第 1 章

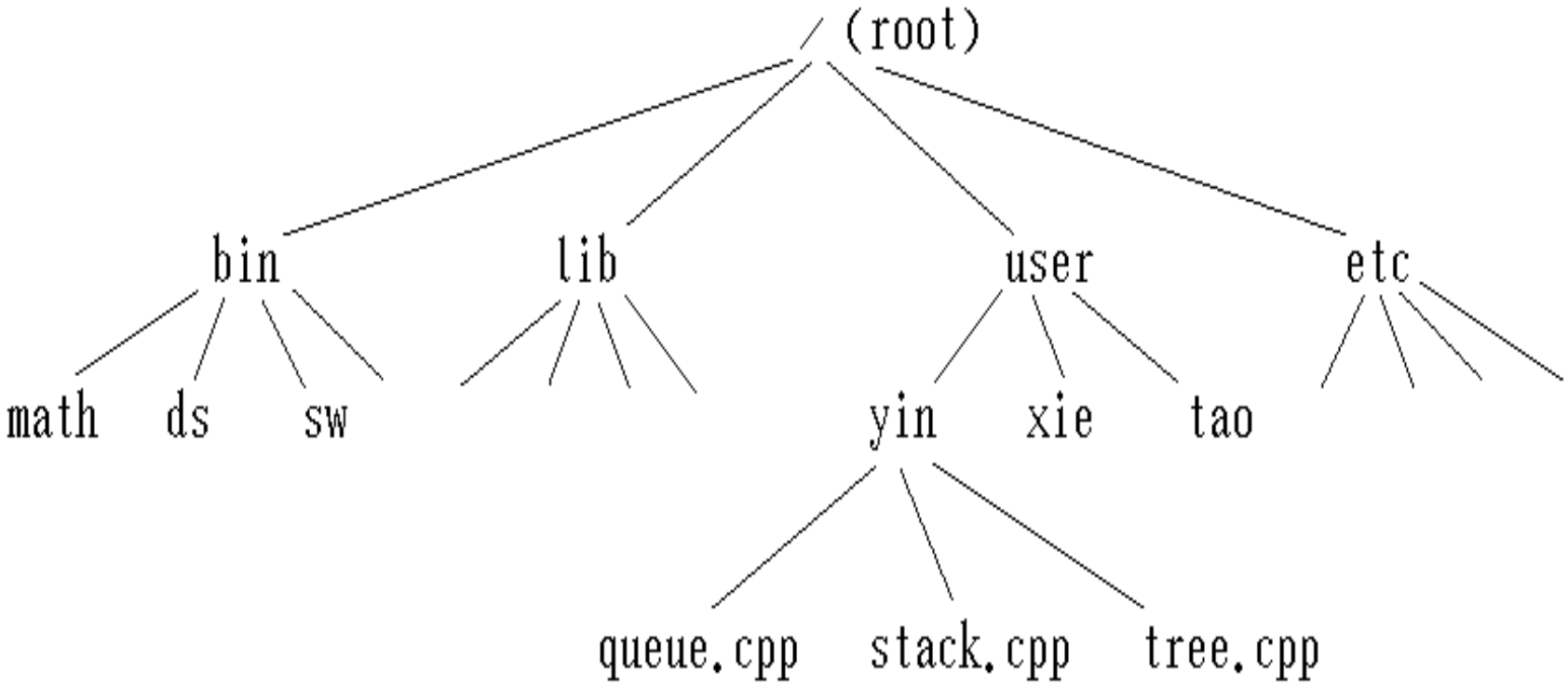
绪 论

提纲

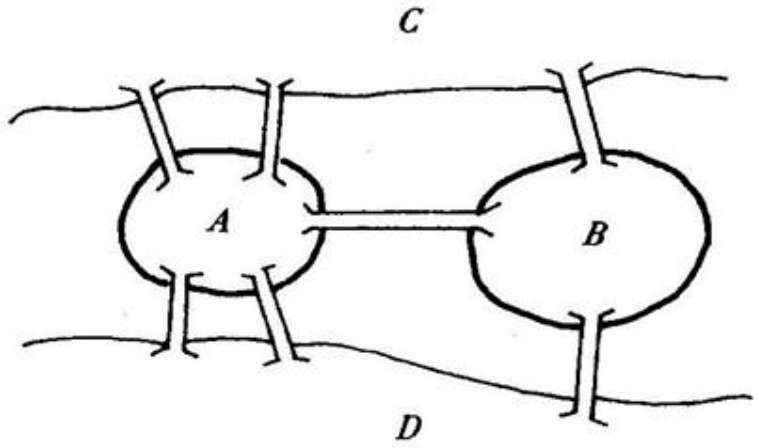
问题场景

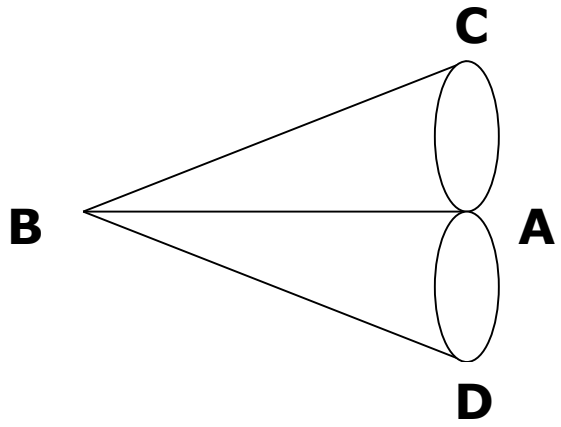
	国家/地区	金	银	铜	总数
1	挪威	16	8	13	37
2	德国	12	10	5	27
3	中国	9	4	2	15
4	美国	8	10	7	25
5	瑞典	8	5	5	18
6	荷兰	8	5	4	17
7	奥地利	7	7	4	18
8	瑞士	7	2	5	14
9	俄罗斯奥委会	6	12	14	32
10	法国	5	7	2	14

问题场景


(root)
bin
lib
user
etc
math
ds
sw
yin
xie
tao
queue.cpp
stack.cpp
tree.cpp

问题场景





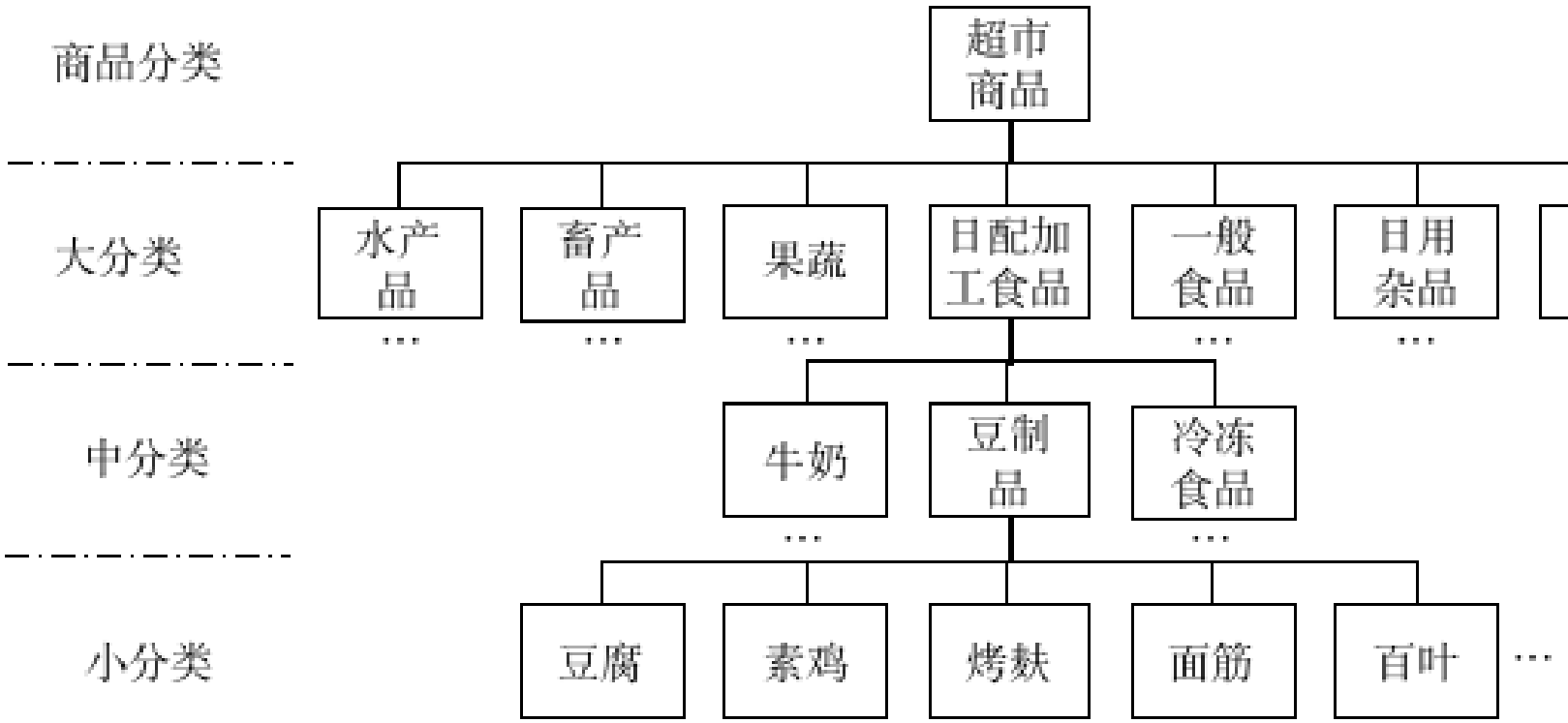
Seven Bridges Problem

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

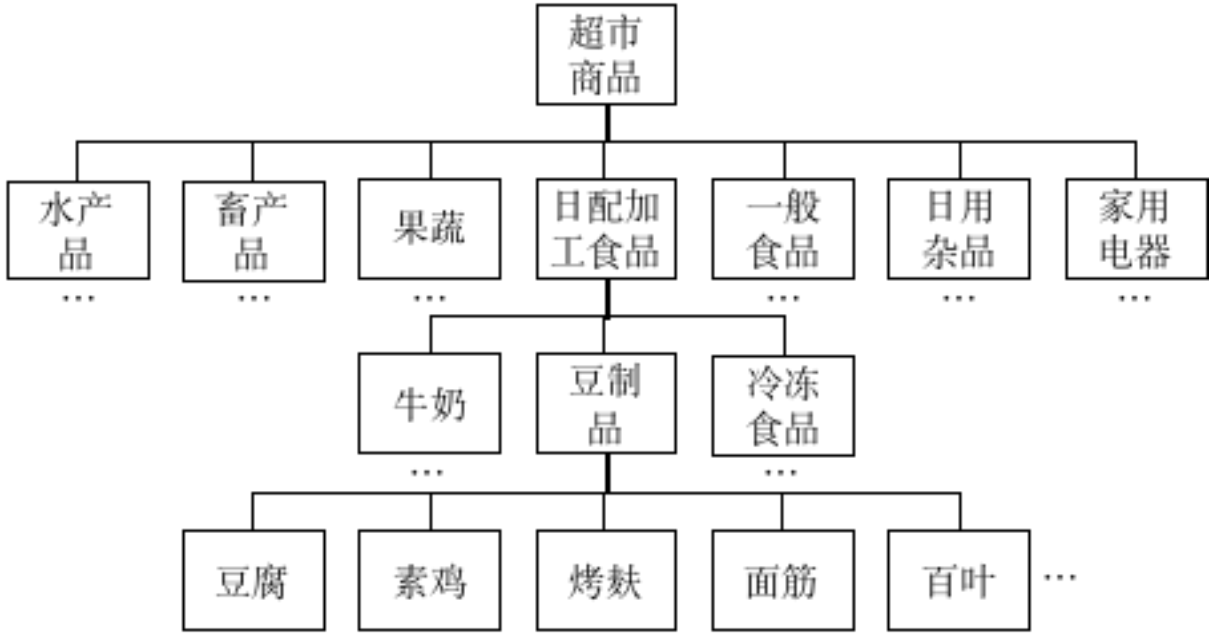
问题分析：为超市寻求一个合适的商品存放方法和所需的对商品的标准操作

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

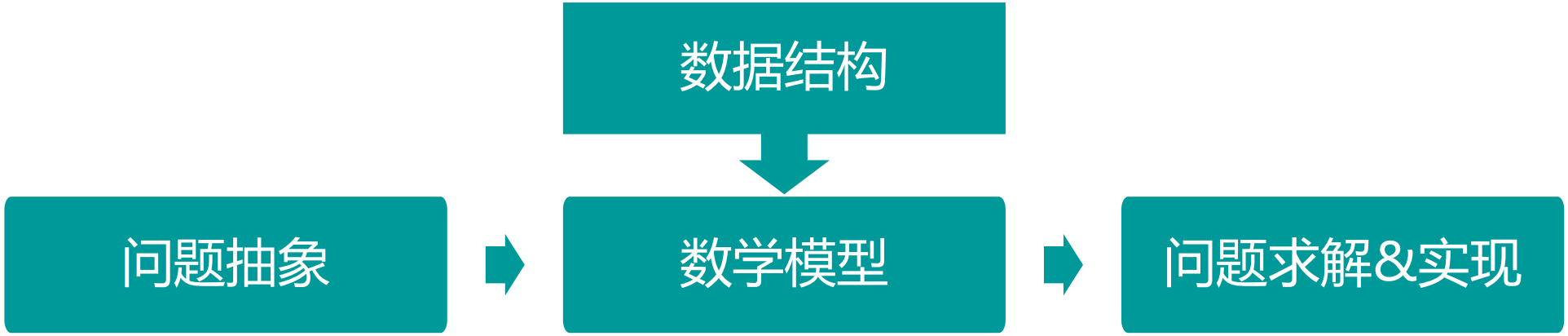
假设当超市货架上某商品已临近有效日期或长时间几乎无售出等情况时，该商品将下架。流程如下：

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

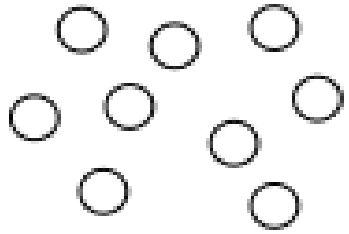


数据结构：一组具有特定关系的同类数据元素的集合。它包括三个要素：**数据的逻辑结构**、**数据的存储结构**及其**操作定义与实现**。

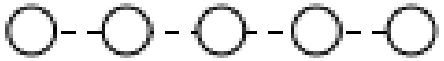
在超市的例子中，商品就是数据元素，商品的编码表示商品的存储结构，商品的上架、下架和补架都是对商品的操作定义与实现。

数据的逻辑结构

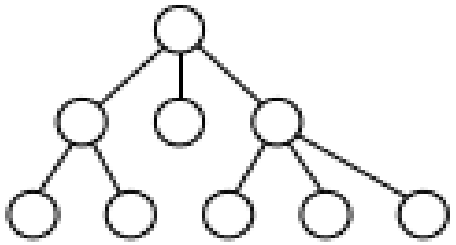
逻辑上，数据元素之间的关系只有4种：无关系、一对一关系、一对多关系、多对多关系，这4种逻辑关系总称为**数据的逻辑结构**。



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。

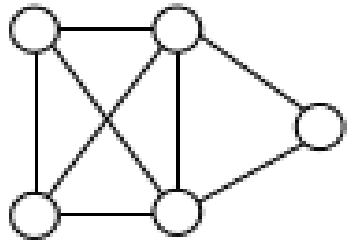


线性结构：包含的数据元素之间存在一对一的关系，即数据元素之间构成一个有序序列。



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：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>
}
```

抽象数据类型

ADT Complex { //复数
    数据元素：real, image ∈ R
        //实部、虚部
    数据关系：{<real, image>}
        //二元组
    基本操作： + //加法运算
}

```
// C++
class Complex{
    private:
        float real, image;

    public:
        Complex(float r=0, float i=0) {
            real = r, image = i;
        } //constructor
        …

        friend Complex operator+(const Complex& a, const Complex& b) ;
                    //友元函数重载
};

Complex operator+(const Complex& a, const Complex& b) {
    return Complex(a.real+b.real, a.image+b.image);
}
```

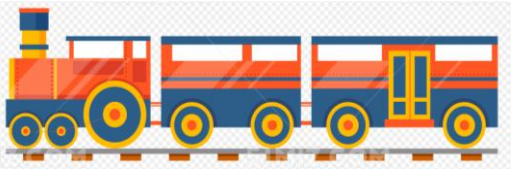
数据的存储结构

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。

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

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

1.4 算法分析与优化

算法0-0-0：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

```
1.  if x < y then                  // 判断x与y的大小
2.  | x ↔ y                        // 如x < y，则交换x与y
3.  end
4.  gcd ← 1                        //最大公约数的初始值
5.  for r ← 2 to y do              //r从小到大取值
6.  | if x mod r = 0 且 y mod r = 0 then
7.  |                              // r是x和y的公因数
8.  | | gcd ← r                    // r赋值给gcd
9.  | end
10. end
11. return gcd
```

易读性高、正确性易证明

欧几里得算法

算法0-0-1：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

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

易读，但正确性较难证明

1.4 算法分析与优化

欧几里得算法的证明

欧几里得定理

设x和y是正整数，并且$0 < y \leq x$，则x和y的最大公约数gcd(x, y)满足以下性质：

(1) 如果 x mod y = 0, 则gcd(x, y) = y;

(2) 否则，gcd(x, y) = gcd(y, x mod y)

性质（1）不用证明

性质（2）：

设$\gcd(x, y) = z$, 则$x = az$ 且 $y = bz$ $(a > b > 1)$

∵ z是az和bz的最大公约数

∴ $\gcd(b, (a \bmod b)) = 1$(?)

$$\gcd(y, x \bmod y) = \gcd(bz, (a \bmod b)z) = z$$

欧几里得算法

算法0-0-1：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

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

易读，但正确性较难证明

欧几里得辗转相除法证明

辗转相除法就是给出了一个关系：$\gcd(a,b)=\gcd(b,r)$，其中a、b、r满足a=bq+r。

设u同时整除a和b，则有

$$a=su,b=tu$$

则$r=a-bq=su-tuq=(s-tq)u$，得到u也整除r。反过来，每一个整除b和r的整数v，则有

$$b=s'v，\ r=t'v.$$

则$a=bq+r=s'vq+t'v=(s'q+t')v$，得到v也整除a。

综上：a和b的每一个公因子也是b和r的一个因子，反之亦然。所以a和b的全体公因子集合就与b和r的全体公因子集合相同。所以gcd（a,b）=gcd(b,r)。

1.4 算法分析与优化

算法0-0-0：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

```
1.  if x < y then                  // 判断x与y的大小
2.  | x ↔ y                        // 如x < y，则交换x与y
3.  end
4.  gcd ← 1                        //最大公约数的初始值
5.  for r ← 2 to y do              //r从小到大取值
6.  | if x mod r = 0 且 y mod r = 0 then
7.                                 // r是x和y的公因数
8.  | | gcd ← r                    // r赋值给gcd
9.  | end
10. end
11. return gcd
```

效率低

欧几里得算法

算法0-0-1：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

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

效率高！

1.4 算法分析与优化

算法0-0-0：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

```
1.  if x < y then                  // 判断x与y的大小
2.  | x ↔ y                        // 如x < y，则交换x与y
3.  end
4.  gcd ← 1                        //最大公约数的初始值
5.  for r ← 2 to y do              //r从小到大取值
6.  | if x mod r = 0 且 y mod r = 0 then
7.  |                              // r是x和y的公因数
8.  | | gcd ← r                    // r赋值给gcd
9.  | end
10. end
11. return gcd
```

效率低

效率低的原因

- 比如求98和63的最大公约数，需要对从1到63的所有整数进行验证
- 代码5-9行循环执行62次

Q：求11002000083和10010300321的最大公约数需要验证多少个整数?

答案：超过10^{10}

Untouchable!

1.4 算法分析与优化

gcd(98, 63) → gcd(63, 35)→ gcd(35, 28) → gcd(28, 7)

代码4-8行只循环3次

求gcd(x, y), 欧几里得算法的循环（代码4-8行）次数不超过 2log(x)

思考：$\log(10^{10}) = ?$

$$\gcd(x,y) \rightarrow \gcd(y, x\ \%\ y) \rightarrow \gcd\big(x\%y, y\ \%\ (x\%y)\big) \rightarrow \cdots$$

$x \bmod y < \frac{x}{2}$

参数至少减小了一半!

欧几里得算法

算法0-0-1：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

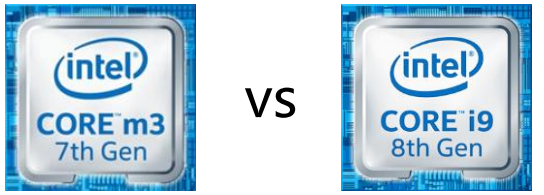
```
1. if x < y then                  // 判断x与y的大小
2. | x ↔ y                        // 如x < y，则交换x与y
3. end
4. while x mod y ≠ 0 do // x不能整除y执行循环
5. | r ← x mod y                  // 计算x除以y的余数r
6. | x ← y                        // 用y重新赋值x值
7. | y ← r                        // 用r重新赋值y值
8. end
9. return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

效率高!

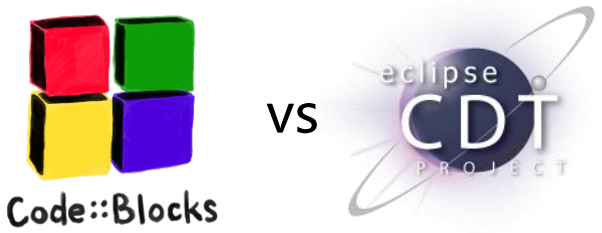
时间复杂性的度量

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：

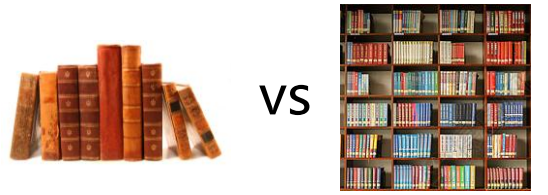
- **计算机的硬件性能**，如CPU、GPU的核心数和频率决定了机器的性能



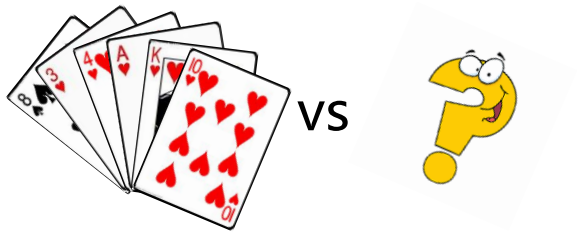
- **编程语言和生成代码的质量**，如Python、C++等不同语言及编译器，所生成的可执行代码效率不同



- **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



- **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



时间复杂性的度量

算法运行时间取决于以下因素：

计算模型: RAM (random-access machine):　指令顺序执行，无并发.

硬件: CPU有多快

输入规模: 有多少数据

基本操作的数量

- 算术运算: add, subtract, multiply, divide, remainder, floor, ceiling
- 逻辑运算: and, or, large, less, equal, not equal
- 数据移动与控制: load, store, copy; conditional and unconditional branch, subroutine call and return

时间复杂性的度量

算法时间复杂度：T(n)=Sum（各种基本操作数量)

渐进分析(Asymptotic Analysis)：一种粗略的估计技术，但很有效。

- 基本思想
  - 忽略机器相关因素
  - 关注当 $n \to \infty$ 时 $T(n)$ 的增长度.

- $T(n)$: 渐进运行时间
  - 忽略事实：每条语句的执行时间取决于编译器、解释器以及硬件平台
  - 表示最坏情况
  - 可以用函数$f(n)$表示T(n)

时间复杂性的度量

四种常见的渐近时间复杂度表示方法：

$T(n)=O(f(n))$	$T(n)=\Omega(f(n))$	$T(n)=\Theta(f(n))$	$T(n)=o(f(n))$
存在一个正数c和n_0，使得对所有$n\geqslant n_0$，满足$T(n)\leqslant c\cdot f(n)$	存在一个正数c和n_0，使得对所有$n\geqslant n_0$，满足$T(n)\geqslant c\cdot f(n)$	存在一组正数c_1、c_2和n_0，使得对所有$n\geqslant n_0$，满足$c_1\cdot f(n)\leqslant T(n)\leqslant c_2\cdot f(n)$	对所有$c>0$，存在一个n_0的选择，满足对所有$n\geqslant n_0$，都有$T(n)\leqslant c\cdot f(n)$

大O表示法表示T(n)的数量级小于等于f(n)的数量级。而与大O表示法的“**小于等于**”不同，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。

时间复杂性的度量

记号	含义	通俗理解
(1)Θ（西塔）	紧确界。	相当于"="
(2)O （大欧）	上界。	相当于"<="
(3)o（小欧）	非紧的上界。	相当于"<"
(4)Ω（大欧米伽）	下界。	相当于">="
(5)ω（小欧米伽）	非紧的下界。	相当于">"

渐近时间复杂度

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$T(n)\leq c\cdot f(n)$，则称$T(n)=O(f(n))$

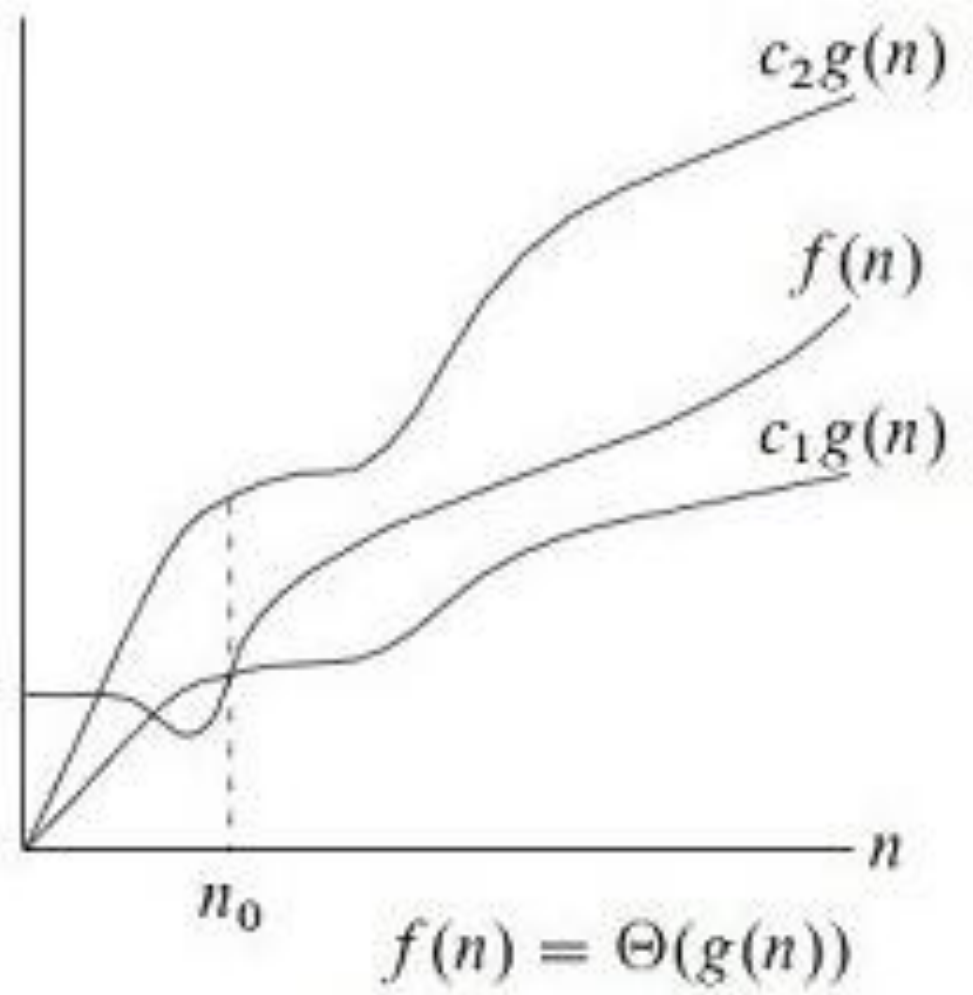
大Ω表示法：若存在一个正数$c>0$和正整数$n_0>0$，使得对所有$n\geq n_0$，满足$T(n)\geq c\cdot f(n)$，则称$T(n)=\Omega(f(n))$

大Θ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$T(n)=\Theta(f(n))$

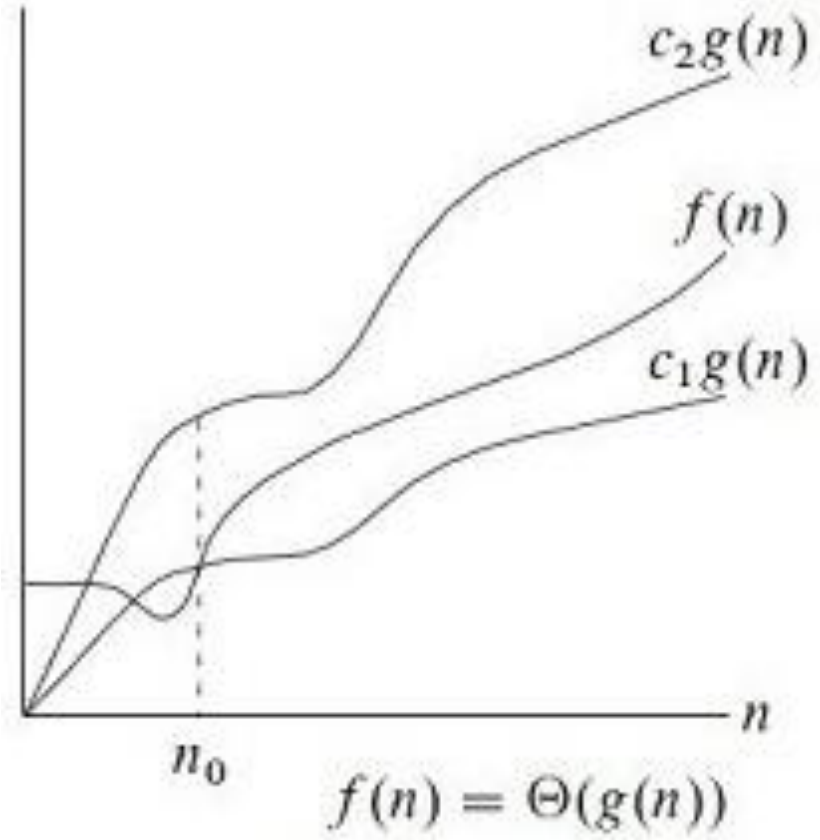
小o表示法：若对所有正数$c>0$，存在一个正整数$n_0>0$，使得对所有$n\geq n_0$，满足$T(n)<c\cdot f(n)$，则称$T(n)=o(f(n))$

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$n\geq 100$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$O(n^k)$。

Big Θ



Big Θ



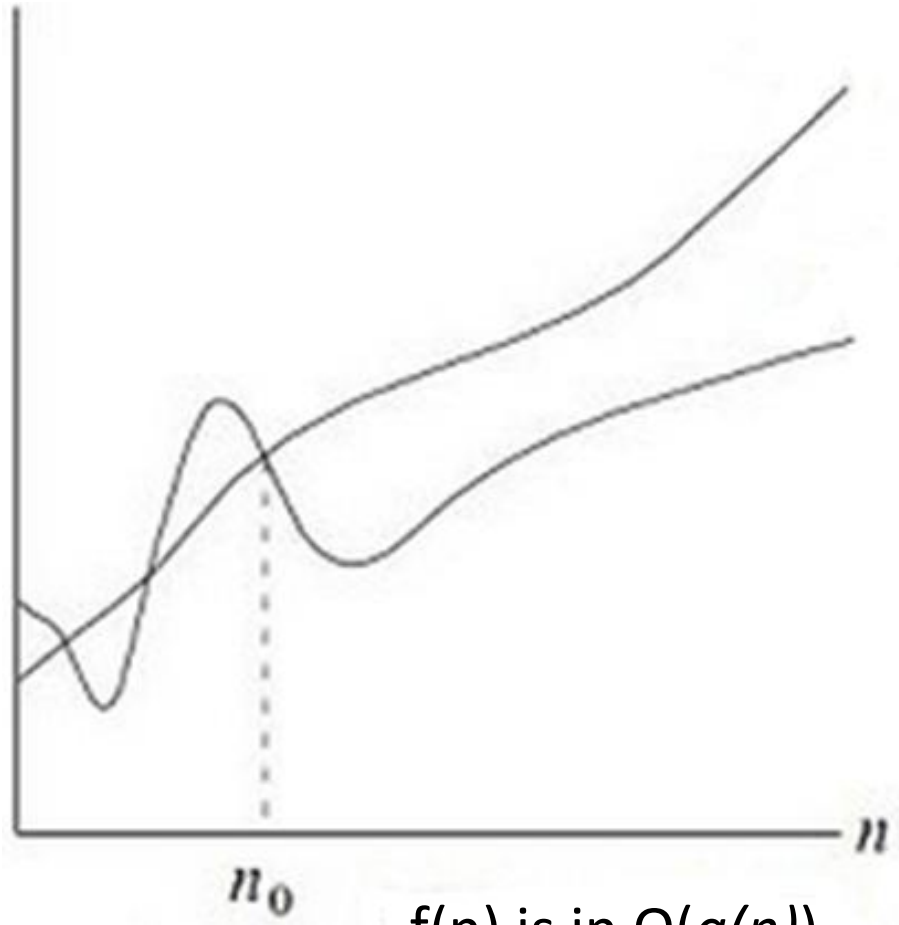
Example: $3n^2 + 90n - 5\log n + 6046 = \Theta(n^2)$

Ignore **Drop**

Big O

For $\mathbf{T}(n)$ a non-negatively valued function, $\mathbf{T}(n)$ is in set $\mathrm{O}(f(n))$ if there exist two positive constants c and n_0 such that $\mathbf{T}(n) \leq cf(n)$ for all $n > n_0$.

$$\lim_{n\to\infty} \frac{T(n)}{f(n)} \leq c$$



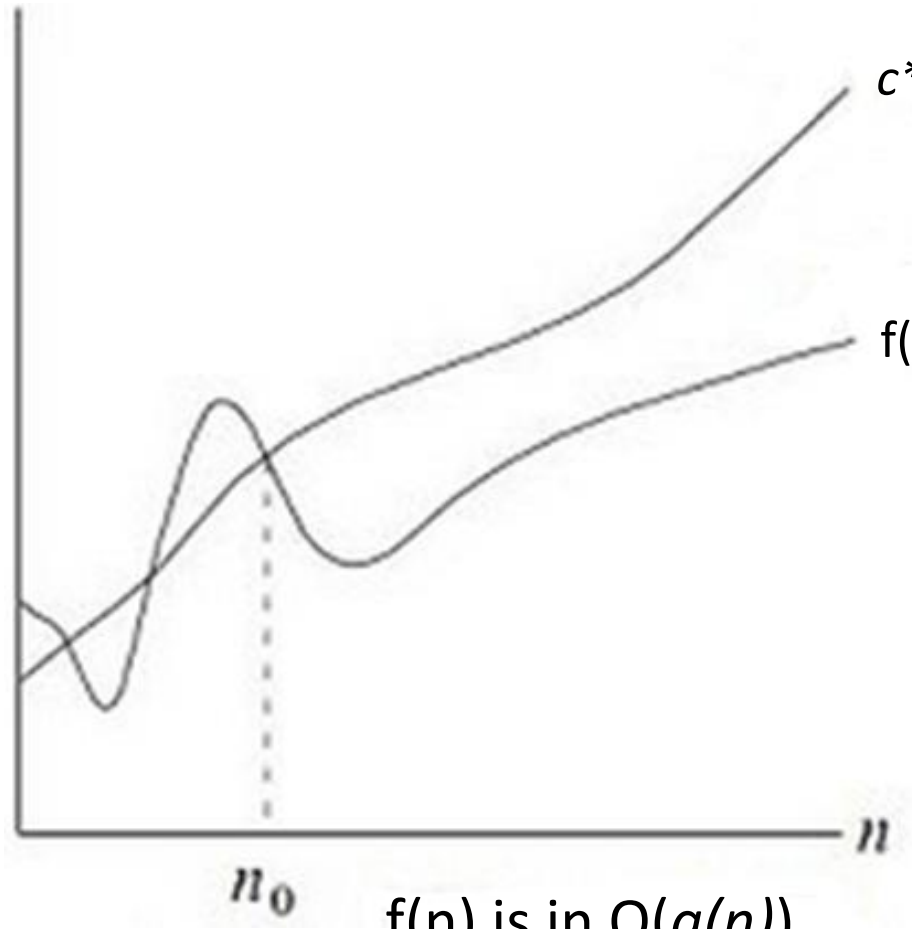
f(n) is in O(*g(n)*)

- 意义：对于问题的所有输入，只要输入规模足够大(即$n>n_0$)，该算法总能在*cf(n)*步以内完成。
- **算法的最高增长率。**

Big O

意义：对于问题的所有输入，只要输入规模足够大(即$n>n_0$)，该算法总能在$cf(n)$步以内完成。

算法的最高增长率。



f(n) is in O(*g(n)*)

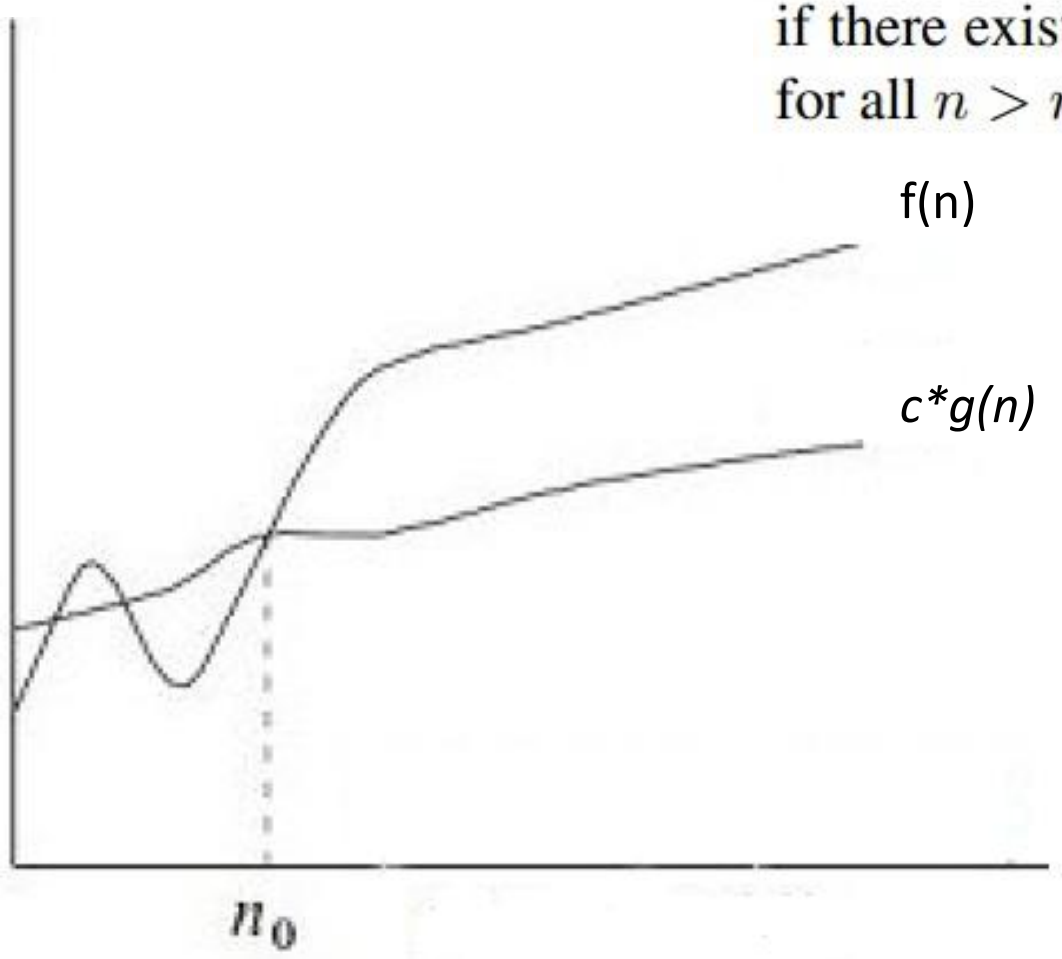
$3n^3 + 20n^2 + 5$ is $O(n^3)$

$3 \log n + 5$ is $O(\log n)$

Big Ω

For $\mathbf{T}(n)$ a non-negatively valued function, $\mathbf{T}(n)$ is in set $\Omega(g(n))$ if there exist two positive constants c and n_0 such that $\mathbf{T}(n) \geq cg(n)$ for all $n > n_0$.[1]

$$\lim_{n\to\infty}\frac{T(n)}{g(n)} \geq c$$



f(n) is in Ω(*g(n)*)

- 意义：对于问题的所有输入，只要输入规模足够大(即n>n0)，该算法至少需要cg(n)步以上才能 完成。
- **算法的最低增长率。**

Example for Θ, O and Ω

$$f(n) = 32n^2 + 17n + 32$$

- $f(n)$ is $O(n^2)$, $O(n^3)$
- $f(n)$ is $\Omega(n^2)$, $\Omega(n)$
- $f(n)$ is $\Theta(n^2)$

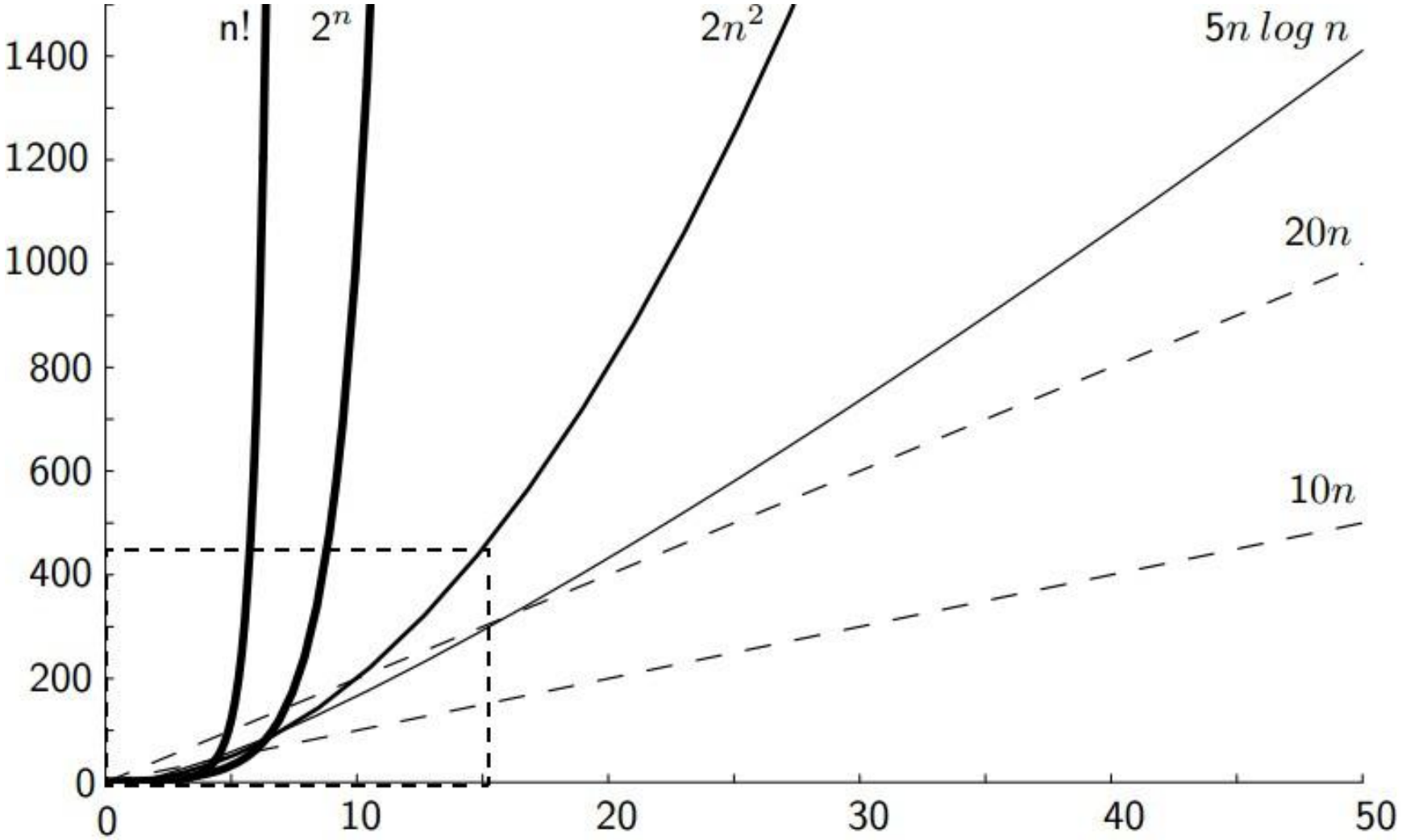
- $f(n)$ is not $O(n)$
- $f(n)$ is not $\Omega(n^3)$
- $f(n)$ is neither $\Theta(n)$ nor $\Theta(n^3)$

时间复杂性的度量

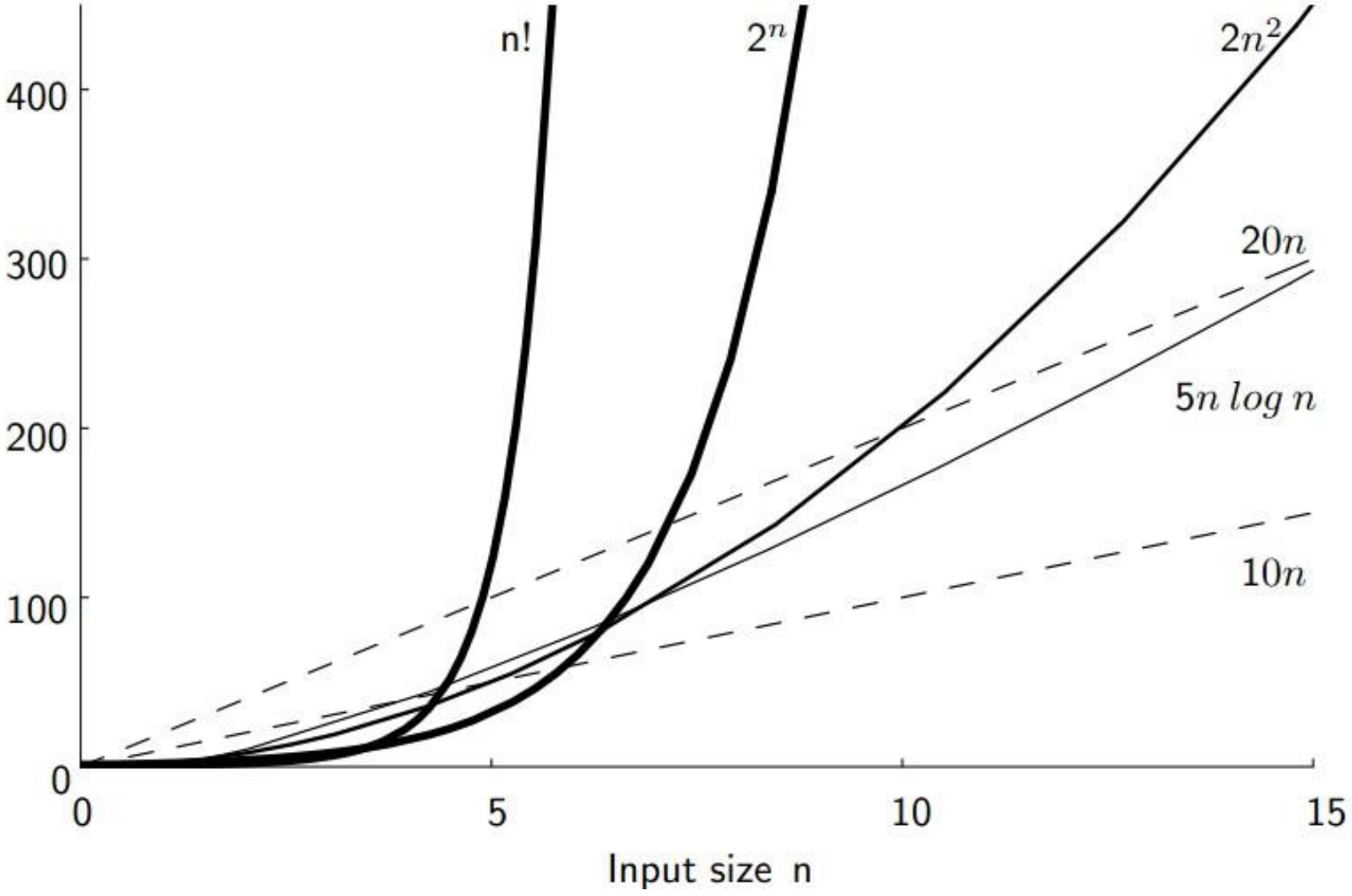
由O记号提供的渐近上界可能是渐近紧确的，也可能是非紧确的。（如：$2n^2 = O(n^2)$是渐近紧确的，而$2n = O(n^2)$是非紧确上界。）
例子：$f(n) = n^2 + n$，则$f(n) = o(n^3)$

例子：$f(n) = n^2 + n$，则$f(n) = \omega(n)$是正确的。$f(n) = \omega(n^2)$则是错误的，$f(n) = \Omega(n^2)$是正确的。

增长率


n!
2^n
$2n^2$
$5n\ log\ n$
$20n$
$10n$
1400
1200
1000
800
600
400
200
0
0
10
20
30
40
50

增长率



增长率

n	log log n	log n	n	n log n	n^2	n^3	2^n
16	2	4	2^4	$2 \cdot 2^4 = 2^5$	2^8	2^{12}	2^{16}
256	3	8	2^8	$8 \cdot 2^8 = 2^{11}$	2^{16}	2^{24}	2^{256}
1024	≈ 3.3	10	2^{10}	$10 \cdot 2^{10} \approx 2^{13}$	2^{20}	2^{30}	2^{1024}
64K	4	16	2^{16}	$16 \cdot 2^{16} = 2^{20}$	2^{32}	2^{48}	2^{64K}
1M	≈ 4.3	20	2^{20}	$20 \cdot 2^{20} \approx 2^{24}$	2^{40}	2^{60}	2^{1M}
1G	≈ 4.9	30	2^{30}	$30 \cdot 2^{30} \approx 2^{35}$	2^{60}	2^{90}	2^{1G}

Figure 3.2 Costs for growth rates representative of most computer algorithms.

更快的机器 or 算法?

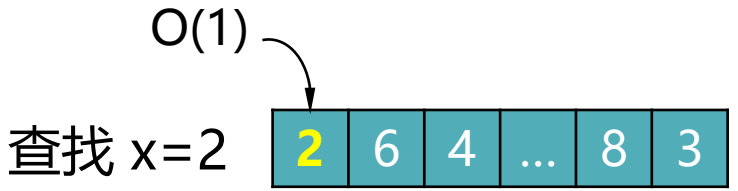
假设：旧机器一个小时能处理规模为n的数据。新机器比旧机器快10倍。则期望新机器能处理10倍大的数据规模。事实上取决于时间复杂度。

f(n)	n	n′	Change	n′/n
10n	1000	10,000	$n' = 10n$	10
20n	500	5000	$n' = 10n$	10
5n log n	250	1842	$\sqrt{10}n < n' < 10n$	7.37
$2n^2$	70	223	$n' = \sqrt{10}n$	3.16
2^n	13	16	$n' = n + 3$	−−

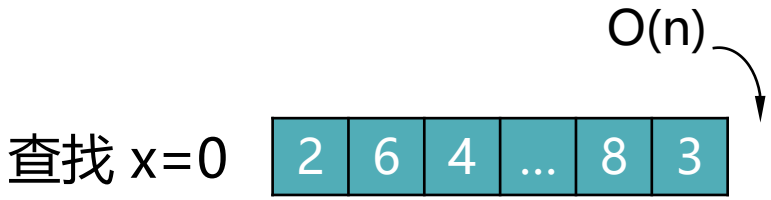
最好、最坏、平均情况时间复杂度

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。以该函数为例，要查找的变量 x 正好是数组的第一个元素，这时对应的是最好情况时间复杂度O(1)



最坏情况复杂度：这是算法可能遇到的最糟糕的情况的效率。在这种情况下，算法耗时最长。以该函数为例，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，这时对应的是最坏情况时间复杂度O(n)



平均情况复杂度：这是算法在所有可能输入的平均效率。在计算平均情况复杂度时，通常假设所有输入出现的概率符合特定的分布（最简单的可以假设所有输入为等可能）。对于函数F，其平均时间复杂度为O(n)

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

最好、最坏、平均情况时间复杂度

上限与最坏情况的区别：

-上限是用来确定运行时间的增长率，体现随着输入规模变化算法的代价变化

-最差情况是指：在一个给定的规模中，所有可能的输入中最糟糕的情况。

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：

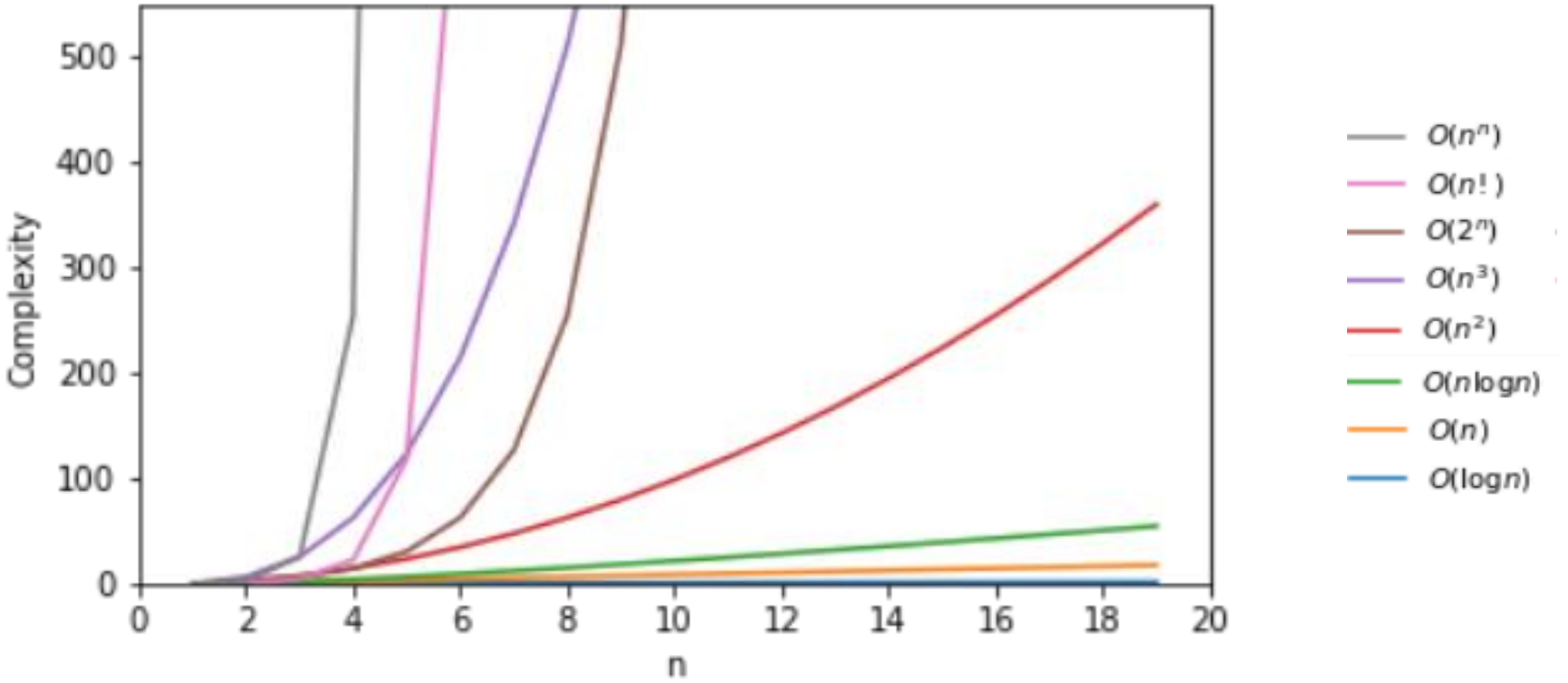
- 算法运行的瓶颈在于内存空间（以TB为单位）与外存空间（以TB为单位）的较大差异引起的
- 每一段程序代码在执行时，都需要将其代码和所需的数据装入内存。若所需空间大于现有内存，则会出现内存溢出、宕机等情况

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

算法分析示例

输入：数值a和b
输出：交换a和b的值

```
1. tmp ← a          O(1)
2. a ← b            O(1)
3. b ← tmp          O(1)
```

时间复杂度：O(1)+O(1)+O(1) = O(1)

输入：顺序表A，长度为n (n>0)
输出：顺序表中的最大值

```
1. largest ← 0  //设顺序表第一个元素是最大值          O(1)
2. for i ← 1 to n-1 do                                O(n)
3. |  if A[largest] < A[i] then                       O(1)
4. |  |  largest ← i    //更新最大值位置
5. |  end
6. end
7. return A[largest]                                  O(1)
```

时间复杂度：O(1)+O(n)*O(1)+O(1) = O(n)

算法分析示例

```
1. sum ← 0                                          O(1)
2. for i ← 1 to n do
3. |  for j ← 1 to i do                  O(i)
4. |  |  sum ← sum + i + j     O(1)                 O(n)
5. |  end
6. end
7. return sum                                       O(1)
```

- 内层循环（代码3-5行）的时间复杂度
  O(i)*O(1) = O(i)

- 外层的循环（代码2-6行）的时间复杂度

$$\sum_{i=1}^{n} O(i) = O\left(\sum_{i=1}^{n} i\right) = O\left(\frac{n(n-1)}{2}\right) = O(n^2)$$

- 总的时间复杂度
  $O(1) + O(n^2) + O(1) = O(n^2)$

单选题 1分

设置

程序（算法）的时间复杂度为：

A $O(1)$

B $O(n)$

C $O(n^2)$

D $O(n^3)$

```
1. for i ← 1 to n do
2. |  for j ← 1 to n do
3. |  |  for k ← 1 to n do
4. |  |  |  sum ← i + j + k
5. |  |  end
6. |  end
7. end
```

提交

单选题 1分

设置

程序（算法）的时间复杂度为：

A $O(1)$

B $O(\sqrt{n})$

C $O(n)$

D $O(\mathrm{n}^2)$

```
1. i ← 0
2. s ← 1
3. while s < n do
4. | s ← s + i
5. | i ← i + 1
6. end
```

提交

单选题 1分

设置

程序（算法）的时间复杂度为：

A $O(1)$

B $O(\mathrm{n})$

C $O(\mathrm{n}^2)$

D $O(\mathrm{n}^3)$

```
1.  if n < 100 then
2.  |  for i ← 1 to n do
3.  |  |  for j ← 1 to n do
4.  |  |  |  sum ← i + j
5.  |  |  end
6.  |  end
7.  else    // n ≥ 100
8.  |  sum ← 2*n + 1000
9.  end
10. return sum
```

提交

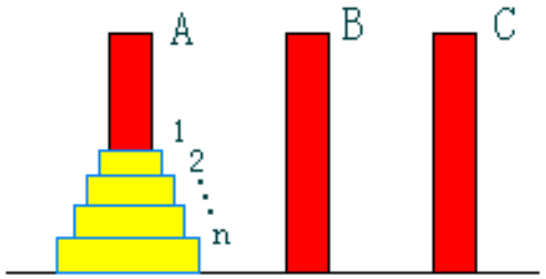
算法分析示例

设A,B,C是3个塔座。开始时，在塔座A上有一叠共n个圆盘，这些圆盘自下而上，由大到小地叠在一起。现要求将塔座A上的这一叠圆盘移到塔座C上，并仍按同样顺序叠置。在移动圆盘时应遵守以下移动规则：

规则1：每次只能移动1个圆盘；

规则2：任何时刻都不允许将较大的圆盘压在较小的圆盘之上；

规则3：在满足移动规则1和2的前提下，可将圆盘移至A, B, C中任一塔座上。



Hanoi塔问题

算法：hanoi(n, A, B, C)

输入：塔座A、B、C，A塔上有n个圆盘（n>0）
输出：把圆盘按规则全部移动至C塔

```
1. if n > 0 then    //圆盘数为0，则不移动 （边界条件）
2. | hanoi(n-1, A, C, B)    //A塔从上至下n-1个圆盘移到B塔
3. | Move(A, B)             //A塔第n个盘子移到B塔
4. | hanoi(n-1, B, A, C)    //B塔从上至下n-1个圆盘移到C塔
5. end
```

递归算法

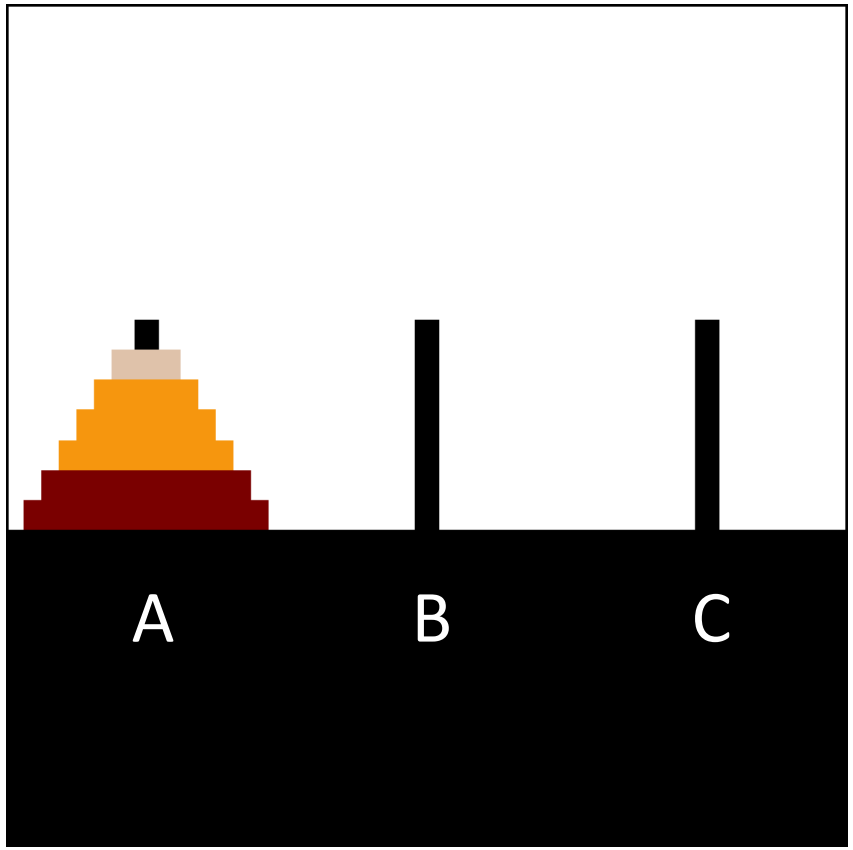
算法分析示例

Hanoi塔问题

算法：hanoi(n, A, B, C)

输入：塔座A、B、C，A塔上有n个圆盘（n>0）
输出：把圆盘按规则移动至B塔

```
1. if n > 0 then    //圆盘数为0，则不移动 （边界条件）
2. | hanoi(n-1, A, C, B)    //A塔从上至下n-1个圆盘移到C塔
3. | Move(A, B)             //A塔第n个盘子移到B塔
4. | hanoi(n-1, C, B, A)    //C塔从上至下n-1个圆盘移到B塔
5. end
```



移动过程

算法分析示例

算法：hanoi(n, A, B, C)

输入：塔座A、B、C，A塔上有n个圆盘（n>0)
输出：把圆盘按规则移动至B塔

```
1. if n > 0 then    //圆盘数为0，则不移动 （边界条件）        O(1)
2. | hanoi(n-1, A, C, B)   //A塔从上至下n-1个圆盘移到C塔     T(n-1)
3. | Move(A, B)            //A塔第n个盘子移到B塔             O(1)
4. | hanoi(n-1, C, B, A)   //C塔从上至下n-1个圆盘移到B塔     T(n-1)
5. end
```

设移动n个圆盘的时间为T(n)

$$T(n)=\begin{cases}2T(n-1)+O(1), & n>0\\ O(1), & n=0\end{cases}$$

$$T(n)=O(2^n)$$

算法优化 – 时间复杂度

例：给定n 个整数（可以为负数）的序列

$$(s_1, s_2, \dots, s_n)$$

求

$$\max\{0, \max_{1\le i\le j\le n} \sum_{k=i}^{j} s_k\}$$

连续子序列最大和问题

实例： (-2, 11, -4, 13, -5, -2)

解：最大子段和 $s_2+s_3+s_4= 20$

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，长度为n，s[i]∈整数集
输出：序列s中最大的连续子序列之和max_sum

```
1.  max_sum ← 0                  //设置最大子序列和初值
2.  for i←1 to n do              //子序列起始位置
3.  | for j←i to n do            //子序列结束位置
4.  | | this_sum ← 0             //设置子序列和初值
5.  | | for k←i to j do          //求子序列和
6.  | | | this_sum ← this_sum+s[k]
7.  | | end
8.  | | if this_sum > max_sum then //如当前子序列和更大
9.  | | | max_sum ← this_sum   //设置当前最大子序列和
10. | | | start ← i            //设置当前最大子序列起始位置
11. | | | finish ← j           //设置当前最大子序列结束位置
12. | | end
13. | end
14. end
```

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
1.  s.max_sum ← 0              //设置最大子序列和初值
2.  for i←1 to s.n do          //子序列起始位置
3.  | for j←i to s.n do        //子序列结束位置
4.  | | this_sum ← 0           //设置子序列和初值
5.  | | for k←i to j do        //求子序列和
6.  | | | this_sum ← this_sum+s.array[k]
7.  | | end
8.  | | if this_sum>s.max_sum than //如当前子序列和更大
9.  | | | s.max_sum ← this_sum   //设置当前最大子序列和
10. | | | s.start ← I          //设置当前最大子序列起始位置
11. | | | s.finish ← j         //设置当前最大子序列结束位置
12. | | end
13. | end
14. end
```

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
1.  s.max_sum ← 0              //设置最大子序列和初值
2.  for i←1 to s.n do          //子序列起始位置
3.  | this_sum ← 0             //设置子序列和初值
4.  | for j←i to s.n do        //子序列结束位置
5.  | |  this_sum ← this_sum+s.array[j]
6.  | end
7.  | if this_sum>s.max_sum than //如当前子序列和更大
8.  | | s.max_sum ← this_sum   //设置当前最大子序列和
9.  | | s.start ← i           //设置当前最大子序列起始位置
10. | | s.finish ← j          //设置当前最大子序列结束位置
11. | end
12.end
```

算法优化 – 时间复杂度

例：给定n 个整数（可以为负数）的序列

$$(s_1, s_2, \cdots, s_n)$$

求

$$\max\{0, \max_{1\le i\le j\le n} \sum_{k=i}^{j} s_k\}$$

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，长度为n，s[i]∈整数集
输出：序列s中最大的连续子序列之和max_sum
1.  max_sum ← 0                    //设置最大子序列和初值
2.  this_sum ← 0                   //设置当前子序列和初值
3.  this_start ← 1      //设置子序列开始位置
4.  for i ←1 to n do
5.  | this_sum ← this_sum+s[i]
6.  | if this_sum>max_sum then //如当前子序列和更大
7.  | |  max_sum ← this_sum   //设置当前最大子序列和
8.  | |  start ← this_start  //设置当前最大子段开始位置
9.  | |  finish ← i     //设置当前最大子序列结束位置
10. | else if this_sum < 0 then  //如子序列和小于0
11. | | this_sum ← 0             //重新计算子序列和
12. | | this_start ← i+1         //从上个子序列结束后开始
13. | end
14. end
```

算法优化 – 时间复杂度

假设： n 个整数的序列 （简单起见，不含0）

$(s_1, s_2, \cdots, s_n)$ 中，**和最大子段为:**

$s_i, s_{i+1}, \cdots, s_{j-1}, s_j \quad (1 \le i \le j \le n)$

问1：$s_{i-1}, s_i, s_j, s_{j+1}$ 这四个值须满足什么条件?

答： $s_{i-1} < 0, s_i > 0, s_j > 0, s_{j+1} < 0$

问2：和最大子段的所有前缀和，即

s_i

$s_i + s_{i+1}$

$s_i + s_{i+1} + s_{i+2}$

…

$s_i + s_{i+1} + \cdots + s_j$ 须满足什么条件?

实例： (-2, 11, -4, 13, -5, -2)

解：最大子段和 $s_2+s_3+s_4= 20$

答：所有前缀和大于等于0!

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，长度为n，s[i]∈整数集
输出：序列s中最大的连续子序列之和max_sum
1.  max_sum ← 0                    //设置最大子序列和初值
2.  this_sum ← 0                   //设置当前子序列和初值
3.  start ← 1                      //设置子序列开始位置
4.  for i ←1 to n do
5.  | this_sum ← this_sum+s[i]
6.  | if this_sum>max_sum then //如当前子序列和更大
7.  | |  max_sum ← this_sum   //设置当前最大子序列和
8.  | |  finish ← i    //设置当前最大子序列结束位置
9.  | else if this_sum < 0 then  //如子序列和小于0
10. | |  this_sum ← 0           //重新计算子序列和
11. | |  start ← i+1            //从上个子序列结束后开始
12. | end
13. 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。

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

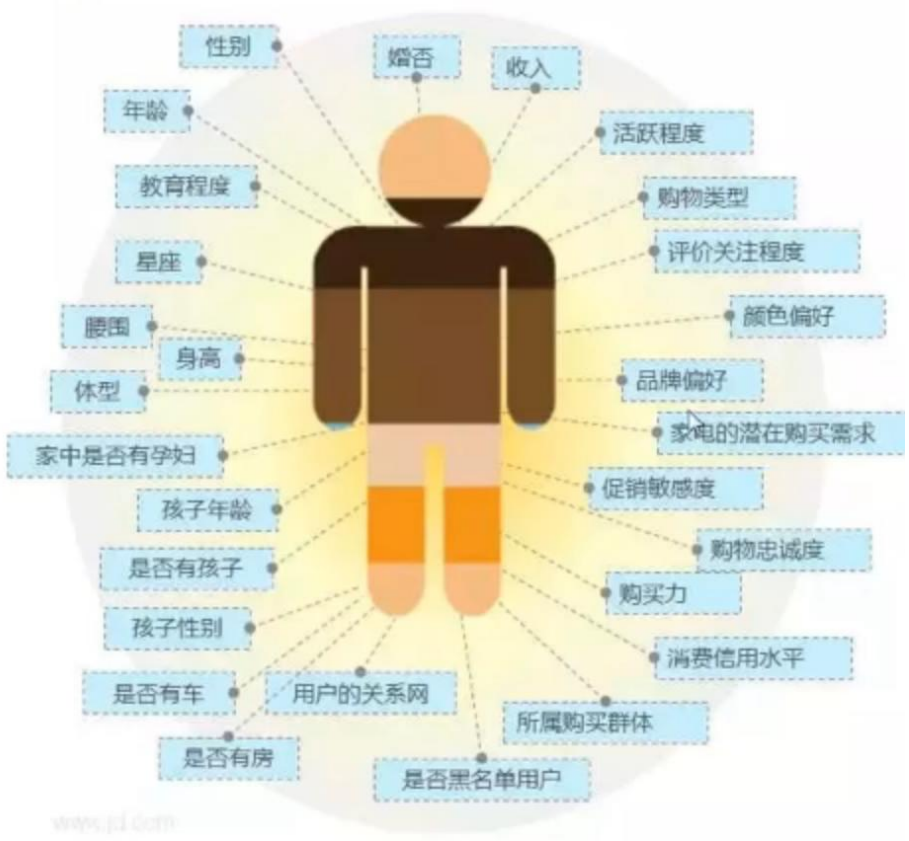
```
1. for i←1 to n do
2. | print (i)
3. end
```

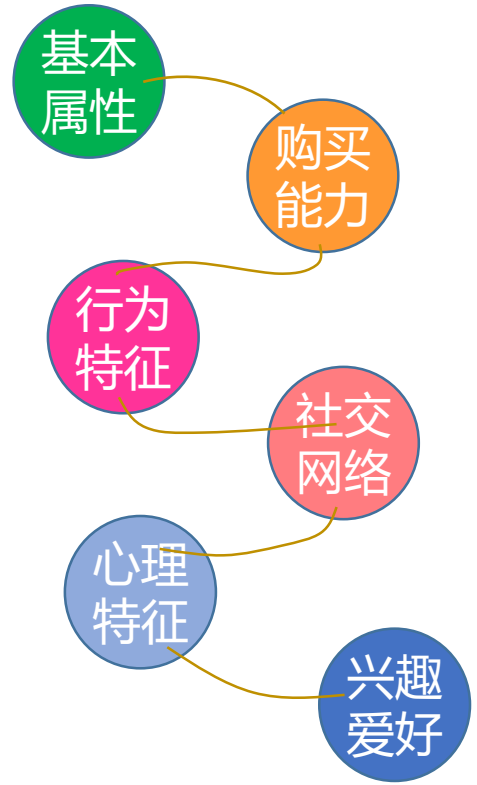
递归算法：直到n=0开始返回上一层，并从1开始输出，一直到最后打印n。该函数通常只能执行数万次，就会因递归层数过多，系统栈空间不足而报错，也称**递归爆栈**。因为递归时，每次进入更深一层，都需要将当前空间的状态进行存储，消耗一定的内存空间。

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。

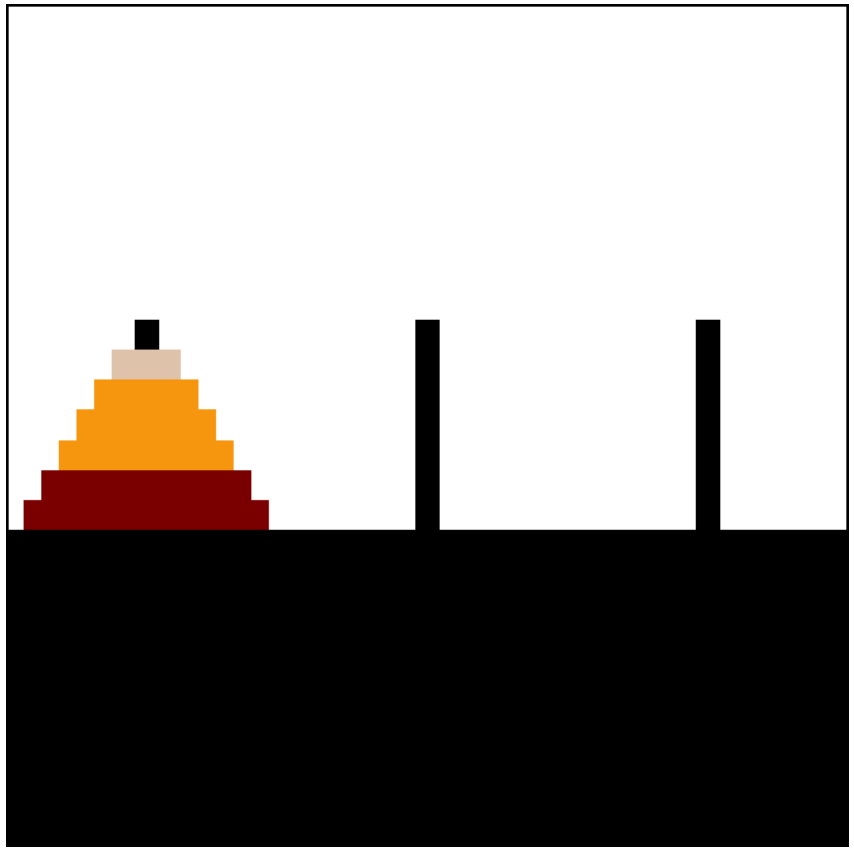
1.5 应用场景：数据挖掘

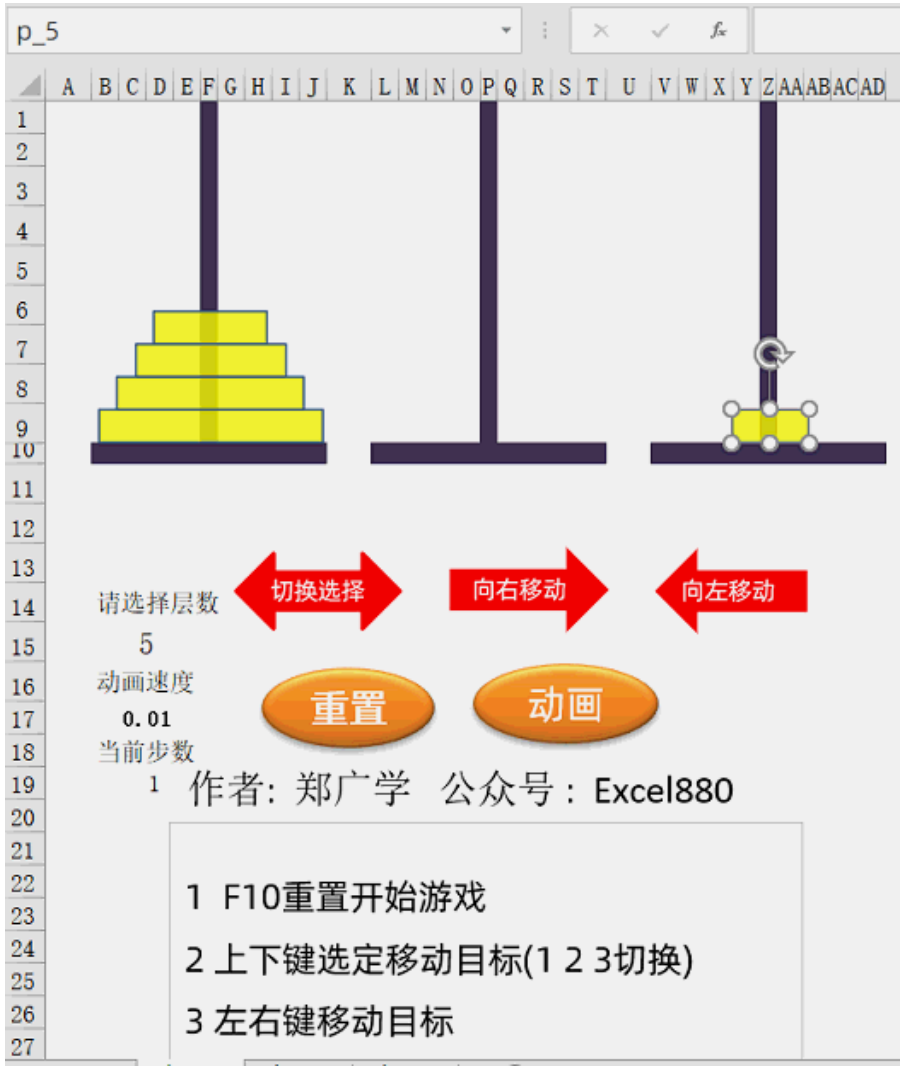
用户画像：基于用户线上行为数据抽象出他/她的信息全貌





汉诺塔游戏




p_5
请选择层数
5
动画速度
0.01
当前步数
1
切换选择
向右移动
向左移动
重置
动画
作者：郑广学 公众号：Excel880
1 F10重置开始游戏
2 上下键选定移动目标(1 2 3切换)
3 左右键移动目标

1.6 习题

第一题：请参照本章1.1节中的“大型超市”的例子，寻找日常生活中场景及发现需要解决的问题，并按问题求解的过程，叙述问题求解的三个步骤。场景如: 在现实场景中，我们如何计算一个人一年的总收入?

答：第一步，问题分析：
首先需要将问题抽象为一个简单的计算问题。要想计算一个人一年的总收入，需要定义以下数据和操作：(1) 收入的类型和金额，如工资、股息、租金等；(2) 每种收入的时间段和频率，如月收入、年收入等；(3) 税收和其他扣除项，如社保、医保、住房公积金等；(4) 计算总收入的方法，在本例中计算方法为累加每种收入并扣除税收和其他扣除项。

- 第二步，存储实现和算法设计：

在本例中，可以使用顺序存储结构存储每种收入和扣除项的金额和时间信息。算法步骤如下：
1. 对于每种收入和扣除项，计算其总金额；
2. 对于每种收入，根据时间段和频率计算一年的总收入；
3. 对于每种扣除项，累加其金额，并扣除总收入；
4. 计算总收入并减去税收和其他扣除项，得到净收入。

- 第三步，程序实现：

可以使用Python编写程序实现算法。通过输入每种收入和扣除项的金额和时间信息，计算一个人一年的总收入和净收入。此外，可以根据需要对数据进行处理、存储和可视化，以便更好地理解和分析结果。

1.6 习题

第二题：数据结构研究的主要内容是什么？

答：数据结构研究的是如何组织和管理数据，以便更高效地进行存储、检索、修改和删除等操作。具体来说，数据结构的主要内容包括基本数据结构、高级数据结构、算法设计和分析、抽象数据类型和数据结构的应用。

第三题：根据数据元素之间的不同逻辑关系，通常将其划分为哪几类结构？

答：集合、线性结构、树形结构、图形结构。

1.6 习题

第四题：试说明下列数据集合中元素的逻辑关系。

（1）超市收银台排队等候结账的乘客
（2）游乐园里的游客
（3）个人通信录中的人员
（4）电脑里的文件夹

答：

（1）一对一的关系（线性结构）

（2）无关系（集合）

（3）无关系（集合）

（4）一对多的关系（树）

1.6 习题

第五题：什么是存储实现？什么是运算实现？

答：

- 存储实现指的是如何在计算机内存中存储数据结构，以便更高效地对其进行操作。不同的数据结构有不同的存储实现方式，如数组、链表、栈和队列等。存储实现的选择会直接影响数据结构的操作效率和空间占用。
- 运算实现指的是如何实现数据结构的各种操作，如插入、删除、查找和排序等。不同的数据结构有不同的运算实现方式，如遍历算法、递归算法和分治算法等。运算实现的选择会直接影响数据结构的操作效率和时间复杂度。

1.6 习题

第六题：何谓算法的时间、空间复杂度？如何度量算法的时间、空间复杂度？

答：

- 算法的时间复杂度指的是算法所需时间随问题规模增长的增长率。度量算法的时间复杂度可以通过分析算法的基本操作执行次数来实现。例如，对于一个循环结构，其时间复杂度可以用循环次数来表示；对于一个递归算法，其时间复杂度可以用递归深度和每次递归的时间复杂度来表示。通过对基本操作执行次数的分析，可以得到算法的时间复杂度。
- 算法的空间复杂度指的是算法所需的额外空间随问题规模增长的增长率。度量算法的空间复杂度可以通过分析算法的数据结构和变量使用情况来实现。例如，对于一个数组或链表，其空间复杂度可以用元素个数来表示；对于一个递归算法，其空间复杂度可以用递归深度和每次递归所需的空间复杂度来表示。通过对数据结构和变量使用情况的分析，可以得到算法的空间复杂度。

1.6 习题

第七题：什么是最好、平均和最坏情况下的时间复杂度?

答:

- 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。
- 最坏情况复杂度：算法可能遇到的最糟糕的情况的效率。
- 平均情况复杂度：算法在所有可能输入的平均效率。

第八题：什么是多项式时间算法？什么是指数时间算法?

答:

- 多项式时间算法指的是算法的时间复杂度是多项式级别的，即算法的时间复杂度可以表示为一个多项式函数，即$O\left(n^k\right)\ (k \geq 0)$。
- 指数时间算法指的是算法的时间复杂度是指数级别的，即算法的时间复杂度可以表示为一个指数函数，即$O(k^n)\ (k > 1)$ 。

1.6 习题

第九题：按增长率排列下列函数：

$n, \sqrt{n}, n^{1.5}, n^2,\ n\log(\mathrm{n}), \mathrm{nloglog(n)}, \mathrm{nlog^2(n)}, \mathrm{nlog(n^2)}, \frac{2}{\mathrm{n}}, 2^n, 37, n^2\log(n), n^3$

指出那些函数以相同的增长率增长

答：按增长率从小到大排序：

$$\frac{2}{n}, 37, \sqrt{n}, n, \mathrm{nloglog(n)}, n\log(\mathrm{n}), \mathrm{nlog(n^2)}, \mathrm{nlog^2(n)}, n^{1.5}, n^2, n^2\log(n), n^3, 2^n$$

增长率相同

1.6 习题

第十题：证明在计算大O表示法表示的时间复杂度时，取运行时间函数的主项是正确的。

答：

- 假设一个算法的运行时间函数为$T(n) = a_k n^k + a_{k-1} n^{k-1} + \cdots + a_1 n + a_0 \ (a_k > 0)$。
- 大O表示法的定义：当n趋于无穷大时，T(n)的增长率不超过g(n)的增长率，即存在一个正常数c和正整数n0，使得对于所有的n≥n0，都有T(n) ≤ c*g(n)。
- 取$g(n) = n^k$，因为T(n)的主项是$a_k n^k$，是n^k的常数倍，而其他项对于趋向无穷大的n来说都可以忽略。因此，存在一个正常数c和正整数n_0，使得对于所有的$n \geq n_0$，都有$T(n) \leq cn^k$。因此，T(n)的时间复杂度为$O(n^k)$。由此可见，取运行时间函数的主项是正确的。

1.6 习题

第十二题：请分析以下算法的时间复杂度。

答：$O(\log n)$

算法：BinarySearch(array, x)

输入：整数数组array，待查找的整数 x
输出：查询x在array 中的位置，输出其下标

```
n ← array.size  //数组长度
left ← 0 //执行一次
right ← n-1 //执行一次
while left ≤ right do   //循环持续到left>right为止
|  middle ← (left +right)/2
|  if  x = array[middle]  then
|  |  return middle
|  else if x < array[middle] then
|  |  right ← middle -1
|  else // x > array[middle]
|  |  left ← middle +1
|  end
end
return NIL
```

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