算法基础

主讲人：　庄连生

Email: { ***lszhuang@ustc.edu.cn*** **}**

Spring 2018，**USTC**

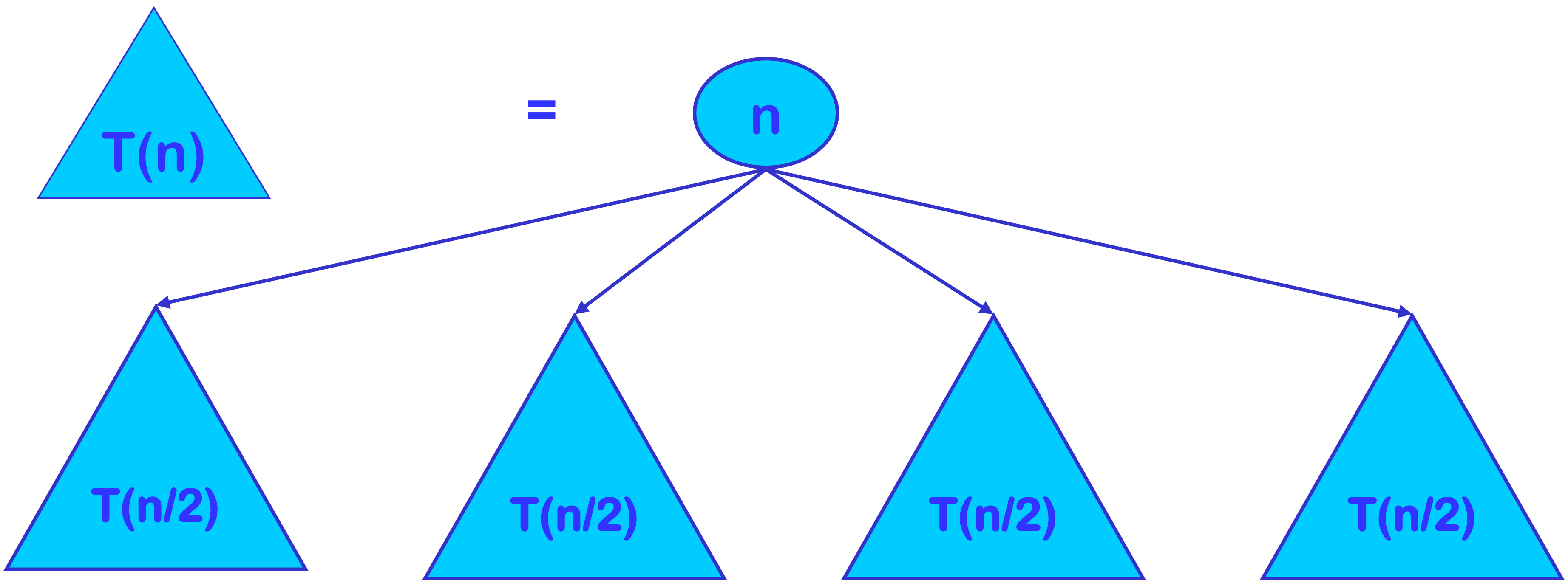
University of Science and Technology of China

第四讲 递归和分治策略

- 通过例子理解递归的概念；
- 掌握设计有效算法的分治策略；
- 通过几个范例学习分治策略设计技巧；
  - ✓ **Merge sort**
  - ✓ **Multiplication of two numbers**
  - ✓ **Multiplication of two matrices**
  - ✓ **Finding Minimum and Maximum**
  - ✓ **Majority problem (多数问题)**

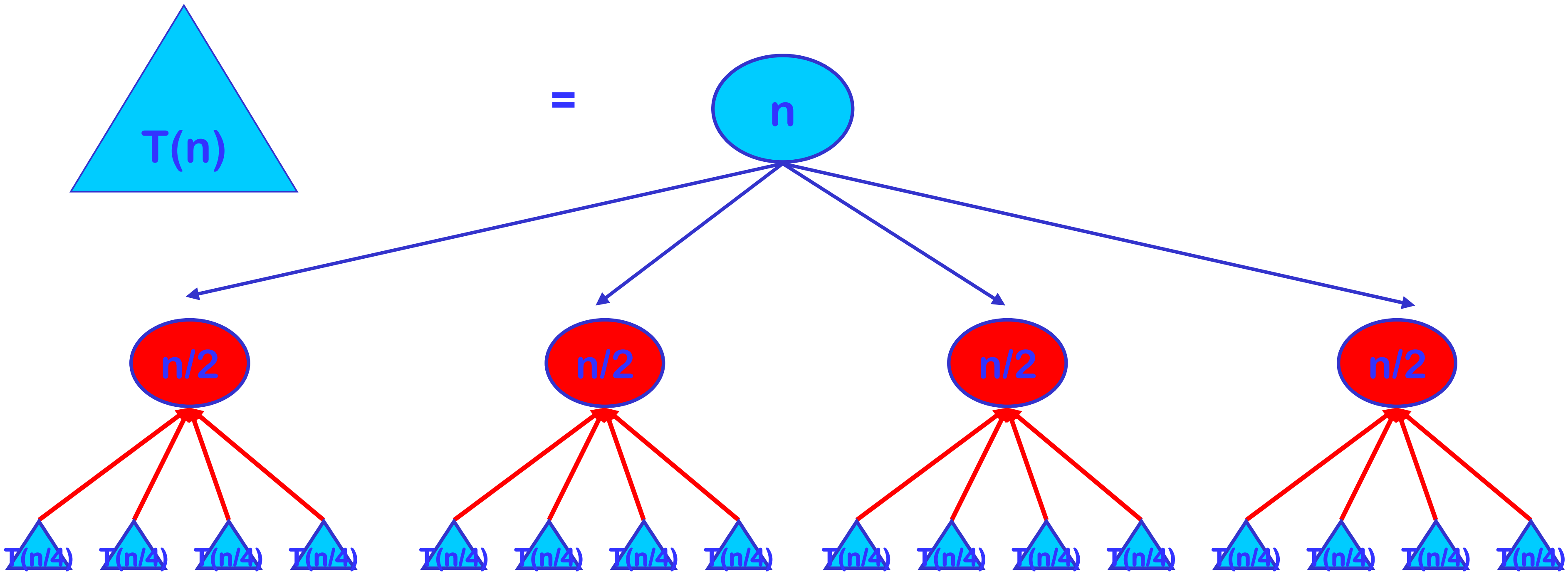
算法总体思想

- 对这k个子问题分别求解。如果子问题的规模仍然不够小，则再划分为k个子问题，如此递归的进行下去，直到问题规模足够小，很容易求出其解为止。



算法总体思想

- 将求出的小规模的问题的解合并为一个更大规模的问题的解，自底向上逐步求出原来问题的解。



算法总体思想

- 将求出的小规模的问题的解合并为一个更大规模的问题的解，自底向上逐步求出原来问题的解。

分治法的设计思想是，将一个难以直接解决的大问题，分割成一些规模较小的相同问题，以便各个击破，分而治之。

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。

下面来看几个实例。

递归的例子

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。

递归的例子

例2　排列问题

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)。$(r_i)$perm(X)表示在全排列perm(X)的每一个排列前加上前缀得到的排列。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(r_1)perm(R_1)，(r_2)perm(R_2)，…，(r_n)perm(R_n)构成。

例3　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$n_1 \geqslant n_2 \geqslant \ldots \geqslant n_k \geqslant 1$，$k \geqslant 1$。

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。

递归的例子

例3　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。

(3)　$q(n,n)=1+q(n,n-1)$;

正整数n的划分由$n_1=n$的划分和$n_1 \leqslant n-1$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$n_1 \leqslant m-1$ 的划分组成。

例3　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)。

递归小结

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。

递归小结

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。

递归小结

- **尾递归**就是从最后开始计算, 每递归一次就算出相应的结果, 也就是说, 函数调用出现在调用者函数的尾部, 因为是尾部, 所以根本没有必要去保存任何局部变量. 直接让被调用的函数返回时越过调用者, 返回到调用者的调用者去.

线性递归:

```
long Rescuvie(long n)
{
    return (n == 1) ? 1 : n * Rescuvie(n - 1);
}
```

尾递归:

```
long TailRescuvie(long n, long a)
{
    return(n == 1) ? a : TailRescuvie(n - 1, a * n);
}

long TailRescuvie(long n)
{//封装用的
    return(n == 0) ? 1 : TailRescuvie(n, 1);
}
```

递归小结

对于线性递归，他的递归过程如下：

Rescuvie(5)
{5 * Rescuvie(4)}
{5 * {4 * Rescuvie(3)}}
{5 * {4 * {3 * Rescuvie(2)}}}
{5 * {4 * {3 * {2 * Rescuvie(1)}}}}
{5 * {4 * {3 * {2 * 1}}}}
{5 * {4 * {3 * 2}}}
{5 * {4 * 6}}
{5 * 24}
120

对于尾递归，他的递归过程如下：

TailRescuvie(5)
TailRescuvie(5, 1)
TailRescuvie(4, 5)
TailRescuvie(3, 20)
TailRescuvie(2, 60)
TailRescuvie(1, 120)
120

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相同问题，即该问题具有**最优子结构性质**
- 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**较好。

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。

分治法的复杂性分析

一个分治法将规模为n的问题分成k个规模为n／m的子问题去解。设分解阀值n0=1，且adhoc解规模为1的问题耗费1个单位时间。再设将原问题分解为k个子问题以及用merge将k个子问题的解合并为原问题的解需用f(n)个单位时间。用T(n)表示该分治法解规模为|P|=n的问题所需的计算时间，则有：

$$T(n)=\begin{cases} O(1) & n=1 \\ kT(n/m)+f(n) & n>1 \end{cases}$$

通过迭代法求得方程的解：$T(n)=n^{\log_m k}+\sum_{j=0}^{\log_m n-1} k^j f(n/m^j)$

注意：递归方程及其解只给出n等于m的方幂时T(n)的值，但是如果认为T(n)足够平滑，那么由n等于m的方幂时T(n)的值可以估计T(n)的增长速度。通常假定T(n)是单调上升的，从而当$m^i \le n < m^{i+1}$时，$T(m^i) \le T(n) < T(m^{i+1})$。

分治法例子

Examples:

- **Merge sort**
- **Multiplication of two numbers**
- **Multiplication of two matrices**
- **Finding Minimum and Maximum**
- **Majority problem (多数问题)**
- **循环赛日程表**

分治法的例子

1. 整数相乘问题。

   X和Y是两个n位的十进制整数，分别表示为

   $X=x_{n-1}x_{n-2}\ldots x_0$, $Y=y_{n-1}y_{n-2}\ldots y_0$，其中$0\le x_i, y_j\le 9$ (i, j=0,1,…n-1)，设计一个算法求X×Y，并分析其计算复杂度。说明：算法中“基本操作”约定为两个个位整数相乘$x_i\times y_j$，以及两个整数相加。

2. 矩阵相乘问题。

   A和B是两个n阶实方阵，表示为 $A=\begin{pmatrix} a_{11}\ldots a_{1n} \\ \ldots\ldots\ldots \\ a_{n1}\ldots a_{nn}\end{pmatrix}$, $B=\begin{pmatrix} b_{11}\ldots b_{1n} \\ \ldots\ldots\ldots \\ b_{n1}\ldots b_{nn}\end{pmatrix}$

   设计一个算法求A×B，并分析计算复杂度。

   说明：算法中“基本操作”约定为两个实数相乘，或两个实数相加。

Exam1 Multiplication of two numbers（大整数的乘法）

two *n*-digit numbers X and Y, Complexity(X × Y) = ?

- **Naive (原始的) pencil-and-paper algorithm**

```
      12
   ×  23
  ------
       6
      3
  ------
      4
     2
  ------
     276
```

```
           31415962
         ×27182818
         ----------
          251327696
         31415962
       251327696
      62831924
    251327696
   31415962
 219911734
62831924
-------------------
853974377340916
```

- **Complexity analysis: n^2 multiplications and at most n^2-1 additions (加法). So, $T(n)=O(n^2)$.**

Exam1 Multiplication of two numbers（大整数的乘法）

two n-digit numbers X and Y, Complexity(X × Y) = ?

- **Divide and Conquer algorithm**

Let $X = a\ b$

$Y = c\ d$

where a, b, c and d are $n/2$ digit numbers, e.g. $1364 = 13 \times 10^2 + 64$.

Let $m = n/2$. Then

$$XY = (10^m a + b)(10^m c + d)$$
$$= 10^{2m} ac + 10^m (bc + ad) + bd$$

Exam1 Multiplication of two numbers（大整数的乘法）

two n-digit numbers X and Y, Complexity(X × Y) = ?

- **Divide and Conquer algorithm**

  Let $X = a\ b$, $Y = c\ d$

  then $XY = (10^m a+b)(10^m c+d) = 10^{2m}ac+10^m(bc+ad)+bd$

```
Multiply(X; Y; n):
if n = 1
   return X×Y
else
   m = ⌈n/2⌉
   a = ⌊X/10^m⌋; b=X mod 10^m
   c = ⌊Y/10^m⌋; d=Y mod 10^m
   e = Multiply(a; c; m)
   f = Multiply(b; d; m)
   g = Multiply(b; c; m)
   h = Multiply(a; d; m)
   return 10^(2m)e + 10^m(g + h) + f
```

Complexity analysis:

$T(1)=1$,

$T(n)=4T(\lceil n/2 \rceil)+O(n)$.

Applying Master Theorem, we have

$T(n)= O(n^2)$.

Exam1 Multiplication of two numbers（大整数的乘法）

two n-digit numbers X and Y, Complexity(X × Y) = ?

- **Divide and Conquer (Karatsuba's algorithm)**

  Let $X = a\ b$, $Y = c\ d$

  then $XY = (10^m a+b)(10^m c+d) = 10^{2m}ac+10^m(bc+ad)+bd$

Note that $bc + ad = ac + bd - (a - b)(c - d)$. **So, we have**

FastMultiply(X; Y; n):
if $n = 1$
return X×Y
else
$m = \lceil n/2 \rceil$
$a = \lfloor X/10^m \rfloor$; b=X mod 10^m
$c = \lfloor Y/10^m \rfloor$; d=Y mod 10^m
e = FastMultiply(a; c; m)
f = FastMultiply(b; d; m)
g = FastMultiply($a-b$; $c-d$; m)
return $10^{2m}e + 10^m(e + f - g) + f$

Complexity analysis:
$T(1)=1$,
$T(n)=3T(\lceil n/2 \rceil)+O(n)$.
Applying Master Theorem, we have

$$T(n) = O(n^{\log_2 3}) = O(n^{1.59})$$

Exam1 Multiplication of two numbers（大整数的乘法）

请设计一个有效的算法，可以进行两个n位大整数的乘法运算

◆小学的方法：$O(n^2)$ ✘效率太低

◆分治法: $O(n^{1.59})$ ✔较大的改进

◆更快的方法??

➢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。

Exam2 Multiplication of two matrices（矩阵相乘）

two $n \times n$ matrices A and B, Complexity(C=A×B) = ?

- **Standard method**

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$$

$$\begin{pmatrix} \cdots\cdots\cdots \; . \; .. \\ \cdots\cdots\cdots c_{ij} \; .. \\ \cdots\cdots\cdots \; . \; .. \\ \cdots\cdots\cdots \; . \; .. \end{pmatrix} = \begin{pmatrix} \cdots\cdots\cdots\cdots \\ {*}{*}{*}{*}{*} \\ \cdots\cdots\cdots\cdots \\ \cdots\cdots\cdots\cdots \end{pmatrix} \begin{pmatrix} \cdots\cdots\cdots {*} \; .. \\ \cdots\cdots\cdots {*} \; .. \\ \cdots\cdots\cdots {*} \; .. \\ \cdots\cdots\cdots {*} \; .. \end{pmatrix}$$

MATRIX-MULTIPLY(A, B)
for $i \leftarrow 1$ to n
  for $j \leftarrow 1$ to n
    $C[i, j] \leftarrow 0$
    for $k \leftarrow 1$ to n
      $C[i, j] \leftarrow C[i, j] + A[i, k] \cdot B[k, j]$
return C

Complexity:
$O(n^3)$ multiplications and additions.
$T(n) = O(n^3)$.

Exam2 Multiplication of two matrices（矩阵相乘）

two $n \times n$ matrices A and B, Complexity(C=A×B) = ?

- **Divide and conquer**

An $n \times n$ matrix can be divided into four $n/2 \times n/2$ matrices,

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

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

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

Complexity analysis:

Totally, 8 multiplications (subproblems), and 4 additions ($n/2 \times n/2 \times 4$).

$T(1)=1$, $T(n) = 8T(\lceil n/2 \rceil)+n^2$.

Applying Master Theorem, we have

$T(n)= O(n^3)$.

Exam2 Multiplication of two matrices（矩阵相乘）

two $n \times n$ matrices A and B, Complexity(C=A×B) = ?

- **Divide and conquer (Strassen Algorithm, 斯特拉森矩阵乘法)**

An $n \times n$ matrix can be divided into four $n/2 \times n/2$ matrices,

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

Define $P_1 = (A_{11}+A_{22})(B_{11}+B_{22})$

$P_2 = (A_{11}+A_{22})B_{11}$

$P_3 = A_{11}(B_{11}-B_{22})$

$P_4 = A_{22}(-B_{11}+B_{22})$

$P_5 = (A_{11}+A_{12})B_{22}$

$P_6 = (-A_{11}+A_{21})(B_{11}+B_{12})$

$P_7 = (A_{12}-A_{22})(B_{21}+B_{22})$

Then $C_{11}=P_1+P_4-P_5+P_7$, $C_{12}=P_3+P_5$

$C_{21}=P_2+P_4$, $C_{22}=P_1+P_3-P_2+P_6$

$$T(n) = O(n^{\log_2 7}) = O(n^{2.81})$$

Complexity analysis:
Totally, 7 multiplications, and 18 additions.
$T(1)=1$,
$T(n) = 7T(\lceil n/2 \rceil) + cn^2$.
Applying Master Theorem,

Exam2 Multiplication of two matrices（矩阵相乘）

- 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。

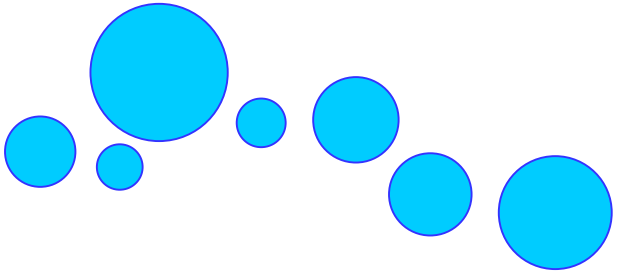
➢在Strassen之后又有许多算法改进了矩阵乘法的计算时间复杂性。目前最好的计算时间上界是 $\mathbf{O(n^{2.376})}$

➢是否能找到$O(n^2)$的算法？

Exam4 Finding Minimum and Maximum（MaxMin）

- **Background**

  Find the lightest and heaviest of *n* elements using a balance that allows you to compare the weight of 2 elements.（对于一个具有 *n* 个元素的数组，用一个天平，通过比较 2个元素的重量，求出最轻和最重的一个）



- **Goal**

  Minimize the number of comparisons.（正确找出需要的元素，最少的比较次数？）

Max element

Find element with max weight (重量) from w[0, n-1]

```
maxElement=0
for (int i = 1; i < n; i++)
    if (w[maxElement] < w[i]) maxElement = i;
```

Number of comparisons (比较次数) is $n-1$.

- **Obvious method（直接法）**
  - Find the max of n elements making $n-1$ comparisons.
  - Find the min of the remaining $n-1$ elements making $n-2$ comparisons.
  - Total number of comparisons is $2n-3$.

Exam4 Finding Minimum and Maximum（MaxMin）

- **Divide and conquer**

Example

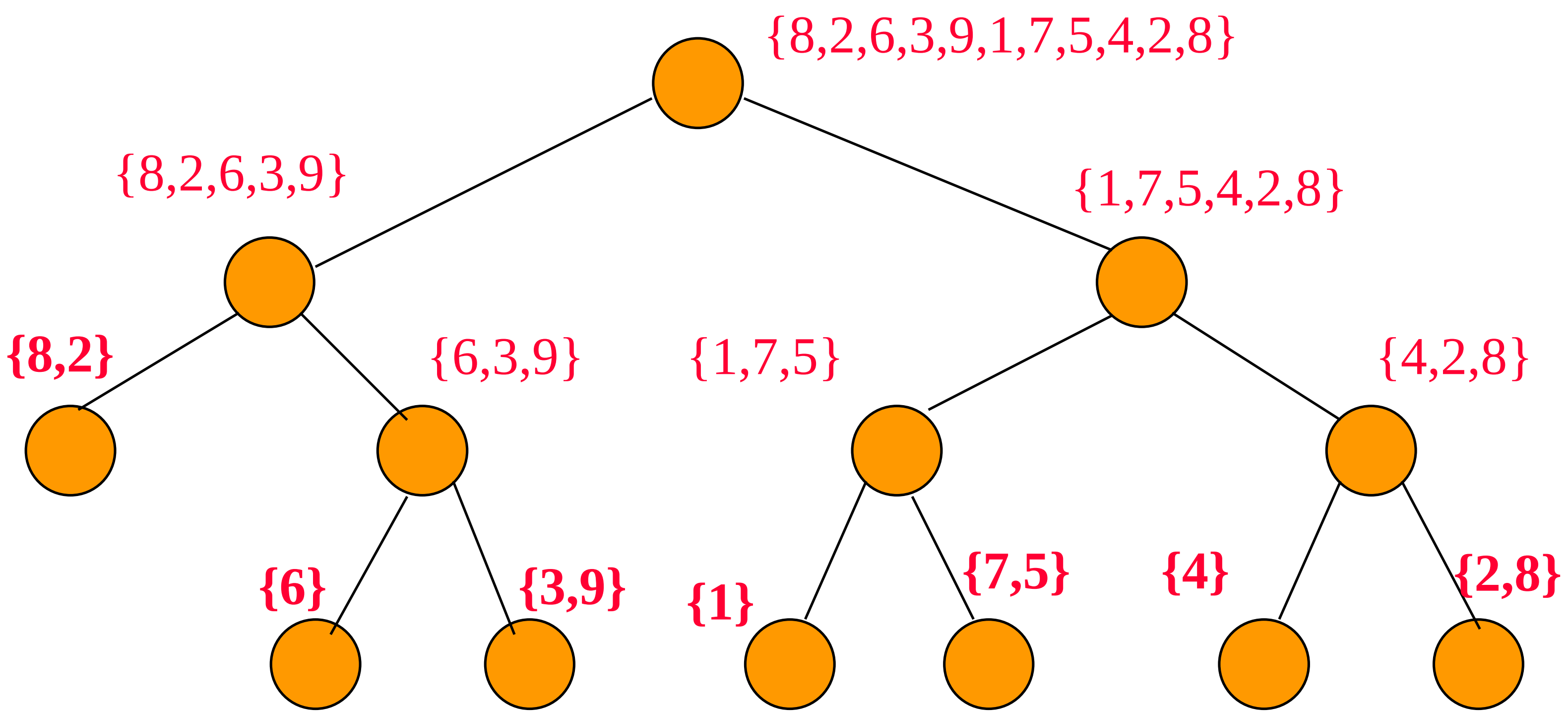
- **Find the min and max of {3,5,6,2,4,9,3,1}.**
  - **A = {3,5,6,2} and B = {4,9,3,1}.**
  - **min(A) = 2, min(B) = 1.**
  - **max(A) = 6, max(B) = 9.**
  - **min{min(A),min(B)} = 1.**
  - **max{max(A), max(B)} = 9.**
- 选苹果；挑运动员；……

Exam4 Finding Minimum and Maximum（MaxMin）

- **Divide and conquer**

  Example

  - **Dividing Into Smaller Problems**

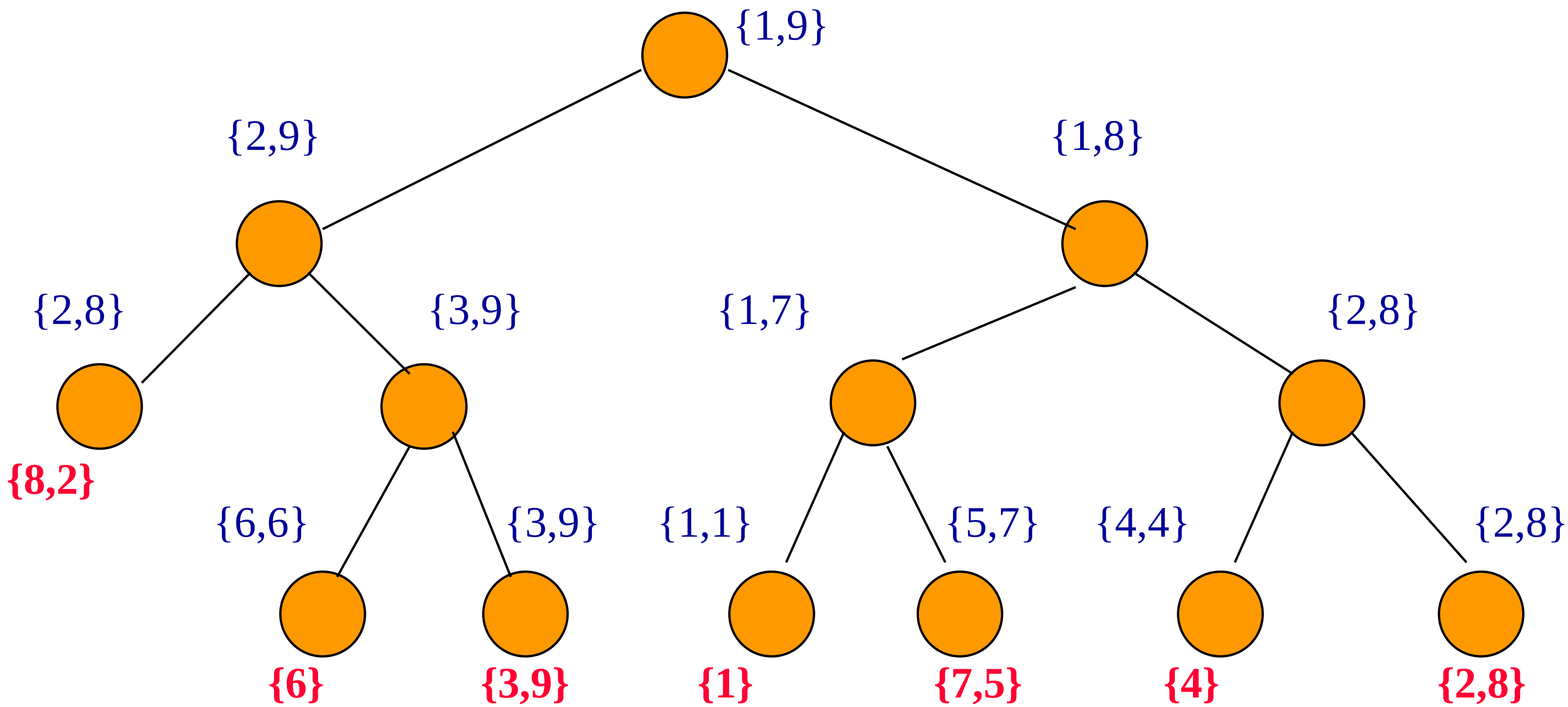


Exam4 Finding Minimum and Maximum（MaxMin）

- **Divide and conquer**

  Example

  - **Solve Small Problems and Combine**



Exam4 Finding Minimum and Maximum（MaxMin）

- **Divide and conquer**

MaxMin(L)
if length(L)=1 or 2, we use at most one comparison.
else
{ split (分裂) L into lists L1 and L2, each of n/2 elements
(min1, max1) = MaxMin(L1)
(min2, max2) = MaxMin(L2)
return (Min(min1, min2), Max(max1, max2))
}

Complexity analysis (Number of Comparisons):

$T(1)=0, T(2)=1,$

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

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

$= 2^{k-1}T(n/2^{k-1})+2^{k-1}+\ldots+2 = 2^{k-1}+2^{k-1}+\ldots+2$

$= 2^{k-1}+2^k-2 = 3n/2 - 2$ (there, assume $n=2^k$)

Exam4 Finding Minimum and Maximum（MaxMin）

- **Comparison between Obvious method (2*n*-3) and Divide-and-Conquer method (3*n*/2-2)**

Assume that one comparison takes one second.

Time	$2n-3$	$3n/2-2$
1 minute	n=31	n=41
1 hour	n=1801	n=2401
1 day	n=43201	n=57601

Exam5 Majority Problem（多数问题）

- **Problem**

  Given an array A of *n* elements, only use "=" test to find the *majority element* (which appears more than *n*/2 times) in A.

- **For example, given (2, 3, 2, 1, 3, 2, 2), then 2 is the majority element because 4>7/2.**

- **Trivial solution:**

  counting (计数) is $O(n^2)$.

```
Majority(A[1, n])
  for(i = 1 to n)
    M = 1
    for(j = 1 to n)
      if (i != j and A[i]==A[j])  M++
    end
    if (M>n/2) return "A[i] is the majortiy"
  end
  return "No majortity"
```

Exam5 Majority Problem（多数问题）

- **Divide and conquer**

Complexity analysis (Counting):
$T(n) = 2T(n/2)+O(n) = O(n\log n)$

Majority(A[1, *n*])
if *n*=1, then
return A[1]
else
m1=Majority(A[1, *n*/2])
m2=Majority(A[*n*/2+1, *n*])
test if m1 or m2 is the majority for A[1, *n*]
return majority or no majority.

A=(2, 1, 3, 2, 1, 5, 4, 2, 5, 2)
f[1] = 2
f[2] = 4
f[3] = 1
f[4] = 1
f[5] = 2

However, there is a linear time algorithm for the problem.

for(*i*=1 to *n*) ++frequency[A[*i*]]
M = Max(frequency[A[*i*]])
if (M > n/2)
check(M = = frequency[A[*j*]])
return “A[*j*] is the majority”

- **Moral (寓意) of the story?**

Exam5 Majority Problem（多数问题）

- **Divide and conquer**

Majority(A[1, *n*])
if *n*=1, then
return A[1]
else
m1=Majority(A[1, *n*/2])
m2=Majority(A[*n*/2+1, *n*])
test if m1 or m2 is the majority for A[1, *n*]
return majority or no majority.

Complexity analysis (Counting):

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

However, there is a linear time algorithm for the problem.

- **Moral (寓意) of the story: Divide and conquer may not always give you the best solution!**

Exam6 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