 = n(n+1)(n+2)/6 = O(n^3)$$

总的时间复杂度就是$O(n^3)$

算法1-7：连续子序列最大和问题 - $O(n^3)$算法片段

输入：序列s，s.array[i]∈整数集
输出：序列s中最大的连续子序列之和max_sum

```
s.max_sum ← 0                       //设置最大子序列和初值
for i←1 to s.n do                   //子序列起始位置
| for j←i to s.n do                 //子序列结束位置
| | this_sum ← 0                    //设置子序列和初值
| | for k←i to j do                 //求子序列和
| | | this_sum ← this_sum+s.array[k]
| | end
| | if this_sum>s.max_sum then //如当前子序列和更大
| | | s.max_sum ← this_sum   //设置当前最大子序列和
| | | s.start ← i            //设置当前最大子序列起始位置
| | | s.finish ← j           //设置当前最大子序列结束位置
| | end
| end
end
```

算法优化 – 时间复杂度

可以优化吗?

设置结束点的循环体和最内层的循环体功能有重合之处，最内层循环有重复计算的现象。

例如： start=1时，对end=1,2,3,…,n，分别计算

array[1]

array[1]+array[2]

array[1]+array[2]+array[3]

……

算法优化 – 时间复杂度

例：连续子序列最大和问题。该问题关注一个序列s，其元素值存储在一维整数数组s.array, 数组大小为s.n，希望从s.array中找出一个连续子序列，该子序列各元素的和最大。如果序列元素都是负数，计算结果返回0。

$O(n^2)$算法：

$O(n^3)$算法的第三个循环是重复了第二个循环求和，于是可以简化为双重循环。

同样运用求积定理，两层for循环：

$$\sum_{i=1}^{n}\sum_{j=i}^{n} 1 = n(n+1)/2 = O(n^2)$$

总的时间复杂度就是$O(n^2)$

算法1-8：连续子序列最大和问题 - $O(n^2)$算法片段

输入：序列s，s.array[i]∈整数集
输出：序列s中最大的连续子序列之和max_sum

```
s.max_sum ← 0                  //设置最大子序列和初值
for i←1 to s.n do              //子序列起始位置
| this_sum ← 0                 //设置子序列和初值
| for j←i to s.n do            //子序列结束位置
| |  this_sum ← this_sum+s.array[j]
| end
| if this_sum>s.max_sum then //如当前子序列和更大
| | s.max_sum ← this_sum   //设置当前最大子序列和
| | s.start ← i            //设置当前最大子序列起始位置
| | s.finish ← j           //设置当前最大子序列结束位置
| end
end
```

算法优化 － 时间复杂度

可以优化吗?

问题：

如果元素均大于零，如何计算？**如序列{2,2,3,4}**

如果出现了一个很小的负值，如何处理？**如序列{2,2,3,-1,4}**

如果出现了一个较大的负值，该如何处理？**如序列{3,4,-9,2,9,7}**

观察：如果一个子序列的和小于零，那么以它为起始或终止的更大子序列可以将该子序列舍去，余下的子序列将有更大的子序列和。

例如：序列3,4,-9,2,9,7

子序列{3,4,-9} {4,-9} {-9} {-9,2}求和均小于零，可以将它们舍去

算法优化 – 时间复杂度

例：连续子序列最大和问题。该问题关注一个序列s，其元素值存储在一维整数数组s.array, 数组大小为s.n，希望从s.array中找出一个连续子序列，该子序列各元素的和最大。如果序列元素都是负数，计算结果返回0。

O(n)算法：

如子序列和小于0，则可放弃，重新计算新的子序列和（从上个子序列结束位置+1开始），这样只需一个循环（依次扫描）就可以计算出最大子序列和。

总的时间复杂度就是O(n)

算法1-9：连续子序列最大和问题 - O(n)算法片段

```
输入：序列s，s.array[i]∈整数集
输出：序列s中最大的连续子序列之和max_sum
s.max_sum ← 0                      //设置最大子序列和初值
this_sum ← 0                       //设置当前子序列和初值
s.start ← 0                        //设置子序列开始位置
for j←1 to s.n do                  //循环
| this_sum ← this_sum+s.array[j]
| if this_sum>s.max_sum then //如当前子序列和更大
| | s.max_sum ← this_sum   //设置当前最大子序列和
| | s.start ← this_start
| | s.finish ← j   //设置当前最大子序列结束位置
| else if this_sum < 0 then  //如子序列和小于0
| | this_sum ← 0                //重新计算子序列和
| | s.start ← j+1               //从上子序列结束后开始
| end
end
```

算法优化 – 空间复杂度

例：假设需要输出由小到大，从1 到n的所有的数字。可用递归调用完成。

递归算法：直到n=0开始返回上一层，并从1开始输出，一直到最后打印n。该函数通常只能执行数万次，就会因递归层数过多，系统栈空间不足而报错，也称**递归爆栈**。因为递归时，每次进入更深一层，都需要将当前空间的状态进行存储，消耗一定的内存空间。

算法1-10：输出1~n的**递归算法** RecursivePrint(n)

输入：正整数 n > 0
输出：从 1 到 n 的数字
初始调用：RecursivePrintt(n)

```
1. if n>0 then
2. | RecursivePrint(n-1)
3. | print(n)
4. end
```

算法优化 – 空间复杂度

例：假设需要输出由小到大，从1 到n的所有的数字。可用递归调用完成。

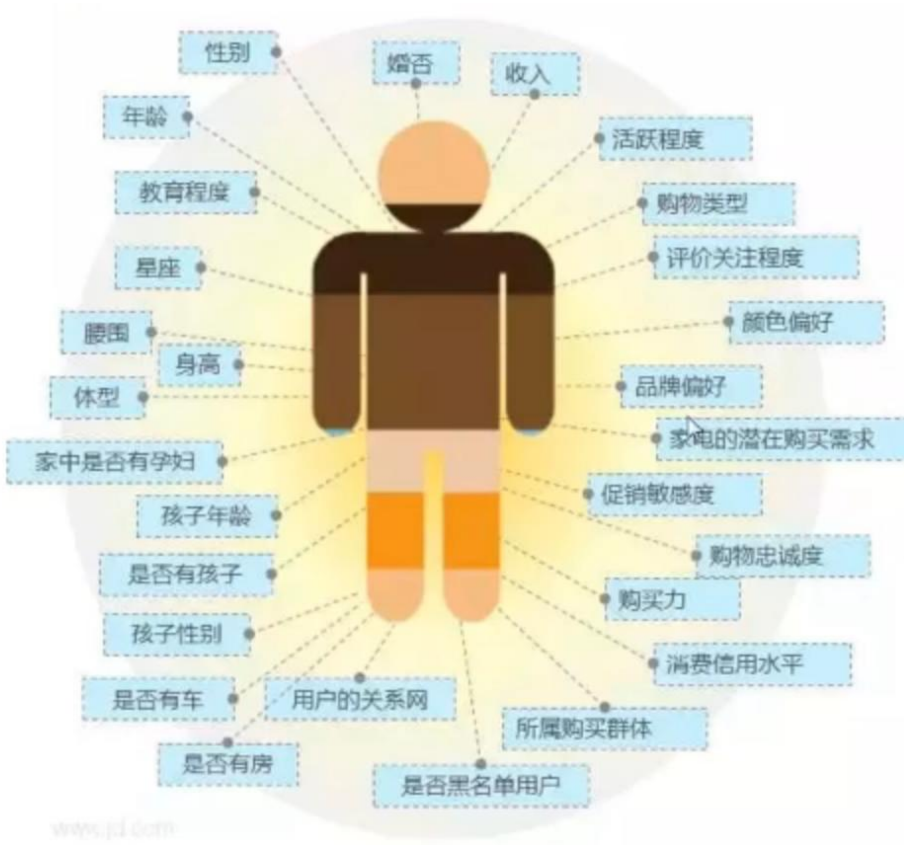
循环算法：只涉及两个变量的维护，循环调用多少次，内存消耗也不变。不会内存溢出。

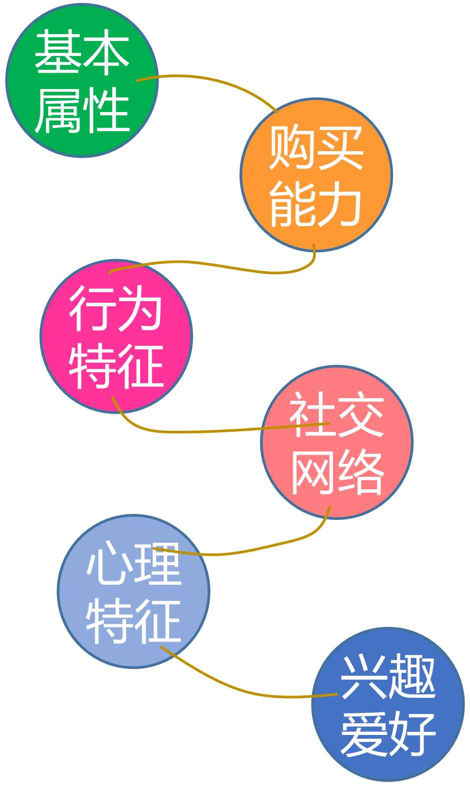
算法1-10：输出1~n的**循环算法** RecursivePrint(n)

输入：正整数 n > 0
输出：从 1 到 n 的数字
1. **for** i←1 **to** n **do**
2. | **print** (i)
3. **end**

1.5 应用场景：数据挖掘

用户画像：基于用户线上行为数据抽象出他/她的信息全貌





1.6 小结

本章主要介绍了数据结构及算法分析两个重要概念：

- **数据结构**：一组具有特定关系的同类数据元素的集合，其主要研究数据的逻辑结构、数据的存储结构及其操作定义与实现
- **逻辑结构**：包括集合、线性结构、树形结构和图形结构
- **存储结构**：包括顺序存储、链接存储、索引存储及散列存储（也称哈希存储）
- **操作**（也称**运算**）：包括操作的定义与实现
- **算法分析**：对一个算法的时间和空间复杂度作定量分析，来衡量算法的优劣
- **算法分析的方法**：通常采用渐近表示法分析算法复杂度的增长趋势，一般使用大O表示法

谢谢观看