算法分析与设计

主讲教师： 刘峤

课程说明

- 课程性质：专业基础课（博）　专业选修课（硕）
- 总学时：**40**　　学分：**2**
- 上课时间地点：（ 2016年秋第**1-10**周）
  - 周一第**7-8**节，二教（**307**）
  - 周四第**5-6**节，二教（**307**）
- 课程序号： **20006026**

课程说明

- 成绩构成：
    - 期末成绩：考试
    - 平时成绩：**30%**（作业、考勤等）
    - 期末成绩：**70%**（闭卷考试）
- 联系方式：
    - 办公室：沙河主楼422
    - 助教：待定　　　邮箱：cuestc@163.com
    - 课程QQ群：待定（我的QQ：21255472）

教材与参考书

- 课程教材
  - **《计算机算法设计与分析》第四版**
    - 王晓东 著，电子工业出版社，2010.02
- 参考书推荐
  - **《算法导论》（第3版）**
    - Thomas H. Cormen, Charles E.Leiserson 等著
  - **《 Algorithm Design》 （第1版）**
    - Kleinberg 等著，清华大学出版社

课程概览

- **《算法分析与设计》课程讨论什么？**
  - 如何在计算机上表示问题和实现对问题的求解
  - 方法：将客观问题抽象为数学模型及其上的操作
- **课程目标**
  - 培养**计算思维**，提高编程解决问题的**能力**
- **课程重要性**
  - 算法是一切程序设计的基础

本学期讲解内容

第1章　算法概述

第2章　递归与分治策略

第3章　动态规划

第4章　贪心算法

第5章　回溯法

第6章　分支限界法

第7章　随机化算法（自学+选讲）

第8章　线性规划与网络流（自学+选讲）

第1章　算法概述

学习算法分析与设计的意义

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

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

算法的概念

- **算法（Algorithm)**
  - 是解决特定问题的方法或过程，是指令的有限序列
- **严格地说：算法是满足下述性质的指令序列**
  1. 输入：有零个或多个外部变量作为算法的输入
  2. 输出：算法产生至少一个输出
  3. 确定性：组成算法的每条指令清晰、无歧义
  4. 有限性：算法中每条指令的执行次数和时间有限
- **程序：是用某种程序设计语言的实现的算法**
  - 程序有可能不满足算法的性质(4)：即有限性

从算法到程序：存在的矛盾

- **算法设计人员**
  - 希望集中精力设计高效算法，不希望被实现细节干扰
  - 若开发环境改变，希望尽量减少对算法的修订
- **程序开发人员**
  - 希望得到符合实际且容易实现的算法
  - 若算法改变，希望被调用的实现过程不受太大影响
- **解决方案**
  - 衔接机制：将设计与开发过程的衔接抽象出来
  - 分层设计：让底层通过抽象接口为顶层服务

算法与程序

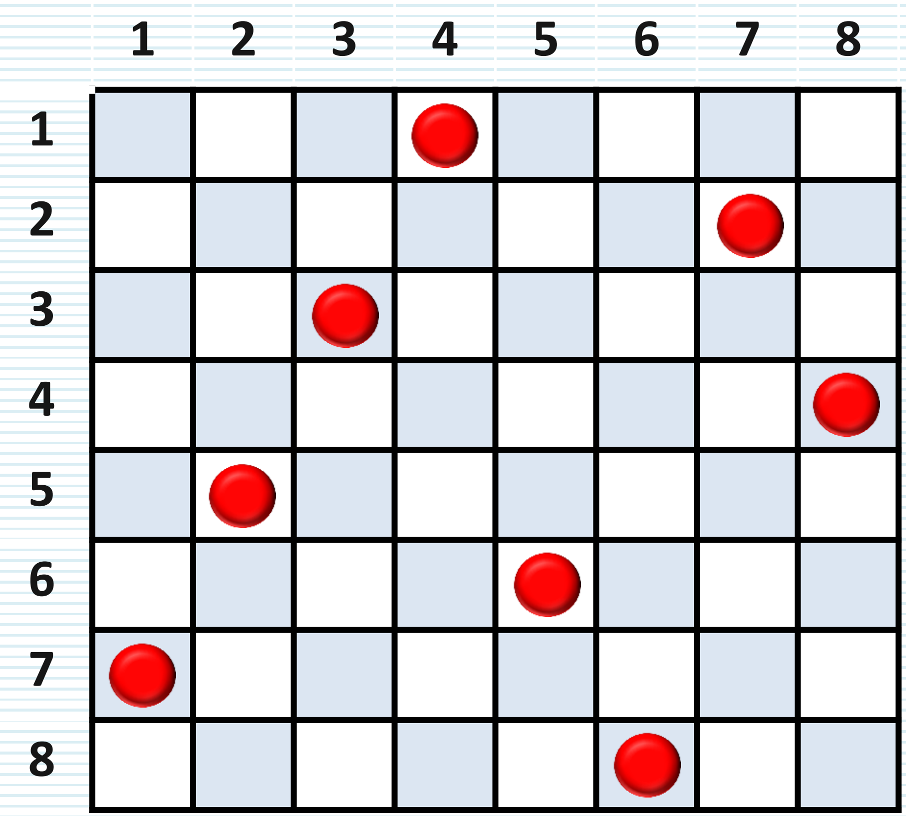
从算法到程序：自顶向下逐步求精

- 首先考虑：算法模型
  - 顶层数据模型层级上的运算模型
- 然后考虑：底层实现（两个要素）
  - 数学模型的结构：运算步骤
  - 数学模型的实现：数据结构

算法设计

☙ 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：从模型到结构

- 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

- 算法设计：主体流程
  - 参数 t 表示当前locate() 函数处理到棋盘的第 t 行

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

8-皇后问题

- 算法设计：具体细节
  - 参数 k 表示在当前行第 k 列放置一枚皇后棋子

```
bool is_ok(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;
}
```

回顾：8-皇后问题

- **问题分析**：明确需求，分析约束条件（显式的和隐含的）
  - 确定定解题策略（即算法的数学模型)
- **算法设计**
  - 确定数据结构
    - 设出问题的解向量：$(x_1, x_2, \ldots, x_8)$
  - 流程设计：自顶向下，逐层分解
    - 主体流程： locate() 函数
    - 具体细节： is_ok() 函数
  - 算法验证：确认算法结果正确，需求得到满足

算法复杂度分析

算法分析的内涵

- **正确性（correctness）**：满足用户具体需求
- **可读性（readability）**：利于读者理解算法
- **健壮性（robustness）**
  - 好的算法在出现异常或用户操作不当时均能作适当处理
- **时间和空间效率（time and space efficiency）**
  - 时间效率：指的是算法的执行时间应足够短
  - 空间效率：指算法执行需要的最大存储空间应足够小

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（算法复杂度）
  - 用高级语言编写的程序运行的时间主要取决于如下因素
    1. 算法复杂度：时间复杂度和空间复杂度
    2. 问题的规模
    3. 编程语言：一般情况下语言级别越高，效率越低；
    4. 编译程序：指令优化的能力
    5. 机器性能

算法复杂度的表示方法

- 算法复杂度函数

Complexity=f(scale, input, algorithm)

- **scale**：问题的规模（通常以符号 ***N*** 表示）
- **input**：算法的输入（相对于 ***N*** 可视为常量，可省略）
- **algorithm**：算法本身（隐含在**f**中，可省略）

- 一般将时间复杂度和空间复杂度分开来度量

- **时间复杂度：T = T(N)**
- **空间复杂度：S = S(N)**

算法复杂度分析

- 算法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（元运算）**
  - **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

- 确定算法中的基本操作（元运算）
- 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

算法复杂度的计算方法

- 假设算法在抽象计算机上运行
  - 有 ***k*** 种**元运算**，记为：$O_1, O_2, \ldots, O_k$
  - 它们的运行时间依次为：$t_1, t_2, \ldots, t_k$
  - 算法执行这些元运算的平均次数依次为：$e_1, e_2, \ldots, e_k$
  - 注意： e_i是问题规模***N***的函数：$e_i = e_i(N)$
- 则算法复杂度计算公式如下：

$$T(N) = \sum_{i=1}^{k} t_i \cdot e_i(N)$$

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;` ·· n^2

`for (k=1; k<=n; ++k) {` ·········· $n^2(n+1)$

`c[i,j] += a[i,k]*b[k,j];` ················ n^3

`}`

`}`

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

`}`

`}`

时间复杂度： $O(n^3)$

渐进时间复杂度

- 对于N的函数T(N)
  - 如果存在N的函数T*(N)，使得：

$$\lim_{N\to\infty}\frac{T(N)-T^*(N)}{T(N)}=0$$

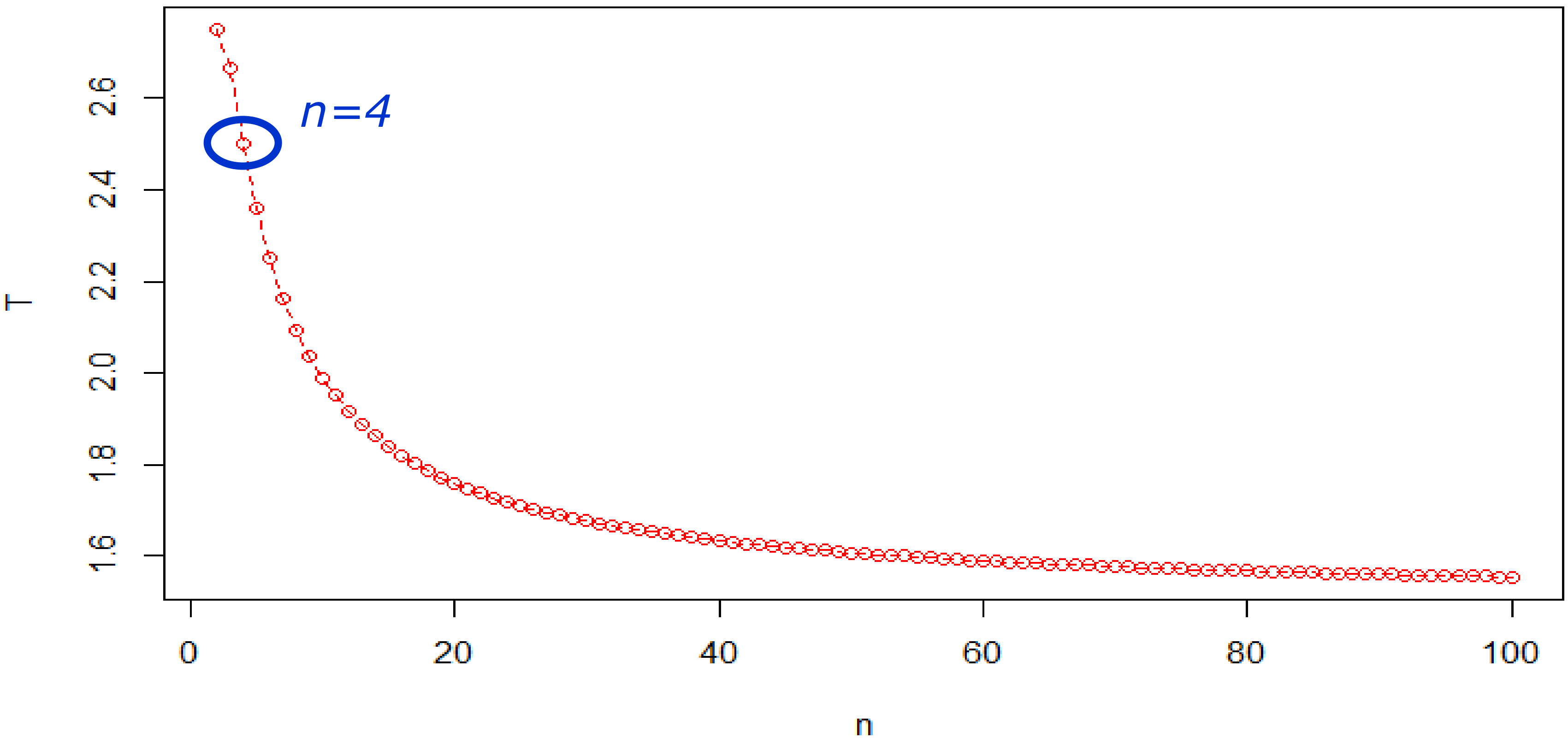
  - 则称T*(N)是T(N)当N $\to\infty$ 时的渐近表达式

- 例如：**T(N) = 3N² + 4NlogN + 7**
  - **T*(N) = 3N²** 为T(N) 的一个渐近表达式

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 $n \ge n_0$，有：**$0 \le c_1 g(n) \le f(n) \le c_2 g(n)$** }
  - 解读： $\Theta(g(n))$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$c_1 g(n)$和$c_2 g(n)$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$n \ge n_0$
    - 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：$0 \leq f(n) \leq cg(n)$ }
  - 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)

*Big-O*的实质

- 正确理解：**f(N) = O (g(N))**
  - 该"等式"只是表明f(N)的阶不高于g(N)的阶
    - 例如：$N^2 = O(N^3)$
  - 这里的"等号"并不具有通常情况下的自反性
    - 例如：由$N^2 = O(N^2)$ 和$N^2 = O(N^3)$ 推出 $O(N^3)=O(N^2)$，就是不正确的界
  - 一般来说g(N)的形式应比f(N)更加简洁

Big-O 的实质

- O 的实质是：问题规模相当大时，算法的复杂度上界
  - 是为评估、比较算法的优劣而引入的（忽略常数系数）
  - 这个上界的阶越低则评估越准确，结果越有价值
  - 例如：证明一个算法的复杂度上界为$O(N^{1.9})$就比证明它为$O(N^2)$有价值（通常也会更困难）
- 可以从集合的角度来理解O，并以$\in$来代替相应的 =
  - 例如： $O(N^2)$ 表示阶不大于 N^2 的函数构成的集合
  - $3N^2 = O(N^2)$ 表示：$3N^2$ 属于$O(N^2)$这个集合

Big-O 的运算规则

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)

Ω符号：渐进下界

- **Ω** 符号给出一个函数的渐进下界
  - $\boldsymbol{\Omega}(g(n))$ 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

渐进复杂度小结：Ω，O 与 θ

- 如果存在正常数 c 和自然数N_0，使得：
  - 当$N \geq N_0$时有：$f(N) \geq cg(N)$
  - 则记：$f(N) = \Omega(g(N))$
- 如果存在正常数 c 和自然数N_0，使得：
  - 当$N \geq N_0$时有：$f(N) \leq cg(N)$
  - 则记：$f(N) = O(g(N))$
- 如果$f(N) = O(g(N))$且$f(N) = \Omega(g(N))$
  - 则称 f(N) 与 g(N) 同阶，记为：$f(N) = \theta(g(N))$

最优算法

- 如果问题的时间复杂度下界为$\Omega(f(n))$
  - 则计算时间复杂度为$O(f(n))$的算法是最优算法
- 例如：排序问题的计算时间下界为$\Omega(n\log n)$
  - 计算时间复杂性为$O(n\log n)$的排序算法是最优算法
  - 快速排序和堆排序算法是最优算法

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

```
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

```c
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;
        }
    }
}
```

`for ( i = 0; i< n-1; ++i ) {`	n
`j = i;`	n-1
`for ( k = i+1; k < n; ++k )`	$\sum_{i=0}^{n-2}(n-i)$
`if (a[k] < a[j] )`	$\sum_{i=0}^{n-2}(n-i-1)$
`j = k;`	$p_1\sum_{i=0}^{n-2}(n-i-1)$
`if ( j != i ) {`	n-1
`tmp = a[i]; a[i] = a[j]; a[j] = tmp;`	p_2(n-1)x3

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

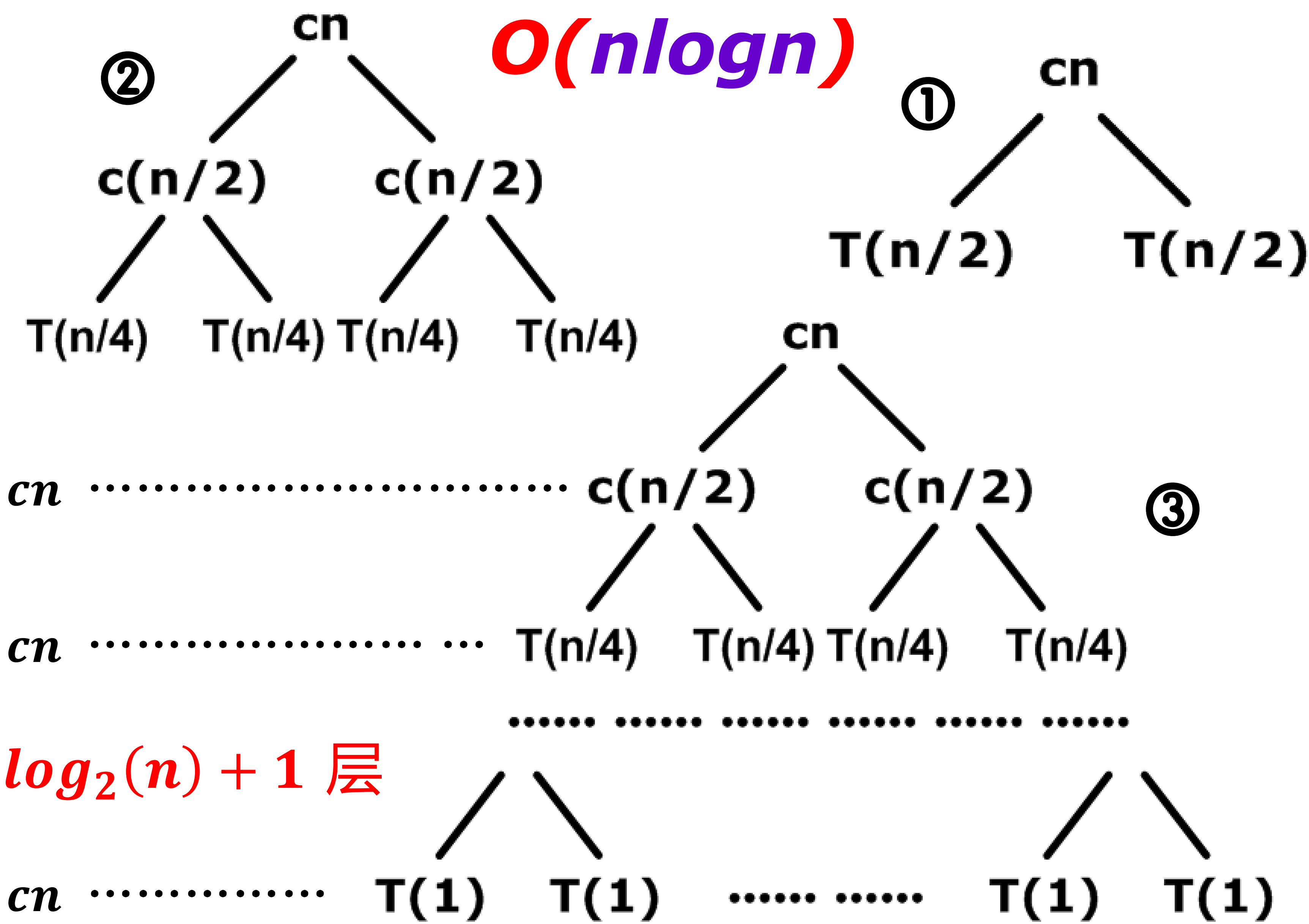
分解	6	15	45	23	9	78	35	38	18	27	20
归并	6	15	23	45	9	78	35	38	18	27	20
归并	6	15	23	45	9	35	38	78	18	20	27
归并	6	9	15	23	35	38	45	78	18	20	27
归并	6	9	15	18	20	23	27	35	38	45	78

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)
    }
}
```

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

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：$T(n) = \Theta(f(n))$

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: $O(\log n)$

本章小结

算法是程序设计的灵魂

对算法的性能起决定性作用

电子科技大学

UeSTC

1956