程序设计与算法基础2
数据结构与算法

主讲教师： 刘峤

课程说明

- 课程性质：学科基础课　　学分：**4**
- 总学时：**64**　（48学时授课，16学时上机）
- 上课时间地点：
  - 2019年春季 **1 - 16** 周
  - 周三**第5、6节**，品学楼（**C-404**）
  - 周五**第5、6节**，品学楼（**C-404**）
- 课程序号：　E0902040.01

课程说明

- **成绩构成：**
  - 平时成绩：**20%**（作业、考勤等）
  - 上机实验：**20%**（独立完成上机作业）
  - 期末考试：**60%** （考试：闭卷、笔试）
- **联系方式：**
  - 办公室：沙河主楼422
  - 助教：（硕士） 邮箱：**@qq.com**
  - 课程QQ群：

教材与参考书推荐

- **课程教材**
  - 《**数据结构教程**》（**第5版**）
    - 陈春葆主编，清华大学出版社
- **参考书推荐**
  - ***Data Structures and Algorithm Analysis in C***
    - Mark A. Weiss 著(2nd)，Addison Wesley Press
  - 《**数据结构：C语言版**》
    - 严蔚敏等著，清华大学出版社，2007

课程概览

- **《程序设计与算法基础 II》课程讨论什么?**
  - 如何在计算机中**表示**问题和实现对问题的**求解**
- **课程目标**
  - 培养**计算思维**，提高解决问题的**能力**
- **课程重要性**
  - 计算机专业的核心基础课
  - 数据结构与算法是一切程序设计的基础

第1章 绪论

学习数据结构的意义及要求

Data Structure + Algorithm = Program

是瑞士苏黎世大学著名的计算机科学家、Pascal程序设计语言之父、结构化程序设计首创者、1984年**图灵奖**获得者沃斯(Niklaus Wirth)于1976年提出的

图灵奖是什么？

公式的含义：

- **数据结构**和**算法**是构成计算机**程序**的两个关键要素
- 程序设计的精髓在于设计算法和相应的数据结构

所谓计算机程序，就是使用计算机程序设计语言描述算法和数据结构，从而在计算机上实现应用问题的求解

知识链接：图灵奖

图灵奖

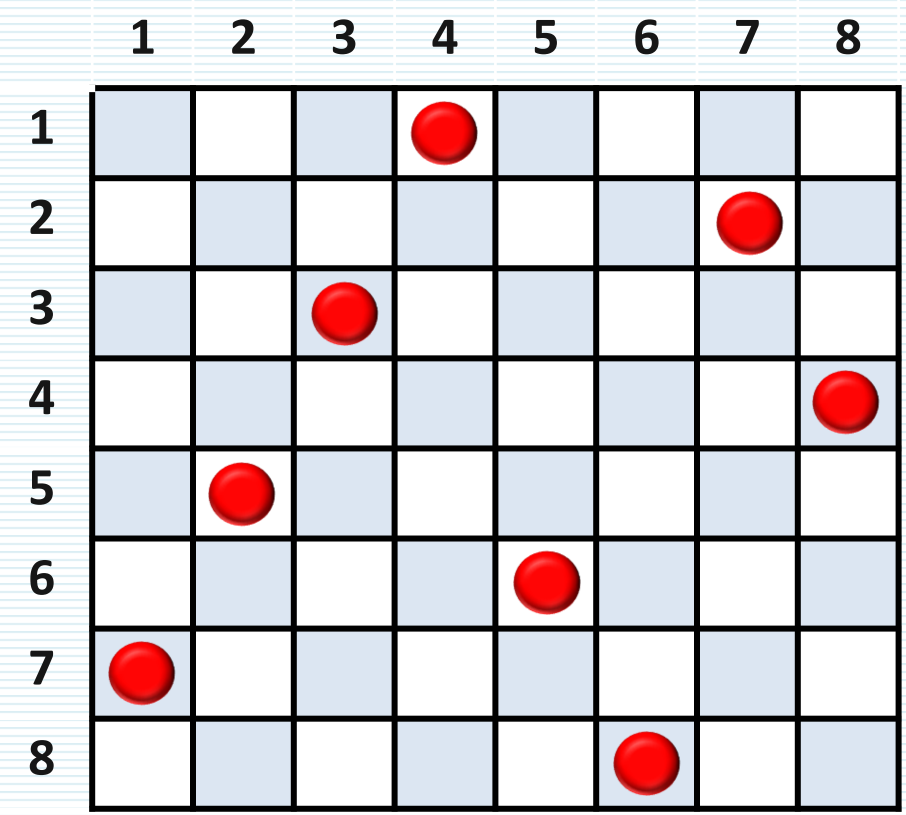
- 图灵奖是美国计算机协会于1966年设立的
- 其名称取自英国科学家**阿兰·图灵**
- 获奖者的贡献必须在计算机领域具有**持久而重大的影响**
- 有“计算机界诺贝尔奖”之称
- 目前由英特尔公司和google公司赞助，奖金为$250,000

1.1 学习数据结构与算法的意义

学习数据结构与算法的意义

八皇后问题（高斯，1850）

- 在8×8的国际象棋棋盘上摆放8个皇后，使其不能互相攻击
- 即：任意两个皇后不能同行同列或同斜线，问有多少种摆法



8-皇后问题

问题分析

- 问题的解向量：$(x_1, x_2, \ldots, x_8)$
  - 采用数组下标 i 表示皇后所在的行号
  - 采用数组元素 x[i] 表示皇后 i 的列号
- 约束条件
  - 显约束（对解向量的直接约束）：$x_i = 1, 2, \ldots, n$
  - 隐约束1：任意两个皇后不同列：$x_i \neq x_j$
  - 隐约束2：任意两个皇后不处于同一对角线?
    - $|i-j| \neq |x_i - x_j|$

8-皇后问题

```
bool Bound(int k){
    for (int i = 1; i < k; i++){
        if ((abs(k-i)==abs(x[k]-x[i]))||(x[i]==x[k]))
            return false;
    }
    return true;
}

void Backtrack(int t){
    if (t > 8) output(x);
    else {
        for (int i = 1; i <= 8; i++) {
            x[t] = i;
            if (Bound(t)) Backtrack(t+1);
        }
    }
}
```

1.2 数据结构的基本概念

什么是数据（data）

- 数据是信息的载体
  - 是描述客观事物的数和字符
  - 以及所有能被计算机程序识别和处理的符号的集合
- 数据共分为两类
  - 数值性数据（用于数学计算）
  - 非数值性数据（文字、图像、音视频等）
- 与数据相关的几个概念
  - 数据元素与数据项
  - 数据与数据对象
  - 数据与数据结构

数据元素与数据对象

- **数据元素**（data element）是数据的基本单位
  - 在计算机程序中常作为一个整体进行考虑和处理
  - 例如：图书馆的图书卡片（页面）
  - 在不同场合下数据元素又称为元素、结点、记录
- 数据元素可以由若干**数据项**（data item）组成
  - 数据项是数据不可分割的最小单位
  - 例如：图书卡片的组成要素（书名、作者等信息）
- **数据对象**（data object）
  - 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Z={0, ±1, ± 2, …}

数据结构

- 数据结构：相互间存在关系的数据元素的集合
  - 所谓结构就是数据元素之间的关系
  - 即：描述数据元素之间的运算及运算规则
- 用集合的形式描述，数据结构是一个二元组：

DS = (D，R)

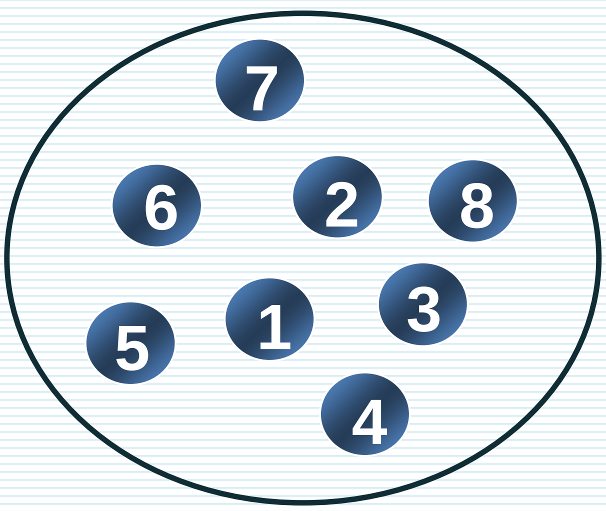
  - 其中：D是数据元素的集合，R是D上关系的集合
- 简言之：数据对象及其成员间的关系合称为数据结构
- 数据结构是数据在计算机中存在（表现）的形式

数据结构

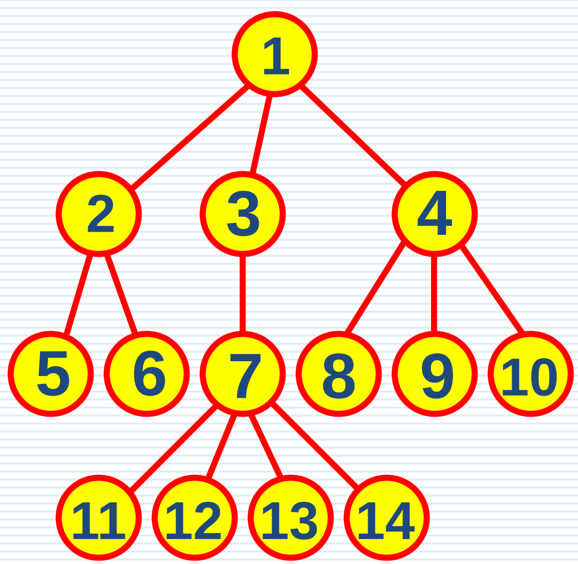
- 数据结构可进一步分为逻辑的和物理的
- 逻辑结构反映数据之间的逻辑关系
  - 包括：集合结构、线性结构、树形结构、图形结构
- 物理结构反映数据在计算机内部的存储安排
  - 分为：顺序存储结构和链式存储结构
  - 顺序存储结构：逻辑上相邻的元素存储位置也相邻
  - 链式存储结构：逻辑上相邻的元素存储位置可以不相邻
  - 散列存储结构：数据以键-值对（key : value）的方式存储，通过对关键字直接计算得到数据元素的存储位置

数据的逻辑结构

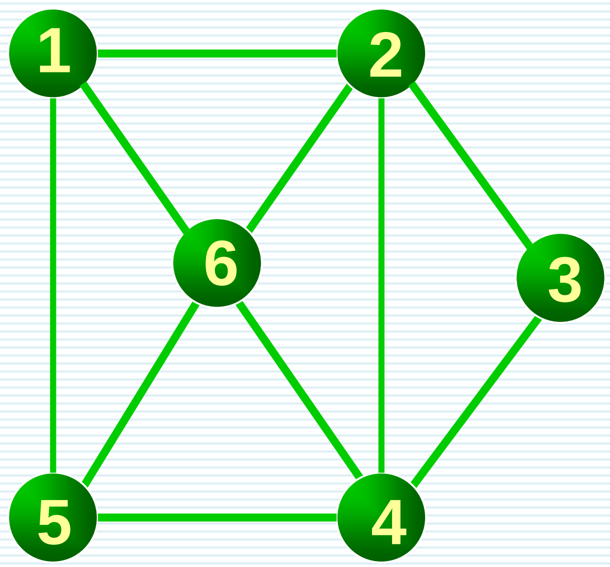
线性结构



集合结构



树形结构



图形结构

常用数据结构的逻辑划分

- 线性结构
  - 直接存取类：**数组，文件**
  - 顺序存取类：**表，栈，队列，优先队列**
- 非线性结构
  - 层次结构类：**树，二叉树，堆**
  - 网状结构类：**集合，图**

1.3 算法

算法的概念

- 算法（Algorithm)
  - 是解题的步骤，是指令的有限序列
  - 规定了解决某一特定类型问题的一系列运算
  - 是对解题方案的准确与完整的描述
- 算法的基本特征
  - **有穷性、确定性、输入、输出、可行性**
- 算法设计的一般过程
  - 设计、确认、分析、编码、测试、调试

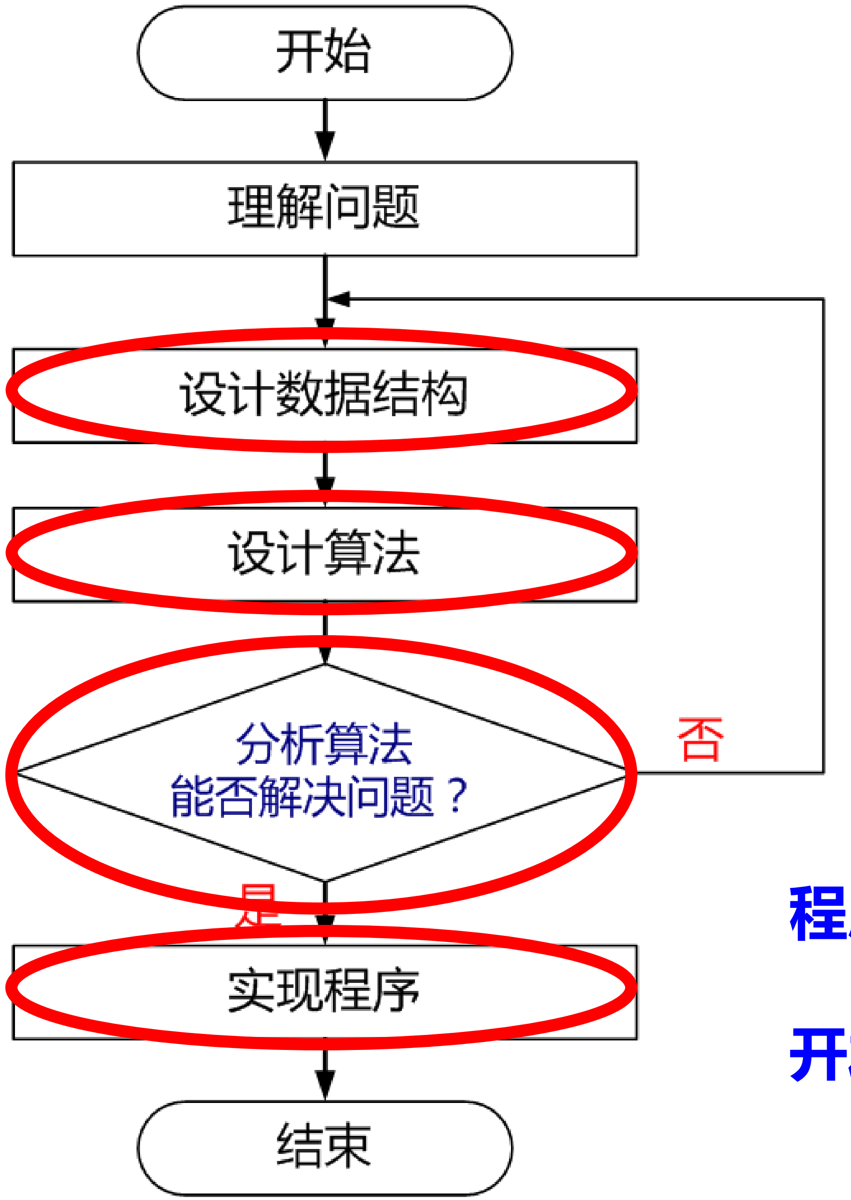
采用计算机求解问题的过程

数据结构：
算法设计的关键

算法：程序的灵魂

算法和数据结构设计是程序设计的核心关键环节

对算法性能起决定性作用



程序设计语言
+
开发调试工具

算法的概念

- 算法描述
  - 可以用自然语言，框图、程序设计语言等多种方式来描述
- 算法分析的内涵
  - 正确性、可读性、健壮性、高效性
  - 健壮性：算法在异常情况下性能表现
    - 好的算法在出现异常或用户操作不当时均能作适当处理
  - 高效性：求解问题所占用的存储空间少，执行时间短
    - 算法性能的度量：时间复杂度和空间复杂度

算法运行性能评价

- 事后统计
  - 利用计算机的时钟对程序的运行时间进行计时
- 事前分析估算
  - 用高级语言编写的程序运行的时间主要取决于如下因素
    - 问题的规模
    - 算法复杂度：时间复杂度和空间复杂度
    - 编程语言：一般情况下语言级别越高，效率越低；
    - 编译程序：指令优化的能力
    - 机器性能

思考题

- 算法应具有什么特征?
  - 有穷、确定、输入、输出、可行
- 衡量算法性能的指标有哪些?
  - 时间复杂度、空间复杂度

1.4 算法复杂度分析

算法复杂度分析

- 算法复杂度分析是指对一个算法所需要的资源进行预测
  - 通常我们关注的是时间复杂度和空间复杂度
- 算法复杂度分析采用的计算模型：单处理器RAM模型
  - RAM（Random-Access Machine）
  - RAM模型包含了真实计算机中的常见指令
    - **算术指令：加、减、乘、除、求余、取整**
    - **数据移动指令：装入、存储、复制**
    - **控制指令：条件和非条件转移、子程序调用和返回指令**
  - 其中每条指令执行所需的时间为常量
  - 指令一条接着一条顺序执行，没有并发操作

算法复杂度分析

- 时间复杂度的分析方法
  - 确定算法中（对于研究的问题而言）的基本操作
  - 以该基本操作重复执行的次数作为算法执行的时间度量
- 时间复杂度分析方法示例
  - **for (i = 1; i<=n ; i++) x = x + 1 ;**
    - 基本操作重复执行的次数为 **n** 次
  - **for (i = 1; i<=n ; i++)**

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

    x = x + 1 ;
    - 基本操作重复执行的次数为 n^2 次

算法复杂度分析

- 算法时间复杂度分析的一般方法
  - 设算法的问题规模为n；
  - 语句重复执行的次数称为该语句的**频度**：记为 $f(n)$
  - 对算法各基本操作的频度求和，即得到算法的时间复杂度
  - 在实际工作中我们所关心的是算法所需执行时间的**数量级**
    - 即：算法各基本操作频度最大值的数量级
  - 因此将算法的时间复杂度记为: $T(n)=O(f(n))$
    - 设：$f(n) = 1 + n + n^2 + n^3$
    - 则：$T(n) = O(n^3)$
- 算法空间复杂度分析方法与之类似：$S(n)=O(f(n))$

示例：矩阵相乘

```
// 以二维数组存储矩阵元素，c 为 a 和 b 的乘积
void mat_multi(int a[], int b[], int& c[]){
    for (i=1; i<=n; ++i){      ……………………………n+1
        for (j=1; j<=n; ++j){  ……………………n(n+1)
            c[i,j] = 0;        ……………………………n²
            for (k=1; k<=n; ++k) {  …………n²(n+1)
                c[i,j] += a[i,k]*b[k,j];  ………n³
            }
        }
    }
}
```

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

时间复杂度：$O(n^3)$

算法的渐进复杂度

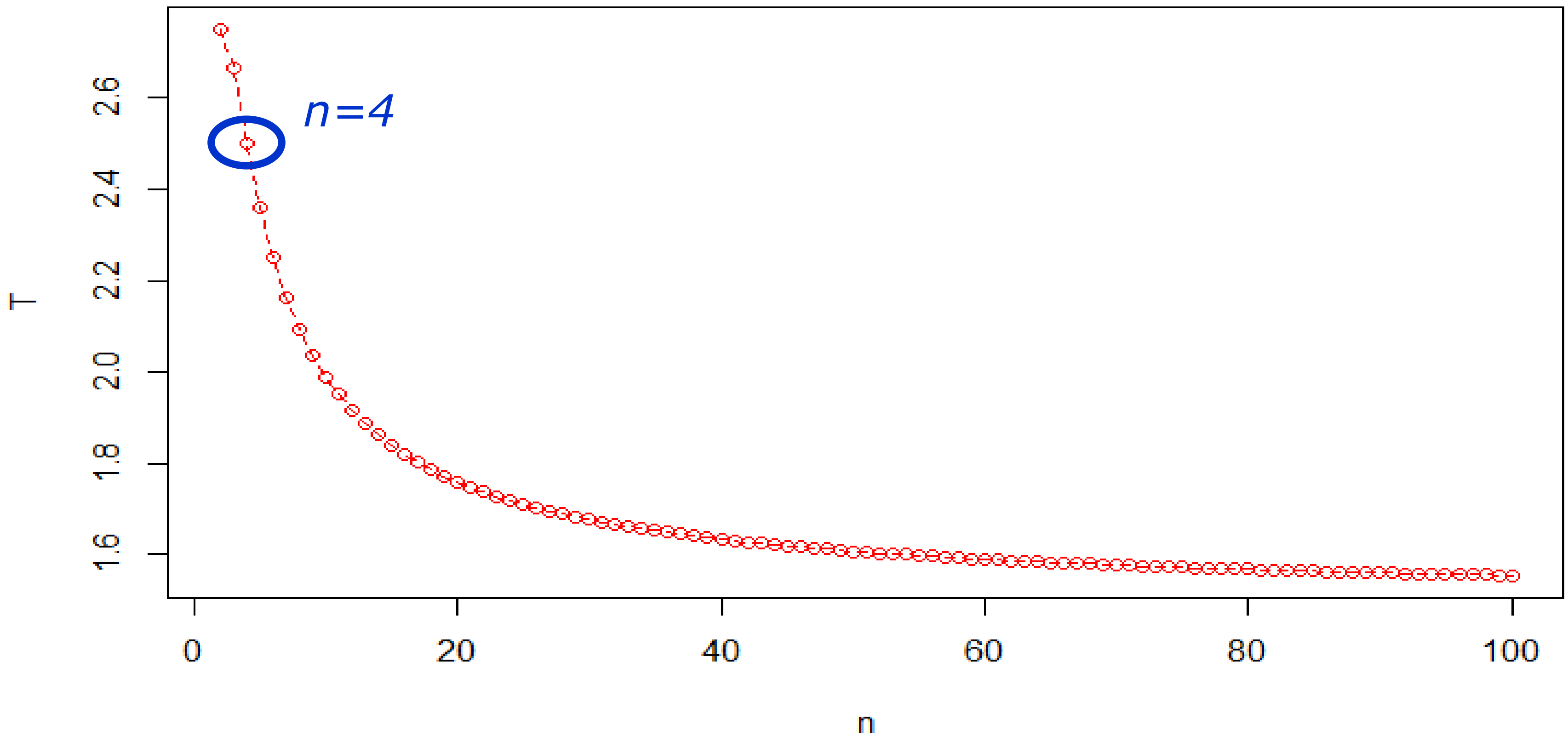
Big-O, Big-Θ & Big-Ω

Θ 符号：渐进确界

- 定义：**$\Theta(g(n))$**= {f(n)：存在正常数c_1，c_2和n_0，使对所有的 $n \geq n_0$，有：$\mathbf{0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)}$}

- 解读： Θ(g(n))表示满足条件的函数f(n)的集合
- 对于给定的g(n)和任意函数 f(n) ，若存在正常数c_1，c_2和n_0
- 使得：当n充分大时， f(n)的值落在$c_1g(n)$和$c_2g(n)$之间
- 则有：$f(n) \in \Theta(g(n))$ → 通常记为： $f(n) = \Theta(g(n))$
- $f(n) = \Theta(g(n))$表示：对所有的$n \geq n_0$
  - f(n)在一个常数因子范围内与g(n) **近似相等**
  - 称：g(n)是f(n)的一个渐进确界

Θ 符号：渐进确界



结论1：渐进正函数中的低阶项在决定渐进确界时可以被忽略

结论2：最高阶项的系数在决定渐进确界时也可以被忽略

任意常数可以表示为：$\Theta(1)$

O 符号：渐进上界

- Θ 符号渐进地给出一个函数的上界和下界
- 当只有渐进上界时，使用 O 符号
  - O (g(n)) 表示函数集合： {f(n)：存在正常数c和n_0，使对所有的$n \geq n_0$，有：$\mathbf{0 \leq f(n) \leq cg(n)}$}
  - f(n)= O (g(n)) 表示：f(n)是集合O (g(n))的一个元素
- 思考：已知 f(n)= Θ (g(n))，能否推断 f(n) = O (g(n)) ？
  - 答案：可以，因为 Θ 符号给出了函数f(n)的渐进上界
- 思考：下面的断言是否正确?
  - 任意线性函数 an+b = O (n^2)

Ω符号：渐进下界

- Ω 符号给出一个函数的渐进下界
  - $\Omega(g(n))$ 表示函数集合：｛f(n)：存在正常数c和n0，使对所有的n≥n0，有：$\mathbf{0 \le cg(n) \le f(n)}$ ｝
  - 通常用来与渐进上界一起来证明渐进确界
- 当渐进符号用于表达式中时，可以将其解释为一个函数
  - 例如： $2n^2+3n+1 = 2n^2+ \Theta(n)$
  - 可以将$\Theta(n)$解释为函数：$f(n)=3n+1$
  - 按照定义：f(n)是属于集合$\Theta(n)$的函数

思考题

1. $O(f)+O(g) = O(\max(f, g))$

2. $O(f) + O(g) = O(f + g)$

3. $O(f)O(g) = O(fg)$

4. $O(cf) = O(f)$ （c为常数）

常见的算法时间复杂度

常数阶	$O(1)$
对数阶	$O(\log n)$
线性阶	$O(n)$
线性对数阶	$O(n\log n)$
多项式阶	$O(n^2)$、$O(n^3)$
指数阶	$O(2^n) < O(n!) < O(n^n)$

算法复杂度分析

经验：现实生活中对数复杂度通常不超过 50

	问题规模	log N
KB	N = 1,000	9.9658≈10
MB	N = 1,000,000	19.9316≈20
GB	N = 1,000,000,000	29.8974≈30
TB	$N = 10^{12}$	39.8631≈40

算法复杂度分析示例1：选择排序

算法基本思想：

- 从无序子序列中选出关键字最小（或最大）的记录
- 将选出的记录按选出顺序加入到有序子序列中
- 逐步增加有序子序列的长度直至长度等于原始序列

排序过程

- 首先通过n-1次关键字比较，从n个记录中找出关键字最小的记录，将它与第一个记录交换
- 再通过n-2次比较，从剩余的n-1个记录中找出关键字次小的记录，将它与第二个记录交换
- 重复上述操作，共进行n-1趟排序后，排序结束

算法复杂度分析示例1：选择排序

k

i = 0	13	38	65	97	76	49	27
i = 1	13	27	65	97	76	49	38
i = 2	13	27	38	97	76	49	65
i = 3							
	13	27	38	49	76	97	65
i = 4							
	13	27	38	49	65	76	97
i = 5							
外层循环结束时	13	27	38	49	65	76	97

算法复杂度分析示例1：选择排序

```
void select_sort(int a[], int n)  {
    int i, j, k, tmp;
    for ( i = 0;  i< n-1;  ++i ) {
        j = i;
        for ( k = i+1;  k < n;  ++k ) {
            if (a[k] < a[j] )  j = k;
        }
        if ( j != i ) {
            tmp = a[i];   a[i] = a[j];  a[j] = tmp;
        }
    }
}
```

算法复杂度分析示例1：选择排序

```
void select_sort(int a[], int n)  {
    int i, j, k, tmp;
    for ( i = 0; i< n-1; ++i ) { ..................... n
        j = i; ........................................ n-1
        for ( k = i+1; k < n; ++k ) ...... ∑_{i=0}^{n-2}(n − i)
            if (a[k] < a[j] ) ........... ∑_{i=0}^{n-2}(n − i − 1)
                j = k; .................. p1 ∑_{i=0}^{n-2}(n − i − 1)
        if ( j != i ) { ................................ n-1
            tmp = a[i];
            a[i] = a[j];     ........................ p2(n-1)x3
            a[j] = tmp;
        }
    }
}
```

考虑最坏的情况：$p_1 = p_2 = 1.0$

$$T(n) = \frac{3}{2}n^2 + \frac{11}{2}n - 6 = O(n^2)$$

算法复杂度分析

为什么要分析最坏情况下的运行时间?

- 最坏情况运行时间是在**任何输入**下运行时间的**上界**
- 对多数算法而言最坏情况是频繁出现的
  - 例如数据库检索：要检索的信息常常是数据库中没有的
- 从统计学角度来看：平均情况通常与最坏情况一样差
  - 例如在选择排序中，每一轮迭代都需要判断当前位置上的元素是否为后续序列中的最小（或最大）值，否则就要进行交换；
  - 如果我们将待排序的序列和有序的序列进行比较，会发现在平均情况下，大约一半的元素是需要执行位置交换的
  - 如果取p=0.5，可以看到平均运行时间仍然是n的一个二次函数
- 所以算法复杂度分析主要关注的是最坏情况运行时间

算法复杂度分析示例2：归并排序

- **基本思想：**
  - 通过划分子序列，降低排序问题的复杂度
  - 通过合并有序的子序列，得到有序的序列
- **排序过程（设初始序列含有n个记录）**
  - 将原始序列划分为n个子序列（子序列长度为1）
  - 两两合并，得到$\lfloor n/2 \rfloor$个长度为2或1的有序子序列
  - 合并规则：如果某一轮归并过程中，单出一个子序列，则该子序列在该轮归并中轮空，等待下一趟归并
  - 如此重复，直至得到一个长度为n的有序序列为止

算法复杂度分析示例2：归并排序

分解 6 15 45 23 9 78 35 38 18 27 20

归并 6 15 | 23 45 | 9 78 | 35 38 | 18 27 | 20

归并 6 15 23 45 | 9 35 38 78 | 18 20 27

归并 6 9 15 23 35 38 45 78 | 18 20 27

归并 6 9 15 18 20 23 27 35 38 45 78

算法复杂度分析示例2：归并排序

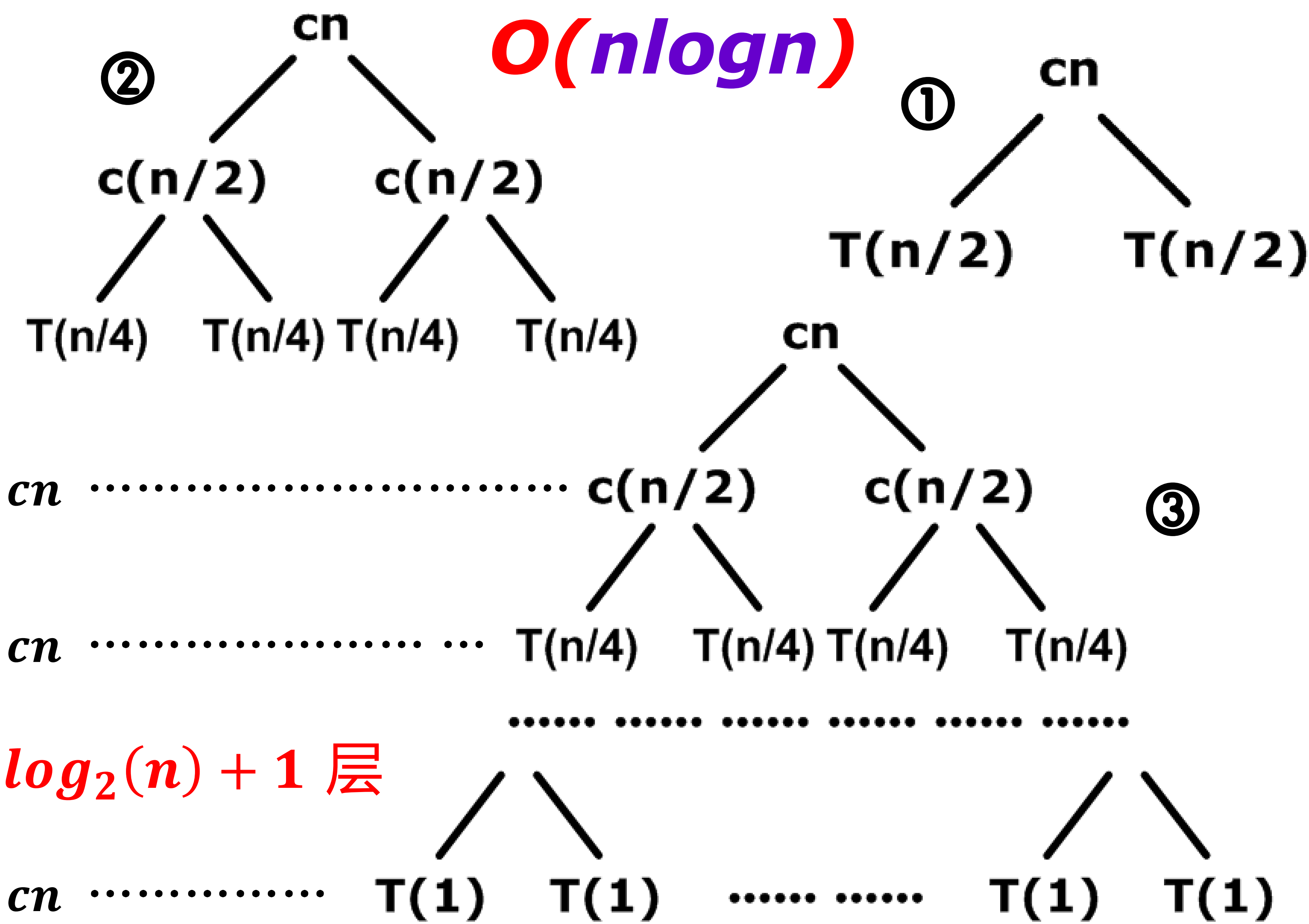
```
void merge_sort(int a[], int start, int end){
    int mid;
    if (start < end){
        mid = (start + end) / 2;
        merge_sort(a, start, mid); ...................... T(n/2)
        merge_sort(a, mid+1, end); .................... T(n/2)
        // 合并相邻的有序子序列
        merge(a, start, mid, end); ......................Θ(n)
    }
}
```

$$T(n) = 2T(n/2) + \Theta(n)$$

二路归并排序

```
// Ra[h..t]的两部分Ra[h..s]和Ra[s+1..t]已按关键字有序
// Ra[h..s]和Ra[s+1..t]合并成有序表（Rb[s..t]为辅助表）
void merge(int Ra[], int Rb[], int h, int s, int t){
    int i = h, k = h; j = s + 1;
    while( i <= s && j <= t ){
        if(Ra[i] < Ra[j]) Rb[k++] = Ra[i++];
        else  Rb[k++] = Ra[j++];
    }
    while (i <= s) Rb[k++] = Ra[i++];
    while (j <= t) Rb[k++] = Ra[j++];
    for (i = h; i <= t; i++) Ra[i] = Rb[i];
}
```

使用递归树分析：$T(n) = 2T(n/2) + \Theta(n)$



使用代换法验证：$T(n) = 2T(n/2) + \Theta(n) = O(nlogn)$

目的是证明： $\exists n_0$， $c > 0$, 使 $T(n) \le cnlogn$

假设这个界对n/2成立： $T(n/2) \le c(\frac{n}{2})log(\frac{n}{2})$

对递归式做代换：

$$T(n) \le 2c\left(\frac{n}{2}\right) log\left(\frac{n}{2}\right) + n$$

$$\le cnlogn - cnlog2 + n$$

$$\le cnlogn - cn + n$$

$$\le cnlogn$$ 当 $c \geq 1$ 成立

数学归纳法的边界条件： $T(1) \le clog1 = 0$？

$T(2) \le 2clog2 = 2c$ $T(3) \le 3clog3$ 取 $c \geq 2$ 即可

递归式渐进界分析：主定理（Master Theorem）

设： $a \geq 1, b > 1$为常数, $T(n)$对非负整数定义为

$$T(n) = aT(n/b) + f(n)$$

(1) 若对于某常数 ε>0，有： $f(n) = O(n^{\log_b a - \epsilon})$

则：$T(n) = \Theta(n^{\log_b a})$

(2) 若：$f(n) = \Theta(n^{\log_b a})$

则：$T(n) = \Theta(n^{\log_b a} \log n)$

(3) 若对于某常数 ε>0，有： $f(n) = \Omega(n^{\log_b a + \epsilon})$

且对于$c < 1$，当n足够大，有：$af(n/b) \leq cf(n)$

则：$T(n) = \Theta(f(n))$

主定理的应用：$T(n)=aT(n/b)+f(n)$

(1) 若：$f(n)=O\left(n^{\log_b a-\epsilon}\right)$ 则：$T(n)=\Theta\left(n^{\log_b a}\right)$

(2) 若：$f(n)=\Theta\left(n^{\log_b a}\right)$ 则：$T(n)=\Theta\left(n^{\log_b a}\log n\right)$

(3) 若：$f(n)=\Omega\left(n^{\log_b a+\epsilon}\right)$

且：$af(n/b)\le cf(n)$ 则：$T(n)=\Theta\left(n^{\log_b a}\right)$

已知归并排序的复杂度解析式：$T(n)=2T(n/2)+\Theta(n)$

显然：a = 2；b = 2； 可知：$n^{\log_b a}=n$

满足(2)的条件，因此有：$T(n)=O(n\log n)$

关于对数复杂度

- 对数复杂度从何而来？ Divide-and-Conquer
- 经验法则：
  - 如果一个算法用常数时间（O（1）） 将问题的规模削减为原始问题的一部分，则该算法的复杂度为：O（logn)
- 分治法典型示例：
  - 幂运算
  - 折半查找
  - 欧几里德算法（辗转相除法)

对数复杂度示例：幂运算

```
int power( int x, int n)
{
      if( n == 0 )
            return 1;
      if( n == 1 )
            return x;
      if(  n % 2 )
            return( power( x*x, n/2 ) * x );
      else
            return( power( x*x, n/2 ) );
}
```

时间复杂度: *O(logn)*

对数复杂度示例：折半查找（数据有序排列）

```
int binary_search( int a[ ], int x, int n ){
        int low = 0, mid, high = n - 1;
        while( low <= high ){
                mid = (low + high)/2;
                if( a[mid] < x )
                        low = mid + 1;
                else if ( a[mid] > x )
                        high = mid - 1;
                else
                        return( mid );  // found
        }
        return -1;
}
```

时间复杂度: *O(logn)*

对数复杂度示例：欧几里德算法

```
int gcd( int m, int n )
{
        unsigned int remainder;
        while( n > 0 )
        {
                remainder = m % n;
                m = n;
                n = remainder;
        }
        return( m );
}
```

时间复杂度: *O(logn)*

本章小结

算法和数据结构

是程序设计的核心关键环节

对算法性能起决定性作用

斐波纳契数列（Fibonacci Sequence）

- 斐波纳契数列的物理模型
  - 假设第一个月初有一对刚出生的兔子
  - 两个月之后（第三个月初）它们可以生育
  - 每月每对成年的兔子会产下一对小兔子
  - 假设在考察期间兔子不会死去

《Liber Abaci》

- 斐波纳契数列：1、1、2、3、5、8、13、21、……
  - F0=0，F1=1
  - Fn = F(n-1) + F(n-2)，（n>=2，n∈N）

斐波纳契数列

- 可以将结果列表如下

1月	2月	3月	4月	5月	6月
1	1	2	3	5	8

	8月	9月	10月	11月	12月
13	21	34	55	89	144

- 12个月以后的小兔子数量是144对

解法1：递归

```
long fib1(int n){
    if (n <= 1) {
        return n;
    } else{
        return fib1(n - 1) + fib1(n - 2);
    }
}
```

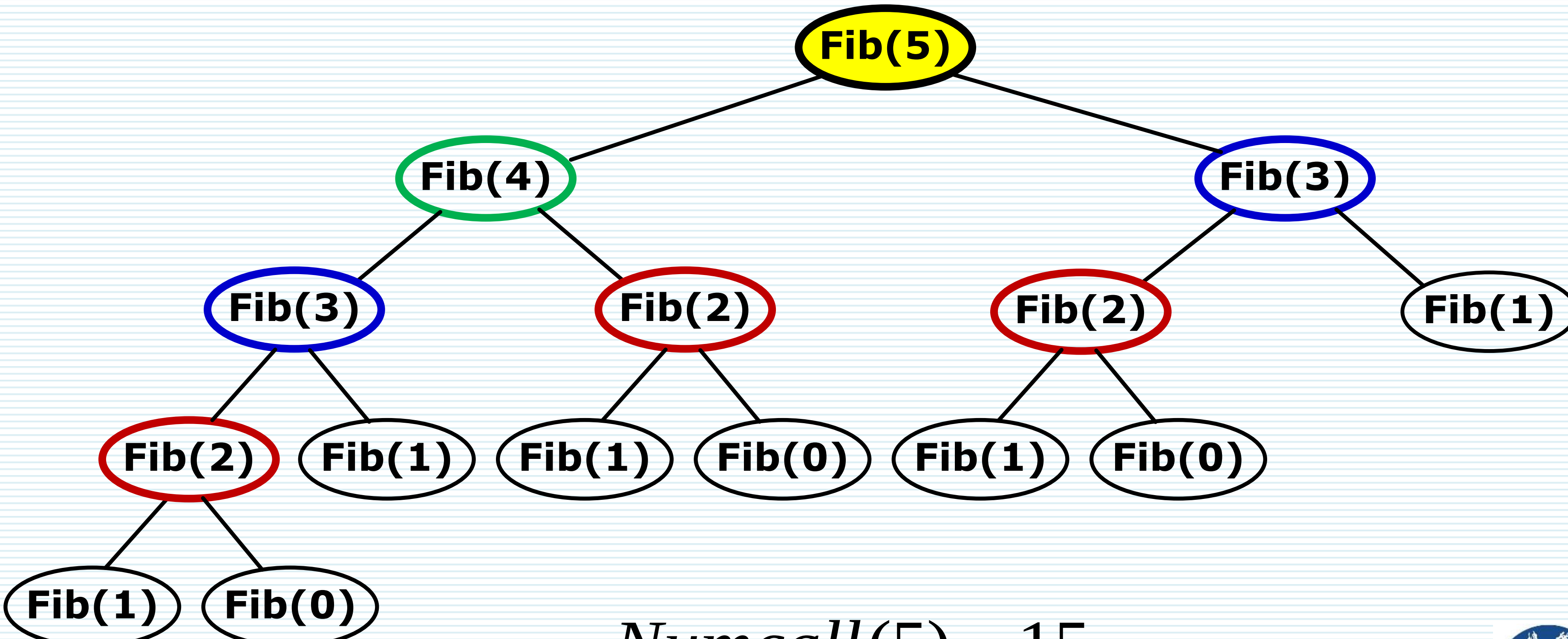
$$T(n) = O\left(\frac{1+\sqrt{5}}{2}\right)^n$$

斐波那契数列的递归求解过程

- fibonacci(5)的递归求解过程

$$Numcall(n) = Numcall(n-1) + Numcall(n-2) + 1$$

$$(n \geq 2)$$



$$Numcall(5) = 15$$

解法2：递推

递归解法的问题在于：重复求解子问题

观察**Fib(n)的定义：F**(n) = F(n-1) + F(n-2)；(n>=2)

F(n) 具有“无后效性”：只需“记住”前两个状态的结果即可

```
long fib2(int n){
    long f1 = 0, f2 = 1, fu;
    for(int i = 2; i <= n; ++i){
        fu = f1 + f2;
        f1 = f2; f2 = fu;  // 记忆
    }
    return fu;
}
```

算法复杂度：***O(n)***

解法3：矩阵

F1=1，F2=1，Fn = F(n-1) + F(n-2) (n>=2)

$$\begin{bmatrix} F_2 \\ F_1 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \end{bmatrix} \qquad \begin{bmatrix} F_{n+2} \\ F_{n+1} \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} F_{n+1} \\ F_n \end{bmatrix}$$

$$\begin{bmatrix} F_n \\ F_{n-1} \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} F_{n-1} \\ F_{n-2} \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^2 \begin{bmatrix} F_{n-2} \\ F_{n-3} \end{bmatrix} = \ldots = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^{n-2} \begin{bmatrix} F_2 \\ F_1 \end{bmatrix}$$

O(log(n))

电子科技大学

UESTC

1956