

分治法

(Divide and Conquer)



内容

■ 分治法的基本思想

■ 应用

- 找出伪币
- 五个元素比较
- 计算 A^n
- 二分搜索算法
- 金块问题(最大最小问题)
- 大整数的乘法
- 矩阵乘法
- 棋盘覆盖

■ 快速排序

- 寻找第 k 小元素
- 中间的中间规则
- 最接近点对问题
- 循环赛日程表

■ 复杂性的下限

- 最大最小问题的下限
- 排序算法的下限



分治法的步骤

Merge-Sort 是用分治法设计算法的范例。

在分治法中，我们**递归**地求解一个问题，在每层递归中应用如下三个步骤：

- **分解(Divide)**：将问题划分为一些子问题，子问题的形式与原问题一样，只是规模更小。
- **解决(Conquer)**：递归地求解出子问题。如果子问题的规模足够小，则停止递归，直接求解。
- **合并(Combine)**：将子问题的解组合成原问题的解。

有时，除了与原问题形式完全一样的规模更小的子问题外，还要求解与原问题不完全一样的子问题。我们将这些子问题的求解看做合并步骤的一部分。



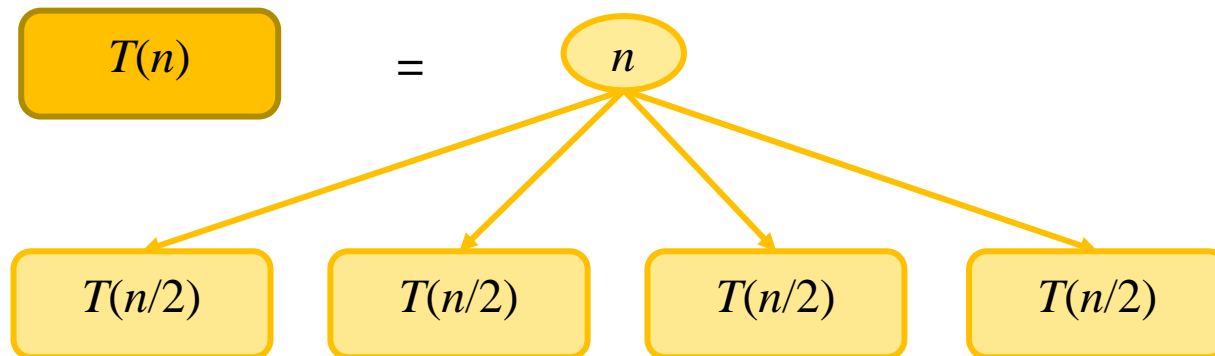
分治法的特征

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；
 - 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。
- 最后一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。

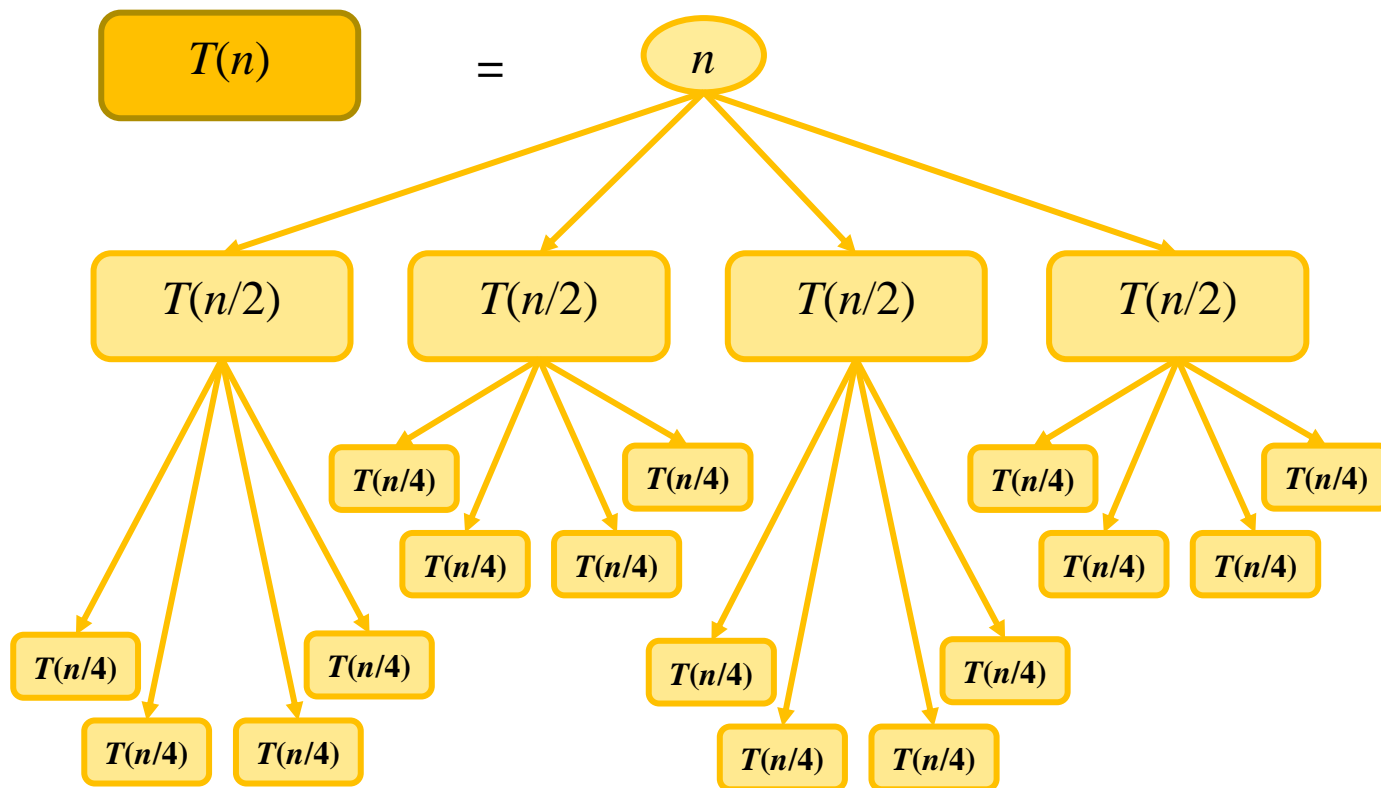
分治法的总体思想

将要求解的较大规模的问题分割成 k 个更小规模的子问题。



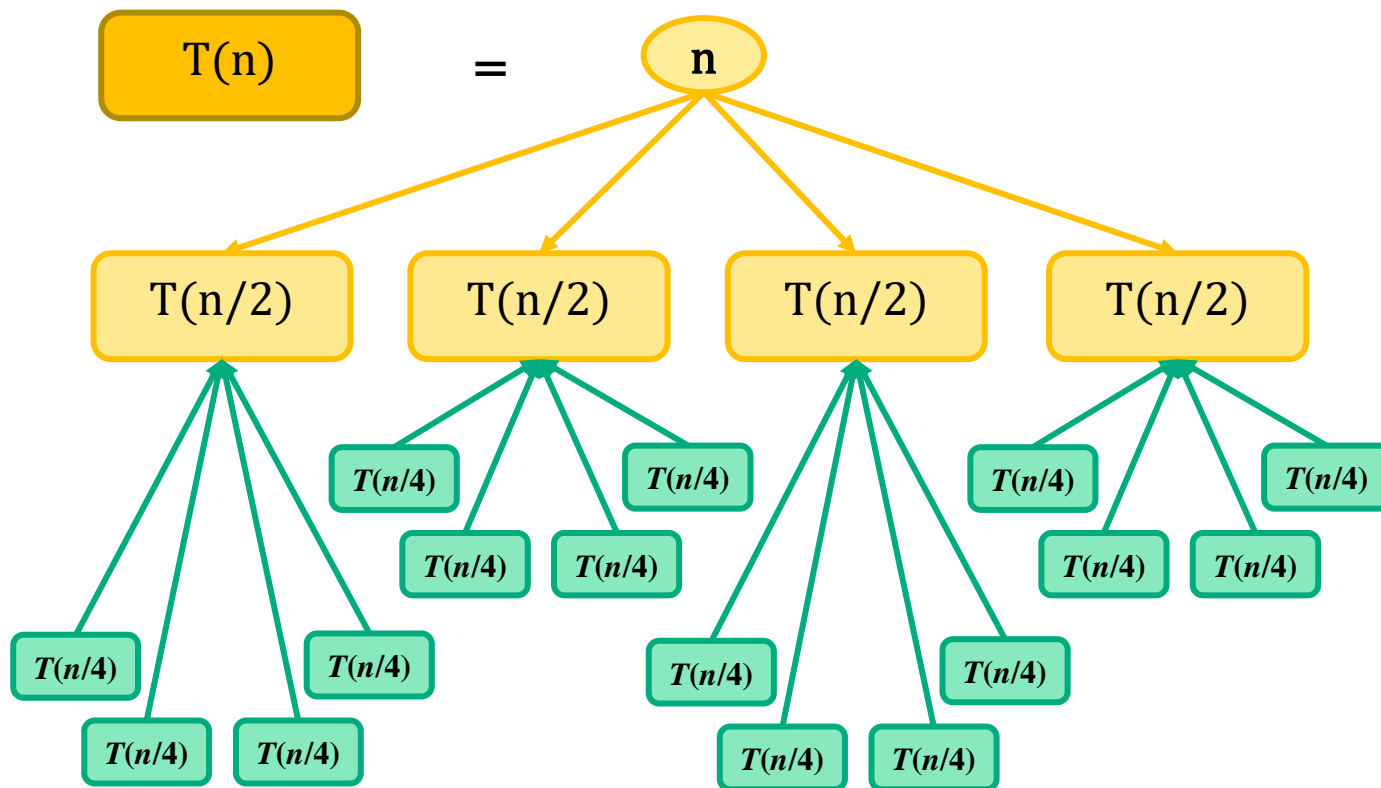
分治法的总体思想

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。



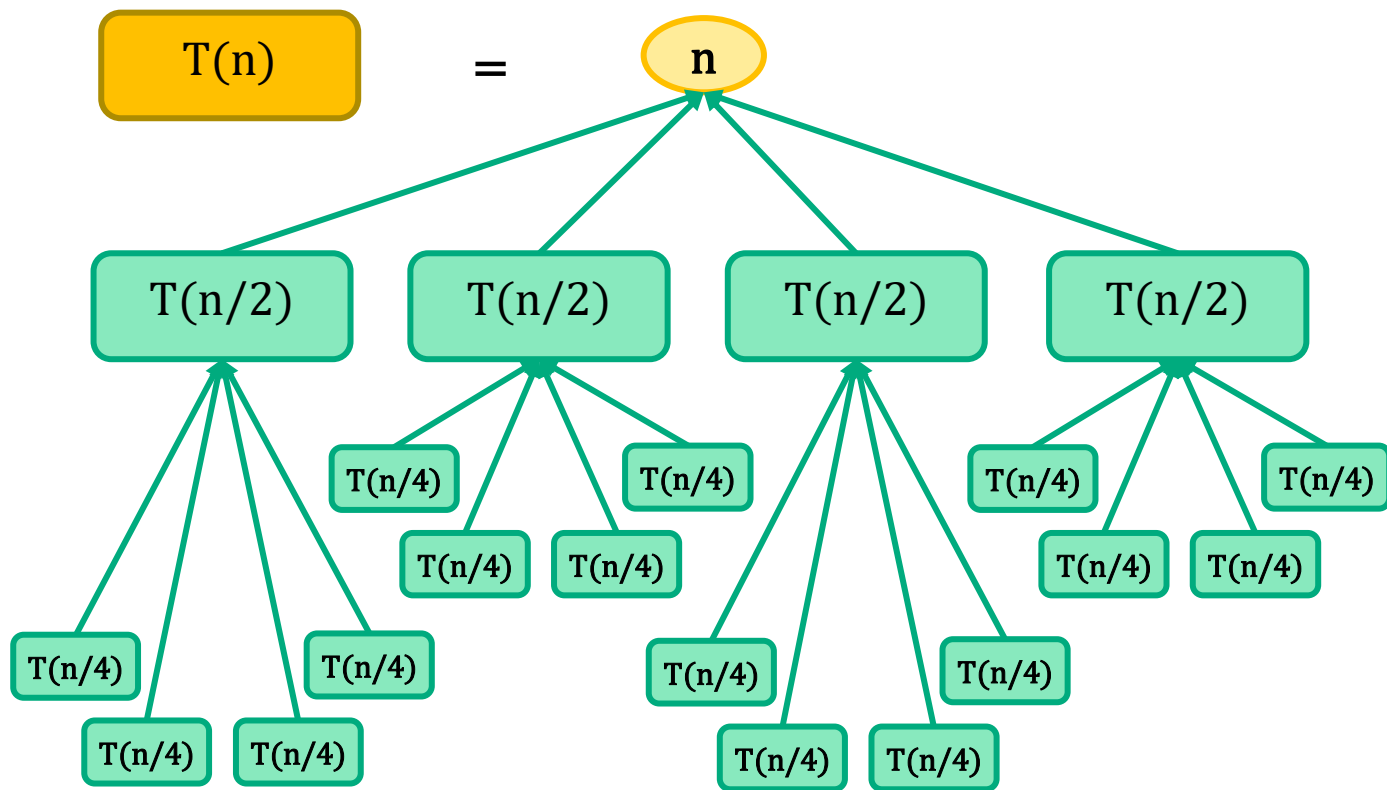
分治法的总体思想

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。



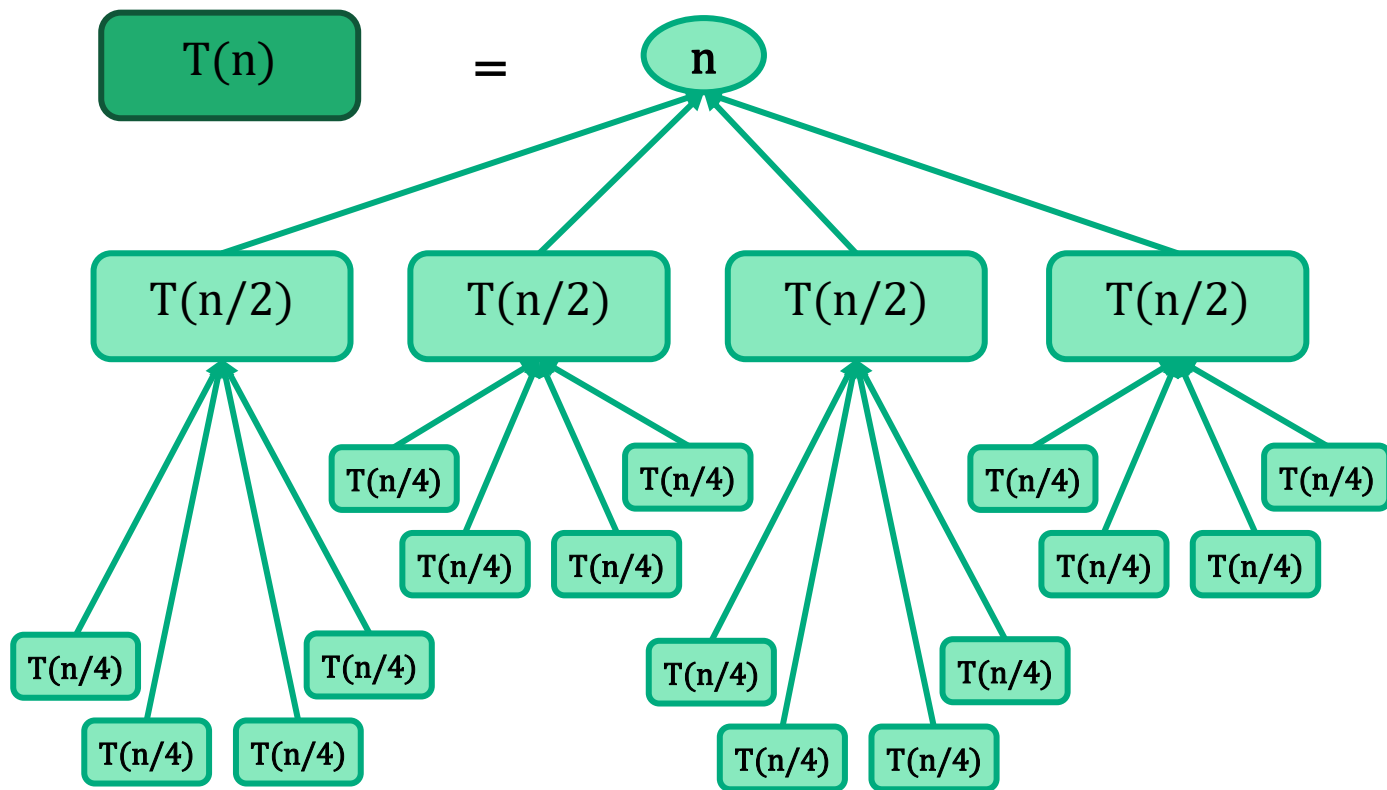
分治法的总体思想

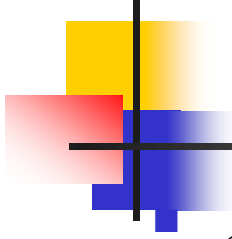
将求出的小规模的问题的解合并为一个更大规模的问题的解，自底向上逐步求出原来问题的解。



分治法的总体思想

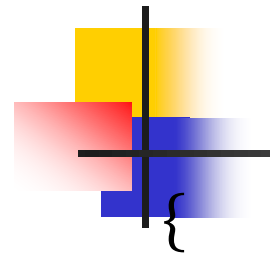
将求出的小规模的问题的解合并为一个更大规模的问题的解，自底向上逐步求出原来问题的解。





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。



```
{  
    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  
}
```



分治法常常得到递归算法

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。



例1 找出伪币

■ 现有 16 个硬币,其中有一个是伪币, 并且伪币重量比真币轻。试用一台天平找出这枚伪币。

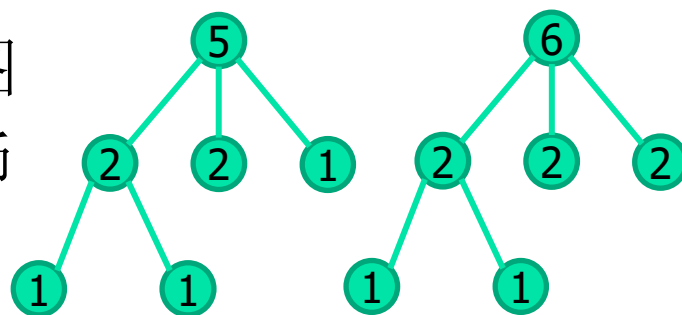
- 两两比较,最坏情形需 8 次比较找出伪币。
- 分治法仅需 4 次比较找出伪币。
- 还可以用更少的次数找出伪币吗?

例1 找出伪币（续）

16 个伪币分为 3 组：5 个、5 个、6 个。用天平称量两组 5 个硬币的，轻者有伪币。同样重则伪币在 6 个组。

(一次比较)

- 将伪币组再次分 3 组，有如右图两种情况。取其中两组 2 个硬币的称量，找出伪币组。



(二次比较)

- 最坏情况下，再比较一次可找出伪币。

(三次比较)

- 合理的分解子问题可以提高效率。



5个元素排序

- 若使用插入排序，最坏情况需要 10 次比较($1+2+3+4$);
- 若使用归并排序，最坏情况需要 9 次比较($n\log n$);
- 7 次比较：
 - 标记待排序元素为 A、B、C、D、E
 - 先将 A 与 B、C 与 D 比较，假设 $A < B$ 、 $C < D$ 。再将 A 与 C 比较，不妨假设 $A < C$ ，则 $A < C < D$ ，当前共三次比较。
 - 再将 E 折半插入 $A < C < D$ ，最多需要 2 次比较 (E 先与 C 比较)。
 - 假设结果为 $A < C < D < E$ ，再将 B 插入 $A < C < D < E$ ，已有条件 $A < B$ ，B 只需插入 A 之后 $C < D < E$ 中，折半最多需要两次比较。



5个元素找中间元

最坏情况下只需要 6 次比较: a, b, c, d, e

- 先令 a 与 b 比较, c 与 d 比较, 假设 $a > b, c > d$;
(两次比较)
- 令 a 与 c 比较: (一次比较)
 - 若 $a > c$, 则 $a > b$ 且 $a > c > d$, a 不可能是中位数;
 - 若 $a < c$, 则 $c > d$ 且 $c > a > b$, c 不可能是中位数。
- 不妨设 $a > c$ 。现在问题转化为在 4 个元素 b, c, d, e 中找一个次大的, 且有条件 $c > d$ 。
- 令 b 与 e 比较; 不妨设 $b > e$ 。(一次比较)
- 令 b 与 c 比较, 大者为四个元素中最大的。不妨设 $b > c$ 。
(一次比较)
- 令 e 与 c 比较, 二者中较大的为四个元素中次大的。



例2 Compute a^n

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

■ Naïve algorithm: $\Theta(n)$

■ Divide-and Conquer algorithm:

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

■ $T(n) = T(n/2) + \Theta(1)$

■ $a = 1, b = 2, \log_b a = 0, n^0 = \Theta(1) = f(n)$ 。

■ 根据主定理, $T(n) = \Theta(\lg n)$ 。

Fibonacci数列的分治算法

■ Fibonacci数列 $F_n = F_{n-1} + F_{n-2}$: 1, 1, 2, 3, 5, 8, 13, 21

■ Fibonacci数列的分治解法:

$$\begin{pmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n$$

■ $T(n) = \Theta(\lg n)$, 为什么?



例3 二分搜索算法

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：

1. 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。



例3 二分搜索算法

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：

2. 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。



例3 二分搜索算法

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：

3. 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。满足分治法的第四个适用条件。



例3 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;  
    }  
}
```

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。
- 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 $O(\log n)$ 。



例4 金块问题-最大最小问题

问题：有若干金块，试用一台天平找出其中最轻和最重的金块。

- 相当于在 n 个数中找出最大和最小的数.
- 直接求解：
 - 先找出最大值($n-1$ 次比较), 然后在剩下的 $n-1$ 个数中再找出最小值($n-1$ 次比较), 共需 $2n-3$ 次比较.
 1. $\text{min} \leftarrow a[0]; \text{max} \leftarrow a[0];$
 2. for $i \leftarrow 1$ to $n-1$ do
 3. if $a[i] < \text{min}$
 4. $\text{min} \leftarrow a[i]$
 5. else if $a[i] > \text{max}$ $\text{max} \leftarrow a[i]$

分治法求解最大最小问题

分治法思想：将金块分成两组，每组找出最大和最小。然后将每组的最大和最大进行比较，最小和最小进行比较。

1. $\text{Max-min}(A[0, n-1], \text{max}, \text{min})$
2. if $n < 1$ $\text{max} \leftarrow \text{min} \leftarrow a[0]$, return;
3. else $m \leftarrow n/2$
4. $\text{Max-min}(A[0, m-1], \text{max1}, \text{min1})$,
5. $\text{Max-min}(A[m, n-1], \text{max2}, \text{min2})$,
6. $\text{max} \leftarrow \text{max}(\text{max1}, \text{max2})$,
7. $\text{min} \leftarrow \text{min}(\text{min1}, \text{min2})$,
8. return.

复杂度分析

设 $C(n)$ 为使用分治法所需要的比较次数。假设 $n=2^k$ 则有:

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

- 迭代展开可得

$$C(n) = 2^{k-1}c(2) + \sum_{1 \leq i \leq k-1} 2^i = (3n/2) - 2$$

- 分治法比直接求解法少用了25%次比较。
- $a = 2, b = 2, \log_b a = 1, n^1 = n, f(n) = 2 = \Theta(1)$ 。
- 根据主定理, $T(n) = \Theta(n)$ 。

最大最小问题分治解法的最优性

定理： n 个元素同时找出最大最小元，至少需要 $(3n/2) - 2$ 次比较。即分治法是解决最大最小问题的最优算法。

证明：

- 假设 n 个元素互不相同。首先设置 4 个状态：
 - A: 未比较过的元素，个数为 a ;
 - B: 没有输过的元素，即最大，个数为 b ;
 - C: 没有赢过的元素，即最小，个数为 c ;
 - D: 赢过也输过，就是中间元素，个数为 d 。
- 算法状态可用元组 (a, b, c, d) 来描述：
 - 算法启动时：起始状态为 $(n, 0, 0, 0)$;
 - 算法结束时：完成状态为 $(0, 1, 1, n-2)$ 。

最大最小问题分治解法的最优性

定理： n 个元素同时找出最大最小元，至少需要 $(3n/2) - 2$ 次比较。

证明(续)：

- 当 A 中的两个元素比较时，
 - 较小的元素放入 C ，较大的元素放入 B ，
 - 引起下面状态转换： $(a, b, c, d) \rightarrow (a-2, b+1, c+1, d)$ 。
- a 从 n 减至 0 ，至少要做 $\lceil n/2 \rceil$ 的比较次数：
- B 中或 C 中元素自己比较，每次比较 D 中元素增加 1 个。
- d 从 0 增至 $n - 2$ ，至少要做 $n - 2$ 次比较；
- 因此，任何求最大最小的算法从初始状态到完成状态所用比较次数不可少与 $\lceil n/2 \rceil + (n-2) = \lceil 3n/2 \rceil - 2$ 次比较。证毕



最大最小问题的非递归程序

```
template<class T>
bool MinMax(T w[], int n, T& Min, T& Max)
{ // 寻找w[0:n-1]中的最小和最大值
  // 如果少于一个元素，则返回false
    // 特殊情况： n <= 1
    if (n < 1) return false;
    if (n == 1) {Min = Max = 0;
                  return true;}

    // 对Min 和Max进行初始化
    int s; // 循环起点
```



最大最小问题的非递归程序

```
if (n % 2) { // n 为奇数
    Min = Max = 0;
    s = 1;}
else { // n为偶数, 比较第一对
    if (w[0] > w[1]){
        Min = 1;
        Max = 0;}
    else {Min = 0;
        Max = 1;}
    s = 2;}
```



最大最小问题的非递归程序

// 比较余下的数对

```
for (int i = s; i < n; i += 2){
```

```
    // 寻找w[i] 和w[i+1]中的较大者
```

```
    // 然后将较大者与 w[Max]进行比较
```

```
    // 将较小者与 w[Min]进行比较
```

```
    if (w[i] > w[i+1]){
```

```
        if (w[i] > w[Max]) Max = i;
```

```
    else {
```

```
        if (w[i+1] > w[Max]) Max = i + 1;
```

```
        if (w[i] < w[Min]) Min = i;}
```

```
    }
```

```
return true;
```

```
}
```

例5 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。

- 直接方法： $O(n^2)$

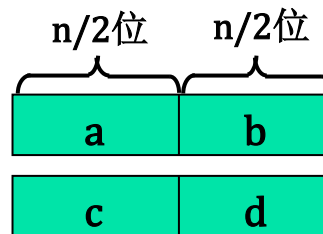
- 效率低！ 有没有改进的方法？

- 分治法： $X = a \cdot 2^{n/2} + b$ $Y = c \cdot 2^{n/2} + d$

$$X \cdot Y = ac \cdot 2^n + (ad + bc) \cdot 2^{n/2} + bd$$

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

- 没有改进！



例5 大整数的乘法

为了降低时间复杂度，必须减少乘法的次数：

$$X \cdot Y = ac \cdot 2^n + [(a-b)(d-c) + ac + bd]2^{n/2} + bd$$

或者

$$X \cdot Y = ac \cdot 2^n + [(a+b)(d+c) - ac - bd]2^{n/2} + bd$$

- $$T(n) = \begin{cases} O(1) & n=1 \\ 3T(n/2) + O(n) & n>1 \end{cases}$$
$$= O(n^{\log 3}) = O(n^{1.59})$$

- **细节问题：**两个 $X \cdot Y$ 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，是问题的规模变大，故不选择第 2 中方案。
- 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。能否找到线性时间的算法？

例6 矩阵乘法($C_{n \times n} = A_{n \times n} \cdot B_{n \times n}$)

传统方法： $c_{ij} = \sum a_{ik} b_{kj} \ (1 \leq k \leq n)$ 。每计算C的一个元素 c_{ij} ，需要做 n 次乘法和 $n - 1$ 次加法。 $T(n) = \Theta(n^3)$ 。

- 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 $C = A \cdot B$ 重写为：

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \cdot \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} \Leftrightarrow \begin{cases} C_{11} = A_{11}B_{11} + A_{12}B_{21} \\ C_{12} = A_{11}B_{12} + A_{12}B_{22} \\ C_{21} = A_{21}B_{11} + A_{22}B_{21} \\ C_{22} = A_{21}B_{12} + A_{22}B_{22} \end{cases}$$

- $T(n) = 8T(n/2) + \Theta(n^2) = \Theta(n^3)$ 。 **8次乘法，4次加减法**
- 对策：要么减小a，要么增大b。f(n)?

Strassen方法

为了降低复杂度，在不改变矩阵分块的情况下，可以减小递归函数前的系数：

$$M_1 = A_{11}(B_{12}-B_{22})$$

$$M_2 = (A_{11}+A_{12})B_{22}$$

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

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

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

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

$$M_7 = (A_{11}-A_{22})(B_{11}+B_{12})$$

7次乘法，18次加减法

$$\Leftrightarrow \begin{cases} C_{11} = M_5 + M_4 - M_2 + M_6 \\ C_{12} = M_1 + M_2 \\ C_{21} = M_3 + M_4 \\ C_{22} = M_5 + M_1 - M_3 - M_7 \end{cases}$$

- $T(n) = 7T(n/2) + \Theta(n^2) = \Theta(n^{\log_2 7}) \approx \Theta(n^{2.81})$
- 因为 $\Theta(n^2)$ 的实际系数比较高，所以当 $n \geq 30$ 时Strassen方法才比传统方法快！



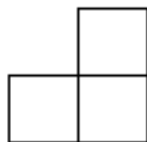
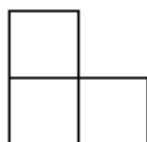
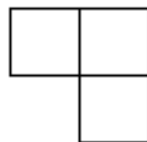
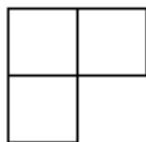
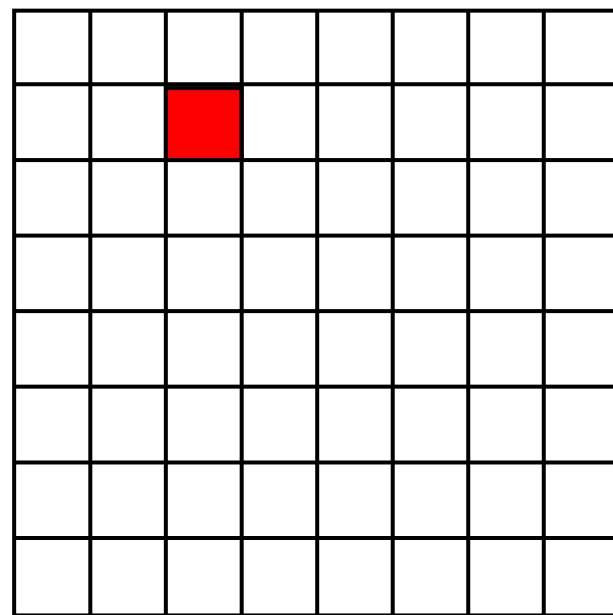
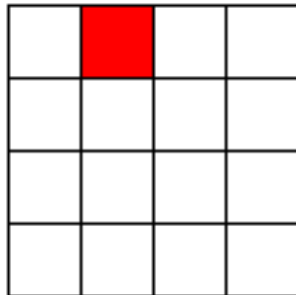
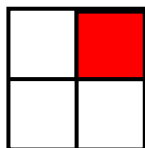
例6 矩阵乘法

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 $O(n^{2.376})$ 。
Don Coppersmith, Shmuel Winograd, Matrix Multiplication via Arithmetic Progressions, STOC 1987: 1-6.
- 是否能找到 $O(n^2)$ 的算法？目前为止还没有结果。

例7 棋盘覆盖

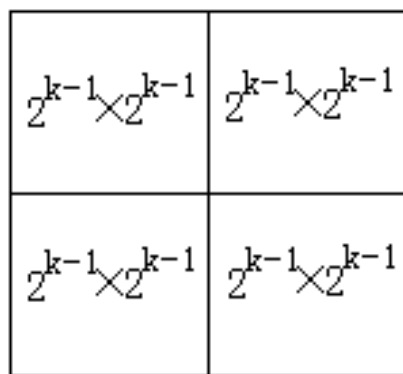
在一个 $2^k \times 2^k$ 个方格组成的棋盘中，恰有一个方格与其它方格不同，称该方格为一特殊方格，且称该棋盘为一特殊棋盘。在棋盘覆盖问题中，要用图示的4种不同形态的L型骨牌覆盖给定的特殊棋盘上除特殊方格以外的所有方格，且任何2个L型骨牌不得重叠覆盖。



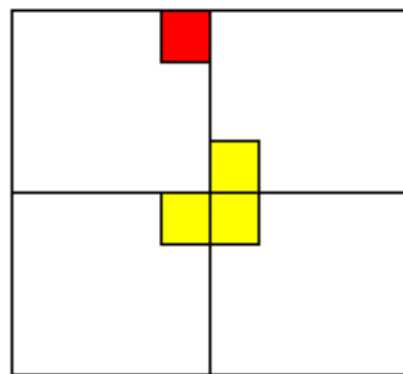
例7 棋盘覆盖

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个较小的子棋盘之一中，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 2×2 。



(a)



(b)

例7 棋盘覆盖

覆盖一个 2×2 棋盘需 $1 = 4^0$ 个 L 型骨牌，覆盖一个 4×4 棋盘需 $4 \times 1 + 1 = 4^1 + 4^0$ 个 L 型骨牌，覆盖一个 8×8 棋盘需 $4(4 \times 1 + 1) + 1 = 4^2 + 4^1 + 4^0$ 个 L 型骨牌，……，覆盖一个 $2^k \times 2^k$ 棋盘所需的 L 型骨牌个数为

$$\sum_{0 \leq i \leq k-1} 4^i = (4^k - 1) / 3$$

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

$$= 4^k T(0) + O(4^{k-1}) = O(4^k)$$

- 亦即，把一个有 $n=4^k$ 个方格组成棋盘全部覆盖，需要 $(n-1)/3$ 个 L 型骨牌，时间复杂度为 $T(n)=4T(n/4)+c=O(n)$.



```
void chessBoard(int tr, int tc, int dr, int dc, int size)
```

```
{
    if (size == 1) return;
    int t = tile++, // L型骨牌个数
        s = size/2; // 子棋盘大小
    // 覆盖左上角子棋盘
    if (dr < tr + s && dc < tc + s)
        // 特殊方格在此子棋盘中
        chessBoard(tr, tc, dr, dc, s);
    else { // 此子棋盘中无特殊方格
        // 用 L型骨牌覆盖右下角
        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)
        // 特殊方格在此子棋盘中
        chessBoard(tr, tc+s, dr, dc, s);
    else { // 此子棋盘中无特殊方格
        // 用 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)
        // 特殊方格在此子棋盘中
        chessBoard(tr+s, tc, dr, dc, s);
    else { // 用 L型骨牌覆盖右上角
        board[tr + s][tc + s - 1] = t;
        // 覆盖其余方格
        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)
        // 特殊方格在此子棋盘中
        chessBoard(tr+s, tc+s, dr, dc, s);
    else { // 用 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);
    }
}

void OutputBoard(int size){
    for (int i = 0; i < size; i++){
        for (int j = 0; j < size; j++){
            cout << setw(5) << Board[i][j];
            cout << endl;
        }
    }
}
```



Tony Hoare

- 1934年在英国出生，1956年获得牛津大学的哲学、古典语言与文学学位。
- 1960年提出了**快速排序算法**(1958年，当Hoare忙着为人们“用机器自动实现语言翻译”的梦想奋斗时。他发现，以当时的技术，需要先把字典录在磁带上，而每翻译一句话，都需要先从磁带的词典中找出每一个词。那么，一句话里面有十几个词，一段文章里面有数百上千个单词，且不论翻译的质量，光查词就已经慢得让人无法忍受了.....)
- 领导开发了经典编程语言Algol 60的早期编译器。
- 1969年发表了论文“计算机程序的公理基础”，提出了名为Hoare逻辑的形式系统理论。
- 1999年从牛津大学退休后，在微软剑桥研究院工作。



Tony Hoare的获奖情况

- 1980年图灵奖；
- 1981年获得AFIPS的Harry Goode奖；
- 1985年获得英国IEE的法拉第奖章；
- 1990年被IEEE授予计算机先驱奖。
- 2000年京都奖的获得者；
- 2000年因为其在计算机科学与教育上做出的贡献被封为爵士。
- 2005年当选英国皇家工程院院士；
- 2011年IEEE John von Neumann Medal；
-



Null references: the billion dollar mistake ——Tony Hoare

2009年3月Tony Hoare在Qcon技术会议上发表了题为“Null引用：代价十亿美元的错误”的演讲，回忆自己1965年设计第一个全面的类型系统时，未能抵御住诱惑，加入了Null引用，仅仅是因为实现起来非常容易。它后来成为许多程序设计语言的标准特性，导致了数不清的错误、漏洞和系统崩溃，可能在之后40年中造成了十亿美元的损失。他在同月出版Communications of the ACM中表示，如何证明程序的正确性仍然是计算机科学中有待解决的重大课题。



Tony Hoare的论著

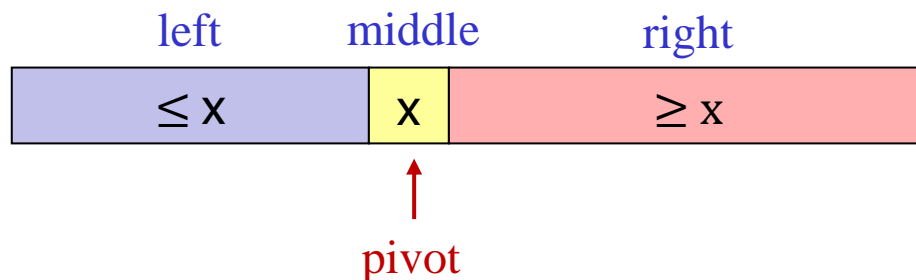
发表过许多高水平的论著。

- ACM在1983年评选出最近25年中发表在Communications of the ACM上的有里程碑式意义的25篇经典论文，只有2名学者各有2篇论文入选，霍尔就是其中之一。
- 1972年他与O.J. Dahl和E. W. Dijkstra三位图灵奖得主合著的Structured Programming一书，更是难以逾越的高峰。

例8 快速排序

快速排序采用了分治的思想：

- n 个元素被分成三段：左段 left, 右段 right 和中段 middle.
- 中段 middle 只包含一个元素，称为支点 (pivot)。
- 左段 left 中的元素全部小于或等于 pivot;
- 右段 right 中的元素全部大于或等于 pivot。
- 这样 left 和 right 中的元素可以独立排序，不需要归并。



- 整个过程可以分为3个步骤：分、治、归并。



例8 快速排序

```
1. Partition (A, p, r) //A[p..r`]  
2.   x ← A[p]  // pivot = A[p]  
3.   i ← p  
4.   for j ← p+1 to q  
5.     do if A[j] ≤ x  
6.       then exchange A[i] ↔ A[j]  
7.         i ← i+1  
8.   exchange A[p] ↔ A[i]  
9.   return i
```

Example of Partitioning

给8个数排序。选左边第一个数做支点。 $i = 1, j = 2$

6	10	13	5	8	3	2	11
i	j						

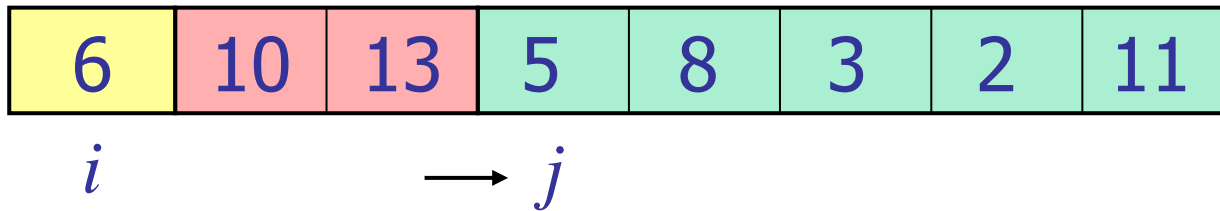
- $A(2) > x$, 继续向右查找, $j = 3$

6	10	13	5	8	3	2	11
i		j					

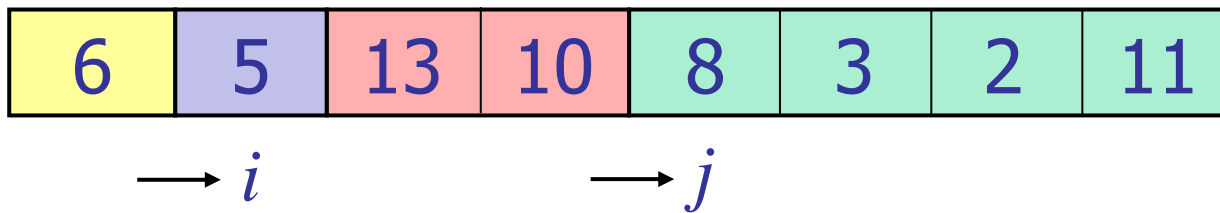
- $A(3) > x$, 继续向右查找, $j = 4$

6	10	13	5	8	3	2	11
i			j				

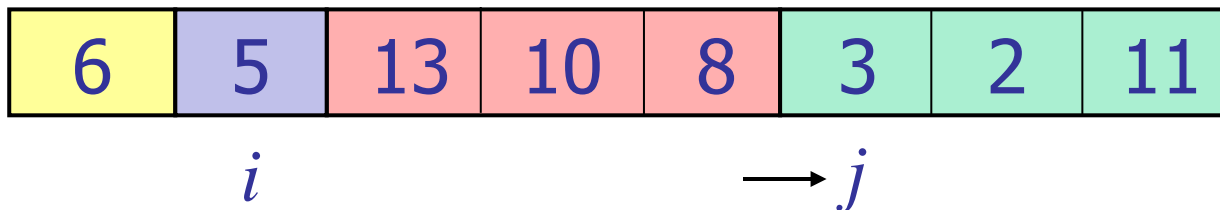
Example of Partitioning



- $A(4) < x$, $i = 2$, $A[2] \leftrightarrow A[4]$, $j = 5$



- $A(5) > x$, 继续向右查找, $j = 6$



Example of Partitioning

6	5	13	10	8	3	2	11
---	---	----	----	---	---	---	----

i

$\longrightarrow j$

- $A(6) < x$, $i = 3$, $A[6] \leftrightarrow A[3]$, $j = 7$

6	5	3	10	8	13	2	11
---	---	---	----	---	----	---	----

$\longrightarrow i$

$\longrightarrow j$

- $A(7) < x$, $i = 4$, $A[7] \leftrightarrow A[4]$, $j = 8$

6	5	3	2	8	13	10	11
---	---	---	---	---	----	----	----

$\longrightarrow i$

$\longrightarrow j$

Example of Partitioning

6	5	3	2	8	13	10	11
---	---	---	---	---	----	----	----

- $A(8) > x$, 循环结束。 $\overrightarrow{i} A[1] \leftrightarrow A[i]$, return \overrightarrow{j}

2	5	3	6	8	13	10	11
---	---	---	---	---	----	----	----

i

$\longrightarrow j$



例8 快速排序

1. QuickSort (A, p, r)

2. **if** $p < r$

3. **then** $q \leftarrow \text{Partition}(A, p, r)$

4. QuickSort (A, p, $q-1$)

5. QuickSort (A, $q+1$, r)

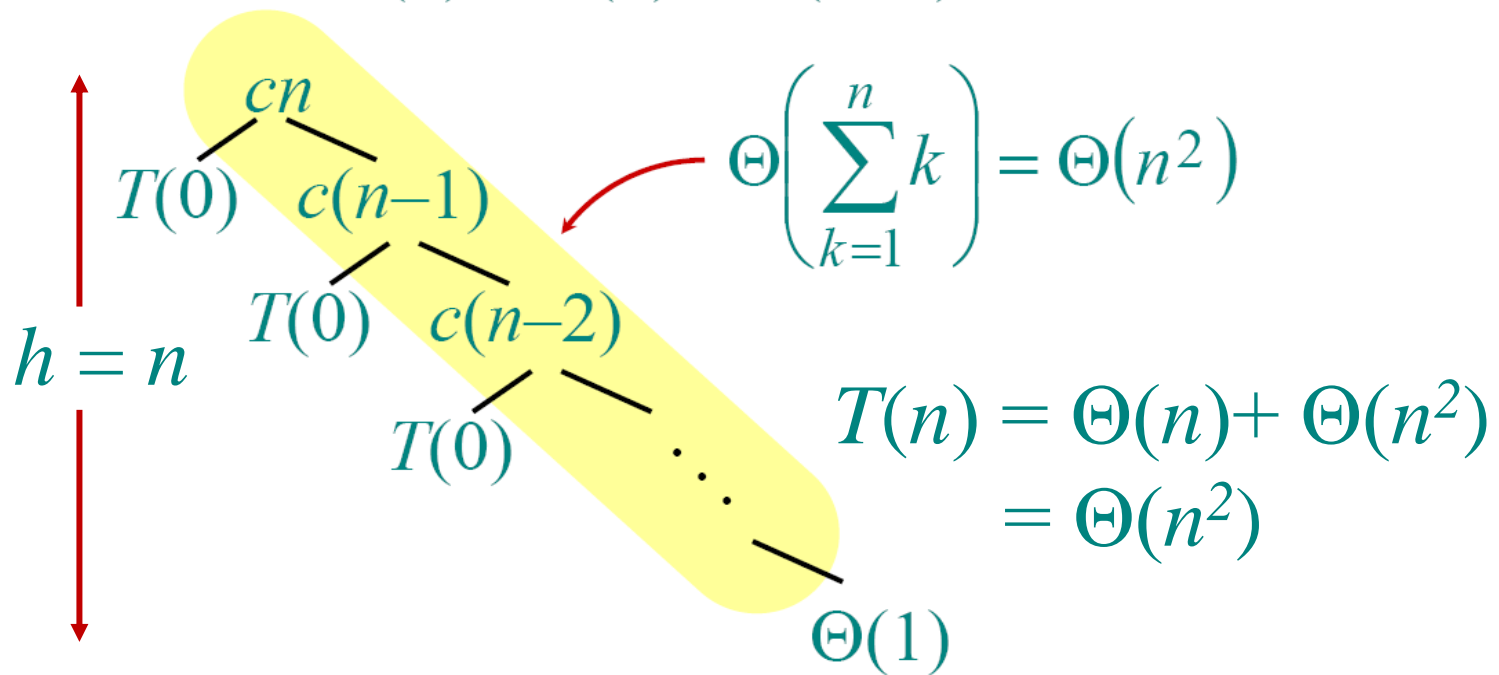
■ **Initial call:** QuickSort (A, 1, n)

■ 对于输入序列 $A[p..r]$, 算法Partition的计算时间为 $O(r-p-1)$.

例8 快速排序 — 最坏情况

Worst-case recursion tree

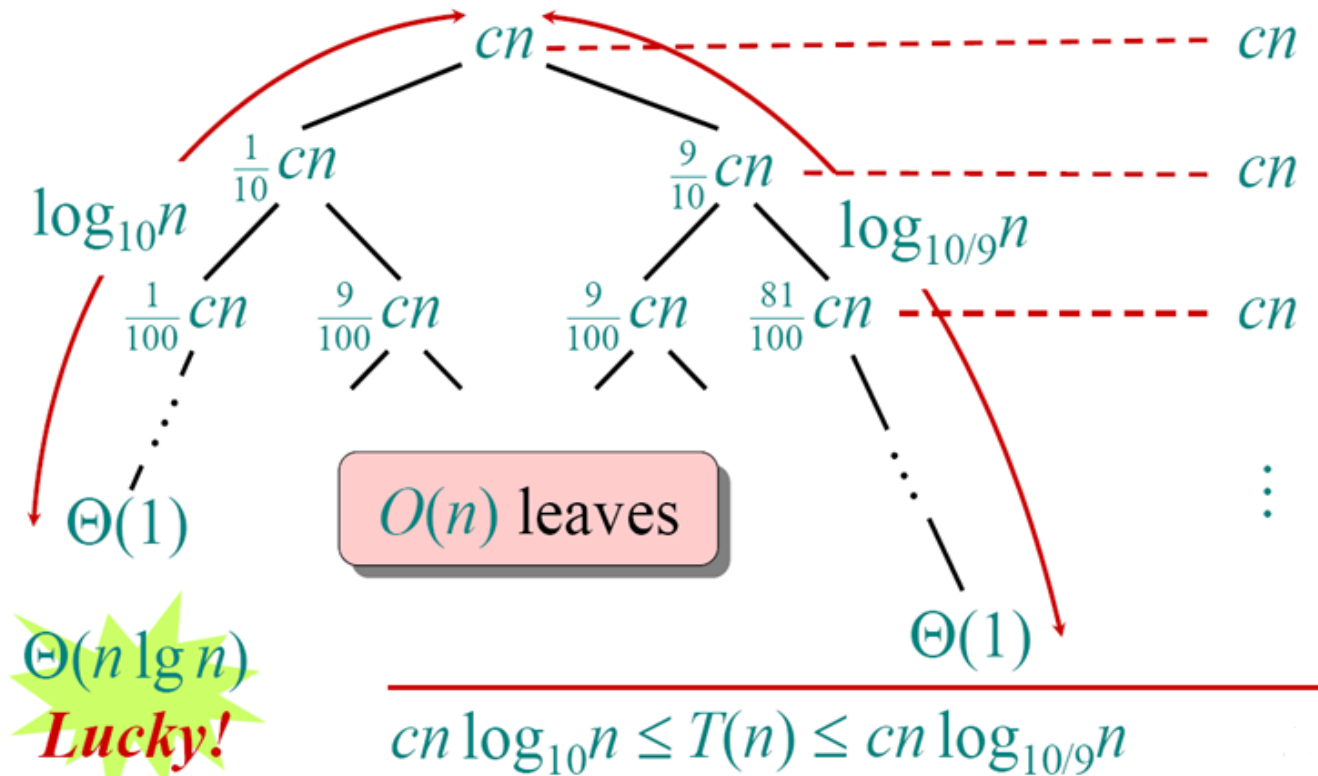
$$T(n) = T(0) + T(n-1) + cn$$



- 最坏情况: $T(n) = T(0) + T(n-1) + O(n) = O(n^2)$

例8 快速排序 — 最好情况

Analysis of “almost-best” case



- 最好情况: $T(n) = 2T(n/2) + O(n) = O(n \log n)$

例8 快速排序 — 平均情况

平均情况下， n 个元素中每一个元素都有 $1/n$ 的概率被选为基准点。当 $n > 1$ 时，时间复杂度函数可以写为：

$$T(n) = cn + \frac{1}{n} \sum_{s=0}^{n-1} [T(s) + T(n-s-1)]$$

- 假定关键字彼此不同，则每回合平均关键字比较次数 $cn \leq n+1$ ，又因为

$$\sum_{0 \leq s \leq n-1} T(n-s-1) = \sum_{0 \leq s \leq n-1} T(s),$$

有

$$nT(n) = n(n+1) + 2 \sum_{s=0}^{n-1} T(s)$$

类似的，有

$$- (n+1)T(n+1) = (n+1)(n+2) + 2 \sum_{s=0}^n T(s)$$

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

例8 快速排序 — 平均情况

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

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

进而,

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

- 令 $f(n) = T(n)/(n+1)$, 有:

$$f(n) = f(n-1) + 2/(n+1) = f(1) + \sum_{2 \leq k \leq n} (2/k)$$

- 调和级数收敛到 $\log_e n$, 所以

$$T(n) = (n+1)f(n) = \Theta(n \log_2 n)$$

例8 快速排序 — 支点选择

快速排序算法的性能取决于划分的对称性。最好情况是每次把待排序数组对分：

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

- 分点落在 $(n/10, 9n/10)$, 我们也有

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

运气也很好！（上式中应取整！）

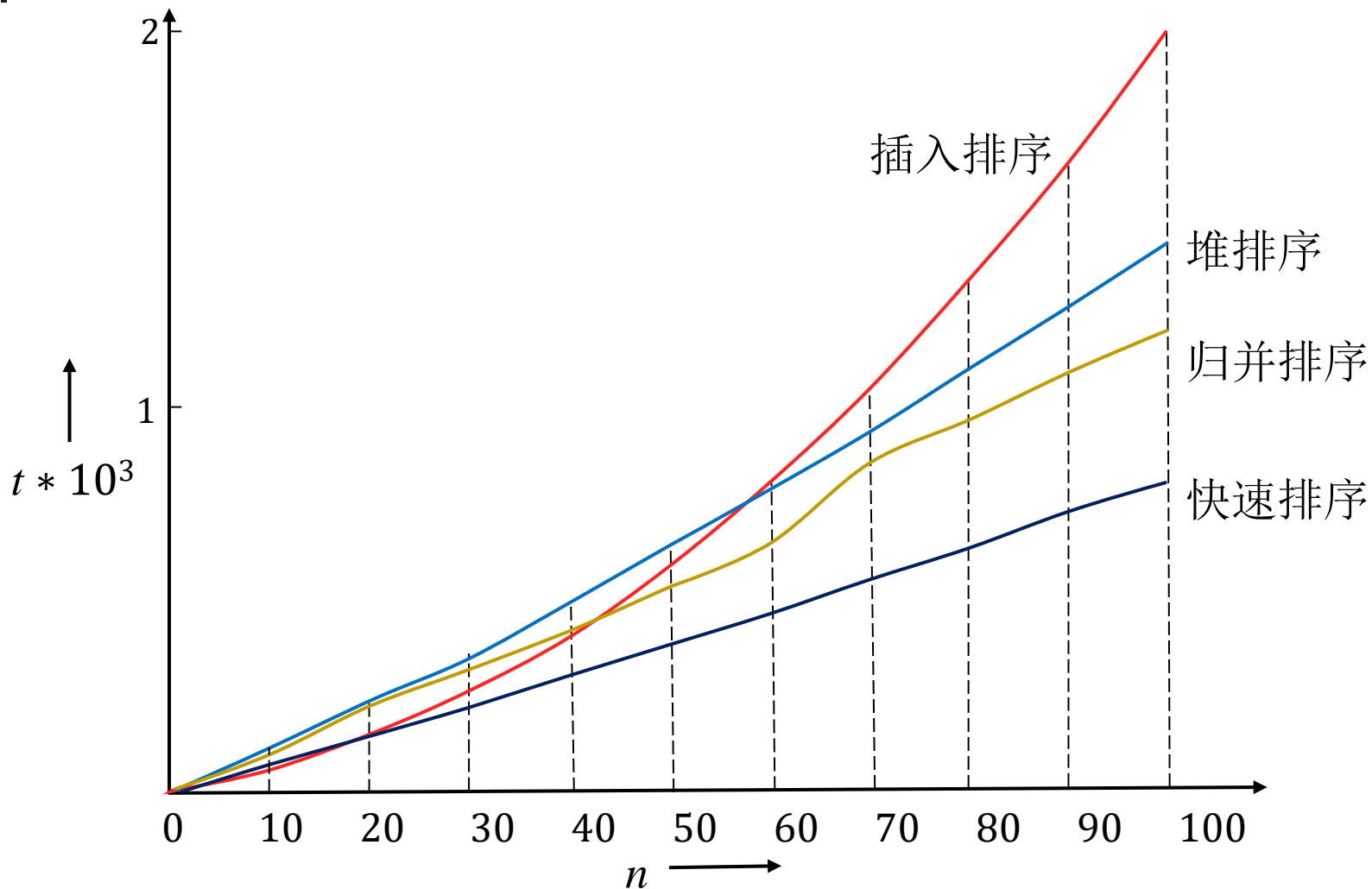
- 分划点碰上好运气的概率很大。如果划分基准的选择是随机的，从而可以期望划分是较对称的。所以快速排序**平均情形**性能很好！
- 通过修改算法 `partition`，可以设计出避开最坏情况的支点选择策略，提高快速排序算法的性能。



各种排序算法的比较

方法	最坏复杂性	平均复杂性
冒泡排序	n^2	n^2
计数排序	n^2	n^2
插入排序	n^2	n^2
选择排序	n^2	n^2
堆排序	$n \log n$	$n \log n$
归并排序	$n \log n$	$n \log n$
快速排序	n^2	$n \log n$

各排序算法平均时间的曲线图



例9 选择问题 — 寻找第 k 小

问题：对于给定的 n 个元素的数组 $a[0:n-1]$ ，要求从中找出第 k 小的元素。(☺ 如果 $k = \lfloor n/2 \rfloor$ 或者 $\lfloor n/2 \rfloor$?)

- 直接方法：首先对这 n 个元素进行排序(可在 $O(n \log n)$ 时间内解决)，然后选择第 k 个元素。
- 分治法：数组 $a[p:r]$ 被划分为两个子数组 $a[p:i]$ 和 $a[i+1:r]$ ，使 $a[p:i]$ 中每个元素都不大于 $a[i+1:r]$ 中每个元素。接着算法计算子数组 $a[p:i]$ 中的元素个数。
 - Worst case -1: $a[i] = \text{minimum}$;
 - Worst case -2: $a[i] = \text{maximum}$;
 - $T(n) = T(n-1) + O(n) = O(n^2)$
 - Best case: $i = \lfloor (n+1)/2 \rfloor$ 或 $\lfloor (n+1)/2 \rfloor$
 - $T(n) = T(1) + cn \cdot \sum_{0 \leq i \leq k} 2^{-i} = T(1) + O(n) = O(n)$ (假设 $n=2^k$)

例9 选择问题 — 寻找第 k 小


分治法：数组 $a[p:r]$ 被划分为两个子数组 $a[p:i]$ 和 $a[i+1:r]$ ，使 $a[p:i]$ 中每个元素都不大于 $a[i+1:r]$ 中每个元素。接着算法计算子数组 $a[p:i]$ 中的元素个数。

- Worst case -1: $a[i] = \text{minimum}$;
- Worst case -2: $a[i] = \text{maximum}$;

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

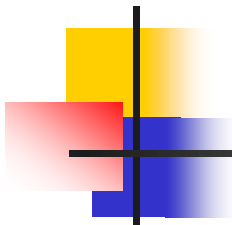
- Best case: $i = \lceil (n+1)/2 \rceil$ 或 $\lfloor (n+1)/2 \rfloor$

$$\begin{aligned} T(n) &= T(n/2) + cn \\ &= T(1) + cn \cdot \sum_{0 \leq i \leq k} 2^{-i} \\ &= T(1) + O(n) \\ &= O(n) \quad (\text{假设 } n=2^k) \end{aligned}$$



例9 选择问题 — 寻找第 k 小

1. RAND-SELECT(A, p, q, k) // kth smallest of A[p..q]
2. if $p = q$ then return A[p]
3. $r \leftarrow$ RAND-PARTITION (A, p, q)
4. $i \leftarrow r - p + 1$ // $i = \text{rank}(A[r])$
5. if $k = i$ then return A[r]
6. if $k < i$
7. then return RAND-SELECT(A, p, r-1, k)
8. else return RAND-SELECT(A, r+1, q, k-i)



例9 选择问题 — 寻找第 k 小

- 和快速排序一样，存在一个支点选择的问题！
- 如果仔细地选择支点元素，使得按这个支点所划分出的 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)，则
 - 选择问题在最坏情况下的时间开销可以变成 $\Theta(n)$;
 - 快速排序在最坏情况下的时间开销可以变成 $\Theta(n \log n)$



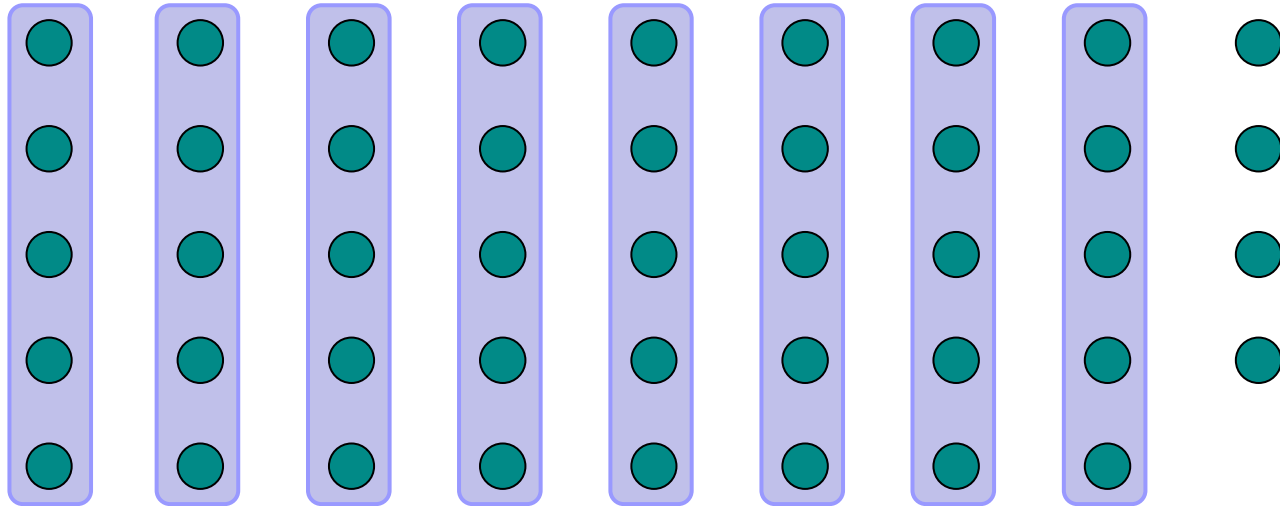
例10 中间的中间规则

一种选择支点元素的方法是使用“中间的中间（median-of-median）”规则：

- 将数组 a 中的 n 个元素分成 n/r 组， r 为某一整常数；
 - 除了最后一组外，每组都有 r 个元素。
 - 对每组中的 r 个元素进行排序，寻找每组的中位数。
 - 对所得到的 n/r 个中位数 (中间元素)，递归使用选择算法，求得“中间之中间”作为支点元素。
- $T(n) \leq T(n/5) + \Theta(n)$
- $n^{\log_a} = 1$, 对于 $\varepsilon = 0.5$, 有 $f(n) = \Omega(n^{\log_a + \varepsilon})$;
 - 并且 $af(n/b) = f(n/5) = (c/5)n \leq cn = f(n)$;
 - 所以 $T(n) = \Theta(n)$ 。

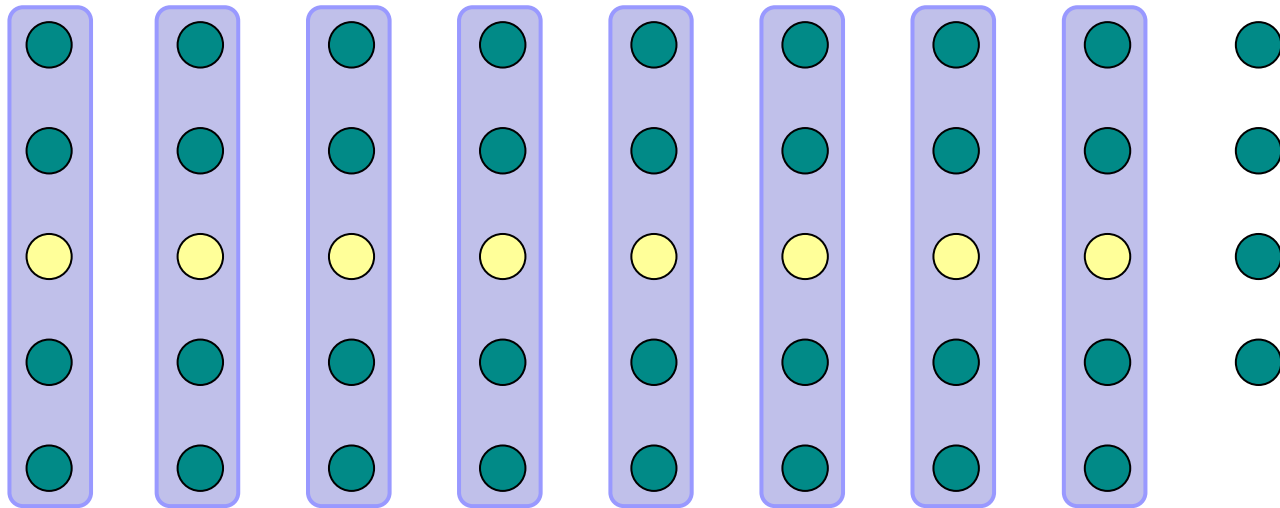
例10 中间的中間規則 — $r = 5$

1. Divide the n elements into groups of 5.



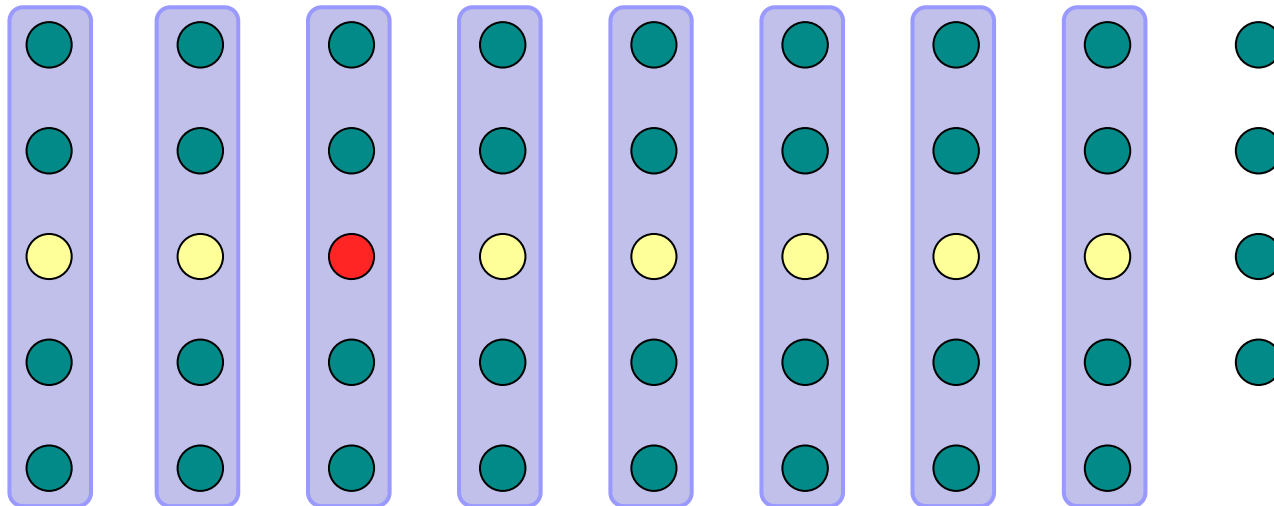
例10 中间的中位规则 — $r = 5$

1. Divide the n elements into groups of 5. Find the median of each 5-element group by rote.



例10 中间的中规则 — $r = 5$

1. Divide the n elements into groups of 5. Find the median of each 5-element group by rote.
2. Recursively SELECT the median x of the $\lfloor n/5 \rfloor$ group medians to be the pivot.





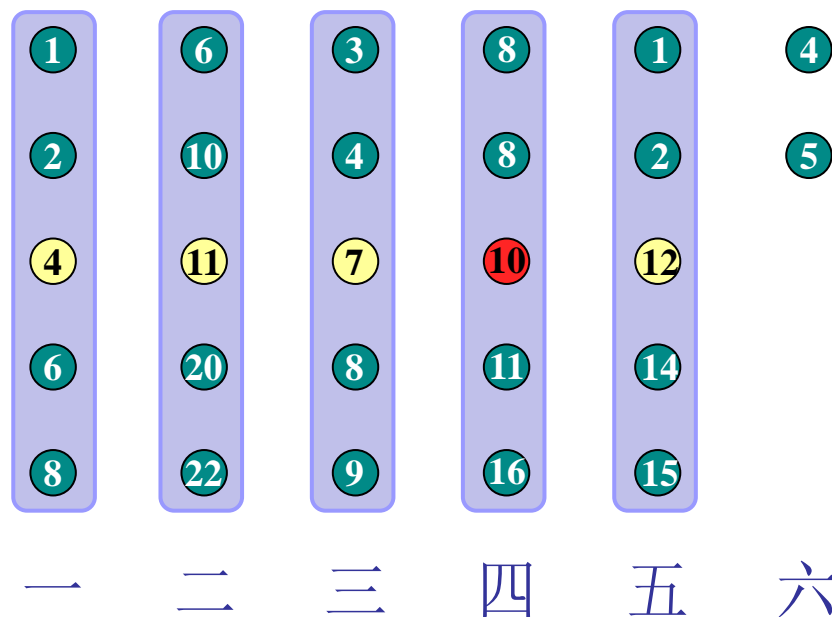
例10 中间的中间 — $r = 5$

1. Divide the n elements into groups of 5. Find the median of each 5-element group by rote. $\left. \vphantom{\begin{array}{l} \text{Divide the } n \text{ elements into groups of 5.} \\ \text{Find the median of each 5-element group by rote.} \end{array}} \right\} \Theta(n)$
 2. Recursively SELECT the median x of the $\lfloor n/5 \rfloor$ group medians to be the pivot. $\left. \vphantom{\text{Recursively SELECT the median } x \text{ of the } \lfloor n/5 \rfloor \text{ group medians to be the pivot.}} \right\} T(n/5)$
- $T(n) \leq T(n/5) + \Theta(n)$
 - $n^{\log_a} = 1$, 对于 $\varepsilon = 0.5$, 有 $f(n) = \Omega(n^{\log_a + \varepsilon})$;
 - 并且 $af(n/b) = f(n/5) = (c/5)n \leq cn = f(n)$;
 - 所以 $T(n) = \Theta(n)$ 。

例10 中间的中问规则 — $r = 5$

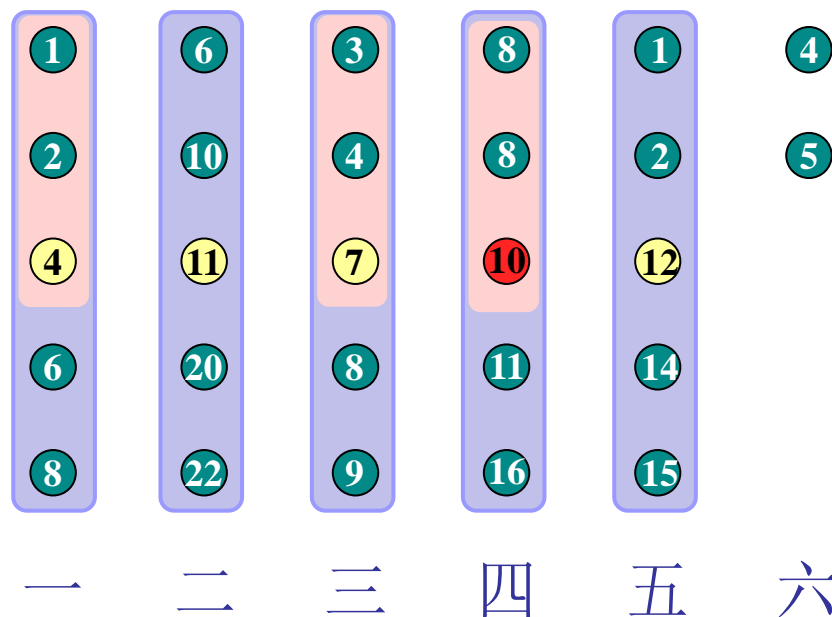
$r=5, n=27, A=[2, 6, 8, 1, 4, 10, 20, 6, 22, 11, 9, 8, 4, 3, 7, 8, 16, 11, 10, 8, 2, 14, 15, 1, 12, 5, 4]$ 。

- 这 27 个元素可以被分为 6 组并分别被排好序。考虑前面 5 组，每组的中间元素分别为 4, 11, 7, 10, 12。
- 这些中间元素的中间元素为 10，将其取为支点元素。



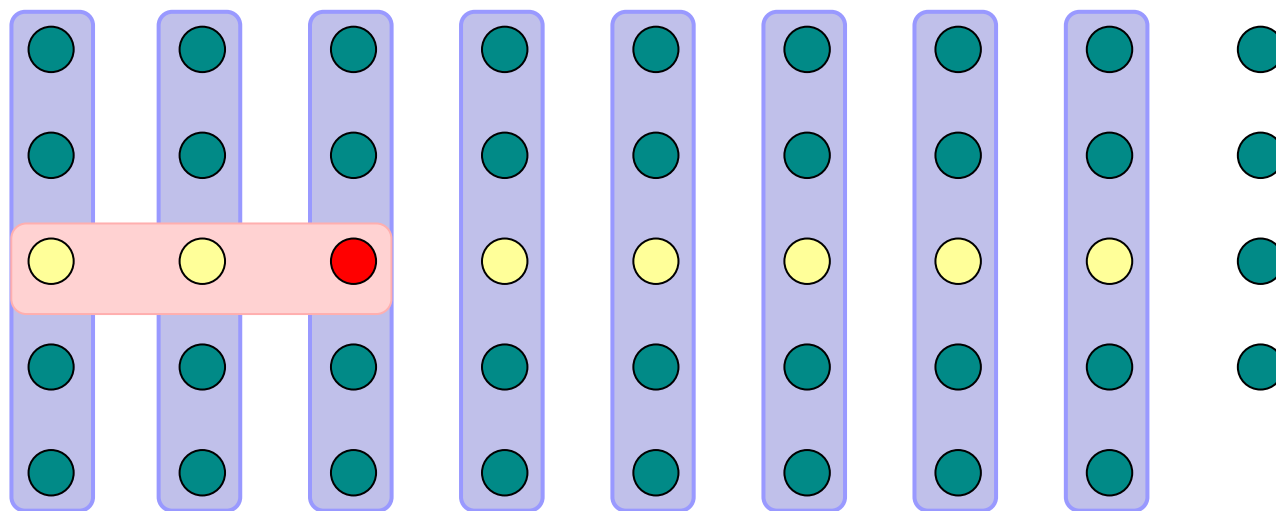
例10 中间的中间规则 — $r = 5$

- 第一、三、四组 15 个元素中至少有 9 个元素不大于中位数 10，即最少 $\lfloor \lfloor n/5 \rfloor / 2 \rfloor \cdot 3$ 个元素不大于支点元素。
- 相应的，第二、四、五组 15 个元素中，至少有 9 个不小于 10，亦即最少 $\lfloor \lfloor n/5 \rfloor / 2 \rfloor \cdot 3$ 个元素小于支点元素。

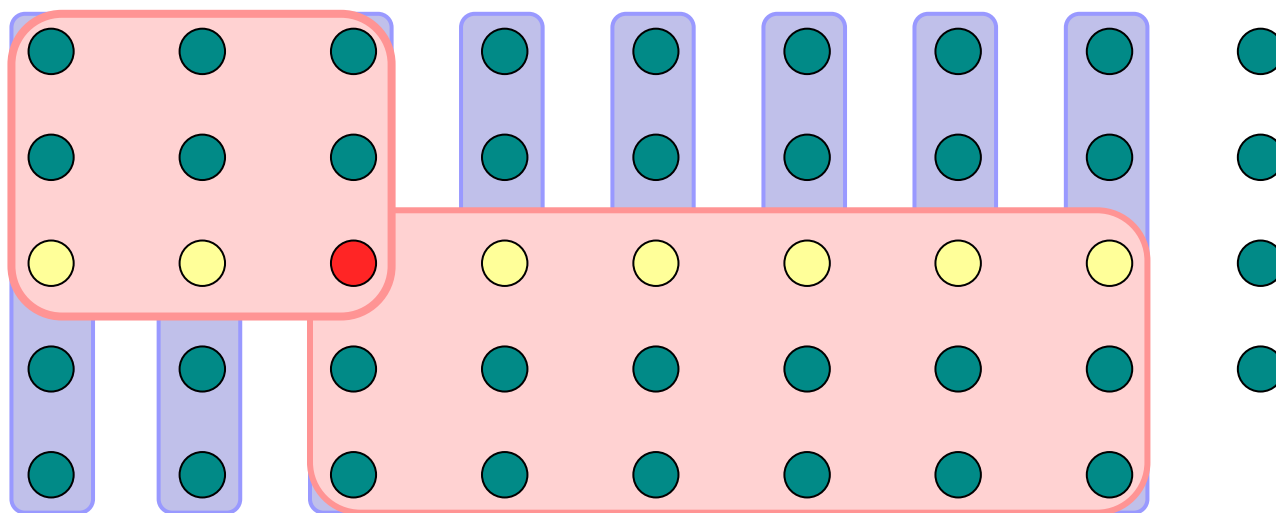


例10 中间的中问规则 — $r = 5$

- Assume all elements are distinct. We have $\lfloor n/5 \rfloor$ groups.
- Let x be the median of medians, at least $\lfloor \lfloor n/5 \rfloor / 2 \rfloor = \lfloor n/10 \rfloor$ group medians are $\leq x$.



- 



例10 中间的中寻找第 k 小

1. Divide the n elements into groups of 5. Find the median of each 5-element group by rote. $\left. \vphantom{\begin{array}{l} 1. \\ 2. \end{array}} \right\} \Theta(n)$
 2. Recursively SELECT the median x of the $\lfloor n/5 \rfloor$ group medians to be the pivot. $\left. \vphantom{\begin{array}{l} 2. \\ 3. \end{array}} \right\} T(n/5)$
 3. Partition around the pivot x . Let $i = \text{rank}(x)$. $\leftarrow \Theta(n)$
 4. **if** $k=i$ **then return** x
 5. **elseif** $k < i$
 6. **then** recursively SELECT the k th smallest element in the lower part
 7. **else** recursively SELECT the $(k-i)$ th smallest element in the upper part
- $\left. \vphantom{\begin{array}{l} 5. \\ 6. \\ 7. \end{array}} \right\} T(3n/4)$

$$T(n) \leq T(n/5) + T(3n/4) + \Theta(n)$$

例10 中间的中寻找第 k 小

定理 若 $r = 5$ ，且 A 中所有元素都不同，那么当 $n \geq 24$ 时，有 $\max\{|left|, |right|\} \leq 3n/4$ 。

证明： 根据算法易知，有 $\lfloor \lfloor n/5 \rfloor / 2 \rfloor$ 个中间元素不大于支点元素。所以，大于支点元素 x 的元素数目有上界

$$n - \lfloor \lfloor n/5 \rfloor / 2 \rfloor \cdot 3 \leq n - 3 \cdot (n - 4)/10 = 0.7n + 1.2$$

当 $n \geq 24$ 时， $0.7n + 1.2 \leq 3n/4$ 。相应的，小于支点元素 x 的元素个数也有相同的上界。这意味着无论第 k 小元素落于 x 左侧还是右侧，时间复杂度函数都有上界

$$T(n) \leq T(n/5) + T(3n/4) + cn。$$

用归纳法可证明，当 $n \geq 24$ 时有 $T(n) \leq 20cn$ 。

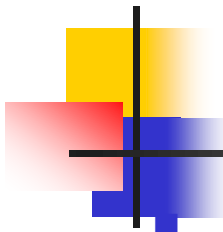


例10 中间的中寻找第k小

求解递归式： $T(n) \leq T(n/5) + T(3n/4) + \Theta(n)$

- 代入法：假设 $T(n) \leq cn$,

$$\begin{aligned} T(n) &\leq T(n/5) + T(3n/4) + \Theta(n) \\ &\leq cn/5 + 3cn/4 + \Theta(n) \\ &= 19cn/20 + \Theta(n) \\ &= cn - (cn/20 - \Theta(n)) \\ &\leq cn \end{aligned}$$



例10 中间的中寻找第 k 小

当 $r = 7$ 时，大于支点元素的元素个数有上界

$$n - \lfloor \lfloor n/7 \rfloor / 2 \rfloor \cdot 4 \leq n - 2 \cdot (n - 6)/7$$

- 当 $n \geq 15$ 时， $n - 2 \cdot (n - 6)/7 \leq 5n/6$ 。
- 所以，当 $n > 14$ 时 $T(n) \leq T(n/7) + T(5n/6) + cn$ 成立。

例10 中间的中寻找第 k 小

- 当 $r = 9$, 当 $n \geq 90$ 时, 有 $\max\{|left|, |right|\} \leq 7n/8$ 。
- 用于寻找第 k 个元素的时间 $T(n)$ 可按如下递归公式来计算:

$$T(n) = \begin{cases} cn \log n & n < 90 \\ T[n/9] + T(\lfloor 7n/8 \rfloor) + cn & n \geq 90 \end{cases}$$

- 在上述递归公式中, 假设当 $n < 90$ 时使用任意的求解算法, 当 $n \geq 90$ 时, 采用“中间的中”规则用分治法求解。利用归纳法可以证明, 当 $n \geq 90$ 时有 $T(n) \leq 72cn$ 。

例10 中间的中寻找第 k 小

- 当 $r = 3$ 时，有 $\lfloor \lfloor n/3 \rfloor / 2 \rfloor$ 个中间元素不大于支点元素。
大于支点元素的元素数目有上界

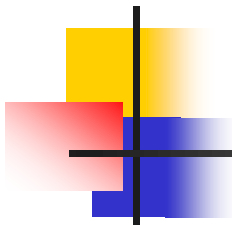
$$n - \lfloor \lfloor n/3 \rfloor / 2 \rfloor \cdot 2 \leq n - (n - 2)/3 = 2n/3 + 2/3$$

- 令 $n - \lfloor \lfloor n/3 \rfloor / 2 \rfloor \cdot 2 \leq 2n/3 + 2/3 \leq \alpha n$ ，得

$$(\alpha - 2/3)n \geq 2/3$$

找不到满足 $\alpha + 1/3 < 1$ 的 α 使上式成立。

- 所以，当 $r = 3$ 时，算法不能保证在 $O(n)$ 时间内求解。



例10 中间的中间寻找第 k 小

思考： 可以用“中间的中间” x 作为快速排序的支点吗？



例11 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。
 - 严格来讲，最接近点对可能多于一对，为简便起见，我们只找其中的一对作为问题的解。
 - 一个简单的做法是将每一个点与其他 $n-1$ 个点的距离算出，找出最小距离的点对即可。该方法的时间复杂性是 $T(n) = n(n-1)/2 + n = O(n^2)$ 。



例11 最接近点对问题——一维情况

■ 一维情形：

- 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。
- 一个简单的办法：
 - 先把 x_1, x_2, \dots, x_n 排好序；
 - 再进行一次线性扫描就可以找出最接近点对。
 - $T(n)=O(n\log n)$ 。
- 这种方法无法推广到二维情形。
- 尝试分治法！

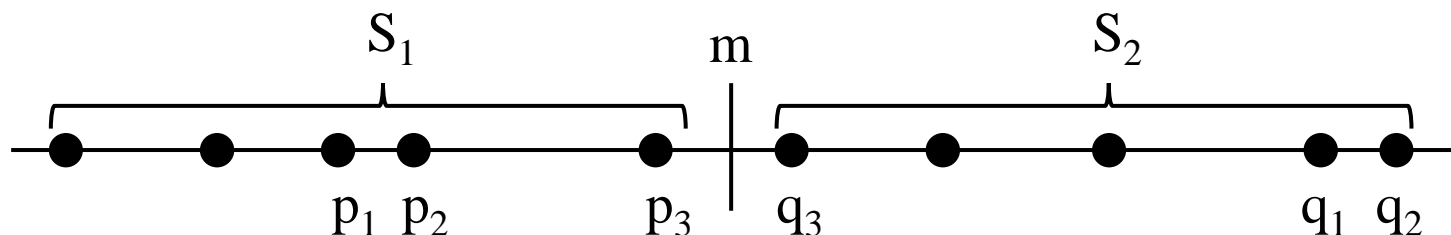
例11 最接近点对问题——一维情况

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 .

- 分割点 m 的选取不当, 会造成 $|S_1|=1, |S_2|=n-1$ 的情形, 使得

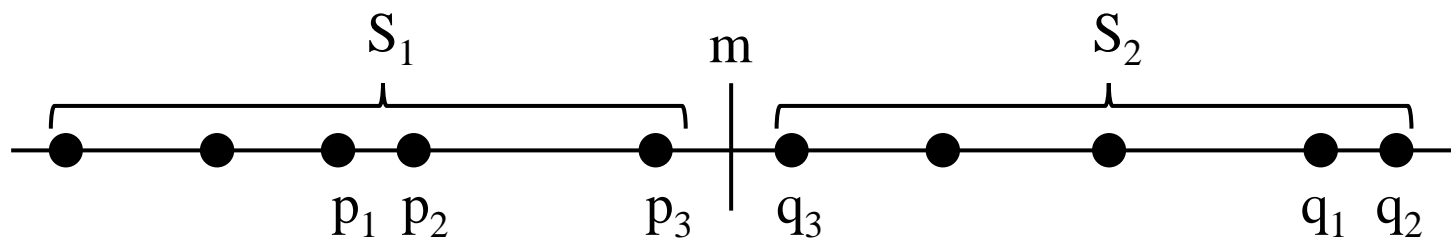
$$T(n) = T(n-1) + O(n) = O(n^2)$$

- 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 ?



例11 最接近点对问题——一维情况

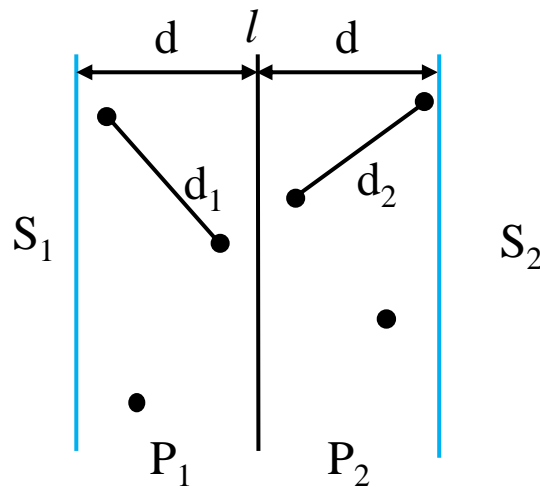
- 如果 S 的最接近点对是 $\{p_3, q_3\}$, 即 $|p_3 - q_3| < d$, 则 p_3 和 q_3 两者与 m 的距离不超过 d , 即 $p_3 \in (m-d, m]$, $q_3 \in (m, m+d]$.
- 在 S_1 中, 每个长度为 d 的半闭区间至多包含一个点(否则必有两点距离小于 d), 并且 m 是 S_1 和 S_2 的分割点, 因此 $(m-d, m]$ 中至多包含 S 中的一个点. 由图可以看出, 如果 $(m-d, m]$ 中有 S 中的点, 则此点就是 S_1 中的最大点。
- 因此, 我们用线性时间就能找到区间 $(m-d, m]$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



例11 最接近点对问题—二维情况

下面来考虑二维的情形:

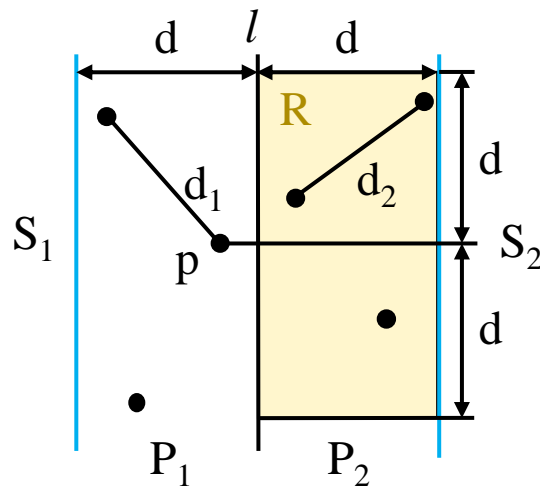
- 选取一垂直线 $l: x = m$ 来作为分割直线。
- 其中 m 为 S 中各点 x 坐标的中位数。由此将 S 分割为 S_1 和 S_2 。
- 递归地在 S_1 和 S_2 上找出其最小距离 d_1 和 d_2 ，并设 $d = \min\{d_1, d_2\}$ ， S 中的最接近点对或者是 d ，或者是某个 $\{p, q\}$ ，其中 $p \in P_1$ 且 $q \in P_2$ 。
- 能否在线性时间内找到 p, q ?



例11 最接近点对问题—二维情况

能否在线性时间内找到 p, q ?

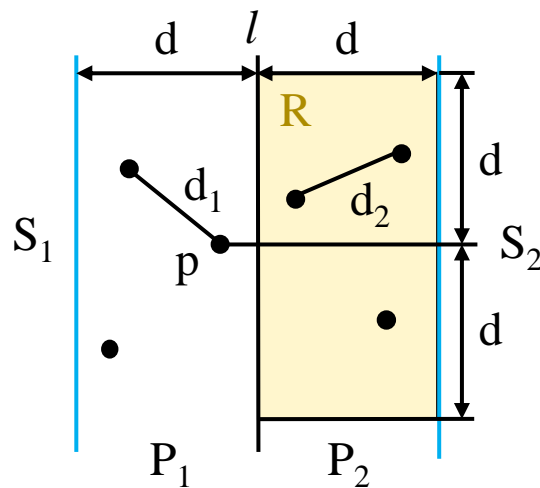
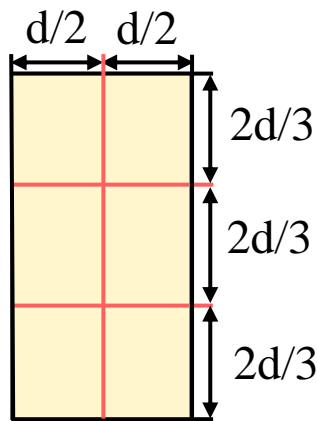
- 考虑 P_1 中任意一点 p ，它若与 P_2 中的点 q 构成最接近点对的候选者，则必有 $\text{distance}(p, q) < d$ 。满足这个条件的 P_2 中的点一定落在一个 $d \times 2d$ 的矩形 R 中。
- 由 d 的意义可知， P_2 中任何 2 个 S 中的点的距离都不小于 d 。由此可以推出**矩形 R 中最多只有 6 个 S 中的点**。
- 因此，在分治法的合并步骤中最多只需要检查 $6 \times n/2 = 3n$ 个候选者。



例11 最接近点对问题—二维情况

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
$$\text{distance}(u, v) \leq (d/2)^2 + (2d/3)^2 = 25d^2/36 < d$$
- 这与 d 的意义相矛盾。





鸽舍原理

鸽舍原理也称“**抽屉原理**”或“**利克雷原则**”，它是一个重要而又基本的数学原理，应用它可以解决各种有趣的问题，并且常常能够得到令人惊奇的结果，许多看起来相当复杂，甚至无从下手的问题，利用它能很容易得到解决：

- **原理1**：把 $n+1$ 个元素分成 n 类，不管怎么分，则一定有一类中有 2 个或 2 个以上的元素。
- **原理2**：把多于 $m \times n$ 个物体放到 n 个抽屉里，那么一定有一个抽屉里有 $m+1$ 个或者 $m+1$ 个以上的物体。
 - **原理2-1**：把 m 个元素任意放入 n ($n < m$) 个集合，则一定有一个集合里至少要有 k 个元素。其中 $k = \lceil m/n \rceil$.

例11 最接近点对问题—二维情况

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 中的点一定在矩形 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 。
- 这种投影点最多只有 6 个。
- 因此，若将 P_1 和 P_2 中所有 S 中的点按其 y 坐标排好序，则对 P_1 中的每一点所有点，对排好序的 P_2 中的点列作一次扫描，最多只需要检查 P_2 中排好序的相继 6 个点。

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



例11 最接近点对问题—二维情况

1. $m = S$ 中各点 x 间坐标的中位数; 构造 S_1 和 S_2 ;
2. $d_1 = \text{cpair2}(S_1)$; $d_2 = \text{cpair2}(S_2)$;
3. $d_m = \min(d_1, d_2)$;
4. 设 P_1 是 S_1 中距垂直分割线 l 的距离在 d_m 之内的所有点组成的集合;
 P_2 是 S_2 中距分割线 l 的距离在 d_m 之内所有点组成的集合;
 将 P_1 和 P_2 中点依其 y 坐标值排序;
 并设 X 和 Y 是相应的已排好序的点列;
5. 通过扫描 X 以及对于 X 中每个点检查 Y 中与其距离在 d_m 之内的所有点
 (最多6个)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 $2d_m$ 的区间内移动;
 设 d_l 是按这种扫描方式找到的点对间的最小距离;
6. $d = \min(d_m, d_l)$;



例12 循环赛日程表

设有 $2k$ 个运动员要进行网球循环赛，现在要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。



例12 循环赛日程表

- 按分治策略，可以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。
- 递归地使用这种一分为二的策略对选手进行分割，直到只剩下 2 个选手时，比赛日程表的制定就变得很简单了。
- 这时只要让这两个选手进行比赛就可以了。

例12 循环赛日程表

右表是 8 个选手的比赛日程表。

- 左上角与左下角的 2 小块分别为选手 1 至选手 4 以及选手 5 至选手 8 前 3 天的比赛日程。
- 将左上角小块的数字复制到右下角，左下角的数字复制到右上角，就排好了选手们后 4 天的比赛日程。

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

例12 循环赛日程表

```
void gametable(int k)
```

```
2.  {
3.    int a[100][100];
4.    int n, temp, i, j, p, t;
5.    n=2; //两个参赛选手日程
6.    a[1][1]=1; a[1][2]=2;
7.    a[2][1]=2; a[2][2]=1;
8.    for(t=1; t<k; t++)
    //迭代处理, 依次处理 $2^n \dots 2^k$ 
    //个选手的比赛日程
9.    {
10.       temp=n; n=n*2; //填左下角元素
11.       for(i=temp+1; i<=n; i++)
12.         for(j=1; j<=temp; j++)
13.           a[i][j]=a[i-temp][j]+temp;
```

//左下角和左上角元素的对应关系

```
14.   for(i=1; i<=temp; i++)
    //将左下角元素抄到右上角
15.     for(j=temp+1; j<=n; j++)
16.       a[i][j]=a[i+temp][(j+temp)%n];
17.   for(i=temp+1; i<=n; i++)
    //将左上角元素抄到右下角
18.     for(j=temp+1; j<=n; j++)
19.       a[i][j]=a[i-temp][j-temp];
20. }
```



求逆序对个数

- 给定自然数 $1, \dots, n$ 的一个排列，如果 $j > i$ 但 j 排在 i 的前面则称 (j, i) 为该排列的一个逆序。
- 例如在 $(1, 3, 4, 2, 5)$ 中， $(3, 2), (4, 2)$ 为该排列的逆序。该排列总共有 2 个逆序。
 - 试用分治法设计一个计算给定排列的逆序总数的算法，并分析该算法的时间复杂度。

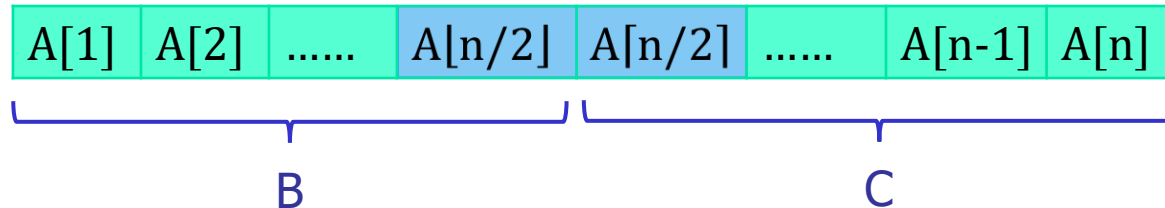


求逆序对个数

提示：可在 $O(n)$ 时间内算出 2 段排好序的子序列之间的逆序数。例如序列(1, 2, 4, 8, 3, 5, 6, 7)中，(1, 2, 4, 8) 为排在左边的子序列，(3, 5, 6, 7)为排在右边的子序列，它们之间的逆序数可用一个线性时间的算法算出。

求逆序对个数

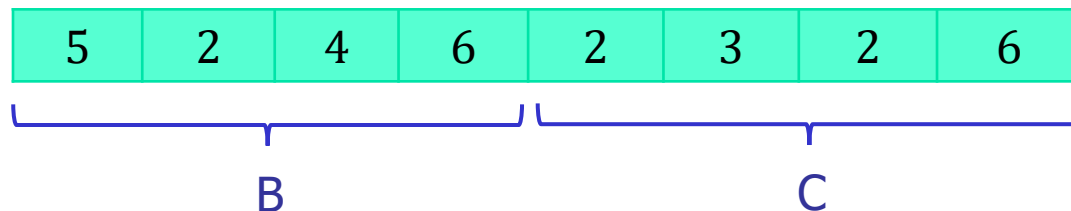
- $f(i, j)$ 为 i 到 j 号元素中的逆序对个数
- $s(i, j, k)$ 表示以 k 为分割点，第一个元素在 i 到 k 中，第二个元素在 $k + 1$ 到 j 中形成的逆序对数。
- 递归关系式： $f(i, j) = f(i, k) + f(k+1, j) + s(i, j, k)$
- $f(i, i) = 0$
- $f(i, i+1) = f(i, i) + f(i+1, i+1) + s(i, i+1, i)$
- 递归求解
- 如何来统计序列 **B** 和 **C** 之间的“逆序对”呢？



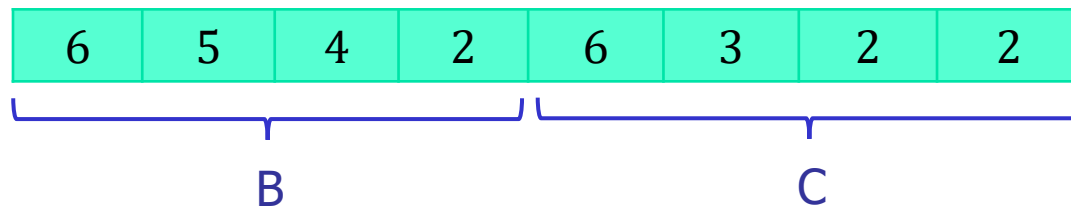
求逆序对个数

在递归的求解 B、C 两个序列中的逆序对的个数以后，如果对 B、C 两个序列当中的元素进行排列的话，要统计 B 和 C 两个序列之间的“逆序对”是非常容易的。如图。

■ 排序前：



■ 排序后：



■ 在 B 数组当中，首先，B 中的 6，5，4 都与 C 数组当中的 3，2，2 都构成了“逆序对”，而 2 不会构成逆序对，因此 B、C 两个数组之间构成的逆序对的个数为 $3+3+3=9$ 。



求逆序对个数

由于 B、C 两个数组已经进行了排序，因此可以有效地统计 B、C 两个数组间的“逆序对”的个数。这一统计步骤可在线性时间内完成。

- 排序的过程可在递归求解子问题时完成，算法的时间复杂度为 $T(n) = 2T(n/2) + O(n\log n)$ 。
- 根据主定理 case 3, $T(n) = O(n\log n)$ 。
- 虽然对 B、C 两个序列当中的元素进行了排序，使得序列 A 当中某一部分元素的顺序被打乱了，但由于求解过程是递归定义的，在排序之前 B、C 两个序列各自其中的元素之间的“逆序对”个数已经被统计过了，并且排不排序对统计 B、C 两个数组之间的“逆序对”的个数不会产生影响。所以排序过程对整个问题的求解的正确性是没有任何影响的。



复杂性的上限(upper bound)

复杂性的上限

- 当且仅当某个问题至少有一个复杂性为 $O(f(n))$ 的求解算法时，存在一个复杂性的上限 $f(n)$ 。
- 证明一个问题复杂性上限为 $f(n)$ 的一种方法是设计一个复杂性为 $O(f(n))$ 的算法。
- 例如，在发现 Strassen 矩阵乘法之前，矩阵乘法的复杂性上限为 n^3 ，Strassen 算法的发现使复杂性上限降为 $n^{2.81}$ 。



复杂性的下限(lower bound)

- 当且仅当求解一个问题所有算法的复杂性均为 $\Omega(g(n))$ 时，则称 $g(n)$ 为该问题的复杂性的下限。
- 为了确定一个问题的复杂性下限 $g(n)$ ，必须证明该问题的每一个求解算法的复杂性均为 $\Omega(g(n))$)
- 要得到这样一个结论相当困难，因为不可能考察所有的求解算法。对于大多数问题，可以建立一个基于输入和/或输出数目的简单下限。



复杂性的下限(lower bound)

- 对 n 个元素进行排序的算法的复杂性为 $\Omega(n)$ ，因为所有的算法对每一个元素都必须检查至少一遍，否则未检查的元素可能会排列在错误的位置上。
- 每一个计算两个 $n \times n$ 矩阵乘法的算法都有复杂性 $\Omega(n^2)$ 。因为结果矩阵中有 n^2 个元素并且产生每个元素所需要的时间为 $\Omega(1)$ 。



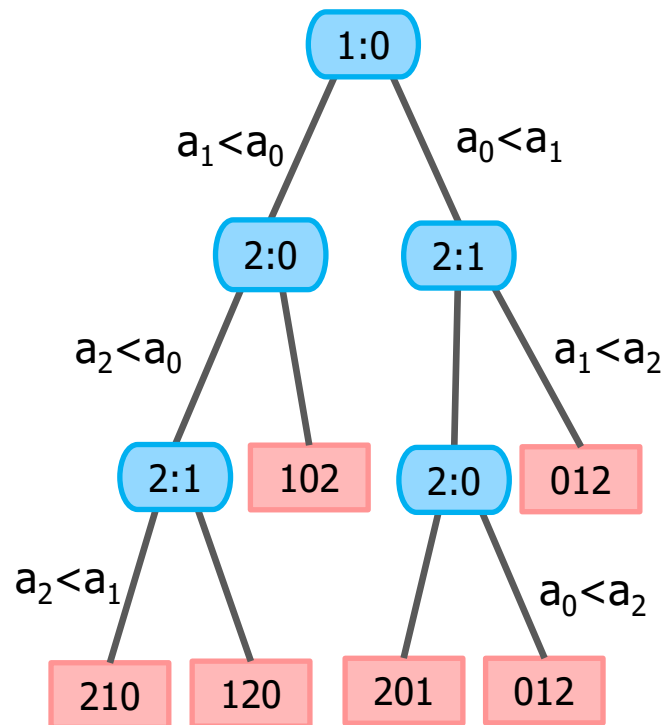
比较算法

- 所谓比较算法是指算法的操作主要限于元素比较和元素移动。
- 基于比较的寻找 n 个元素中的**最小最大值问题**，任何求最大最小的算法从初始状态到完成状态所用比较次数不可少于 $\lceil 3n/2 \rceil - 2$ 。
- 基于比较的排序算法在最坏情况下至少要进行 $\Omega(n \log n)$ 次比较。
 - 可用决策树（decision-tree）来证明下限。
 - 证明过程中用树来模拟算法的执行过程。
 - 对于树的每个内部节点，算法执行一次比较并根据比较结果移向它的某一子节点。
 - 算法在叶节点处终止。

插入排序的决策树

- 右图给出了对三个元素 a_0, a_1, a_2 使用插入排序时的决策树。
- 每个内部节点有一个 $i:j$ 的标志, 表示 a_i 与 a_j 进行比较。
- 如果 $a_i < a_j$, 算法移向左分枝; 如果 $a_i > a_j$, 移向右分枝。
- 因为元素互不相同, 所以 $a_i = a_j$ 不会发生。
- 叶节点标出了所产生的排序。

左小右大顺序:





决策树说明

假定输入是 $1, \dots, n$ 的一个排列, 决策树中每个叶节点代表对输入排列的排序结果。

- 正确的排序算法对于 n 个输入排列必须产生 $n!$ 排序结果, 因此决策树有 $n!$ 个叶节点。
- 有 n 个叶节点的二叉树的深度至少为 $\log n$, 因此有 $n!$ 个叶子结点的决策树的高度至少为 $\log(n!) = \Theta(n \log n)$ 。
- 每一个比较排序算法在最坏情况下至少要进行 $\Omega(n \log n)$ 次比较。
- 每个有 $n!$ 个叶节点的决策树的平均外路长度为 $\Omega(n \log n)$; 所以每个比较排序算法的平均复杂性也是 $\Omega(n \log n)$ 。



结论

堆排序和归并排序的时间复杂度上限都是 $O(n \log n)$, 我们称它们为**渐进最优**。快速排序在平均情况下性能较优。

- 注意:

- 这一下限仅仅指最坏情况;
- 这一下限不依赖于任何特定的算法。

- 如果我们知道待排序的元素有两个特点:

- 每一个排序关键字是 1 或者是 2;
- 元素仅仅包含排序关键字。

你能做到的计算复杂度下限是多少?

- 对于排序算法, 我们还有一个通用的下限 $\Omega(n)$, 因为我们至少需要对每个元素检查一遍。