第2章 分治策略

苏州大学 计算机科学与技术学院
汪笑宇
Email: xywang21@suda.edu.cn

引入

■凡治众如治寡，分数是也。

——《孙子兵法》

孙 膑（战国初期） 明人绘

释义：

- ➢治众：治理人数众多的军队
- ➢治：治理；分数：军队的组织编制

本章内容

- ■分治策略（教材Chapter 4）
- ■快速排序（教材Chapter 7）
- ■中位数和顺序统计量（教材Chapter 9）

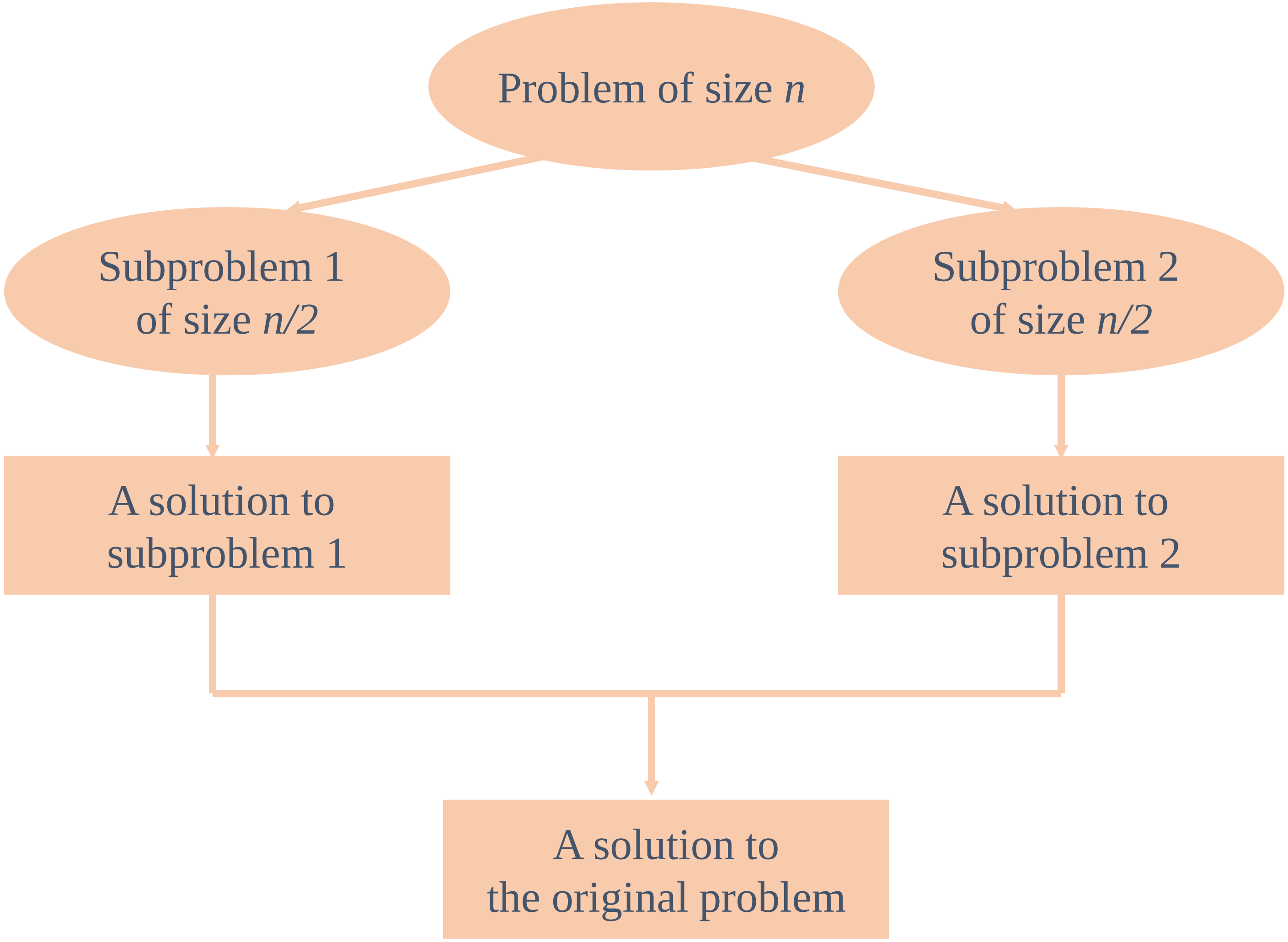
主要内容

- 理解递归（Recursion）的概念
- 掌握设计有效的分治策略算法及时间性能分析
- 通过范例学习分治策略设计技巧

递归式与分治法

- 直接或间接地调用自身的算法称为递归算法；用函数自身给出定义的函数称为递归函数
- 分治法产生的子问题往往是原问题的较小模式
- 分治与递归像一对孪生兄弟，经常同时应用在算法设计之中，并由此产生许多高效算法

递归式与分治法 (续)


Problem of size *n*
Subproblem 1
of size *n/2*
Subproblem 2
of size *n/2*
A solution to
subproblem 1
A solution to
subproblem 2
A solution to
the original problem

递归式

■例1：阶乘函数定义

$$n! = \begin{cases} 1, & n = 0 \\ n \cdot (n-1)!, & n > 0 \end{cases}$$

边界条件

递归方程

➢边界条件与递归方程是递归函数的两个要素，递归函数只有具备了这两个要素，才能在有限次计算后得出结果

递归式 (续)

■例2：斐波那契数列

$$F(n)=\begin{cases}0, & n=0,\\ 1, & n=1,\\ F(n-1)+F(n-2), & n\geq 2.\end{cases}$$

- 产生的序列为：0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ...
- 第*n*个Fibonacci数可递归地计算如下：

F(*n*)
1 **if** *n* = 0 **then return** 0
2 **elseif** *n* = 1 **then return** 1
3 **else return** (F(*n*-1) + F(*n*-2))

渐近上界?

递归式 (续)

■例2：斐波那契数列

➢除了直接递归以外的另3种求解方案：

- 方法1：递推方法 时间$O(n)$
- 方法2：求解通项公式 时间$O(1)$
- 方法3：分治策略 时间$O(\lg n)$

递归式 (续)

■例1, 2中的函数都可以找到非递归方式定义：

$$n! = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot n = \prod_{i=1}^{n} i$$

$$F(n) = \frac{\phi^n - \hat{\phi}^n}{\sqrt{5}} = \left\lfloor \frac{\phi^n}{\sqrt{5}} + \frac{1}{2} \right\rfloor$$

$$\phi = \frac{1+\sqrt{5}}{2} \approx 1.618$$

$$\hat{\phi} = \frac{1-\sqrt{5}}{2} \approx -0.618$$

递归式 (续)

■例3：Ackermann函数

当一个函数及它的一个变量是由函数自身定义时，称这个函数是双递归函数

Ackermann函数*A*(*m*, *n*)定义如下（双变量函数）：

$$A(m,n) = \begin{cases} n+1, & m=0 \text{ and } n \geq 0, \\ A(m-1,1), & m \geq 1 \text{ and } n = 0, \\ A(m-1, A(m, n-1)), & m \geq 1 \text{ and } n \geq 1. \end{cases}$$

- *A*(*m*, *n*)对*m*的每一个值都定义了一个单变量函数

递归式 (续)

■例3：Ackermann函数

$$A(1,n) = 2 + (n+3) - 3$$

$$A(2,n) = 2 * (n+3) - 3$$

$$A(3,n) = 2^{n+3} - 3$$

$$A(4,n) = \underbrace{2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}}_{n+3} - 3$$

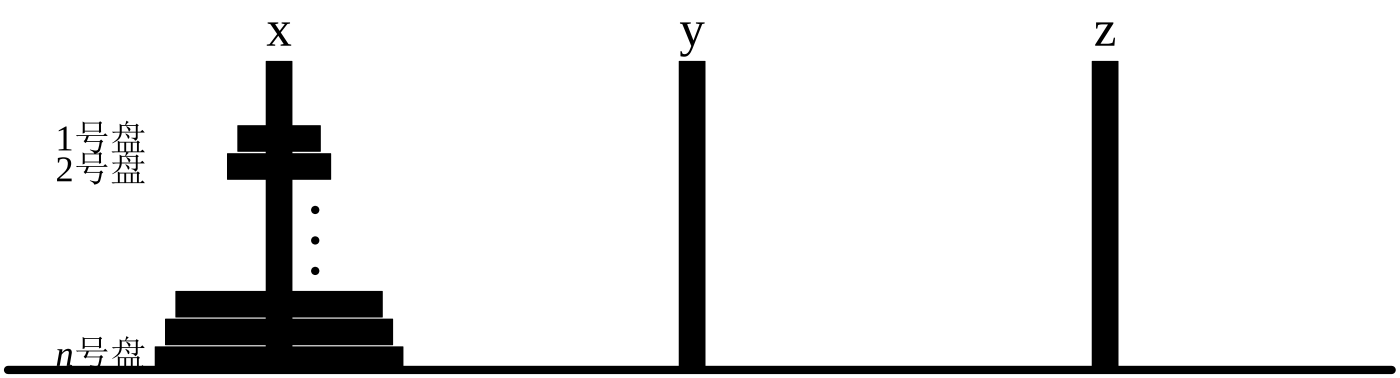
$$A(m,n) = \begin{cases} n+1, & m = 0 \text{ and } n \geq 0, \\ A(m-1,1), & m \geq 1 \text{ and } n = 0, \\ A(m-1, A(m,n-1)), & m \geq 1 \text{ and } n \geq 1. \end{cases}$$

递归式 (续)

■例4：n阶Hanoi塔问题

将x上的圆盘移到z上，要求按同样次序排列，且满足：

- ➢每次只能移动一片
- ➢圆盘可插在x, y, z任一塔座上
- ➢任一时刻大盘不能压在小盘上



递归式 (续)

- 例4：n阶Hanoi塔问题 Hanoi(n, x, y, z)
  - 当n=0时，没盘子可供移动，什么也不做
  - 当n=1时，可直接将1号盘子从x移动到z上
  - 当n=2时，可先将1号盘子移动到y，再将2号盘子移动到z，最后将1号盘子移动到z
  - 对于n>0的一般情况可采用如下分治策略进行移动：
    1. 将1至n-1号盘从x移动至y，递归求解Hanoi(n-1, x, z, y)
    2. 将n号盘从x移动至z
    3. 将1至n-1号盘从y移动至z，递归求解Hanoi(n-1, y, x, z)

递归式 (续)

■例4：n阶Hanoi塔问题

➢分解：设$n>1$，将n个圆盘从x移到z，y为辅助塔座：

1. 将上面n-1个盘从x移至y，z为辅助塔座
2. 将第n号圆盘（最大的圆盘）从x移至z
3. 将y上n-1个圆盘移至z，x为辅助塔座

子问题特征属性与原问题相同规模小1，参数不同

➢终结条件：n=1时，直接将编号为1的盘子从x移到z

递归式 (续)

■例4：n阶Hanoi塔问题

Hanoi(n, x, y, z)
1 **if** $n = 0$ **then return**
~~2 **elseif** $n = 1$ **then** move(1, x, z) //将1号盘子从x移到z~~
3 **else**
4 　Hanoi(n-1, x, z, y) //源x、目的y、辅助z
5 　move(n, x, z)
6 　Hanoi(n-1, y, x, z) //源y、目的z、辅助x

➢时间复杂度分析：设$T(n)$表示n个圆盘的Hanoi塔问题移动圆盘的次数

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

- $T(0)=0$
- $n>0$时，第4、6行：$T(n-1)$次；第5行：1次

递归式 (续)

- 例4：*n*阶Hanoi塔问题
  - 时间复杂度分析：设$T(n)$表示n个圆盘的Hanoi塔问题移动圆盘的次数，则$T(n)=2T(n-1)+1$

$$
\begin{aligned}
T(n) &= 2T(n-1)+1 \\
&= 2(2T(n-2)+1)+1 \\
&= 2(2(2T(n-3)+1)+1)+1 \\
&\ \vdots \\
&= 2^n T(n-n)+1+2+\ldots+2^{n-1} \\
&= 2^n-1 \\
&= O(2^n)
\end{aligned}
$$

递归式 (续)

■例5：全排列问题
设计递归算法生成n个元素$\{r_1, r_2, \ldots ,r_n\}$的全排列

- 设$R=\{r_1, r_2, \ldots ,r_n\}$是要进行排列的$n$个元素
- 集合$R_i=R-\{r_i\}$，即R_i表示R去除元素r_i的集合
- 集合X中元素的全排列记为$\mathrm{perm}(X)$
- $\mathrm{perm}(X)(r_i)$表示在全排列$\mathrm{perm}(X)$的每一个排列后加上后缀r_i得到的排列
- R的全排列可归纳定义如下：
  - 当$n=1$时，$\mathrm{perm}(R)=\{(r_1)\}$，其中$r_1$是集合$R$中唯一的元素；
  - 当$n>1$时，$\mathrm{perm}(R)=\{\mathrm{perm}(R_n)(r_n), \mathrm{perm}(R_{n-1})(r_{n-1}), \ldots, \mathrm{perm}(R_1)(r_1)\}$

递归式 (续)全排列问题 (续)

■例5：全排列问题

```
PERM(A, n)
if n = 0 then
    save(A)  //保存A的全排列
else
    PERM(A, n-1)  //perm(R_n)(A[n])
    for i ← n-1 downto 1 do
        exchange A[i] with A[n]  //交换A[i]和A[n]，A[n]作为排除元素
        PERM(A, n-1)             //相当于原数组A的perm(R_i)(A[i])
        exchange A[i] with A[n]  //数组A恢复原状
```

■时间复杂度 $T(n)=\begin{cases} T_{\text{save}}, & n=0, \\ n(T(n-1)+O(1)), & n\geq 1. \end{cases}$ $\quad O(T_{\text{save}}\cdot n!)$

递归式 (续)

- 优点：结构清晰，可读性强，而且容易用数学归纳法来证明算法的正确性，因此它为设计算法、调试程序带来很大方便
- 缺点：递归算法的运行效率较低，无论是耗费的计算时间还是占用的存储空间都比非递归算法要多
  - 解决方法：在递归算法中消除递归调用，使其转化为非递归算法

递归式与分治法

- 递归式与分治方法紧密相关，使用递归式可以很自然地刻画分治算法的运行时间
- 分治算法设计：将一个问题分解为与原问题相似但规模更小的若干子问题，递归地解这些子问题，然后将这些子问题的解结合起来构成原问题的解。这种方法在每层递归上均包括三个步骤：
  - 分解（Divide）：将问题划分为若干个子问题
  - 解决（Conquer）：递归地解这些子问题；若子问题规模足够小，则直接解决
  - 合并（Combine）：将子问题的解组合成原问题的解

递归式与分治法 (续)

■求解递归式方法（得出算法Θ或O渐近界）：

- 代入法：猜测一个界，然后用数学归纳法证明这个界是正确的
- 递归树法：将递归式转换为一棵树，结点表示不同层次的递归调用产生的代价，然后采用边界和技术来求解递归式
- 主方法：可求解形如$T(n)=aT(n/b)+f(n)$递归式的界

归并排序

- 分解：分解待排序的n个元素的序列成各具$n/2$个元素的两个子序列
- 解决：使用归并排序递归地排序两个子序列
- 合并：合并两个已排序的子序列以产生已排序的答案
  - 重复以下步骤直至两个子序列没有未处理的元素：
    - 比较两个子序列中第一个未处理的元素
    - 将较小的元素复制到输出序列A中，将相应子序列下标后移一位表示该元素已被处理过
  - 当一个子序列中的所有元素都被处理过后，复制另一个子序列中所有元素至A中

归并排序

■分解：分解待排序的n个元素的序列成各具$n/2$个元素的两个子序列

人们从大量实践中发现，在用分治法设计算法时，最好使子问题的规模大致相同。即将一个问题分成大小相等的k个子问题的处理方法是行之有效的。这种使子问题规模大致相等的做法是出自一种平衡(balancing)子问题的思想，它几乎总是比子问题规模不等的做法要好。

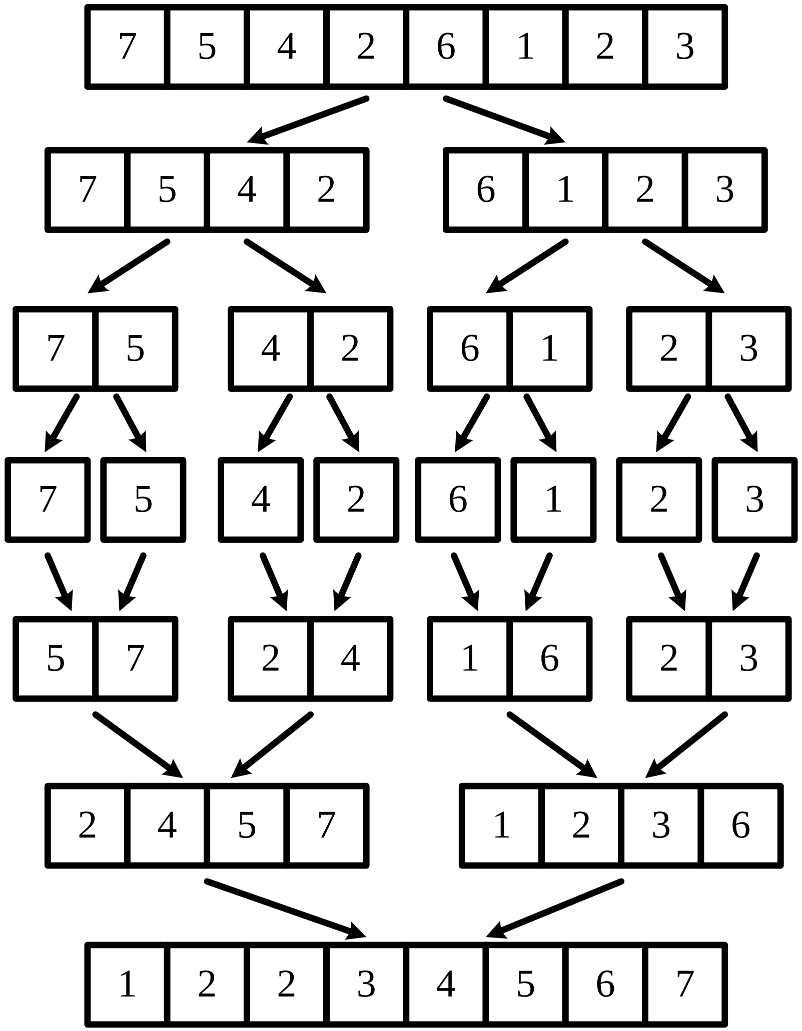
归并排序 (续)

```
MERGE_SORT(A, p, r)
if p < r
    q ← ⌊(p + r)/2⌋
    MERGE_SORT(A, p, q)
    MERGE_SORT(A, q+1, r)
    MERGE(A, p, q, r)
```

```
MERGE(A, p, q, r)
n1 ← q – p + 1
n2 ← r – q
let L[1..n1+1] and R[1..n2+1] be new arrays
for i ← 1 to n1 do L[i] ← A[p+i–1]
for j ← 1 to n2 do R[j] ← A[q+j]
L[n1+1] ← ∞,  R[n2+1] ← ∞
i ← 1;  j ← 1
for k ← p to r do
    if L[i] ≤ R[j]
        A[k] ← L[i],  i ← i + 1
    else A[k] ← R[j],  j ← j + 1
```

所有情形时间复杂度均为$\Theta(n\lg n)$

归并排序 (续)


7 5 4 2 6 1 2 3
7 5 4 2
6 1 2 3
7 5
4 2
6 1
2 3
7
5
4
2
6
1
2
3
5 7
2 4
1 6
2 3
2 4 5 7
1 2 3 6
1 2 2 3 4 5 6 7

分治算法时间性能分析

■（教材p20）设$T(n)$是输入规模为n的执行时间，若规模足够小，如$n \le c$ (常数)，则直接求解的时间为$\Theta(1)$

- 设完成划分的时间为$D(n)$
- 设分解时，划分为a个子问题，每个子问题规模为原问题的$1/b$，则解各子问题的时间为$aT(n/b)$
- 设合并时间$C(n)$

$$T(n)=\begin{cases}\Theta(1), & n \le c, \\ aT(n/b)+D(n)+C(n), & \text{otherwise.}\end{cases}$$

边界

$b>1$，否则无限递归

例如归并排序中：$a=2$，$b=2$，$D(n)=O(1)$，$C(n)=\Theta(n)$

分治算法时间性能分析 (续)

```
MERGE_SORT(A, p, r)
if p < r
    q ←⌊(p + r)/2⌋
    MERGE_SORT(A, p, q)
    MERGE_SORT(A, q+1, r)
    MERGE(A, p, q, r)
```

■归并排序的最坏情况运行时间

$$T(n)=\begin{cases}\Theta(1), & n=1,\\ T(\lceil n/2\rceil)+T(\lfloor n/2\rfloor)+\Theta(n), & n>1.\end{cases}$$

$$T(n)=\begin{cases}\Theta(1), & n=1,\\ 2T(n/2)+\Theta(n), & n>1.\end{cases}$$

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

■（教材p38）一般地，解递归式时可忽略细节

- 假定函数参数为整数，如n为奇数时两个子问题规模为$\lfloor n/2\rfloor$和$\lceil n/2\rceil$
- 边界条件可忽略，当n较小时$T(n)=\Theta(1)$

■因为这些细节一般只影响常数因子的大小，不改变量级。求解时，先忽略细节，然后再决定其是否重要

以下我们首先讨论细节问题！

递归式求解——代入法

- （教材p47-49）代入法求解递归式分为两步：
  - 猜测解的形式
  - 用数学归纳法求出解中的常数，并证明解是正确的
- 关键：用猜测的解代入到递归式中
- 例：确定 $T(n) = 2T(\lfloor n/2 \rfloor) + n$ 的上界
  - 猜测解 $T(n)=O(n\lg n)$，需要证明恰当选择常数 $c>0$，有 $T(n)\leq cn\lg n$

递归式求解——代入法 (续)

■例：确定$T(n) = 2T(\lfloor n/2 \rfloor) + n$的上界

➢猜测解$T(n)=O(n\lg n)$，需要证明恰当选择常数$c>0$，有$T(n)\le cn\lg n$

➢先看此猜测是否正确：假定该上界对所有正数$m<n$都成立，特别是对于$m = \lfloor n/2 \rfloor$有

$$T(\lfloor n/2 \rfloor) \le c\lfloor n/2 \rfloor \lg(\lfloor n/2 \rfloor)$$

代入递归式得到：

$$\begin{aligned} T(n) &\le 2c\lfloor n/2 \rfloor \lg(\lfloor n/2 \rfloor) + n \le cn\lg(n/2) + n \\ &= cn\lg n - cn\lg 2 + n \\ &= cn\lg n + (1-c)n \le cn\lg n \end{aligned}$$

只要$c\ge 1$即可

递归式求解——代入法 (续)

■例：确定$T(n) = 2T(\lfloor n/2 \rfloor) + n$的上界

- 猜测解$T(n)=O(n\lg n)$，需要证明恰当选择常数$c>0$，有$T(n)\leq cn\lg n$
- 数学归纳法要求证明边界条件成立：假设$T(1)=1$是递归式唯一边界条件，但$T(1)\leq c\cdot 1\cdot \lg 1=0$并不成立
- 渐近符号仅要求对$n\geq n_0$证明$T(n)\leq cn\lg n$
  - 扩展边界条件
  - 令$n_0=2$，之前的证明成立需要对$n_0 \leq m = \lfloor n/2 \rfloor < n$都成立，因此数学归纳法中需满足$n\geq 4$，此时只要找到足够大的$c$使得$T(2)$和$T(3)$满足$T(n)\leq cn\lg n$即可
  - $T(2)=4$，$T(3)=5$，即任何$c\geq 2$即可满足

递归式求解——代入法 (续)

- 注意事项1：做出好的猜测（没有一般方法，只能凭经验）
  - 与见过的解类似，则相应猜测

    例：$T(n) = 2T(\lfloor n/2 \rfloor + 17) + n$

    当n足够大时,$\lfloor n/2 \rfloor$与$\lfloor n/2 \rfloor + 17$ 相差无几，可猜测$T(n)=O(n\lg n)$
  - 先证较宽松的上下界，减小猜测范围

    例：$T(n) = 2T(\lfloor n/2 \rfloor) + n$

    显然$T(n)=\Omega(n)$ //式中有n相关的项

    $T(n)=O(n^2)$ //最多分解$O(n)$次，每次执行时间为n

    逐渐降低上界提升下界，直至收敛到渐近紧确界$T(n)=\Theta(n\lg n)$

递归式求解——代入法 (续)

■注意事项2：细节修正

- 有时猜测解是正确的，但数学归纳法却不能直接证明其细节
- 可从猜测解中减去一个低阶项使数学归纳法满足

例：$T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + 1$

猜测$T(n)=O(n)$，即需证明某个常数$c>0$，$T(n)\leq cn$成立

将猜测代入递归式得到$T(n) \leq c(\lfloor n/2 \rfloor) + c(\lceil n/2 \rceil) + 1 = cn + 1$

并不能得出$T(n)\leq cn$的结论

减去一个低阶项，新猜测为$T(n)\leq cn-d$ ($d\geq 0$为常数)

$T(n) \leq (c\lfloor n/2 \rfloor - d) + (c\lceil n/2 \rceil - d) + 1 = cn - 2d + 1 \leq cn - d$

只要$d\geq 1$即可。c通过边界条件选择

递归式求解——代入法 (续)

■注意事项3：避免陷阱

- 证明时渐近记号的使用易产生错误

  例：$T(n) = 2T(\lfloor n/2 \rfloor) + n$

  猜测$T(n)$=$O(n)$，此时“证明”$T(n) \le cn$

  $T(n) \le 2(c\lfloor n/2 \rfloor) + n \le cn + n = O(n)$　错误！

  必须证明$T(n) \le cn$的精确形式！

递归式求解——代入法 (续)

■注意事项4：变量代换

- 有时进行变量代换能使未知递归式变为熟悉的形式

例：$T(n) = 2T(\lfloor \sqrt{n} \rfloor) + \lg n$

令m=lg$n$得：$T(2^m) = 2T(2^{m/2}) + m$　(考虑$\sqrt{n}$是整数的情形)

再令$S(m)=T(2^m)$得：$S(m) = 2S(m/2) + m$

$S(m)=O(m\lg m) \Rightarrow T(n)=T(2^m)=S(m)=O(m\lg m)=O(\lg n\lg\lg n)$

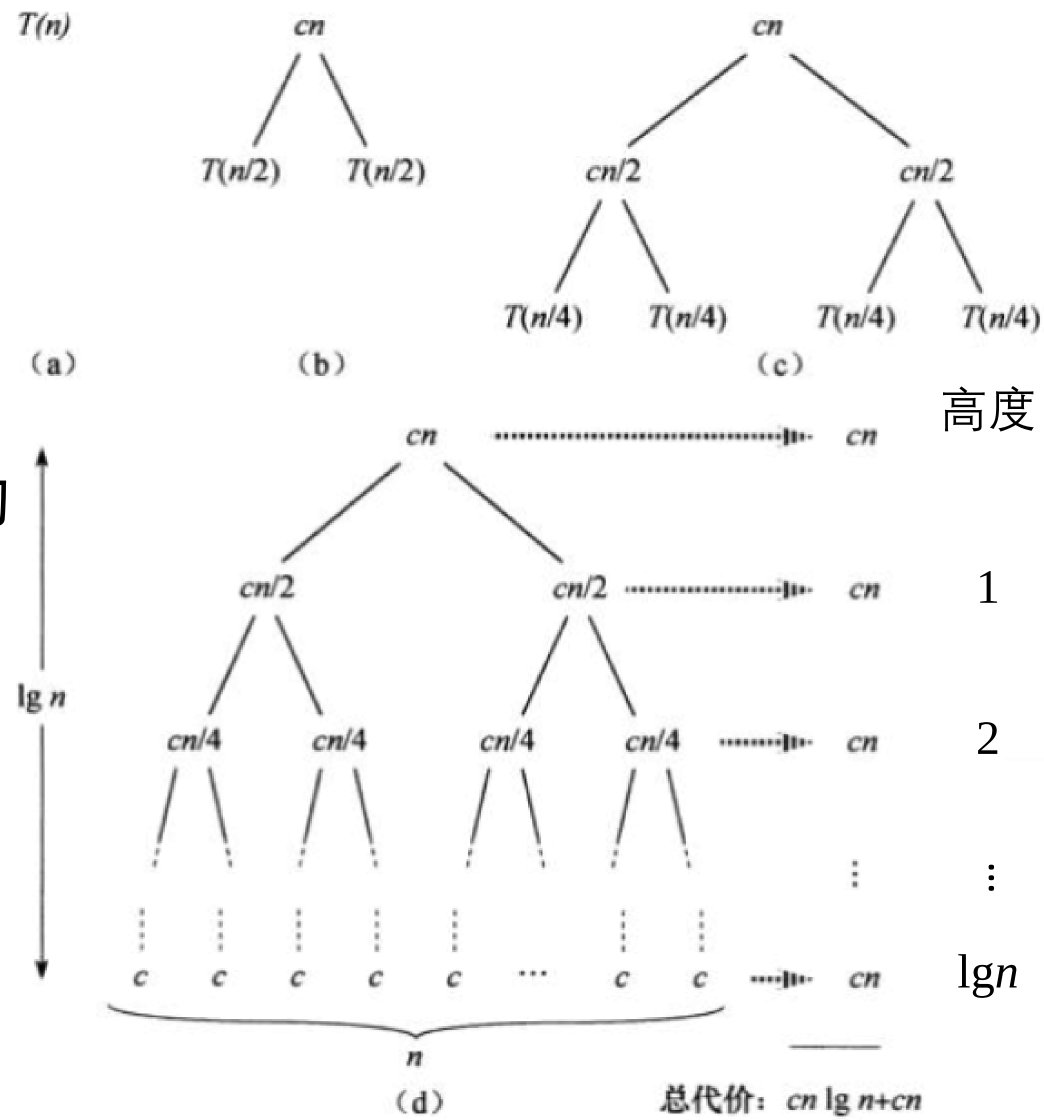
递归式求解——递归树法

- 递归树中每个结点表示一个单一子问题的代价，子问题对应某次递归函数调用
- 树中每层的代价求和得到每层代价，将所有层的代价求和，得到所有层次递归调用总代价
- 递归树是展开过程的形象化，从$T(n)$逐步展开直到$T(1)$

递归式求解——递归树法 (续)

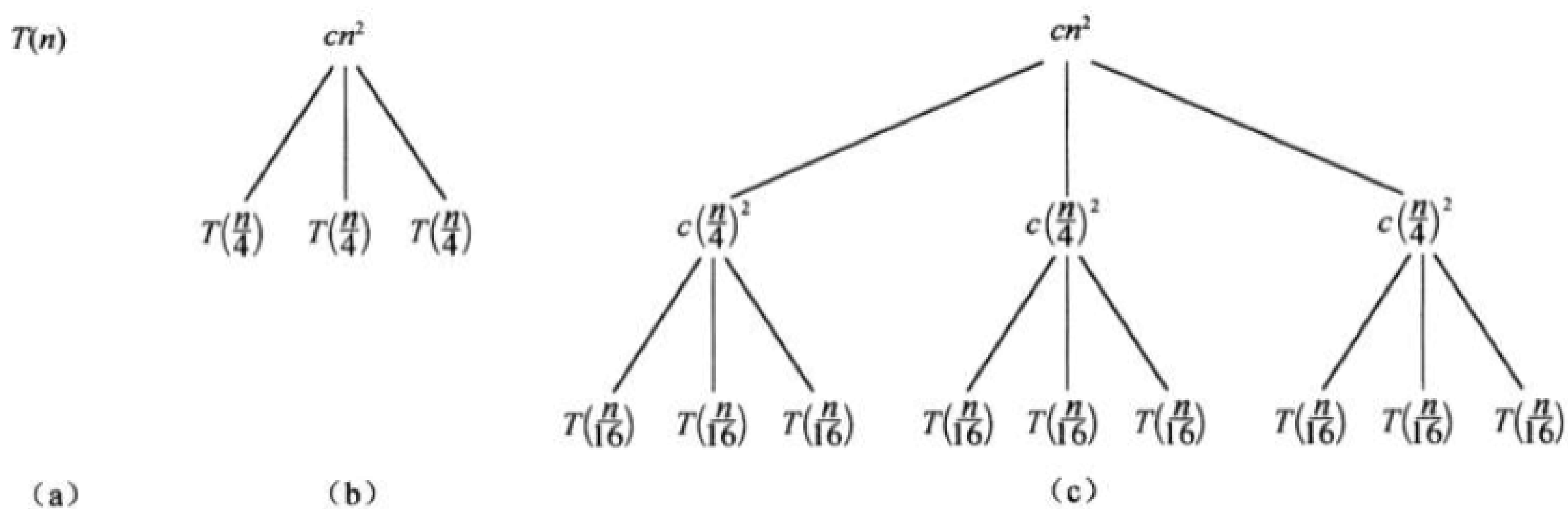
■例1：（教材p21）归并排序 $T(n)=2T(n/2)+cn$ （不妨设$n=2^k$）

- 树高（从根到叶的最长简单路径长度）：$\lg n$
- 总层数：$\lg n+1$
- 每层代价：cn
- 总代价：$cn\lg n+cn$



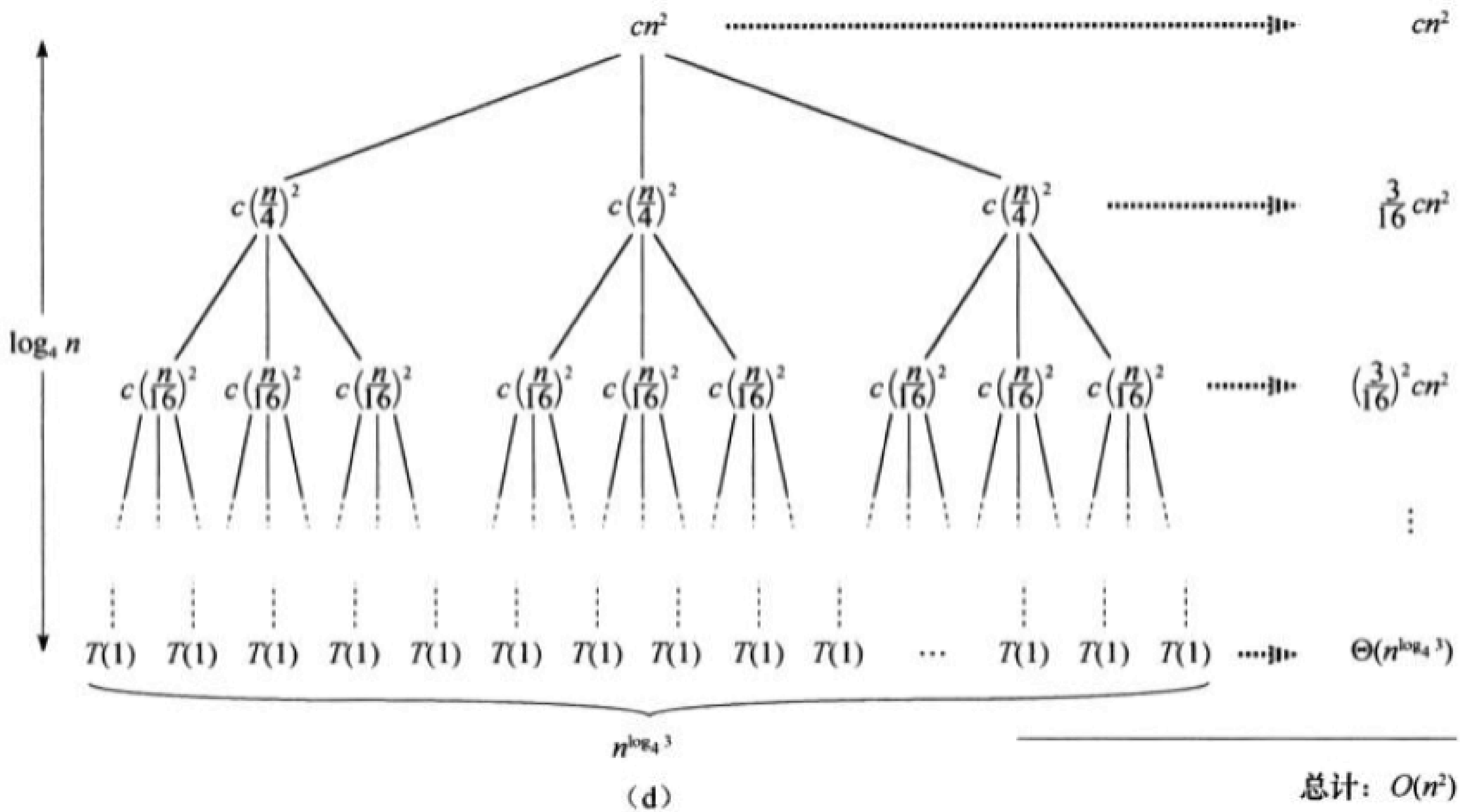
递归式求解——递归树法 (续)

■例2：（教材p51）$T(n) = 3T(\lfloor n/4 \rfloor) + cn^2$ （设n=4^k）



递归式求解——递归树法 (续)

■例2：（教材p51）$T(n) = 3T(\lfloor n/4 \rfloor) + cn^2$ （设$n=4^k$）



项数

3^1

3^2

⋮

$3^{\log_4 n} = n^{\log_4 3}$

递归式求解——递归树法 (续)

■例2： （教材p51）$T(n) = 3T(\lfloor n/4 \rfloor) + cn^2$ （设n=4^k）

$$
\begin{aligned}
T(n) &= cn^2 + \frac{3}{16}cn^2 + \left(\frac{3}{16}\right)^2 cn^2 + \ldots + \left(\frac{3}{16}\right)^{\log_4 n-1} cn^2 + \Theta(n^{\log_4 3}) \\
&= \sum_{i=0}^{\log_4 n-1} \left(\frac{3}{16}\right)^i cn^2 + \Theta(n^{\log_4 3}) \\
&= \frac{1-(3/16)^{\log_4 n}}{13/16} cn^2 + \Theta(n^{\log_4 3}) \\
&< \frac{16}{13} cn^2 + \Theta(n^{\log_4 3}) \qquad //0 < (3/16)^{\log_4 n} \le 1 \\
&= O(n^2)
\end{aligned}
$$