算法设计与分析

主讲教师：邵蓥侠
联系方式：shaoyx@bupt.edu.cn
个人主页：https://shaoyx.github.io/

第2章 递归与分治策略

学习要点:

- 理解递归的概念
- 理解分治法的策略
- 通过例子,学会设计算法，实现算法，评价算法：
  1. 二分搜索
  2. 大整数乘法
  3. 施特拉森 (Strassen)矩阵乘法
  4. 归并排序
  5. 快速排序
  6. 线性时间选择
  7. 最接近点对问题
  8. 循环赛日程表

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

用函数自身给出定义的函数称为**递归函数**。

边界条件

递归举例

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。

阶乘函数代码实现

```
int  Factorial ( int  n )
{
/* 1*/   if( n = 0) return 1;
/* 2*/   return  n * Factorial(n-1);
}
```

递归举例

例2　Fibonacci数列

无穷数列1，1，2，3，5，8，13，21，34，55，…，被称为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：

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

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Cooper变换、反演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。

分治法 *Divide and Conquer*

`Divide:`

Divide the problem (instance) into subproblems.

`Conquer:`

`Conquer the subproblems by solving them recursively.`

Smaller problems are solved *recursively* (except base cases).

`Combine:`

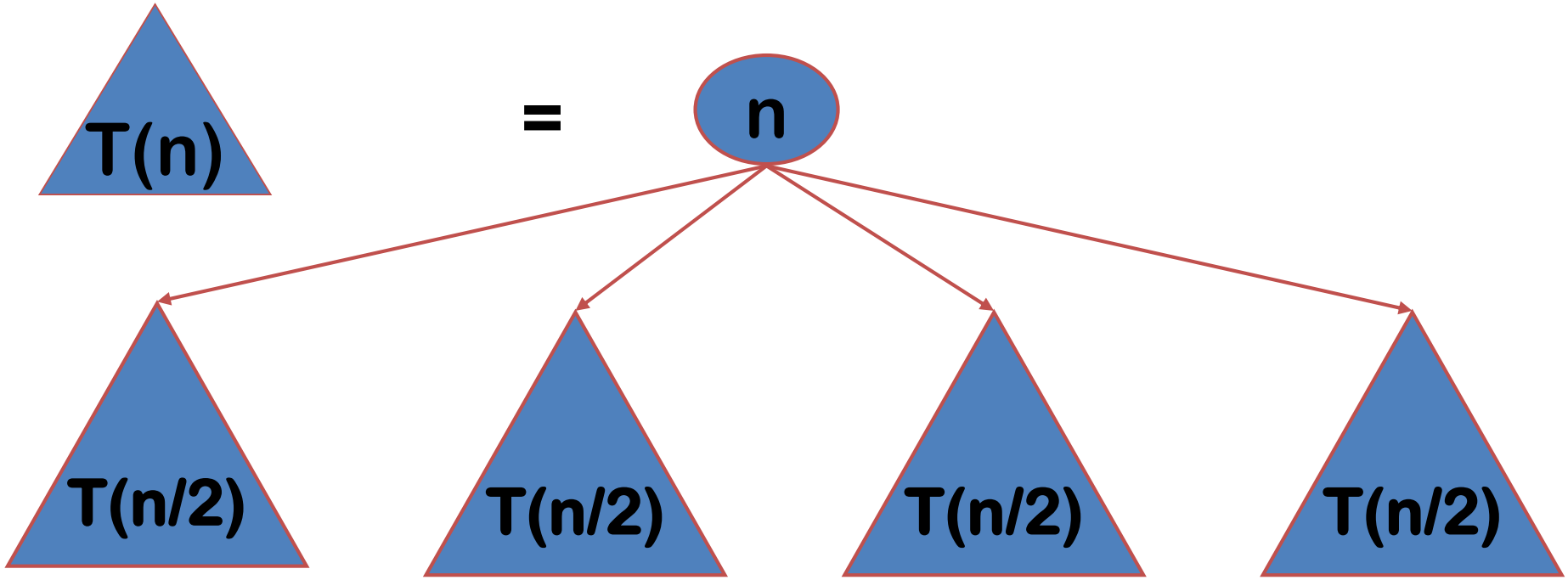
Combine subproblem solutions.

The solution to the original problem is then *formed from* the solutions to the subproblems.

- 学习分治法的策略，合理结合递归求解问题，提高算法效率。

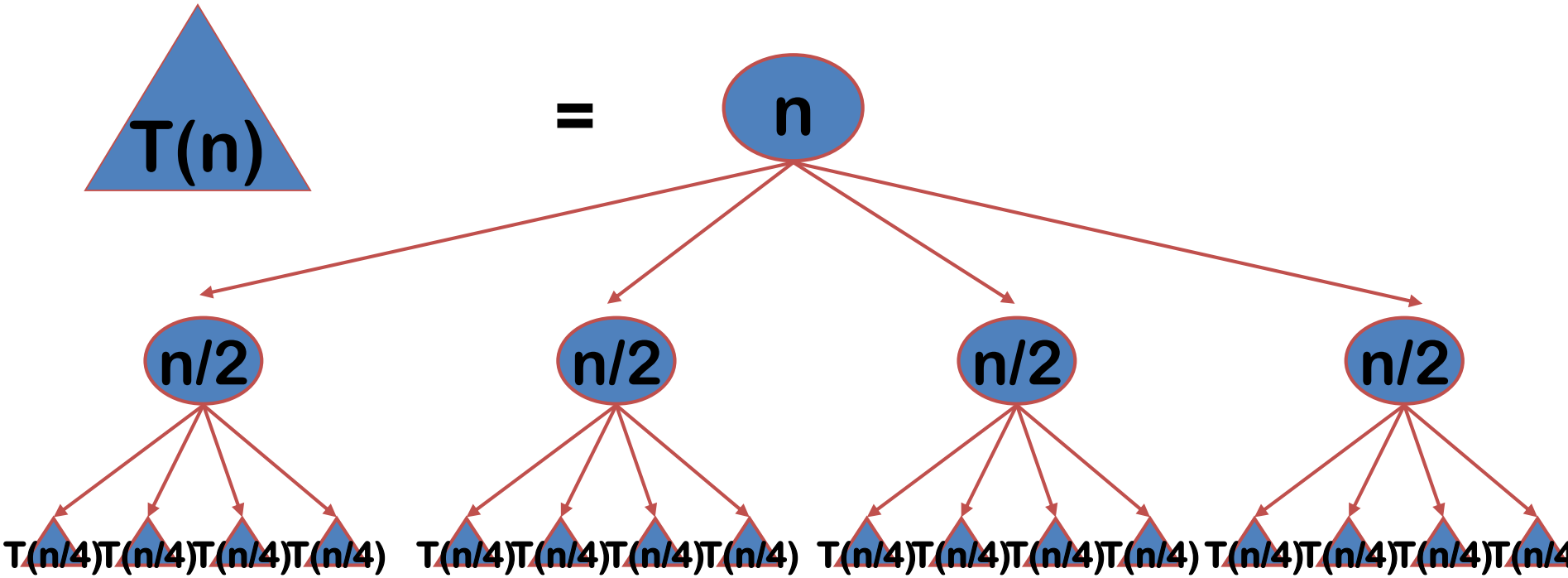
算法总体思想 - Divide, Conquer

- 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。



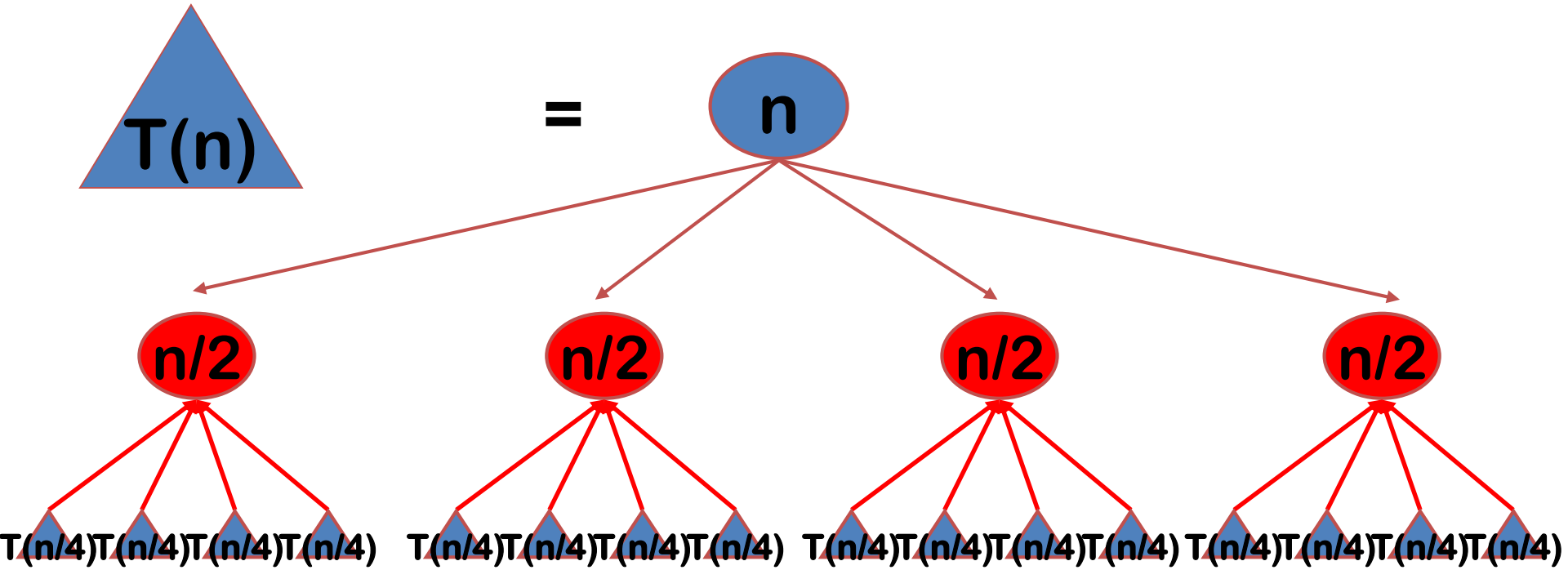
算法总体思想-Combine

- 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。



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。

凡治众如治寡，分数是也。

----孙子兵法

对程序设计的影响：

多核、多处理机上的多线程编程

——multithread programming

```
//分治法的基本步骤

divide-and-conquer(P)     //P为解决问题的规模
{
        if ( | P | <= n0) adhoc(P);    //解决小规模的问题，n0表示阈值，即不需要分解
        divide P into smaller subinstances P1,P2,...,Pk; //分解问题，k个
        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
}
```

- **分割原则：**

  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$n_0 = 1$，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。

通过例子来理解分治法

二分搜索
减小规模？ == 提高效率？

Binary Search

Find an element in a sorted array:

Example: Find 9

3	5	7	8	9	12	15

1. ***Divide:*** Check middle element.
2. ***Conquer:*** Recursively search 1 subarray.
3. ***Combine:*** Trivial.

Binary search

Find an element in a sorted array:

1. ***Divide:*** Check middle element.
2. ***Conquer:*** Recursively search 1 subarray.
3. ***Combine:*** Trivial.

Example: Find 9

3	5	7	8	9	12	15

Binary search

Find an element in a sorted array:

1. ***Divide:*** Check middle element.
2. ***Conquer:*** Recursively search 1 subarray.
3. ***Combine:*** Trivial.

Example: Find 9

Binary search

Find an element in a sorted array:

1. ***Divide:*** Check middle element.
2. ***Conquer:*** Recursively search 1 subarray.
3. ***Combine:*** Trivial.

Example: Find 9

3 5 7 8 9 12 15

Binary search

Find an element in a sorted array:

1. ***Divide:*** Check middle element.
2. ***Conquer:*** Recursively search 1 subarray.
3. ***Combine:*** Trivial.

Example: Find 9

Binary search

Find an element in a sorted array:

1. ***Divide:*** Check middle element.
2. ***Conquer:*** Recursively search 1 subarray.
3. ***Combine:*** Trivial.

Example: Find 9

3 5 7 8 9 12 15

Binary search

Find an element in a sorted array:

1. ***Divide:*** Check middle element.
2. ***Conquer:*** Recursively search 1 subarray.
3. ***Combine:*** Trivial.

Example: Find 9

Binary search 二分搜索技术

问题：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；
- ✓ 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。

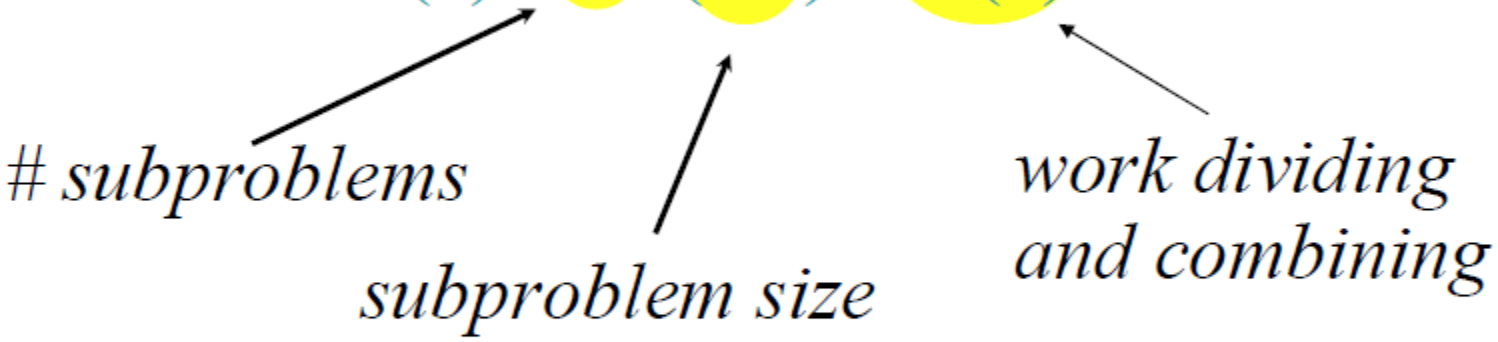
分析：很显然此问题分解出的子问题相互独立，即在a[i]的前面或后面查找x是独立的子问题，因此满足分治法的第四个适用条件。

```java
//分治法：二分搜索
public static int binarySearch(int [] a, int x, int n)
{
    // 在 a[0] <= a[1] <= ... <= a[n-1] 中搜索 x
    // 找到x时返回其在数组中的位置，否则返回-1
    int left = 0; int right = n - 1;
    while (left <= right)
    {
        int middle = (left + right)/2;
        if (x == a[middle]) return middle;
        if (x > a[middle]) left = middle + 1; //在右半部查找
        else right = middle - 1;              //在左半部查找
    }
    return -1; // 未找到x
}
```

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(logn) 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(logn) 。

Recurrence for binary search

$$T(n) = 1\,T(n/2) + \Theta(1)$$



$$n^{\log_b a} = n^{\log_2 1} = n^0 = 1 \Rightarrow \text{CASE 2 } (k = 0)$$
$$\Rightarrow T(n) = \Theta(\lg n) .$$

时间复杂性

- O(n)>O(logn)
- 例如：$n=10^{10}$

思考题：求数组的最大值和最小值。

思考：利用分治真的比直接求最大值、最小值的效率高吗?

再看分治算法

- 一般，分治算法常与递归方式结合；
- 一般，将问题分成两个子问题更合理，对于递归来说，分得太细对于效率和复杂性并无好处。

Powering a number

Problem: Compute a^n, where $n \in \mathbb{N}$.

Naive algorithm: $\Theta(n)$.

Divide-and-conquer algorithm:

$$a^n = \begin{cases} a^{n/2} \cdot a^{n/2} & \text{if } n \text{ is even;} \\ a^{(n-1)/2} \cdot a^{(n-1)/2} \cdot a & \text{if } n \text{ is odd.} \end{cases}$$

$$T(n) = T(n/2) + \Theta(1) \Rightarrow T(n) = \Theta(\lg n) .$$

〖Example〗 **Exponentiation**: computing X^N.

```
long int Pow ( long int  X,  unsigned int  N )
{
        long int  P = 1;
        while ( N -- )    P *= X;
        return  P;
}
```

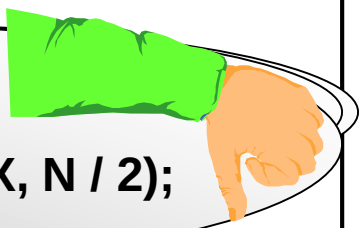
$T(N) = O(N)$

$$X^N = \begin{cases} (X^2)^{N/2} & \text{if } N \text{ is even} \\ \text{or} & \\ (X^2)^{(N-1)/2} \times X & \text{if } N \text{ is odd} \end{cases}$$

```
long int Pow ( long int  X,  unsigned int  N )
{
/* 1*/     if ( N == 0 )
/* 2*/         return 1;

/* 5*/     if ( IsEven( N ) )
/* 6*/         return Pow( X * X, N / 2 );
           else
/* 7*/         return Pow( X * X, N / 2 ) * X;
}
```

return Pow(X, N / 2) * Pow(X, N / 2);



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

通过例子来理解分治法

大整数乘法
提高效率？ 乘法减少？

大整数的乘法

- 问题：X和Y都是n位的二进制整数，要计算他们的乘积XY。
- 分治法：将n位的二进制整数X和Y都分成2段，每段的长为n/2位。

- 所以，$X = A2^{n/2} + B$, $Y = C2^{n/2} + D$

$$XY = AC\ 2^{n} + (AD+BC)\ 2^{n/2} + BD$$

算法复杂性分析

- $XY = AC\ 2^n + (AD+BC)\ 2^{n/2} + BD$
  - 乘法次数：4次n/2位的整数乘法；
  - 加法次数：3次整数加法；
  - 移位次数：2次；
  - 当n>1时，有T(n)=4T(n/2)+O(n)
  - $T(n)=O(n^2)$

算法复杂度：与小学算法没有改变

再接再厉

- $XY = AC\ 2^n + (AD+BC)\ 2^{n/2} + BD$
- $XY = AC\ 2^n + ((A-B)(D-C)+AC+BD)\ 2^{n/2} + BD$
  - 乘法次数：3次n/2位的整数乘法；
  - 加法：6次整数加法；
  - 移位：2次
  - 当n>1时，有$T(n)=3T(n/2)+O(n)$
  - $T(n)=O(n^{\log 3})=O(n^{1.59})$

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

复杂度分析

$$T(n)=\begin{cases} O(1) & n=1 \\ 3T(n/2)+O(n) & n>1 \end{cases}$$

$T(n)=O(n^{\log 3})=O(n^{1.59})$ ✓较大的改进☺

XY

为

1.

2. **XY = ac 2^n + ((a+c)(b+d)-ac-bd) $2^{n/2}$ + bd**

细节问题：两个XY的复杂度都是$O(n^{\log 3})$，但考虑到a+c, b+d可能得到m+1位的结果，使问题的规模变大，故不选择第2种方案。

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，对于大整数乘法，它能在**O(nlogn)**时间内解决。

➢是否能找到线性时间的算法？？？目前为止还没有结果。

通过例子来理解分治法

STRASSEN矩阵乘法
提高效率？ 乘法减少？

Matrix multiplication

Input: $A = [a_{ij}], B = [b_{ij}].$
Output: $C = [c_{ij}] = A \cdot B.$

$\Big\}$ $i, j = 1, 2, \ldots, n.$

$$\begin{bmatrix} c_{11} & c_{12} & \cdots & c_{1n} \\ c_{21} & c_{22} & \cdots & c_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ c_{n1} & c_{n2} & \cdots & c_{nn} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix} \cdot \begin{bmatrix} b_{11} & b_{12} & \cdots & b_{1n} \\ b_{21} & b_{22} & \cdots & b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n1} & b_{n2} & \cdots & b_{nn} \end{bmatrix}$$

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

Standard algorithm

for $i \leftarrow 1$ **to** n
  do for $j \leftarrow 1$ **to** n
    do $c_{ij} \leftarrow 0$
      for $k \leftarrow 1$ **to** n
        do $c_{ij} \leftarrow c_{ij} + a_{ik} \cdot b_{kj}$

Running time = $\Theta(n^3)$

Divide-and-conquer algorithm

IDEA:

$n \times n$ matrix = 2×2 matrix of $(n/2) \times (n/2)$ submatrices:

$$\begin{bmatrix} r & s \\ t & u \end{bmatrix} = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \cdot \begin{bmatrix} e & f \\ g & h \end{bmatrix}$$

$$C = A \cdot B$$

$$\left.\begin{aligned} r &= ae + bg \\ s &= af + bh \\ t &= ce + dh \\ u &= cf + dg \end{aligned}\right\}$$

8 mults of $(n/2) \times (n/2)$ submatrices

4 adds of $(n/2) \times (n/2)$ submatrices

Analysis of D&C algorithm

$$T(n) = 8T(n/2) + \Theta(n^2)$$

$$n^{\log_b a} = n^{\log_2 8} = n^3 \Rightarrow \text{CASE 1} \Rightarrow T(n) = \Theta(n^3).$$

No better than the ordinary algorithm.

Strassen's idea

- Multiply 2×2 matrices with only 7 recursive mults.

$$P_1 = a \cdot (f - h)$$
$$P_2 = (a + b) \cdot h$$
$$P_3 = (c + d) \cdot e$$
$$P_4 = d \cdot (g - e)$$
$$P_5 = (a + d) \cdot (e + h)$$
$$P_6 = (b - d) \cdot (g + h)$$
$$P_7 = (a - c) \cdot (e + f)$$

$$r = P_5 + P_4 - P_2 + P_6$$
$$s = P_1 + P_2$$
$$t = P_3 + P_4$$
$$u = P_5 + P_1 - P_3 - P_7$$

7 mults, 18 adds/subs.
Note: No reliance on commutativity of mult!

Strassen's idea

- Multiply 2×2 matrices with only 7 recursive mults.

$$
\begin{aligned}
P_1 &= a \cdot (f - h) \\
P_2 &= (a + b) \cdot h \\
P_3 &= (c + d) \cdot e \\
P_4 &= d \cdot (g - e) \\
P_5 &= (a + d) \cdot (e + h) \\
P_6 &= (b - d) \cdot (g + h) \\
P_7 &= (a - c) \cdot (e + f)
\end{aligned}
$$

$$
\begin{aligned}
r &= P_5 + P_4 - P_2 + P_6 \\
&= (a + d)(e + h) \\
&\quad + d(g - e) - (a + b)h \\
&\quad + (b - d)(g + h) \\
&= ae + ah + de + dh \\
&\quad + dg - de - ah - bh \\
&\quad + bg + bh - dg - dh \\
&= ae + bg
\end{aligned}
$$

Strassen's algorithm

1. ***Divide:*** Partition A and B into $(n/2)\times(n/2)$ submatrices. Form terms to be multiplied using $+$ and $-$.
2. ***Conquer:*** Perform 7 multiplications of $(n/2)\times(n/2)$ submatrices recursively.
3. ***Combine:*** Form C using $+$ and $-$ on $(n/2)\times(n/2)$ submatrices.

$$T(n) = 7\,T(n/2) + \Theta(n^2)$$

Analysis of Strassen

- 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 $O(n^{2.376})$
- 是否能找到$O(n^2)$的算法？？？目前为止还没有结果。

$T(n) = 7T(n/2) + \Theta(n^2)$

$n^{\log_b a} = n^{\log_2 7} \approx n^{2.81} \Rightarrow$ CASE 1 $\Rightarrow T(n) = \Theta(n^{\lg 7})$.

The number 2.81 may not seem much smaller than 3, but because the difference is in the exponent, the impact on running time is significant. In fact, Strassen's algorithm beats the ordinary algorithm on today's machines for $n \geq 30$ or so.

Best to date (of theoretical interest only): $\Theta(n^{2.376\cdots})$.

通过例子来理解分治法

归并排序
分治法在排序中的应用！

排序: Sort

设n个记录的序列为 $\{R_1, R_2, R_3, \ldots, R_n\}$

其相应的关键字序列为 $\{K_1, K_2, K_3, \ldots, K_n\}$

若规定$1, 2, 3, \ldots, n$的一个排列$p_1, p_2, p_3, \ldots, p_n$，使得相应的关键字满足如下非递减关系:

$$K_{p_1} \leq K_{p_2} \leq K_{p_3} \leq \ldots \leq K_{p_n}$$

则原序列变为一个按关键字有序的序列:

$$\{R_{p_1}, R_{p_2}, R_{p_3}, \ldots, R_{p_n}\}$$

此操作过程称为排序。

稳定排序与不稳定排序

假设 $K_i = K_j$，且排序前序列中 R_i 领先于 R_j；

若在排序后的序列中 R_i 仍领先于 R_j，则称排序方法是稳定的。

若在排序后的序列中 R_j 领先于 R_i，则称排序方法是不稳定的。

例，序列　3　15　8　8　6　9

若排序后得　3　6　8　8　9　15　　**稳定的**

若排序后得　3　6　8　8　9　15　　**不稳定的**

通俗地讲就是能保证排序前2个相等的数其在序列的前后位置顺序和排序后它们两个的前后位置顺序相同。

内部排序与外部排序

内部排序：指的是待排序记录存放在计算机随机存储器中进行的排序过程。

外部排序：指的是待排序记录的数量很大，以致内存一次不能容纳全部记录，在排序过程中尚需对外存进行访问的排序过程。

排序的时间复杂性

排序过程主要是对记录的排序码进行比较和记录的移动过程。

因此排序的时间复杂性以算法执行中的数据比较次数及数据移动次数来衡量。

当一种排序方法使排序过程在最坏或平均情况下所进行的比较和移动次数越少，则认为该方法的时间复杂性就越好。

分析一种排序方法，不仅要分析它的时间复杂性，而且要分析它的空间复杂性、稳定性和简单性等。

内部排序

按照排序过程中所依据的原则的不同可以分类为:

- 插入排序
- 归并排序
- 快速排序

插入排序 —— 直接插入排序

思想：利用有序表的插入操作进行排序

有序表的插入：将一个记录插入到已排好序的有序表中，从而得到一个新的有序表。

例，序列 13 27 38 65 76 97

插入 49

13 27 38 49 65 76 97

直接插入排序算法描述

初始，令第 1 个元素作为初始有序表；

依次插入第 2 , 3 , …, k 个元素构造新的有序表；

直至最后一个元素；

例，序列 49 38 65 97 76 13 27

初始，S = { 49 }；

{ 13 27 38 49 65 76 97 }

直接插入排序算法主要应用比较和移动两种操作。

直接插入排序代码示例

```
void insertsort(ElemType R[], int n) {
  //待排序元素用一个数组R表示，数组有n个元素
  for ( int i=1; i<n; i++) {   //i表示插入次数，共进行n-1次插入
     ElemType temp = R[i]; //把待排序元素赋给temp
     int j = i-1;
     while ((j>=0)&& (temp<R[j])) {
        R[j+1] = R[j]; j--;
      } // 顺序比较和移动
      R[j+1] = temp;
    }
}
```

直接插入排序的效率分析

从空间来看，它只需一个元素的辅助空间，用于元素的位置交换。

从时间分析，首先外层循环要进行n-1次插入，每次插入：

- 最少比较一次（正序），移动两次；
- 最多比较i次，移动i＋2次（逆序）（i=1，2，…，n-1）。

$C_{min} = n - 1$　　$M_{\min} = （n - 1）\times 2$;

$$C_{max} = 1 + 2 + \cdots + （n - 1） = \frac{n^2 - n}{2}$$

$$M_{\max} = 3 + 4 + \cdots + （n + 1） = \frac{n^2 + 3n - 4}{2}$$

$$C_{ave} = \frac{n^2 + n - 2}{4}$$

$$M_{ave} = \frac{n^2 + 7n - 8}{4}$$

因此，直接插入排序的时间复杂度为0（n^2）。

分析插入排序的稳定性

如果碰见一个和插入元素相等的，那么插入元素把想插入的元素放在相等元素的后面。所以，相等元素的前后顺序没有改变，从原无序序列出去的顺序就是排好序后的顺序，所以插入排序是稳定的。

直接插入算法的元素移动是顺序的，该方法是稳定的。

为什么强调“稳定性”？

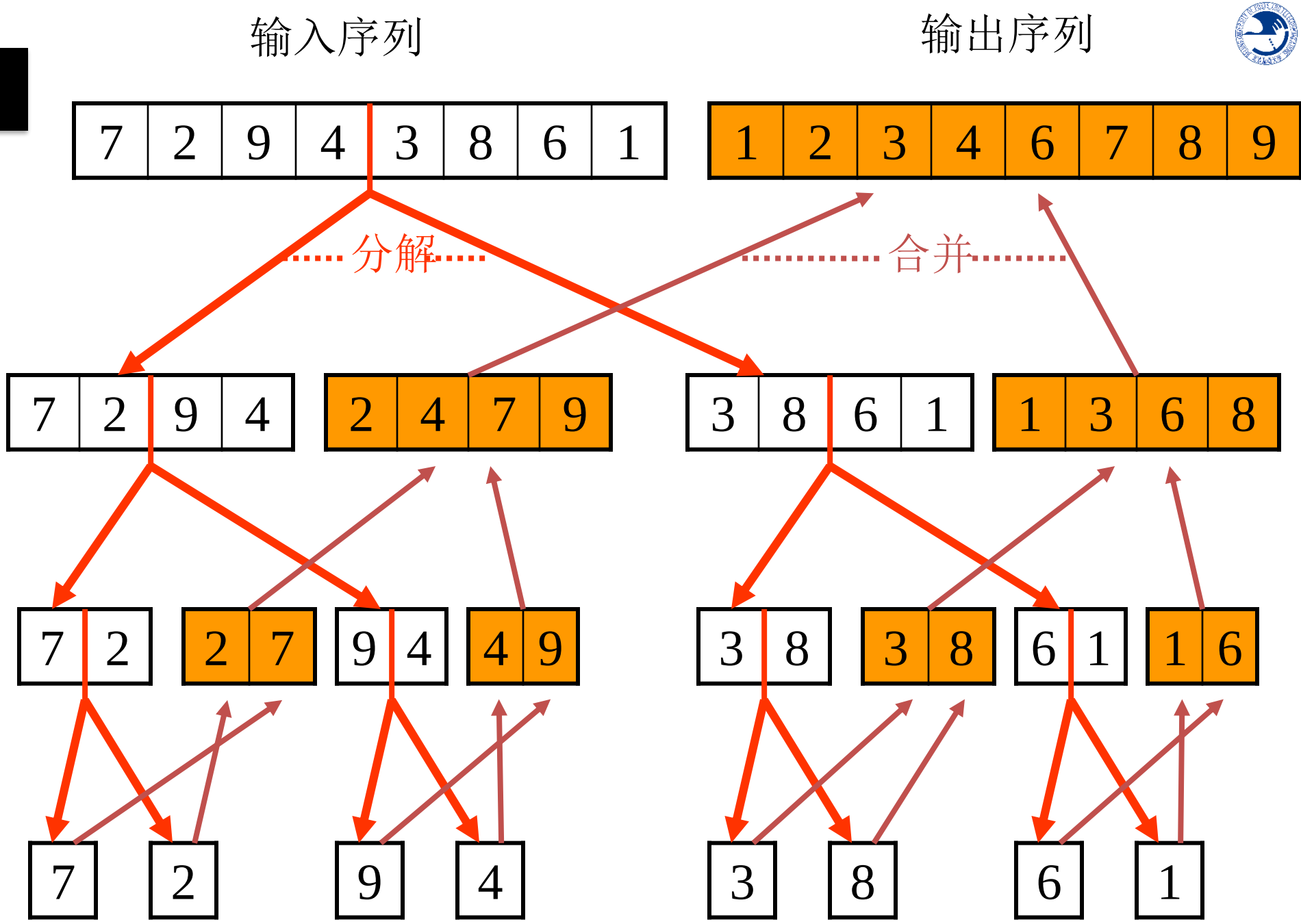
插入排序的复杂性分析

- $O(n^2)$
- 在这个算法中，大部分的时间都用在比较和挪动元素中，随着已经排好顺序的数组的增长，比较和被挪动的元素的个数也在增加，而且在整个过程中，很多元素不止挪动一次。

思考：如何用分治的思想解决问题？

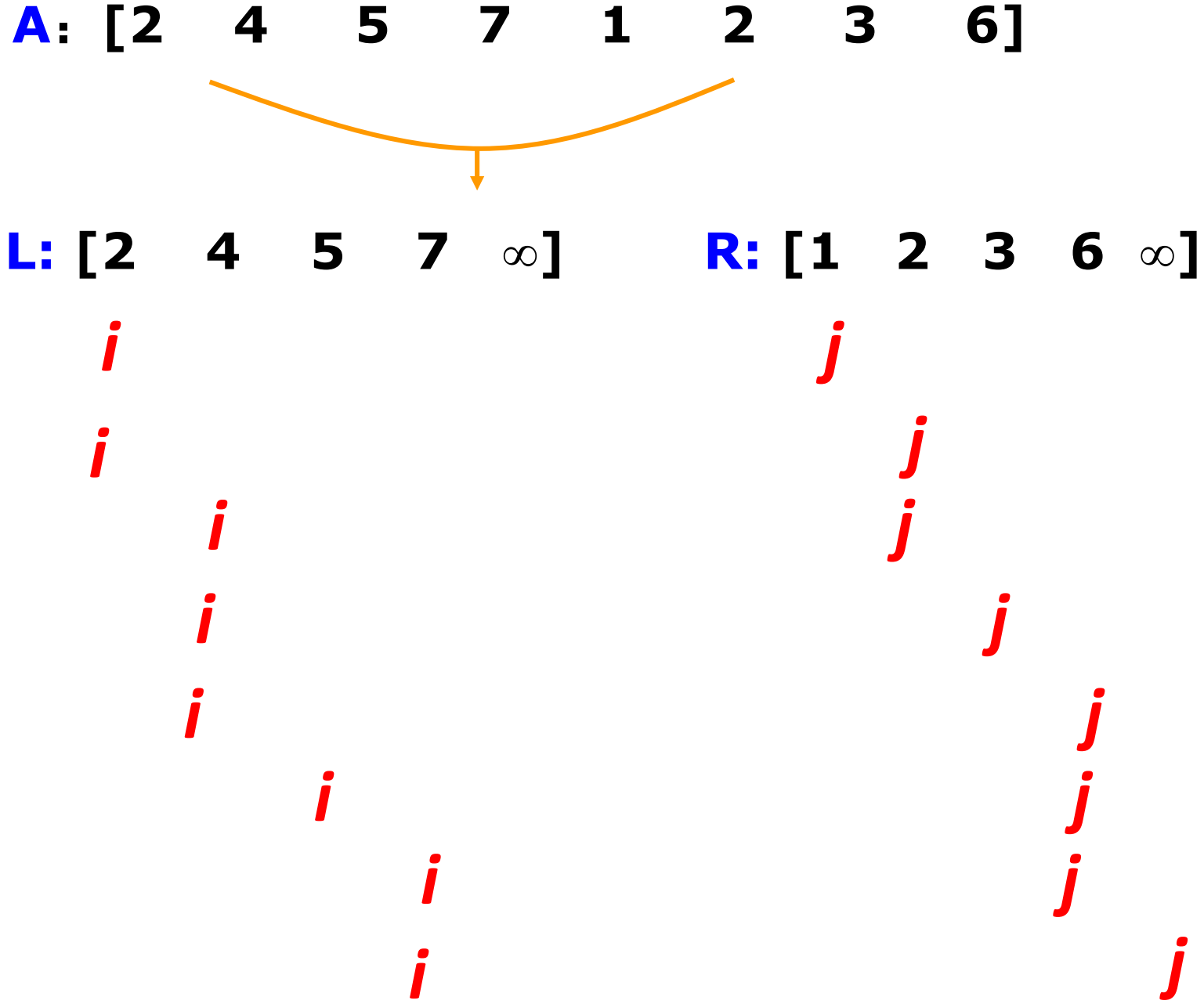
基本思想

- 分解（Divide)：
  - 将n个元素分成各含n/2个元素的子序列；
- 解决（Conquer)：
  - 递归地用合并排序法对两个子序列排序；
- 合并（Combine)：
  - 合并两个已经排好序的子序列。
- 在对子序列排序时，其长度为1时递归结束。单个元素被视为是已经排好序的。


输入序列
输出序列
7 2 9 4 3 8 6 1
1 2 3 4 6 7 8 9
分解
合并
7 2 9 4
2 4 7 9
3 8 6 1
1 3 6 8
7 2
2 7
9 4
4 9
3 8
3 8
6 1
1 6
7
2
9
4
3
8
6
1

MERGE-SORT

```
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);
      }
}
```


A: [2 4 5 7 1 2 3 6]
L: [2 4 5 7 ∞]
R: [1 2 3 6 ∞]
i
i
i
i
i
i
i
i
j
j
j
j
j
j
j

```
MERGE(A, p, q, r)
{
1:    n1 = q-p+1;              //length(L)
2:    n2 = r-q;                //length(R)
3:    creat_arrays L[1...n1+1],R[1...n2+1];
4:    for i=1:n1
5:       do L[i] = A[p+i-1];
6:    for j=1:n2
7:       do R[j] = A[q+j];
8:    L[n1+1] = MAX;
9:    R[n2+1] = MAX;
10:   i = 1;
11:   j = 1;
12:   for k = p:r
13:   {
14:      if L[i] <= R[j]
15:      {
16:         A[k] = L[i];
17:         i = i+1;
18:      }
19:      else
20:      {
21:         A[k] = Q[j];
22:         j = j+1;
23:      }
24:   }
}
```

归并排序

基本思想：将待排序元素分成大小大致相同的2个子集合，分别对2个子集合进行排序，最终将排好序的子集合合并成为所要求的排好序的集合。

```
public static void mergeSort(Comparable a[], int left, int right)
{
    if (left<right) {//至少有2个元素
        int i=(left+right)/2;  //取中点
        mergeSort(a, left, i);
        mergeSort(a, i+1, right);
        merge(a, b, left, i, right);  //合并到数组b
        copy(a, b, left, right);    //复制回数组a
    }
}
```

用T(n)表示归并排序所用的时间，

并假定合并过程所用时间：cn，其中c是一个正数，则有

$$T(n)=\begin{cases} a & n=1 \\ 2T(n/2)+cn & n>1 \end{cases}$$

其

复杂度分析

$$T(n)=\begin{cases} O(1) & n\le 1 \\ 2T(n/2)+O(n) & n>1 \end{cases}$$

$T(n)=O(nlogn)$ 渐进意义下的最优算法

$$= 2^k T(1) + kcn$$

$$= an + cn \log n$$

对于一般的整数n，我们可以假定 $2^k < n \le 2^{k+1}$，于是，

$$T(n) \le T(2^{k+1}) \quad , \quad T(n) = O(n \log n)$$

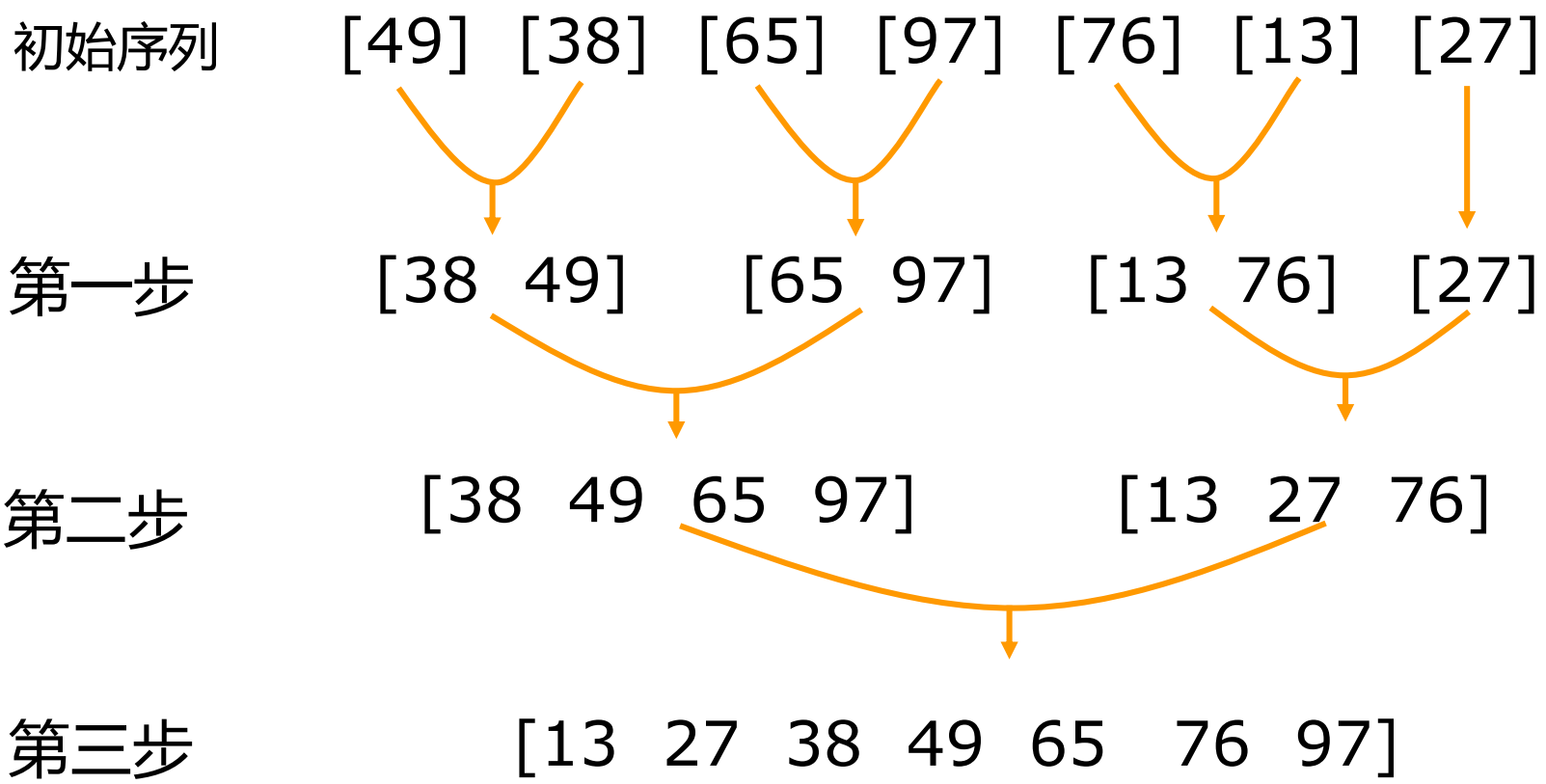
归并排序的灵活革新

基本思想：将待排序元素分成大小大致相同的2个子集合，分别对2个子集合进行排序，最终将排好序的子集合合并成为所要求的排好序的集合。

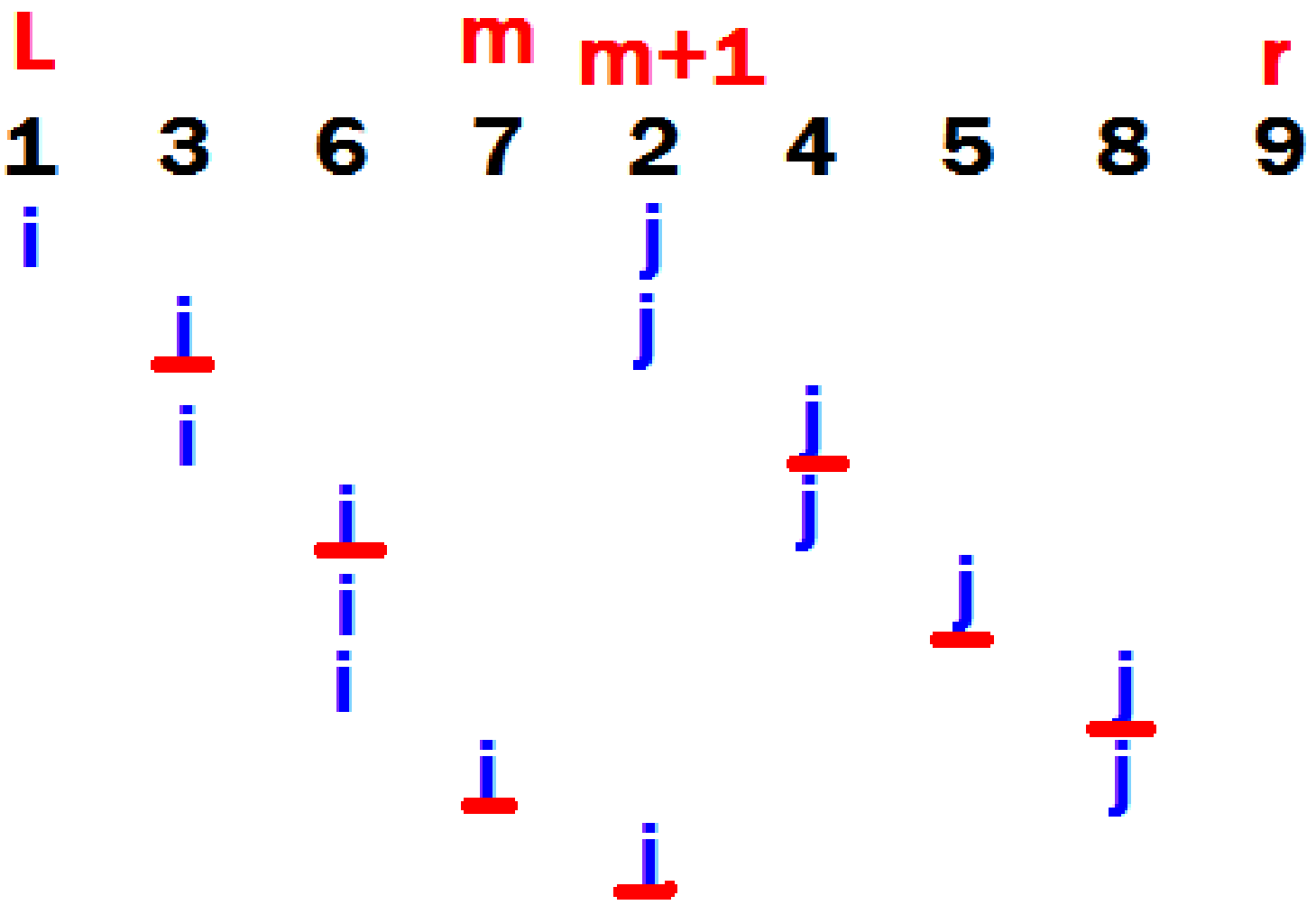
- 首先将长度为1的n个数组相邻元素两两配对，构成了长度为2的n/2个数组，合并时用比较算法对这每个子数组中元素进行排序；
- 再将这些长度为2的n/2个数组两两合并，构成了长度为4，个数为n/4的子数组，合并时用比较算法对每个子数组元素排序。
- 继续
- 直到形成长度为n，子数组个数=1的整个数组为止。

归并排序

算法**mergeSort**的递归过程可以消去。



如何将两个有序段合并成一个有序段?



```
//合并算法

void Merge( Type c[], Type d[], int l, int m, int r)
{
        int i = l, j = m+1, k = 1;
        while(( i <=m ) && ( j <= r ))
        {
                if ( c[i] <= c [j] ) d[k++] = c[i++];
                else d[k++] = c [j++];
        }
        if ( i > m )
        {
                for (int q = j; q <= r; q ++)
                        d[k++] = c[q];
        }
        else    for ( int q =i; q <= m; q++ )
                        d[k++} = c[q];
}
```

```
//mergepass

void MergePass ( Type x[], Type y[], int s, int n )
{
        int i = 0;
        while ( i <= n - 2 * s )
        {//合并大小为s的相邻2段子数组
                Merge( x, y, i, i+s-1, i+2*s-1);
                i = i + 2 * s;
        }
        //剩下的元素个数少于2s
        if ( i + s < n ) Merge ( x, y, i, i+s-1, n-1);
        else for (int j = i; j <= n-1; j++)
                y[j] = x[j];
}
```

//合并排序算法

```
void MergeSort ( Type a[], int n )
{
        Type *b = new Type [n];
        int s = 1;
        while ( s < n )
        {
                MergePass( a, b, s, n);
                s += s;
                MergePass( b, a, s, n);
                s += s;
        }
}
```

归并排序（Merge Sort)

- 算法分析
  - 最坏情况：归并排序是一个递归算法，所以很容易得到算法计算量的递推公式

$$W(n) = W(\lfloor n/2 \rfloor) + W(\lceil n/2 \rceil)$$
$$W(1) = 0$$

所以算法最坏情况的复杂度为 $\theta(n \log n)$

  - 算法需要 $\theta(n)$ 的辅助空间

归并排序（Merge Sort)

- 其他策略
  - 上面归并排序每次迭代时将原序列分割为两个基本等长的序列，称为二路归并排序
  - 也可以分割成其他的分，如3，4等等
- **分析归并排序的稳定性？**

归并排序是把序列递归地分成短序列，递归出口是短序列只有1个元素(认为直接有序)或者2个序列(1次比较和交换),然后把各个有序的段序列合并成一个有序的长序列，不断合并直到原序列全部排好序。
可以发现，在1个或2个元素时，1个元素不会交换，2个元素如果大小相等也没有人故意交换，这不会破坏稳定性。
那么，在短的有序序列合并的过程中，稳定性是否受到破坏？没有，合并过程中我们可以保证如果两个当前元素相等时，我们把处在前面的序列的元素保存在结果序列的前面，这样就保证了稳定性。

所以，归并排序也是稳定的排序算法。

归并排序

- 最坏时间复杂度：O(nlogn)
- 平均时间复杂度：O(nlogn)
- 稳定性：稳定