

## Part2：递归与分治策略（II）

---

- 分治法：快速排序
- 分治法：大整数乘法和Strassen矩阵乘法
- 用分治法解最近对问题和凸包问题

## 5.2 快速排序

### 算法思想

- 合并排序按照元素的位置划分并合并的方式进行排序。划分过程很快，主要工作在合并子问题。
- 考虑如果按照元素的值进行划分子问题：

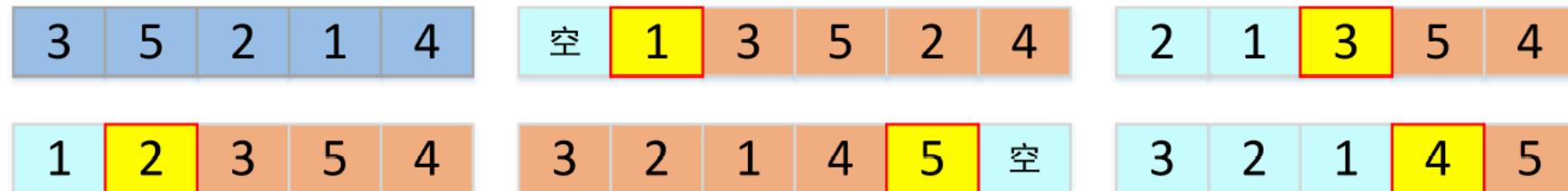
$$\underbrace{A[0] \dots A[s-1]}_{< A[s]} \quad A[s] \quad \underbrace{A[s+1] \dots A[n-1]}_{> A[s]}$$

- 子问题递归排序后，合并很简单（拼接），主要工作在如何划分（如何有效地划分？），这就是快速排序算法。



## 5.2 快速排序

### 子问题划分方式



## 5.2 快速排序

X  
||  
初始关键字:    27    38    13    49    76    97    65    50  
↑            ↑            ↑            ↑↑j            ↑            ↑            ↑            ↑  
i            i            i            ij            j            j            j            j

完成一趟排序: ( 27    38    13)    49    (76    97    65    50)

分别进行快速排序: ( 13)    27    (38)    49    (50    65)    76    (97)

快速排序结束:    13    27    38    49    50    65    76    97

## 5.2 快速排序

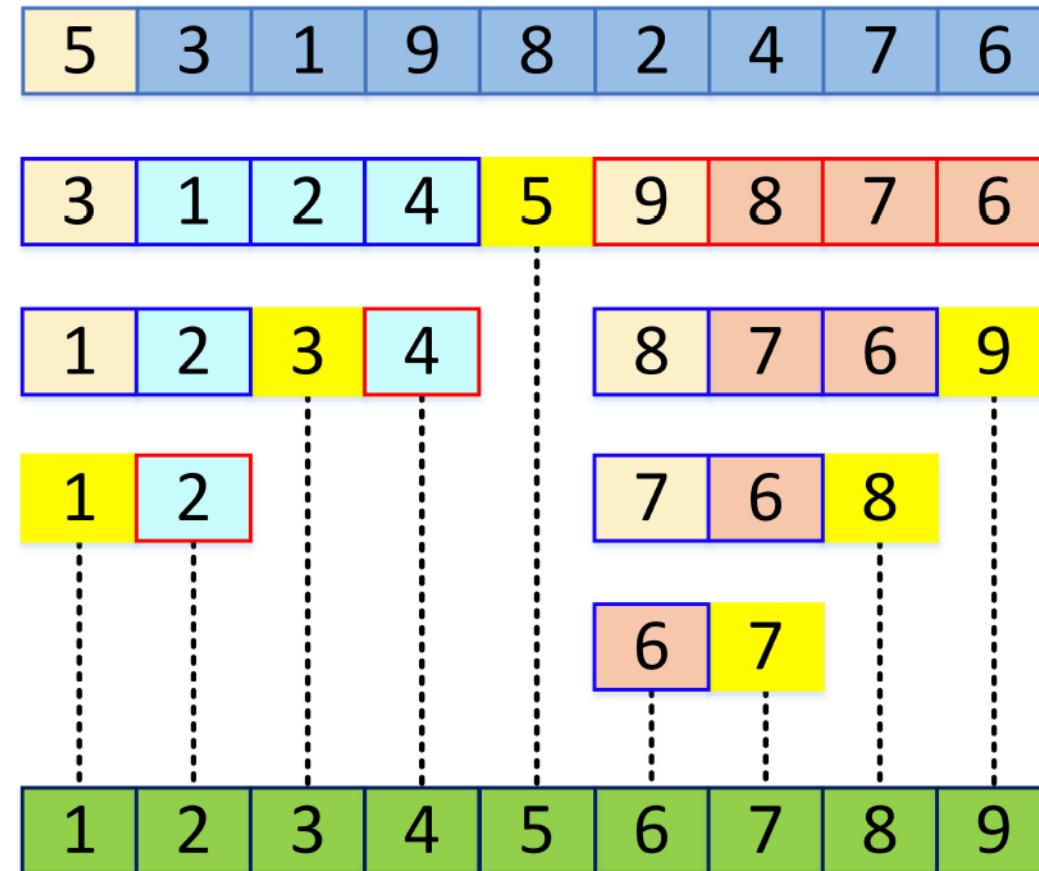
伪代码

### 快速排序算法

```
1: function Quick_Sort( $A[l...r]$ )
2:   if  $l < r$  then
3:      $s \leftarrow \text{Partition}(A[l...r])$  //找到分裂位置
4:     Quick_Sort( $A[l...s - 1]$ )
5:     Quick_Sort( $A[s + 1...r]$ )
6:   end if
7: end function
```

## 5.2 快速排序

依据首元素划分（递归调用）



## 5.2 快速排序——霍尔划分方法

- 比划分值 $p$ 小的元素位于左边，大的位于右边，简单地 $p = A[0]$ ；
- 从左到右的扫描从第二个元素开始，直到 $A[i] \geq p$ ；
- 从右到左的扫描从最后的元素开始，直到 $A[j] \leq p$ ；
- 两边的扫描都停止后，会有下面三种情况：

- ①  $i < j$ ，则 $A[i] \leftrightarrow A[j]$ ；



- ②  $i > j$ ，则 $A[j] \leftrightarrow A[0]$ ；



- ③  $i = j$ ，则 $A[i] = A[j] = p$ ；



## 5.2 快速排序——霍尔划分方法伪代码

```
1: function Hoare_Partition( $A[l\dots r]$ )
2:    $p \leftarrow A[l]; i \leftarrow l; j \leftarrow r + 1$ 
3:   while  $i < j$  do
4:     while  $A[i] < p$  do // 找出  $A[i] \geq p$  的位置
5:        $i \leftarrow i + 1$ 
6:     end while
7:     while  $A[j] > p$  do // 找出  $A[j] \leq p$  的位置
8:        $j \leftarrow j - 1$ 
9:     end while
10:     $A[i] \leftrightarrow A[j]$  //  $i < j$  时，交换  $A[i]$  和  $A[j]$ 
11:   end while
12:    $A[i] \leftrightarrow A[j]$  // 当循环满足  $i \geq j$  后，执行了  $A[i] \leftrightarrow A[j]$ ，需撤销
13:    $A[l] \leftrightarrow A[j]$ 
14: end function
```

## 5.2 快速排序——霍尔划分方法实例

初始



$A[i] \geq p, A[j] \leq p$



交换 $A[i]$ 和 $A[j]$



$A[i] \geq p, A[j] \leq p$



交换 $A[i]$ 和 $A[j]$



$i \geq j$



交换 $A[j]$ 和 $p$



## 5.2 快速排序——霍尔划分方法实例

初始



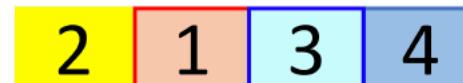
$A[i] \geq p, A[j] \leq p$



交换 $A[i]$ 和 $A[j]$



$i \geq j$



交换 $A[j]$ 和 $p$



初始



$p$

$i \geq j$



$A[i]$

交换 $A[j]$ 和 $p$

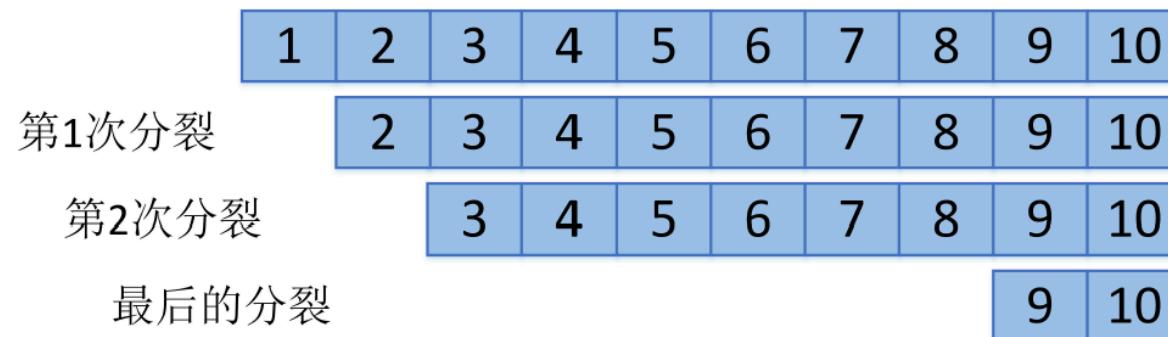


$A[j]$

## 5.2 快速排序——算法效率分析

### 最差情况

- 考虑对一个已经排好序的序列[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]进行排序；
- 从左到右扫描，停止在 $A[1]$ ；从右到左扫描，停止在 $A[0]$ ；
- $A[0]$ 和本身交换，然后继续递归排序[2, 3, 4, 5, 6, 7, 8, 9, 10]；
- 第1次分裂需要的比较次数为 $1 + n$ ；
- 第2次分裂需要的比较次数为 $n$ ；
- 最后一次分裂比较3次（ $A[i]$ 比较1次， $A[j]$ 比较2次）；
- $C_{worst}(n) = (n + 1) + n + \dots + 3 = (n + 1)(n + 2)/2 - 3 \in \Theta(n^2)$



## 5.2 快速排序——算法效率分析

### 最优情况

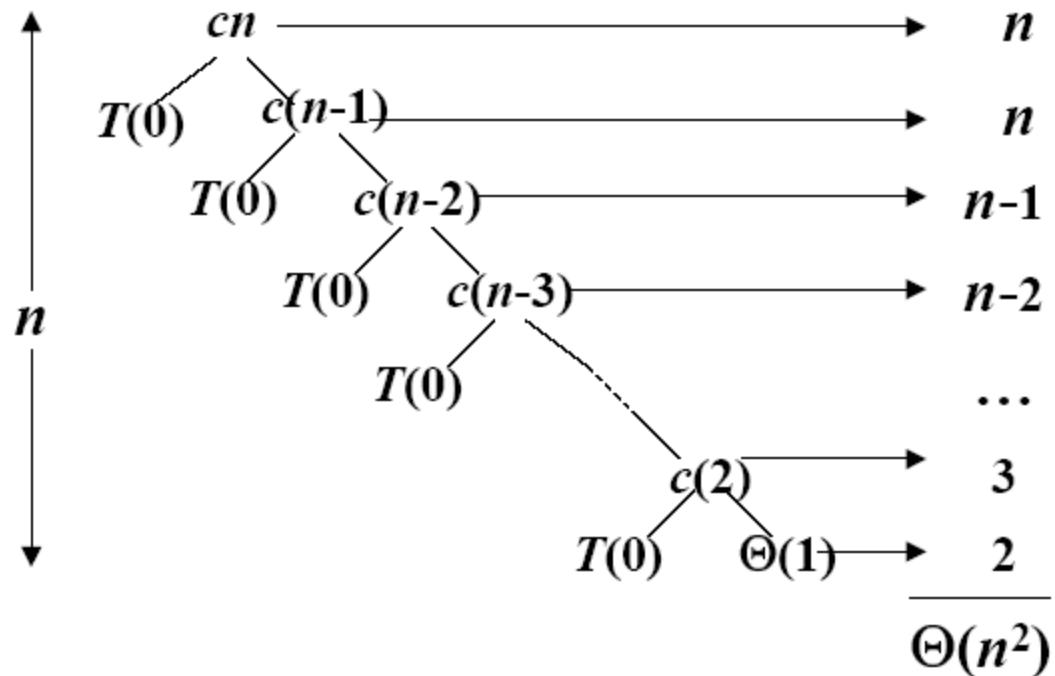
- 最优情况下，数组序列被分为两个大小差不多规模的子序列；
- $C_{best}(n) = 2C_{best}(n/2) + \Theta(n)$ ,  $C_{best}(1) = 0$ ;
- 根据主定理可得  $C_{best}(n) \in \Theta(n \log_2 n)$ ;

### 平均情况

- 经过  $n + 1$  次比较，分裂点出现在任意的位置  $s (0 \leq s \leq n - 1)$ ，划分数组为大小为  $s$  和  $n - 1 - s$  两个部分，每种情况出现的概率为  $1/n$ ;
- $C_{avg}(n) = \frac{1}{n} \sum_{s=0}^{n-1} [(n+1) + C_{avg}(s) + C_{avg}(n-1-s)]$   
 $C_{avg}(1) = 0, C_{avg}(0) = 0$ ;
- 计算可得  $C_{avg}(n) \approx 2n \ln 2 \approx 1.39n \log_2 n$ ;
- 算法平均情况下的效率仅比最优情况下仅仅多执行 39% 的计算;

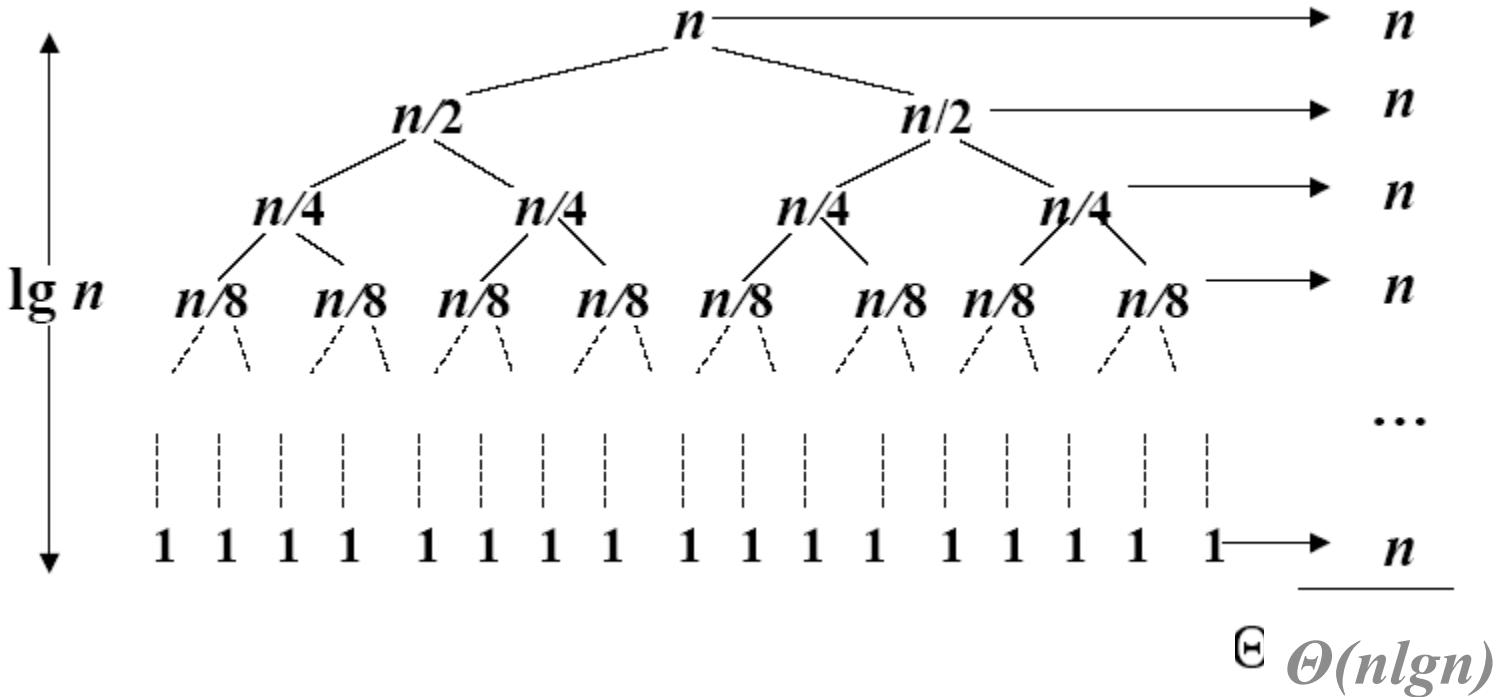
# 最坏情况递归树

- $T(n) = T(n - 1) + T(0) + \Theta(n)$



- A recursion tree for QUICKSORT in which the Partition procedure always puts only a single element on one side of the partition (the worst case). The resulting running time is  $\Theta(n^2)$

# 最优情况递归树

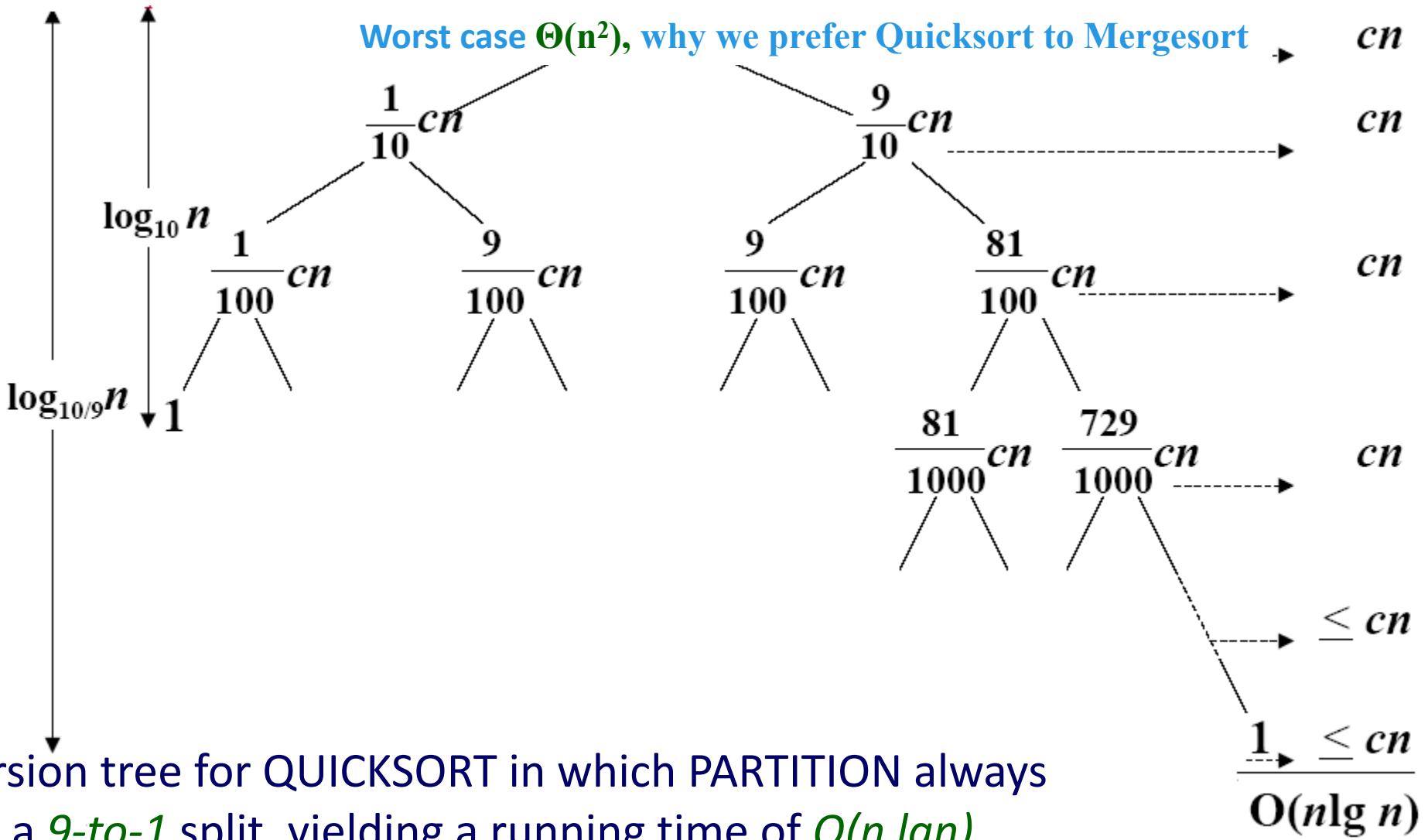


- A recursion tree for QUICKSORT in which the Partition always balances the two sides of the partition equally (the best case). The resulting running time is  $\Theta(n \lg n)$

# 最优情况分析

- If we're lucky, PARTITION splits the array evenly:
  - $T(n) = 2T(n/2) + \Theta(n) = \Theta(n\log n)$ .
- What if the split is always 1/10:9/10?
  - $T(n) \leq T(9n/10) + T(n/10) + \Theta(n)$
  - What is the solution to this recurrence?

# 特殊情况分析



# 随机快速排序

## RANDOMIZED-PARTITION

```
1  i  $\leftarrow$  RANDOM(p, r)
2  exchange A[r]  $\leftrightarrow$  A[i]
3  return PARTITION(A, p, r)
```

## RANDOMIZED-QUICKSORT

```
1  if p < r
2    then q  $\leftarrow$  RANDOMIZED-PARTITION(A, p, r)
3        RANDOMIZED-QUICKSORT(A, p, q-1)
4        RANDOMIZED-QUICKSORT(A, q+1, r)
```

# 合并排序 VS 快速排序

Test environment:

- IBM370/158
- Programming in PL/1
- Input consists of random integers range in (0, 10000)

n	1000	1500	2000	2500	3000	3500	4000	4500
MERGESORT	500	750	1050	1400	1650	2000	2250	2650
QUICKSORT	400	600	850	1050	1300	1550	1800	2050
n	5000	5500	6000	6500	7000	7500	8000	8500
MERGESORT	2900	3450	3500	3850	4250	4550	4950	5200
QUICKSORT	2300	2650	2800	3000	3350	3700	3900	4100

# 排序算法总结

- 插入排序时间效率  $\Theta(n^2)$ ; 在位排序 (**in place**)
- 合并排序 时间效率  $\Theta(n \lg n)$ ; 需要额外存储空间  $\Theta(n)$ .
- 快速排序平均时间效率  $\Theta(n \lg n)$  **on average**; 更符合实际情况中，被广泛使用
- 堆排序 时间效率  $\Theta(n \lg n)$ ; 在位排序.
- 所有基于比较的排序算法中，堆排序和合并排序是渐近效率最优的算法
- 线性时间排序（计数排序、基数排序、桶排序）

# 排序算法总结

Sorting methods	Worst Case	Best Case	Average Case	Application
Insert Sort	$n^2$	$n$	$n^2$	Very fast when $n < 50$
Bubble Sort	$n^2$	$n$	$n^2$	Very fast when $n < 50$
Merge Sort	$n \lg n$	$n \lg n$	$n \lg n$	Need extra space; good for external sort
Heap Sort	$n \lg n$	$n \lg n$	$n \lg n$	Good for real-time app.
Quick Sort	$n^2$	$n \lg n$	$n \lg n$	Practical and fast
Counting Sort	$k+n$	$k+n$	$k+n$	Small, fixed range; Need extra space
Radix Sort	$d(k+n)$	$d(k+n)$	$d(k+n)$	Fixed range; Need extra space
Bucket Sort	$n$	$n$	$n$	Uniform distribution

- Which in place?
- Which stable?

# 堆排序——堆结构

---

设 $T$ 是一棵二叉树， $T$ 具有堆结构当且仅当满足下列条件：

1. 从根开始直到 $h-1$ 层都是完备的；
2. 所有的叶节点都位于第 $h$ 层或第 $h-1$ 层；
3. 所有具有高度为 $h$ 的叶节点都在高度为 $h-1$ 的叶节点的左边。

上面条件中的第1、2条其实是等价的。第3条意味着第 $h-1$ 层中最右边的非叶节点可以只有一个左子节点而没有右子节点，而其它的非叶节点则必须既有左子节点又有右子节点。所以**具有堆结构的二叉树**又称为**左完备二叉树**。



## 最大堆的定义

---

$T$ 是一棵二叉树，称 $T$ 是部分有序的当且仅当 $T$ 的任何一个节点的值都大于或等于（不小于）它的任何一个子节点的值（如果它有子节点的话）。

$T$ 是一棵二叉树，如果 $T$ 同时具有堆结构和部分有序性质，则称 $T$ 为一个堆（或最大堆）。



# 堆的其它定义和性质

---

最大堆的定义如下：

设数组 $a$ 中存放了 $n$ 个数据元素，数组下标从0开始，如果当数组下标 $2i+1 < n$ 时有： $a[i] \geq a[2i+1]$ ；

如果当数组下标 $2i+2 < n$ 时有： $a[i] \geq a[2i+2]$ ，则这样的数据结构称为最大堆。



## 最小堆的定义如下：

---

设数组a中存放了n个数据元素，数组下标从0开始，如果当数组下标 $2i+1 < n$ 时有： $a[i] \leq a[2i+1]$ ；如果当数组下标 $2i+2 < n$ 时有： $a[i] \leq a[2i+2]$ ，则这样的数据结构称为最小堆。



# 树的一维表达方法

---

给定一个数组  $E$ , 其中的元素下标  $1, \dots, n$

数组中的第 $i$ 个节点与树中节点的对应关系为

- 节点 $i$ 的 左子节点在数组的 $2i$ 位置
- 节点 $i$ 的 右子节点在数组的 $2i + 1$ 位置
- 节点 $i$ 的 父节点在数组的 $\lfloor i/2 \rfloor$ 位置

# 树的一维表达方法

9	5	7	1	4	3	6
---	---	---	---	---	---	---

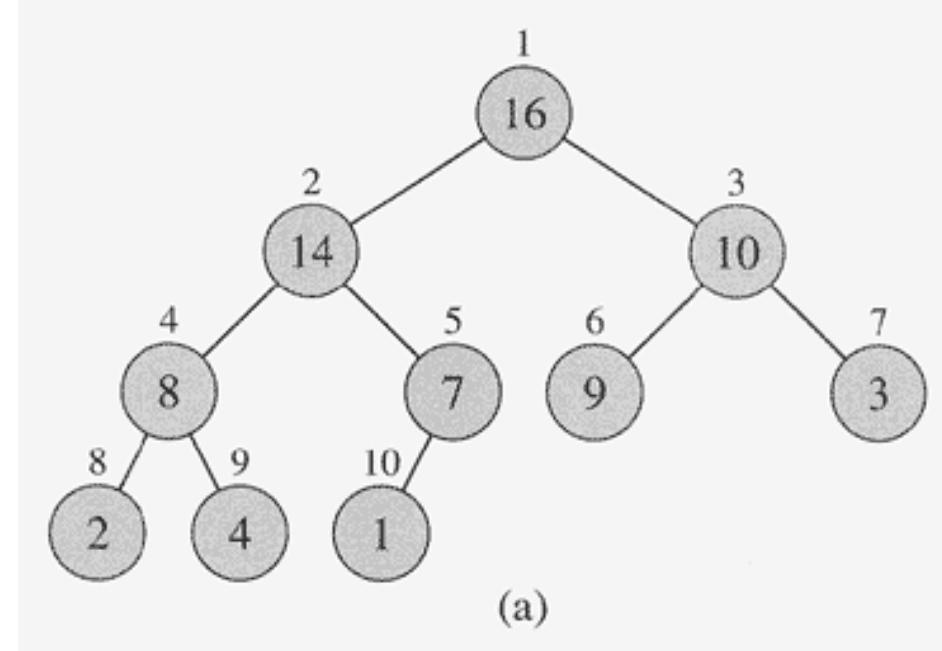
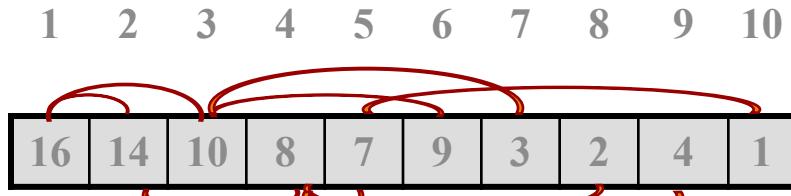
Heap 1

50	24	30	20	21	18	3	12	5	6
----	----	----	----	----	----	---	----	---	---

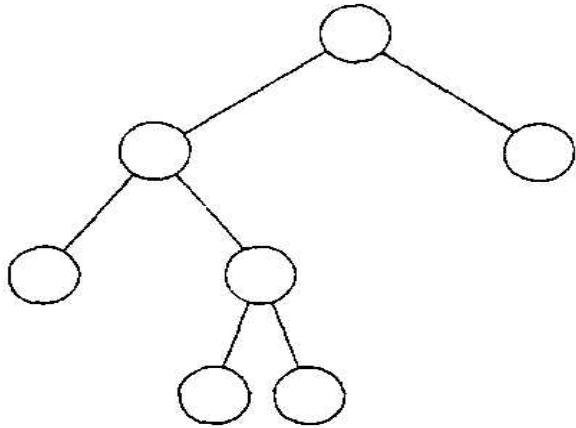
Heap 2

# 堆

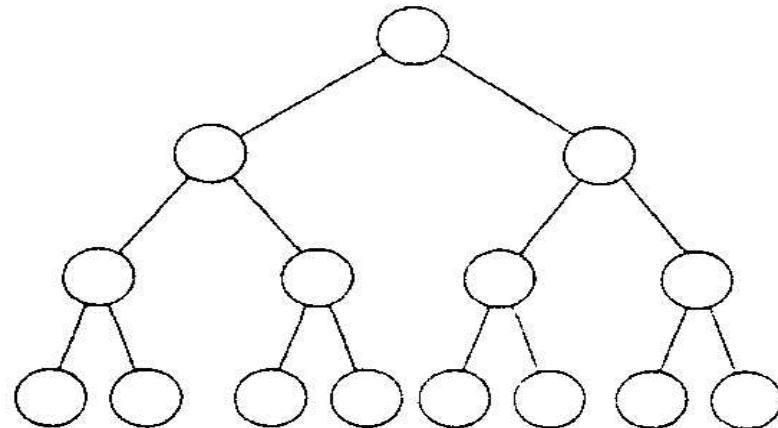
- $PARENT(i) = \lfloor i / 2 \rfloor$ ,  $LEFT(i) = 2i$ , and  $RIGHT(i) = 2i + 1$ .



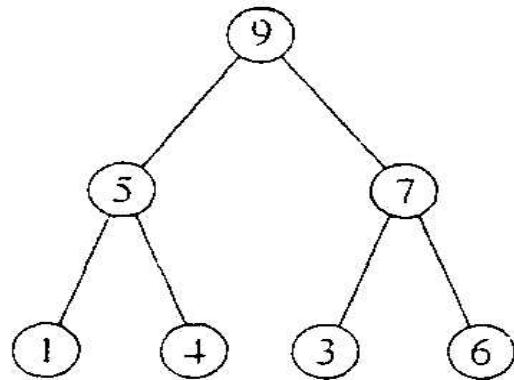
# 实例



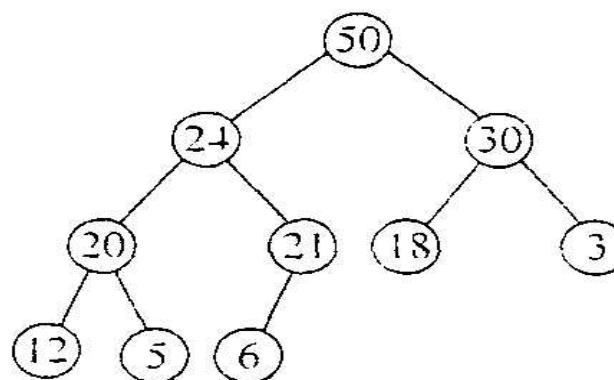
(a) A 2-tree



(b) A complete binary tree

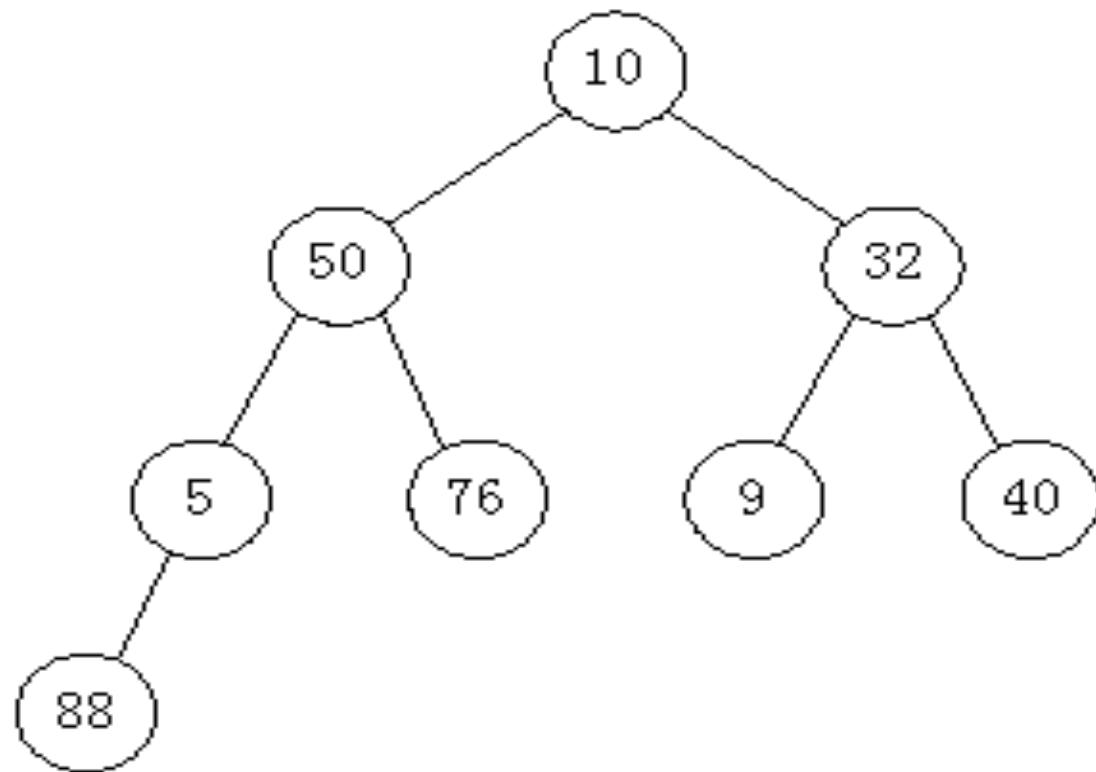


(c) Heap 1



(d) Heap 2

初始状态

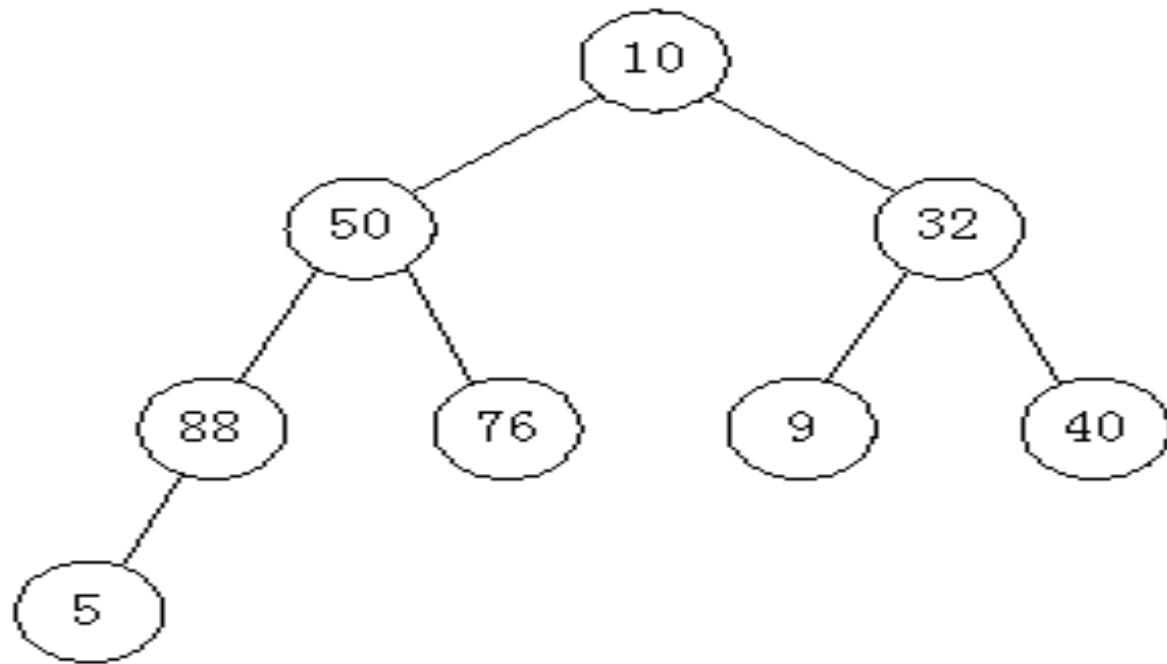


数组

10	50	32	5	76	9	40	88
----	----	----	---	----	---	----	----

(a)

调整结点5后

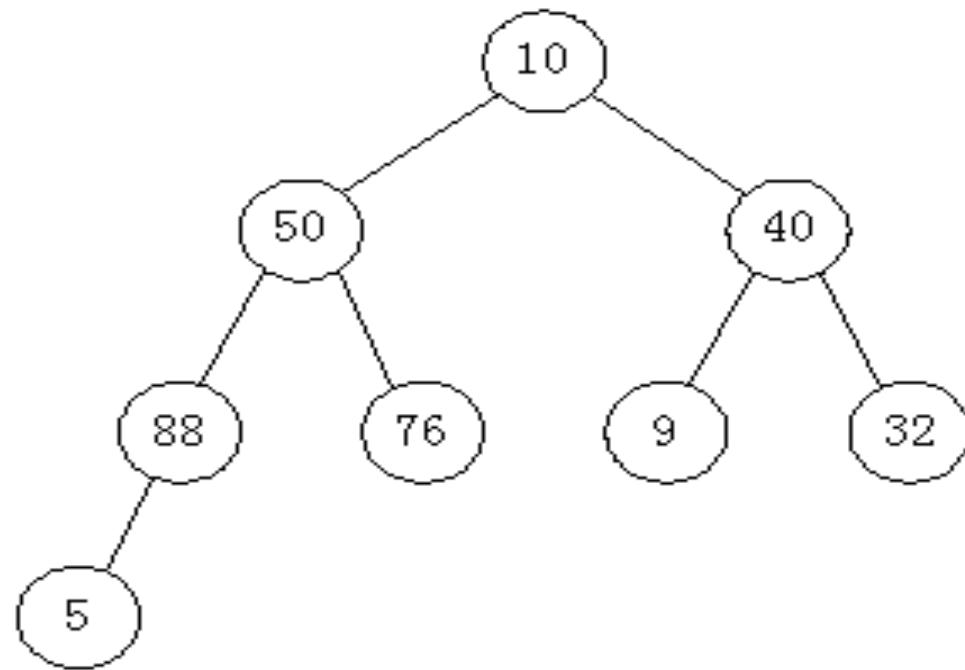


数组

10	50	32	88	76	9	40	5
----	----	----	----	----	---	----	---

(b)

调整结点32后

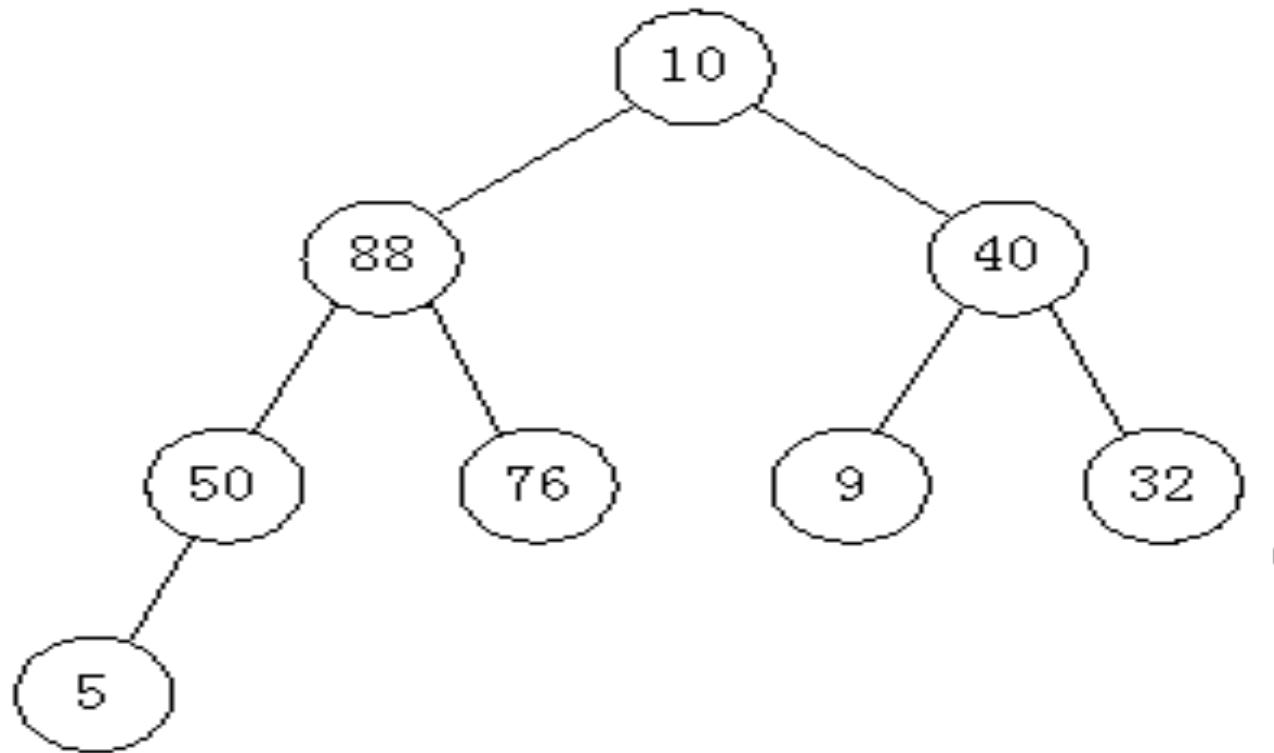


数组

10	50	40	88	76	9	32	5
----	----	----	----	----	---	----	---

(c)

调整结点50后

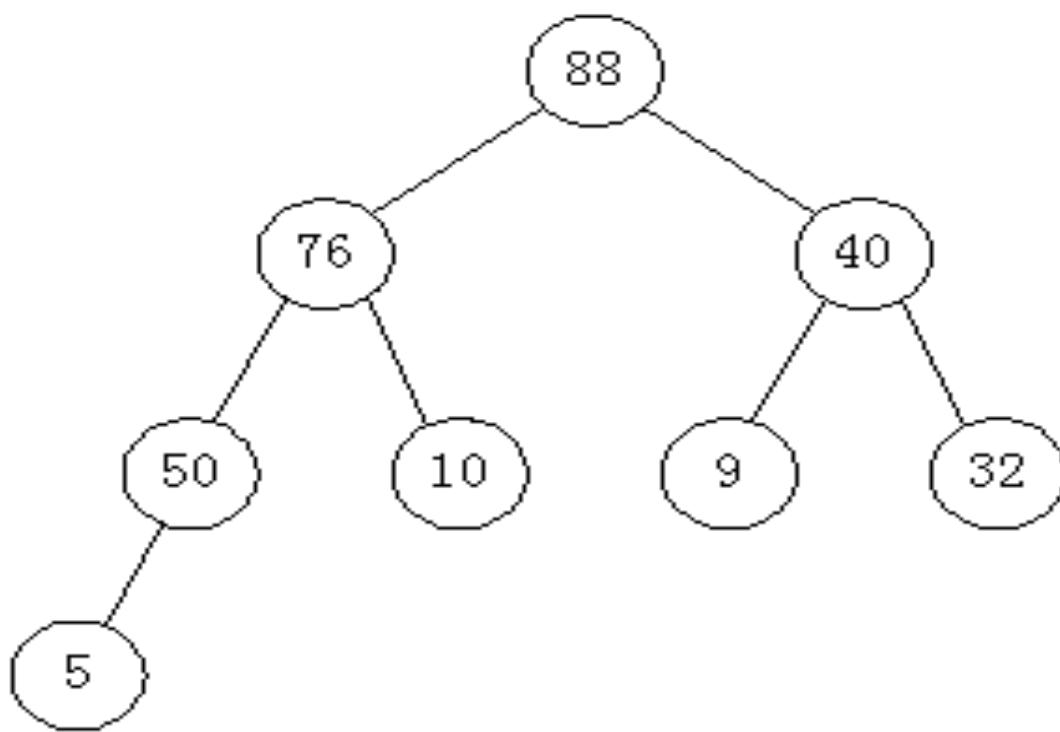


数组

10	88	40	50	76	9	32	5
----	----	----	----	----	---	----	---

(d)

调整结点10后



数组

88	76	40	50	10	9	32	5
----	----	----	----	----	---	----	---

(e)

# BUILD-MAX-HEAP (最大堆构建算法)

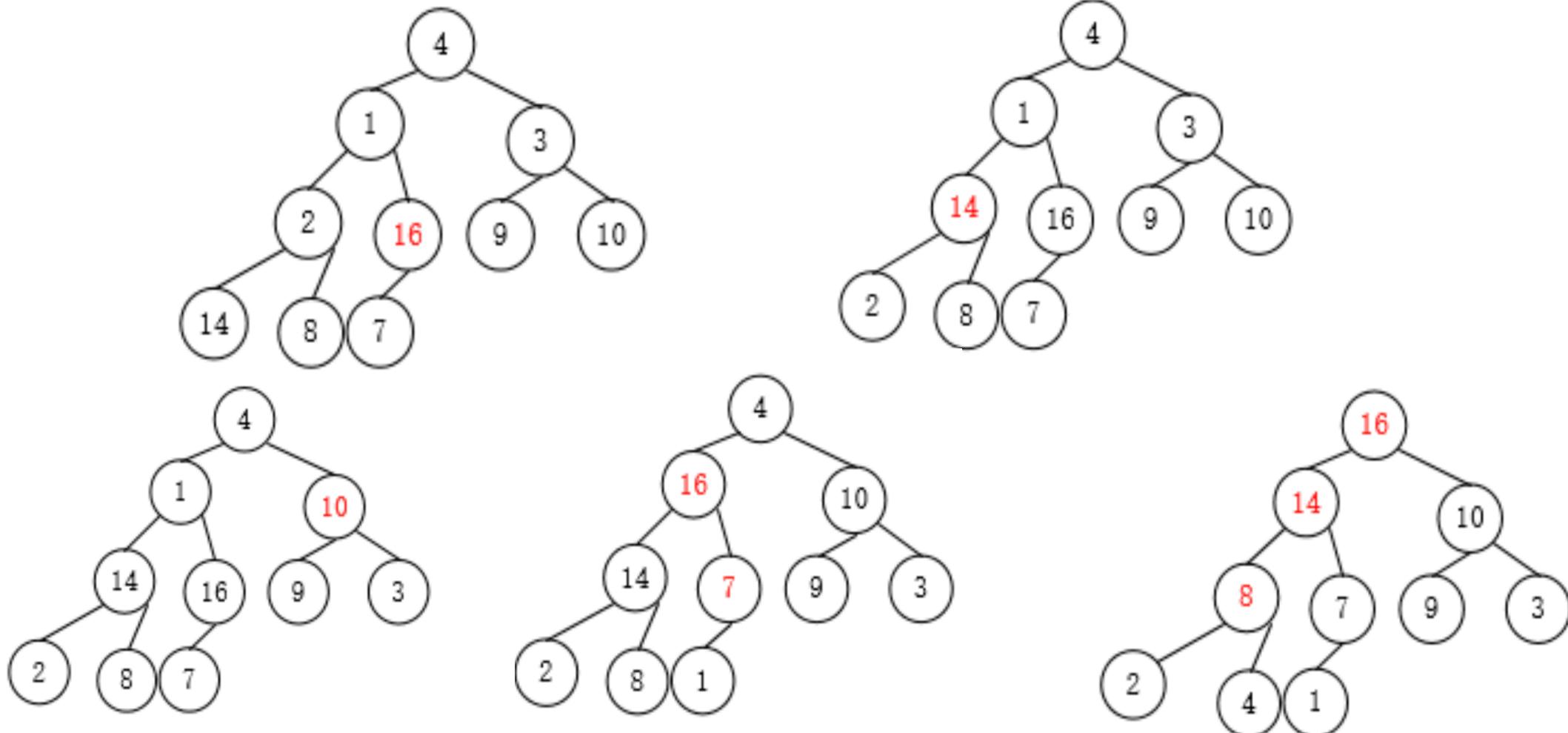
## BUILD-MAX-HEAP

```
BUILD-MAX-HEAP (A)
1 heap-size[A]  $\leftarrow$  length[A]
2 for i  $\leftarrow \lfloor n/2 \rfloor$  downto 1
3   do MAX-HEAPIFY (A, i)
```

- **Loop invariant.** At the start of the for loop, the nodes  $i + 1, i + 2, \dots, n$  each roots of a max-heap.
- **Initialization.** Before the first iteration,  $i = \lfloor n/2 \rfloor$ . Since the nodes  $\lfloor n/2 \rfloor + 1, \dots, n$  are leaves, they are each roots of a max-heap.
- **Maintenance.** The children of node  $i$  are numbered higher than  $i$ , so by the loop-invariant, they are both roots of a max-heap.  $\text{MAX-HEAPIFY}(A, i)$  will then make the tree rooted at node  $i$  a max-heap.
- **Termination.**  $i = 0$ .

# 创建最大堆过程

$A$	4	1	3	2	16	9	10	14	8	7
-----	---	---	---	---	----	---	----	----	---	---



# MAX-HEAPIFY (最大堆算法)

- *heap-size[A]* 是数组A中存储的堆的大小

```
MAX-HEAPIFY
```

```
MAX-HEAPIFY(A, i)
1  l ← LEFT(i)
2  r ← RIGHT(i)
3  if l ≤ heap-size[A] and A[l]> A[i]
4      then largest ← l
5      else largest ← i
6  if r ≤ heap-size[A] and A[r]> A[largest]
7      then largest ← r
8  if largest ≠ i
9      then exchange A[i] ↔ A[largest]
10         MAX-HEAPIFY(A, largest)
```

- Running time:  $O(h)$ , where  $h$  is the height of element  $A[i]$ .

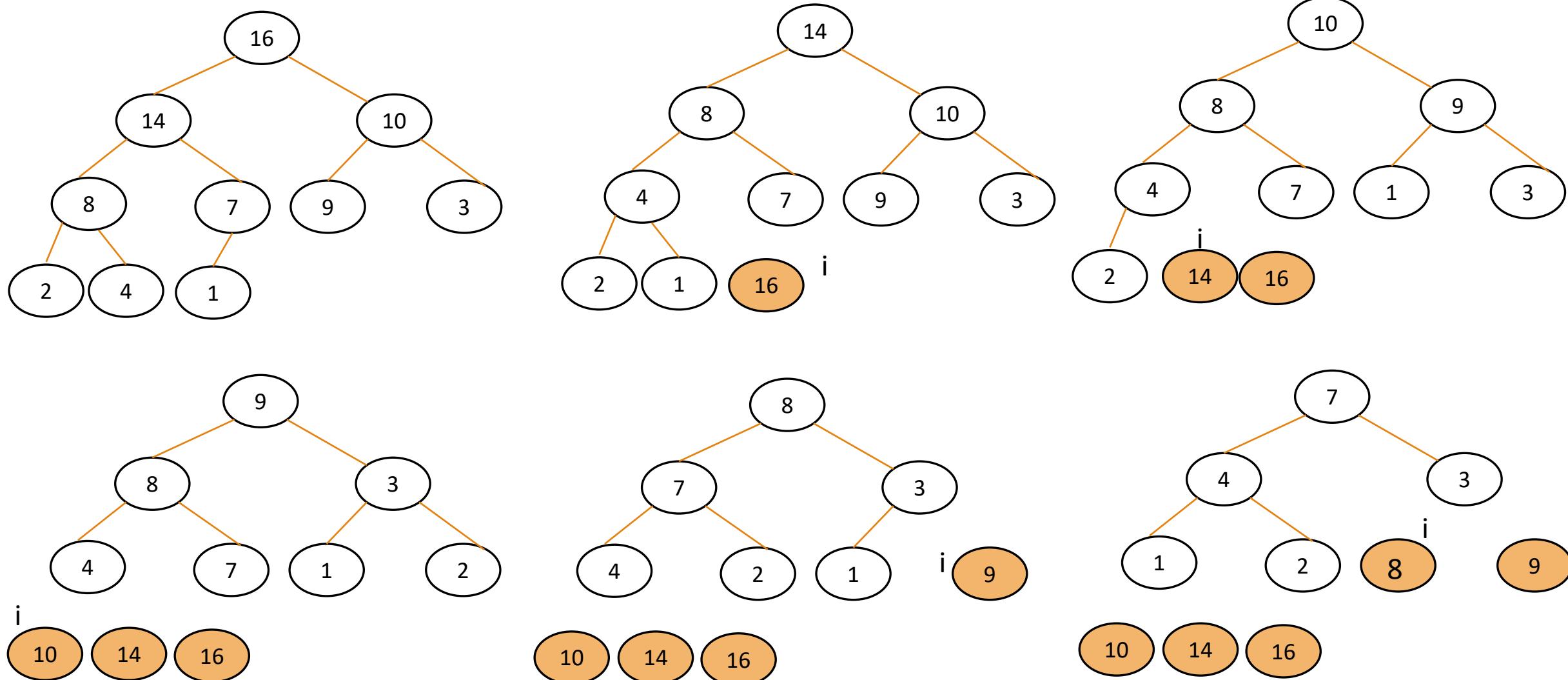
# HEAPSORT (堆排序)

## HEAPSORT

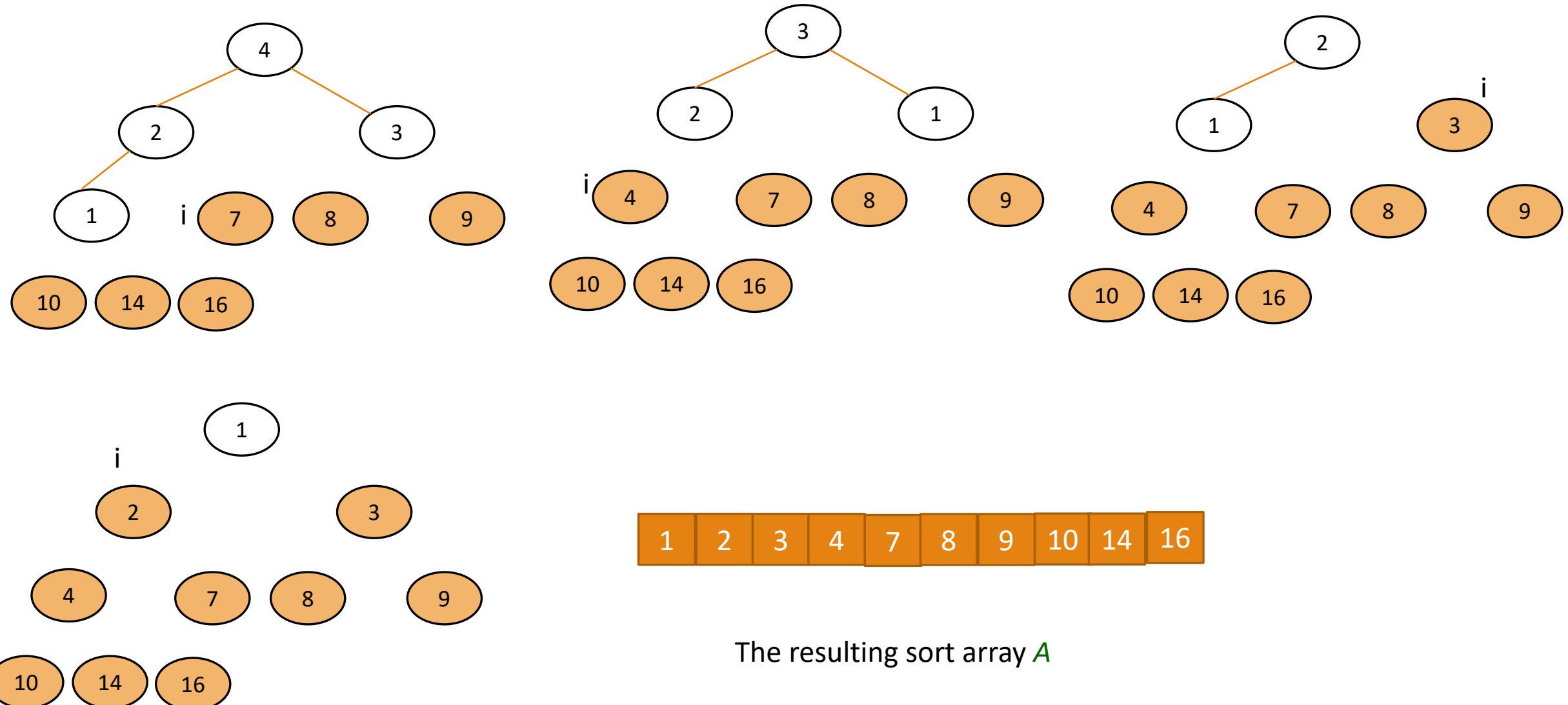
```
HEAPSORT (A)
1  BUILD-MAX-HEAP (A)
2  for i ← length[A] downto 2
3      do exchange A[1] ↔ A[i]
4          heap-size[A] ← heap-size[A]-1
5      MAX-HEAPIFY (A, 1)
```

- Running time.  $O(n \lg n)$  The call to BUILD-MAX-HEAP takes time  $O(n)$  and there are  $n - 1$  calls to MAX-HEAPIFY, each taking  $O(\lg n)$  time.
- In-place. Not Stable.
- Note that even if BUILD-MAX-HEAP ran in  $O(n \lg n)$  time, we'd get the same running time.

# HEAPSORT (堆排序实例)



# HEAPSORT (堆排序实例)



## 5.4.1 大整数乘法

### 大整数乘法问题

- 某些应用中，如当代的密码技术，需要计算超过上千位的二进制数的乘积；
- 假设 $X$ 和 $Y$ 是两个 $n$ 位的二进制数， $n = 2^k$ ，计算 $XY$ ；

### 解决办法

- 按照通常的做法，需要总共 $n^2$ 次乘法计算；
- 考虑分治法，将 $X$ 和 $Y$ 分成相等的两段，每段 $n/2$ 位，即

$$X = A2^{n/2} + B$$

$$Y = C2^{n/2} + D$$

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

- 规模为 $n$ 的原问题转换为4个规模为 $n/2$ 的子问题；

## 5.4.1 大整数乘法

### 复杂度分析

- 计算需要的乘法次数递归方程：

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

- 根据主定理可得到  $W(n) \in \Theta(n^2)$

### 注意

- 虽然采用了分治法，但是时间复杂度并没有降低；
- 回顾主定理，当问题规模减半  $b = 2$ ，合并的复杂度为  $\Theta(n)$ （加减法运算次数）时，子问题数  $a > 2$  时，时间复杂度为  $\Theta(n^{\log_b a})$ ，减少子问题数可降低时间复杂度。

## 5.4.1 大整数乘法

### 改进思路

- $AD + BC = (A - B)(D - C) + AC + BD$
- $AC$ 和 $BD$ 已知，子问题数目从原来的4个变为3个；

### 复杂度分析

- 乘法次数递归方程： $M(n) = 3M(n/2)$ ,  $n > 1$ ;  $M(1) = 1$ ;
- 根据主定理可得到 $M(n) \in \Theta(n^{\log_2 3}) \approx \Theta(n^{1.59})$ ;
- 合并子问题的复杂度：6次规模为 $n$ 的整数加减法操作，可记为 $cn$ ;
- 加减次数递归方程： $A(n) = 3A(n/2) + cn$ ,  $n > 1$ ;  $A(1) = 0$ ;
- 根据主定理可得到 $A(n) \in \Theta(n^{\log_2 3}) \approx \Theta(n^{1.59})$ ;
- 算法总的时间效率为 $\Theta(n^{1.59})$ ，效率有明显的提升；

## 5.4.2 矩阵乘法

### 矩阵相乘问题

假设 $A$ 和 $B$ 是两个 $n$ 阶的矩阵， $n = 2^k$ ，计算 $C = AB$ ；

### 解决办法

- $C_{ij} = a_{i1}b_{1j} + a_{i2}b_{2j} + \dots + a_{in}b_{nj}$
- 计算 $C_{ij}$ 需要 $n$ 次乘法（不考虑加法），计算 $C$ 需要 $n^3$ 次乘法；
- 考虑分治法：

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

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

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

- 规模为 $n$ 的原问题转换为8个规模为 $n/2$ 的子问题；

## 5.4.2 矩阵乘法

### 复杂度分析

- 乘法次数递归方程：

$$M(n) = \begin{cases} 8M(n/2) & n > 1 \\ 1 & n = 1 \end{cases}$$

- 根据主定理可得到  $M(n) \in \Theta(n^{\log 8}) = \Theta(n^3)$

### 注意

- 与传统方法比，时间复杂度并没有降低；
- 同样地，考虑减少子问题数以降低时间复杂度。

## 5.4.2 Strassen (施特拉森) 矩阵乘法

- 分块矩阵相乘:

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

- 中间结果:

$$M_1 = A_{11}(B_{12} - B_{22}), M_2 = (A_{11} + B_{12})B_{22}$$

$$M_3 = (A_{21} + B_{22})B_{11}, M_4 = A_{22}(B_{21} - B_{11})$$

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

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

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

- 计算最终结果:

$$C_{11} = M_5 + M_4 - M_2 + M_6$$

$$C_{12} = M_1 + M_2, C_{21} = M_3 + M_4$$

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

## 5.4.2 Strassen (施特拉森) 矩阵乘法——时间复杂度分析

- 合并子问题的复杂度：矩阵加法 ( $(n/2)^2$ 个元素相加) 18次；
- 乘法计算总次数递归方程：

$$M(n) = \begin{cases} 7M(n/2) & n > 1 \\ 1 & n = 1 \end{cases}$$

- 根据主定理可得到  $M(n) \in \Theta(n^{\log_2 7}) \approx \Theta(n^{2.807})$
- 加法计算总次数递归方程：

$$A(n) = \begin{cases} 7A(n/2) + 18(n/2)^2 & n > 1 \\ 0 & n = 1 \end{cases}$$

- 根据主定理可得到  $A(n) \in \Theta(n^{\log_2 7}) \approx \Theta(n^{2.807})$
- 算法总的时间效率为  $\Theta(n^{2.807})$ ，效率有明显的提升；

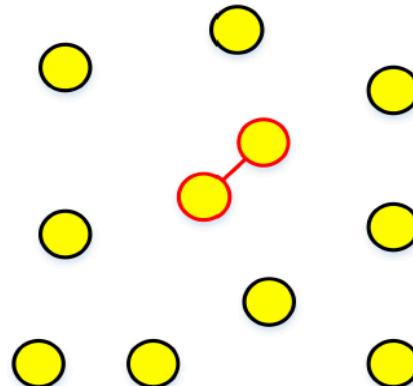
## 5.5.1 最接近点对问题

### 问题描述

- 在一个包含 $n$ 个点的集合中，找出距离最近的两个点。
- 航空交通控制人员监测最有可能发生碰撞的飞机。
- 区域邮政管理人员寻找地理位置最近的邮局。

---

### 示例



## 5.5.1 最接近点对问题

### 问题回顾

- 平面上有 $n$ 个点， $P_1, P_2, \dots, P_n$ ， $n > 1$ ， $P_i$ 的直角坐标是 $(x_i, y_i)$ ，求距离最近的两个点之间的距离。 $P_i$ 和 $P_j$ 距离计算如下：

$$d(P_i, P_j) = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$$

- 蛮力查找算法距离计算的总次数为 $C_n^2 = n(n - 1)/2$ 。

# 蛮力算法

算法 BruteForceClosestPoints( $P$ )

//输入:一个 $n(n \geq 2)$ 个点的列表 $P, P_1=(x_1, y_1), \dots, P_n=(x_n, y_n)$

//输出:两个最近点的下标,index1和index2

$d_{min} = \infty$

for  $i = 1$  to  $n-1$  do

    for  $j = i+1$  to  $n$  do

$d = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$

        if  $d < d_{min}$

$d_{min} = d ; index1 = i ; index2 = j$

return  $index1, index2$

计算距离:  $n(n-1)/2$  次

If 语句最多执行 $n$ 次

$n(n-1)/2+n$

总结:此算法的时间复杂度是 $O(n^2)$

# 一维空间找最接近点对

---

怎么样在一条线上找最邻近的点对?

- (1) 用 $O(n \log n)$ 时间对它们排序.
- (2) 走过排好序的表, 计算每一个点到跟在它后面的点的距离, 容易看出这些距离的最小值. 时间复杂性为:  $n-1$

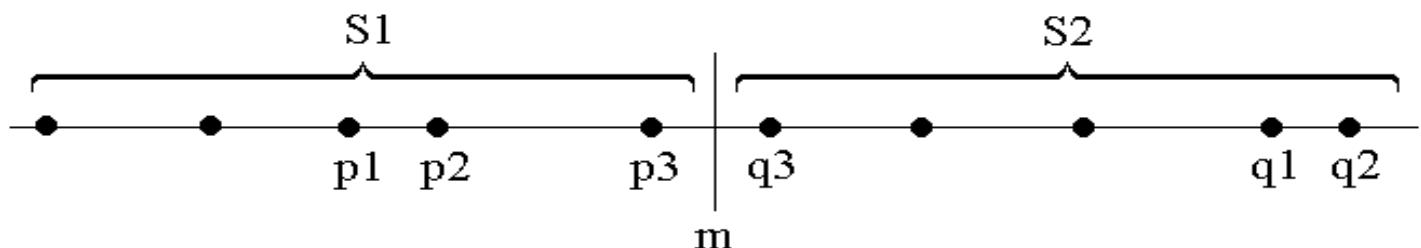
$$\text{故 } T(n) = O(n \log n) + (n-1) = O(n \log n)$$

显然, 这种方法不能推广到二维的情形。

故尝试用分治法来求解, 并希望推广到二维的情形。

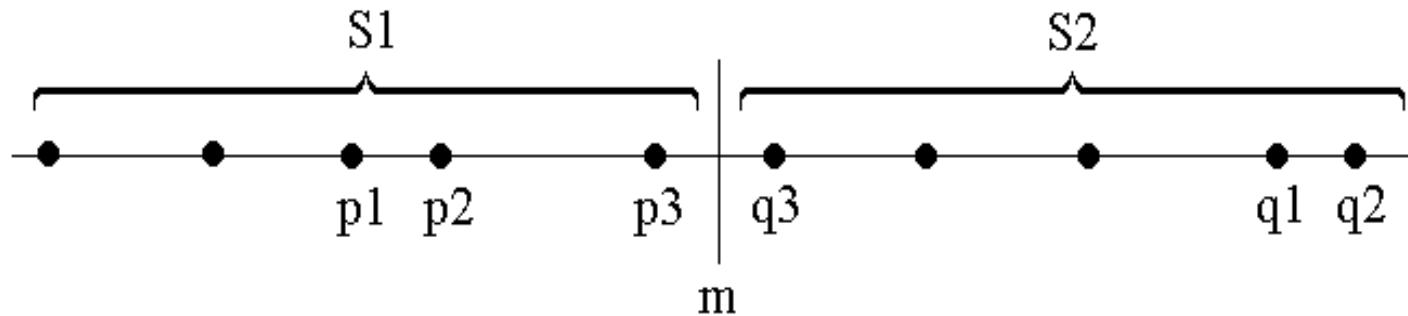
# 分治策略下一维的情形

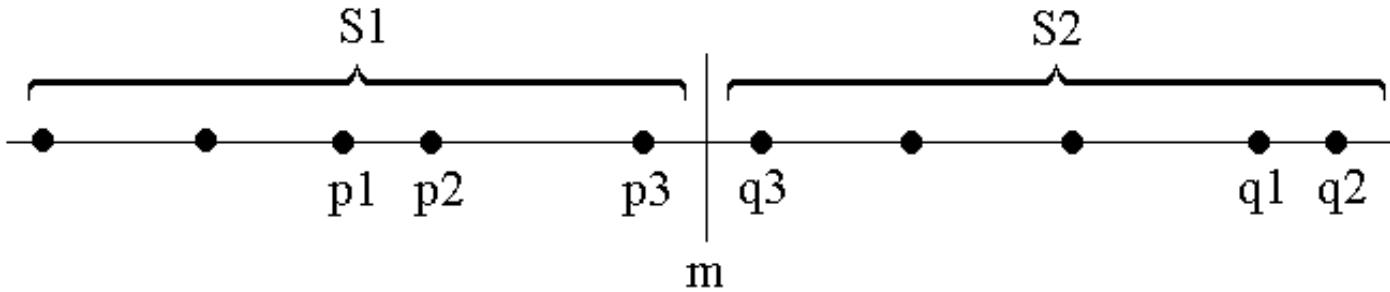
- 先把 $x_1, x_2, \dots, x_n$ 排好序，再设 $T(n)=O(n\log n)$ 。
- 原问题不太好解决，可以把原问题分解为若干个规模较小的相同子问题，再把子问题的解合并为原问题的解。
- 假设我们用x轴上某个点m将S划分为2个子集 $S_1$ 和 $S_2$ ，基于**平衡子问题**的思想，用S中各点坐标的中位数来作分割点。
- 递归地在 $S_1$ 和 $S_2$ 上找出其最接近点对 $\{p_1, p_2\}$ 和 $\{q_1, q_2\}$ ，并设 $d=\min\{|p_1-p_2|, |q_1-q_2|\}$ ，S中的最接近点对或者是 $\{p_1, p_2\}$ ，或者是 $\{q_1, q_2\}$ ，或者是某个 $\{p_3, q_3\}$ ，其中 $p_3 \in S_1$ 且 $q_3 \in S_2$ 。



## 大致算法：

- 1、用 $S$ 中各点坐标的中位数来作分割点，将 $S$ 分成 $S_1$ 和 $S_2$ 。
- 2、递归地在 $S_1$ 和 $S_2$ 上找出其最接近点对 $\{p_1, p_2\}$ 和 $\{q_1, q_2\}$ 。
- 3、合并： $S$ 中的最接近点对在 $\{p_1, p_2\}$ 、 $\{q_1, q_2\}$ 、 $\{p_3, q_3\}$ 中， $p_3 \in S_1$ 且 $q_3 \in S_2$ 。





- ◆ 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 $p_3 - m < d$ ,  $q_3 - m < d$ , 即 $p_3 \in (m-d, m]$ ,  $q_3 \in (m, m+d]$ 。
- ◆ 由于在S1中, 每个长度为 $d$ 的半闭区间至多包含一个点(否则必有两点距离小于 $d$ ), 并且 $m$ 是S1和S2的分割点, 因此 $(m-d, m]$ 中至多包含S中的一个点。由图可以看出, 如果 $(m-d, m]$ 中有S中的点, 则此点就是S1中最大点。同理, 如果 $(m, m+d]$ 中有S中的点, 则此点就是S2中最小点。
- ◆ 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S1的解和S2的解合并成为S的解。

```

public static double Cpair1(S,d) //找S中最接近点对的距离d
{
    n=|S|; //S中点的个数
    if(n<2)
        d=∞;
    m=S中各点坐标的中位数;
    构造S1和S2;
    //S1={x∈S | x≤m},S2={x∈S | x>m}
    Cpair1(S1,d1);
    Cpair1(S2,d2);
    p=max(S1);
    q=min(S2);
    d=min(d1,d2,q-p);
    return true;
}

```

$O(n)$

$2T(n/2)$

$O(n)$

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

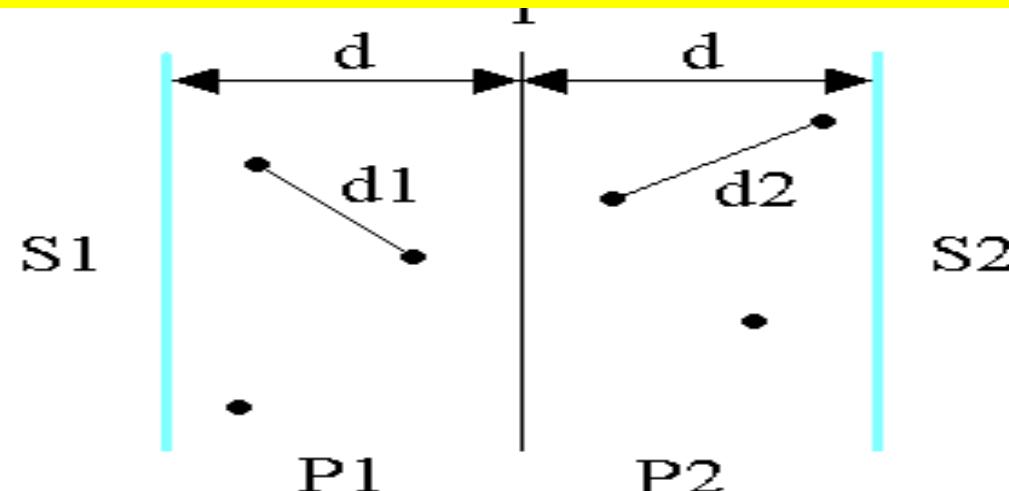
$T(n)=O(n\log n)$

# 最接近点对问题

- 下面来考虑二维的情形。

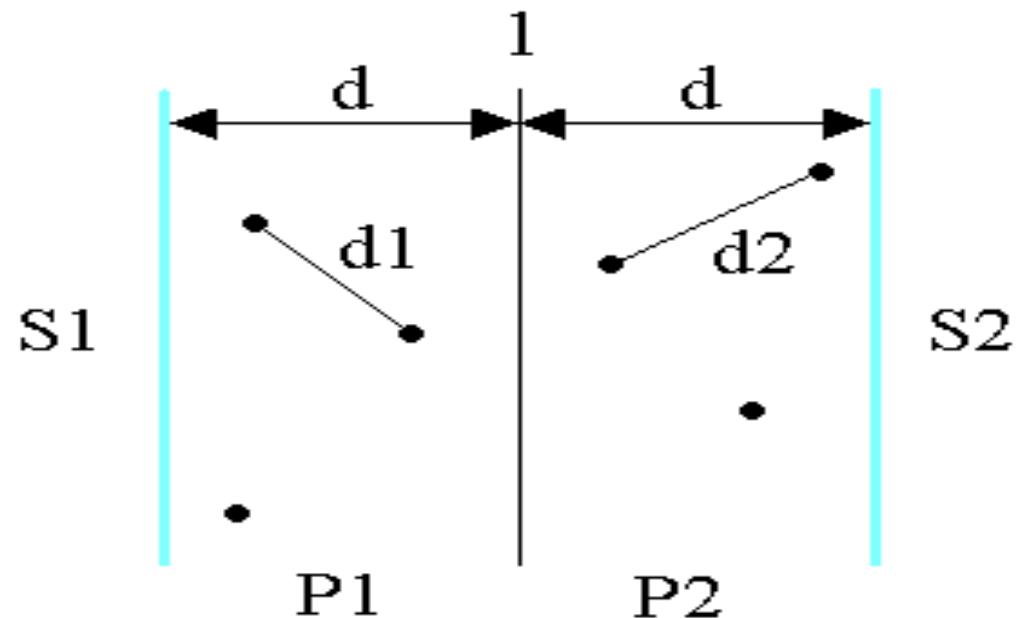
- 选取一垂直直线 $x=m$ 来作为分割直线。其中 $m$ 为 $S$ 中各点 $x$ 坐标的中位数。由此将 $S$ 分割为 $S_1$ 和 $S_2$ 。
- 递归地在 $S_1$ 和 $S_2$ 上找出其最小距离 $d_1$ 和 $d_2$ ，并设 $d=\min\{d_1, d_2\}$ ， $S$ 中的最接近点对或者是 $d$ ，或者是某个 $\{p, q\}$ ，其中 $p \in S_1$ 且 $q \in S_2$ 。

► 能否在线性时间内找到 $p, q$ ？



能否在线性时间内找到 $p_3, q_3$

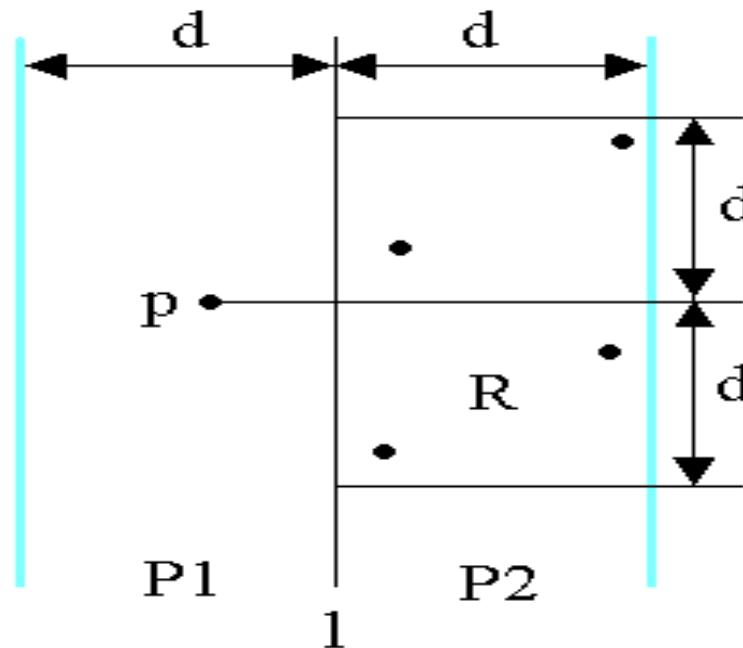
第一步筛选：如果最近点对由 $S_1$ 中的 $p_3$ 和 $S_2$ 中的 $q_3$ 组成，则 $p_3$ 和 $q_3$ 一定在划分线 $L$ 的距离 $d$ 内。



需要计算**P1**中的每个点与**P2**中的每个点的距离？

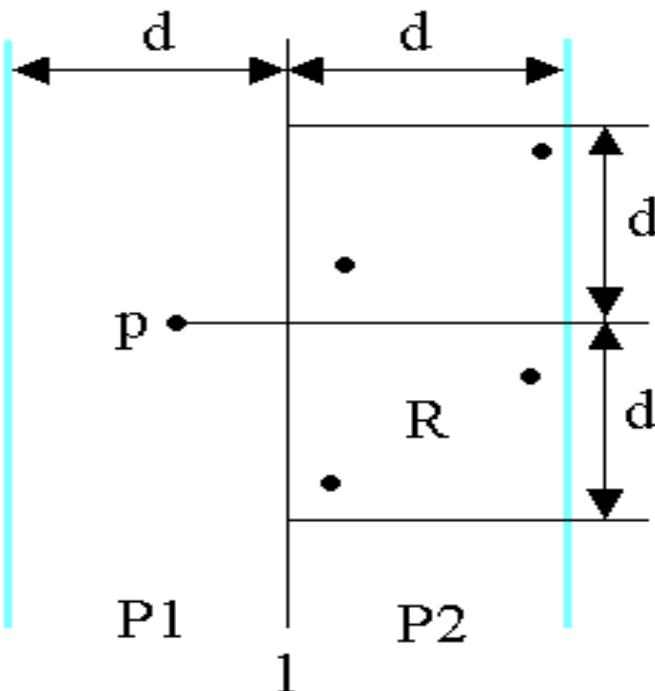
并不必计算P1中的每个点与P2中的每个点的距离！ $O(n^2)$

- 第二步筛选：考虑P1中任意一点p，它若与P2中的点q构成最近点对的候选者，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



# R中的点具有稀疏性

- 重要观察结论：P<sub>2</sub>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。



- 重要结论：在分治法的合并步骤中最只需要检查 $6 \times n/2 = 3n$ 个候选点对！

# R中最多只有6个S中的点

证明：将矩形R划分成 $(d/2) \times (2d)$ 的小矩形。

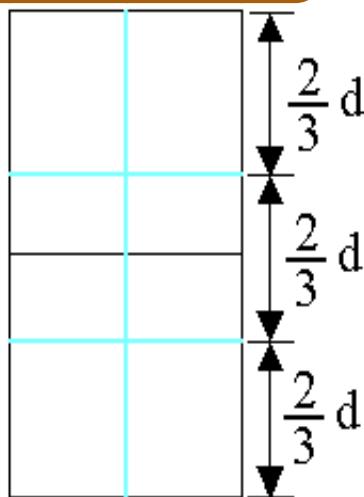
\* 鸽舍原理（也称抽屉原理）

将它的 $n+1$ 个球，放入 $n$ 个抽屉，则一定有一个抽屉内至少有2个球。

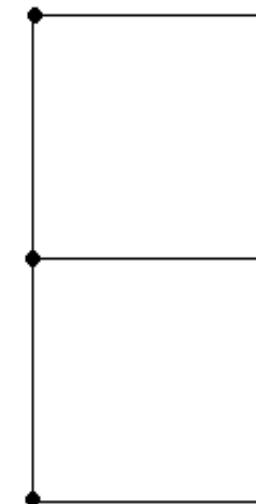
若矩形R中有 $n+1$ 个球，放入 $n$ 个抽屉，则一定有一个抽屉内至少有2个球。

若矩形R中有 $n+1$ 个球，放入 $n$ 个抽屉，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



(a)



(b)

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

$\text{distance}(u, v) < d$ 。这与 $d$ 的意义相矛盾。

# 如何确定要检查哪6个点

- P2中与点p最接近这6个候选点的纵坐标与p的纵坐标相差不超过d.
- 因此，若将P1和P2中所有S中点按其y坐标排好序，则对P1中所有点，对排好序的点列作一次扫描，就可以找出所有最接近点对的候选者。对P1中每一点最多只要检查P2中排好序的相继6个点。

```
double cpair2(S)
```

```
{
```

```
    n=|S|;
```

```
    if (n < 2) return 0;
```

1、 m=S中各点x间坐标的中位数；

构造s1和s2；

```
//S1={p∈S|x(p)≤m},
```

```
S2={p∈S|x(p)>m} O(n)
```

2、 d1=cpair2(S1);

```
d2=cpair2(S2); 2T(n/2)
```

3、 dm=min(d1,d2); 常数时间

4、 设P1是S1中距垂直分割线1的距离在dm之内的所有点组成的集合；

P2是S2中距分割线1的距离在dm之内所有点组成的集合；

将P1和P2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； O(n)

5、 通过扫描X以及对于X中每个点检查Y中与其距离在dm之内的所有点(最多6个)可以完成合并； O(n)

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d1是按这种扫描方式找到的点对间的最小距离；

6、 d=min(dm, d1)；

return d; 常数时间

```
}
```

# 伪代码：

```
1: function Efficient_Closest_Pair( $X, Y$ ) // $X$ 和 $Y$ 分别为点的已经排序的横纵坐标
2:   if  $n \leq 3$  then
3:     return 由蛮力计算法求出的最小距离;
4:   else
5:     根据 $X$ 的中位数 $m$ , 划分得到 $X_L$ 和 $X_R$ , 进而从 $Y$ 中抽取得到 $Y_L$ 和 $Y_R$ ;
6:      $d_L \leftarrow \text{Efficient\_Closest\_Pair } (X_L, Y_L); d_R \leftarrow \text{Efficient\_Closest\_Pair } (X_R, Y_R);$ 
7:      $d \leftarrow \min(d_L, d_R); dminsq \leftarrow d^2;$ 
8:     从 $Y$ 中去除不在距离中位线 $l$ 为 $d$ 的窄缝里点, 得到点集 $Y'[1\dots m]$ ;
9:     for  $i \leftarrow 0 \rightarrow m - 1$  do
10:       $k \leftarrow i + 1;$ 
11:      while  $k \leq 7$  and  $(Y'[k].y - Y'[i].y)^2 < d^2$  do
12:         $dminsq \leftarrow \min(dminsq, (Y'[k].x - Y'[i].x)^2 + (Y'[k].y - Y'[i].y)^2);$ 
13:         $k \leftarrow k + 1;$ 
14:      end while
15:    end for
16:    return  $\sqrt{dminsq};$ 
17:  end if
18: end function
```

# 复杂度分析

- ①、⑤用了 $O(n)$ 时间；
- ②用了 $2T(n/2)$ 时间
- ③、⑥用了常数时间
- ④在预排序的情况下用时 $O(n)$

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

- 假设采用复杂度为 $\Theta(n \log n)$ 的排序算法预排序，总体的复杂度为 $\Theta(n \log n) + \Theta(n \log n) \in \Theta(n \log n)$ ；

## 注意

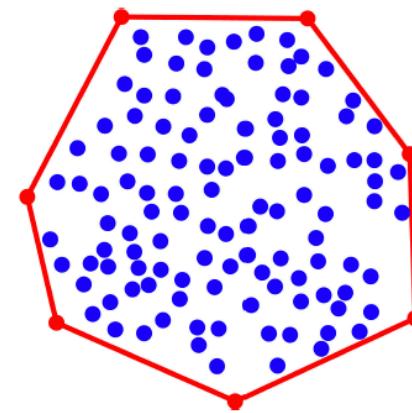
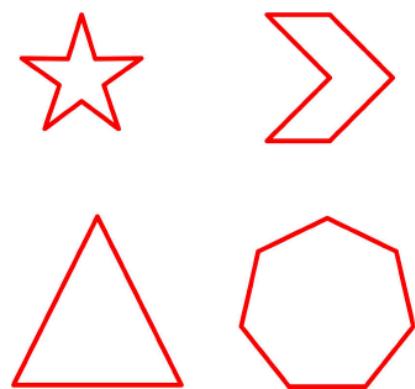
- 如果不进行预排序，需要在每次的递归调用中排序，运行时间递归方程为： $T(n) = 2T(n/2) + O(n \log n)$
- 计算可得 $T(n) \in O(n \log^2 n)$

# 凸包问题

## 问题描述

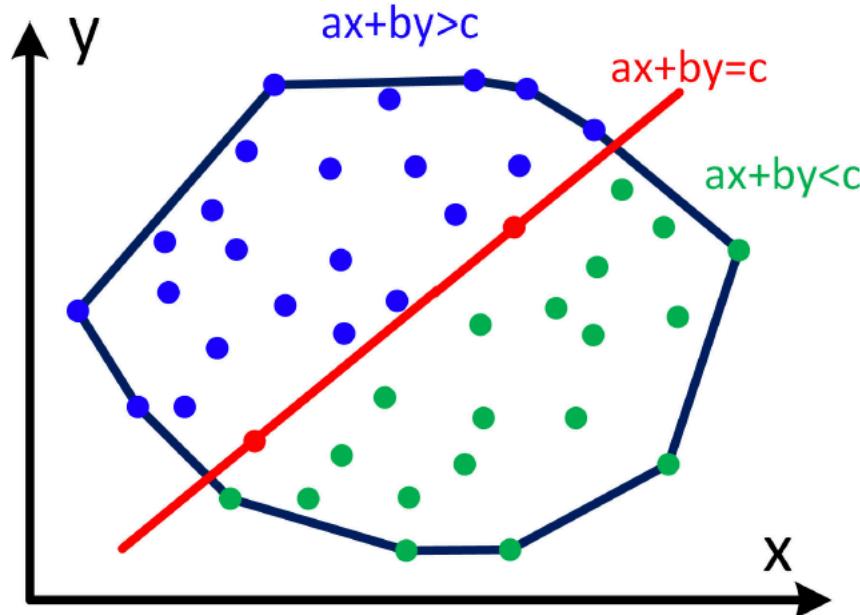
- 凸集合：对于平面上的一个点的集合，如果任意两个点的连线都在集合中，就称为凸集合。
- 凸包问题：寻找包含所有点的最小凸集合。

## 示例



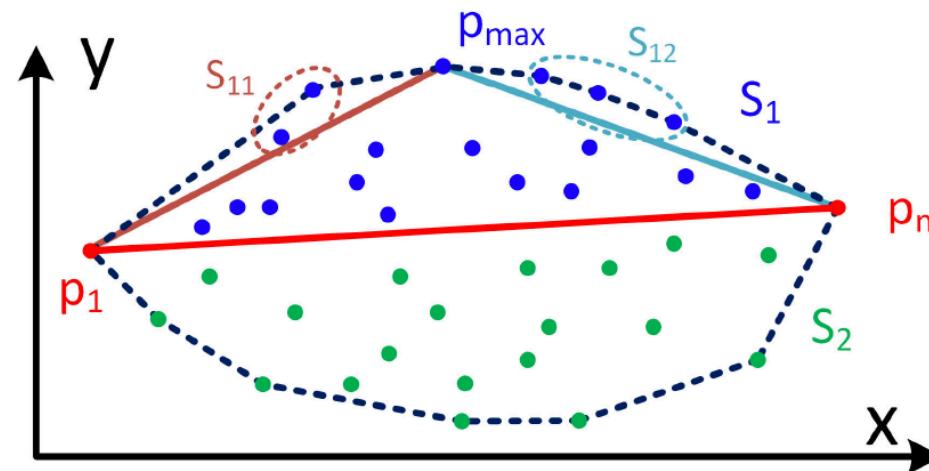
## 凸包问题——蛮力法求解

- 对于 $n$ 个点集合中的两个点 $p_i$ 和 $p_j$ ，当且仅当该集合中的其他点都位于穿过这两个点的直线的同一边时， $p_i$ 和 $p_j$ 的连线是该集合凸包边界的一部分。
- 一共有 $C_n^2$ 条线，对于每一条线，需要检查其他 $n - 2$ 个点的符号，比较次数为 $n(n - 1)(n - 2)/2$ ；



# 凸包问题——分治法求解

- 集合 $S$ 是平面上 $n > 1$ 个点,  $p_1(x_1, y_1), p_2(x_2, y_2) \dots p_n(x_n, y_n)$ , 按照 $x$ 和 $y$ 的大小排序, 则最左边的点 $p_1$ 和最右边的点 $p_n$ 一定是凸包的顶点;
- $p_1p_n$ 线将 $S$ 分为 $S_1$ 和 $S_2$ , 寻找 $p_1 \cup S_1 \cup p_n$ 和 $p_1 \cup S_2 \cup p_n$ 的凸包;
- 找到 $S_1$ 中距离 $p_1p_n$ 线最远的点 $p_{max}$ , 并可得到集合 $S_{11}$ 和 $S_{12}$ ;
- 寻找集合 $p_1 \cup S_{11} \cup p_{max}$ 和集合 $p_{max} \cup S_{12} \cup p_n$ 的凸包;
- 最坏情况下, 子集的规模只能少一个点, 效率为 $\Theta(n^2)$



## 作业 2

- 阅读
  - 2.4, 4.1, 5.1-5.2, 5.4-5.5, 6.4
- 习题
  - 5.2: 11
  - 5.4: 7
  - 5.5: 1 (a)