算法设计与分析
Design and Analysis of Algorithms

第四章　分治法
Divide and Conquer

主讲：　朱东杰 博士、硕导
地点：　M楼305
电话/微信：　18953856806
Email：　zhudongjie@hit.edu.cn

分治法

- 教学内容
  - 分治法的一般方法
  - 合并排序
  - 棋盘覆盖
  - 快速排序
  - 折半查找
  - 最大值和最小值
  - 二叉树遍历及其相关特性
  - 大整数乘法
  - Strassen矩阵乘法
  - 最接近点问题
- 要求
  - 掌握分治法的原理、效率分析以及在常见问题问题中的应用。

Divide and Conquer

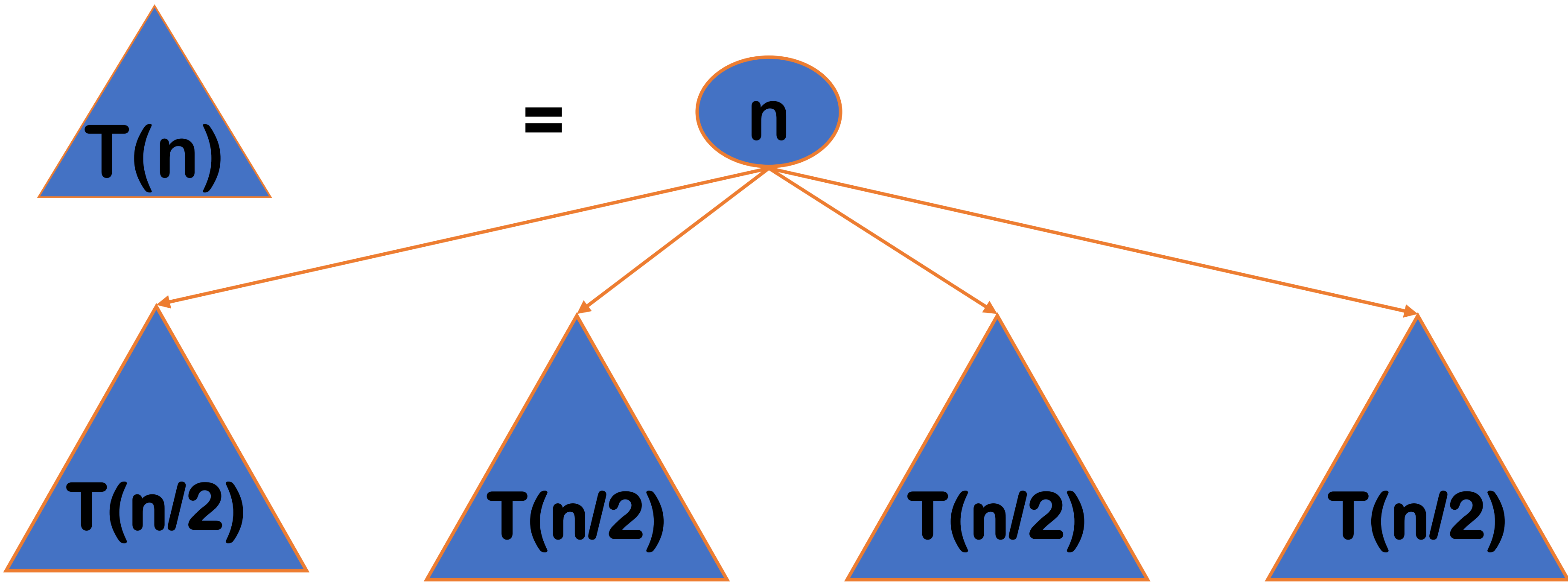
- A divide and conquer algorithm consists of two parts:
  - Divide the problem into smaller subproblems of the *same type*, and solve these subproblems *recursively*
  - Combine the solutions to the subproblems into a solution to the original problem
- Traditionally, an algorithm is only called “divide and conquer” if it contains at least two recursive calls

分治法

对于一个规模为n的问题，若该问题可以容易地解决（比如说规模n较小）则直接解决，否则将其分解为k个规模较小的子问题，这些子问题互相独立且与原问题形式相同，递归地解这些子问题，然后将各子问题的解合并得到原问题的解。这种算法设计策略叫做*分治法*。

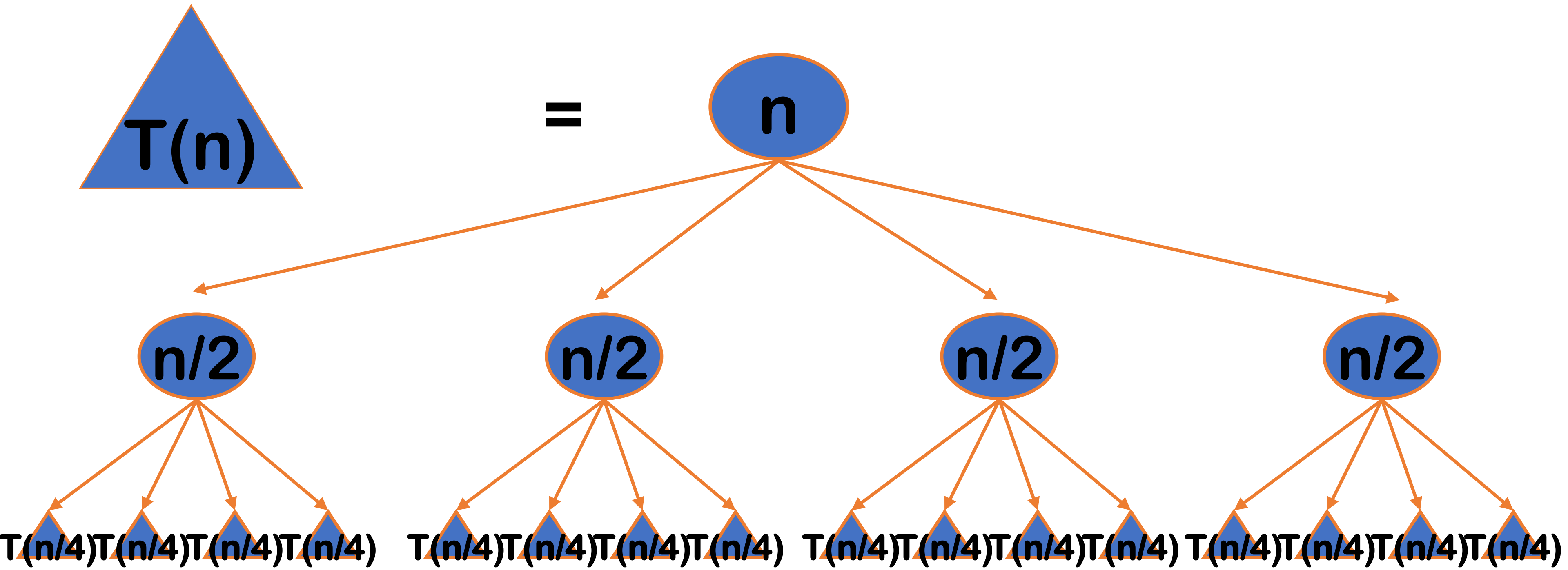
算法总体思想

- 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。



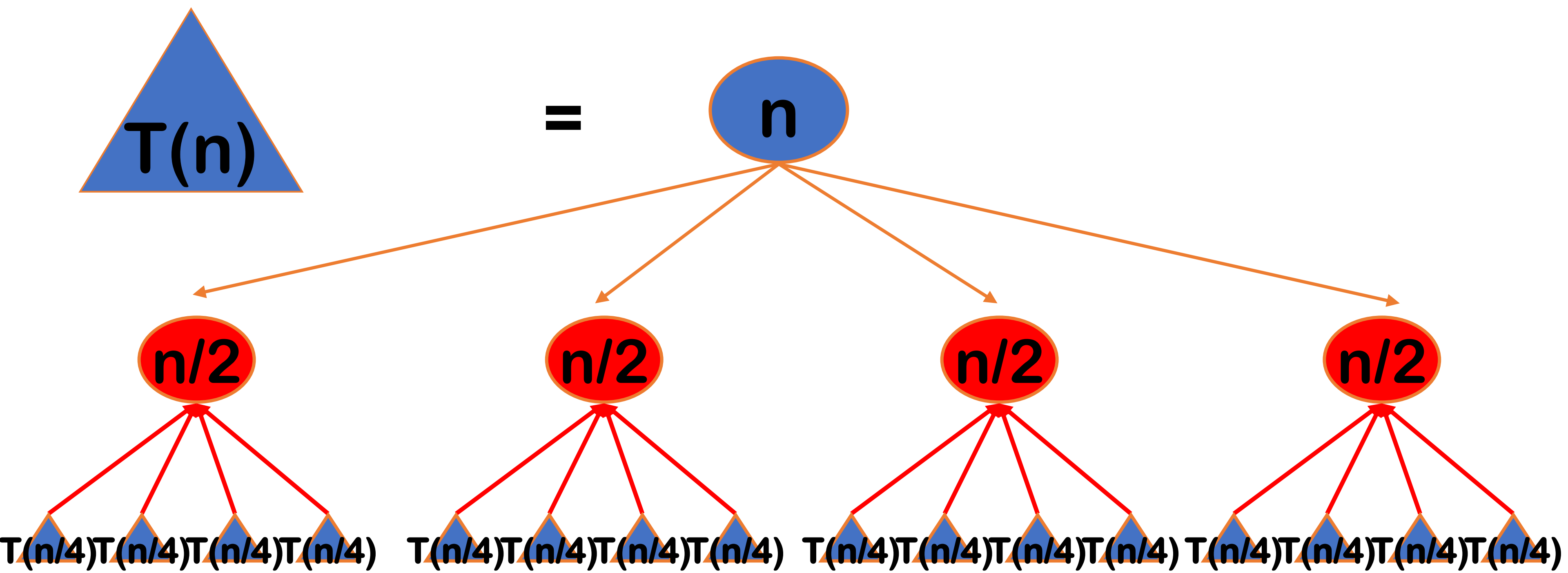
算法总体思想

- 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。

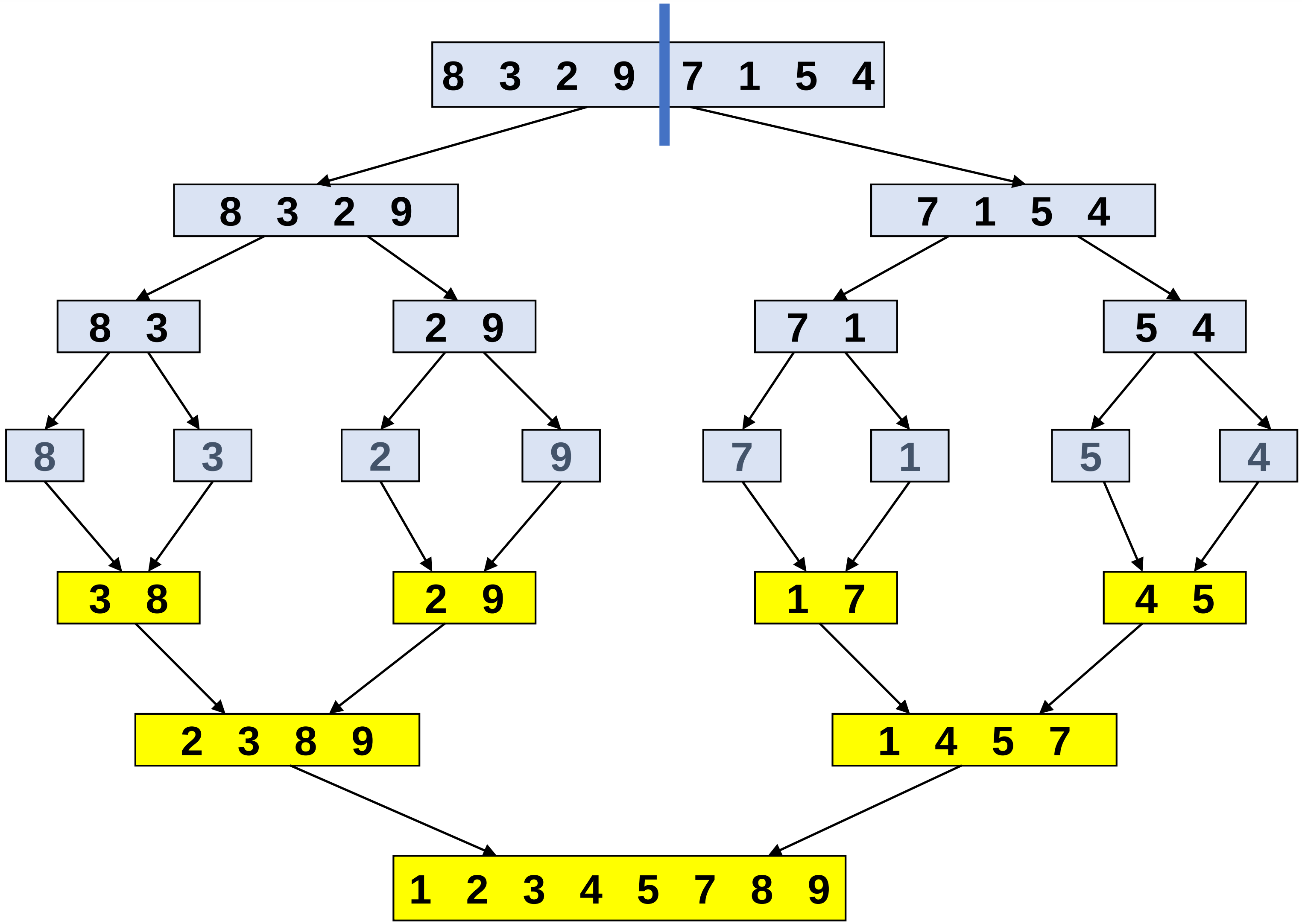


算法总体思想

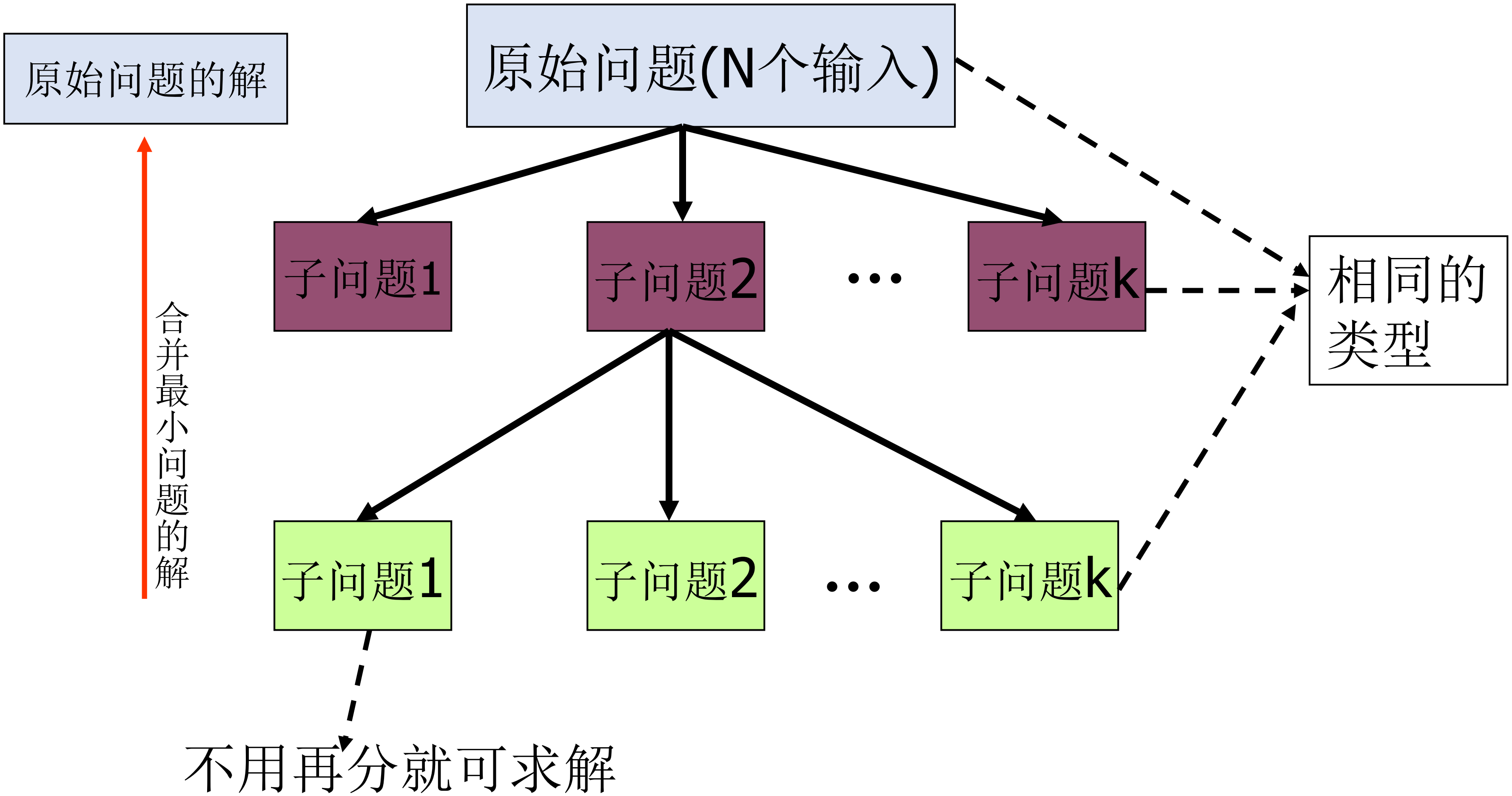
- 将求出的小规模的问题的解合并为一个更大规模的问题的解，自底向上逐步求出原来问题的解。



例：合并排序


8 3 2 9 7 1 5 4
8 3 2 9
7 1 5 4
8 3
2 9
7 1
5 4
8
3
2
9
7
1
5
4
3 8
2 9
1 7
4 5
2 3 8 9
1 4 5 7
1 2 3 4 5 7 8 9

分治法的一般方法


原始问题的解
原始问题(N个输入)
合并最小问题的解
子问题1
子问题2
…
子问题k
相同的类型
子问题1
子问题2
…
子问题k
不用再分就可求解

分治法的步骤及适用条件

- 分治法的几个步骤：
- 1、将原始问题划分为k个相同类型的子问题。（问题：为什么？）
- 2、子问题不可解还可继续划分。（问题：分到什么时候结束？）
- 3、求解每个子问题。
- 4、将每个最小子问题的解合并成原问题的解。
- 分治法适用条件：
- 1、原始可分解，且分解出来的子问题和原始问题就有相同的类型。
- 2、分解出来的子问题到很小时可以很容易（在很短的时间和空间内能求解）求解。
- 3、子问题的解能合并。

分治法的抽象化控制

- Divide-and-Conquer(P)

1. if $|P| \le n_0$ 判断输入规模是否足够的小
2. then return(ADHOC(P))//直接解小规模的问题P
3. 将P分解为较小的子问题 $P_1, P_2, \ldots, P_k$
4. for $i \leftarrow 1$ to k
5. do $yi \leftarrow$ Divide-and-Conquer(P_i)//递归解决Pi
6. $T \leftarrow$ MERGE($y_1, y_2, \ldots, y_k$) // 合并子问题
7. return(T)

将子问题的解合成原问题

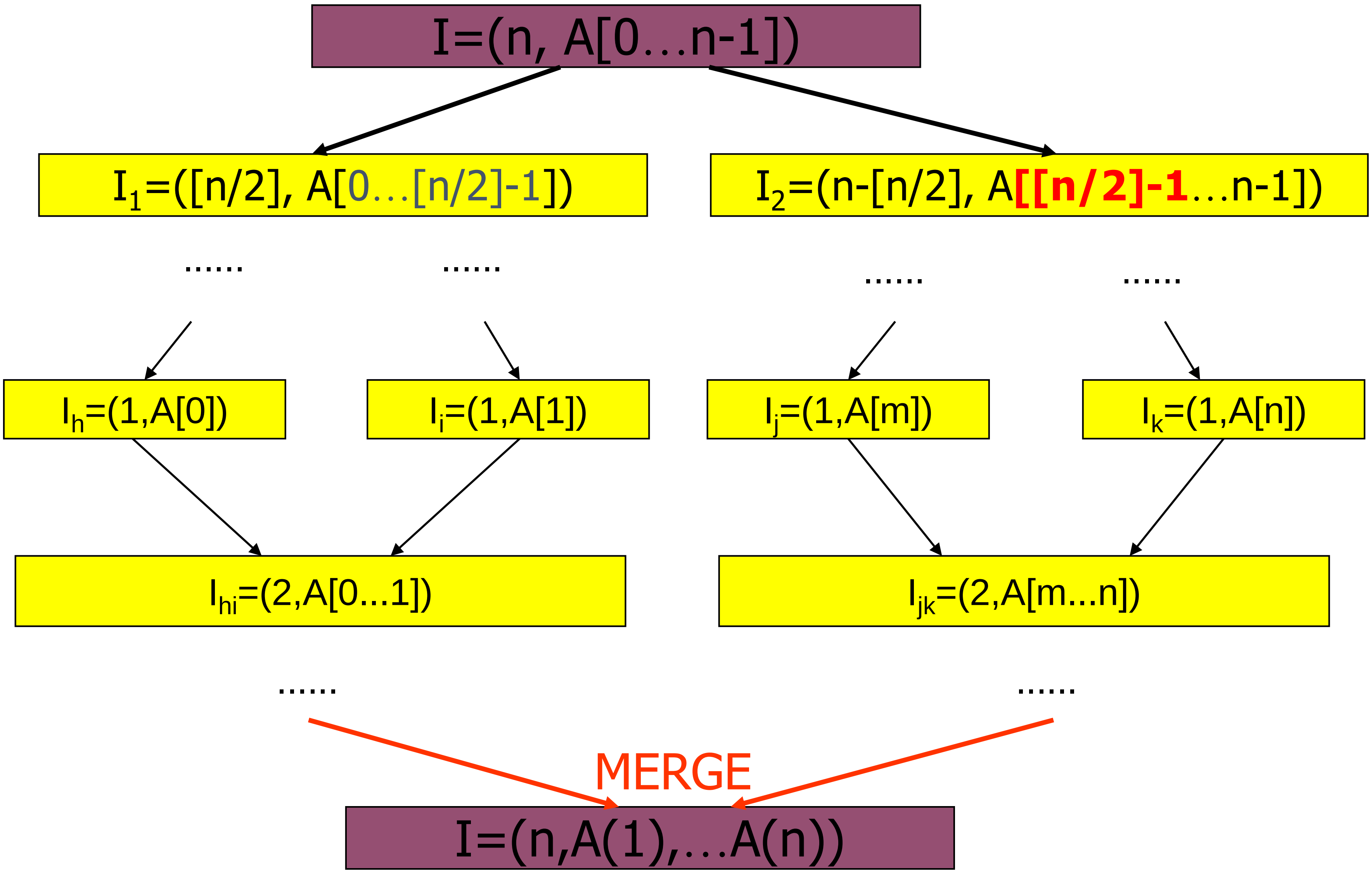
Divide-and-Conquer的计算时间

- 倘若所分成的两个子问题的输入规模大致相等，则Divide-and-Conquer的计算时间可表示为：

$$T(n)=\begin{cases} g(n) & n\text{足够小} \\ 2T(n/2)+f(n) & \text{否则} \end{cases}$$

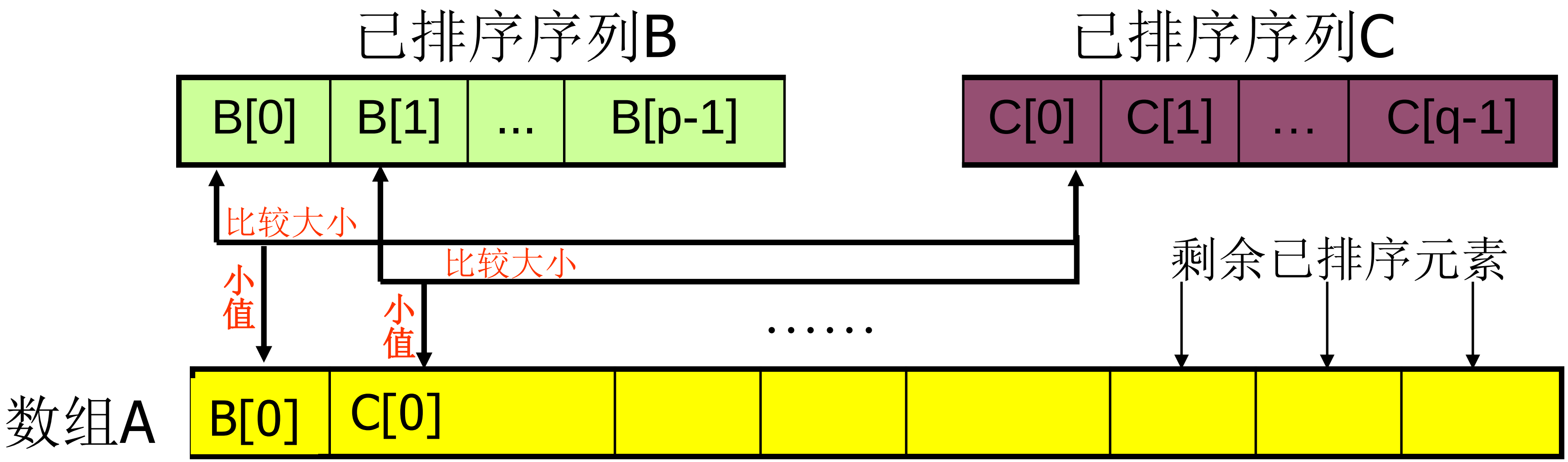
说明：T(n) 输入规模为n的Divide-and-Conquer的计算时间
g(n)是对足够小的输入规模能直接计算出答案的时间
f(n)是COMBINE的计算时间（Master/主定理）

合并排序merge sort



合并函数MERGE的实现

合并函数MERGE的实现思想



合并排序算法

算法 Mergesort(A[0...n-1])

//递归调用Mergesort来对数组A排序

//输入：可排序数组A[0..n-1]

//输出：非降序列数组A[0..n-1]

```
if (n>1){
    copy A[0.. [n/2]-1] to B[0..[n/2]-1] ;
    copy A[[n/2]..n-1] to C[0..[n/2]-1];
    Mergesort(B[0..[n/2]-1]);
    Mergesort(C[0..[n/2]-1]);
    Merge(B,C,A);
}
```

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

//将两个有序数组合并成一个有序数组

//输入：两个有序数组B[0..p-1]和C[0..q-1]

//输出：非降序列数组A-0..p+q-1]

```
i←0; j←0; k←0;
while (i<p and j<q do){
    if (B[i] ≤ C[j]) {
        A[k]←B[i];i←i+1;}
    else{
        A[k]←C[j];j←j+1;}
    k←k+1;
}
if (i=p)
    copy C[j..q-1] to A[k...p+q-1]
else
    copy B[j..p-1] to A[k...p+q-1]
```

算法分析

算法 Mergesort(A[0...n-1])

如果归并运算的时间与n成正比，则**归并分类的计算时间**可用递归关系式描述如下

```
if (n>1){
    copy A[0.. [n/2]-1] to B[0..[n/2]-1] ;
    copy A[[n/2]..n-1] to C[0..[n/2]-1];
    Mergesort(B[0..[n/2]-1]);
    Mergesort(C[0..[n/2]-1]);
    Merge(B,C,A);
}
```

$$T(n) = \begin{cases} a & \text{if } n = 1 \\ 2 \times T(\frac{n}{2}) + cn & \text{if } n \geq 2 \end{cases}$$

合并排序的计算时间

$$T(n)=\begin{cases} 0 & n=1 \\ 2T(n/2)+T_{merge}(n) & n>1 \end{cases}$$

当$n=2^k$时，$T_{merge}(n)=n$

可得:

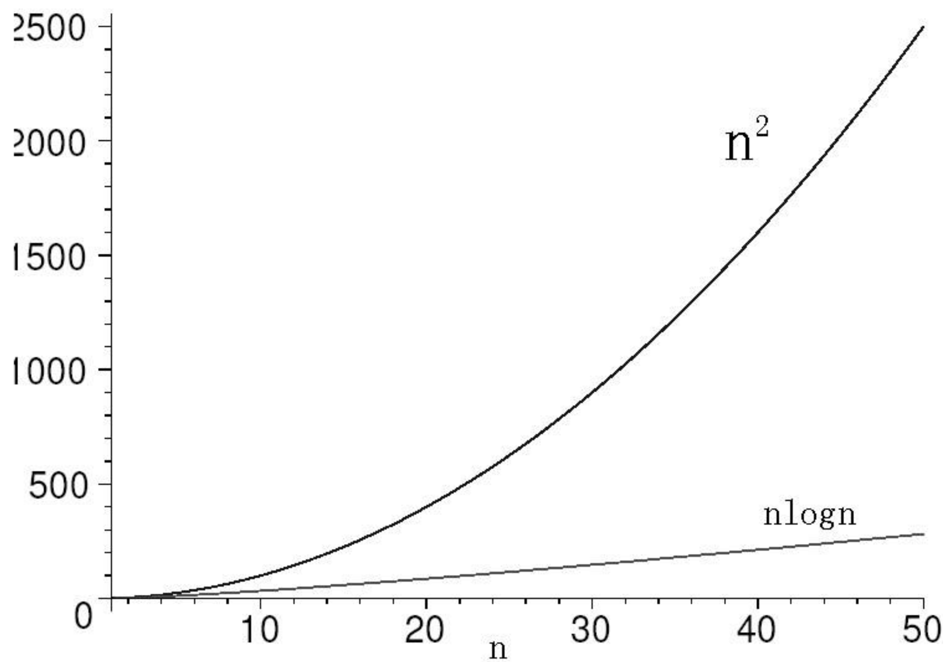
$$\begin{aligned}T(n)&=2(2T(n/4)+n/2)+n\\&=4T(n/4)+2n\\&=4(2T(n/8)+n/4)+2n\\&\ldots\ldots\\&=2^kT(1)+kn=kn\end{aligned}$$

因为： $n=2^k$，$k=\log_2 n$

所以： $T(n)=kn=n\log_2 n$

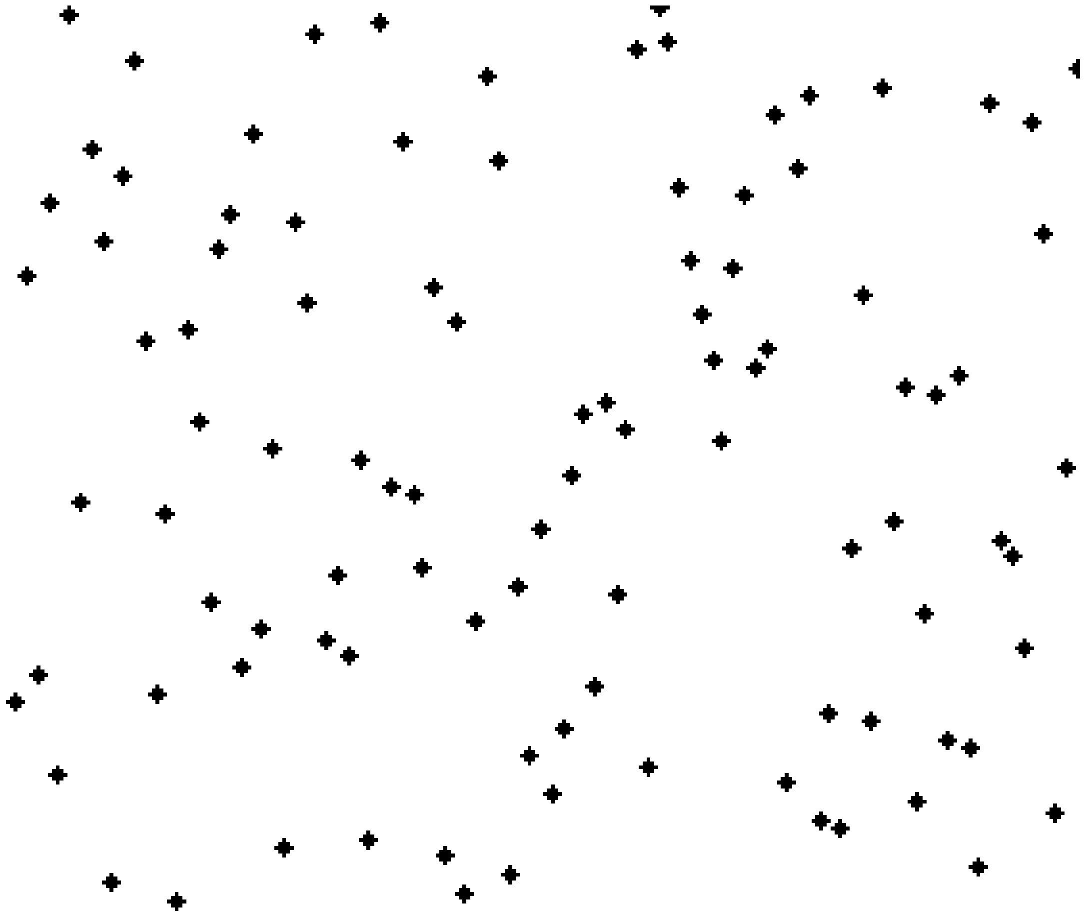
如果$2^k<n<2^{k+1}$，有$T(n)\le T(2^{k+1})$，

有 $T(n)=\Theta(n\log n)$



Example of merge sort

- Example of merge sort sorting a list of random numbers



思考

- 当实例较少时，合并排序的效率如何？
- 合并排序的空间效率如何？
- 合并排序对特殊数据是否会退化？
- 在划分子问题时是否可以大于2等分，如果可以，效率如何？

问题？

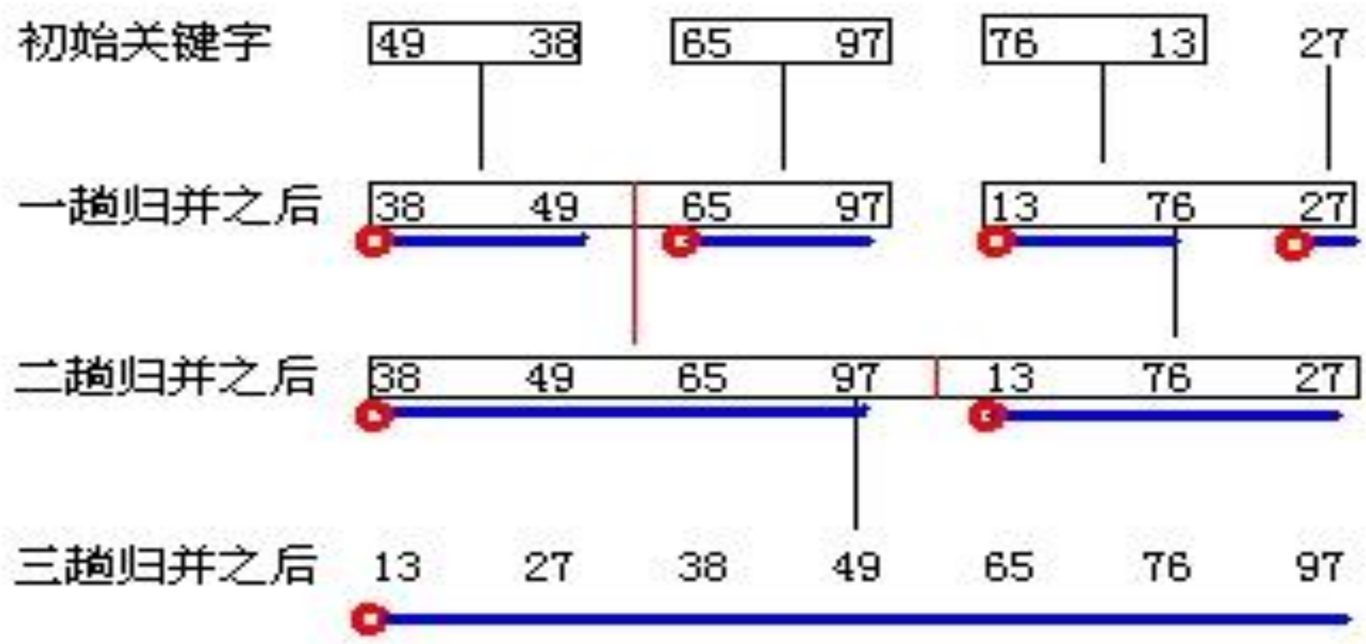
- 低层递归调用过多
- 复制空间

二路归并排序算法

使用链表的归并分类

二路归并排序算法

首先将每两个相邻的大小为1的子序列归并，然后对上一次归并所得到的大小为2的子序列进行相邻归并，如此反复，直至最后归并到一个序列，归并过程完成。通过轮流地将元素从a 归并到b 并从b 归并到a，可以虚拟地消除复制过程。



2－路归并排序示例

二路归并排序算法1

- 首先将每两个相邻的大小为1的子序列归并，然后对上一次归并所得到的大小为2的子序列进行相邻归并，如此反复，直至最后归并到一个序列，归并过程完成。通过轮流地将元素从a 归并到b 并从b 归并到a，**可以虚拟地消除复制过程**。二路归并排序算法见程序：

程序：二路归并排序

```
template<class T>
void MergeSort(T a[], int n)
{// 使用归并排序算法对a[0:n-1] 进行排序
T *b = new T [n];
int s = 1; // 段的大小
while (s < n) {
MergePass(a, b, s, n); // 从a归并到b
s += s;
MergePass(b, a, s, n); // 从b 归并到a
s += s;
}
}
```

二路归并排序算法2

为了完成排序代码，首先需要完成函数M e rg e P a s s。函数M e rg e P a s s仅用来确定欲归并子序列的左端和右端，实际的归并工作由函数M e rg e 来完成。函数M e rg e 要求针对类型T定义一个操作符< =。如果需要排序的数据类型是用户自定义类型，则必须重载操作符< =。这种设计方法允许我们按元素的任一个域进行排序。重载操作符< =的目的是用来比较需要排序的域。

程序 MergePass函数

```
template<class T>
void MergePass(T x[], T y[], int s, int n)
{// 归并大小为s的相邻段
int i = 0;
while (i <= n - 2 * s) {
// 归并两个大小为s的相邻段
Merge(x, y, i, i+s-1, i+2*s-1);
i = i + 2 * s;
}
// 剩下不足2个元素
if (i + s < n) Merge(x, y, i, i+s-1, n-1);
else for (int j = i; j <= n-1; j++)
// 把最后一段复制到y
y[j] = x[j];
}
```

二路归并排序算法3

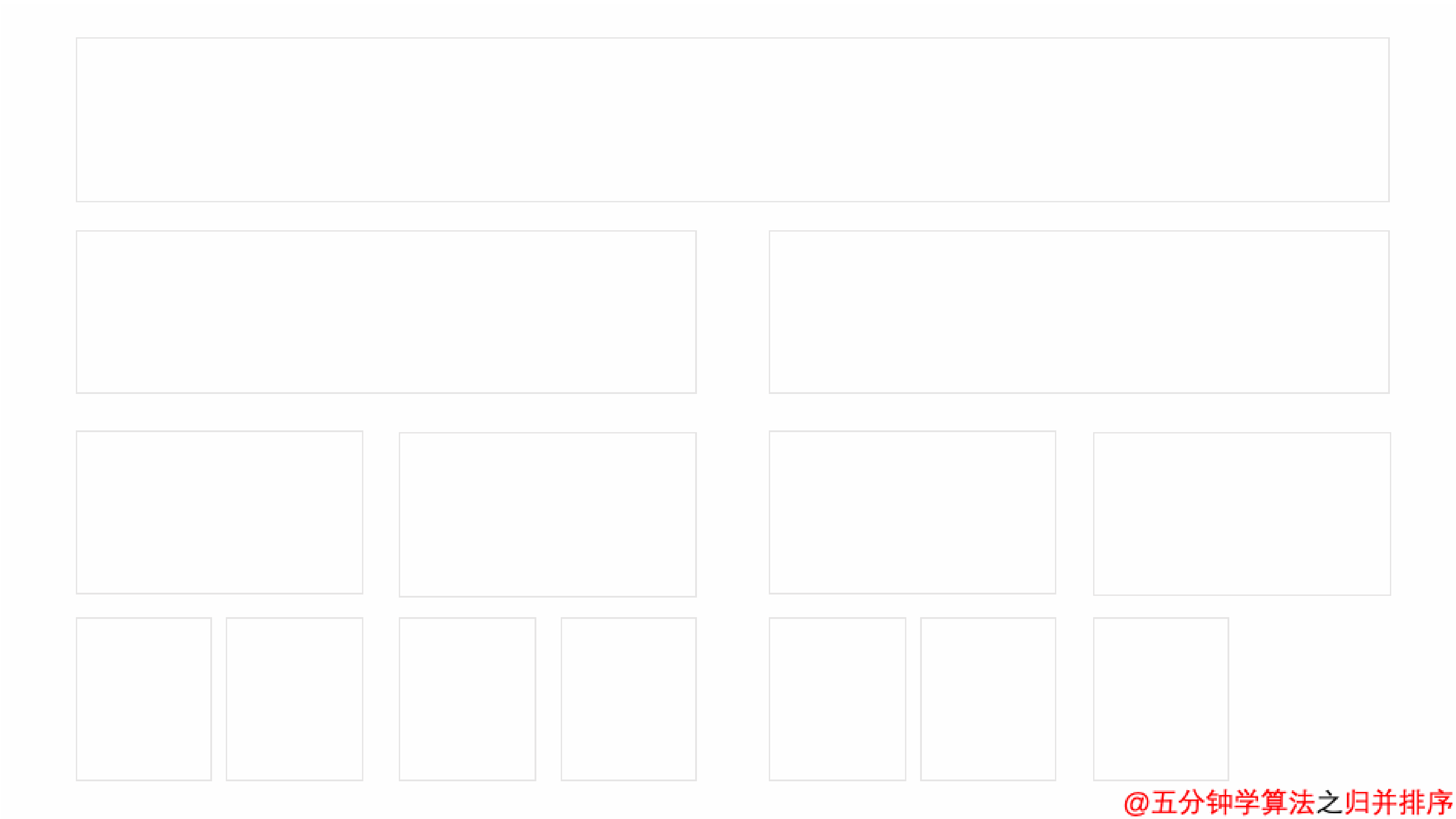
•程序 Merge函数

```
template<class T>
void Merge(T c[], T d[], int l, int m, int r)
{// 把c[l:m]] 和c[m:r] 归并到d [ l : r ] .
int i = l, // 第一段的游标
j = m+1, // 第二段的游标
k = l; // 结果的游标
/ /只要在段中存在i和j，则不断进行归并
while ((i <= m) && (j <= r))
if (c[i] <= c[j]) d[k++] = c[i++];
else d[k++] = c[j++];
// 考虑余下的部分
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];
}
```

归并排序时间复杂度：

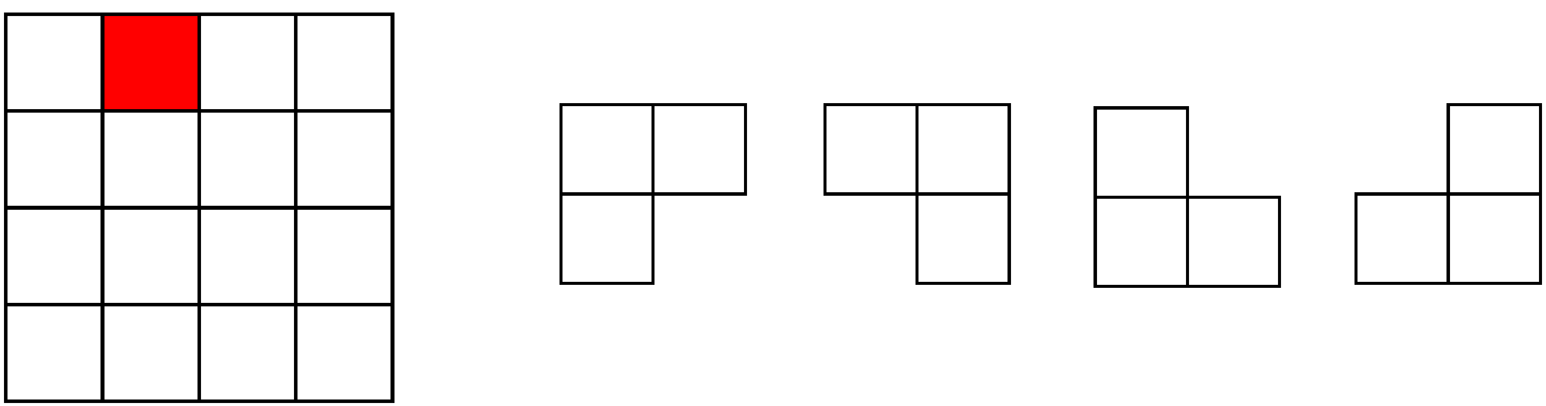
O(n log n)

二路归并排序算法4



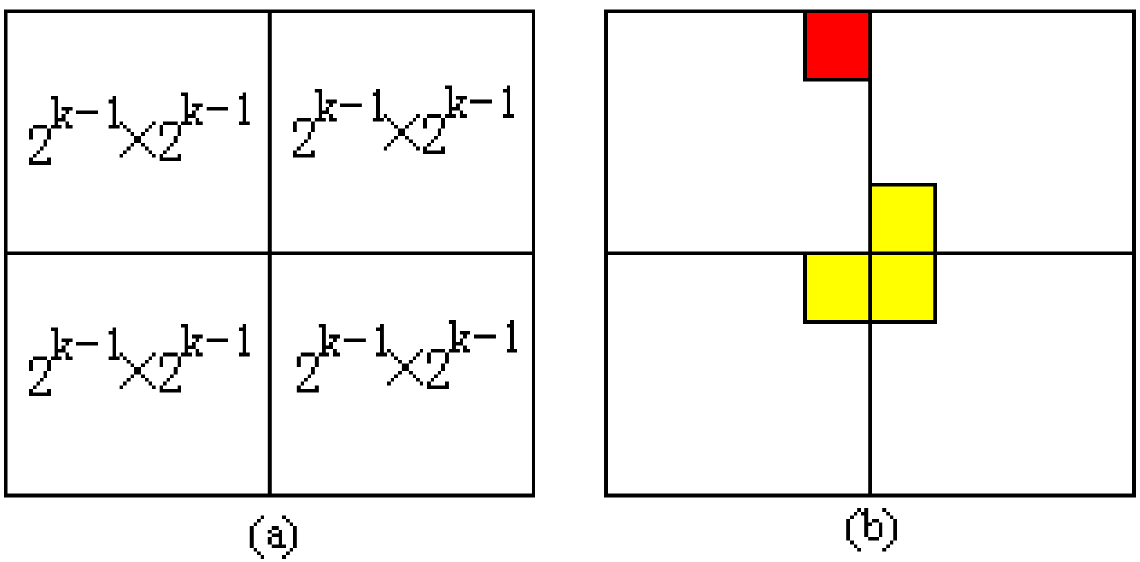
棋盘覆盖

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。

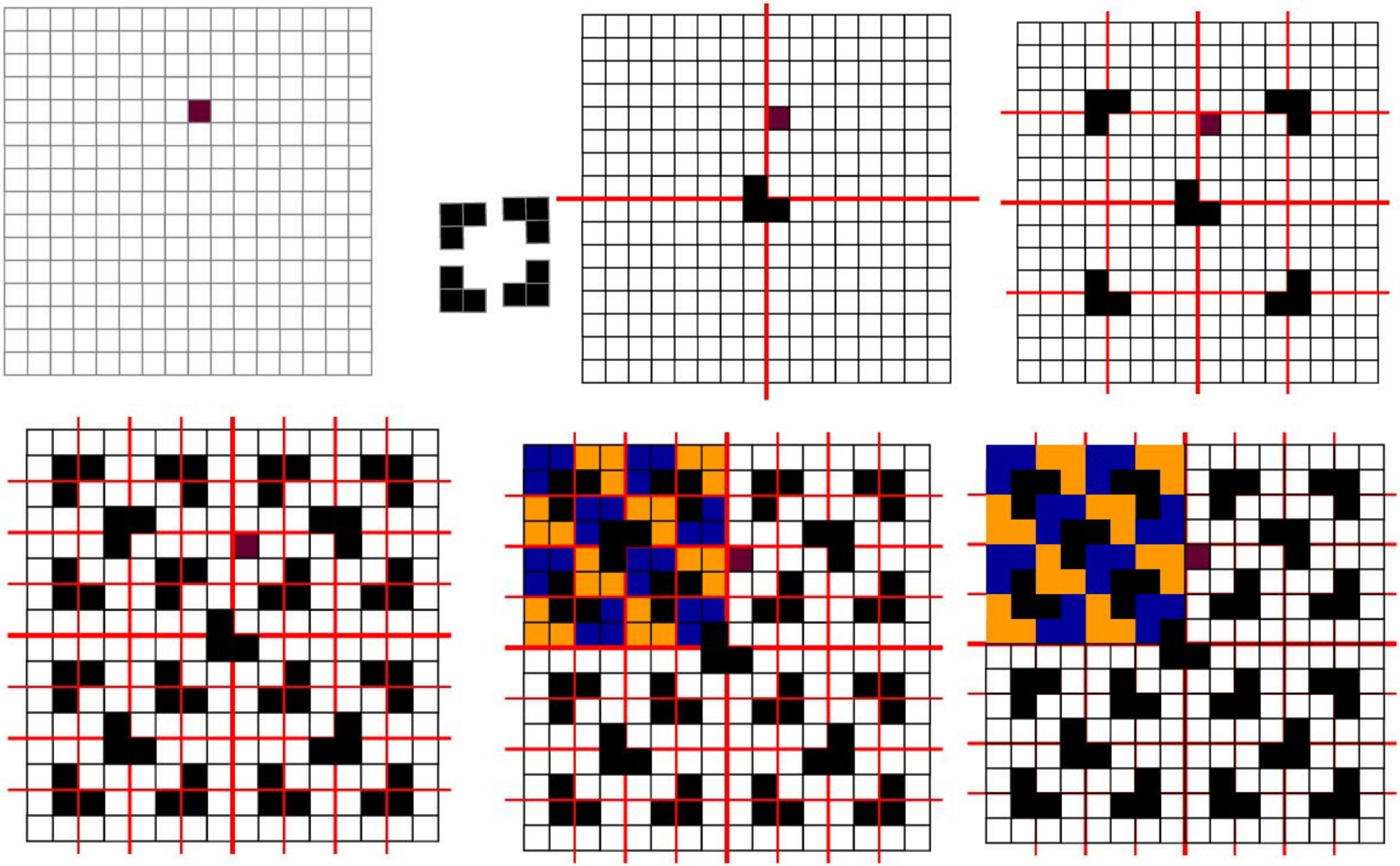


棋盘覆盖

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1×1。



棋盘覆盖



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
      // 特殊
      chess
    else {// 此
      // 用 t
      board[
      // 覆盖其
      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)
    ;}
    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);}
  }
```

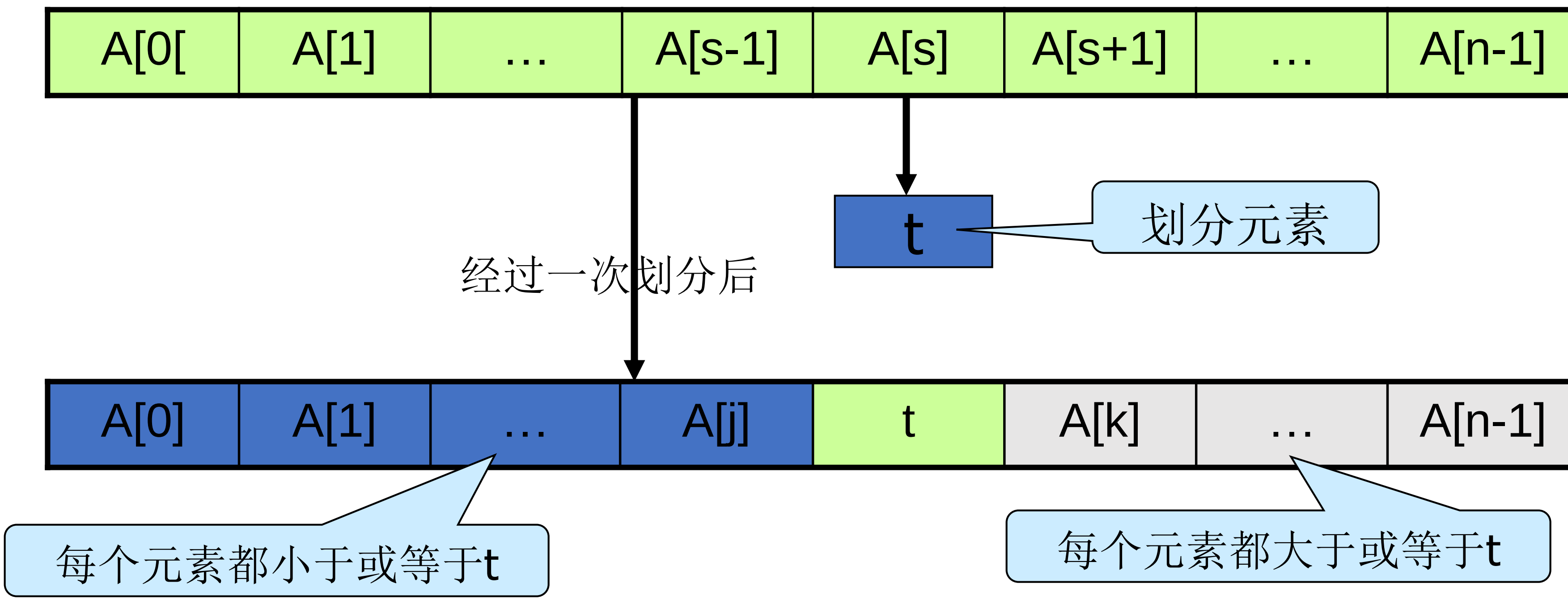
复杂度分析

$$T(k)=\begin{cases} O(1) & k=0 \\ 4T(k-1)+O(1) & k>0 \end{cases}$$

$T(n)=O(4^k)$ 渐进意义下的最优算法

快速排序quicksort

- 基本思想
  - 选取A的某个元素t=A[s]，然后将其他元素重新排列，使A[0..n-1]中所有在t以前出现的元素都小于或等于t，而在t之后出现的元素都大于或等于t。



划分的实现

- 划分元素（中轴）的选取：
  - 可以随机选取，**最简单的选择策略是数组第一个元素**
- 划分的实现思想
  - **同时**从左、右开始扫描，左边找到一个比划分元素大的，右边找到一个比划分元素小的，然后交换两个元素的位置。

		→i		j←	
p	全部≤p	≥p	...	≤p	全部≥ p

		j←	→i	
p	全部≤p	≤ p	≥p	全部≥ p

		→i= j←	
p	全部≤p	= p	全部≥ p

Quicksort in action on a list of random numbers

@五分钟学算法之快速排序

快速排序算法

算法 Quicksort(A[l...r])

```
//用Quicksort对子数组排序
//输入：数组A[0..n-1]中的子数组
//输出：非降序的子数组A[l...r]
if (l<r){
    s ← Partition(A[l...r]);
    Quicksort(A[l...s-1]);
    Quicksort(A[s+1...r]);
}
```

算法 Partition(A[l..r])

```
//以第一个元素作为中轴，划分数组
//输入：数组A[l..r]，l,r为左右下标
//输出：A[l..r]的一个分区，返回划分点位置
p←A[l];
i←l; j←r+1;
reapeat
    reapeat i←i+1 until A[i] ≥p；
    reapeat j←j-1 until A[j] ≤p；
    swap(A[i],A[j]);
until i ≥ j
swap(A[i],A[j]);//i ≥ j,撤销最后一次交换
swap(A[l],A[j]);
return j;
```

快速排序算法时间复杂度分析

```
算法 Partition(A[l..r])
//以第一个元素作为中轴，划分数组
//输入：数组A[l..r]，l,r为左右下标
//输出：A[l..r]的一个分区，返回划分点位置
p←A[l];
i←l; j←r+1;
reapeat
    reapeat i←i+1 until A[i] ≥p;
    reapeat j←j+1 until A[j] ≤p;
    swap(A[i],A[j]);
until i ≥ j
swap(A[i],A[j]);//i ≥ j,撤销最后一次交换
swap(A[l],A[j]);
return j;
```

```
算法  Quicksort(A[l...r])
//用Quicksort对子数组排序
//输入：数组A[0..n-1]中的子数组
//输出：非降序的子数组A[l...r]
if (l<r){
    s ← Partition(A[l...r]);
    Quicksort(A[l...s-1]);
    Quicksort(A[s+1...r]);
}
```

快速排序算法计算时间

$$T_{best}(n)=\begin{cases} 0 & n=1 \\ 2T_{best}(n/2)+n & n>1 \end{cases}$$

最优：分裂点在中间

$$T_{best}(n)=\Theta(n\log n)$$

$$T_{worst}(n)=(n+1)+n+(n-1)+\ldots+3=\Theta(n^2)$$

$$T_{avg}(n)=\begin{cases} 0 & n=0,1 \\ (\Sigma[(n+1)+T_{avg}(s)+T_{avg}(n-1-s)])/n & n>1 \end{cases}$$

思考：平均效率的递推式是如何得到的，如何计算？

$$T_{avg}(n) \approx 2n\ln n \approx 1.38n\log n$$

快速排序算法分析

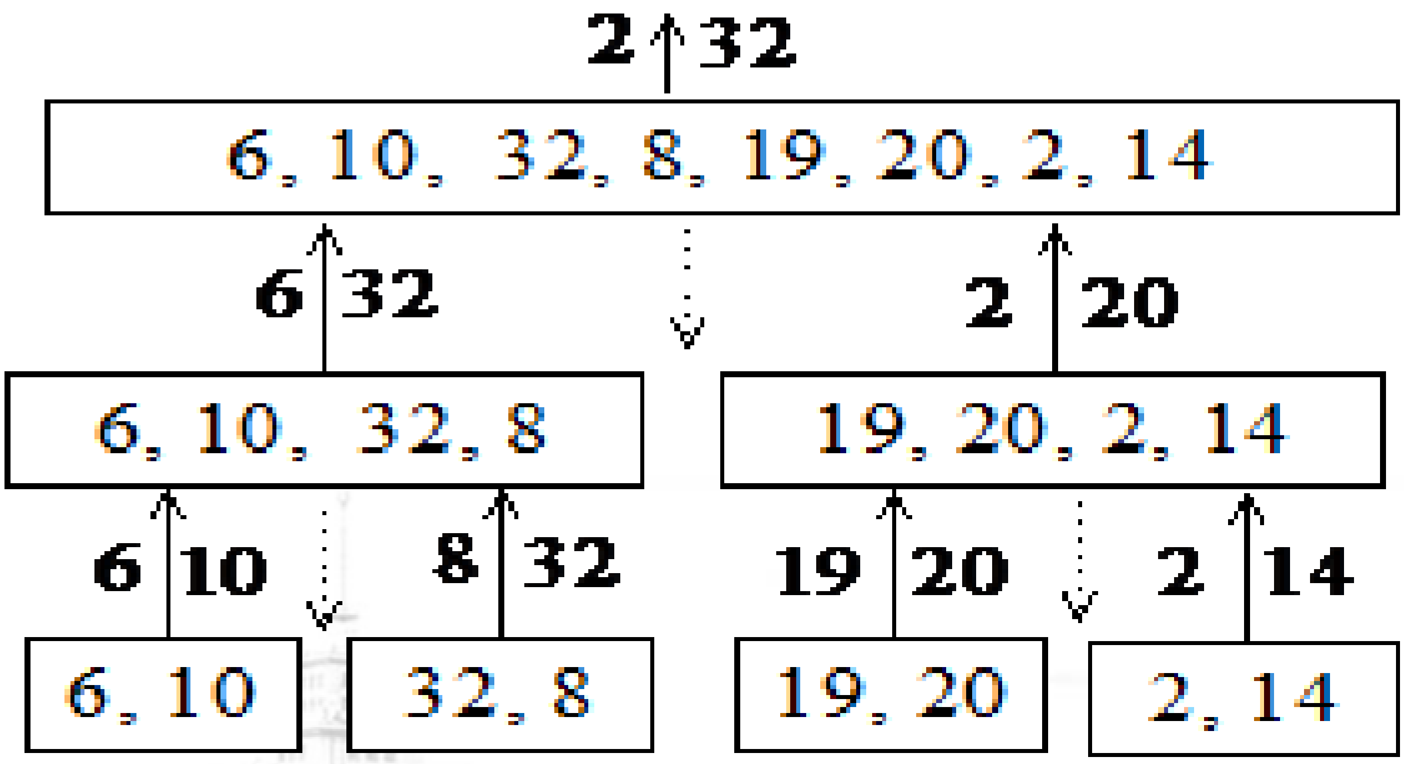
- 快速排序在平均情况下仅比最优情况多执行38％的比较操作。
- 它的最内循环效率非常高，在处理随机排列数组时，速度比合并排序快。
- 更好的划分元素选择方法：三平均分区
- 当子数组足够小时改用更简单的排序方法

最大值和最小值

输入：数组$A[1,\ldots,n]$
输出：A中的max和min

通常，直接扫描需要$2n-2$次比较操作
我们给出一个仅需$\lceil 3n/2-2 \rceil$次比较操作的算法。

最大值和最小值



2 32
6, 10, 32, 8, 19, 20, 2, 14
6 32
2 20
6, 10, 32, 8
19, 20, 2, 14
6 10
8 32
19 20
2 14
6, 10
32, 8
19, 20
2, 14

最大值和最小值

算法MaxMin(A)

输入: 数组$A[i,\ldots,j]$

输出:数组$A[i,\ldots,j]$中的max和min

1. If j-i+1 =1 Then 输出$A[i]$,$A[i]$,算法结束
2. If j-i+1 =2 Then
3. If $A[i]$< $A[j]$ Then输出$A[i]$,$A[j]$;算法结束
4. $k \leftarrow (j-i+1)/2$
5. $m_1,M_1 \leftarrow$MaxMin($A[i:k]$);
6. $m_2,M_2 \leftarrow$MaxMin($A[k+1:j]$);
7. $m \leftarrow \min(m_1,m_2)$;
8. $M \leftarrow \min(M_1,M_2)$;
9. 输出m,M

最大值和最小值

$T(1)=0$

$T(2)=1$

$$
\begin{aligned}
T(n)&=2T(n/2)+2\\
&=2^2T(n/2^2)+2^2+2\\
&= \ldots\\
&=2^{k-1}T(2)+2^{k-1}+2^{k-2}+\ldots+2^2+2 \qquad n=2^k\\
&=2^{k-1}+2^k-1\\
&=n/2+n-1\\
&=3n/2-1
\end{aligned}
$$

折半查找

- 问题描述
  - 已知一个按非降次序排列的元素表$a_1,a_2,\ldots,a_n$，判定某个给定元素x是否在该表中出现，若是，则找出该元素在表中的位置，并置于j，否则，置j为-1。
- 将问题表示为：$I=(n,a_1,\ldots,a_n,x)$
- 选取一个下标k(假设k是需要找的)，可得到三个子问题：
- $I_1=(k-1,a_1,\ldots,a_{k-1},x)$
- $I_2=(1,a_k,x)$
- $I_3=(n-k,a_{k+1},\ldots,a_n,x)$

如果对所求解的问题（或子问题）所选的下标k都是中间元素的下标，$k=[(n+1)/2]$，则由此产生的算法就是二分检索算法。

Example of binary search

Search the following array a for 36:

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
a	5	7	10	13	13	15	19	19	23	28	28	32	32	37	41	46

1. (0+15)/2=7; a[7]=19;
   too small; search 8..15

2. (8+15)/2=11; a[11]=32;
   too small; search 12..15

3. (12+15)/2=13; a[13]=37;
   too large; search 12..12

4. (12+12)/2=12; a[12]=32;
   too small; search 13..12

...but 13>12, so quit: 36 not found

折半查找算法

```
算法 BinarySearch(A[0..n-1],K)
//非递归折半查找
//输入：升序数组A[0..n-1]和查找键K
//输出：找到键K，返回K所在下标，否则返回-1
l←0;r←n-1;
while (l≤r) do{
    m←(int)(l+r)/2;
    if (K=A[m]) return m;
    else if (K<A[m]) r←m-1;
    else l←m+1;
}
return -1;
```

当n>1时，$T_w(n)=T_w([n/2])+1$，$T(1)=1$

$$
\begin{aligned}
T_w(n) &= T_w([n/2])+1 \\
&= T_w([n/4])+1+1 \\
&\dots \\
&= T_w(1)+1+\dots+1=1+k=1+\log_2 n
\end{aligned}
$$

折半查找结论

- 成功检索
  - 最好
  - 平均
  - 最坏
- 不成功检索

$\Theta(1)$

$\Theta(\log n)$

$\Theta(\log n)$

$$\Theta(\log n)$$

以比较为基础检索算法

- 有其他的以比较为基础检索的算法在最坏情况下比二分检索在计算时间上有更低的数量级吗？
- 比较为基础算法
  - 只允许元素间的比较，不允许对他们实施运算。
  - 二元比较来描述执行过程

以比较为基础检索的时间下界

- 定理2.3

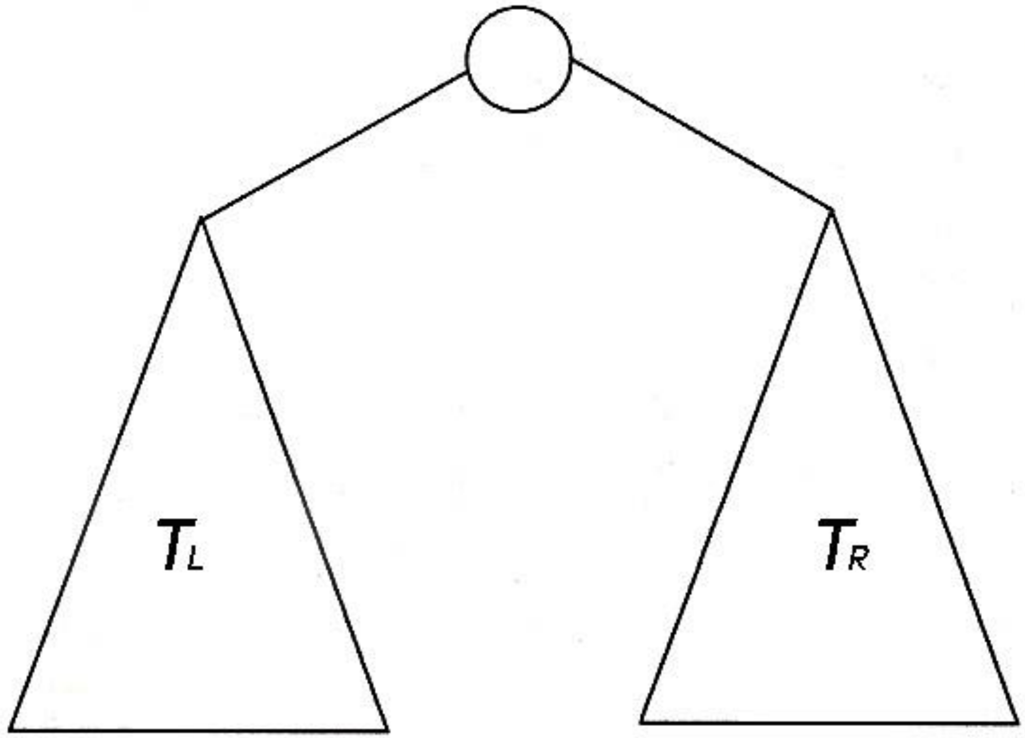
$$FIND(n) \geq \lceil \log(n+1) \rceil$$

1) 任何一个以比较为基础的检索算法的时间下界O(logn).

2) 二分检索是解决检索问题最优的最坏情况算法。

二叉树遍历及其相关特性

- 二叉树的标准定义
  - 若干个节点的有限集合，它要么为空，要么由一个根和两棵成功为T_L和T_R的不相交的二叉树构成，它们分别为根的左右子树。



二叉树遍历及其相关特性

二叉树的遍历

前序遍历（Preorder）
中序遍历（Inorder）
后序遍历（Postorder）

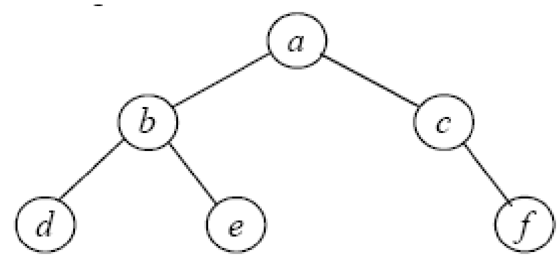
算法 Height(T)
//递归计算二叉树的高度
//输入：一棵二叉树
//输出：T的高度
If T = return -1
Else return max{Height(T_L),Height(T_R)}+1

递推关系式：

当n(T)>0时，A(n(T))=A(n(TL))+A(n(TR))+1

A(0)=0

其中，n(T)为二叉树节点数， A(n(T))为基本操作次数



a. Preorder: a b d e c f

b. Inorder: d b e a c f

c. Postorder: d e b f c a

大整数乘法

输入：n位二进制整数X和Y
输出：X和Y的乘积

通常，计算$X*Y$时间复杂性位$O(n^2)$，
我们给出一个复杂性为$O(n^{1.59})$的算法。

分治法求解大整数乘法

	n/2位	n/2位
X=	A	B

	n/2位	n/2位
Y=	C	D

$$XY = (A2^{n/2} + B)(C2^{n/2} + D)$$
$$= AC2^{n} + (AD+BC)2^{n/2} + BD$$

算法

1. 划分产生A,B,C,D;
2. 计算n/2位乘法AC、AD、BC、BD;
3. 计算AD+BC;
4. AC左移n位，(AD+BC)左移n/2位;
5. 计算XY。

时间复杂性

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

分治法求解大整数乘法

	n/2位	n/2位			n/2位	n/2位
X=	A	B		Y=	C	D

$$XY = (A2^{n/2} + B)(C2^{n/2} + D)$$
$$= AC2^{n} + (AD+BC)2^{n/2} + BD$$

$$AD+BC=(A+B)(C+D)-AC-BD$$

算法

1. 划分产生A,B,C,D;
2. 计算A-B和C-D;
3. 计算$n/2$位乘法AC、BD、(A+B)(C+D);
4. 计算(A+B)(C+D)-AC-BD;
5. AC左移n位，(A+B)(C+D)-AC-BD左移$n/2$位;
6. 计算XY

分治法求解大整数乘法

- 建立递归方程

$$T(n)=\theta(1) \qquad \text{if } n=1$$

$$T(n)=3T(n/2)+O(n) \qquad \text{if } n>1$$

- 使用Master定理

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

Strassen矩阵乘法

- 矩阵乘法是线性代数中最常见的运算之一，它在数值计算中有广泛的应用。

矩阵乘法

- 传统方法：$O(n^3)$
- 分治法:
  - 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:

由此可得:

$$C_{11}=A_{11}B_{11}+A_{12}B_{21} \quad (2)$$

$$C_{12}=A_{11}B_{12}+A_{12}B_{22} \quad (3)$$

$$C_{21}=A_{21}B_{11}+A_{22}B_{21} \quad (4)$$

$$C_{22}=A_{21}B_{12}+A_{22}B_{22} \quad (5)$$

矩阵乘法

如果n=2，则2个2阶方阵的乘积可以直接用(2)-(5)式计算出来，共需8次乘法和4次加法。当子矩阵的阶大于2时，为求2个子矩阵的积，可以继续将子矩阵分块，直到子矩阵的阶降为2。这样，就产生了一个分治降阶的递归算法。依此算法，计算2个n阶方阵的乘积转化为计算8个n/2阶方阵的乘积和4个n/2阶方阵的加法。2个n/2×n/2矩阵的加法显然可以在c*n^2/4时间内完成，这里c是一个常数。因此，上述分治法的计算时间耗费T(n)应该满足：

因此，上述分治法的计算时间耗费T(n)应该满足：

$$\begin{cases} T(2) = b \\ T(n) = 8T(n/2) + cn^2 & n > 2 \end{cases}$$

这个递归方程的解仍然是T(n)=*O*(n^3)。因此，该方法并不比用原始定义直接计算更有效。

究其原因，乃是由于式(2)-(5)并没有减少矩阵的乘法次数。而矩阵乘法耗费的时间要比矩阵加减法耗费的时间多得多。要想改进矩阵乘法的计算时间复杂性，必须减少子矩阵乘法运算的次数。

按照上述分治法的思想可以看出，要想减少乘法运算次数，关键在于计算2个2阶方阵的乘积时，能否用少于8次的乘法运算。

改进—分治算法

1969年，Strassen采用分治技术，将计算2个n阶矩阵乘积所需的计算时间改进到$O(n^{\log 7})=O(n^{2.8075})$。

Strassen算法

Strassen提出了一种新的算法来计算2个2阶方阵的乘积。他的算法只用了7次乘法运算，但增加了加、减法的运算次数。这7次乘法是:

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

做了这7次乘法后，
再做若干次加、减法就可以得到:
$C_{11}=M_5+M_4-M_2+M_6$
$C_{12}=M_1+M_2$
$C_{21}=M_3+M_4$
$C_{22}=M_5+M_1-M_3-M_7$

以上计算的正确性很容易验证。例如:

$C_{22}=M_5+M_1-M_3-M_7 \quad =(A_{11}+A_{22})(B_{11}+B_{22})+A_{11}(B_{12}-B_{22})-(A_{21}+A_{22})B_{11}-(A_{11}-A_{21})(B_{11}+B_{12}) \quad =A_{11}B_{11}+A_{11}B_{22}+A_{22}B_{11}+A_{22}B_{22}+A_{11}B_{12}$
$-A_{11}B_{22}-A_{21}B_{11}-A_{22}B_{11}-A_{11}B_{11}-A_{11}B_{12}+A_{21}B_{11}+A_{21}B_{12}$
$=A_{21}B_{12}+A_{22}B_{22}$ 由(2)式便知其正确性。

完整的Strassen算法

```
procedure STRASSEN(n,A,B,C);
begin
        if n=2 then MATRIX-MULTIPLY(A，B，C)
        else begin
                将矩阵A和B依(1)式分块;
                STRASSEN(n/2,A11,B12-B22,M1);
                STRASSEN(n/2,A11+A12,B22,M2);
                STRASSEN(n/2,A21+A22,B11,M3);
                STRASSEN(n/2,A22,B21-B11,M4);
                STRASSEN(n/2,A11+A22,B11+B22,M5);
                STRASSEN(n/2,A12-A22,B21+B22,M6);
                STRASSEN(n/2,A11- A21,B11+B12,M7)；
```

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

```
         end;
end;
```

Strassen矩阵乘法的计算时间复杂性

Strassen矩阵乘积分治算法中，用了7次对于n/2阶矩阵乘积的递归调用和18次n/2阶矩阵的加减运算。由此可知，该算法的所需的计算时间T(n)满足如下的递归方程:

$$
\begin{cases}
T(2) = b \\
T(n) = 7T(n/2) + an^2 & n > 2
\end{cases}
$$

Strassen矩阵乘法的计算时间复杂性

解：

$$
\begin{aligned}
T(n) &= an^2(1+7/2+(7/4)^2+\ldots+(7/4)^{k-1})+7^kT(1) \\
&\le cn^2(7/4)^{\log n} + 7^{\log n} \qquad (\because n=2^k, \therefore k=\log n) \\
&= cn^{\log 4+\log 7-\log 4} + n^{\log 7} \qquad (\because a^{\log x} = a^{\log a^{\log_a x}} \\
&= (c+1)n^{\log 7} \qquad\qquad = a^{\log_a x\cdot\log a} = x\log a) \\
&= O(n^{\log 7}) = O(n^{2.81})
\end{aligned}
$$

其解为**T(n)=*O*(n$^{\log 7}$)≈*O*(n$^{2.81}$)**。由此可见，**Strassen**矩阵乘法的计算时间复杂性比普通矩阵乘法有阶的改进。

Strassen矩阵乘法的计算时间复杂性

有人曾列举了计算2个2阶矩阵乘法的36种不同方法。但所有的方法都要做7次乘法。除非能找到一种计算2阶方阵乘积的算法，使乘法的计算次数少于7次，按上述思路才有可能进一步改进矩阵乘积的计算时间的上界。

但是Hopcroft和Kerr(197l)已经证明，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，就不能再寄希望于计算2×2矩阵的乘法次数的减少。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$O(n^{2.367})$。而目前所知道的矩阵乘法的最好下界仍是它的平凡下界$\Omega(n^2)$。

因此到目前为止还无法确切知道矩阵乘法的时间复杂性。关于这一研究课题还有许多工作可做。

线性时间选择

给定线性序集中n个元素和一个整数k，1≤k≤n，要求找出这n个元素中第k小的元素

```
template<class Type>
Type RandomizedSelect(Type a[],int p,int r,int k)
{
    if (p==r) return a[p];
    int i=RandomizedPartition(a,p,r),
    j=i-p+1;
    if (k<=j) return RandomizedSelect(a,p,i,k);
    else return RandomizedSelect(a,i+1,r,k-j);
}
```

在最坏情况下，算法**randomizedSelect**需要$O(n^2)$计算时间
但可以证明，算法**randomizedSelect**可以在O(n)平均时间内找出n个输入元素中的第k小元素。

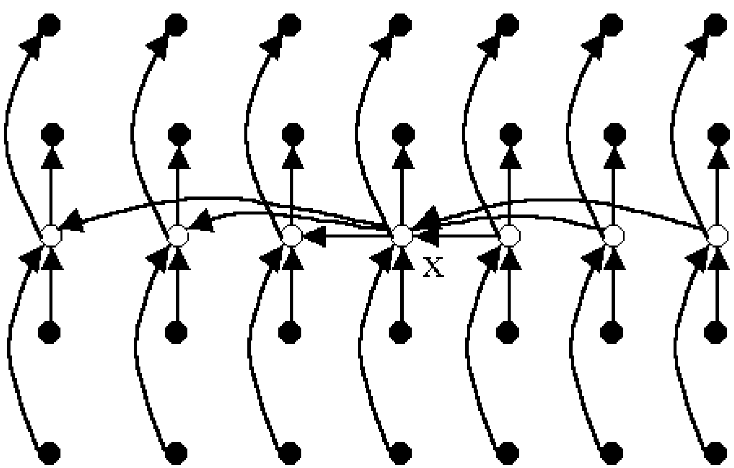
线性时间选择

如果能在线性时间内找到一个划分基准，使得按这个基准所划分出的2个子数组的长度都至少为原数组长度的ε倍(0<ε<1是某个正常数)，那么就可以**在最坏情况下**用O(n)时间完成选择任务。

例如，若ε=9/10，算法递归调用所产生的子数组的长度至少缩短1/10。所以，在最坏情况下，算法所需的计算时间T(n)满足递归式$T(n) \le T(9n/10)+O(n)$。由此可得$T(n)=O(n)$。

线性时间选择

- 将n个输入元素划分成⌈n/5⌉个组，每组5个元素，只可能有一个组不是5个元素。用任意一种排序算法，将每组中的元素排好序，并取出每组的中位数，共⌈n/5⌉个。
- 递归调用**select**来找出这⌈n/5⌉个元素的中位数。如果⌈n/5⌉是偶数，就找它的2个中位数中较大的一个。以这个元素作为划分基准。



设所有元素互不相同。在这种情况下，找出的基准x至少比3(n-5)/10个元素大，因为在每一组中有2个元素小于本组的中位数，而n/5个中位数中又有(n-5)/10个小于基准x。同理，基准x也至少比3(n-5)/10个元素小。而当n≥75时，3(n-5)/10≥n/4所以按此基准划分所得的2个子数组的长度都至少缩短1/4。

线性时间选择

```
Type Select(Type a[], int p, int r, int k)
{
    if (r-p<75) {
      用某个简单排序算法对数组a[p:r]排序
      ret
      };
    for (
      将
      与a
    //找中位数的中位数，r-p-4即上面所说的n-5
```

复杂度分析

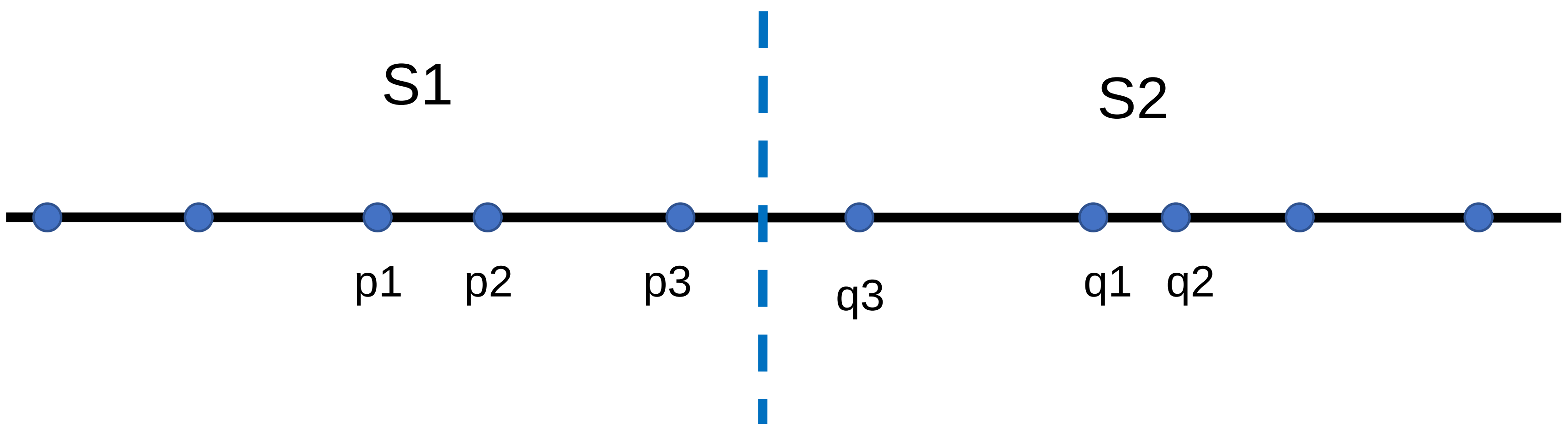
$$T(n) \le \begin{cases} C_1 & n < 75 \\ C_2 n + T(n/5) + T(3n/4) & n \ge 75 \end{cases}$$

T(n)=**O(n)**

上述算法将每一组的大小定为5，并选取75作为是否作递归调用的分界点。这2点保证了T(n)的递归式中2个自变量之和n/5+3n/4=19n/20=εn，0<ε<1。这是使T(n)=O(n)的关键之处。当然，除了5和75之外，还有其他选择。

```
}
```

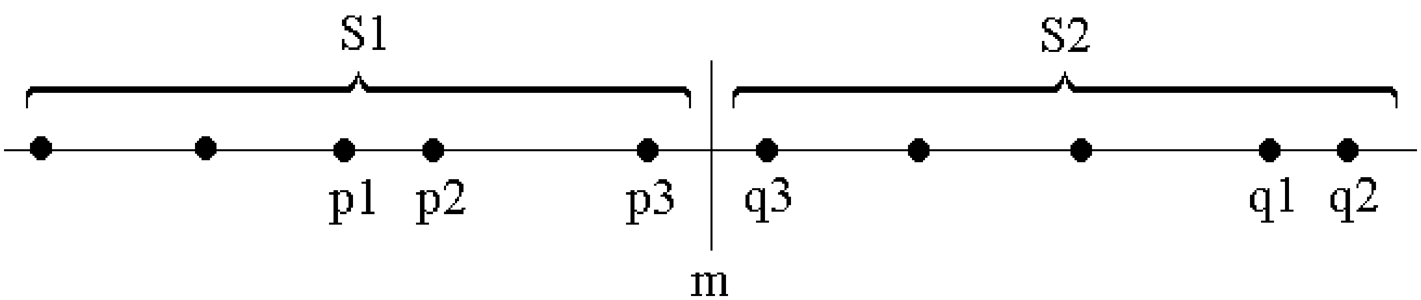
一维最近对问题



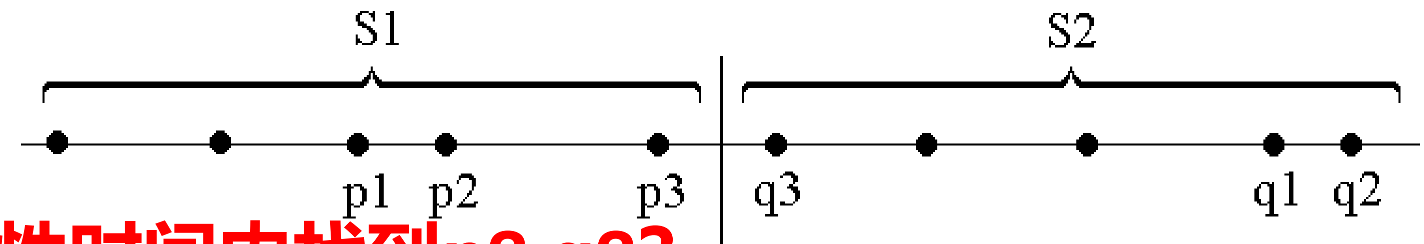
最接近点对问题

- 为了使问题易于理解和分析，先来考虑**一维**的情形。此时，S中的n个点退化为x轴上的n个实数 x1,x2,…,xn。最接近点对即为这n个实数中相差最小的2个实数。

> 假设我们用x轴上某个点m将S划分为2个子集S1和S2 ，基于**平衡子问题**的思想，用S中各点坐标的中位数来作分割点。
>
> 递归地在S1和S2上找出其最接近点对{p1,p2}和{q1,q2}，并设**d=min{|p1-p2|,|q1-q2|}**，S中的最接近点对或者是{p1,p2}，或者是{q1,q2}，或者是某个{p3,q3}，其中p3∈S1且q3∈S2。
>
> **能否在线性时间内找到p3,q3?**



最接近点对问题

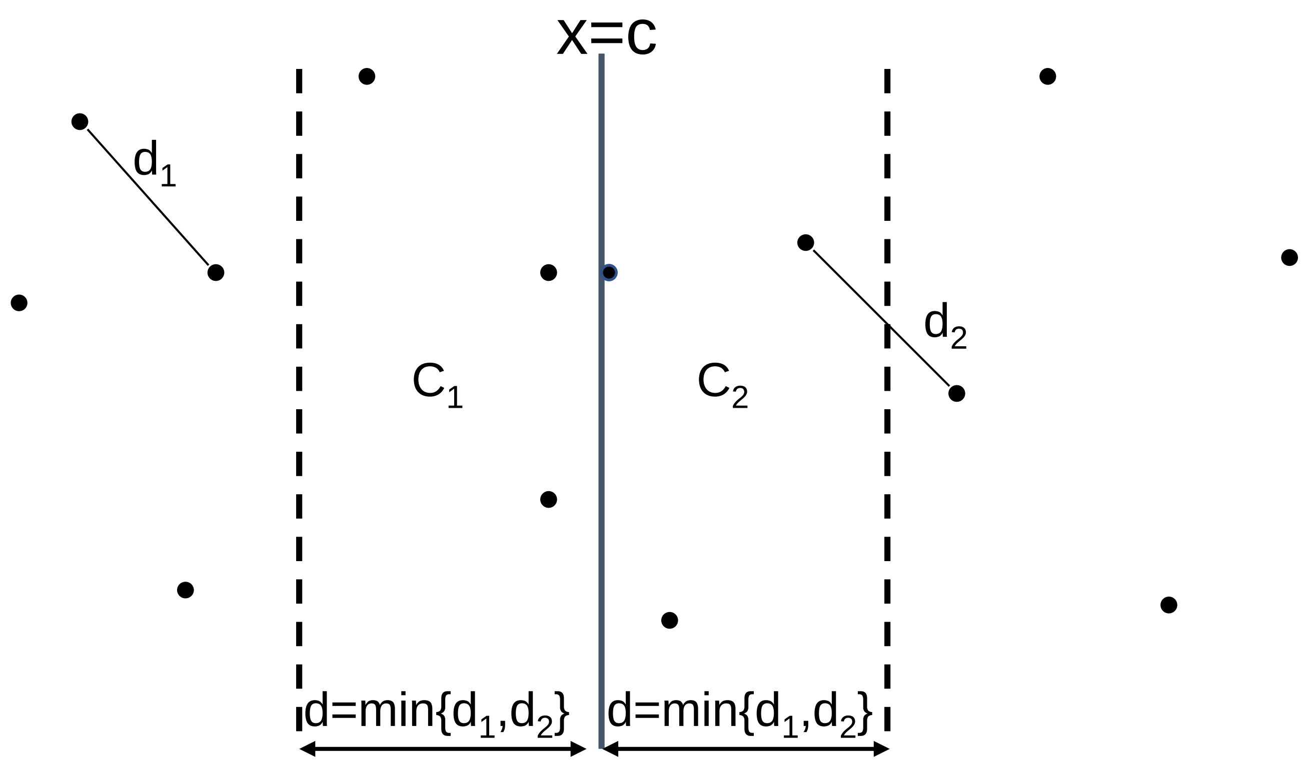


能否在线性时间内找到p3,q3?

- 如果S的最接近点对是{p3,q3}，即|p3-q3|<d，则p3和q3两者与m的距离不超过d，即**p3∈(m-d,m]**，**q3∈(m,m+d]**。
- 由于在S1中，每个长度为d的半闭区间至多包含一个点（否则必有两点距离小于d），并且m是S1和S2的分割点，因此(m-d,m]中至多包含S中的一个点。由图可以看出，**如果(m-d,m]中有S中的点，则此点就是S1中最大点。**
- 因此，我们用线性时间就能找到区间(m-d,m]和(m,m+d]中所有点，即p3和q3。**从而我们用线性时间就可以将S1的解和S2的解合并成为S的解。**

二维最近对问题

- $P_1(x_1,y_1),\ldots,P_n=(x_n,y_n)$是平面上n个点构成的集合S，假设$n=2^k$。



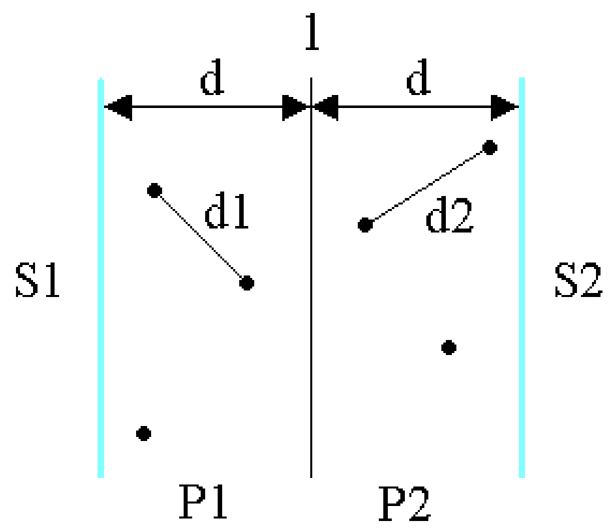
最接近点对问题

- 下面来考虑二维的情形。

➢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S1和S2。

➢递归地在S1和S2上找出其最小距离d1和d2，并设**d=min{d1,d2}**，S中的最接近点对或者是d，或者是某个{p,q}，其中p∈P1且q∈P2。

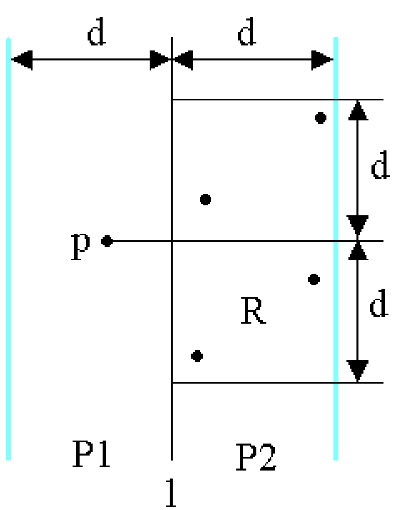
➢**能否在线性时间内找到p,q?**

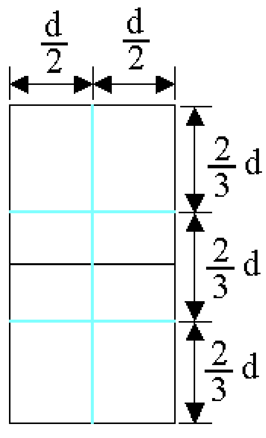


最接近点对问题

能否在线性时间内找到p3,q3?

- 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**
- 由d的意义可知，P2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。**
- 因此，在分治法的合并步骤中**最多只需要检查6×n/2=3n个候选者**





(a)

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

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

distance(u,v)<d。这与d的意义相矛盾。

最接近点对问题

- 为了确切地知道要检查哪6个点，可以将p和P2中所有S2的点投影到垂直线l上。由于能与p点一起构成最接近点对候选者的S2中点一定在矩形R中，所以它们在直线l上的投影点距p在l上投影点的距离小于d。由上面的分析可知，这种投影点最多只有6个。
- 因此，若将P1和P2中所有S中点按其y坐标排好序，则对P1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P1中每一点最多只要检查P2中排好序的相继6个点。

最接近点对问题

double **cpair2**(S)

{

n=|S|;

if (n <

1、m=S中

构造S

//S1={p∈S|x(p)<m};

S2={p∈S|x(p)>m}

2、d1=**cpair2**(S1);

d2=**cpair2**(S2);

3、dm=**min**(d1,d2);

4、设P1是S1中距垂直分割线l的距离在dm之内的所有点组成的集合；

P2是S2中距分割线l的距离在dm之内所有

中与

完成

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2dm的区间内移动；

设dl是按这种扫描方式找到的点对间的最小距离；

6、d=**min**(dm,dl);

return d;

}

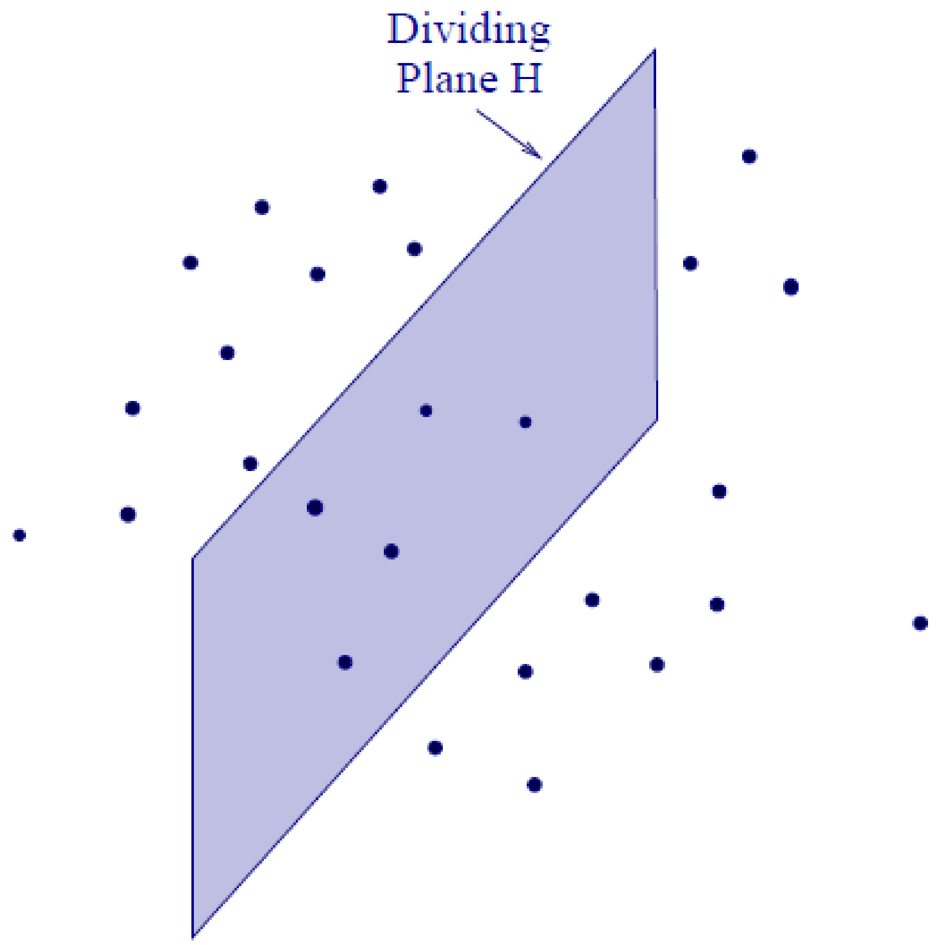
复杂度分析

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

T(n)=O(nlogn)

三维最近点对

- 怎么办？



循环赛日程表

- 设有$n=2^k$个运动员要进行网球循环赛，设计一个满足以下要求的比赛日程表：
  - 每个选手必须与其他n-1个选手各赛一次；
  - 每个选手一天只能赛一次；
  - 循环赛一共进行n-1天。
- 分治
  - 将选手分为两组，n个选手的日程表就可通过n/2个选手设计的日程表来决定。

8个选手的比赛日程表

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

分治法小结

- 分治法是一种一般性的算法设计技术，他将问题的实例划分为若干个较小的实例(最好用有相同的规模)，对这些小的问题求解，然后合并这些解，得到原是问题的解。
- 分治法的时间效率满足：$T(n)=aT(n/b)+f(n)$
- 合并排序是一种分治排序算法，任何情况下，该算法的时间效率都是$\Theta(n\log n)$，它的键值比较次数非常接近理论的最小值，取点是需要大量的额外存储空间。
- 快速排序也是一种分治排序算法，具有出众的时间效率nlogn，最差效率是平方级的。
- 折半查找是一种对有序数组进行查找的算法，效率为logn
- n位大整数乘法的分治算法，大约需要做$n^{1.585}$次乘法。

分治法的基本步骤

分治法在每一层递归上都有三个步骤：

1. 分解：将原问题分解为若干个规模较小，相互独立，与原问题形式相同的子问题；

2. 解决：若子问题规模较小而容易被解决则直接解，否则递归地解各个子问题；

3. 合并：将各个子问题的解合并为原问题的解。

分治法的合并步骤是算法的关键所在

- 有些问题的合并方法比较明显，如二分查找　快速排序
- 有些问题合并方法比较复杂，或者是有多种合并方案，如_大整数乘法　Strassen矩阵乘法
- 或者是合并方案不明显，如最接近点对问题。

作业：快速傅里叶变换

问题定义

输入：$a_0,a_1,\ldots,a_{n-1}$， $n=2^k$, a_i是实数 , $(0\le i\le n-1)$
输出：$A_0,A_1,\ldots,A_{n-1}$

$A_j=\sum_{k=0}^{n} a_k e^{\frac{2jk\pi}{n}i}$,其中$j=0,1,,\ldots,n-1$,
e是自然对数的底数，i是虚数单位

蛮力法利用定义计算每个A_j，时间复杂度为$\Theta(n^2)$

- 分治算法 $T(n)=\Theta(n \log n)$

其它例子[找出伪币]

例 [找出伪币] 给你一个装有1 6个硬币的袋子。1 6个硬币中有一个是伪造的，并且那个伪造的硬币比真的硬币要轻一些。你的任务是找出这个伪造的硬币。为了帮助你完成这一任务，将提供一台可用来比较两组硬币重量的仪器，利用这台仪器，可以知道两组硬币的重量是否相同。

比较硬币1与硬币2的重量。假如硬币1比硬币2轻，则硬币1是伪造的；假如硬币2比硬币1轻，则硬币2是伪造的。这样就完成了任务。假如两硬币重量相等，则比较硬币3和硬币4。同样，假如有一个硬币轻一些，则寻找伪币的任务完成。假如两硬币重量相等，则继续比较硬币5和硬币6。按照这种方式，可以最多通过8次比较来判断伪币的存在并找出这一伪币。

另外一种方法就是利用分而治之方法。假如把1 6硬币的例子看成一个大的问题。第一步，把这一问题分成两个小问题。随机选择8个硬币作为第一组称为A组，剩下的8个硬币作为第二组称为B组。这样，就把1 6个硬币的问题分成两个8硬币的问题来解决。第二步，判断A和B组中是否有伪币。可以利用仪器来比较A组硬币和B组硬币的重量。假如两组硬币重量相等，则可以判断伪币不存在。假如两组硬币重量不相等，则存在伪币，并且可以判断它位于较轻的那一组硬币中。最后，在第三步中，用第二步的结果得出原先1 6个硬币问题的答案。若仅仅判断硬币是否存在，则第三步非常简单。无论A组还是B组中有伪币，都可以推断这1 6个硬币中存在伪币。因此，仅仅通过一次重量的比较，就可以判断伪币是否存在。

现在假设需要识别出这一伪币。把两个或三个硬币的情况作为不可再分的小问题。注意如果只有一个硬币，那么不能判断出它是否就是伪币。在一个小问题中，通过将一个硬币分别与其他两个硬币比较，最多比较两次就可以找到伪币。这样，**1 6**硬币的问题就被分为两个**8**硬币（**A**组和**B**组）的问题。通过比较这两组硬币的重量，可以判断伪币是否存在。如果没有伪币，则算法终止。否则，继续划分这两组硬币来寻找伪币。假设**B**是轻的那一组，因此再把它分成两组，每组有**4**个硬币。称其中一组为**B1**，另一组为**B2**。比较这两组，肯定有一组轻一些。如果**B1**轻，则伪币在**B1**中，再将**B1**又分成两组，每组有两个硬币，称其中一组为**B1a**，另一组为**B1b**。比较这两组，可以得到一个较轻的组。由于这个组只有两个硬币，因此不必再细分。比较组中两个硬币的重量，可以立即知道哪一个硬币轻一些。较轻的硬币就是所要找的伪币。

[金块问题]

例 **[金块问题]** 有一个老板有一袋金块。每个月将有两名雇员会因其优异的表现分别被奖励一个金块。按规矩，排名第一的雇员将得到袋中最重的金块，排名第二的雇员将得到袋中最轻的金块。根据这种方式，除非有新的金块加入袋中，否则第一名雇员所得到的金块总是比第二名雇员所得到的金块重。如果有新的金块周期性的加入袋中，则每个月都必须找出最轻和最重的金块。假设有一台比较重量的仪器，我们希望用最少的比较次数找出最轻和最重的金块。

直接查找法

假设袋中有**n** 个金块。可以通过**n-1**次比较找到最重的金块。找到最重的金块后，可以从余下的**n-1**个金块中用类似的方法通过**n-2**次比较找出最轻的金块。这样，比较的总次数为**2n-3**。

分而治之方法

下面用分而治之方法对这个问题进行求解。当**n**很小时，比如说，**n≤2**，识别出最重和最轻的金块，一次比较就足够了。当**n** 较大时（**n**＞**2**），第一步，把这袋金块平分成两个小袋**A**和**B**。第二步，分别找出在**A**和**B**中最重和最轻的金块。设**A**中最重和最轻的金块分别为**HA** 与**LA**，以此类推，**B**中最重和最轻的金块分别为**HB** 和**LB**。第三步，通过比较**HA** 和**HB**，可以找到所有金块中最重的；通过比较**LA** 和**LB**，可以找到所有金块中最轻的。在第二步中，若**n**＞**2**，则递归地应用分而治之方法。

假设n= 8。这个袋子被平分为各有4个金块的两个袋子A和B。为了在A中找出最重和最轻的金块，A中的4个金块被分成两组A1和A2。每一组有两个金块，可以用一次比较在A中找出较重的金块HA1和较轻的金块LA1。经过另外一次比较，又能找出HA 2和LA 2。现在通过比较HA1和HA2，能找出HA；通过LA 1和LA2的比较找出LA。这样，通过4次比较可以找到HA 和LA。同样需要另外4次比较来确定HB 和LB。通过比较HA 和HB（LA 和LB），就能找出所有金块中最重和最轻的。因此，当n= 8时，这种分而治之的方法需要1 0次比较。设c(n)为使用分而治之方法所需要的比较次数。为了简便，假设n是2的幂。当n= 2时，c(n) = 1。对于较大的n，c(n) = 2c(n/ 2) + 2。当n是2的幂时，使用迭代方法可知

c(n) = 3n/ 2 - 2。在本例中，使用分而治之方法比逐个比较的方法少用了2 5％的比较次数。