Introduction to Algorithms

Chapter 4. Divide & Conquer

Dr. He Emil Huang (黄河)
School of Computer Science and Technology
Soochow University

E-mail: huangh@suda.edu.cn

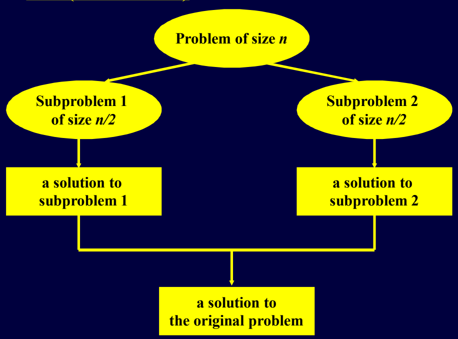
与课本对应关系

教材 Chapter 4 & Chapter 7

Chapter 4. 分治策略

Chapter 7. Quick Sort

- **Main Topics:**
  - 理解递归(Recursion)的概念



- **Main Topics (Cont.):**
  - 掌握设计有效的分治策略算法及时间性能分析 (本章重点讨论)
    - 而时间性能分析其实就是有效分治策略算法设计的依据
  - 通过下面的范例学习分治策略设计技巧

    ①二分搜索技术(Binary Search);
    ②归并排序和快速排序(Merge Sort & Quick Sort);
    ③大整数乘法;
    ④Strassen矩阵乘法;
    ⑤最接近点对问题 Closest pairwise points;

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;
- 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,最终使子问题缩小至很容易直接求出其解的程度时终止。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。

Now, we will give some recursion instances in the following part.

递归式和分治法

- **例1. Def. of factorial function**

阶乘函数可递归地定义为:

边界条件

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

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。

递归式和分治法

■ 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);
}
```

Asymptotically upper bound?

递归式和分治法

■ 例2 Fibonacci数列

除了直接递归以外的另4种求解方案：

方法1：用户自定义一个栈，模拟系统递归调用工作栈
方法2：递推关系式的优化 时间O(n), 空间O(n)
方法3：求解通项公式 时间O(1)
方法4：分治策略 时间$O(\log_2 n)$

Non-recursive Fibonacci Iterative Function

```
int Fibonacci (int n)
/* fibonacci: iterative version*/
{
  int last_but_one; // second previous Fibonacci number, F_{i-2}
  int last_value; // previous Fibonacci number, F_{i-1}
  int current; // current Fibonacci number F_i
  if (n <= 0) return 0;
    else if (n == 1) return 1;
    else {
          last_ but_ one = 0;
          last_ value = 1;
          for (int i = 2; i <= n; i++) {
            current = last_but_one + last_value;
            last_but_ one = last_value;
            last_value = current;
          }
          return current;
    }
}
```

递归式和分治法

例1, 2中的函数都可以找到相应的非递归方式定义：

$$n!=1\cdot 2\cdot 3\cdots(n-1)\cdot n$$

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

下例中的Ackerman函数却无法找到非递归的定义

递归式和分治法

■ 例3 Ackerman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。
Ackerman函数A(n, m)定义如下(双变量函数)：

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

递归式和分治法

■ 例3 Ackerman函数

A(n,m)的自变量m的每一个值都定义了一个单变量函数：

- m=0时，A(n,0)=n+2
- m=1时，A(n,1)=A(A(n-1,1),0)=A(n-1,1)+2，和A(1,1)=2故A(n,1)=2*n
- m=2时，A(n,2)=A(A(n-1,2),1)=2A(n-1,2)，和A(1,2)=A(A(0,2),1)=A(1,1)=2，故A(n,2)= 2^n
- m=3时，类似的可以推出…
- m=4时，A(n,4)的增长速度非常快，以至于没有适当的数学式子来表示这一函数。

递归式和分治法

■ 例3 Ackerman函数

- ❖定义单变量的Ackerman函数A(n)为，A(n)=A(n,n)。
- ❖定义其拟逆函数α(n)为：α(n)=min{k|A(k)≥n}。即α(n)是使n≤A(k)成立的最小的k值。
- ❖α(n)在复杂度分析中常遇到。对于通常所见到的正整数n，有α(n)≤4。但在理论上α(n)没有上界，随着n的增加，它以难以想象的慢速度趋向正无穷大。

递归式和分治法

■ 例4 排列问题

有些问题表面上不是递归定义的，但可通过分析，抽象出递归的定义

设计一个递归算法生成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，$R_i=R-\{r_i\}$。

集合X中元素的全排列记为perm(X)。

perm(X) (r_i)表示在全排列perm(X)的每一个排列后加上后缀得到的排列。

R的全排列可归纳定义如下：

当n=1时，perm(R)=(r)，其中r是集合R中唯一的元素；

当n>1时，perm(R)由$perm(R_n)(r_n)$, $perm(R_{n-1})(r_{n-1})$, …, $perm(R_1)(r_1)$构成。

例4：写一个就地生成n个元素$a_1, a_2, \ldots, a_n$全排列 (n!) 的算法，要求算法终止时保持$a_1, a_2, \ldots, a_n$原状

解：设 A[0...n-1] 基类型为char,"就地"不允许使用 A 以外的数组

① 生成$a_1, a_2, \ldots, a_n$全排列 ⇨ n个子问题

求n-1个元素的全排列 + n^{th}个元素

1^{st}子问题	$a_1, a_2, \ldots, a_{n-1}$	a_n //
2^{nd}子问题	$a_1, \ldots, a_{n-2}, a_n$	a_{n-1} //A[n-2]↔A[n-1]
3^{rd}子问题	$a_1, \ldots, a_n, a_{n-1}$	a_{n-2} //A[n-3]↔A[n-1]
⋮	⋮	⋮
n^{th}子问题	$a_n, a_2 \ldots, a_{n-1}$	a_1 //A[0]↔A[n-1]

② 递归终结分支

当 n=1 时，一个元素全排列只有一种，即为本身。实际上无须进一步递归，可直接打印输出A

15

③ 算法：以A[0..7]为例

```
void  permute (char  A[ ], int  n) { //外部调用时令 n=7
      if (n==0)
          print (A);  // 打印A[0…7]
      else {
         permute(A,n-1); //求A[0..n-1]的全部排列。1st子问题不用交换
         for (i=n-1; i>=0; i--) {
                Swap(A[i], A[n]); // 交换ai 和an 内容，说明为引用
                permute(A,n-1); // 求A[0..n-1] 全排列
                Swap(A[i],A[n]); //交换，恢复原状
         }//endfor
      }//endif
 }
```

时间：

$O(2^n) < n! < O(n^n)$ 所以实验时，n不能太大

16

递归式和分治法

■ 例5 整数划分问题

将正整数n表示成一系列正整数之和：$n=n_1+n_2+\ldots+n_k$，其中$n_1\geq n_2\geq\ldots\geq n_k\geq 1$，k≥1。

正整数n的这种表示称为正整数n的划分。求正整数n的不同划分个数。

例如，正整数6有如下11种不同的划分：

6；
5+1；
4+2，4+1+1；
3+3，3+2+1，3+1+1+1；
2+2+2，2+2+1+1，2+1+1+1+1；
1+1+1+1+1+1。

递归式和分治法

■ 例5 整数划分问题

前面的几个例子中，问题本身都具有比较明显的递归关系，因而容易用递归函数直接求解。在本例中，如果设p(n)为正整数n的划分数，则难以找到递归关系，因此考虑增加一个自变量：将最大加数n_1不大于m的划分个数记作q(n,m)。可以建立q(n,m)的如下递归关系。

递归式和分治法

(1) q(n,1)=1,n≥1;
当最大加数n_1不大于1时，任何正整数n只有一种划分形式，
即n=1+1+……+1

(2) q(n,m)=q(n,n),m≥n;
最大加数n_1实际上不能大于n。 q(1,m)=1。

(3) q(n,n)=1+q(n,n-1);
正整数n的划分由n_1=n的划分和n_1≤n-1的划分组成。

(4) q(n,m)=q(n,m-1)+q(n-m,m),n>m>1;
正整数n的最大加数n_1不大于m的划分由n_1=m的划分和n_1≤m-1的划分组成。

19

递归式和分治法

■ 例5 整数划分问题

前面的几个例子中，问题本身都具有比较明显的递归关系，因而容易用递归函数直接求解。在本例中，如果设p(n)为正整数n的划分数，则难以找到递归关系，因此考虑增加一个自变量：将最大加数n_1不大于m的划分个数记作q(n,m)。可以建立q(n,m)的如下递归关系。

$$q(n,m)=\begin{cases} 1 & n=1, m=1 \\ q(n,n) & n<m \\ 1+q(n,n-1) & n=m \\ q(n,m-1)+q(n-m,m) & n>m>1 \end{cases}$$

正整数n的划分数 $p(n)=q(n,n)$

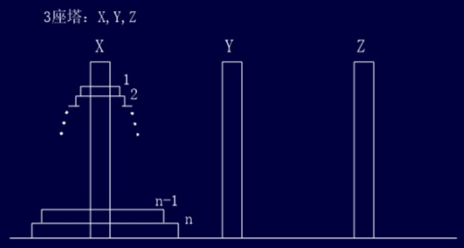
递归式和分治法

■ 例5 整数划分问题

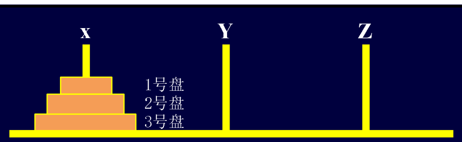
```
int q (int n, int m){
    if((n<1)||(m<1)) return 0;
    if((n==1)||(m==1)) return 1;
    if(n<m) return q(n,n);
    if(n==m) return q(n,m-1)+1;
    return q(n,m-1)+q(n-m,m);
}
```

■ 例6：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：
1. 每次只能移动一片
2. 圆盘可插在X,Y,Z任一塔座上
3. 任一时刻大盘不能压在小盘上



22



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轴移动到z轴上；当n=2时，可先将1号盘子移动到y轴，再将2号盘子移动到z轴，最后将1号盘子移动到z轴；对于一般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 轴移动至 y 轴，可递归求解
Hanoi(n-1, x, z, y)；
(2)将 n号盘从 x轴移动至 z轴；
(3)将1至n-1号盘从y轴移动至z轴，可递归求解
Hanoi(n-1, y, x, z)。

23/44

① 分解
设 n>1
原问题：将n片从X移到Z，Y为辅助塔，可分解为：
I. 将上面n-1 个盘从X移至Y，Z为辅助盘
II. 将 n^{th} 片从X移至 Z
III. 将Y上n-1个盘子移至Z，X为辅助盘

//子问题特征属性与原
//问题相同规模小1，参
//数不同

② 终结条件

n = 1时，直接将编号为1的盘子从 X 移到Z

```
void   Hanoi (int n, char x, char y, char z ) {
 // n个盘子从 X 移至 Z，Y 为辅助
 if ( n==1 )  move(X,1,Z); // 将1号盘子从 X 移至 Z, 打印
 else {
    Hanoi (n-1,x,z,y);  //源X，辅Z，目Y
    move (x,n,z);
    Hanoi (n-1,y,x,z);  //源Y，辅X，目Z
  }
}
```

思考题：如果塔的个数变为4个，将n个圆盘从一个塔移动到另外一个塔，移动规则不变，求移动步数最小的方案

24

分析Hanoi塔问题移动圆盘的次数

设T(n)表示n个圆盘的Hanoi塔问题移动圆盘的次数，显然T(0)=0，对于n>0的一般情况采用如下分治策略：

(1)将1至n-1号盘从X轴移动至Y轴，可递归求解Hanoi(n-1, X, Z, Y)；

(2)将 n号盘从X轴移动至Z轴；

(3)将1至n-1号盘从Y轴移动至Z轴，可递归求解Hanoi(n-1, Y, X, Z)。

在(1)与(3)中需要移动圆盘次数T(n-1)，(2)需要移动一次圆盘。可得如下的关系：

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

展开上式可得：

$$
\begin{aligned}
T(n) &=2T(n-1)+1 \\
&=2[2T(n-2)+1]+1 \\
&=2^2T(n-2)+1+2 \\
&\ldots\ldots \\
&=2^nT(n-n)+1+2+\ldots+2^{n-1} \\
&=2^n-1
\end{aligned}
$$

使用$O(2^n)$限界

25/44

递归式和分治法

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。

递归式和分治法

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

1、采用一个用户定义的栈来模拟系统的递归调用工作栈。该方法通用性强，但本质上还是递归，只不过人工做了本来由编译器做的事情，优化效果不明显。

2、用递推来实现递归函数。

3、通过变换能将一些递归转化为尾递归，从而迭代求出结果。

后两种方法在时空复杂度上均有较大改善，但其适用范围有限。

递归至非递归机械转化

- 机械地将任何一个递归程序转换为与其等价的非递归程序
- 五条规则：
  - (1) 设置一个栈（不妨用 S 表示），并且开始时将其置为空。
  - (2) 在子程序入口处设置一个标号（不妨设为L0）。
  - (3) 对子程序中的每一递归调用，用以下几个等价操作来替换：
    - a) 保留现场：开辟栈顶存储空间，用于保存返回地址（不妨用
    - b) Li，i=1,2,3，…）、调用层中的形参和局部变量的值（最外层调用不必考虑）。
    - c) 准备数据：为被调子程序准备数据，即计算实在参数的值,并赋给对应的形参
    - d) 转入（子程序）执行, 即执行goto L0。
    - e) 在返回处设一个标号Li(i=1,2,3，…)，并根据需要设置以下语句：若函数需要返回值，从回传变量中取出所保存的值并传送到相应的位置。

28

递归至非递归机械转化 (Cont.)

(4) 对返回语句，可用以下几个等价操作来替换：

如果栈不空，则依次执行如下操作，否则结束本子程序，返回。

- a) 回传数据：若函数需要返回值，将其值保存到回传变量中。
- b) 恢复现场：从栈顶取出返回地址（不妨保存到 X 中）及各变量、形参值，并退栈。
- c) 返回：按返回地址返回（即执行goto X）。

(5) 对其中的非递归调用和返回操作可照搬。

29

递归式和分治法

- 作用：分析递归算法的运行时间
- 三种方法 (P37)
  - 替换法、迭代法(递归树法)、通用法(master method)
- 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

  - Divide(分解)：将问题划分为若干个子问题
  - Conquer(求解)：递归地解这些子问题；若子问题Size足够小，则直接解决之
  - Combine(组合)：将子问题的解结合成原问题的解

30

递归式和分治法(续)

其中的第二步：递归调用或直接求解 (递归终结条件)

有的算法“分解”容易，有的则“组合”容易

- 分治法举例

归并排序

①分解：把n个待排序元素划分为两个Size为n/2的子序列

②求解：递归调用归并排序将这两个子序列排序，若子序列长度为1时，已自然有序，无需做任何事情(直接求解)

③组合：将这两个已排序的子序列合并为一个有序的序列

显然，分解容易(一分为二)，组合难。

快速排序

分解难，组合易。 A[1…k-1] ≤ A[k] ≤ A[k+1…n]

Mergesort

- Split array A[0..*n*-1] in two about equal halves and make copies of each half in arrays B and C
- Sort arrays B and C recursively
- Merge sorted arrays B and C into array A as follows:
  - Repeat the following until no elements remain in one of the arrays:
    - compare the first elements in the remaining unprocessed portions of the arrays
    - copy the smaller of the two into A, while incrementing the index indicating the unprocessed portion of that array
  - Once all elements in one of the arrays are processed, copy the remaining unprocessed elements from the other array into A.

Pseudocode of Mergesort

ALGORITHM *Mergesort*$(A[0..n-1])$

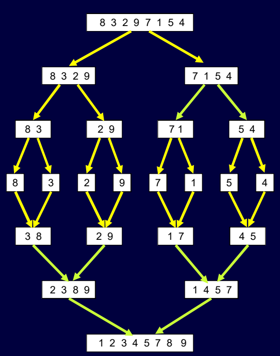
//Sorts array $A[0..n-1]$ by recursive mergesort
//Input: An array $A[0..n-1]$ of orderable elements
//Output: Array $A[0..n-1]$ sorted in nondecreasing order
if $n > 1$
  copy $A[0..\lfloor n/2\rfloor - 1]$ to $B[0..\lfloor n/2\rfloor - 1]$
  copy $A[\lfloor n/2\rfloor..n-1]$ to $C[0..\lceil n/2\rceil - 1]$
  Mergesort$(B[0..\lfloor n/2\rfloor - 1])$
  Mergesort$(C[0..\lceil n/2\rceil - 1])$
  Merge(B, C, A)

Pseudocode of Merge

ALGORITHM *Merge*$(B[0..p-1], C[0..q-1], A[0..p+q-1])$

//Merges two sorted arrays into one sorted array
//Input: Arrays $B[0..p-1]$ and $C[0..q-1]$ both sorted
//Output: Sorted array $A[0..p+q-1]$ of the elements of B and C
$i \leftarrow 0$; $j \leftarrow 0$; $k \leftarrow 0$
while $i < p$ **and** $j < q$ **do**
  if $B[i] \le C[j]$
    $A[k] \leftarrow B[i]$; $i \leftarrow i + 1$
  else $A[k] \leftarrow C[j]$; $j \leftarrow j + 1$
  $k \leftarrow k + 1$
if $i = p$
  copy $C[j..q-1]$ to $A[k..p+q-1]$
else copy $B[i..p-1]$ to $A[k..p+q-1]$

Mergesort Example



Analysis of Mergesort

- <u>All cases have same efficiency</u>: $\Theta(n \log n)$
- Number of comparisons in the worst case is close to theoretical minimum for comparison-based sorting:

It was proved that any sorting method that uses comparisons of keys must do at least

$$\lceil \log_2 n! \rceil \approx n \log_2 n - 1.44n$$

Comparisons of keys (P107~108 textbook)

- Space requirement: $\Theta(n)$ (<u>not</u> in-place)
- Can be implemented without recursion (bottom-up)

递归式和分治法(续)

- 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。

递归式和分治法(续)

- 分治算法时间性能分析
  - 设$T(n)$是Size为n的执行时间，若Size足够小，如$n \le C$（常数），则直接求解的时间为$\theta(1)$
  - ①设完成划分的时间为$D(n)$
  - ②设分解时，划分为a个子问题，每个子问题为原问题的1/b，则解各子问题的时间为$aT(n/b)$
  - ③设组合时间$C(n)$

$$\therefore T(n)=\begin{cases}\theta(1) & \text{if } n \le c \quad //\text{边界} \\ aT(n/b)+D(n)+C(n) & \text{otherwise} \quad // \text{ n/b<n,否则无限递归}\end{cases}$$

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

38

递归式和分治法(续)

- 分治算法分析(续)
  - 一般地，解递归式(Recurrence,定义见P37)时可忽略细节
  - ① 假定函数参数为整数，如 $2T(n/2)$应为$T(\lceil n/2 \rceil)$或$T(\lfloor n/2 \rfloor)$
  - ② 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(P38)

但下面讨论时，我们尽量注意细节！

39

§ 4.1 替换法 (代入法, Page 47~49)

- 1)猜测解; 2) 用数学归纳法确定常数C，证明解正确
- Key：用猜测的解代入到递归式中。

例1：

确定$T(n)=2T(\lfloor n/2 \rfloor)+n$的上界

猜测解$T(n)=O(n\lg n)$　　假定对于所有正数m, 满足m<n均成立

要证$T(n) \le cn\lg n$，对某个常数$c>0$成立

假定它对于$\lfloor n/2 \rfloor$成立，i.e.，$T(\lfloor n/2 \rfloor) \le c\lfloor n/2 \rfloor \lg \lfloor n/2 \rfloor$，

将它代入递归式中

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

$\le cn\lg(n/2)+n$

$=cn\lg n-cn\lg 2+n=cn\lg n-cn+n \le cn\lg n$　只要$c \ge 1$

40

§ 4.1 替换法(续)

- 例1(续)

下面证此解对边界条件亦成立

数学归纳法要求证明解在边界条件下也成立

假定$T(0)=0, T(1)=1$,

而$T(1) \le C \cdot 1 \cdot \lg 1 = 0$不成立

但渐近界只要证$T(n) \le cn\lg n \;\; for \;\; n \ge n_0$就行了

$\therefore T(2)=2T(1)+2=4$

$T(2) \le C \cdot 2 \cdot \lg 2 = 2C$　只要 $c \ge 2$即可

41

§ 4.1 替换法(续)

1. 做出好的猜测(没有一般方法，只能凭经验)
   - ① 与见过的解类似，则猜测之。例如：

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

当n足够大时，$\lfloor n/2 \rfloor$和$\lfloor n/2 \rfloor+17$相差无几，故上界应为$cn\lg n$

   - ② 先证较宽松的上、下界，减小猜测范围。例如：

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

显然，$T(n)=\Omega(n)$ ∵式中有“n”这个项

$T(n)=O(n^2)$ ∵最多分解$O(n)$次，每次时间为n

然后降低上界，升高下界，使它收敛于渐紧界 $T(n)=\theta(n\lg n)$

42

§ 4.1 替换法(续)

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，这是因为数学归纳法没有强大到足以证明其细节。这时可从猜测解中减去一个低阶项以使数学归纳法得以满足
- 例：

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

显然，该解是 $O(n)$，即证明 $T(n)\le cn$

$pf: T(n)\le c(\lfloor n/2 \rfloor)+c(\lceil n/2 \rceil)+1$ //由归纳假设代入

$=cn+1$ //并不蕴含 $T(n)\le cn$

从解中减去一个常数猜测为：$T(n)\le cn-b$ //常数 $b\ge 0$

$pf: T(n)\le (c\lfloor n/2 \rfloor-b)+(c\lceil n/2 \rceil-b)+1$

$=cn-2b+1\le cn-b$ //只要b≥1，c>0

§ 4.1 替换法(续)

3. 避免陷阱

- 与求和式的数学归纳法类似，证明时渐近记号的使用易产生错误。
- 例：

设 $T(n)=2T(\lfloor n/2 \rfloor)+n$

猜测 $T(n)\le cn$ //正确应为 $n\lg n$

$pf: T(n)\le 2(c\lfloor n/2 \rfloor)+n$

$\le cn+n$

$\le 0(n)$ //wrong！须证明 $T(n)\le cn$ 的精确形式

§ 4.1 替换法(续)

4. 变量变换

- 有时改动变量能使未知递归式变为熟悉的式子。例如：

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

令 $m=\lg n$，$2^m=n$

得 $T(2^m)=2T(2^{m/2})+m$

再令 $S(m)=T(2^m)$

得 $S(m)=2S(m/2)+m$ //例1的形式

$\therefore S(m)=O(m\lg m)$。将其改回到T的形式

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

§ 4.2 迭代法(包含递归树方法求解递归式)

① 展开

- 无须猜测，展开递归式，使其成为仅依赖于n和边界条件的和式，然后用求和方法定界。
- 例：$T(n)=3T(\lfloor n/4 \rfloor)+n$

§ 4.2 迭代法

$T(n)=3T(\lfloor n/4 \rfloor)+n$

$=n+3(\lfloor n/4 \rfloor+3T(\lfloor n/4^2 \rfloor))$ //$\lfloor \lfloor n/4 \rfloor/4 \rfloor=\lfloor n/16 \rfloor$

$=n+3\lfloor n/4 \rfloor+3^2T(\lfloor n/4^2 \rfloor)=\ldots$ //再展开一次

$=n+3\lfloor n/4 \rfloor+3^2\lfloor n/4^2 \rfloor+3^3T(\lfloor n/4^3 \rfloor)$

已知规律，无须继续展开，要迭代展开多少次才能达其边界？取决于自变量的大小。不妨设最后项为 i^{th} 项:$3^iT(\lfloor n/4^i \rfloor)$，边界应为 $\lfloor n/4^i \rfloor\le 1$，即 $i\ge \log_4 n$

$\therefore$ 当 $i=\log_4 n$ 时，有 $T(1)=\theta(1)$

§ 4.2 迭代法(续)

① 展开(续)

- 例：(接上页)

$T(n)\le n+3n/4+3^2n/4^2+\ldots+3^{\log_4 n}\theta(1)$

$\le n\sum_{i=0}^{\infty}\left(\frac{3}{4}\right)^i+\theta(n^{\log_4 3})$ //Note: $3^{\log_4 n}=n^{\log_4 3}$

$=4n+o(n)$ //小o

$=O(n)$ //大O

§ 4.2 迭代法(续)

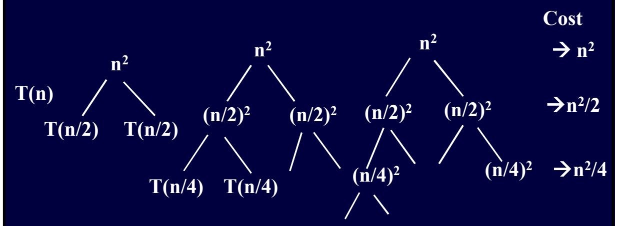
① 展开(续)

- **Keys**
  - 达到边界条件所需的迭代次数
  - 迭代过程中的和式。若在迭代过程中已估计出解的形式，亦可用替换法
  - 当递归式中包含floor和ceiling函数时，常假定参数n为一个整数次幂，以简化问题。例如上例可假定$n=4^k(k \geq 0$的整数)，但这样T(n)的界只对4的整数幂成立。下节方法可克服此缺陷。

§ 4.2 迭代法(续)

② 递归树

- 使展开过程直观化
- 例：$T(n)=2T(n/2)+n^2$ (不妨设$n=2^k$)



§ 4.2 迭代法(续)

② 递归树(续)

- 例：$T(n)=2T(n/2)+n^2$ (续)

  树高(层数)：树中最长路径，求总成本时和式的项数

  令：$(n/2^k)^2 = 1 \rightarrow n = 2^k \rightarrow k = \lg n$

  树高：$\lg n + 1$

  总成本：$\theta(n^2)$
- 例：更复杂，树不一定是满二叉树，叶子深度不尽相同

  $T(n)=T(n/3)+T(2n/3)+n$

  Fig.4.6

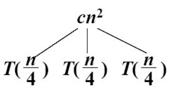
The Construction of a Recursion Tree

- Solve $T(n) = 3T(n/4) + \Theta(n^2)$, we have

$T(n)$

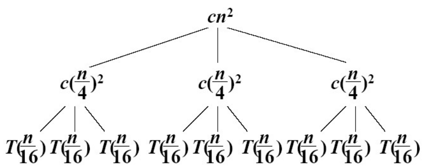
The Construction of a Recursion Tree

- Solve $T(n) = 3T(n/4) + \Theta(n^2)$, we have

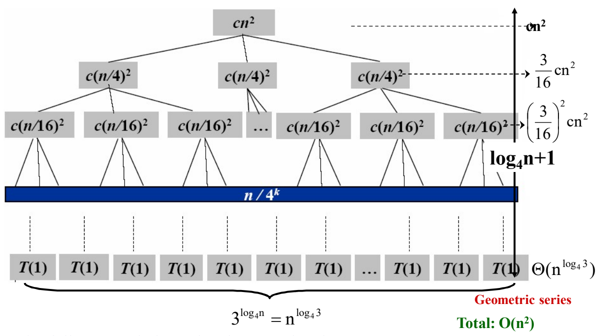


The Construction of a Recursion Tree

- Solve $T(n) = 3T(n/4) + \Theta(n^2)$, we have



Construction of Recursion Tree



- The fully expanded tree has $\log_4 n+1$ levels

55

§ 4.1 The master method (通用法，万能法)

- 可迅速求解
  - T(n)=aT(n/b)+f(n) //常数a ≥1, b>1, f(n)渐近正
  - 物理意义：将Size为n的问题划分为a个子问题，每个子问题 Size 为 n/b 。每个子问题的时间为 T(n/b)，划分和combine的时间为f(n)。
  - Note：n/b不一定为整数，应为⌊n/b⌋或⌈n/b⌉，不会影响渐近界。
- Th4.1 (master theorem *Page 53*)

  设a≥1，b>1是整数，f(n)是函数，T(n)是定义在非负整数上的递归方程T(n)=aT(n/b)+f(n),这里n/b解释为⌊n/b⌋或⌈n/b⌉,则T(n)的渐近界为：

56

§ 4.3 The master method(通用法，万能法)(续)

- Th4.1(master theorem)

1. 若$f(n)=O(n^{\log_b a-\varepsilon})$对某一常数$\varepsilon>0$成立，

   则 $T(n)=\theta(n^{\log_b a})$

2. 若$f(n)=\theta(n^{\log_b a})$则 $T(n)=\theta(n^{\log_b a}\cdot\lg n)$

3. 若$f(n)=\Omega(n^{\log_b a+\varepsilon})$对某一常数$\varepsilon>0$成立，

   且$af(n/b)\le cf(n)$对某常数$c<1$及足够大的n成立,则 $T(n)=\theta(f(n))$

证明从略

57

§ 4.3 The master method(通用法，万能法)(续)

- 该定理意义
  - 比较f(n)和 $n^{\log_b a}$，直观上两函数中较大者决定方程的解。

$case\ 1: n^{\log_b a}$较大，$\therefore T(n)=\theta(n^{\log_b a})$

比$f(n)$大一个多项式因子n^{ε}

$case\ 3: f(n)$较大，$\therefore T(n)=\theta(f(n))$

比$n^{\log_b a}$大一个多项式因子n^{ε}

$case\ 2:$二者相同，其解乘上一对数因子

$\therefore T(n)=\theta(n^{\log_b a}\cdot\lg n)=\theta(f(n)\lg n)$

58

§ 4.3 The master method(通用法，万能法)(续)

- Note: case1 & 3中，比较 f 和 $n^{\log_b a}$ 的大小均是相对多项式因子n^{ε} 而言
- *这三种情况并未覆盖所有可能的* f(n)，即case 1 & 2及case 2 & 3间有间隙。
- 例1： $T(n)=9T(n/3)+n$

  解：$a=9, b=3, f(n)=n$

  $n^{\log_b a}=n^{\log_3 9}=\theta(n^2)$

  $\therefore f(n)=O(n^{\log_3 9-\varepsilon})$,这里$\varepsilon=1$

  即$f(n)$比$n^{\log_3 9}$小一多项式因子n^1

  故$T(n)=\theta(n^2)$ //case1

59

§ 4.3 The master method(通用法，万能法)(续)

- 例2：

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

解：$a=1, b=3/2, f(n)=\theta(1)$

$n^{\log_{3/2}1}=n^0=1 \quad //case2$

$\therefore T(n)=\theta(\lg n)$

60

§ 4.3 The master method(通用法，万能法)(续)

- **例3：** $T(n) = 3T(n/4) + n\lg n$

解：$n^{\log_b a} = n^{\log_4 3} = O(n^{0.793})$

$f(n) = n\lg n$

$\because f(n) = \Omega(n^{\log_4 3+\varepsilon})$

即$f(n)$比$n^{\log_b a}$大一多项式因子$n^{0.2}$

对足够大的n：$af(n/b) = 3(n/4)\lg(n/4)$

$\leq \frac{3}{4} n\lg n = cf(n)$成立

$\therefore$满足case3，解为$T(n) = \theta(n\lg n)$

61

§ 4.3 The master method(通用法，万能法)(续)

- **例4：** $T(n) = 2T(n/2) + n\lg n$

解：$n^{\log_b a} = n < f(n) = n\lg n$

但是$f(n)$并不大于n一个多项式因子$n^{\varepsilon}(\varepsilon > 0)$

$\because$ 对给定$\varepsilon > 0$，对足够大的n，$n^{\varepsilon} > \lg n$，

$$\frac{n^{\varepsilon}}{\lg n} \to \infty$$

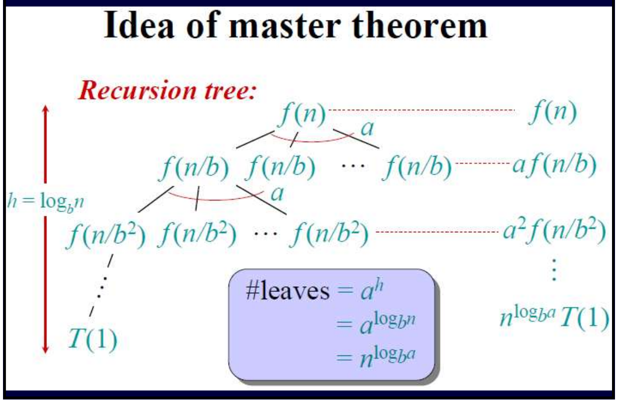
$\therefore$此解属于case2和case3之间，

不能用master定理

62

§ 4.3 The master method(通用法，万能法)(续)

- **Master Theorem**



递归式和分治法

- **分治法的适用条件**

分治法所能解决的问题一般具有以下几个特征：

(1)该问题的规模缩小到一定的程度就可以容易地解决；

- 因为问题的计算复杂性一般是随着问题规模的增加而增加，因此大部分问题满足这个特征

(2)该问题可以分解为若干个规模较小的相同问题，即该问题具有最优子结构性质；

- 这条特征是应用分治法的前提，它也是大多数问题可以满足的，此特征反映了递归思想的应用

递归式和分治法

- **分治法的适用条件(续)**

(3)利用该问题分解出的子问题的解可以合并为该问题的解；

- 能否利用分治法完全取决于问题是否具有这条特征，如果具备了前两条特征，而不具备第三条特征，则可以考虑贪心算法或动态规划。

(4)该问题所分解出的各个子问题是相互独立的，即子问题之间不包含公共的子问题。

- 这条特征涉及到分治法的效率，如果各子问题是不独立的，则分治法要做许多不必要的工作，重复地解公共的子问题，此时虽然也可用分治法，但一般用动态规划较好。

65

分治法的基本步骤

```
divide-and-conquer(P){
    if ( | P | <= n0) adhoc(P);  //解决小规模的问题
    divide P into smaller subinstances P1,P2,...,Pk;  //分解问题
    for (i=1,i<=k,i++)
      yi=divide-and-conquer(Pi); //递归的解各子问题
    return merge(y1, ..., yk); //将各子问题的解合并为原问题的解
}
```

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。

二分搜索技术

给定已按升序排好序的n个元素a[0:n-1]，现要在这n个元素中找出一特定元素x。

分析：

- ✓ 该问题的规模缩小到一定的程度就可以容易地解决；

分析：如果n=1即只有一个元素，则只要比较这个元素和x就可以确定x是否在表中。因此这个问题满足分治法的第一个适用条件

- ✓ 该问题可以分解为若干个规模较小的相同问题;
- ✓ 分解出的子问题的解可以合并为原问题的解；

分析：比较x和a的中间元素a[mid]，若x=a[mid]，则x在L中的位置就是mid；如果x<a[mid]，由于a是递增排序的，因此假如x在a中的话，x必然排在a[mid]的前面，所以我们只要在a[mid]的前面查找x即可；如果x>a[i]，同理我们只要在a[mid]的后面查找x即可。无论是在前面还是后面查找x，其方法都和在a中查找x一样，只不过是查找的规模缩小了。这就说明了此问题满足分治法的第二个和第三个适用条件。

二分搜索技术(续)

- ✓ 分解出的各个子问题是相互独立的。

分析：很显然此问题分解出的子问题相互独立，即在a[i]的前面或后面查找x是独立的子问题，因此满足分治法的第四个适用条件。

68

二分搜索技术

- 适用范围：顺序表、有序
- 基本思想(分治法)

(1)设R[low..high] 是当前查找区间，首先确定该区间的中点位置：mid= (low+high)/2 //整除

(2)将待查的K值与R[mid]比较，

① K＝R[mid].key：查找成功，返回位置mid

② K＜R[mid].key：则左子表R[low..mid-1]是新的查找区间

③ K＞R[mid],key：则右子表R[mid+1..high]是新的查找区间

初始的查找区间是R[1..n]，每次查找比较K和中间点元素，若查找成功则返回；否则当前查找区间缩小一半，直至当前查找区间为空时查找失败。

69

二分搜索技术

- 算法：

```
int BinSearch( SeqList R, KeyType K ) {
    int mid, low=1, high=n;
    while ( low < high ) { //当前查找区间R[low..high]非空
        mid= (low+high)/2;  //整除
        if ( R[mid].key==K )return mid; //成功返回位置mid
        if ( K＜R[mid].key ) //两个子问题求解其中的一个
            high=mid-1; //在左区间中查找
        else
            low=mid+1;  //在右区间中查找
    } // endwhile
    return 0; //当前查找区间为空时失败
}
```

70

二分搜索技术

算法复杂度分析：

每执行一次算法的while循环，待搜索数组的大小减少一半。因此，在最坏情况下，while循环被执行了O (lg n) 次。循环体内运算需要O(1) 时间，因此整个算法在最坏情况下的计算时间复杂性为O (lg n)

请同学思考如何用递归式的方法去分析binary search的时间上界

71

大整数乘法

- 考虑两个n位的大整数A和B相乘，例如:

A = 12345678901357986429   B = 87654321284820912836

小学的方法:

$$a_1\ a_2 \ldots\ a_n$$
$$b_1\ b_2 \ldots\ b_n$$
$$(d_{10})\ d_{11}d_{12} \ldots d_{1n}$$
$$(d_{20})\ d_{21}d_{22} \ldots d_{2n}$$
$$\ldots \ldots \ldots \ldots \ldots \ldots \ldots$$
$$(d_{n0})\ d_{n1}d_{n2} \ldots d_{nn}$$

时间复杂度：$O(n^2)$

72

大整数乘法——第一种划分和组合算法

一个简单的例子: $A * B$，其中 $A = 2135$， $B = 4014$

$A = (21 \cdot 10^2 + 35)$， $B = (40 \cdot 10^2 + 14)$

所以, $A * B = (21 \cdot 10^2 + 35) * (40 \cdot 10^2 + 14)$

$= 21 * 40 \cdot 10^4 + (21 * 14 + 35 * 40) \cdot 10^2 + 35 * 14$

■采用分治法解决该问题：将一个n位的大整数划分为两个n/2位的大整数，即令 $A = A_1A_2$， $B = B_1B_2$（其中A和B是两个n位的整数， A_1, A_2, B_1, B_2 是 $n/2$ 位的整数），那么

$A * B = A_1 * B_1 \cdot 10^n + (A_1 * B_2 + A_2 * B_1) \cdot 10^{n/2} + A_2 * B_2$

■时间复杂度： $T(n)=4T(n/2)+O(n)=O(n^2)$

■计算复杂度没有得到改进！如果要改进时间复杂度，就必须减少子问题数量！

73

大整数乘法——第二种划分和组合算法

■改进的思想是将子问题的数量从4降到3：

$(A_1 + A_2)(B_1 + B_2) = A_1B_1 + (A_1B_2 + A_2B_1) + A_2B_2$,

i.e.,

$(A_1B_2 + A_2B_1) = (A_1 + A_2)(B_1 + B_2) - A_1B_1 - A_2B_2$,

这样，我们就仅需3次n/2位的大整数乘法即可($(A_1 + A_2)(B_1 + B_2)$， A_1B_1 和 A_2B_2)。

■由此可得改进后的时间复杂度为:

$$T(n) = 3T(n/2)+O(n) = 3^{\log_2 n} = n^{\log_2 3} \approx n^{1.585}$$

➢如果将大整数分成更多段，用更复杂的方式把它们组合起来，将有可能得到更优的算法。

➢最终的，这个思想导致了快速傅利叶变换(Fast Fourier Transform)的产生。该方法也可以看作是一个复杂的分治算法。

74

Strassen矩阵乘法

◆传统方法：$O(n^3)$

A和B的乘积矩阵C中的元素C[i,j]定义为: $c[i][j] = \sum_{k=1}^{n} A[i][k]B[k][j]$

若依此定义来计算A和B的乘积矩阵C,则每计算C的一个元素C[i][j]，需要做n次乘法和n-1次加法。

因此，算出矩阵C的 n^2 个元素所需的计算时间为 $O(n^3)$。

Strassen矩阵乘法

■ 分治法:

使用与大整数乘法类似的技术，将矩阵A，B和C中每一矩阵都分块成4个大小相等的子矩阵。由此可将方程C=AB重写为：

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

由此可得：

$$C_{11} = A_{11}B_{11} + A_{12}B_{21}$$
$$C_{12} = A_{11}B_{12} + A_{12}B_{22}$$
$$C_{21} = A_{21}B_{11} + A_{22}B_{21}$$
$$C_{22} = A_{21}B_{12} + A_{22}B_{22}$$

复杂度分析: $T(n) = \begin{cases} O(1) & n = 2 \\ 8T(n/2) + O(n^2) & n > 2 \end{cases}$ $\quad T(n) = O(n^3)$

Strassen矩阵乘法

为了降低时间复杂度，必须减少乘法的次数。

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

子问题数量从8降到了7:

$$M_1 = A_{11}(B_{12} - B_{22})$$
$$M_2 = (A_{11} + A_{12})B_{22}$$
$$M_3 = (A_{21} + A_{22})B_{11}$$
$$M_4 = A_{22}(B_{21} - B_{11})$$
$$M_5 = (A_{11} + A_{22})(B_{11} + B_{22})$$
$$M_6 = (A_{12} - A_{22})(B_{21} + B_{22})$$
$$M_7 = (A_{11} - A_{21})(B_{11} + B_{12})$$

➡

$$C_{11} = M_5 + M_4 - M_2 + M_6$$
$$C_{12} = M_1 + M_2$$
$$C_{21} = M_3 + M_4$$
$$C_{22} = M_5 + M_1 - M_3 - M_7$$

复杂度分析: $T(n) = \begin{cases} O(1) & n = 2 \\ 7T(n/2) + O(n^2) & n > 2 \end{cases}$ $\quad T(n) = O(n^{\log 7}) = O(n^{2.81})$

Strassen's Matrix Multiplication

Strassen observed [1969] that the product of two matrices can be computed as follows:

$$\begin{pmatrix} C_{00} & C_{01} \\ C_{10} & C_{11} \end{pmatrix} = \begin{pmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{pmatrix} * \begin{pmatrix} B_{00} & B_{01} \\ B_{10} & B_{11} \end{pmatrix}$$

$$= \begin{pmatrix} M_1 + M_4 - M_5 + M_7 & M_3 + M_5 \\ M_2 + M_4 & M_1 + M_3 - M_2 + M_6 \end{pmatrix}$$

78

Formulas for Strassen's Algorithm

$M_1 = (A_{00} + A_{11}) * (B_{00} + B_{11})$

$M_2 = (A_{10} + A_{11}) * B_{00}$

$M_3 = A_{00} * (B_{01} - B_{11})$

$M_4 = A_{11} * (B_{10} - B_{00})$

$M_5 = (A_{00} + A_{01}) * B_{11}$

$M_6 = (A_{10} - A_{00}) * (B_{00} + B_{01})$

$M_7 = (A_{01} - A_{11}) * (B_{10} + B_{11})$

79

Strassen矩阵乘法

- 传统方法：$O(n^3)$
- 分治法: $O(n^{2.81})$
- 更快的方法??

- Hopcroft和Kerr已经证明(1971)，计算2个2×2矩阵的乘积，7次乘法是必要的。因此，要想进一步改进矩阵乘法的时间复杂性，就不能再基于计算2×2矩阵的7次乘法这样的方法了。或许应当研究3×3或5×5矩阵的更好算法。
- 在Strassen之后又有许多算法改进了矩阵乘法的计算时间复杂性。目前最好的计算时间上界是 $O(n^{2.376})$

Strassen矩阵乘法 discussion

- 采用Strassen算法需要创建大量动态二维数组，其中分配堆内存空间将占用大量计算时间，从而掩盖了Strassen算法的优势；
- 可以对Strassen算法做出改进，设定一个界限。当n<界限时，使用brute force method计算矩阵相乘，而不继续分治递归；
- 动手题：实际实现Strassen alg时，当矩阵规模小于threshold时，常常会切换到Brute Force实现，在自己计算机上确定最佳threshold

Closest-Pair Problem

最接近点对问题

Closest-Pair Problem

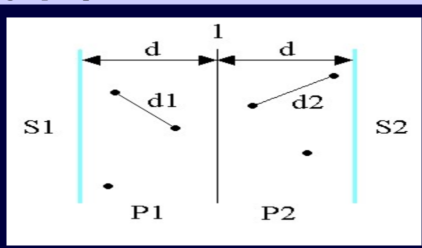
- Find the two closest points in a set of n points (for instance, in the two-dimensional Cartesian plane). 给定平面上n个点，找其中的一对点，使得在n个点所组成的所有点对中，该点对间的距离最小
- Brute-force algorithm
  - Compute the distance between every pair of distinct points and return the indexes of the points for which the distance is the smallest.
  - 将每一个点与其他n-1个点的距离算出，找出最小距离的点对即可。

```
ALGORITHM BruteForceClosestPoints(P)
    //Input: A list P of n (n ≥ 2) points P1 = (x1, y1), . . . , Pn = (xn, yn)
    //Output: Indices index1 and index2 of the closest pair of points
    dmin ← ∞
    for i ← 1 to n − 1 do
        for j ← i + 1 to n do
            d ← sqrt((xi − xj)^2 + (yi − yj)^2) //sqrt is the square root function
            if d < dmin
                dmin ← d; index1 ← i; index2 ← j
    return index1, index2
```

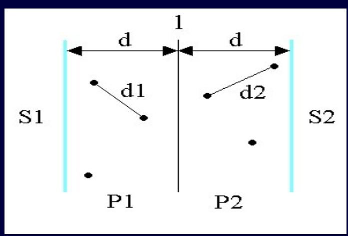
- Sketch：
  - 严格来讲，最接近点对可能多于一对，为简便起见，我们只找其中的一对作为问题的解。
  - 已经证明，该算法的计算时间下界是Ω(nlogn)。

分治法解决二维空间最接近点问题

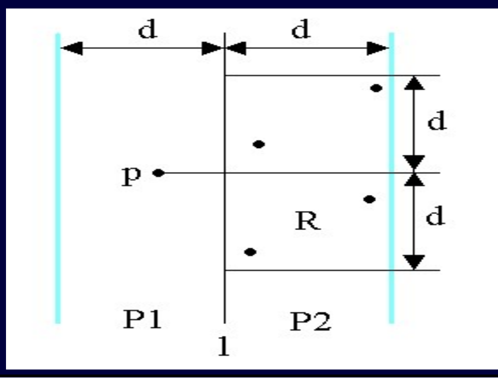
- 选取一垂直线l: x=m来作为分割直线。其中m为S中各点x坐标的中位数。由此将S分割为S_1和S_2;
- 递归地在S_1和S_2上找出其最小距离d_1和d_2，并设d=min{d_1, d_2}，S中的最接近点对或者是d，或者是某个{p, q}，其中p∈S_1且q∈S_2;



- 第一步筛选：如果最近点对由S_1中的p_3和S_2中的q_3组成，则p_3和q_3一定在划分线L的距离d内。

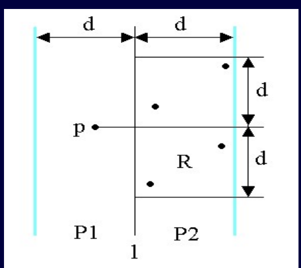


- 第二步筛选：考虑P1中任意一点p，它若与P2中的点q构成最接近点对的候选者，则必有distance(p，q)＜d。满足这个条件的P2中的点一定落在一个d×2d的矩形R中



R中的点具有稀疏性

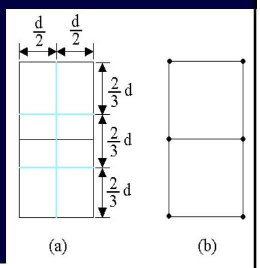
P_2中任何2个S中的点的距离都不小于d。由此可以推出矩形R中最多只有6个S中的点。



在分治法求解过程中的合并(Combine)步骤中最多只需要检查6 × n/2 = 3n O(n)个候选点对!

R中最多只有6个S中的点

- 证明:将矩形R的长为2d的边3等分，将它的长为d的边2等分，由此导出6个(d/2)×(2d/3)的矩形。
- 若矩形R中有多于6个S中的点，则由鸽舍原理易知至少有一个(d/2)×(2d/3)的小矩形中有2个以上S中的点。
- 设u，v是位于同一小矩形中的2个点，则令他们之间距离表示为：distance (u, v)，计算公式如下



$$(x(u)-x(v))^2+(y(u)-y(v))^2 \le (d/2)^2+(2d/3)^2=\frac{25}{36}d^2$$

如何确定要检查哪6个点

- P_2中与点p最接近这6个候选点的纵坐标与p的纵坐标相差不超过d.
- 因此，若将P_1和P_2中所有点按其y坐标排好序，则对P_1中所有点，对排好序的点列作一次扫描，就可以找出所有最接近点对的候选者。对P_1中每一点最多只要检查P_2中排好序的相继6个点。

```
double cpair2(S)
{
    n=|S|;
    if (n < 2) return ;
1、m=S中各点x坐标的中位数;
    构造S1和S2;                          ⟹ O(n)
    //S1={p∈S|x(p)<=m},
    S2={p∈S|x(p)>m}
2、d1=cpair2(S1);
    d2=cpair2(S2);                       ⟹ 2T(n/2)
3、dm=min(d1,d2);                        ⟹ 常数时间
```

4、设P_1是S_1中距垂直分割线ℓ的距离在d_m之内的所有点组成的集合；
P_2是S_2中距分割线ℓ的距离在d_m之内所有点组成的集合；
将P_1和P_2中点依其y坐标值排序；
并设X和Y是相应的已排好序的点列； O(n)

5、通过扫描X以及对于X中每个点检查Y中与其距离在d_m之内的所有点(最多6个)可以完成合并；
当X中的扫描指针逐次向上移动时，Y中的扫描指针可在宽为$2d_m$的区间内移动；
设d_l是按这种扫描方式找到的点对间的最小距离； O(n)

6、$d=\min(d_m,d_l)$;
return d; 常数时间

}

Time Complexity Analysis

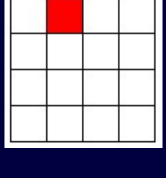
- ①、⑤用了O(n)时间；
- ②用了2T(n/2)时间
- ③、⑥用了常数时间
- ④在预排序的情况下用时O(n)

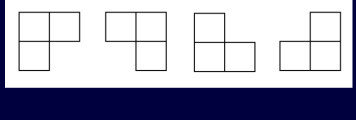
$$T(n)=\begin{cases} O(1) & n<4 \\ 2T(n/2)+O(n) & n\ge 4 \end{cases}$$

$$T(n)=O(n\log n)$$

棋盘覆盖

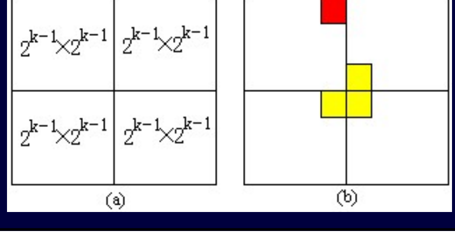
在一个$2^k\times 2^k$ 个方格组成的棋盘中，恰有一个方格与其它方格不同，称该方格为一特殊方格，且称该棋盘为一特殊棋盘。在棋盘覆盖问题中，要用图示的4种不同形态的L型骨牌覆盖给定的特殊棋盘上除特殊方格以外的所有方格，且任何2个L型骨牌不得重叠覆盖。





棋盘覆盖

当k>0时，将$2^k\times 2^k$棋盘分割为4个$2^{k-1}\times 2^{k-1}$ 子棋盘(a)所示。特殊方格必位于4个较小子棋盘之一中，其余3个子棋盘中无特殊方格。为了将这3个无特殊方格的子棋盘转化为特殊棋盘，可以用一个L型骨牌覆盖这3个较小棋盘的会合处，如 (b)所示，从而将原问题转化为4个较小规模的棋盘覆盖问题。递归地使用这种分割，直至棋盘简化为棋盘k=1。



棋盘覆盖

```
void chessBoard(int tr, int tc, int dr, int dc, int size)
 {
   if (size == 1) return;
   int t = tile++,  // L型骨牌号
    s = size/2;  // 分割棋盘
   // 覆盖左上角子棋盘
   if (dr < tr + s && dc < tc + s)
     // 特殊方格在此棋盘中
     chessBoard(tr, tc, dr, dc, s);
   else {// 此棋盘中无特殊方格
     // 用 t 号L型骨牌覆盖右下角
     board[tr + s - 1][tc + s - 1] = t;
     // 覆盖其余方格
     chessBoard(tr, tc, tr+s-1, tc+s-1, s);}
   // 覆盖右上角子棋盘
   if (dr < tr + s && dc >= tc + s)
     // 特殊方格在此棋盘中
     chessBoard(tr, tc+s, dr, dc, s);
   else {// 此棋盘中无特殊方格
     // 用 t 号L型骨牌覆盖左下角
     board[tr + s - 1][tc + s] = t;
     // 覆盖其余方格
     chessBoard(tr, tc+s, tr+s-1, tc+s, s);}
   // 覆盖左下角子棋盘
   if (dr >= tr + s && dc < tc + s)
     // 特殊方格在此棋盘中
     chessBoard(tr+s, tc, dr, dc, s);
   else {// 用 t 号L型骨牌覆盖右上角
     board[tr + s][tc + s - 1] = t;
     // 覆盖其余方格
     chessBoard(tr+s, tc, tr+s, tc+s-1, s);}
   // 覆盖右下角子棋盘
   if (dr >= tr + s && dc >= tc + s)
     // 特殊方格在此棋盘中
     chessBoard(tr+s, tc+s, dr, dc, s);
   else {// 用 t 号L型骨牌覆盖左上角
     board[tr + s][tc + s] = t;
     // 覆盖其余方格
     chessBoard(tr+s, tc+s, tr+s, tc+s, s);}
 }
```

复杂度分析: $T(k)=\begin{cases} O(1) & k=0 \\ 4T(k-1)+O(1) & k>0 \end{cases}$ $T(n)=O(4^k)$渐进意义下的最优算法

循环赛日程表

设计一个满足以下要求的比赛日程表:
(1) 每个选手必须与其他n-1个选手各赛一次;
(2) 每个选手一天只能赛一次;
(3) 循环赛一共进行n-1天。

按分治策略，将所有的选手分为两半，**n**个选手的比赛日程表就可以通过为**n/2**个选手设计的比赛日程表来决定。递归地用对选手进行分割，直到只剩下**2**个选手时，比赛日程表的制定就变得很简单。这时只要让这**2**个选手进行比赛就可以了。

1	2	3	4	5	6	7	8
2	1	4	3	6	5	8	7
3	4	1	2	7	8	5	6
4	3	2	1	8	7	6	5
5	6	7	8	1	2	3	4
6	5	8	7	2	1	4	3
7	8	5	6	3	4	1	2
8	7	6	5	4	3	2	1

Chapter 7. Quicksort

99

§7 快速排序

尽管最坏时间是$\theta(n^2)$,但期望时间为$\theta(n\lg n)$

基于比较排序的时间下界为:$\lg n!$□ $n\lg n-1.44n+O(\lg n)$

快速排序平均:$1.39n\lg n+O(n)$,系数较小故称“快排”

1. 算法描述

① 方法

$Divide: A[p..r] \Rightarrow A[p..q-1] \le A[q] \le A[q+1..r]$

$Conquer:$递归对$A[p..q-1]$，$A[q+1..r]$快排

终结条件，区间长度为1时空操作

$Combine:$空操作

100

§7 快速排序（续）

② 算法

$QuickSort\ (A,p,r)$ {
  $if\,(p<r)${
    $q \leftarrow Partition(A,p,r)$; //划分元A[q]已正确
    $QuickSort(A,p,q-1)$;
    $QuickSort(A,q+1,r)$;
  }
}

101

§7 快速排序（续）

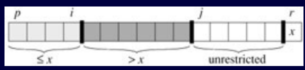
③划分

$Partition(A,p,r)${
  $x \leftarrow A[r]$; //区间最后元素为划分元
  $i \leftarrow p-1$;
  $for\ j \leftarrow p\ to\ r-1\ do$ { //始终有i ≤ j
    $if\,(A[j] \le x)${ //使$A[p..i] \le x$
      $i \leftarrow i+1$;
      $A[i] \leftrightarrow A[j]$;
    } //end if 若含A[j]>x时，j加1
  } //end for
  $A[i+1] \leftrightarrow A[r]$; //划分元位置为i+1
  $return\ i+1$;
}

102

§ 7 快速排序（续）

- 循环不变量
  - I. A[p..i] ≤ x
  - II. A[i+1..j-1] > x
  - III. A[j..r-1] 尚未确定
  - IV. A[r]=x

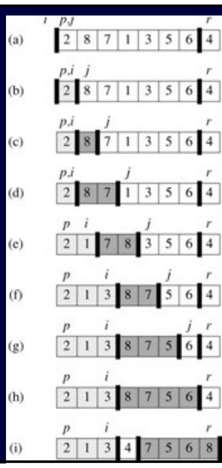


- Note：

这种划分使得：A[p..q-1] ≤ A[q] < A[q+1..r]

§ 7 快速排序（续）

- 循环不变量
  - I. A[p..i] ≤ x
  - II. A[i+1..j-1] > x
  - III. A[j..r-1] 尚未确定
  - IV. A[r]=x



§ 7 快速排序 (续)

2. 性能分析

- 划分是否平衡？

① 最坏划分：（已有序）

$$T(n)=T(n-1)+T(0)+\theta(n) \quad //\text{有一子区间为空}$$
$$=T(n-1)+\theta(n)$$
$$=\sum_{k=1}^{n}\theta(k)=\theta(\sum_{k=1}^{n}k)=\theta(n^2)$$

② 最好划分：（两子问题大小大致相等）

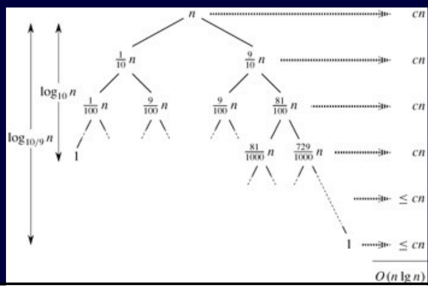
$$T(n)=2T(n/2)+\theta(n)$$
$$=\theta(n\lg n) \quad //\text{由master定理}$$

§ 7 快速排序 (续)

③ 平衡划分

快排平均时间接近于最好情况，设划分总是产生9:1划分

$$T(n)\le T(9n/10)+T(n/10)+cn$$



§ 7 快速排序 (续)

③ 平衡划分

∵任何底大于1的对数与以2为底的对数之间只相差一常数因子

∴任何常数比例（如99:1，999:1）的划分，树高仍为O(lgn)，从而快排时间为O(nlgn)

§ 7 快速排序 (续)

3. 随机版本 (P100)

- 快速排序的平均性能假定：输入的所有排列是等可能的
- 算法随机化是指：

  算法行为不仅由输入确定，而且与随机数发生器产生的值有关。强迫输入分布是随机的

$$RandomizedPartition\ (A,p,r)\ \{//\text{取代原partition}$$
$$i\leftarrow Random(p,r); \quad //\text{在}[p..r]\text{中选随机数i}$$
$$A[r]\leftrightarrow A[i];$$
$$return\ \ Partition\ (A,p,r); //\text{取A[r]作划分元}$$
$$\}$$

§7 快速排序 (续)

3. 随机版本
   - 但随机化算法分析较困难
   - 该算法非常有效，在排序过程中，某次随机选择最坏不会影响总体效果

- Ex 7.2-5
- 上机作业：写2个快排版本比较之

109