Part2：递归与分治策略（I）

- **减治法：插入排序**
- 分治法：归并排序
- 递归分析方法

减治法（第4章）

- 减治法基本策略：利用了一个问题给定实例的解和同样问题较小实例的解之间的某种关系，自顶向下或自底向上运用这种关系
- 从求解一个问题的较小实例开始，该方法有时称为“增量法”
- 减治法有三种主要的变化形式：
  - 减去一个常量
  - 减去一个常量因子
  - 减去的规模是可变的

减治法

- 减常量：每次迭代总是从实例中减去一个相同的常量
- 减常因子：每次迭代总是从实例中减去常数因子，一般等于2

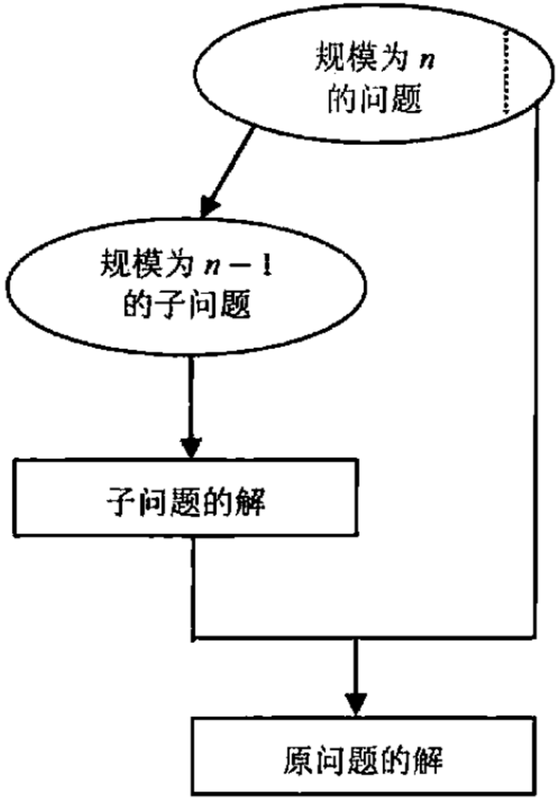


图 4.1 减(一)治技术

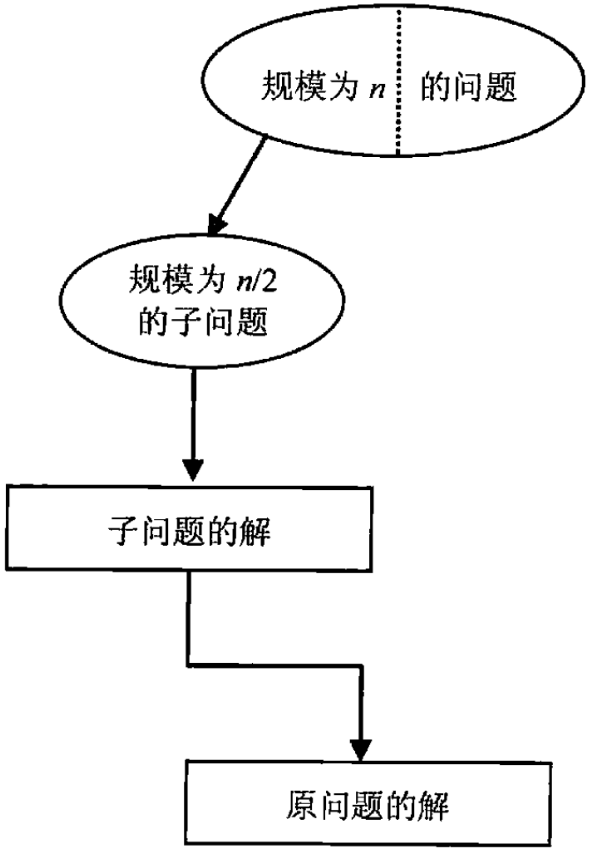


图 4.2 减(半)治技术

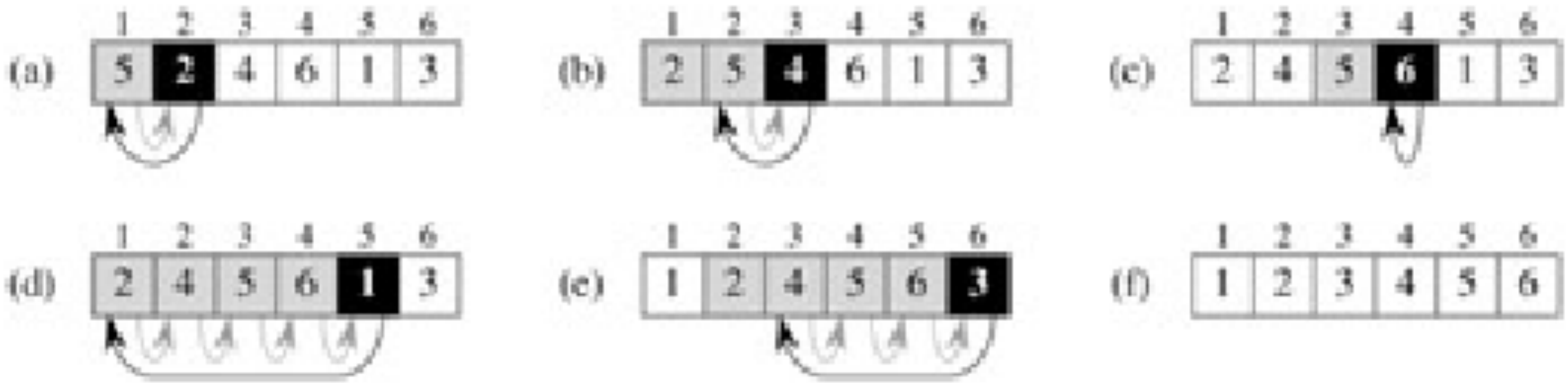
4.1 减治法：插入排序

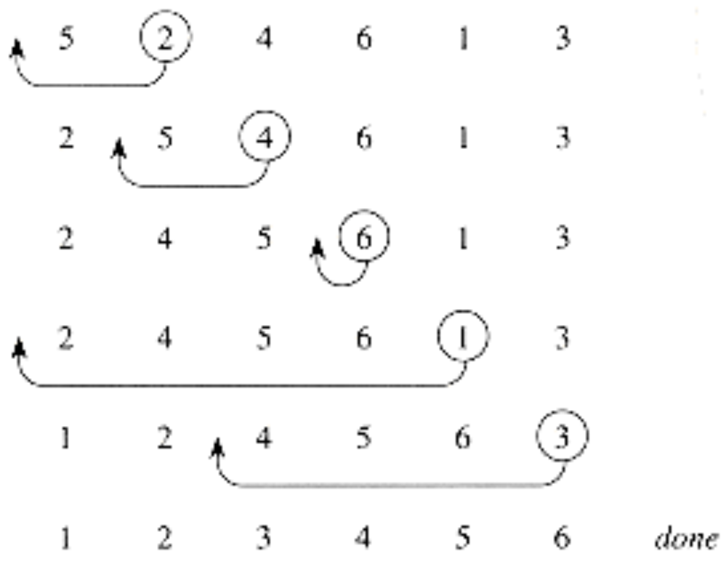
Problem:to sort a sequence of numbers into nondecreasing order

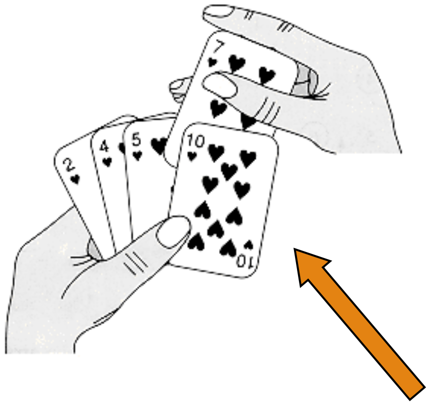
Input: A sequence of n numbers $<a_1, a_2, \cdots, a_n>$

Output: A permutation (reordering) $<a_1', a_2', \cdots, a_n'>$ of the input sequence such that $a_1' \le a_2' \le \cdots \le a_n'$

4.1 插入排序


(a) 1 2 3 4 5 6
5 2 4 6 1 3
(b) 1 2 3 4 5 6
2 5 4 6 1 3
(c) 1 2 3 4 5 6
2 4 5 6 1 3
(d) 1 2 3 4 5 6
2 4 5 6 1 3
(e) 1 2 3 4 5 6
1 2 4 5 6 3
(f) 1 2 3 4 5 6
1 2 3 4 5 6


5 2 4 6 1 3
2 5 4 6 1 3
2 4 5 6 1 3
2 4 5 6 1 3
1 2 4 5 6 3
1 2 3 4 5 6 done



4.1 插入排序——伪代码

算法 InsertionSort($A[0..n-1]$)

//用插入排序对给定数组排序

//输入：n 个可排序元素构成的一个数组 $A[0..n-1]$

//输出：非降序排列的数组 $A[0..n-1]$

for $i \leftarrow 1$ **to** $n-1$ **do**

  $v \leftarrow A[i]$

  $j \leftarrow i-1$

  while $j \geqslant 0$ **and** $A[j] > v$ **do**

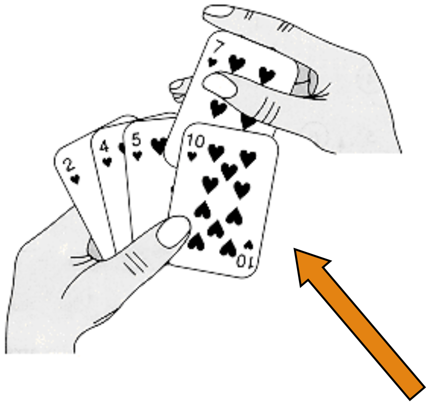
    $A[j+1] \leftarrow A[j]$

    $j \leftarrow j-1$

  $A[j+1] \leftarrow v$

$$A[0] \leqslant \cdots \leqslant A[j] < A[j+1] \leqslant \cdots \leqslant A[i-1] \mid A[i] \cdots A[n-1]$$

小于等于$A[i]$ 大于$A[i]$

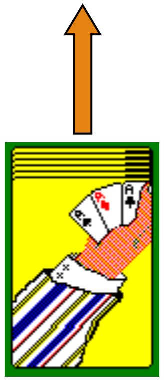


伪代码表示规则

- 用缩进代表块结构
- 循环结构(**while**, **for**, **repeat**) 和条件结构 (**if**, **then**, **else**)的表示与C类似
- 用“// ” or “▷”表示注释.
- 多赋值语句$i \leftarrow j \leftarrow e$ 相当于 $j \leftarrow e$ *then* $i \leftarrow j$.
- 数组元素访问: $A[i]$; $A[1 .. j] = < A[1], A[2], \ldots, A[j]>$
- 复合数据表示为对象

4.1 分析插入算法

INSERTION-SORT(*A*)	cost	times
for($j = 2$; $j <= n$; j++)	c_1	n
{ $v = A[j]$	c_2	n-1
// Insert $A[j]$ into the sorted sequence $A[1 .. j-1]$	0	n-1
$i = j$-1	c_4	n-1
while(i >0 && $A[i] > v$)	c_5	$\sum_{j=2}^{n} t_j$
{ $A[i+1] = A[i]$	c_6	$\sum_{j=2}^{n} (t_j - 1)$
$i = i$-1	c_7	$\sum_{j=2}^{n} (t_j - 1)$
}		
$A[i+1] = v$	c_8	n-1
}		



$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7 \sum_{j=2}^{n} (t_j - 1) + c_8(n-1)$$

4.1 分析插入算法

$$T(n)=c_1n+c_2(n-1)+c_4(n-1)+c_5\sum_{j=2}^{n}t_j+c_6\sum_{j=2}^{n}(t_j-1)+c_7\sum_{j=2}^{n}(t_j-1)+c_8(n-1)$$

- **相同输入规模的情况下，输入不同，运算时间也可能不同**
  - 最好情况：数组已经排序
  - For each $j = 2, 3, \ldots, n$, $A[i] \le v$ in line 5 when i has its initial value of j-1. Thus t_j=1, and the best-case running time is

$$T(n)=c_1n+c_2(n-1)+c_4(n-1)+c_5(n-1)+c_8(n-1)$$
$$=(c_1+c_2+c_4+c_5+c_8)n-(c_2+c_4+c_5+c_8)=an+b$$

结果是n的线性函数.

4.1 分析插入算法

$$T(n)=c_1n+c_2(n-1)+c_4(n-1)+c_5\sum_{j=2}^{n}t_j+c_6\sum_{j=2}^{n}(t_j-1)+c_7\sum_{j=2}^{n}(t_j-1)+c_8(n-1)$$

- 最坏情况：数组是倒序.
- 需要比较每一个$A[j]$ 和整个已经排序的$A[1\,..\,j\text{-}1]$中每个元素，所以$t_j=j$.

$$\sum_{j=2}^{n}j=\frac{n(n+1)}{2}-1\ ,\qquad \sum_{j=2}^{n}(j-1)=\frac{n(n-1)}{2}$$

$$T(n)=c_1n+c_2(n-1)+c_4(n-1)+c_5\left(\frac{n(n+1)}{2}-1\right)$$
$$+c_6\left(\frac{n(n-1)}{2}\right)+c_7\left(\frac{n(n-1)}{2}\right)+c_8(n-1)=an^2+bn+c$$

结果是n的二次函数

4.1 分析插入算法

- 最坏情况: 输入数据是倒序排列

$$T(n) = \sum_{j=2..n} \Theta(j) = \Theta(n^2)$$

- 平均情况：所有位置置换发生的概率相同

$$T(n) = \sum_{j=2..n} \Theta(j/2) = \Theta(n^2)$$

- 插入排序是一个快速的排序算法吗？
  - **--Moderately so, for small n.**
  - **--Not at all, for large n.**

Part2：递归与分治策略（I）

- 减治法：插入排序
- 分治法：归并排序
- 递归分析方法

掌握

- 描述和分析算法的框架
- 用伪代码描述算法
- 使用一致性符号来分析算法的运算时间

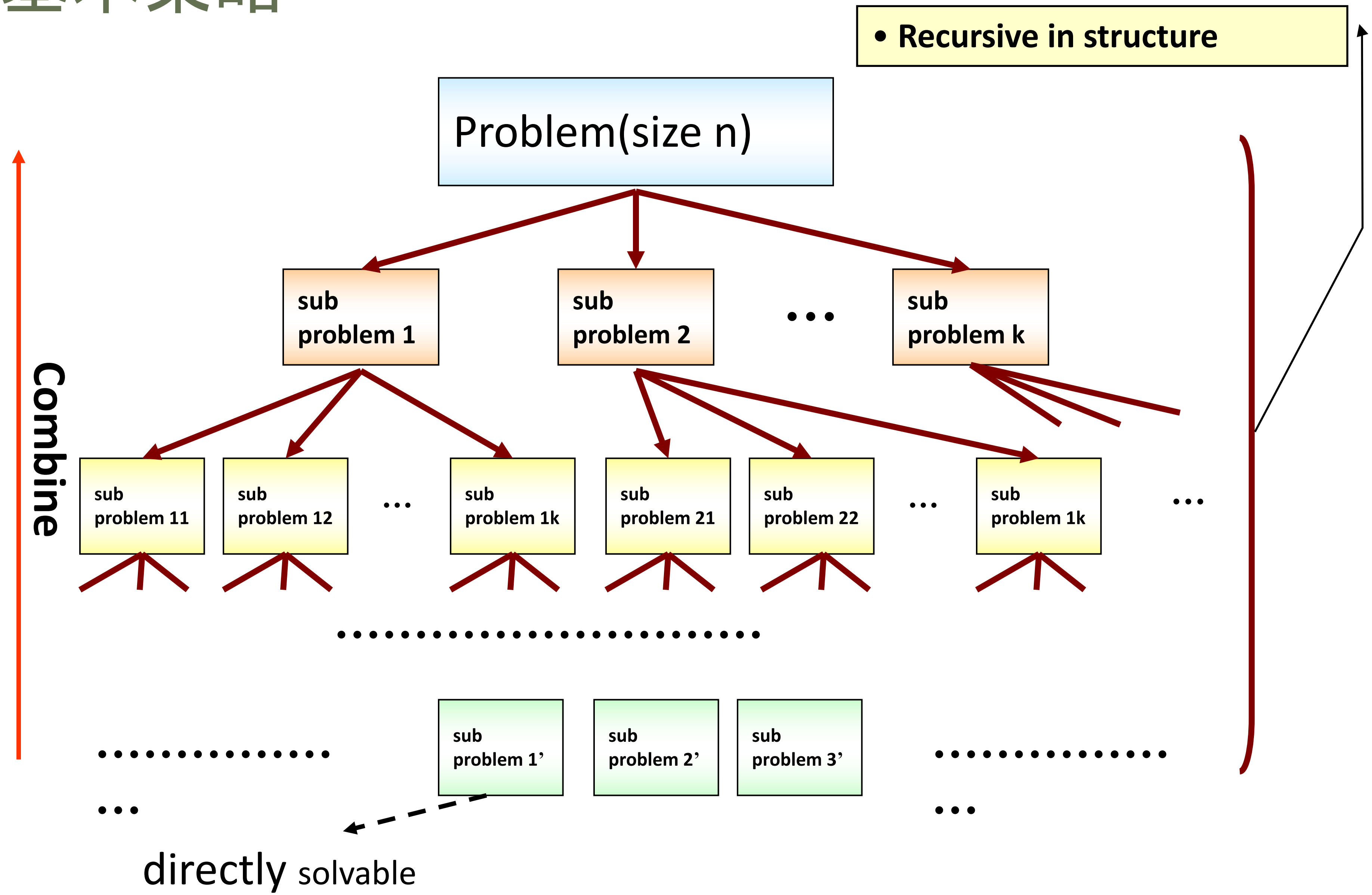
分治法（第5章）

- **分治法基本策略（分而治之，各个击破）：**
  - （分）将一个问题划分为同一类型的若干子问题，子问题最好规模相同。
  - （治）对这些子问题求解（通常使用递归的方式）。
  - （合并）如果有必要，合并这些子问题的解，以得到原问题的解。

例子

- 计算n个数字$a_1, a_2, ..., a_{n-1}$的和；
- 如果$n > 1$，可以把该问题划分为两个子问题，前一半的数字之和和后一半的数字之和；
- $a_1 + a_2 + ... + a_{n-1} + a_n = (a_0 + a_1 + ... + a_{n/2-1}) + (a_{n/2} + a_{n/2+1} + ... + a_{n-1})$
- 类似地一直划分，直到含有一个元素，直接返回该元素值；

分治法基本策略

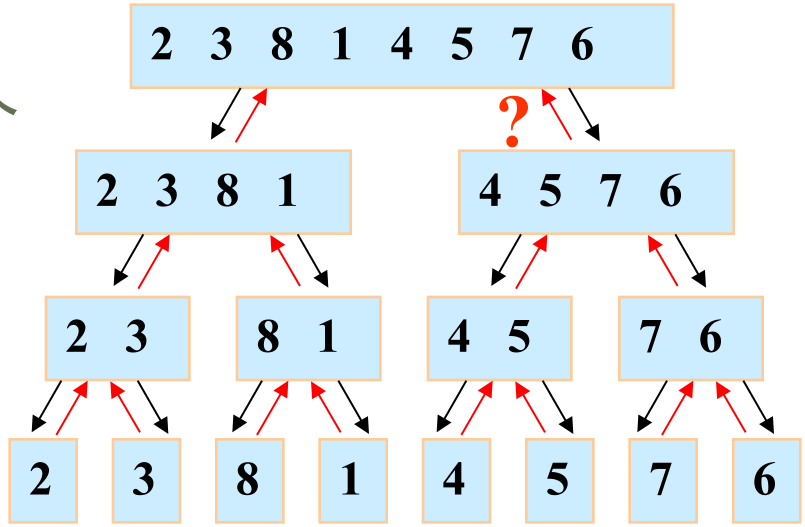


Recursive in structure
Problem(size n)
sub problem 1
sub problem 2
sub problem k
Combine
sub problem 11
sub problem 12
sub problem 1k
sub problem 21
sub problem 22
sub problem 1k
sub problem 1'
sub problem 2'
sub problem 3'
directly solvable

5.1 合并排序

- **分解:** 将原来n个元素的序列分成两个包含$n/2$元素的待排序的子序列
  (Exercise: how about four subsequences, $n/4$ elements each?)
- **解决:** 对两个子序列递归地进行合并排序
- **合并:** 把两个已排序的子序列进行合并，得到排序结果

- 当递归过程到到最底端，此时要排序的序列只包含一个元素，不需要再进行分解，一个元素是已排好序的



Merge-Sort (A, p, r)

- 假设有合并过程 **MERGE (A,p,q,r)**， 将已排序的**A[p…q]** 和已排序的**A[q+1…r]**在**(r - p)**时间进行合并.

MERGE (A,p,q,r)

- **INPUT: a sequence of n numbers stored in array A**
- **OUTPUT: an ordered sequence of n numbers**

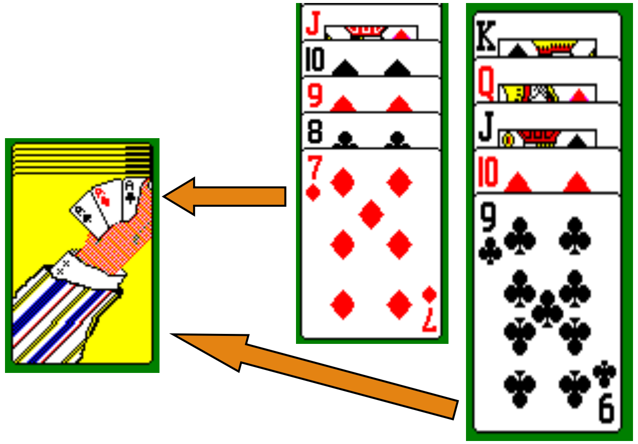
```
MERGE-SORT (A, p, r)
    if p < r
        then q ← ⌊(p + r)/2⌋
            MERGE-SORT(A, p, q)
            MERGE-SORT(A, q + 1, r)
            MERGE(A, p, q, r)
```

5.1 合并排序

- **MERGE(A,p,q,r): 合并操作是合并排序的关键操作.**
  - A 是数组，p, q, and r 是数组中的下标 $p \le q < r$.
  - 这个过程假设$A[p .. q]$ 和 $A[q+1 .. r]$ 已经排序. 合并过程将两个子序列合并成一个排好序的子序列$A[p .. r]$.
  - 步骤:

?

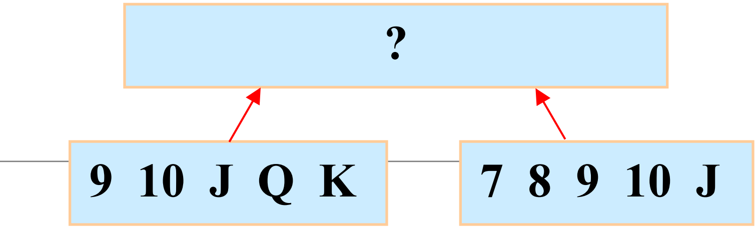
9 10 J Q K **7 8 9 10 J**

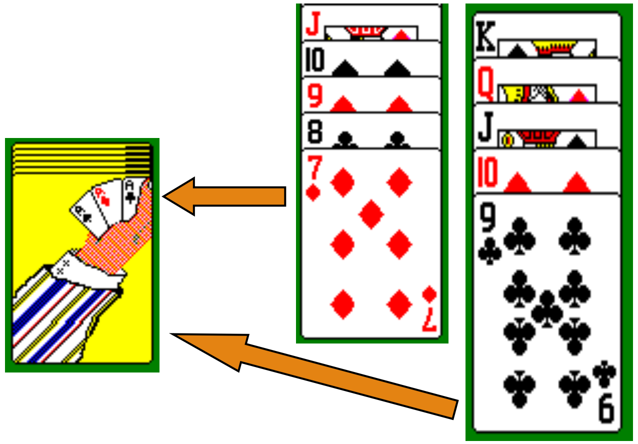


这个过程需要$\Theta(n)$, $n=r-p+1$

合并过程伪代码

```
MERGE(A, p, q, r)
    n1 ← q-p+1
    n2 ← r-q
    create arrays L[1 .. n1+1] and R[1 .. n2+1]
    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]←∞        // To avoid having to check whether either pile is empty in each
    R[n2+1]←∞        // basic step, a sentinel card is put on the bottom of each pile.
    i←1
    j←1
    for k←p to r
        do if L[i]≤R[j]
              then A[k]←L[i]
                   i←i+1
              else A[k]←R[j]
                   j←j+1
```





合并过程

L/R	(1)	(2)	(3)	(4)	(1)	(2)	(3)	(4)
V	3	20	22	26	10	12	14	16

i=1 j=1

k=9

A	(9)	(10)	(11)	(12)	(13)	(14)	(15)	(16)
V	3							

i=1　j=1　k=9

合并过程

L/R	(1)	(2)	(3)	(4)	(1)	(2)	(3)	(4)
V	3	20	22	26	10	12	14	16

i=2　　j=1

k=10

A	(9)	(10)	(11)	(12)	(13)	(14)	(15)	(16)
V	3	10						

i=2　　j=1　　k=10

合并过程

L/R	(1)	(2)	(3)	(4)	(1)	(2)	(3)	(4)
V	3	20	22	26	10	12	14	16

i=2 j=2

k=11

A	(9)	(10)	(11)	(12)	(13)	(14)	(15)	(16)
V	3	10	12					

i=2 j=2 k=11

合并过程

L/R	(1)	(2)	(3)	(4)	(1)	(2)	(3)	(4)
V	3	20	22	26	10	12	14	16

A	(9)	(10)	(11)	(12)	(13)	(14)	(15)	(16)
V	3	10	12	14				

i=2　j=3　k=12

合并过程

L/R	(1)	(2)	(3)	(4)	(1)	(2)	(3)	(4)
V	3	20	22	26	10	12	14	16

i=2 j=4

k=13

A	(9)	(10)	(11)	(12)	(13)	(14)	(15)	(16)
V	3	10	12	14	16			

i=2 j=4 k=13

合并过程

L/R	(1)	(2)	(3)	(4)	(1)	(2)	(3)	(4)
V	3	20	22	26	10	12	14	16

∞

i=2

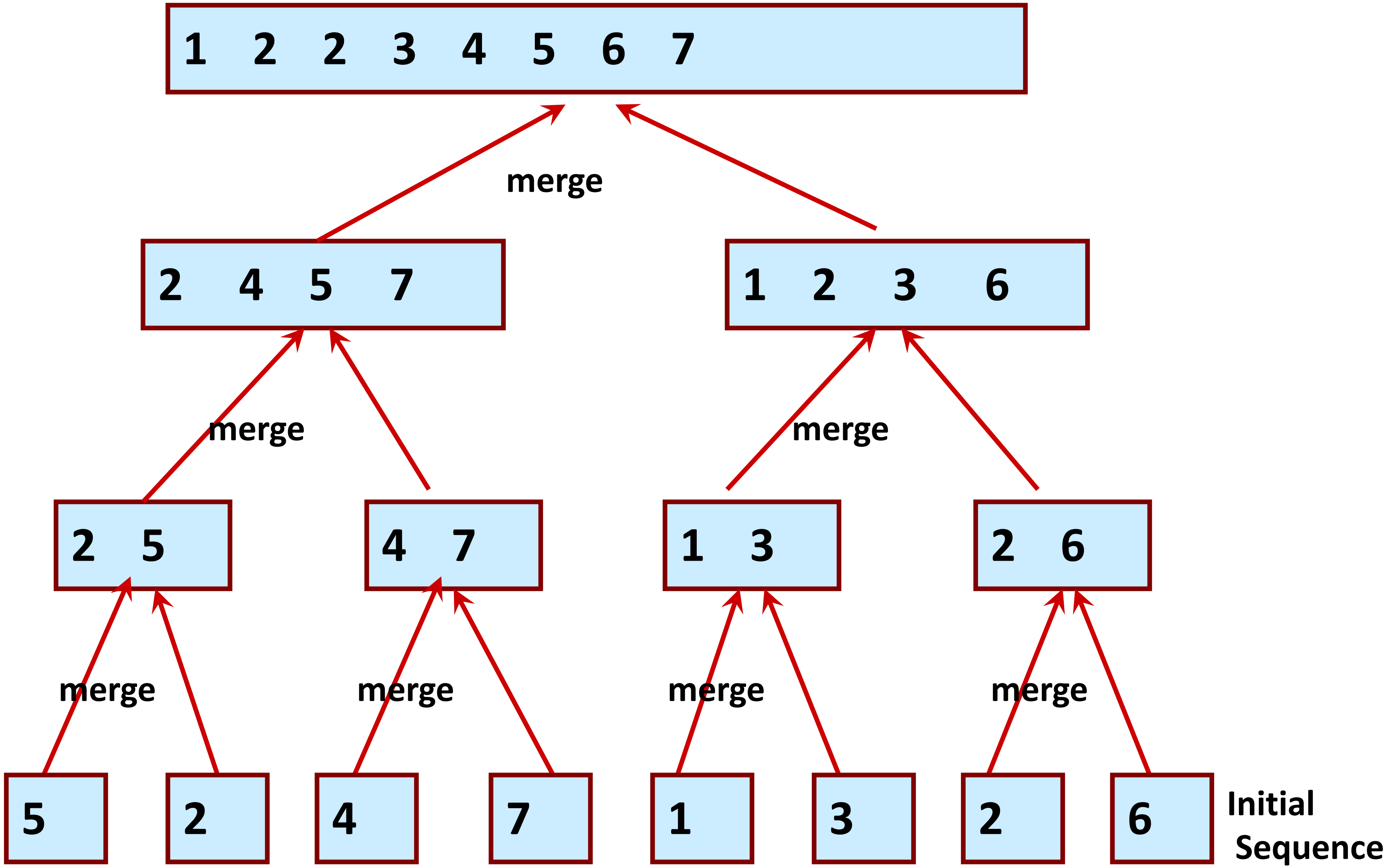
k=14

j=5

A	(9)	(10)	(11)	(12)	(13)	(14)	(15)	(16)
V	3	10	12	14	16	20	22	26

i=2　j=5　k=14

合并排序算法


1 2 2 3 4 5 6 7
merge
2 4 5 7
1 2 3 6
merge
merge
2 5
4 7
1 3
2 6
merge
merge
merge
merge
5
2
4
7
1
3
2
6
Initial
Sequence

合并过程分析

MERGE(A, p, q, r)	cost	times
$n_1 \leftarrow q-p+1$	c	1
$n_2 \leftarrow r-q$	c	1
create arrays $L[1 .. n_1+1]$ and $R[1 .. n_2+1]$	c	1
for $i \leftarrow 1$ **to** n_1	c	n_1+1
do $L[i] \leftarrow A[p+i-1]$	c	n_1
for $j \leftarrow 1$ **to** n_2	c	n_2+1
do $R[j] \leftarrow A[q+j]$	c	n_2
$L[n_1+1] \leftarrow \infty$	c	1
$R[n_2+1] \leftarrow \infty$	c	1
$i \leftarrow 1$	c	1
$j \leftarrow 1$	c	1
for $k \leftarrow p$ **to** r	c	$r-p+2$
do if $L[i] \leq R[j]$	c	$r-p+1$
then $A[k] \leftarrow L[i]$	c	x
$i \leftarrow i+1$	c	x
else $A[k] \leftarrow R[j]$	c	$r-p+1-x$
$j \leftarrow j+1$	c	$r-p+1-x$

$\boldsymbol{r-p+1}$

$\boldsymbol{= n_1 + n_2 = n}$

$\boldsymbol{1 =< x =< n_1}$

$\boldsymbol{\Theta(n_1+n_2)=\Theta(n)}$

合并排序算法伪代码

MERGE-SORT(A, p, r)

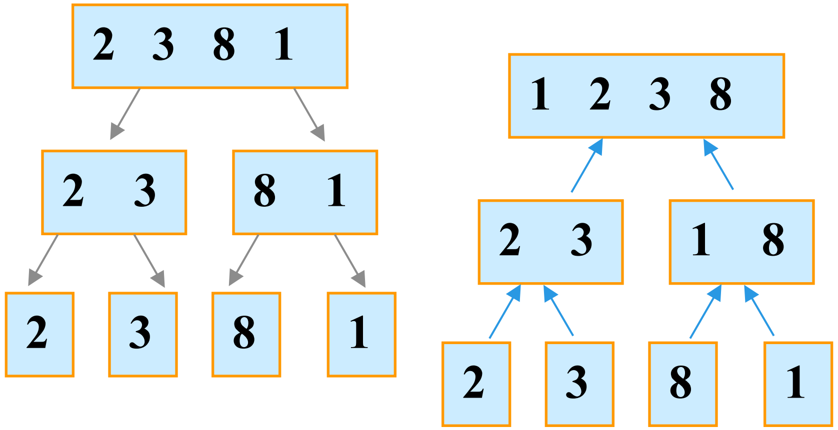
1 **if** $p < r$

2   **Then** $q \leftarrow \lfloor (p+r)/2 \rfloor$

3   MERGE-SORT(A, p, q)

4   MERGE-SORT(A, q+1, r)

5   MERGE(A, p, q, r)



MERGE-SORT(A, 1, 4)

1 if $1 < 4$

2   Then $q \leftarrow \lfloor (1+4)/2 \rfloor = 2$

3   MERGE-SORT(A, 1, 2)

        1 if $1 < 2$

        2 Then $q \leftarrow \lfloor (1+2)/2 \rfloor = 1$

        3 MERGE-SORT(A, 1, 1)

            1 if $1 < 1$

        4 MERGE-SORT(A, 2, 2)

            1 if $2 < 2$

        5 MERGE(A, 1, 1, 2)

4   MERGE-SORT(A, 3, 4)

        1 if $3 < 4$

        2 Then $q \leftarrow \lfloor (3+4)/2 \rfloor = 3$

        3 MERGE-SORT(A, 3, 3)

            1 if $3 < 3$

        4 MERGE-SORT(A, 4, 4)

            1 if $4 < 4$

        5 MERGE(A, 3, 3, 4)

5   MERGE(A, 1, 2, 4)

分治算法分析

- 当一个算法包含一个递归地调用自己的过程，其运行时间通常用**递归式**描述
- 递归式通过描述输入规模较小的子问题来描述输入规模为n 的原问题

```
MERGE-SORT(A, p, r)
1   if p < r
2       Then q ←
3           MERGE-SORT(A, p, q)
4           MERGE-SORT(A, q+1, r)
5           MERGE(A, p, q, r)
```

$$\lfloor (p+r)/2 \rfloor$$

分治算法分析

- 假设 **T(n)**为输入规模为 **n** 问题的运行时间
- 假设将原问题分解为**a** 个子问题，每个子问题大小为原问题的**1/b**
- **D(n):** 将原问题分解为子问题所需时间
- **C(n):** 将子问题的解合并为原问题的解所需时间

递归式

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

Θ-Notation

合并排序算法分析

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

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

- 合并排序算法最坏情况的运行时间$T(n)$的递归式？

  当n=1, 花费常数时间

  当 n>1,

  - **分解**: 计算子序列的中间位置

    one step. $D(n)=\Theta(1)$.
  - **解决**: 递归地解决两个子问题，每个大小$n/2$，分别需要$2T(n/2)$ 运行时间
  - **合并**: MERGE 过程需要$C(n)=\Theta(n)$.

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

合并排序算法分析

- 合并排序算法需要花费多长时间?
  - -- 瓶颈 = **merging (and copying).**
    - >> 合并两个大小**n/2** 的文件需要**n** 次比较
  - -- **T(n) = n**个元素的合并排序时间
    - >>为方便分析，假设**n**是**2**的幂次

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ \underbrace{2T(n/2)}_{\text{Sorting both halves}} + \underbrace{\Theta(n)}_{\text{merging}} & \text{otherwise} \end{cases}$$

- **结论：T(n) = n $\lg_2 n$**
  - -- 注意: 合并排序对任意文件都是相同的比较次数
    - >> 已排序序列也一样

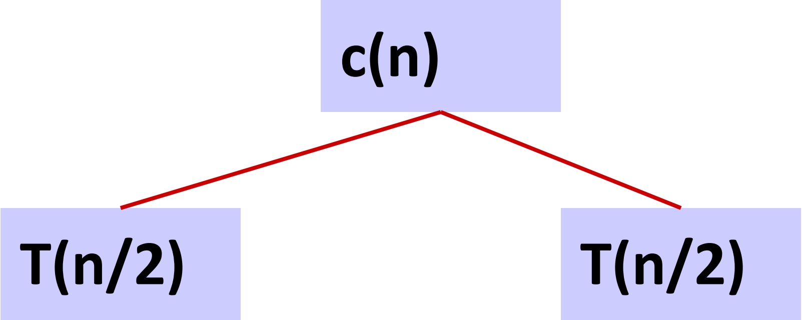
利用递归树进行算法效率分析

$$T(n) = \begin{cases} \Theta(1) & \textit{if } n = 1 \\ \underbrace{2T(n/2)}_{\textit{Sorting both halves}} + \underbrace{\Theta(n)}_{\textit{merging}} & \textit{otherwise} \end{cases}$$

T(n)

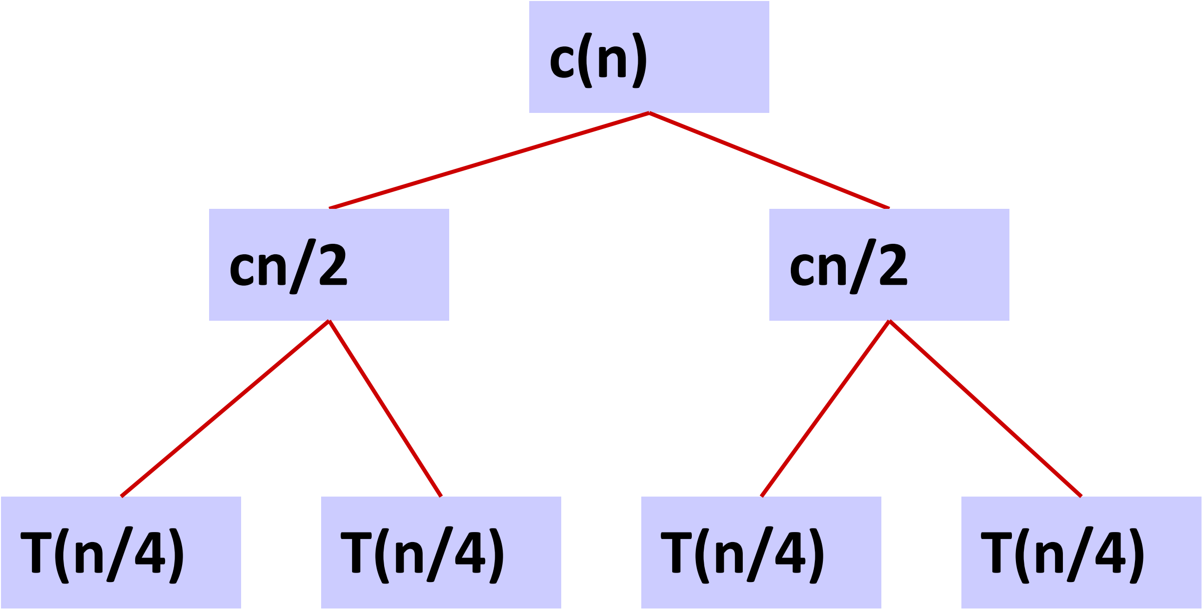
利用递归树进行算法效率分析

$$T(n)=\begin{cases}\Theta(1) & if\ n=1\\ \underbrace{2T(n/2)}_{Sorting\ both\ halves}+\underbrace{\Theta(n)}_{merging} & otherwise\end{cases}$$

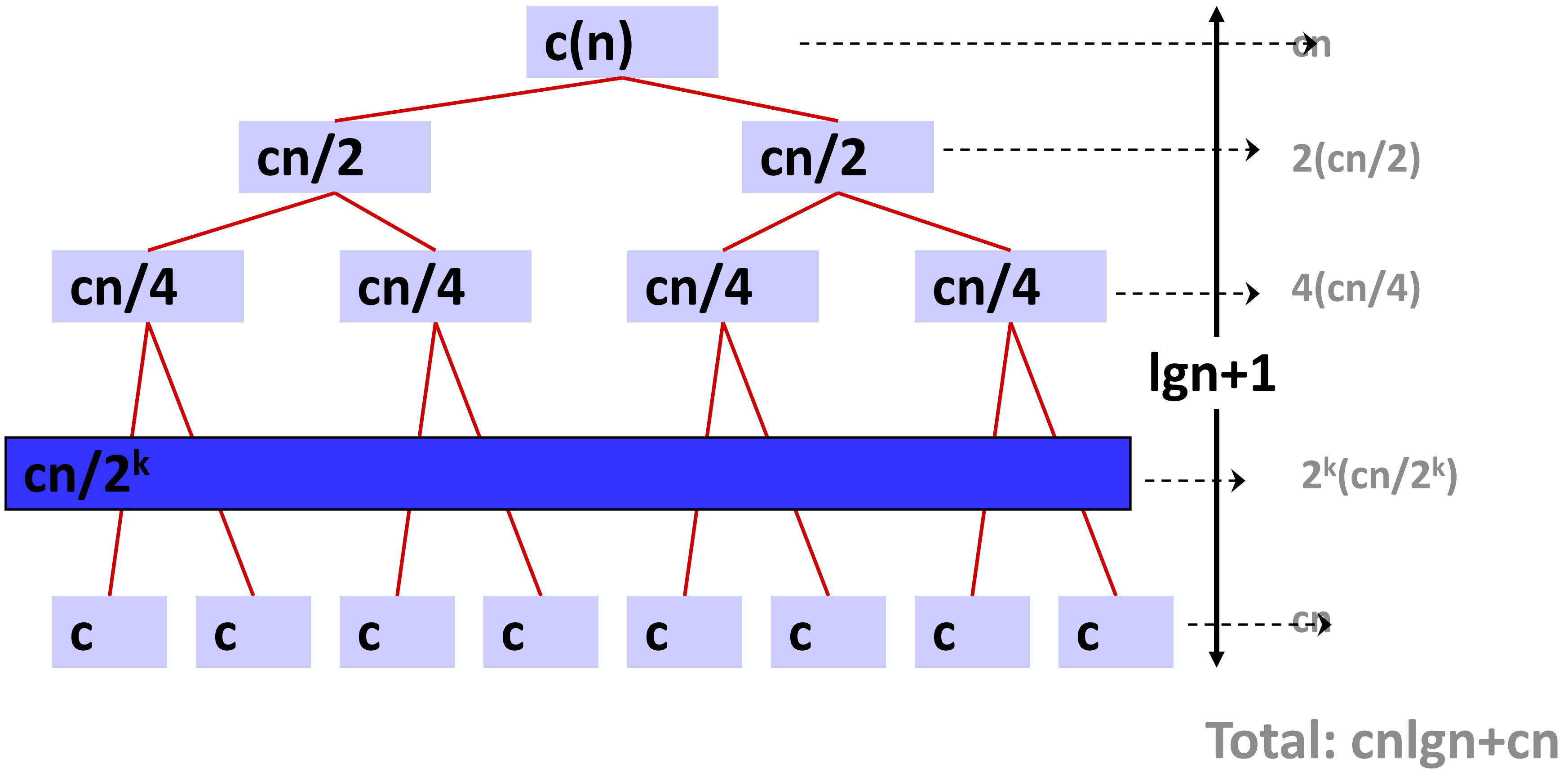


利用递归树进行算法效率分析

$$T(n)=\begin{cases}\Theta(1) & \text{if } n=1\\ \underbrace{2T(n/2)}_{\text{Sorting both halves}}+\underbrace{\Theta(n)}_{\text{merging}} & \text{otherwise}\end{cases}$$



利用递归树进行算法效率分析



完全二叉树，有 lgn+1 层，每层需要cn 的时间花费，总花费为**Θ(nlgn)**

利用归纳法证明

- Claim. $T(n)=n\log_2 n$ (when n is a power of 2).

$$T(n)=\begin{cases}\Theta(1) & \text{if } n=1\\ \underbrace{2T(n/2)}_{\text{Sorting both halves}}+\underbrace{\Theta(n)}_{\text{merging}} & \text{otherwise}\end{cases}$$

$$\begin{aligned}T(2n)&=2T(n)+2n\\&=2n\log_2 n+2n\\&=2n(\log_2(2n)-1)+2n\\&=2n\log_2(2n)\end{aligned}$$

- Proof. (by induction on n)

  --Base case: n = 1.

  --Inductive hypothesis: $T(n) = n\log_2 n$.

  --Goal: show that $T(2n) = 2n\log_2(2n)$

合并排序算法分析

-- 基本操作

>>计算比较次数

-- 上界（最坏情况）：

>>$n\log_2 n$

--下界

>>$n\log_2 n$ 、 $n\log_2 e$

--优化算法: 下界 ～上界

>>合并排序算法

Part2：递归与分治策略（I）

- 减治法：插入排序
- 分治法：归并排序
- 递归分析方法

递归的概念

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。

由分治法产生的子问题往往是原问题的较小模式，这就为使用递归技术提供了方便。在这种情况下，反复应用分治手段，可以使子问题与原问题类型一致而其规模却不断缩小，最终使子问题缩小到很容易直接求出其解。这自然导致递归过程的产生。

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。

递归的概念

直接递归

```
fun_a()
{  …
 fun_a()  …
}
```

间接递归

```
fun_a()
{  …
fun_b()
…}
```

```
fun_b()
{  …
fun_a()
 …}
```

递归的概念

例1　阶乘函数

阶乘函数可递归地定义为：

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

边界条件

递归方程

边界条件与递归方程是递归函数的二个要素，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。

递归的概念

引例

- 对于任意非负整数n，计算阶乘函数$F(n) = n!$的值；
- 当$n > 1$时，$n! = 1 * 2 * \ldots * (n-1) * n = (n-1)! * n$，并且$n! = 1$；
- 可以使用递归的方法计算$F(n) = F(n-1) * n$。

伪代码

```
function F(n)
    if n = 0 then
        return 1
    else
        return F(n − 1) ∗ n
    end if
end function
```

乘法执行次数$M(n)$

- $n > 0$时，$M(n) = M(n-1) + 1$
- 计算$F(n-1)$需要$M(n-1)$次乘法
- 计算$F(n-1) * n$需要1次乘法
- $n = 0$时，$M(0) = 0$，不需要乘法
- 替换法：$M(n) = M(n-i) + i = n$

递归的概念

例2 Fibonacci数列

无穷数列1，1，2，3，5，8，13，21，34，55，……，称为Fibonacci数列。它可以递归地定义为：

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

边界条件

递归方程

第n个Fibonacci数可递归地计算如下：

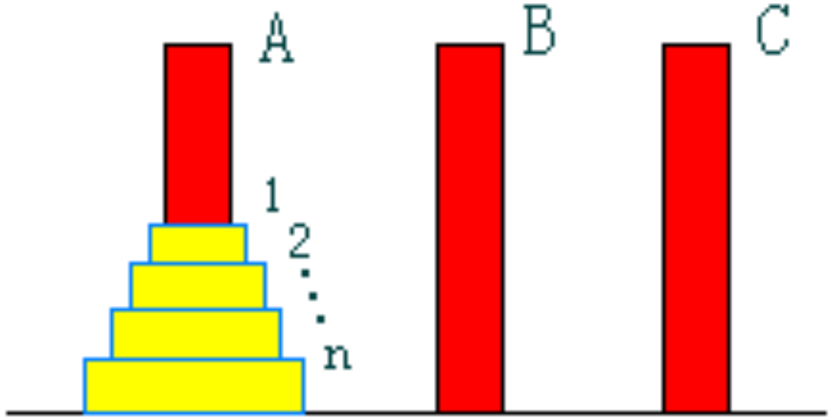
```
int fibonacci(int n)
   {
       if (n <= 1) return 1;
       return fibonacci(n-1)+fibonacci(n-2);
   }
```

递归的概念

例3　Hanoi塔问题

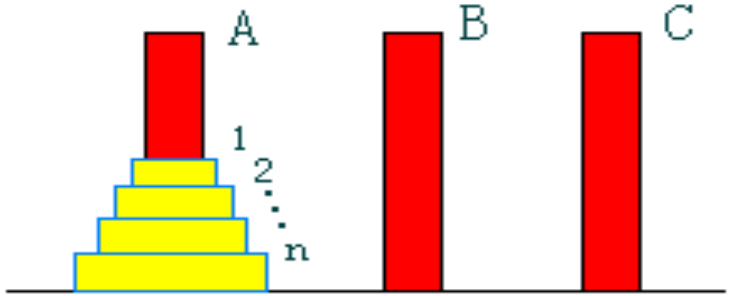
设a, b, c是3个塔座。开始时，在塔座a上有一叠共n个圆盘，这些圆盘自下而上，由大到小地叠在一起。各圆盘从小到大编号为1, 2, …, n，现要求将塔座a上的这一叠圆盘移到塔座b上，并仍按同样顺序叠置。在移动圆盘时应遵守以下移动规则：

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。



递归的概念

例6　Hanoi塔问题




```
void hanoi(int n, int a, int b, int c)
    {
        if (n > 0)
        {
            hanoi(n-1, a, c, b);
            move(a,b);
            hanoi(n-1, c, b, a);
        }
    }
```

递归方程的求解

- **置换法**：不断展开为级数，并求和。
- **递归树法**：（迭代法的另一种表示)
- **主定理，母函数法**（**Master method**）：提供形如$T(n) = aT(n/b)+f(n)$递归方程的通解, 其中 $a \geq 1$, $b > 1$，$f(n)$ 是给定函数.

递归方程的求解——替代法

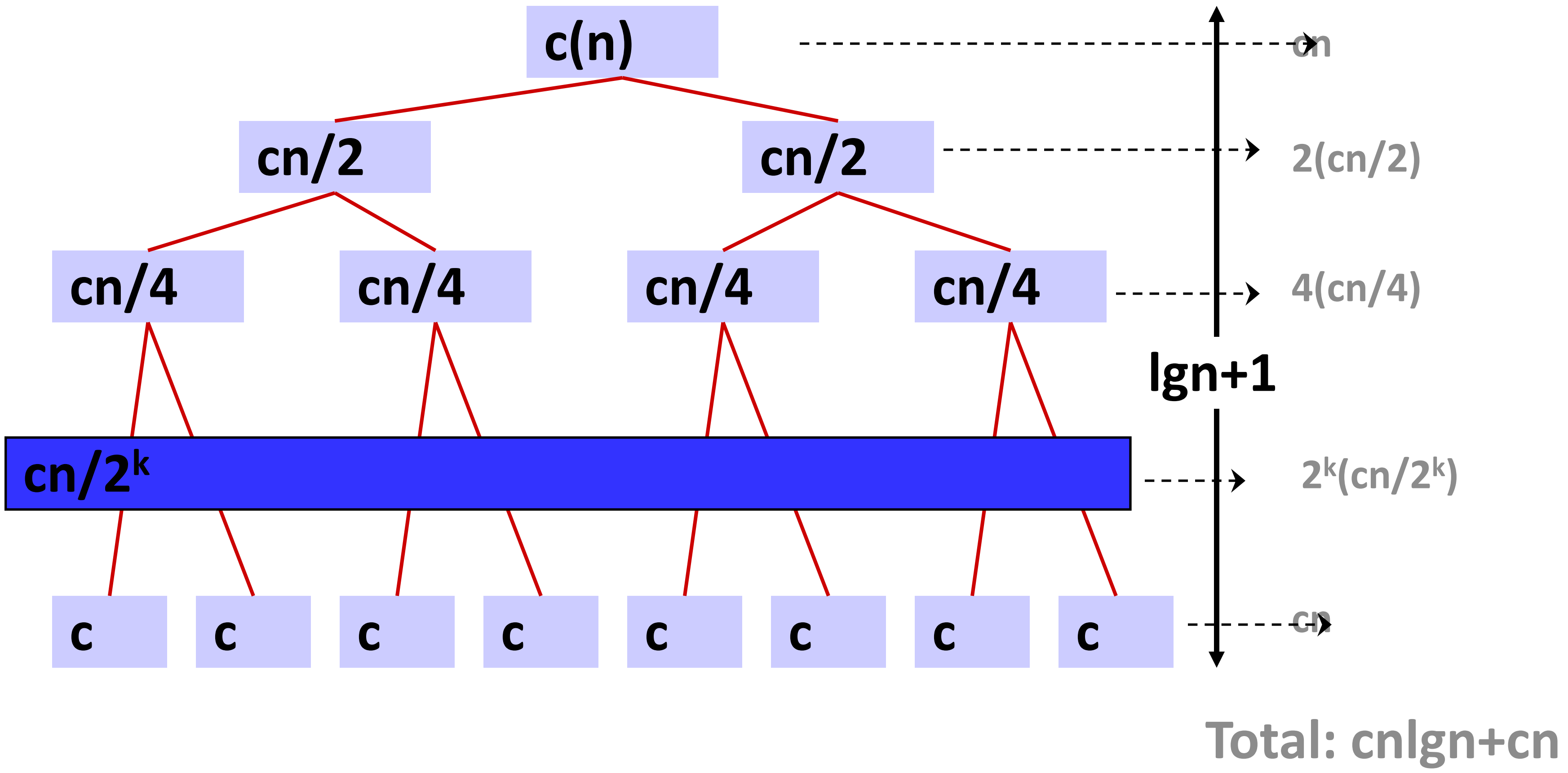
例1：计算$W(n)$，$W(n) = W(n-1) + n - 1, W(1) = 0$

$$\begin{aligned} W(n) &= W(n-1) + n - 1 = W(n-2) + (n-2) + (n-1) \\ &= W(n-3) + (n-3) + (n-2) + (n-1) = \ldots\ldots \\ &= W(1) + 1 + 2 + \ldots + (n-2) + (n-1) = 1 + 2 + \ldots + (n-2) + (n-1) \\ &= n(n-1)/2 \end{aligned}$$

例2：计算$W(n)$，$W(n) = 2W(n/2) + n - 1, n = 2^k, W(1) = 0$

$$\begin{aligned} W(n) &= 2W(2^{k-1}) + 2^k - 1 = 2[2W(2^{k-2}) + 2^{k-1} - 1] + 2^k - 1 \\ &= 2^2 W(2^{k-2}) + 2^k - 2 + 2^k - 1 \\ &= 2^2 [2W(2^{k-3}) + 2^{k-2} - 1] + 2^k - 2 + 2^k - 1 \\ &= 2^3 W(2^{k-3}) + 2^k - 2^2 + 2^k - 2 + 2^k - 1 = \ldots\ldots \\ &= 2^k W(1) + k2^k - (2^{k-1} + 2^{k-2} + \ldots + 2 + 1) = k2^k - 2^k + 1 \\ &= nlogn - n + 1 \end{aligned}$$

递归方程的求解——替代法



完全二叉树，有 lgn+1 层，每层需要cn 的时间花费，总花费为**Θ(nlgn)**

递归方程的求解——主定理

主定理

设$T(n)$是一个非递减函数（定义见课本P376），并且满足递推式

$$T(n) = aT(n/b) + f(n)\text{，其中}\ n = b^k\text{，}\ k = 1, 2, \ldots$$

$$T(1) = c$$

其中$a \geqslant 1$，$b \geqslant 2$，$c > 0$，如果$f(n) \in \Theta(n^d)$，$d \geqslant 0$，那么

- 当$a < b^d$时，$T(n) \in \Theta(n^d)$；
- 当$a = b^d$时，$T(n) \in \Theta(n^d logn)$；　　**结论对符号$O$和$\Omega$也成立。**
- 当$a > b^d$时，$T(n) \in \Theta(n^{log_b a})$；

递归方程的求解

求解如下递归方程

1. $T(n) = 9T(n/3) + n$
2. $T(n) = T(n/2) + n$
3. $T(n) = T(n/3) + 1$

答案

1. $a = 9$，$b = 3$，$f(n) = n \in \Theta(n)$，即$d = 1$
   由于$a = 9 > b^d = 3^1$，所以$T(n) \in \Theta(n^{log_b a}) = \Theta(n^2)$
2. $a = 1$，$b = 2$，$f(n) = n \in \Theta(n)$，即$d = 1$
   由于$a = 1 < b^d = 2^1$，所以$T(n) \in \Theta(n^d) = \Theta(n)$
3. $a = 1$，$b = 3$，$f(n) = 1 \in \Theta(1)$，即$d = 0$
   由于$a = 1 = b^d = 3^0$，所以$T(n) \in \Theta(n^d logn) = \Theta(logn)$

常见递归类型

减一算法

- 算法利用一个规模为n的实例和规模为$n-1$的给定实例之间的关系来对问题求解（插入排序，课本4.1）。
- $T(n) = T(n-1) + f(n)$

减常因子算法

- 规模为n的实例化简为一个规模为n/b的给定实例来求解（俄式乘法）。
- $T(n) = T(n/b) + f(n)$

分治算法

- 给定实例划分为若干个较小的实例，对每个实例递归求解，然后再把较小的实例合并成给定实例的一个解（快速排序，合并排序）。
- $T(n) = aT(n/b) + f(n)$

Examples:

(1) $T(n)=\begin{cases}1 & \text{, if } n=1,\\ T(n-1)+1 & \text{, if } n>1.\end{cases}$

Solution: $T(n)=n.$

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

Solution: $T(n)=n\lg n+n.$

(3) $T(n)=\begin{cases}0 & \text{, if } n=2,\\ T(\sqrt{n})+1 & \text{, if } n>2.\end{cases}$

Solution: $T(n)=\lg\lg n.$

(4) $T(n)=\begin{cases}1 & \text{, if } n=1,\\ T(n/3)+T(2n/3)+n, & \text{if } n>1.\end{cases}$

Solution: $T(n)=\Theta(n\lg n)$

作业 2

- **阅读**
  - **2.4，4.1，5.1-5.2，5.4-5.5**
- **习题**
  - **2.4: 3，4，9**
  - **5.1：8，9**