

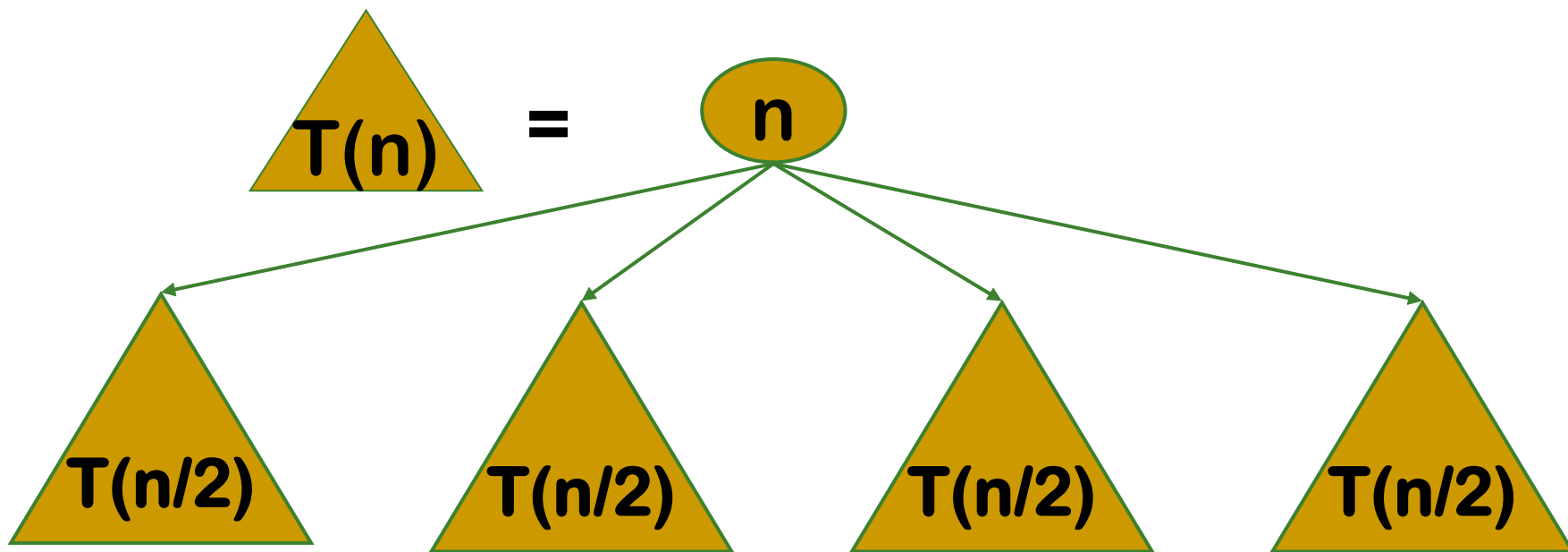


第2章 递归与分治策略



算法总体思想

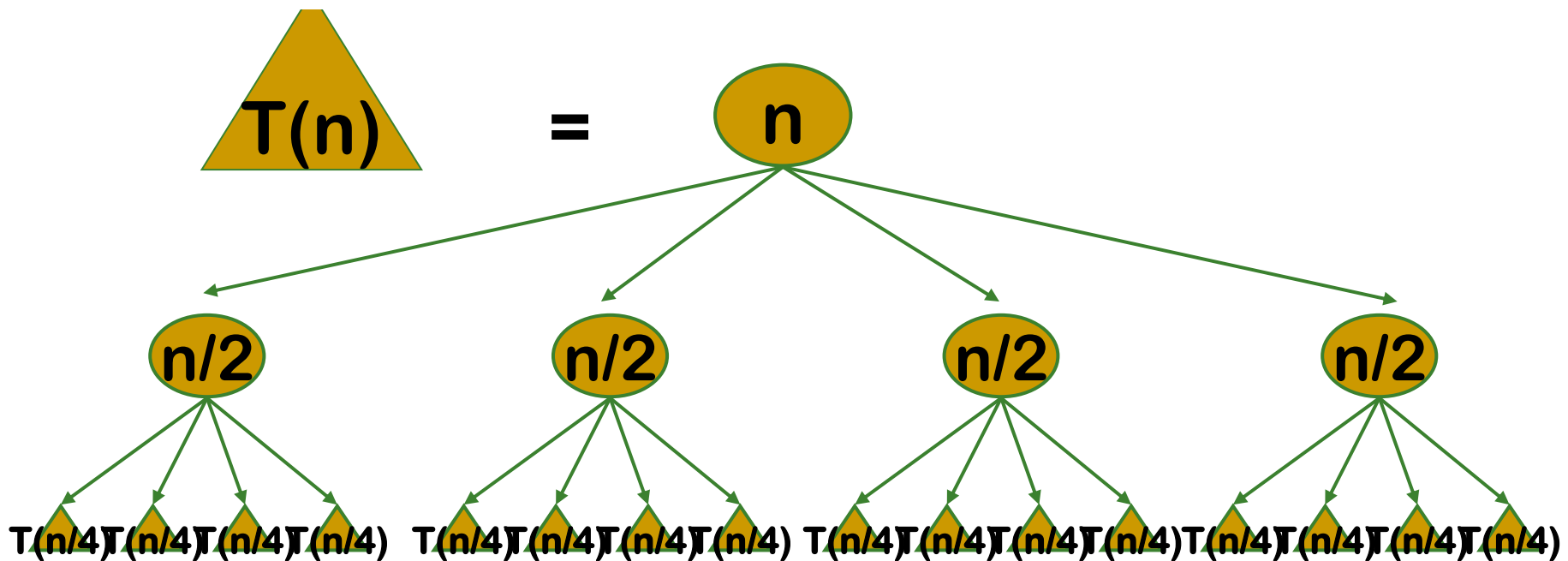
- 对这 k 个子问题分别求解。如果子问题的规模仍然不够小，则再划分为 k 个子问题，如此递归的进行下去，直到问题规模足够小，很容易求出其解为止。





算法总体思想

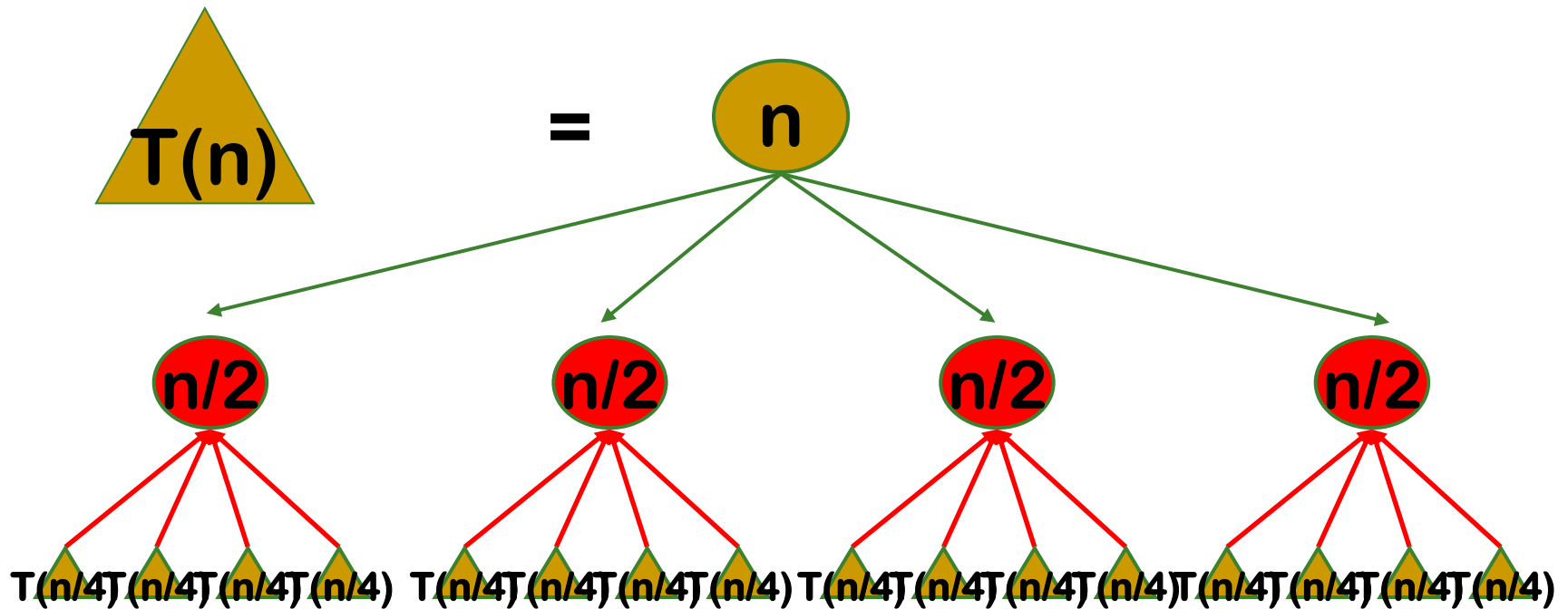
- 将求出的小规模的问题的解合并为一个更大规模的问题的解，自底向上逐步求出原来问题的解。





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。





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



2.1 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。



2.1 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。



2.1 递归的概念

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);
}
```




2.1 递归的概念

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：

$$\left\{ \begin{array}{ll} A(1,0) = 2 \\ A(0,m) = 1 & m \geq 0 \\ A(n,0) = n + 2 & n \geq 2 \\ A(n,m) = A(A(n-1,m), m-1) & n, m \geq 1 \end{array} \right.$$



2.1 递归的概念

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：

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

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

但本例中的Ackerman函数却无法找到非递归的定义。



2.1 递归的概念

- $A(n, m)$ 的自变量 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, $A(n, 1)=A(A(n-1, 1), 0)=A(n-1, 1)+2$, 和 $A(1, 1)=2$ 故 $A(n, 1)=2^n$
- $M=2$ 时, $A(n, 2)=A(A(n-1, 2), 1)=2A(n-1, 2)$, 和 $A(1, 2)=A(A(0, 2), 1)=A(1, 1)=2$, 故 $A(n, 2)=2^{2^n}$ 。
$$\underbrace{2^{2^{2^{\cdot^{\cdot^{\cdot}}}}}}_n$$
- $M=3$ 时, 类似的可以推出
- $M=4$ 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。



2.1 递归的概念

- 定义单变量的Ackerman函数 $A(n)$ 为, $A(n)=A(n, n)$ 。
- 定义其拟逆函数 $\alpha(n)$ 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。



2.1 递归的概念

例4 排列问题

设计一个递归算法生成 n 个元素 $\{r_1, r_2, \dots, r_n\}$ 的全排列。

设 $R = \{r_1, r_2, \dots, r_n\}$ 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 $\text{perm}(X)$ 。

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



2.1 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： $n = n_1 + n_2 + \dots + n_k$ ，其中 $n_1 \geq n_2 \geq \dots \geq n_k \geq 1$ ， $k \geq 1$ 。

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。



2.1 递归的概念

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 的划分数记作 $q(n, m)$ 。可以建立 $q(n, m)$ 的如下递归关系。

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

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。

$$(4) \quad q(n, m) = q(n, m-1) + q(n-m, m), n > m > 1;$$

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 $n_1 \leq n-1$ 的划分组成。



2.1 递归的概念

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 $p(n) = q(n, n)$ 。



2.1 递归的概念

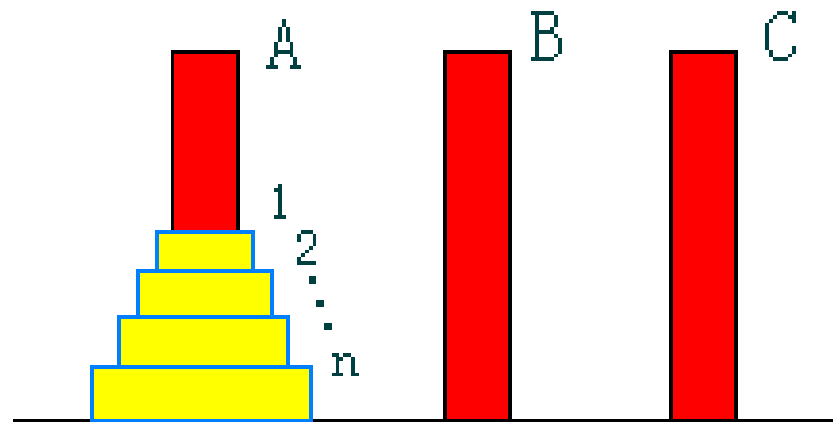
例6 Hanoi塔问题

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 $1, 2, \dots, n$ ，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：

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。





2.1 递归的概念

例6 Hanoi塔问题

```
public static 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);
    }
}
```

思考：如果塔的个数变为a,b,c,d四个，现要将n个圆盘从a全部移动到d，移动规则不变，求移动步数最小的方案。



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。



递归-分治的复杂性分析

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。设分解阈值 $n_0=1$ ，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 $m^i \leq n < m^{i+1}$ 时， $T(m^i) \leq T(n) < T(m^{i+1})$ 。



递归-分治的复杂性分析

分析方法：

- (1) 根据算法建立复杂度的递归方程。
- (2) 求解递归方程，得到时间复杂度。

如：阶乘函数：

```
int factorial(int n)
{
    if (n == 0) return 1;
    return n*factorial(n-1);
}
```

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

$$T(n) = n + 1 = \Theta(n)$$



递归方程的求解

递推求解

【例】 汉诺塔求解算法：
$$T(n) = \begin{cases} 1 & n = 1 \\ 2T(n-1) + 1 & n > 1 \end{cases}$$

递推展开：

$$T(n) = 2T(n-1) + 1$$

$$= 2[2T(n-2) + 1] + 1 = 2^2T(n-2) + 1 + 2$$

$$= 2^2[2T(n-3) + 1] + 2 + 1 = 2^3T(n-3) + 1 + 2 + 2^2$$

.....

$$= 2^{n-1}[2T(n-(n-1)) + 1] + 1 + 2 + 2^2 + 2^3 + \dots + 2^{n-3} + 2^{n-2}$$

$$= 1 + 2 + 2^2 + 2^3 + \dots + 2^{n-2} + 2^{n-1} + 2^n = \dots$$

$$= 1 + [2 + \dots + 2^{n-1}] + 2^n$$

$$= 1 + 2^n + 2(2^{n-1} - 1)$$

$$= 3 \cdot 2^{n-1} = O(2^n)$$



递推求解

【定理1】 设 a, b, c 均为正整数, $a \geq 1, b > 1, c > 0$, 递归方程

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

$$\text{有: } T(n) = \begin{cases} c(\log_b n + 1) & a = 1 \\ c \frac{an^{\log_b a} - 1}{a - 1} & n \neq 1 \end{cases}$$

设 $n > b^k, k > 0$, 递推展开

$$\begin{aligned} T(n) &= aT(n/b) + c = a^2T(n/b^2) + ac + c = a^3T(n/b^3) + a^2c + ac + c = \dots \\ &= a^kT(n/b^k) + a^{k-1}c + \dots + ac + c = a^kT(1) + a^{k-1}c + \dots + ac + c \\ &= c \frac{a^{k+1} - 1}{a - 1} = c(a^k + a^{k-1} + \dots + a + 1) = c(k+1) \end{aligned}$$

其中: $k = \log_b n; a^{\log_b n} = n^{\log_b a}$



递推求解

【定理2】 设 a, b, c 均为正整数, $a \geq 1, b > 1, c > 0$, 递归方程

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

的解是: $T(n) = \begin{cases} O(n) & a < b \\ O(n \log_b n) & a = b \\ O(n^{\log_b a}) & a > b \end{cases}$

设 $n > b^k, k > 0$, 递推展开

$$T(n) = a^k + \sum a^i (cn/b^i) = a^k + cn \sum (a/b)^i$$

$a = b$ $T(n) = a^k + cnk = b^k + cn \log_b n = n + cn \log_b n$

$a > b$ $\log_b a > 1, n^{\log_b a} > n, T(n) = a^k + cn \frac{(a/b)^k - 1}{(a/b) - 1} \approx cn^{\log_b a}$

$a < b$ $\log_b a < 1, n^{\log_b a} < n, T(n) = n + cn$



递推求解

一般化，设 a, b, c 均为正整数, $a \geq 1, b > 1, c > 0$, 递归方程

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

的解是:
$$T(n) = n^{\log_b a} + \sum_{j=0}^{\log_b n - 1} a^j d(n/b^j)$$

设 $n = b^k$, 递推展开

$$T(n) = aT(n/b) + d(n)$$

$$= a^2T(n/b^2) + ad(n/b) + d(n)$$

$$= a^3T(n/b^3) + a^2d(n/b^2) + ad(n/b) + d(n)$$

.....

$$= a^k + \sum_{j=0}^k a^j d(b^{k-j})$$



例：快速排序的复杂度分析

```
Void QSort(int R[], int left, int right){  
    if(left < right){  
        int m = Partition(R, left, right); //Partition 复杂度为 O(right-left+1)  
        QSort(R, left, m-1);  
        QSort(R, m+1, right);  
    }  
} //Qsort
```

$$\text{最好情况: } T(n) = \begin{cases} c & n = 1 \\ 2T(n/2) + cn & n > 1 \end{cases} \quad T(n) = \Theta(n \log n)$$

$$\text{最坏情况: } T(n) = \begin{cases} c & n = 1 \\ T(n-1) + cn & n > 1 \end{cases} \quad T(n) = \Theta(n^2)$$



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：如果 $n=1$ 即只有一个元素，则只要比较这个元素和 x 就可以确定 x 是否在表中。因此这个问题满足分治法的第一个适用条件



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 。

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 $x > a[i]$ ，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 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。



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：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：

```
public static int binarySearch(int [] a, int x, int n)
{
    // 在  $a[0] \leq a[1] \leq \dots \leq a[n-1]$  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;
        else right = middle - 1;
    }
    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 $O(\log n)$ 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 $O(\log n)$ 。



思考：搜索技术

给定一个长度为 n 的字符串 $a[0:n-1]$ ，现要在这个字符串中找出一长度为 m 的特定子串 $b[0:m-1]$ ， n 远大于 m ；字符串包含的字符集合为 A ，子串包含的字符集合为 B ，其中 B 是 A 的子集。

怎么分？

- ✓ 对字符串 a 进行划分
- ✓ 对字符串 b 进行划分

启示：虽然我们可以对字符串 a 或者对字符串 b 进行划分，观察发现，对 a 或者 b 划分后得到的子问题的规模缩小了，然而问题的规模缩小到一定的程度是否可以容易地解决？在这种划分下不能。因此没有起到简化问题或加快求解速度的目的。



思考：搜索技术

给定一个长度为 n 的字符串 $a[0:n-1]$ ，现要在这个字符串中找出一长度为 m 的特定子串 $b[0:m-1]$ ， n 远大于 m ；字符串包含的字符集合为 A ，子串包含的字符集合为 B ，其中 B 是 A 的子集。

- 怎么分？
- ✓ 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。

实际中，此类问题的常见需求是：

给定一个长度为 n 的字符串 $a[0:n-1]$ 和一系列长度为 m 的特定子串 $b[0:m-1]$ ， n 远大于 m

场景：检索服务、论文查重等



思考：搜索技术

构建对应的数据结构（1），举例：满字典树（变形的k叉树）

对于字符串 $a[0:n-1]$:

1. 从 $a[0]$ 开始，取一个长度为 m 的子串，即得到字符串 a 的子串 $a[1][0,m-1]$;
2. 将起始位置右移1位，从 $a[1]$ 开始，取一个长度为 m 的子串，即得到字符串 a 的子串 $a[2][1,m]$;
3. 以此类推，直至得到字符串 a 的子串 $a[n-m][n-m-1,n-1]$;
4. 令字符集合 A 中的字符数为 $\|A\|$ ，构建一颗变形的 k 叉树（其中 $k = \|A\|$ ）；树深 m 层；根节点、中间节点不包含字符，从这些每个节点射出的 k 条边各包含 A 中的1个字符；从根节点到某一叶节点的路径经过 m 条边，边的字符连接起来为对应的字符串；



思考：搜索技术

5. 将a的所有长度为m的子串依次在树上遍历，对于任一子串 $a[s][i,j]$ ，子串必然终止与某一叶节点，则在该叶节点记录子串信息。

算法复杂度分析： $O(m)$

对于字符串 $b[0:m-1]$:

6. 从树的根节点开始遍历字符串b，直至终止于某一叶节点：若该叶节点为空，则说明b不是a的子串；若该叶节点不为空，则说明b是a的子串，且在a中的起始位置为记录的值。

启示1：结合上述数据结构，可以对字符串b进行划分，观察发现，可以显著降低求解的空间复杂度。**要注意，虽然教材中的举例多为降低时间复杂度，不要忘记空间复杂度。**

启示2：也可以使用其他更先进的数据结构等提升搜索效率。



思考：搜索技术

构建对应的数据结构（1），举例：前缀树（字典树）

字典树也叫Trie树、前缀树，是一种针对字符串进行快速检索的多叉树结构。字典树把字符串看成字符序列，根据字符串中字符序列的先后顺序构造从上到下的树结构，树结构中的每一条边都对应着一个字符。字典树上存储的字符串被视为从根节点到某个节点之间的一条路径，并在终点节点上做个标记“该节点对应词语的结尾”。**优点是：利用字符串的公共前缀来减少查询时间，最大限度地减少无谓的字符串比较。**

字典树的基本操作增删改查，也就是插入（insert）、删除（delete）、修改（update）和查询（search）。



思考：搜索技术

构建对应的数据结构（2），举例：后缀树和后缀数组

观察字典树：相邻的2个字符串只有2位不同，如果能够将相似字符放在相邻的位置，是否有助于节约空间？

后缀数组构建算法： Burrows–Wheeler transform

1. 循环获得全部子串，构建字符矩阵M；
2. 按照字典序重排矩阵M；
3. 提取重排后的矩阵M的首列和末列。

后缀数组检索算法： Last-First Column算法

启示：结合上述数据结构，可以对字符串a进行划分，适当降低LF检索的时间复杂度（也有针对LF以空间换时间的方法）。



思考：搜索技术

构建对应的数据结构（2），举例：后缀树和后缀数组

举例：字符串abcbacba

其后缀子串为：

| a | b | c | b | a | c | b | a | \$ | | | | | | | | |
|---|---|---|---|---|---|---|---|----|--|--|--|--|--|--|--|--|
| a | b | c | b | a | c | b | a | \$ | | | | | | | | |
| | b | c | b | a | c | b | a | \$ | | | | | | | | |
| | | c | b | a | c | b | a | \$ | | | | | | | | |
| | | | b | a | c | b | a | \$ | | | | | | | | |
| | | | | a | c | b | a | \$ | | | | | | | | |
| | | | | | c | b | a | \$ | | | | | | | | |
| | | | | | | b | a | \$ | | | | | | | | |
| | | | | | | | a | \$ | | | | | | | | |
| | | | | | | | | \$ | | | | | | | | |



思考：搜索技术

构建对应的数据结构（2），举例：后缀树和后缀数组

举例：字符串abcbacba

将后缀
子串补全：

字符矩阵M

| a | b | c | b | a | c | b | a | \$ | | | | | | | | |
|---|---|---|---|---|---|---|---|----|---|---|---|---|---|---|---|---|
| a | b | c | b | a | c | b | a | \$ | | | | | | | | |
| | b | c | b | a | c | b | a | \$ | a | | | | | | | |
| | | c | b | a | c | b | a | \$ | a | b | | | | | | |
| | | | b | a | c | b | a | \$ | a | b | c | | | | | |
| | | | | a | c | b | a | \$ | a | b | c | b | | | | |
| | | | | | c | b | a | \$ | a | b | c | b | a | | | |
| | | | | | | b | a | \$ | a | b | c | b | a | c | | |
| | | | | | | | a | \$ | a | b | c | b | a | c | b | |
| | | | | | | | | \$ | a | b | c | b | a | c | b | a |



思考：搜索技术

构建对应的数据结构（2），举例：后缀树和后缀数组

举例：字符串abcbacba

将补足后的
子串对齐：

字符矩阵M

| a | b | c | b | a | c | b | a | \$ | | | | | | | | |
|----|----|----|----|----|----|----|----|----|--|--|--|--|--|--|--|--|
| a | b | c | b | a | c | b | a | \$ | | | | | | | | |
| b | c | b | a | c | b | a | \$ | a | | | | | | | | |
| c | b | a | c | b | a | \$ | a | b | | | | | | | | |
| b | a | c | b | a | \$ | a | b | c | | | | | | | | |
| a | c | b | a | \$ | a | b | c | b | | | | | | | | |
| c | b | a | \$ | a | b | c | b | a | | | | | | | | |
| b | a | \$ | a | b | c | b | a | c | | | | | | | | |
| a | \$ | a | b | c | b | a | c | b | | | | | | | | |
| \$ | a | b | c | b | a | c | b | a | | | | | | | | |



思考：搜索技术

构建对应的数据结构（2），举例：后缀树和后缀数组

举例：字符串abcbacba

按照字典序
重排矩阵M

| a | b | c | b | a | c | b | a | \$ | | | | | | | | |
|----|----|----|----|----|----|----|----|----|--|--|--|--|--|--|--|--|
| \$ | a | b | c | b | a | c | b | a | | | | | | | | |
| a | \$ | a | b | c | b | a | c | b | | | | | | | | |
| a | b | c | b | a | c | b | a | \$ | | | | | | | | |
| a | c | b | a | \$ | a | b | c | b | | | | | | | | |
| b | a | \$ | a | b | c | b | a | c | | | | | | | | |
| b | a | c | b | a | \$ | a | b | c | | | | | | | | |
| b | c | b | a | c | b | a | \$ | a | | | | | | | | |
| c | b | a | \$ | a | b | c | b | a | | | | | | | | |
| c | b | a | c | b | a | \$ | a | b | | | | | | | | |



思考：搜索技术

构建对应的数据结构（2），举例：后缀树和后缀数组

举例：字符串abcbacba

提取重排后的
矩阵M的首列
和末列

| a | b | c | b | a | c | b | a | \$ | | | | | | | | |
|----|---|---|---|---|---|---|---|----|--|--|--|--|--|--|--|--|
| \$ | | | | | | | | a | | | | | | | | |
| a | | | | | | | | b | | | | | | | | |
| a | | | | | | | | \$ | | | | | | | | |
| a | | | | | | | | b | | | | | | | | |
| b | | | | | | | | c | | | | | | | | |
| b | | | | | | | | c | | | | | | | | |
| b | | | | | | | | a | | | | | | | | |
| c | | | | | | | | a | | | | | | | | |
| c | | | | | | | | b | | | | | | | | |



思考：搜索技术

构建对应的数据结构（2），举例：后缀树和后缀数组

First-Last Column算法举例：查找字符串cbac

1. 定位后缀c
2. 查找c的前缀
3. 定位a
4. 查找a的前缀
5. ...

| 首 | | | | | | | | 末 | | | | | | | | |
|----|--|--|--|--|--|--|--|----|--|--|--|--|--|--|--|--|
| \$ | | | | | | | | a | | | | | | | | |
| a | | | | | | | | b | | | | | | | | |
| a | | | | | | | | \$ | | | | | | | | |
| a | | | | | | | | b | | | | | | | | |
| b | | | | | | | | c | | | | | | | | |
| b | | | | | | | | c | | | | | | | | |
| b | | | | | | | | a | | | | | | | | |
| c | | | | | | | | a | | | | | | | | |
| c | | | | | | | | b | | | | | | | | |



思考：搜索技术

构建对应的数据结构（2），举例：后缀树和后缀数组

后缀数组构建算法： Burrows–Wheeler transform

1. 循环获得全部子串，构建字符矩阵M；
2. 按照字典序重排矩阵M；
3. 提取重排后的矩阵M的首列和末列。

后缀数组检索算法： Last-First Column算法

进一步思考： 找到第k个字符需要多少时间？

启示：结合上述数据结构，可以对字符串a进行划分，适当降低LF检索的时间复杂度（也有针对LF以空间换时间的方法）。



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:

$$\begin{array}{l} X = \boxed{a} \boxed{b} \\ Y = \boxed{c} \boxed{d} \end{array}$$

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

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

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☹



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：

XY 复杂度分析

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

1. $T(n) = O(n^{\log_3 3}) = O(n^{1.59})$ ✓ 较大的改进😊

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

细节问题：两个XY的复杂度都是 $O(n^{\log_3 3})$ ，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 $O(n \log n)$ 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。



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
个大小

复杂度分析

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

$T(n) = O(n^3)$ ✖没有改进☹

由此可得:

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

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

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

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

成4



Strassen矩阵乘法

为了降低

复杂度分析

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

$$T(n) = O(n^{\log_2 7}) = O(n^{2.81}) \quad \checkmark \text{较大的改进}\odot$$

$$M_1 = A_{11}B_{22}$$

$$M_2 = (A_{11} + A_{12})B_{21}$$

$$M_3 = (A_{21} + A_{22})B_{11}$$

$$M_4 = A_{22}(B_{21} - B_{11})$$

$$M_5 = (A_{11} + A_{22})(B_{11} + B_{22})$$

$$M_6 = (A_{12} - A_{22})(B_{21} + B_{22})$$

$$M_7 = (A_{11} - A_{21})(B_{11} + B_{12})$$



$$C_{12} = M_1 + M_2$$

$$C_{21} = M_3 + M_4$$

$$C_{22} = M_5 + M_1 - M_3 - M_7$$



Strassen矩阵乘法

◆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? ? ? 目前为止还没有结果。



对递归时间复杂度函数的讨论

$$a^k + \sum_{j=0}^K a^j d(b^{k-j})$$

第一项 a^k

由 $n = b^k$, 可得 $k = \log_b n$, 从而 $a^k = a^{\log_b n} = n^{\log_b a}$, 对这个结果, 一般讲如 b 保持不动, a 越大指数越高复杂度阶数越高;
如 a 保持不动, b 越大指数越低复杂度阶数越低。

第二项 $\sum_{j=0}^K a^j d(b^{k-j})$

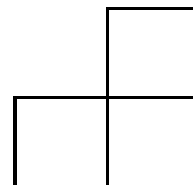
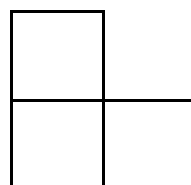
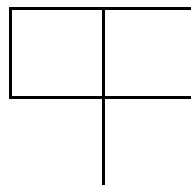
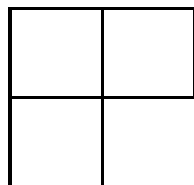
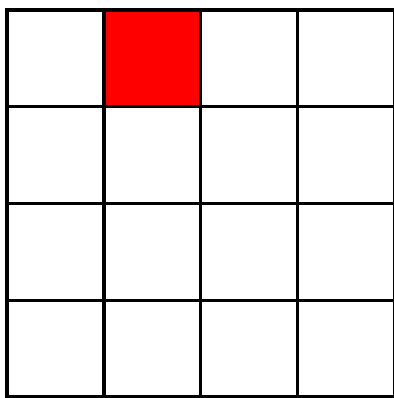
一般讲不给定 $d(b)$ 的解析式是无法解出 \sum 的; 如 $d(b)$ 是可积函数, 则 $d(b^{k-j}) = (d(b))^{k-j}$, 可积 $\forall n, m > 0$ 有 $f(nm) = f(n) \cdot f(m)$

$$\begin{aligned} \sum_{j=0}^K a^j d(b^{k-j}) &= \sum_{j=0}^K a^j (d(b))^{k-j} = \sum_{j=0}^K a^j (d(b))^k (d(b))^{-j} = (d(b))^k \sum_{j=0}^K (a/d(b))^{-j} \\ &= d^k(b) \frac{(a/d(b))^k - 1}{a/d(b) - 1} = \frac{a^k - d^k(b)}{a/d(b) - 1} \end{aligned}$$



棋盘覆盖

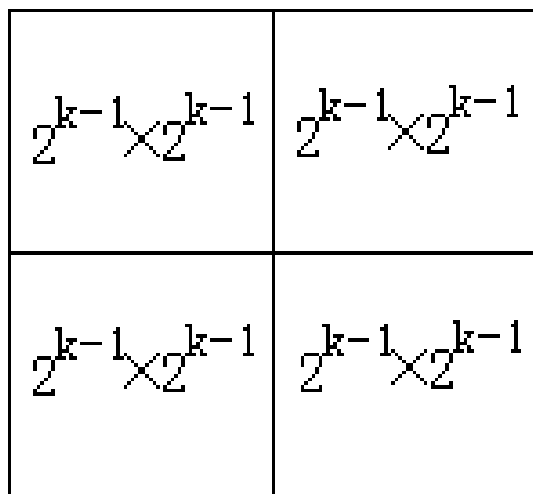
在一个 $2^k \times 2^k$ 个方格组成的棋盘中，恰有一个方格与其他方格不同，称该方格为一特殊方格，且称该棋盘为一特殊棋盘。在棋盘覆盖问题中，要用图示的4种不同形态的L型骨牌覆盖给定的特殊棋盘上除特殊方格以外的所有方格，且任何2个L型骨牌不得重叠覆盖。



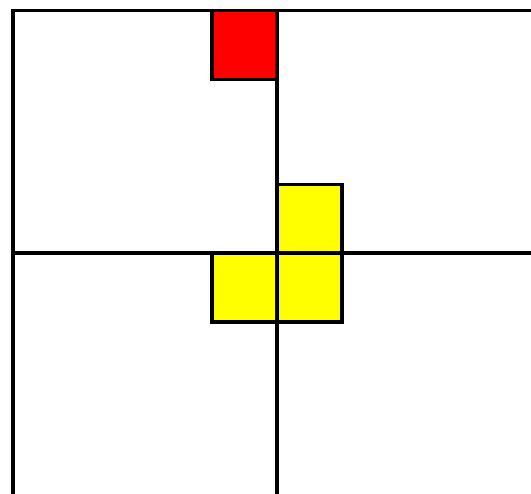


棋盘覆盖

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

```

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 + 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

复杂度分析

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

$T(n)=O(n\log n)$ 渐进意义下的最优算法

```
public
```

```
{
```

```
if (
```

```
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
```

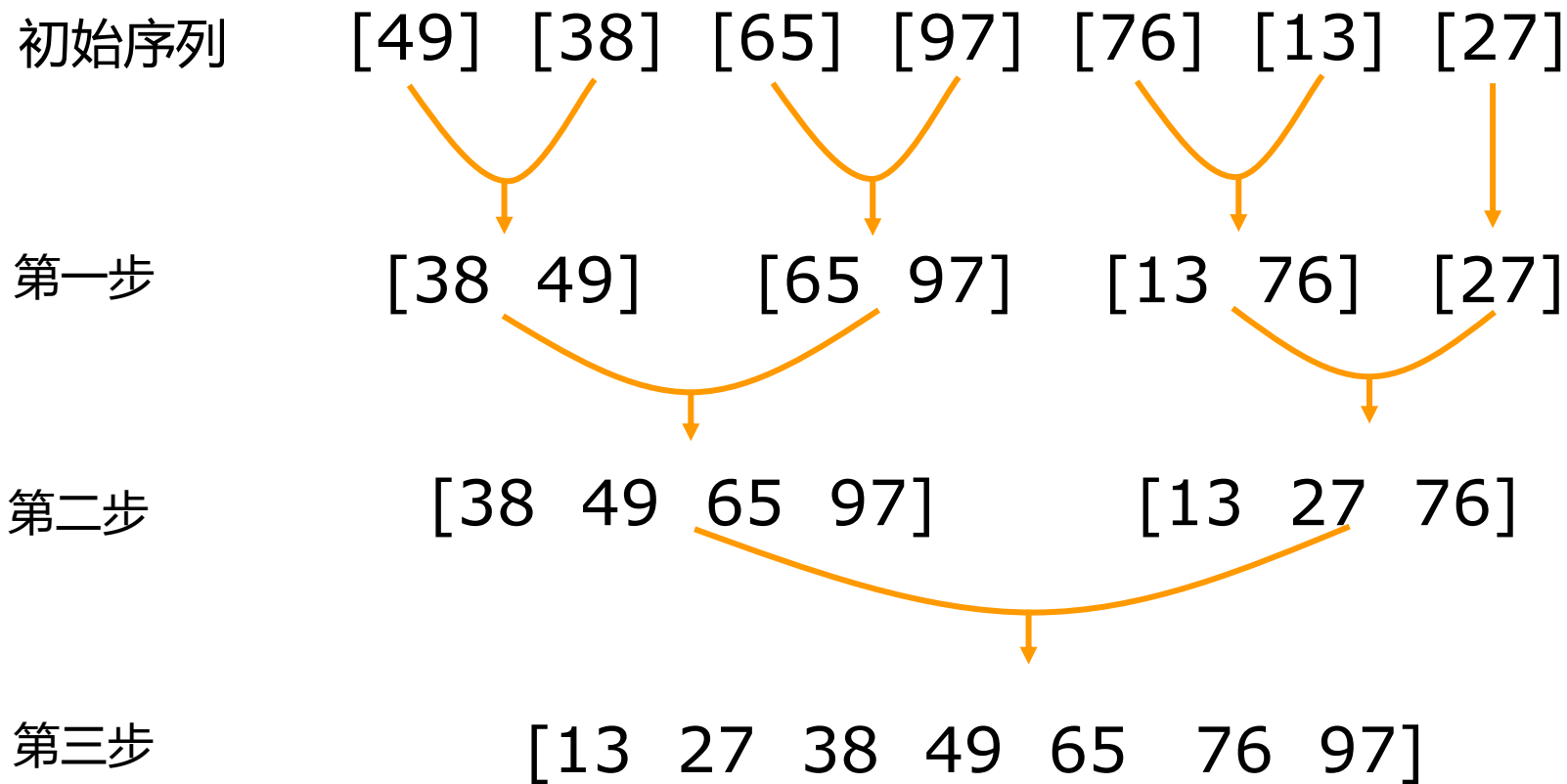
```
}
```

```
}
```



合并排序

算法mergeSort的递归过程可以消去。





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: $O(n)$

📖 稳定性: 稳定



快速排序

快速排序是对气泡排序的一种改进方法，它是由C. A. R. Hoare于1962年提出的

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。

```
private static void qSort(int p, int r)
```

```
{
```

```
    if (p<r) {
```

```
        int q=partition(p,r); //以a[p]为基准元素将a[p:r]划分成  
        3段a[p:q-1],a[q]和a[q+1:r], 使得a[p:q-1]中任何元素小于等  
        于a[q], a[q+1:r]中任何元素大于等于a[q]。下标q在划分过  
        程中确定。
```

```
        qSort (p,q-1); //对左半段排序
```

```
        qSort (q+1,r); //对右半段排序
```

```
    }
```

```
}
```



快速排序

```
private static int partition (int p, int r)
```

```
{
    int i = p,
        j = r + 1;
    Comparable 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].compareTo(x) < 0);
        while (a[--j].compareTo(x) > 0);
        if (i >= j) break;
        MyMath.swap(a, i, j);
    }
    a[p] = a[j];
    a[j] = x;
    return j;
}
```

{ 6, 7, 5, 2, 5, 8} 初始序列

{ 6, 7, 5, 2, 5, 8} j--;

{ 5, 7, 5, 2, 6, 8} i++;

{ 5, 6, 5, 2, 7, 8} j--;

{ 5, 2, 5, 6, 7, 8} i++;

{ 5, 2, 5} 6 {7, 8} 完成

快速排序具有不稳定性。



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

📖 **最坏时间复杂度： $O(n^2)$**

📖 **平均时间复杂度： $O(n \log n)$**

📖 **辅助空间： $O(n)$ 或 $O(\log n)$**

📖 **稳定性： 不稳定**

```
private static int randomizedPartition (int p, int r)
{
    int i = random(p,r);
    MyMath.swap(a, i, p);
    return partition (p, r);
}
```



例：快速排序的复杂度分析

平均情况：

设 $T(n)$ 代表快速排序对 n 个待排序记录进行排序平均时间，假设每次划分产生一个长度为 r 的子序列，另一个长度为 $n-r-1$ 的子序列。则 $T(n)=n-1+T(r)+T(n-r-1)$ 。

假定在1到 n 之间进行划分，选择了第 p 个位置为基准位置。在划分后，左边的子序列为 $\{1, 2, \dots, p-1\}$ ，右边的子序列为 $\{p+1, p+2, \dots, n\}$ 。假设 p 在1到 n 之间的取值是等概率的，即为 $1/n$ 。则：

$$\begin{aligned} T(n) &= n-1 + \frac{1}{n} \sum_{p=1}^n T[p-1] + T[n-p] \\ &= n-1 + \frac{2}{n} \sum_{k=0}^{n-1} T[k] \end{aligned}$$



递推求解

$$T(n) = n - 1 + \frac{2}{n} \sum_{k=0}^{n-1} T[k] \quad (1)$$

$$T(n-1) = n - 2 + \frac{2}{n-1} \sum_{k=0}^{n-2} T[k] \quad (2)$$

由（1）和（2）可以得到：

$$nT(n) - (n-1)T(n-1) = 2(n-1) + 2T(n-1)$$

$$\begin{aligned} \frac{T(n)}{n+1} &= \frac{T(n-1)}{n} + 2 \frac{n-1}{n(n+1)} = \frac{T(n-1)}{n} + 2 \left[-\frac{1}{n} + \frac{2}{n+1} \right] \\ &= \frac{T(n-2)}{n-1} + 2 \left[-\frac{1}{n-1} + \frac{2}{n} \right] + 2 \left[-\frac{1}{n} + \frac{2}{n+1} \right] = \frac{T(n-2)}{n-1} + 2 \left[-\frac{1}{n-1} + \frac{1}{n} + \frac{2}{n+1} \right] = \dots \\ &= \frac{T(2)}{3} + 2 \left[-\frac{1}{3} + \frac{1}{4} + \dots + \frac{1}{n} + \frac{2}{n+1} \right] \end{aligned}$$

$$T(n) = 2(n+1)[\ln n + O(1)] = 2n \ln n + O(n)$$



推测验证

先推测递归方程的一个解，然后用数学方法证明解得正确性。例如：

$$T(n) = \begin{cases} c_1 & n \leq 6 \\ T(\frac{n}{2}) + T(\frac{n}{3}) + c_2 n & n > 6 \end{cases}$$

推测：

首先，由于 $T(n) > c_2 n$ ，所以 $T(n) = \Omega(n)$ 。其次，由于 $1/2 + 1/3 < 1$ ，可猜测 $T(n) = O(n)$ 。



推测验证

验证：

只需证明 $T(n)=O(n)$ 。用数学归纳法。

当 $n \leq 6$ 时， $T(n)=c_1 \leq c_1 n$ ， $T(n)=O(n)$ 成立(取 $c=c_1, n_0=1$)。

假设当 $n \leq k$ 时， $T(n)=O(n)$ 成立。即存在一个 c 和 n_0 ，当 $n > n_0$ 时， $T(n) \leq c n$ 。

$$\begin{aligned} \text{当 } n=k+1 \text{ 时, } T(n) &= T\left(\frac{n}{2}\right) + T\left(\frac{n}{3}\right) + c_2 n \leq c \frac{n}{2} + c \frac{n}{3} + c_2 n \\ &= \left(\frac{5}{6} c + c_2\right) n \end{aligned}$$

取 $c = \frac{5}{6} c + c_2$ ，即 $c = 6c_2$ 时， $T(n) \leq c n$ 。

所以，取 $c = \max\{c_1, 6c_2\}$ ， $n_0=1$ ，当 $n > n_0$ 时， $T(n) \leq c n$ 。

即： $T(n)=O(n)$



线性时间选择

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

```
private static Comparable randomizedSelect(int p,int r,int k)
{
    if (p==r) return a[p];
    int i=randomizedpartition(p,r),
        j=i-p+1;
    if (k<=j) return randomizedSelect(p,i,k);
    else return randomizedSelect(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

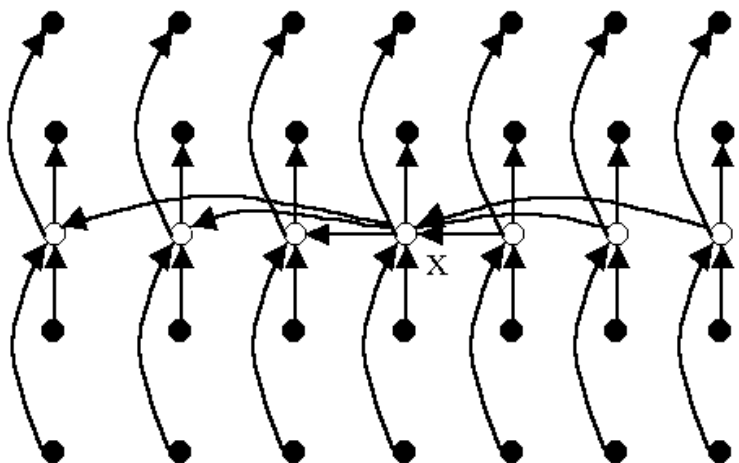
如果能在线性时间内找到一个划分基准，使得按这个基准所划分出的2个子数组的长度都至少为原数组长度的 ε 倍($0 < \varepsilon < 1$ 是某个正常数)，那么就可以在最坏情况下用 $O(n)$ 时间完成选择任务。

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。用任何一种排序算法，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 $n \geq 75$ 时， $3(n-5)/10 \geq n/4$ 所以按此基准划分所得的2个子数组的长度都至少缩短 $1/4$ 。



```
private static Comparable select (int p, int r, int k)
{
    if (r-p<5) { //用某个简单排序算法对数组a[p:r]排序;
        bubbleSort(p,r);
        return a[p+k-1];
    }
}
```

复杂度分析

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

$$T(n) = O(n)$$

```
for (int j=0;j<3;j++) bubble(s,t-j);
```

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。

```
else return select(l+1,r,k-j),
```

```
}
```

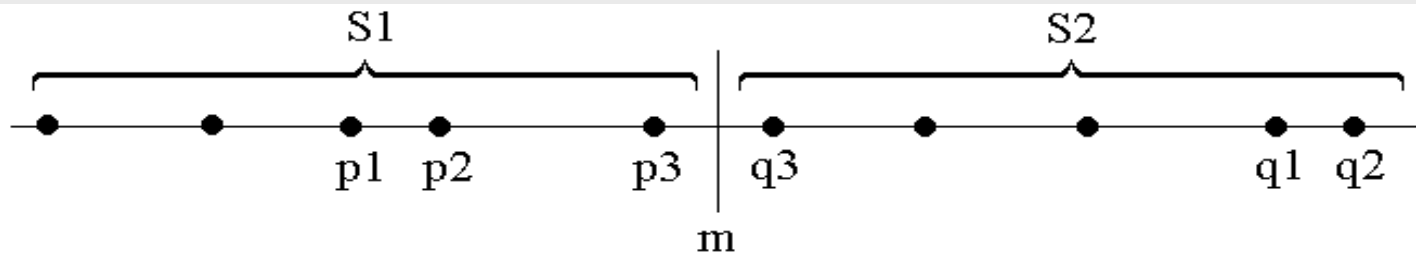


最接近点对问题

给定平面上 n 个点的集合 S ，找其中的一对点，使得在 n 个点组成的所有点对中，该点对间的距离最小。

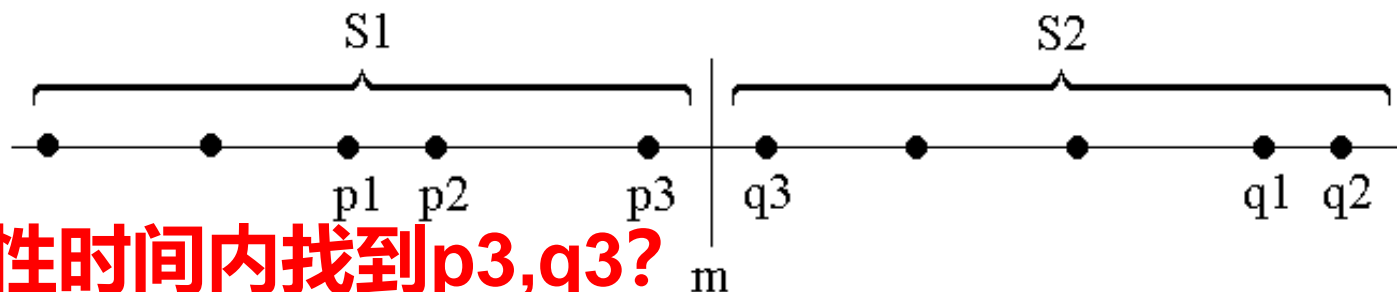
◆为了使问题易于理解和分析，先来考虑**一维**的情形。此时， S 中的 n 个点退化为 x 轴上的 n 个实数 x_1, x_2, \dots, x_n 。最接近点对即为这 n 个实数中相差最小的2个实数。

- 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 $d = \min\{|p_1 - p_2|, |q_1 - q_2|\}$ ， 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 p_3, q_3 ?

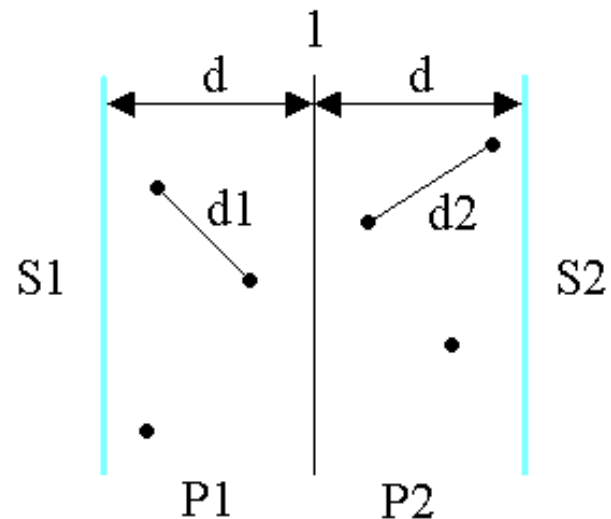
- ◆如果S的最接近点对是 $\{p_3, q_3\}$ ，即 $|p_3 - q_3| < d$ ，则 p_3 和 q_3 两者与 m 的距离不超过 d ，即 $p_3 \in (m-d, m]$ ， $q_3 \in (m, m+d]$ 。
- ◆由于在 S_1 中，每个长度为 d 的半闭区间至多包含一个点（否则必有两点距离小于 d ），并且 m 是 S_1 和 S_2 的分割点，因此 $(m-d, m]$ 中至多包含S中的一个点。由图可以看出，如果 $(m-d, m]$ 中有S中的点，则此点就是 S_1 中最大点。
- ◆因此，我们用线性时间就能找到区间 $(m-d, m]$ 和 $(m,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

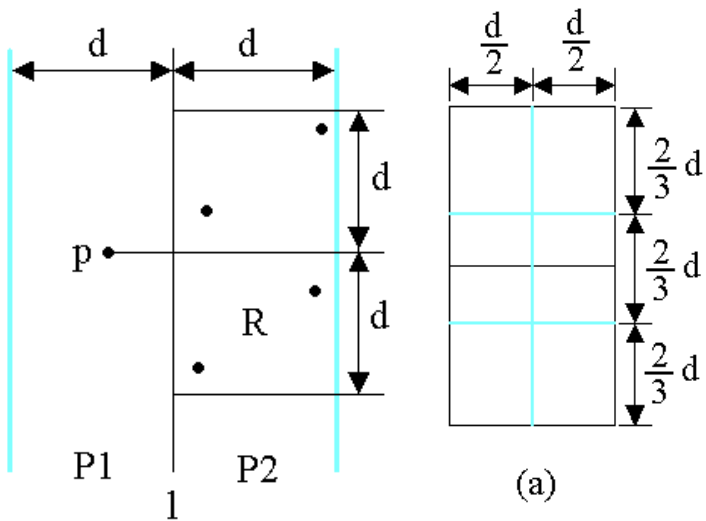
- 选取一垂直线 $l:x=m$ 来作为分割直线。其中 m 为 S 中各点 x 坐标的中位数。由此将 S 分割为 S_1 和 S_2 。
- 递归地在 S_1 和 S_2 上找出其最小距离 d_1 和 d_2 ，并设 $d=\min\{d_1, d_2\}$ ， S 中的最接近点对或者是 d ，或者是某个 $\{p, q\}$ ，其中 $p \in P_1$ 且 $q \in P_2$ 。
- 能否在线性时间内找到 p, q ?





最接近点对问题

- ◆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 $\text{distance}(p, q) < d$ 。满足这个条件的P2中的点一定落在一个 $d \times 2d$ 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 $6 \times n/2 = 3n$ 个候选者



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

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

$\text{distance}(u, v) < d$ 。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者。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

```
public static 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

复杂度分析

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

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

中与其
成合

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;
```

```
}
```



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 |



总结

- 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。
- 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。