递归与分治

算法设计与分析

武汉大学
国家网络安全学院
李雨晴

分治策略的思想

- **划分**：把规模较大的问题(n)分解为若干(通常为2)个规模较小的子问题(<n)， 这些子问题相互独立且与原问题同类; (该子问题的规模减小到一定的程度就可以容易地解决)
- **治理**：依次求出这些子问题的解
- **组合**：把这些子问题的解组合起来得到原问题的解。

由于子问题与原问题是同类的，故分治法可以很自然地应用递归。

分治算法形式

- 如果实例I规律是小的，则直接求解，否则继续做下一步
- 把实例I分割成p个大小几乎相同的子实例$I_1, I_2 \ldots I_p$，对每个子实例I_j, $1 \le j \le p$，递归调用算法，并得到个p部分解
- 组合 p个部分解的结果得到原实例I的解，返回实例 I的解

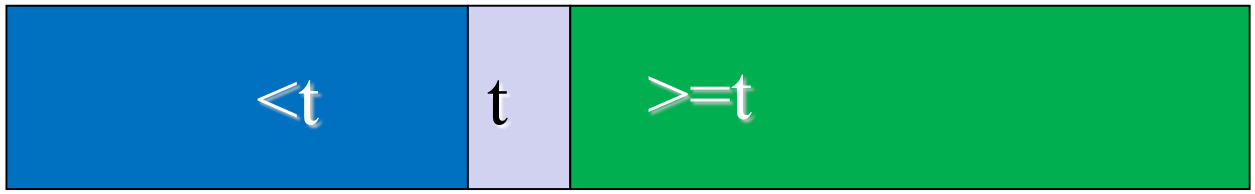
分治算法

- 引例：二分搜索
- 合并排序
- 寻找中项和第k小元素
- 快速排序

快速排序思想

1）寻找一个中心元素（通常为第一个数）

2）将小于中心点的元素移动至中心点之前，大于中心点的元素移动至中心点之后



3）对上步分成的两个无序数组段重复1）和 2）操作直到段长为1

快速排序

■ Quicksort

- 对原数组进行划分
- 对划分后的左、右子数组进行递归调用

■ Split

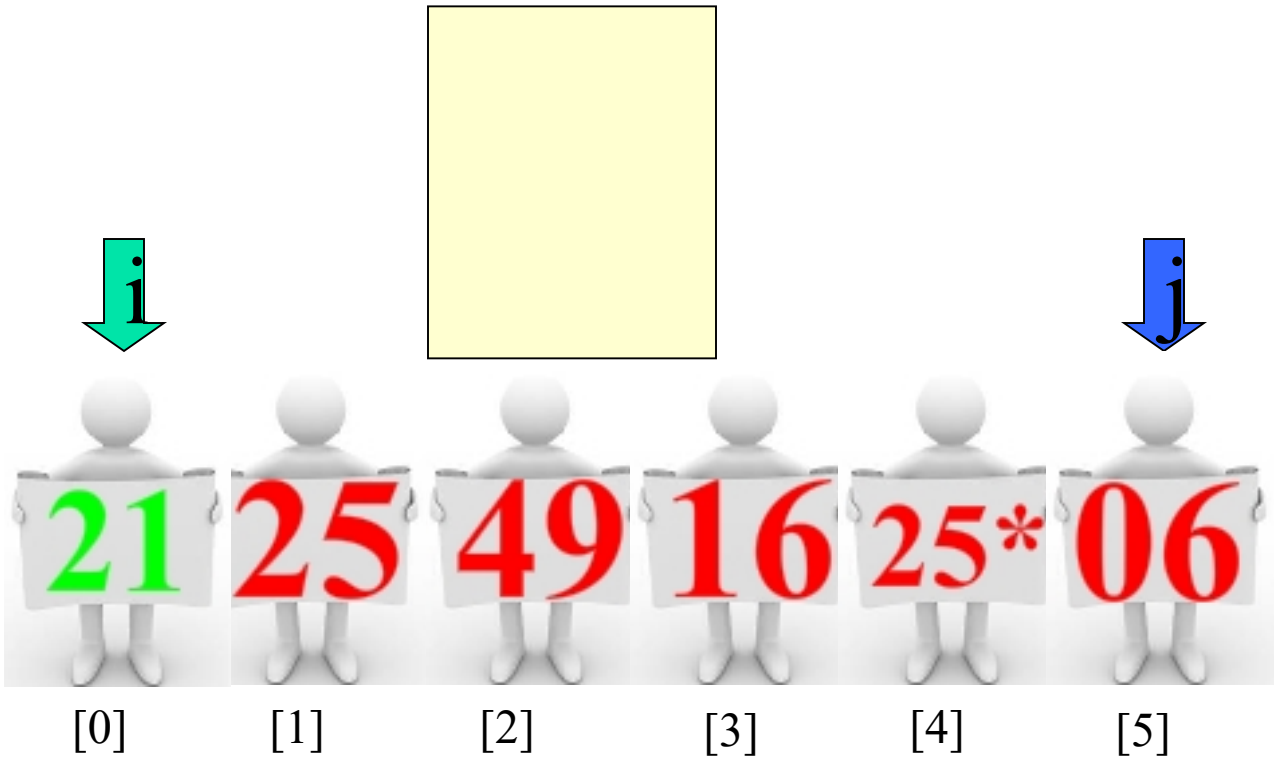
- 左向右找第一个大于等于中心点的数字
- 右向左找第一个小于等于中心点的数字
- 两个数字交换

左右重合

```
Algorithm: QUICKSORT(A[low…high])
输入: n个元素的数组A[low…high]
输出：按非降序排列的数组A[low…high]
1.  if low<high then
2.     w ← SPLIT(A[low…high])  {w为基准元素A[low]的新位置}
3.     quicksort(A, low, w-1)
4.     quicksort(A, w+1, high)
5.  end if
```

通过动画，可以看出每次中心元素都要交换。
根据划分的思想最后位置一定是中心元素

可以申请一个变量保存中心元素，以避免交换



i=0	j=5
i=1	j=5
i=1	j=4
i=1	j=3
i=2	j=3
i=2	j=2

算法终止

程序填空

left,right用于限定要排序数列的范围,temp即为中心元素

```
i=left;j=right;int temp=a[left];
do
{       //从右向左找第1个小于等于中心元素的位置j
        while( [   ] > [     ] && i<j)     j--;
        if(i<j)
        {       a[  ]  =  a[  ];
                i++;
        }
```

当前元素小于中心元素结束循环时，应当在中心元素的左边

移至左边

程序填空

```
        //从左向右找第1个大于等于中心元素的位置i
        while(a[i]<temp && i<j )      i++;
        if(i<j)
        {       a[j]=a[i];
                j--;
        }
    }while(i<j);
    [          ]; //将中心元素填入最终位置
    w=i;
```

排序方法对比

- 冒泡排序 $\Theta(n^2)$
- 选择排序$\Theta(n^2)$
- 插入排序$\Theta(n^2)$
- 合并排序$\Theta(n\log n)$
- 堆排序$\Theta(n\log n)$
- **快速排序?**

时间复杂度分析

理想情形：每次SPLIT后得到的左右子数组规模相当，因此有：

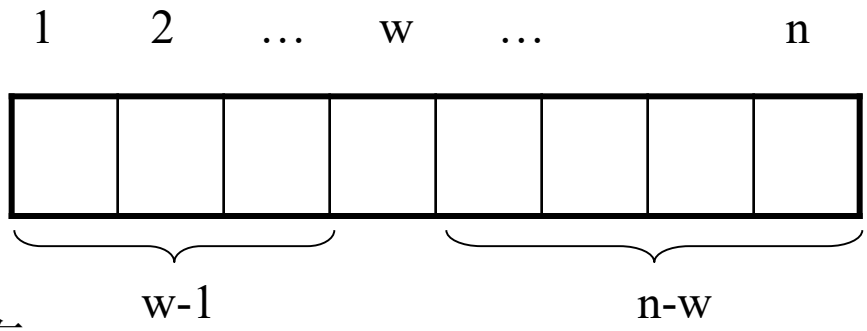
$$T(n)=\begin{cases}O(1) & if\ n=1\\ 2T(n/2)+\Theta(n) & if\ n>1\end{cases} \longrightarrow T(n)=\Theta(n\log n)$$

最差情形(已经排好序或是逆序的数组)：每次SPLIT后，只得到左或是右子数组，因此有：

$$T(n)=\begin{cases}O(1) & if\ n=1\\ T(n-1)+\Theta(n) & if\ n>1\end{cases} \longrightarrow T(n)=\Theta(n^2)$$

平均情形：

我们用 $C(n)$ 表示对一个n个元素的数组进行快速排序所需要的总的比较次数。



因此，我们有：

$$C(n)=(n-1)+\frac{1}{n}\sum_{w=1}^{n}\left(C(w-1)+C(n-w)\right)$$

$$\because \sum_{w=1}^{n}C(n-w)=C(n-1)+C(n-2)+\cdots+C(0)=\sum_{w=1}^{n}C(w-1)$$

$$\therefore C(n)=(n-1)+\frac{2}{n}\sum_{w=1}^{n}C(w-1)$$

$$n \cdot C(n) = n(n-1) + 2\sum_{w=1}^{n} C(w-1) \ldots\ldots (a)$$

↓ n-1替换n

$$(n-1)C(n-1) = (n-1)(n-2) + 2\sum_{w=1}^{n-1} C(w-1) \ldots\ldots (b)$$

(a)-(b),并适当变换 →

$$\frac{C(n)}{n+1} = \frac{C(n-1)}{n} + \frac{2(n-1)}{n(n+1)}$$

令 $D(n) = \frac{C(n)}{n+1}$ ↓

$$D(n) = D(n-1) + \frac{2(n-1)}{n(n+1)}, D(1) = 0$$

↓

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

$$= 4\sum_{j=2}^{n+1} \frac{1}{j} - 2\sum_{j=1}^{n} \frac{1}{j} = 2\sum_{j=1}^{n} \frac{1}{j} - \frac{4n}{n+1} = \Theta(\log n)$$

$$\therefore C(n) = (n+1)D(n) = \Theta(n \log n)$$

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：**
  - 该问题的规模缩小到一定的程度就可以容易地解决；
  - 该问题可以分解为若干个规模较小的同类问题；
  - 利用该问题分解出的子问题的解可以合并为该问题的解；
  - 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，但一般用动态规划更为合适。

使用分治策略的算法设计模式

```
divide_and_conquer(P)
{
    if(|P|<=n0)
        direct_process(P);  //解决小规模的问题
    else
    {
        divide P into smaller subinstances P1,P2,…,Pa ;  //分解问题
        for(int i=1;i<=a;i++)
            yi=divide_and_conquer(Pi);  //递归地解各子问题
        merge(y1,y2,…,ya);  //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

分治法的复杂度分析

- 从分治法的一般设计模式可以看出，用它设计出的算法通常可以是递归算法。因而，算法的时间复杂度通常可以用递归方程来分析。
- 假设算法将规模为n的问题分解为a(a>=1)个规模为n/b(b>1)的子问题解决。分解子问题以及合并子问题的解耗费的时间为s(n)，则算法的时间复杂度可以递归表示为：

$$T(n)=\begin{cases} c & , \ n <= n_0 \\ aT(n/b)+s(n) & , \ n > n_0 \end{cases}$$

- 回顾Master Theorem

Master Theorem

设$a \geq 1$，$b > 1$为常数。$s(n)$为一给定的函数，$T(n)$递归定义如下：

$$T(n) = a \cdot T(n/b) + s(n)$$

并且$T(n)$有适当的初始值。那么，当n充分大时，有：

(1) 若存在$\varepsilon > 0$，使得$s(n) = \Omega(n^{\log_b^a + \varepsilon})$ 成立，并且存在$c < 1$，使得$a \cdot s(n/b) \leq c \cdot s(n)$，那么有$T(n) = \Theta(s(n))$

(2) 若$s(n) = \Theta(n^{\log_b^a})$，那么$T(n) = \Theta(n^{\log_b^a} \cdot \log n)$

(3) 若存在$\varepsilon > 0$，使得$s(n) = O(n^{\log_b^a - \varepsilon})$ 成立，那么有$T(n) = \Theta(n^{\log_b^a})$

Master Theorem

- https://www.youtube.com/watch?v=2H0GKdrIowU

Theorem 5.1 *Let a be an integer greater than or equal to 1 and b be a real number greater than 1. Let c be a positive real number and d a nonnegative real number. Given a recurrence of the form*

$$T(n) = \begin{cases} aT(n/b) + n^c & \text{if } n > 1 \\ d & \text{if } n = 1 \end{cases}$$

then for n a power of b,

1. *if* $\log_b a < c$, $T(n) = \Theta(n^c)$,
2. *if* $\log_b a = c$, $T(n) = \Theta(n^c \log n)$,
3. *if* $\log_b a > c$, $T(n) = \Theta(n^{\log_b a})$.

2.8.3分治递推关系的解

- 分治算法中

$$f(n)=\begin{cases} d & \text{若 } n \leqslant n_0 \\ a_1 f(n/c_1)+a_2 f(n/c_2)+\cdots+a_p f(n/c_p)+g(n) & \text{若 } n > n_0 \end{cases}$$

- 展开递推式
- 代入法：猜想一个解，尝试用数学归纳法来证明

分治算法

- 大整数相乘
- 矩阵乘法
- 棋盘覆盖问题
- 最接近点对问题
- 排列问题
- 最大子数组问题

大整数相乘

- 通常，在分析算法的计算复杂度时，都将加法和乘法运算当作基本运算来处理，即将执行一次加法或乘法运算所需要的计算时间当作一个常数，该常数仅仅取决于计算机硬件处理速度。
- 然而，这个假定仅仅在参加运算的整数处于一定范围内时才是合理的。这个整数范围取决于计算机硬件对整数的表示。
- 在某些情况下，要处理很大的整数，它无法在计算机硬件能直接表示的整数范围内进行处理。这时候，就必须使用软件的方法来实现大整数的算术运算。

- 问题：设有两个n bit位的二进制整数X, Y，要计算XY.

```
    23                      10111
x   12      转换为2进制  x   01100
-------                 ---------
    46                      00000
+   23                     00000
-------                   10111
   276                    10111
                        +00000
                        ----------
                        1000101100
```

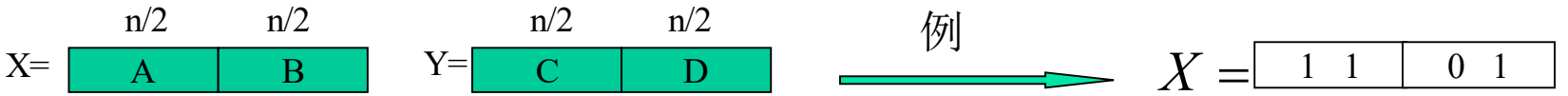
转换回10进制 → 276

时间复杂度分析

将位(bit)的乘法或加法当作基本运算，则逐位相乘算法的时间复杂度为：

$$T(n) = \Theta(n^2)$$

使用分治策略来求解之



$$A = (11)_2 = 3$$
$$B = (01)_2 = 1$$

我们有：

$$X = A \cdot 2^{n/2} + B \qquad Y = C \cdot 2^{n/2} + D$$

例 $\Rightarrow$ $X = 3 \cdot 2^{4/2} + 1 = 13$

下式成立：

$$XY = (A \cdot 2^{n/2} + B)(C \cdot 2^{n/2} + D)$$
$$= A \cdot C2^n + (A \cdot D + C \cdot B)2^{n/2} + B \cdot D$$

时间复杂性分析

$$XY=(A\cdot 2^{n/2}+B)(C\cdot 2^{n/2}+D)$$

$$=A\cdot C\cdot 2^{n}+(A\cdot D+C\cdot B)\cdot 2^{n/2}+B\cdot D$$

(1)4次$n/2$ bit位数的乘法($A\cdot C, A\cdot D, C\cdot B, B\cdot D$).//$4T(n/2)$

(2)$A\cdot C$左移n位($A\cdot C\cdot 2^{n}$). //$\Theta(n)$

(3)求$A\cdot D+C\cdot B$所作的n位加法. //$\Theta(n)$

(4)$A\cdot D+C\cdot B$左移$n/2$位($(A\cdot D+C\cdot B)2^{n/2}$). //$\Theta(n)$

(5)两个加法($A\cdot C\cdot 2^{n}+(A\cdot D+C\cdot B)\cdot 2^{n/2}+B\cdot D$). //$\Theta(n)$

$$T(n)=\begin{cases} a & if\ n=1 \\ 4T(n/2)+\Theta(n) & if\ n>1 \end{cases} \longrightarrow T(n)=\Theta(n^2)$$

如何降低时间复杂性？

观察：
$$A\cdot D+B\cdot C=(A+B)(C+D)-AC-BD$$
$$A\cdot D+B\cdot C=(A-B)(D-C)+AC+BD$$

所以：
$$XY=(A\cdot 2^{n/2}+B)(C\cdot 2^{n/2}+D)$$
$$=A\cdot C\cdot 2^n+(A\cdot D+C\cdot B)\cdot 2^{n/2}+B\cdot D$$
$$=A\cdot C\cdot 2^n+((A+B)(C+D)-AC-BD)\cdot 2^{n/2}+B\cdot D$$

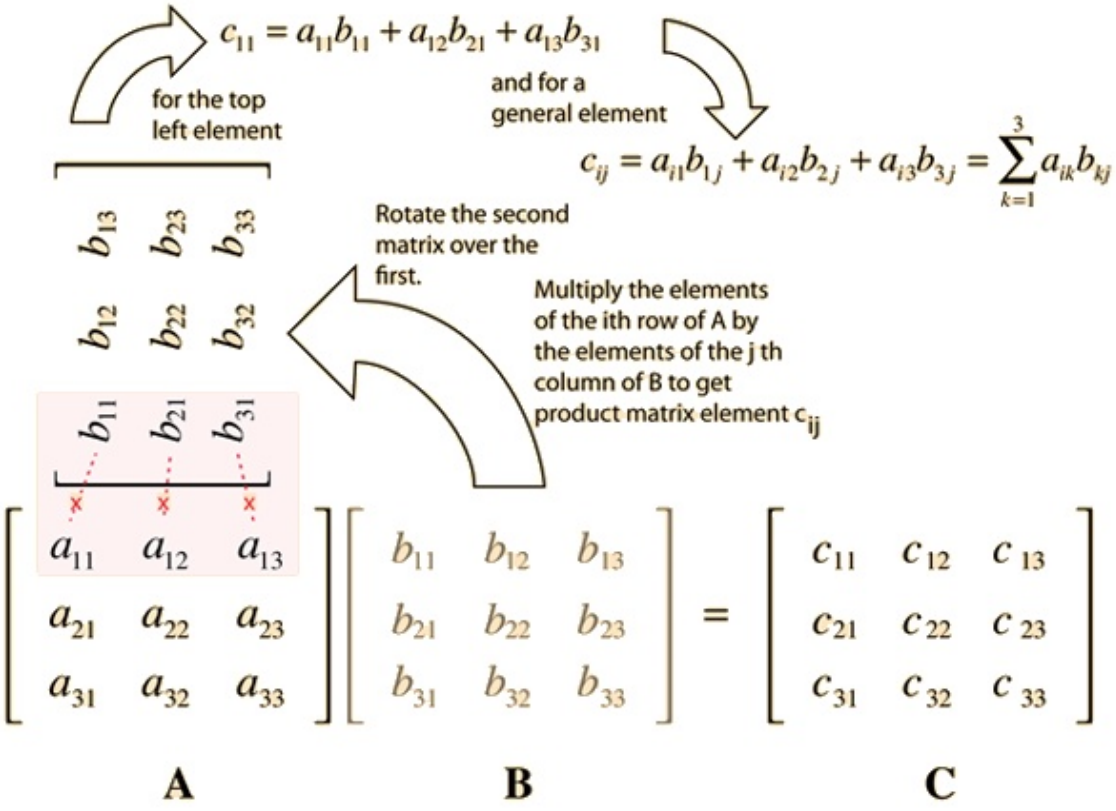
$$T(n)=\begin{cases} a & ,if\ \ n=1 \\ 3T(n/2)+\Theta(n) & ,if\ \ n>1 \end{cases} \longrightarrow T(n)=\Theta(n^{\log 3})=\Theta(n^{1.59})$$

细节问题：两个XY的复杂度都是O(n^log3)，但考虑到A+B, C+D可能得到n+1位的结果，使问题的规模变大，故不选择第1种方案

矩阵乘法

- 设A,B是两个n×n的矩阵，求C=AB.
- 方法1: 直接相乘法
- 方法2: 分块矩阵法(直接应用分治策略)
- 方法3: Strassen算法(改进的分治策略)

方法1：直接相乘


$c_{11} = a_{11}b_{11} + a_{12}b_{21} + a_{13}b_{31}$
for the top left element
and for a general element
$c_{ij} = a_{i1}b_{1j} + a_{i2}b_{2j} + a_{i3}b_{3j} = \sum_{k=1}^{3} a_{ik}b_{kj}$
Rotate the second matrix over the first.
Multiply the elements of the ith row of A by the elements of the j th column of B to get product matrix element c_{ij}
A
B
C

方法1：直接相乘

$$C=[c_{ij}]_{i=1,2,\cdots,n;\,j=1,2,\cdots,n} \qquad c_{ij}=\sum_{k=1}^{n} a_{ik}b_{kj}$$

时间复杂度分析：

假设每做一次标量乘法耗费时间为m,每做一次标量加法耗费时间为a,那么直接相乘算法的时间复杂度为：

$$T(n)=n^3m+n^2(n-1)a=\Theta(n^3)$$

矩阵分块

$$A=\begin{pmatrix}1&1&2&2\\1&2&3&3\\3&3&1&1\\2&1&2&2\end{pmatrix}\qquad B=\begin{pmatrix}2&1&1&2\\3&2&2&3\\1&3&1&1\\2&1&2&4\end{pmatrix}$$

$$A\square B=\begin{pmatrix}1&1&2&2\\1&2&3&3\\3&3&1&1\\2&1&2&2\end{pmatrix}\square\begin{pmatrix}2&1&1&2\\3&2&2&3\\1&3&1&1\\2&1&2&4\end{pmatrix}$$

$$A\square B=\left(\begin{array}{cc:cc}1&1&2&2\\1&2&3&3\\\hdashline3&3&1&1\\2&1&2&2\end{array}\right)\square\left(\begin{array}{cc:cc}2&1&1&2\\3&2&2&3\\\hdashline1&3&1&1\\2&1&2&4\end{array}\right)$$

方法2:分块矩阵法(直接应用分治策略)

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

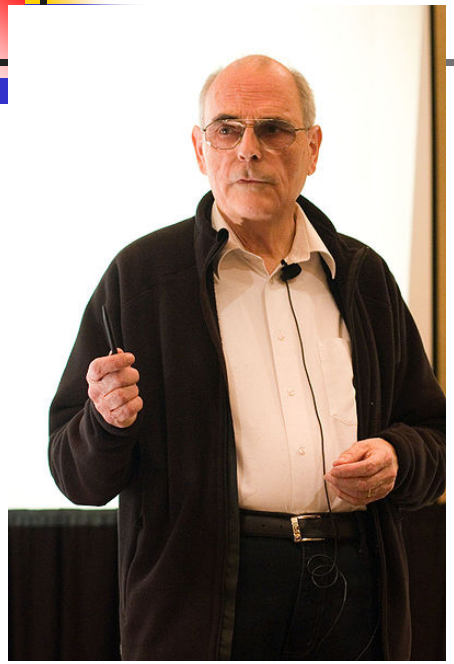
$$C_{11}=A_{11}B_{11}+A_{12}B_{21}$$
$$C_{12}=A_{11}B_{12}+A_{12}B_{22}$$
$$C_{21}=A_{21}B_{11}+A_{22}B_{21}$$
$$C_{22}=A_{21}B_{12}+A_{22}B_{22}$$

$$T(n)=\begin{cases} m & if \quad n=1 \\ 8T(n/2)+4(n/2)^2 a & if \quad n\geq 2 \end{cases}$$

$$\Downarrow$$

$$T(n)=\Theta(n^3)$$

Strassen算法



Volker Strassen giving the Knuth Prize lecture at SODA 2009.

Strassen was born on April 29, 1936, in Germany. In 1969, Strassen shifted his research efforts towards the analysis of algorithms with a paper on Gaussian elimination, introducing Strassen's algorithm, the first algorithm for performing matrix multiplication faster than the $O(n^3)$ time bound that would result from a naive algorithm. In the same paper he also presented an asymptotically-fast algorithm to perform matrix inversion, based on the fast matrix multiplication algorithm. This result was an important theoretical breakthrough, leading to much additional research on fast matrix multiplication, and despite later theoretical improvements it remains a practical method for multiplication of dense matrices of moderate to large sizes.

—From Wikipedia, the free encyclopedia

Strassen算法

引入下列 $M_i(i=1,2,\cdots,7)$:

$M_1=(A_{12}-A_{22})(B_{21}+B_{22})$

$M_2=(A_{11}+A_{22})(B_{11}+B_{12})$

$M_3=(A_{11}-A_{21})(B_{11}+B_{12})$

$M_4=(A_{11}+A_{12})B_{22}$

$M_5=A_{11}(B_{12}-B_{22})$

$M_6=A_{22}(B_{21}-B_{11})$

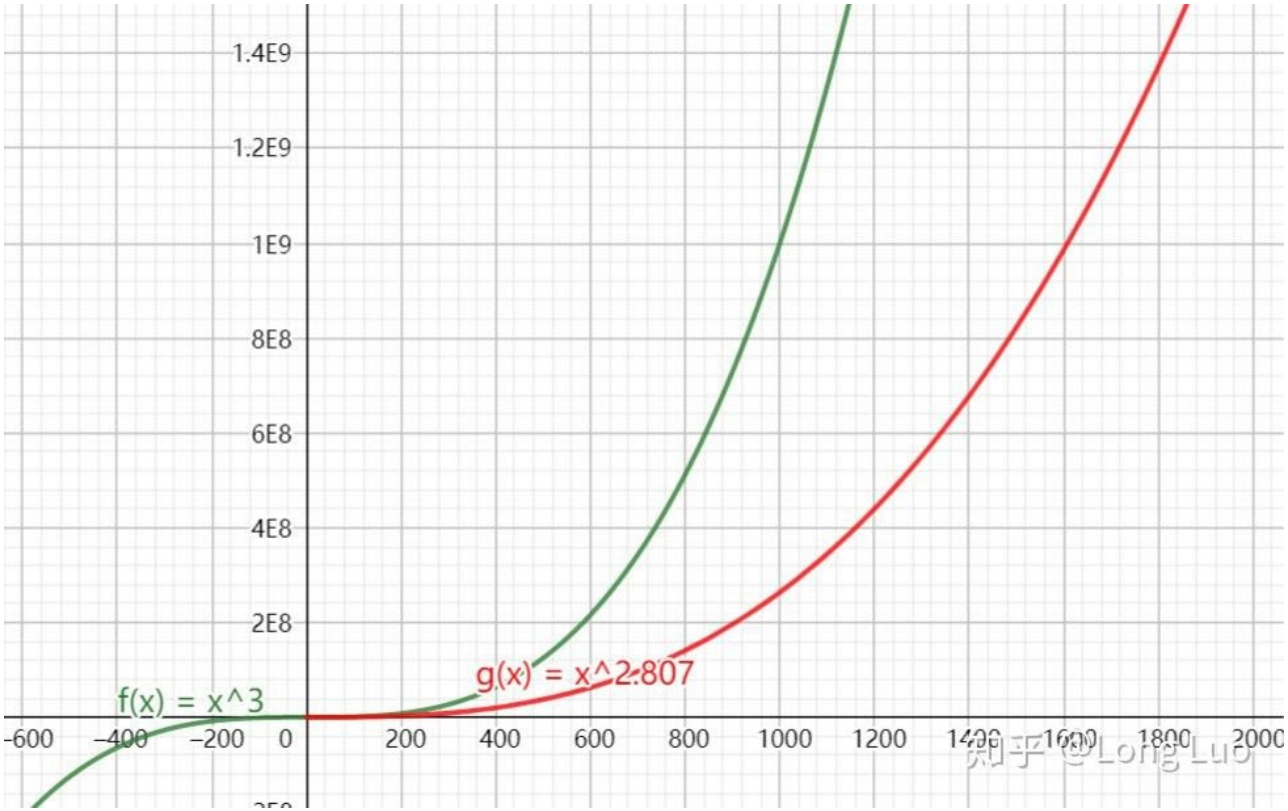
$M_7=(A_{21}+A_{22})B_{11}$

则有：$C_{11}=M_1+M_2-M_4+M_6$，$C_{12}=M_4+M_5$，

$C_{21}=M_6+M_7$，　　$C_{22}=M_2-M_3+M_5-M_7$

$$T(n)=\begin{cases} m & if \quad n=1 \\ 7T(n/2)+18(n/2)^2 a & if \quad n\geq 2 \end{cases}$$

$$T(n)=\Theta(n^{\log_b^a})=\Theta(n^{\log_2^7})=\Theta(n^{2.81})$$


1.4E9
1.2E9
1E9
8E8
6E8
4E8
2E8
f(x) = x^3
g(x) = x^2.807
-600
-400
-200
0
200
400
600
800
1000
1200
1400
1600
1800
2000
知乎 @Long Luo

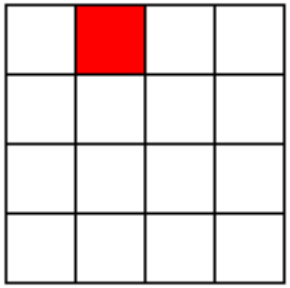
算法对比

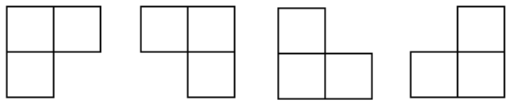
表 6.3 STRASSEN 算法和传统算法的比较

	n	乘法	加法
传统算法	100	1 000 000	990 000
STRASSEN 算法	100	411 822	2 470 334
传统算法	1000	1 000 000 000	999 000 000
STRASSEN 算法	1000	264 280 285	1 579 681 709
传统算法	10 000	10^{12}	9.99×10^{12}
STRASSEN 算法	10 000	0.169×10^{12}	10^{12}

棋盘覆盖问题

在一个$2^k \times 2^k$个方格组成的棋盘中，恰有一个方格与其他方格不同，称该方格为一**特殊方格**(红色表示)，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。



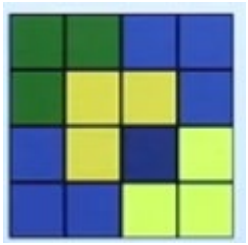


棋盘覆盖问题

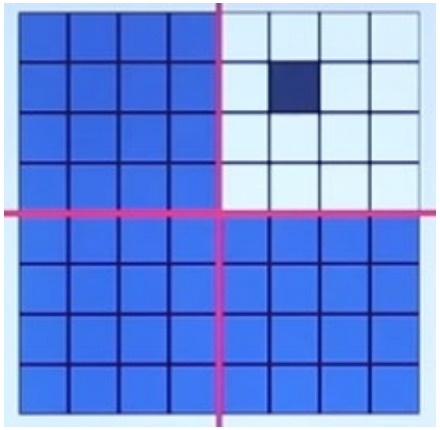
用 $(n^2 -1)/3$个L型骨牌放置在n×n的特殊棋盘上，正好能够覆盖所有的方格。

1×1

2×2



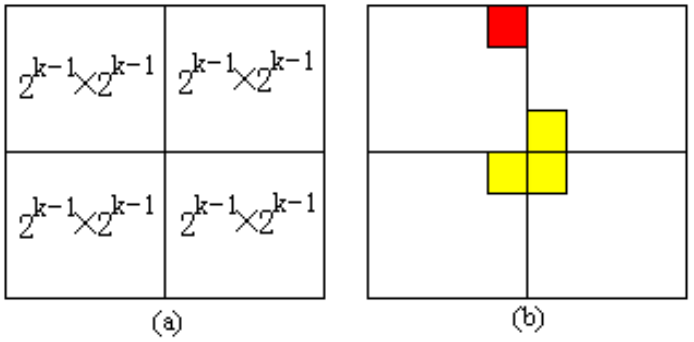
$2^2 \times 2^2$



$2^3 \times 2^3$

分析

当k>0时，可以将$2^k \times 2^k$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1×1。



2	2	3	3	7	7	8	8
2	1	1	3	7	6	6	8
4	1	5	5	9	9	6	10
4	4	5	0		9	10	10
12	12	13	0	0	17	18	18
12	11	13	13	17	17	16	18
14	11	11	15	19	16	16	20
14	14	15	15	19	19	20	20

一个实例

算法描述

```
public 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);
     }
  }
```

tr: 棋盘左上角方格的行号
tc: 棋盘左上角方格的列号
dr: 特殊方格所在的行号
dc: 特殊方格所在的列号
size: 棋盘大小

$$T(k) = 4T(k-1) + O(1)$$
$$= \Theta(4^k)$$

最接近点对问题

- 给定平面上的点集S，|S|=n。
- 若p∈S,q∈S,p!=q,则(p,q)称为一个点对。
- d(p,q)表示该点对(p,q)中点p和点q之间的欧几里得距离，
- 问d(p,q)的最小值是多少？

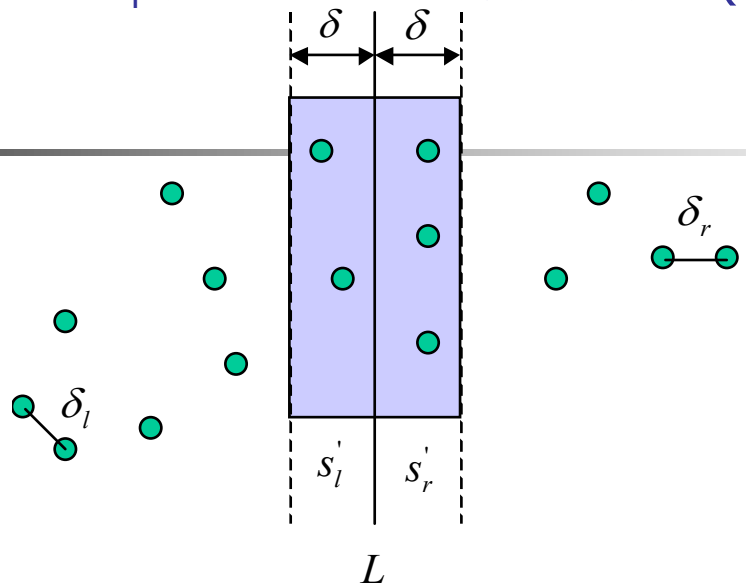
$$\delta = \min_{p \in S, q \in S, p \neq q} d(p,q)$$

- 严格说来，最接近点对可能多于1，为了简单起见，我们找到其中的1对作为问题的解。
- 最直观的方法：将每一点与其它n-1个点的距离算出来，然后找出其中最小的即可。 $T(n)=\Theta(n(n-1)/2)=\Theta(n^2)$

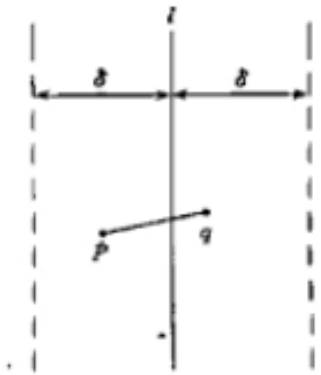
使用分治策略

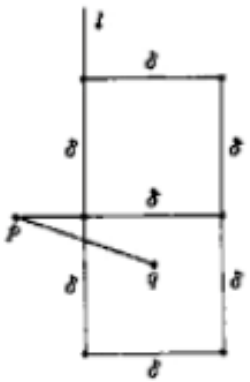
- 首先，将点集S中的点按照x坐标进行排序。
- 其次，使用一根垂直于x轴的直线L将S分割为两个子集S_l和S_r，使得$|S_l|=\lfloor |S|/2 \rfloor$，$|S_r|=\lceil |S|/2 \rceil$。$S_l$中的点都落在直线L的左边或是L上；$S_r$中的点都落在直线L的右边或是L上。
- 计算S_l中的最小距离δ_l，S_r中的最小距离δ_r(递归)； 计算S_l中的点与S_r中的点之间的最短距离δ'(难点所在)。
- 最接近点对的距离为$\min\{\delta_l, \delta_r, \delta'\}$。

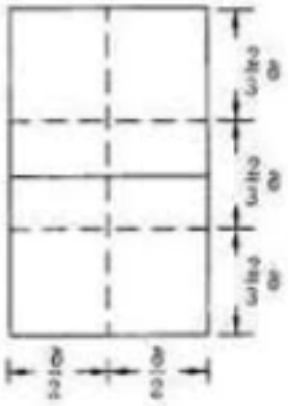
计算S_l中的点与S_r中的点之间的最短距离δ′(难点所在)



- 直接的方法：求解S_l中每个点与S_r中每个点之间距离，然后找出最小值。然而，这种方法需要的时间T(n)=Θ(n2)。
- 我们设δ＝min{δ_l,δ_r}，如果最接近点对是由 S_l中的某个点p和S_r中的某个点q所组成，那么，p,q至L的水平距离均不会超过δ。
- 用S_l'表示S_l中与L水平距离小于等于δ的点集，用S_r'表示S_r中与L水平距离小于等于δ的点集。也就是有$p \in S_l'$, $q \in S_r'$
- 然而，计算S_l'和S_r'中点对的最小值在最坏情形下时间复杂性仍旧为$T(n)=\Theta(n^2)$（当$S_l'= S_l$和$S_r' = S_r$时）。







$$sqrt(\frac{2\delta}{3} * \frac{2\delta}{3} + \frac{\delta}{2} * \frac{\delta}{2}) = \frac{5\delta}{6} < \delta。$$

- 用T表示两个垂直带之间的点的集合。
- 进一步分析，可以发现：如果最接近点对是由 S_l中的某个点p和S_r中的某个点q所组成，那么，p,q之间的垂直距离不会超过δ。
- 如果最接近点对是由 S_l中的某个点p和S_r中的某个点q所组成，p在带域左半部分，则q点必在如图所示的δ*2δ长方形上，而在该长方形上，最多只能由右边点集的6个点。证明如右图所示
- 所以T中的每个点至多只需要和T中的6个点计算距离(O(6n))，而不需要计算和T中所有的其它点之间的距离(O(n^2))。

- 哪六个点？
- S_l'和S_r'中所有的点按y排序好设数组M，则依次扫描数组M的每个元素x属于S_l'，并找到与之相邻的元素x′，x′属于S_r'且$|x.y-x'.y|<\delta$.
- 计算x和这些x′之间的距离

则分治算法复杂度的递归式为：

$$T(n) = 2T(n/2) + O(n \log n)$$

但还能不能进一步 降低复杂度？

在算法初始化时，对所有的点依据 y 进行排序，放入数组 Y，之后，在计算 δ_m 时，从数组 Y 中，按顺序提取那些 x 值位于 S_l' 和 S_r' 的点。因为提取操作的复杂度为 $\Theta(n)$，则分治算法复杂度的递归式为：

$$T(n) = 2T(n/2) + \Theta(n) \tag{5.1}$$

所以 $T(n) = O(n \log n)$。

Algorithm 15 ClosestPair(S)

1: 初始化：$X \leftarrow$ 对 S 中所有的点按照 x 进行排序；$Y \leftarrow$ 对 S 中所有的点按照 y 进行排序

2: 调用 ClosestPairDivide(S)

Algorithm 16 ClosestPairDivide(S)

1: 初始化：$\delta_m \leftarrow \infty$；
2: **if** $|S| \leq 3$ **then**
3: 直接计算 δ^*，return δ^*；
4: **else**
5: 取 X 中间点的 x 坐标值：$x_m \leftarrow x(X[\lfloor |S|/2 \rfloor])$；
6: $S_l \leftarrow X[1, \lfloor |S|/2 \rfloor]$，$S_r \leftarrow X[\lceil |S|/2 \rceil, n]$；
7: $\delta_l \leftarrow ClosestPairDivide(S_l)$；
8: $\delta_r \leftarrow ClosestPairDivide(S_r)$；
9: $\delta \leftarrow \min(\delta_l, \delta_r)$；
10: $S'_l \leftarrow S_l$ 中 x 值介于 $[x_m - \delta, x_m]$ 的点；
11: $S'_r \leftarrow S_r$ 中 x 值介于 $[x_m, x_m + \delta]$ 的点；
12: $S_p \leftarrow$ 从 Y 中按顺序提取 S'_r 和 S'_l 中所有的点；
13: **for** i=1 to $|S'_l|$ **do**
14: $p \leftarrow S'_l[i]$；
15: 从 S_p 中定位 p 点，并提取其前后各 8 个点，计算 p 和这些点的距离；
16: 得出最小的距离 δ_m；
17: **end for**
18: return $\min\{\delta_m, \delta\}$；
19: **end if**

排列问题

- 已知集合R={$r_1, r_2,..., r_n$}，请设计一个算法生成集合R中n个元素的全排列，并给出算法时间复杂性分析。

最大子数组问题

问题描述：给出一个数组，找出此数组的最大子数组，即子数组的和最大

暴力求解的复杂度为 $\Theta(n^2)$

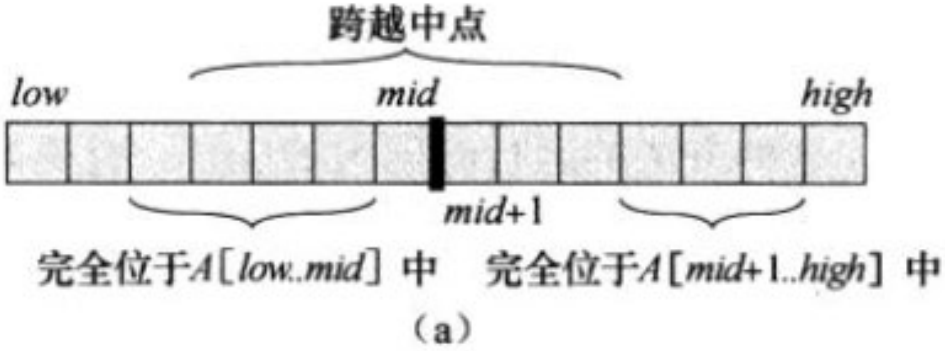
最大子数组问题

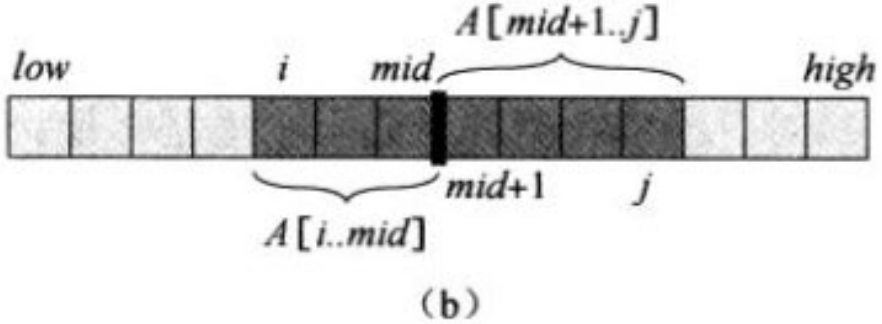
求解方法：将数组划分为两个规模相等的子数组

$A[low..high]$ ⇨ $A[low..mid]$和 $A[mid+1..high]$

最大子数组必然是以下三种情况之一

- 完全位于子数组 $A[low..mid]$中，因此 $low \leqslant i \leqslant j \leqslant mid$。
- 完全位于子数组 $A[mid+1..high]$中，因此 $mid < i \leqslant j \leqslant high$。
- 跨越了中点，因此 $low \leqslant i \leqslant mid < j \leqslant high$。





最大子数组问题

- 寻找跨越中点的最大子数组

我们可以很容易地在线性时间(相对于子数组 $A[low..high]$的规模)内求出跨越中点的最大子数组。此问题并非原问题规模更小的实例，因为它加入了限制——求出的子数组必须跨越中点。如图 4-4(b)所示，任何跨越中点的子数组都由两个子数组 $A[i..mid]$和 $A[mid+1..j]$组成，其中 $low \leqslant i \leqslant mid$ 且 $mid < j \leqslant high$。因此，我们只需找出形如 $A[i..mid]$和 $A[mid+1..j]$的最大子数组，然后将其合并即可。过程 FIND-MAX-CORSSING-SUBARRAY 接收数组 A 和下标

最大子数组问题

- 寻找跨越中点的最大子数组

```
FIND-MAX-CROSSING-SUBARRAY(A, low, mid, high)
 1  left-sum = −∞
 2  sum = 0
 3  for i = mid downto low
 4      sum=sum+A[i]
 5      if sum > left-sum
 6          left-sum = sum
 7          max-left = i
 8  right-sum = −∞
 9  sum = 0
10  for j = mid + 1 to high
11      sum=sum + A[j]
12      if sum > right-sum
13          right-sum = sum
14          max-right = j
15  return (max-left, max-right, left-sum + right-sum)
```

最大子数组问题

- 求解最大子数组问题的分治算法

```
FIND-MAXIMUM-SUBARRAY(A, low, high)
 1  if high == low
 2       return (low, high, A[low])                    // base case: only one element
 3  else mid=⌊(low+high)/2⌋
 4       (left-low, left-high, left-sum) =
              FIND-MAXIMUM-SUBARRAY(A, low, mid)
 5       (right-low, right-high, right-sum) =
              FIND-MAXIMUM-SUBARRAY(A, mid+1, high)
 6       (cross-low, cross-high, cross-sum) =
              FIND-MAX-CROSSING-SUBARRAY(A, low, mid, high)
 7       if left-sum ≥ right-sum and left-sum ≥ cross-sum
 8            return (left-low, left-high, left-sum)
 9       elseif rightr-sum ≥ left-sum and right-sum ≥ cross-sum
10            return (right-low, right-high, right-sum)
11       else return (cross-low, cross-high, cross-sum)
```

最大子数组问题

- 算法分析
  - FIND-MAX-CROSSING-SUBARRAY的复杂度为Θ(n)

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

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

作业

- P125-127
  - 6.6, 6.14
  - 6.19
  - 6.33