USTC

Chapter 2
Divide and Conquer

王子磊（Zilei Wang）
Email: zlwang@ustc.edu.cn
http://vim.ustc.edu.cn/

学习要点

- 理解递归的概念
- 掌握设计有效算法的分治策略
- 通过下面的范例学习分治策略设计技巧

  （1）二分搜索技术

  （2）大整数乘

  （3）Strassen 矩阵乘法

  （4）棋盘覆盖

  （5）合并排序和快速排序

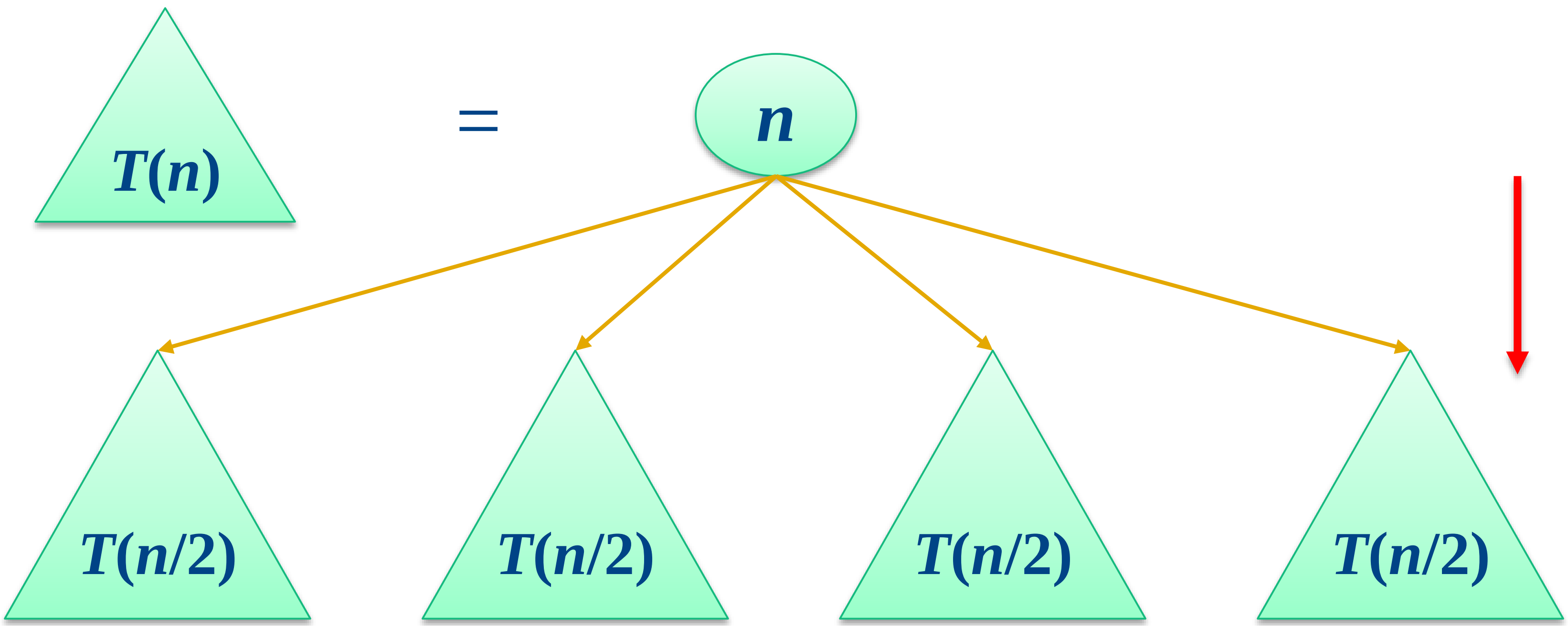
  （6）线性时间选择

  （7）最接近点对问题

  （8）循环赛日程表

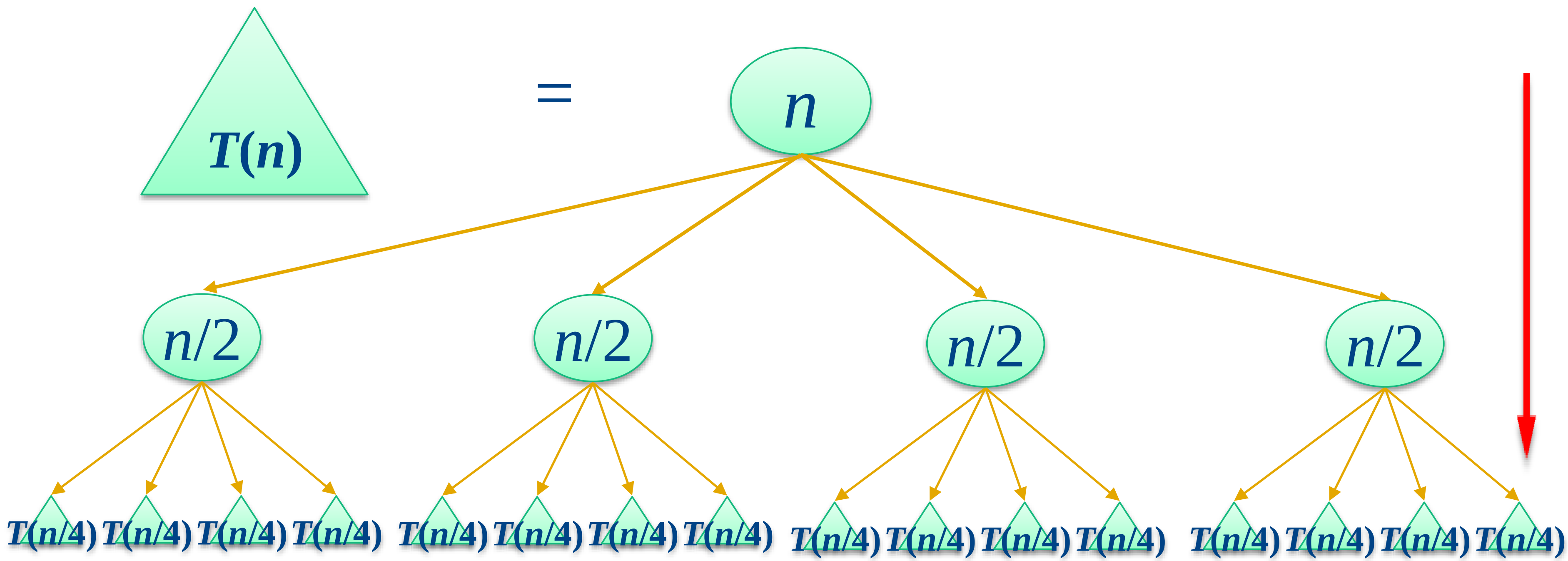
算法总体思想

- 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



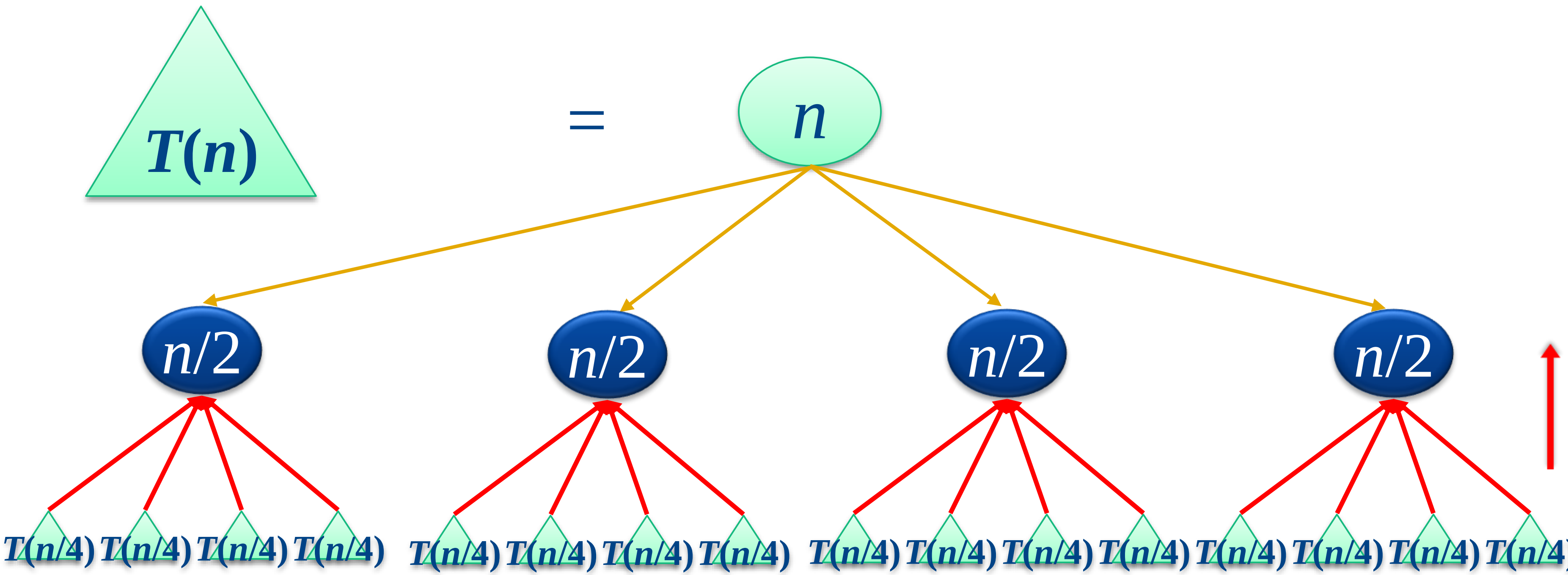
算法总体思想

- 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

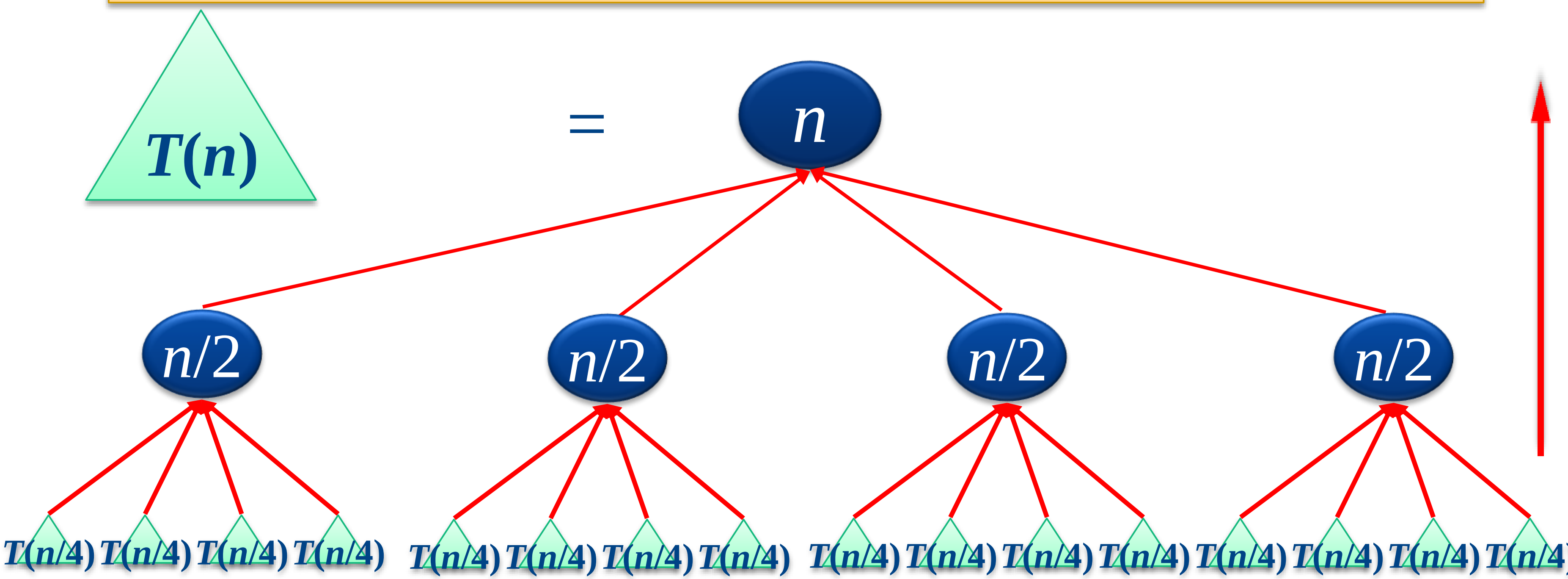
- 将求出的小规模的问题的解合并为一个更大规模的问题的解，自底向上逐步求出原来问题的解



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

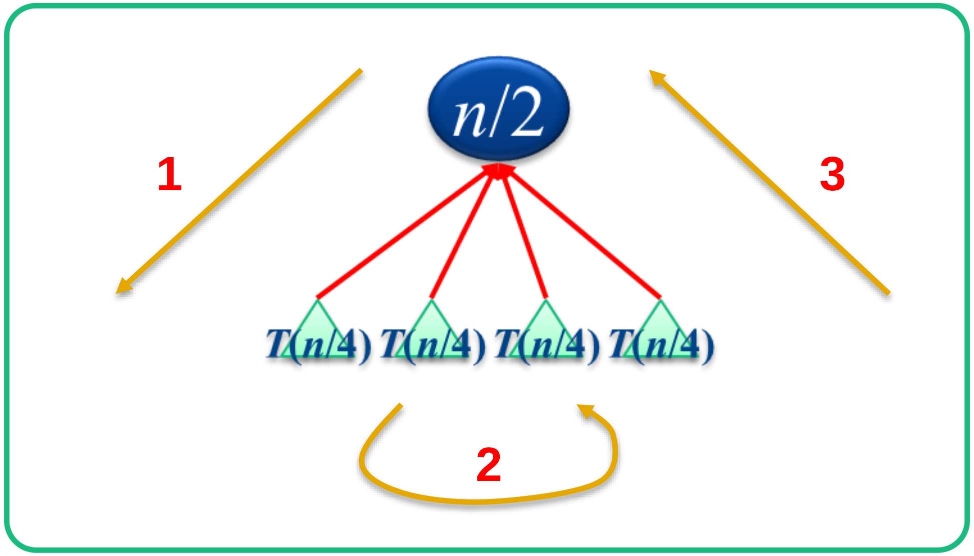


递归的概念

- 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**
- 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
  - 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

- 分治的基本过程
  - Divide：将问题规模变小
  - Conquer：递归的处理小规模问题
  - Combine：将小规模问题的解合并为原始问题的解



递归的概念

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

递归的概念

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);
}
```

Fibonacci 实现

- 非递归调用
  - 自底向上
    - $T(n) = T(n-1) + 1 = \Theta(n)$
  - 幂运算求解
    - $\begin{bmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^{n+1}$
      - proof by induction
    - 幂运算
      - $y = x^n$
      - $T(n) = T\left(\frac{n}{2}\right) + \Theta(1) = \Theta(\log n)$

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：

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

例3 Ackerman函数

前 2 例中的函数都可以找到相应的非递归方式定义：

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

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

本例中的Ackerman函数却无法找到非递归的定义!

递归的概念

例3 Ackerman函数

- $A(n, m)$ 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，$A(n,1)=A(A(n\text{-}1,1),0)=A(n\text{-}1,1)+2$，和$A(1,1)=2$
    故：$A(n, 1)=2*n$
  - m=2时，$A(n,2)=A(A(n\text{-}1,2),1)=2A(\text{n-}1,2)$，和
    $A(1,2)=A(A(0,2),1)=A(1,1)=2$，故：$A(n, 2)= 2\wedge n$
  - m=3时，类似的可以推出 $\underbrace{2^{2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}}}_{n}$
  - 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

递归的概念

例3 Ackerman函数

- 定义单变量的 Ackerman 函数 $A(n)$ 为，$A(n)=A(n，n)$
- 定义其逆函数 $\alpha(n)$ 为：$\alpha(n)=\min\{k \mid A(k)\geq n\}$，即 $\alpha(n)$ 是使 $n\leq A(k)$ 成立的最小的 k 值
- $\alpha(n)$ 在复杂度分析中常遇到
  - 对于通常所见到的正整数n，有 $\alpha(n)\leq 4$
  - 但在理论上 $\alpha(n)$ 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

例4 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 $\mathrm{perm}(X)$; $(r_i)\mathrm{perm}(X)$ 表示在全排列 $\mathrm{perm}(X)$ 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，$\mathrm{perm}(R)=(r)$，其中 r 是集合 R 中唯一的元素
当 $n>1$ 时，$\mathrm{perm}(R)$ 由 $(r_1)\mathrm{perm}(R_1)$，$(r_2)\mathrm{perm}(R_2)$，…，$(r_n)\mathrm{perm}(R_n)$ 构成

递归的概念

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 $n_1 \geq n_2 \geq \ldots \geq n_k \geq 1$，$k \geq 1$

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

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：

(1) $q(n,1)=1, n\geq 1$;

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 $$n=\overbrace{1+1+\cdots+1}^{n}$$

(2) $q(n,m)=q(n,n), m\geq n$;

最大加数 n_1 实际上不能大于 n，因此，$q(1,m)=1$

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：

(3) $q(n,n)=1+q(n,n-1)$;

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

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：

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

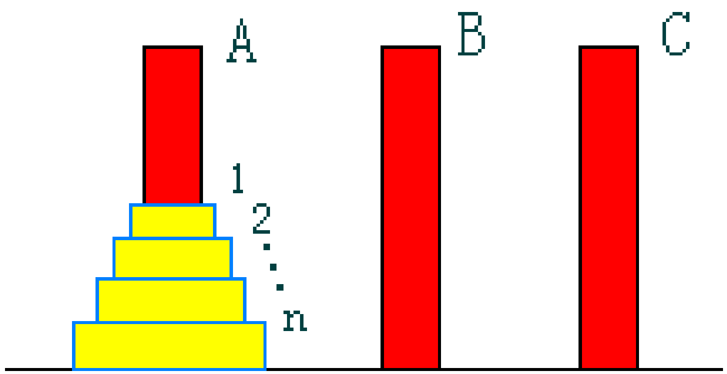
正整数 n 的划分数 $p(n)=q(n,n)$

递归的概念

例6 Hanoi塔问题

设 a,b,c 是3个塔座。开始时，在塔座 a 上有一叠共 n 个圆盘，这些圆盘自下而上，由大到小地叠在一起；各圆盘从小到大编号为 $1,2,\ldots,n$，现要求将塔座 a 上的这一叠圆盘移到塔座 b 上，并仍按同样顺序叠置。在移动圆盘时应遵守以下移动规则：

中任一塔座上



规则1：每次只能移动1个圆盘

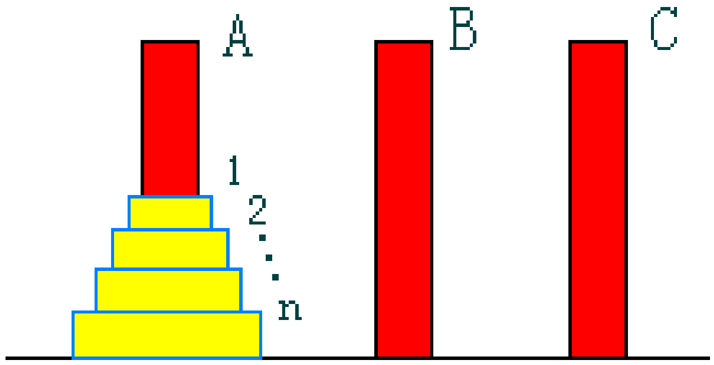
规则2：任何时刻都不允许将较大的圆盘压在较小的圆盘之上

规则3：在满足移动规则1和2的前提下，可将圆盘移至 a,b,c 中任一塔座上

递归的概念

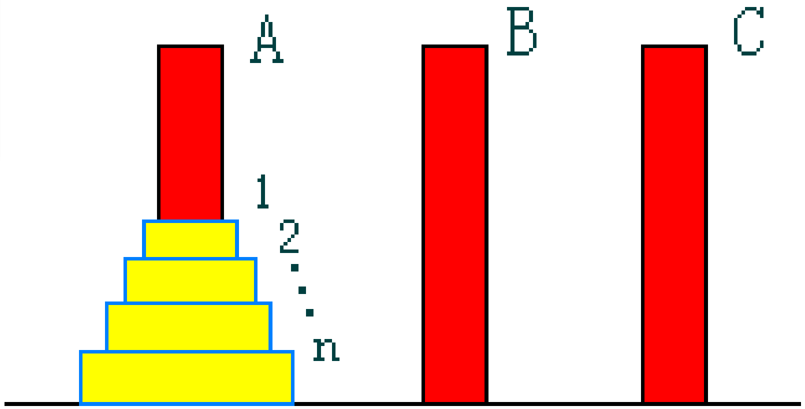
例6 Hanoi塔问题

- 当 $n=1$ 时，问题比较简单，此时，只要将编号为 1 的圆盘从塔座 a 直接移至塔座 b 上即可
- 当 $n>1$ 时，需要利用塔座 c 作为辅助塔座，此时若能设法将 n-1 个较小的圆盘依照移动规则从塔座 a 移至塔座 c，然后，将剩下的最大圆盘从塔座 a 移至塔座 b，最后，再设法将 n-1 个较小的圆盘依照移动规则从塔座 c 移至塔座 b
- 由此可见，n 个圆盘的移动问题可分为 2 次 n-1 个圆盘的移动问题，这又可以递归地用上述方法来做，由此可以设计出解 Hanoi 塔问题的递归算法



例6 Hanoi塔问题

```
void hanoi(int n, int a, int b, int c)
  {
      if (n > 0)
      {
        hanoi(n-1, a, c, b);
        move(a,b);
        hanoi(n-1, c, b, a);
      }
  }
```



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、用迭代来实现递归函数

后一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 sub-instances 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

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divide原问题和用 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：

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

应用主定理求得方程的解

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
- 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： 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$x>a[mid]$，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

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

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

据此容易设计出**二分搜索算法**：

```
template<class Type>
int BinarySearch(Type a[], const Type& x, int l, int r)
{
    while (r >= l){
        int m = (l+r)/2;
        if (x == a[m])  return m;
        if (x < a[m])  r = m-1; else l = m+1;
    }
    return -1;
}
```

$$T(n) = 2T\left(\frac{n}{2}\right) + \Theta(1)$$

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(log$n$)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(log$n$)

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

X

Y

复杂度分析

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

$T(n)=O(n^2)$ ✘没有改进

$$X = a\,2^{n/2} + b \qquad Y = c\,2^{n/2} + d$$

$$XY = ac\,2^{n} + (ad+bc)\,2^{n/2} + bd$$

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：

X

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

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

➢这个思想产生了**快速傅利叶变换** (Fast Fourier Transform)
   ➢该方法也可以看作是一个复杂的分治算法

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

- 传统方法：O(n^3)

- 分治法：

使用与上例类似的技术，将矩阵A，B和C中每一矩阵都分块成

复杂度分析

$$T(n) = \begin{cases} O(1) & n = 2 \\ 8T(n/2) + O(n^2) & n > 2 \end{cases}$$

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

由此

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

Strassen矩阵乘法

- 传统方法：O(n³)
- 分治法：

为了降低时间复杂度，必须减少乘法的次数

复杂度分析

$$T(n) = \begin{cases} O(1) & n = 2 \\ 7T(n/2) + O(n^2) & n > 2 \end{cases}$$

$T(n)=O(n^{\log 7}) = O(n^{2.81})$ ✓ 较大的改进

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

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

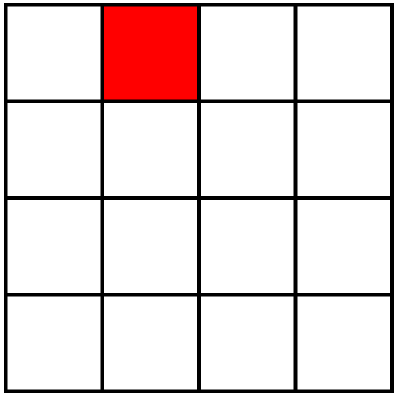
Strassen矩阵乘法

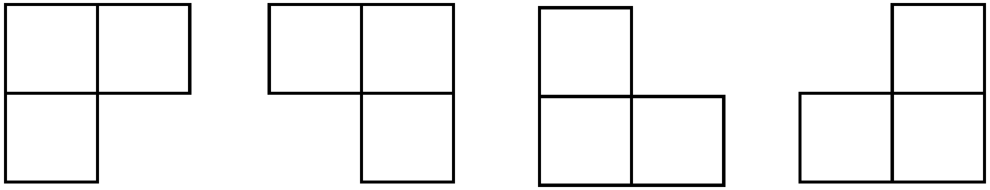
- 传统方法：O(n^3)
- 分治法: O($n^{2.81}$)
- 更快的方法??

- 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($n^{2.376}$)**
- 是否能找到 O(n^2) 的算法？

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

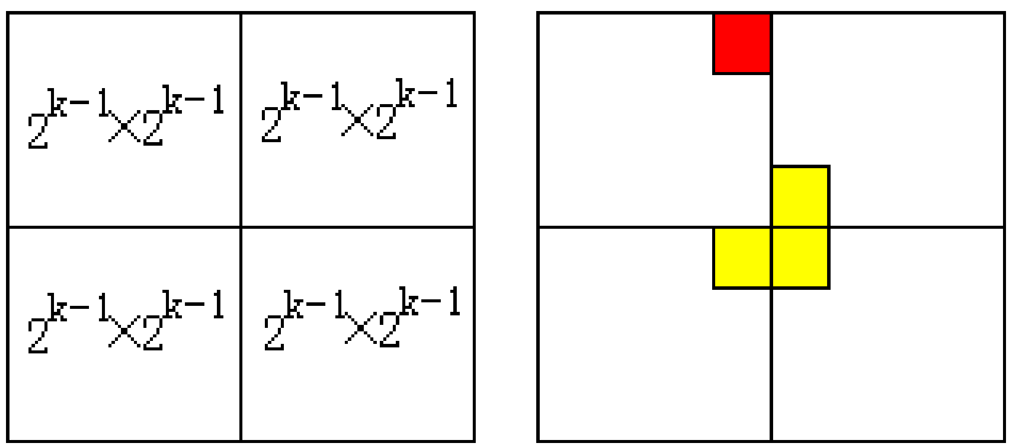




棋盘覆盖

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。



(a) (b)

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
    if (dr <
      // 特
      chess
    else {//
      // 用
      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

      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

合并排序

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

```
vo
  {




        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
     }
}
```

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($n$log$n$) 渐进意义下的最优算法

合并排序

算法 mergeSort 的递归过程可以消去

初始序列 [49] [38] [65] [97] [76] [13] [27]

第一步 [38 49] [65 97] [13 76] [27]

第二步 [38 49 65 97] [13 27 76]

第三步 [13 27 38 49 65 76 97]

合并排序

- 最坏时间复杂度：O($n$log$n$)
- 平均时间复杂度：O($n$log$n$)
- 辅助空间：O(n)

快速排序

在快速排序中，记录的比较和交换是从两端向中间进行的，关键字较大的记录一次就能交换到后面单元，关键字较小的记录一次就能交换到前面单元，记录每次移动的距离较大，因而总的比较和移动次数较少

$\leq x$	x	$\geq x$

```
template<class Type>
void QuickSort (Type a[], int p, int r)
{
      if (p<r) {
           int q=Partition(a,p,r);
           QuickSort (a,p,q-1); //对左半段排序
           QuickSort (a,q+1,r); //对右半段排序
      }
}
```

快速排序

```
template<class Type>
int Partition (Type a[], int p, int r)
{
        int i = p, j = r + 1;
        Type x=a[p];
        // 将< x的元素交换到左边区域
        // 将> x的元素交换到右边区域
        while (true) {
          while (a[++i] <x && i<=r);
          while (a[- -j] >x);
          if (i >= j) break;
          Swap(a[i], a[j]);
        }
      a[p] = a[j];
      a[j] = x;
      return j;
}
```

x	$\leq x$	?	$\geq x$

p i j r

$\{6, 7, 5, 2, \overline{5}, 8\}$ 初始序列

$\{6, 7, 5, 2, \overline{5}, 8\}$ j--;

↑ i ↑ j

$\{6, 7, 5, 2, \overline{5}, 8\}$ i++;

↑ i ↑ j

$\{6, \overline{5}, 5, 2, 7, 8\}$ j--;

↑ i ↑ j

$\{6, \overline{5}, 5, 2, 7, 8\}$ i++;

↑ j ↑ i

$\{2, \overline{5}, 5\}$ 6 $\{7, 8\}$ 完成

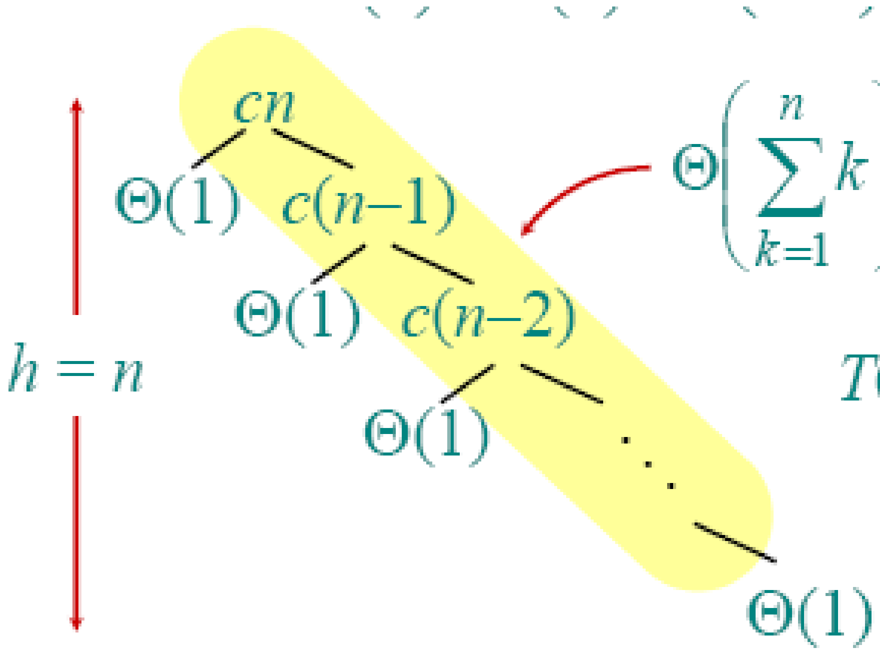
快速排序

- 最坏时间复杂度：$O(n^2)$
- 平均时间复杂度：$O(n\log n)$

- 算法分析
  - 假定输入的所有元素都不相同
    - 实际上，当输入元素有重复时，可以有更好的划分方法
  - $\boldsymbol{T(n)}$ = worst-case running time on an array of $\boldsymbol{n}$ elements.

最坏情况下的递归树

$T(n) = T(0) + T(n\text{-}1) + cn$



$$\Theta\left(\sum_{k=1}^{n} k\right) = \Theta(n^2)$$

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

最好情况下的直观分析

PARTITION 等分的分割数组（2的幂次）

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

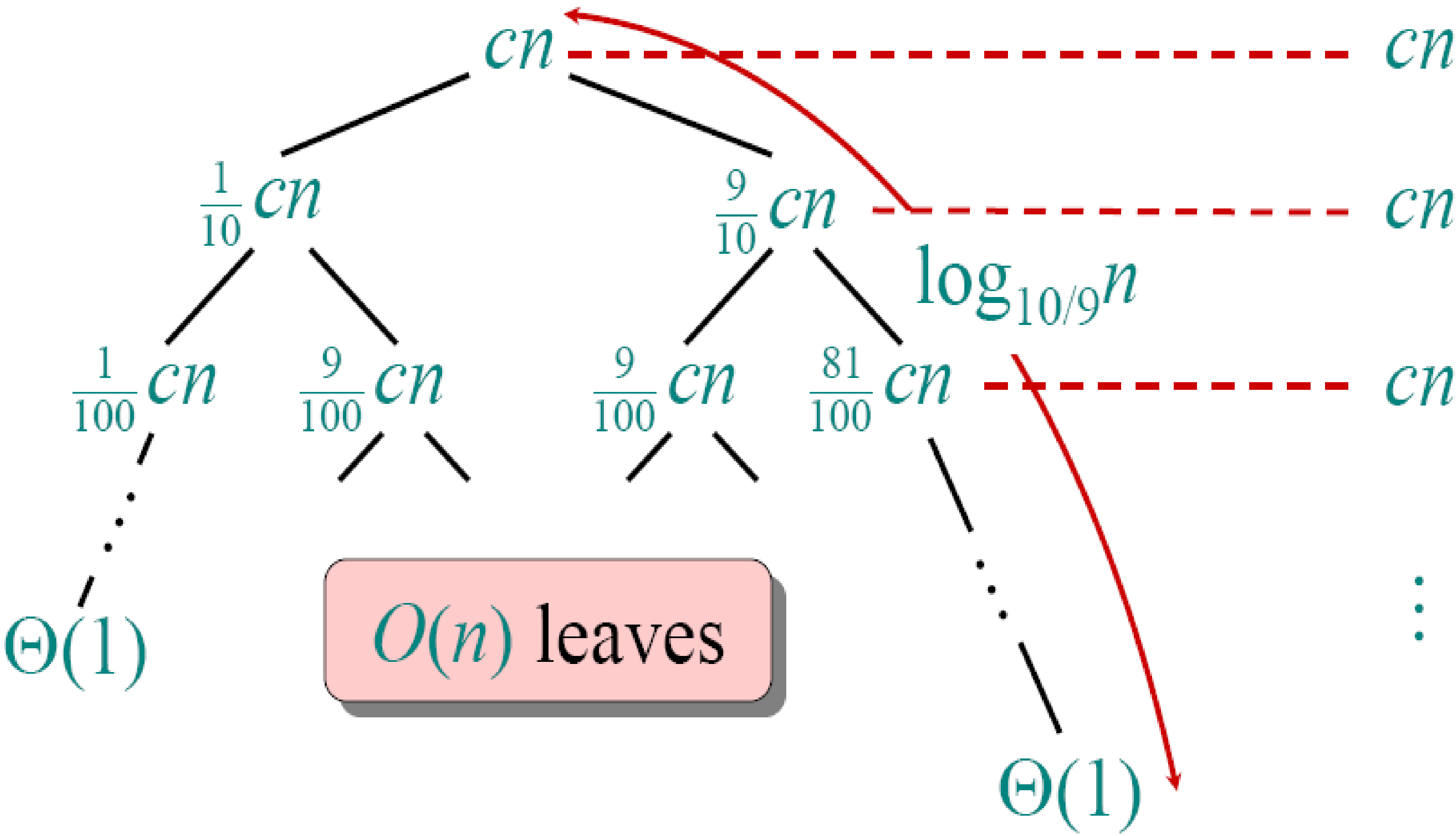
$$= \Theta(n \lg n)$$ （与归并排序一样）

如果按1/10:9/10的比例划分呢?

$$T(n) = T(1/10n) + T(9/10n)+\Theta(n)$$

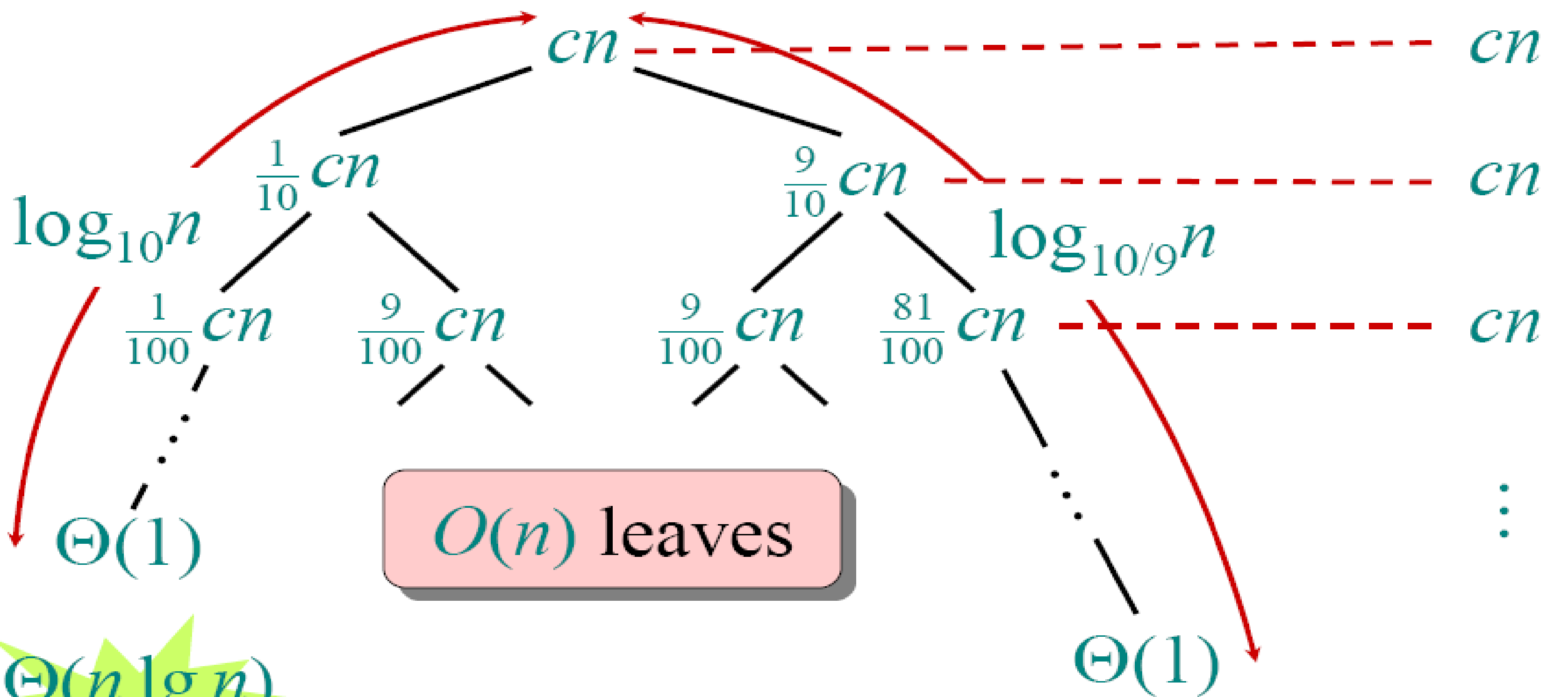
这时候的递归解是什么？

较好情况下的递归树



cn
cn
$\frac{1}{10}cn$
$\frac{9}{10}cn$
cn
$\log_{10/9}n$
$\frac{1}{100}cn$
$\frac{9}{100}cn$
$\frac{9}{100}cn$
$\frac{81}{100}cn$
cn
$\Theta(1)$
$O(n)$ leaves
$\Theta(1)$

较好情况下的递归树



$\Theta(n \lg n)$

Lucky!

$$cn \log_{10} n \le T(n) \le cn \log_{10/9} n + O(n)$$

更进一步

- 如果在下述两种情况下交替出现 (lucky, unlucky, lucky, ….

$$L(n) = 2U(n/2) + \Theta(n) \qquad \text{lucky}$$

$$U(n) = L(n-1) + \Theta(n) \qquad \text{unlucky}$$

- 求解：

$$\begin{aligned} L(n) &= 2(L(n/2-1) + \Theta(n/2)) + \Theta(n) \\ &= 2L(n/2-1) + \Theta(n) \\ &= \Theta(n \lg n) \end{aligned}$$

Lucky !

- 问题：
  - 如果确保经常会出现lucky的情况？

随机快速排序

快速排序算法的性能取决于划分的对称性

通过修改算法 **partition**，可以设计出采用随机选择策略的快速排序算法

在快速排序算法的每一步中，当数组还没有被划分时，可以在$a[p:r]$中随机选出一个元素作为划分基准，这样可以使划分基准的选择是随机的，从而可以期望划分是较对称的

```
template<class Type>
int RandomizedPartition (Type a[], int p, int r)
{
        int i = Random(p,r);
        Swap(a[i], a[p]);
        return Partition (a, p, r);
}
```

随机快排分析

- $T(n)$ 为一个随机变量，表示在输入大小为 n 时的随机快排运行时间
  - 假定随机数是独立的
- 对 $k = 0, 1, \ldots, n-1$, 定义标记随机变量

$$X_k = \begin{cases} 1 & \text{if PARTITION generates a } k : n-k-1 \text{ split,} \\ 0 & \text{otherwise.} \end{cases}$$

  - $E[X_k] = \Pr\{X_k = 1\} = 1/n$, 因为所有的分割是等概率出现的（所有元素均不同）

随机快排分析

$$T(n) = \begin{cases} T(0) + T(n-1) + \Theta(n) & \text{if } 0 : n-1 \text{ split,} \\ T(1) + T(n-2) + \Theta(n) & \text{if } 1 : n-2 \text{ split,} \\ \quad\vdots & \\ T(n-1) + T(0) + \Theta(n) & \text{if } n-1 : 0 \text{ split,} \end{cases}$$

$$= \sum_{k=0}^{n-1} X_k \left(T(k) + T(n-k-1) + \Theta(n)\right).$$

随机快排分析

$$E[T(n)] = E\left[\sum_{k=0}^{n-1} X_k\left(T(k) + T(n-k-1) + \Theta(n)\right)\right]$$

$$= \sum_{k=0}^{n-1} E\left[X_k\left(T(k) + T(n-k-1) + \Theta(n)\right)\right]$$

$$= \sum_{k=0}^{n-1} E[X_k] \cdot E[T(k) + T(n-k-1) + \Theta(n)]$$

$$= \frac{1}{n}\sum_{k=0}^{n-1} E[T(k)] + \frac{1}{n}\sum_{k=0}^{n-1} E[T(n-k-1)] + \frac{1}{n}\sum_{k=0}^{n-1} \Theta(n)$$

$$= \frac{2}{n}\sum_{k=1}^{n-1} E[T(k)] + \Theta(n)$$

Summations have identical terms.

随机快排分析

$$E[T(n)] = \frac{2}{n}\sum_{k=2}^{n-1} E[T(k)] + \Theta(n)$$

- 代入法证明
  - 这里 k=0,1被放入到尾项中
  - $E[T(n)] \le an\lg n$ 足够大的常数 $a > 0$

Use fact: $\sum_{k=2}^{n-1} k \lg k \le \frac{1}{2} n^2 \lg n - \frac{1}{8} n^2$ (exercise).

随机快排分析

$$
\begin{aligned}
E[T(n)] &\le \frac{2}{n}\sum_{k=2}^{n-1} ak\lg k + \Theta(n) \\
&= \frac{2a}{n}\left(\frac{1}{2}n^2\lg n - \frac{1}{8}n^2\right) + \Theta(n) \\
&= an\lg n - \left(\frac{an}{4} - \Theta(n)\right) \\
&\le an\lg n,
\end{aligned}
$$

- 如果 a 选择的足够大，使得 $an/4$ 超过 $\Theta(n)$ 即可

快速排序的实际使用

- 快速排序是一个非常好的通用排序算法
  - Quicksort 通常是归并排序速度的两倍
  - 快速排序在缓存和虚拟内存上同样工作良好——原址的

线性时间选择

给定线性序集中 n 个元素和一个整数 k，$1 \le k \le n$，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);  // left part
      else return RandomizedSelect(a,i+1,r,k-j);    // right part
}
```

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 ($0<\varepsilon<1$ 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

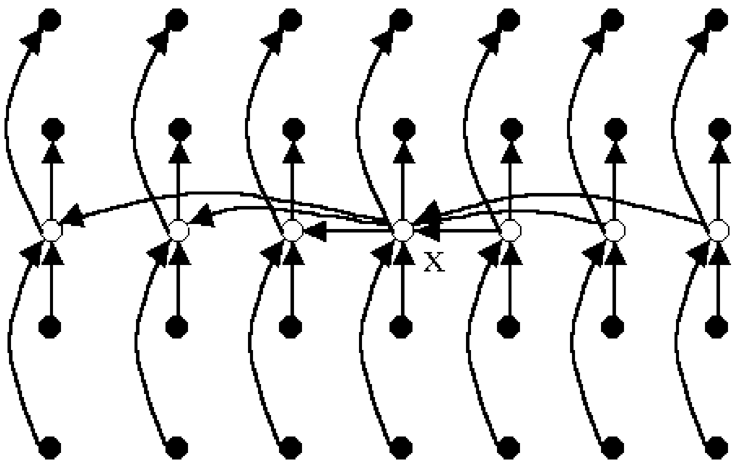
例如，若 $\varepsilon=9/10$，算法递归调用所产生的子数组的长度至少缩短 1/10

所以，在最坏情况下，算法所需的计算时间 $T(n)$ 满足递归式 $T(n)\leq T(9n/10)+O(n)$

由此可得 $T(n)=O(n)$

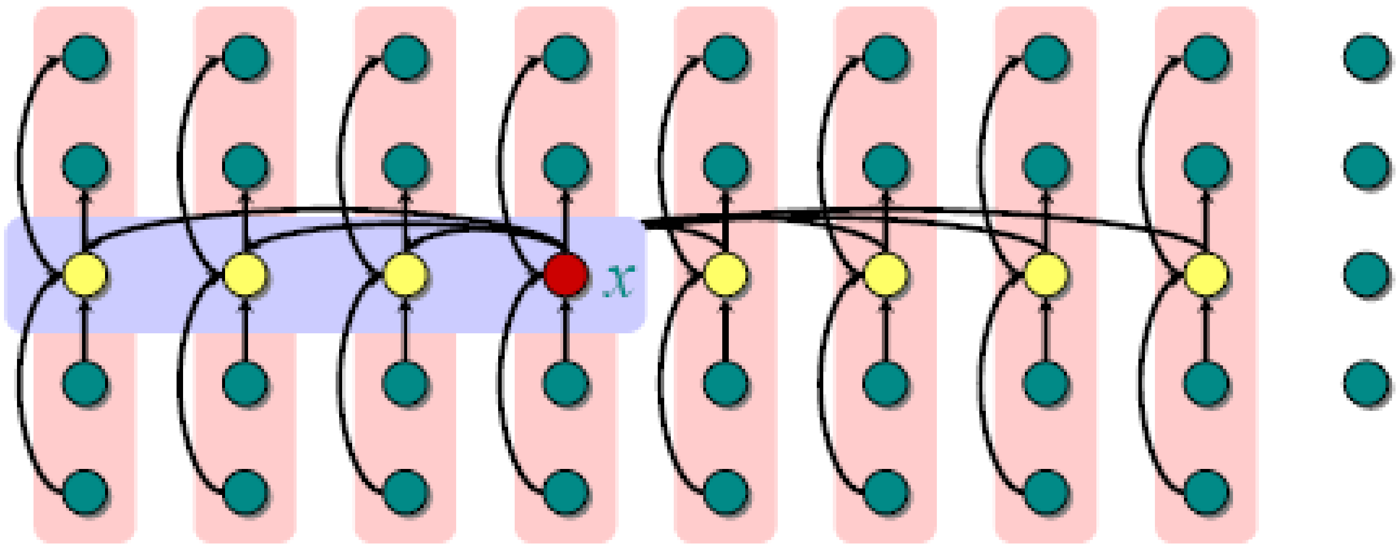
线性时间选择

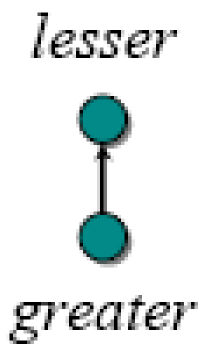
- 将 n 个输入元素划分成 $\lceil n/5 \rceil$ 个组，每组 5 个元素，只可能有一个组不是 5 个元素；用任意一种排序算法，将每组中的元素排好序，并取出每组的中位数，共 $\lceil n/5 \rceil$ 个
- 递归调用 **select** 来找出这 $\lceil n/5 \rceil$ 个元素的中位数。如果 $\lceil n/5 \rceil$ 是偶数，就找它的2个中位数中较大的一个，以这个元素作为划分基准



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。

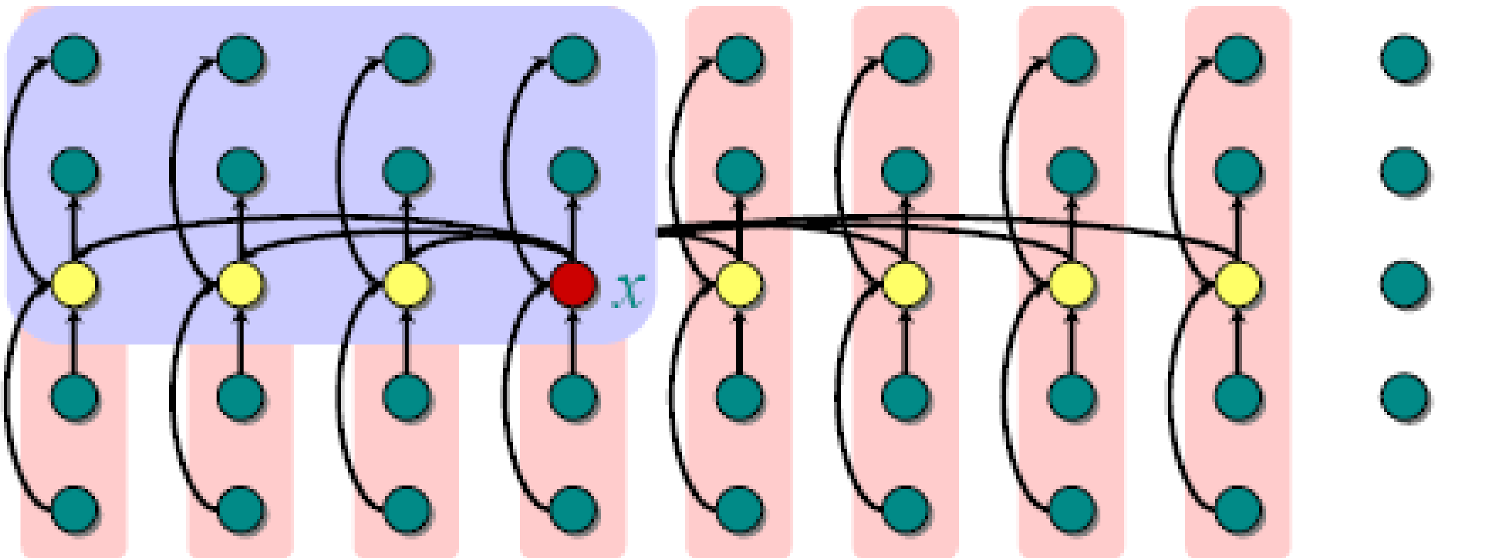
线性时间选择





至少一半的组中位数是 $\leq x$, 且组的个数至少为 $\lfloor \lfloor n/5 \rfloor /2 \rfloor = \lfloor n/10 \rfloor$

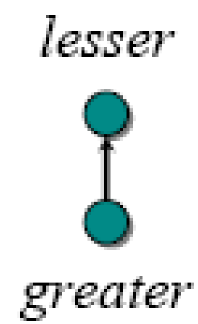
线性时间选择



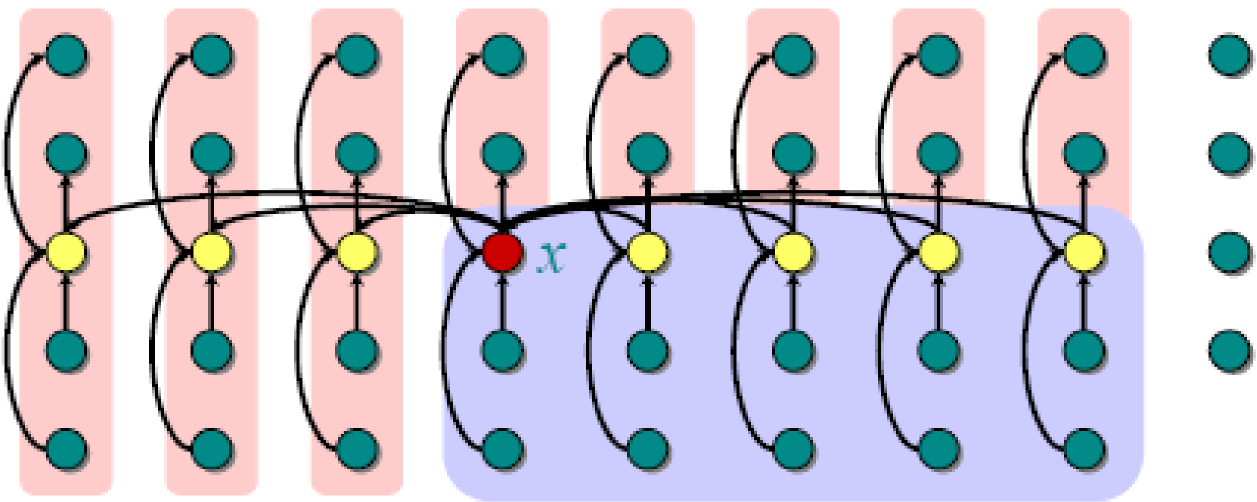
至少一半的组中位数是$\le x$, 且组的个数至少为

$\lfloor\lfloor n/5\rfloor /2\rfloor = \lfloor n/10\rfloor$

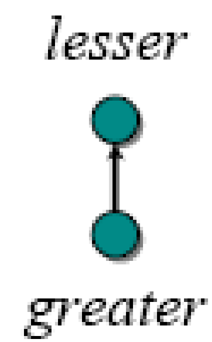
→ 至少 $3\lfloor n/10\rfloor$ 个元素是$\le x$



线性时间选择



至少一半的组中位数是$\le x$, 且组的个数至少为 $\lfloor \lfloor n/5 \rfloor /2 \rfloor = \lfloor n/10 \rfloor$



→至少有 $3\lfloor n/10 \rfloor$ 个元素是$\le x$

→至少有 $3\lfloor n/10 \rfloor$ 个元素是$\ge x$

```
Type Select(Type a[], int p, int r, int k)
{
    }
    for ( int i = 0; i<=(r-p-4)/5; i++ )
      将a[p+5*i]至a[p+5*i+4]的第3小元素
      与a[p+i]交换位置;
    if (k<=j) return Select(a,p,i,k); // left part
    else return Select(a,i+1,r,k-j);   // right part
}
```

复杂度分析

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

$$T(n)=\mathbf{O}(n)$$

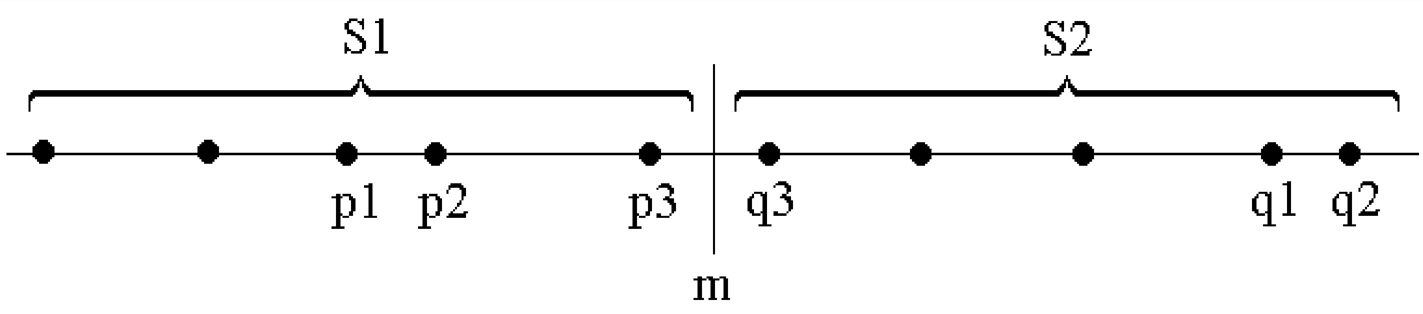
上述算法将每一组的大小定为 5，并选取 75 作为是否作递归调用的分界点，这 2 点保证了 $T(n)$ 的递归式中 2 个自变量之和 $n/5+3n/4=19n/20=\varepsilon n$，$0<\varepsilon<1$，这是使 $T(n)=O(n)$ 的关键之处

当然，除了5和75之外，还有其他选择

最接近点对问题

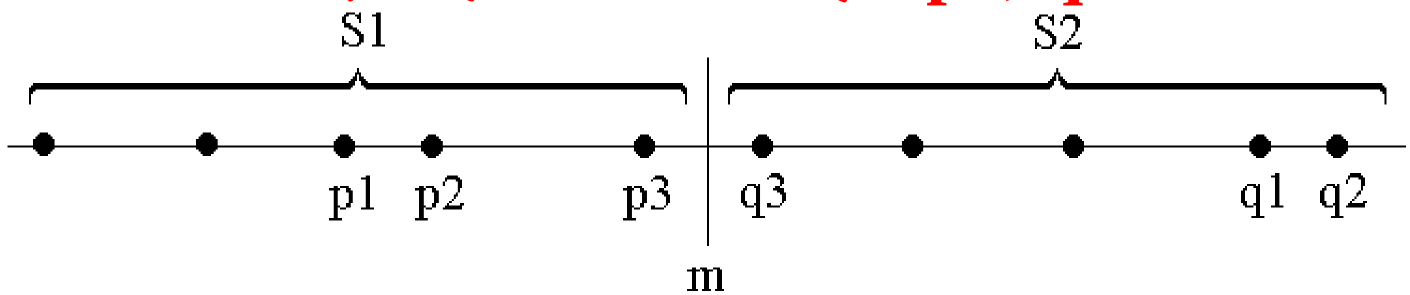
- 为了使问题易于理解和分析，先来考虑一维的情形。此时，S中的 n 个点退化为 x 轴上的 n 个实数 $x_1,x_2,\ldots,x_n$，最接近点对即为这 n 个实数中相差最小的 2 个实数

- 假设我们用 x 轴上某个点 m 将 S 划分为 2 个子集 S_1 和 S_2，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 S_1 和 S_2 上找出其最接近点对$\{p_1,p_2\}$和$\{q_1,q_2\}$，并设 $\boldsymbol{d=\min\{|p_1-p_2|,|q_1-q_2|\}}$，S 中的最接近点对或者是$\{p_1,p_2\}$，或者是$\{q_1,q_2\}$，或者是某个$\{p_3,q_3\}$，其中$p_3\in S_1$且$q_3\in S_2$
- **能否在线性时间内找到 p_3, q_3?**



最接近点对问题

能否在线性时间内找到p3, q3?

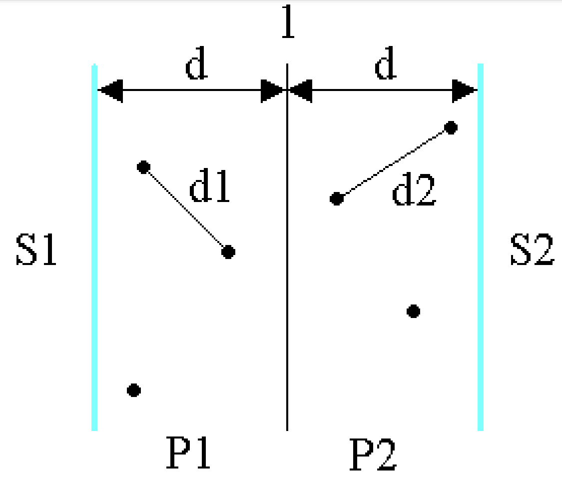


- 如果S的最接近点对是$\{p_3,q_3\}$，即 $|p_3-q_3|<d$，则 p_3 和 q_3 两者与 m 的距离不超过 d，即 $p_3\in(m-d,m]$，$q_3\in(m,m+d]$
- 由于在S_1中，每个长度为 d 的半闭区间至多包含一个点（否则必有两点距离小于d），并且 m 是 S_1 和 S_2 的分割点，因此 **$(m-d,m]$**中至多包含 S 中的一个点。由图可以看出，如果$(m-d, m]$中有S中的点，则此点就是S_1中最大点
- 因此，我们用线性时间就能找到区间 $(m-d,m]$和 $(m,m+d]$中所有点，即 p_3 和 q_3。从而我们用线性时间就可以将 S_1 的解和 S_2 的解合并成为 S 的解

最接近点对问题

下面来考虑二维的情形

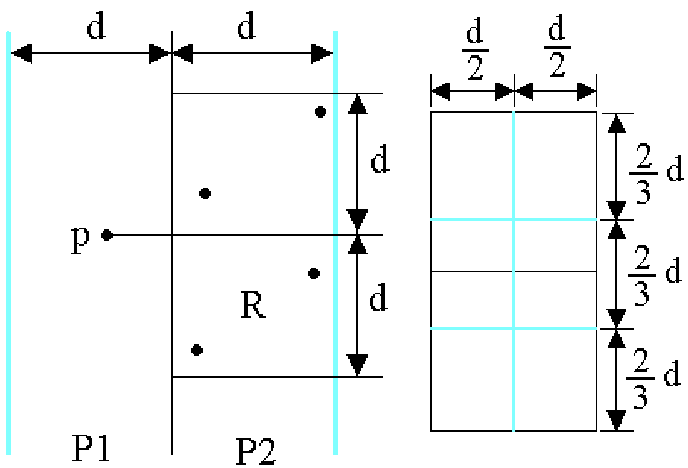
- 选取一垂直线 $l: x=m$ 来作为分割直线，其中 m 为 S 中各点 x 坐标的中位数，由此将 S 分割为 S_1 和 S_2
- 递归地在 S_1 和 S_2 上找出其最小距离 d_1 和 d_2，并设 $d=\min\{d_1,d_2\}$，S 中的最接近点对或者是 d，或者是某个 $\{p,q\}$，其中 $p\in P_1$ 且 $q\in P_2$
- **能否在线性时间内找到 p, q?**



最接近点对问题

能否在线性时间内找到 p_3, q_3?

- 考虑 P_1 中任意一点p，它若与 P_2 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 P_2 中的点一定落在一个 $d\times 2d$ 的矩形 R中
- 由 d 的意义可知，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
- 因此，在分治法的合并步骤中最多只需要检查 $6\times n/2=3n$ 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 $(d/2)\times(2d/3)$ 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 $(d/2)\times(2d/3)$ 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 P_2 中所有S_2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S_2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 P_1 和 P_2 中所有 S 中点按其 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
  - 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

最接近点对问题

double **cpair2**(S)

{

n=|S|

if (n

1、m=S

数;

构造S1和S2;

//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

列；

查Y

)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

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

$$\boldsymbol{T(n)=O(n\log n)}$$

循环赛日程表

设计一个满足以下要求的比赛日程表： $n = 2^k$

(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

Next

- 数据结构
  - Data structures