

# 数据结构

## Data Structure

Xia Tian

Email: [xiat\(at\)ruc.edu.cn](mailto:xiat(at)ruc.edu.cn)

Renmin University of China

## 内部排序大纲

- 排序的基本概念
- 具体排序方法
  1. 插入排序：直接插入排序
  2. 插入排序：折半插入排序
  3. 插入排序：希尔排序
  4. 交换排序：冒泡排序
  5. 交换排序：快速排序
  6. 选择排序：简单选择排序
  7. 选择排序：堆排序
  8. 归并排序：二路归并
  9. 基数排序

- 对一个数据元素集合或序列重新排列成一个按数据元素某个项值有序的序列就是排序。

\* 例如将关键字序列:

52, 49, 80, 36, 14, 58, 61, 23, 97, 75

调整为

14, 23, 36, 49, 52, 58, 61, 75, 80, 97

\* 再如将:

< Susie, 26 >, < Jack, 22 >, < Michel, 25 >, < Richard, 25 >

调整为:

< Jack, 22 >, < Michel, 25 >, < Richard, 25 >, < Susie, 26 >



# 排序的稳定性

- 请注意刚才第二个序列的排序结果不唯一!

< Susie, 26 >, < Jack, 22 >, < Michel, 25 >, < Richard, 25 >

< Jack, 22 >, < Michel, 25 >, < Richard, 25 >, < Susie, 26 >

< Jack, 22 >, < Richard, 25 >, < Michel, 25 >, < Susie, 26 >

- 排序算法的稳定性

- \* 若存在相同的关键字, 对应位置的记录在排序后仍然保持原来的顺序, 则称所使用的排序方法是稳定的。反之称为不稳定的。

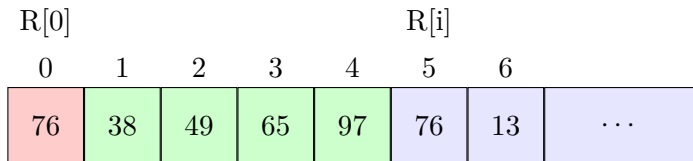
## 插入排序

- 直接插入排序
- 折半插入排序
- 希尔排序



# 直接插入排序

对于要插入的元素  $R[i]$ , 从  $R[i - 1]$  起向前进行顺序查找, 当  $R[j - 1]$  小于  $R[i]$  时停止, 插入位置为  $R[j]$ 。注意在顺序表中要移动元素实现元素的插入。



设置“哨兵”存储  $R[i]$ ,  
只要待比较的元素大于  $R[0]$ ,  
就继续往前比较, 从而实现  
 $R[j] \cdots R[i - 1]$  的后移。

$j$ , 指示插入的位置,  
 $R[j - 1] \leq R[i]$   
(稳定的排序方法)

## 直接插入排序举例



---

47

38

65

97

13

27

47

## 直接插入排序举例



47	38	65	97	13	27	47
47	38	65	97	13	27	47





## 直接插入排序举例

47	38	65	97	13	27	47
47	38	65	97	13	27	47
38	47	65	97	13	27	47

## 直接插入排序举例



47	38	65	97	13	27	47
47	38	65	97	13	27	47
38	47	65	97	13	27	47
38	47	65	97	13	27	47

## 直接插入排序举例



47	38	65	97	13	27	(47)
47	38	65	97	13	27	(47)
38	47	65	97	13	27	(47)
38	47	65	97	13	27	(47)
38	47	65	97	13	27	(47)

## 直接插入排序举例



47	38	65	97	13	27	(47)
47	38	65	97	13	27	(47)
38	47	65	97	13	27	(47)
38	47	65	97	13	27	(47)
38	47	65	97	13	27	(47)
13	38	47	65	97	27	(47)

## 直接插入排序举例



47	38	65	97	13	27	47
47	38	65	97	13	27	47
38	47	65	97	13	27	47
38	47	65	97	13	27	47
38	47	65	97	13	27	47
13	38	47	65	97	27	47
13	27	38	47	65	97	47



## 直接插入排序举例

47	38	65	97	13	27	(47)
47	38	65	97	13	27	(47)
38	47	65	97	13	27	(47)
38	47	65	97	13	27	(47)
38	47	65	97	13	27	(47)
13	38	47	65	97	27	(47)
13	27	38	47	65	97	(47)
13	27	38	47	(47)	65	97

注意：两个 47 的位置 (带有圆圈和不带有圆圈)。



# 直接插入排序算法分析

- 空间效率: 用一个辅助单元
- 时间效率: 时间复杂度为  $O(n^2)$ 
  - \* 进行了  $n-1$  次向有序表插入记录的操作, 每趟是“比较 + 移动”
  - \* 最好情况: 记录按关键字正序
    - $n-1$  次比较, 0 次移动
  - \* 最差情况: 记录按关键字逆序
    - 比较次数:  $\sum_{i=2}^n i = 2 + 3 + \dots + n = \frac{(n+2) \times (n-1)}{2}$
    - 移动次数:  $\sum_{i=2}^n (i+1) = \frac{(n+4) \times (n-1)}{2}$
  - \* 平均情况: 不妨取上述各值的平均, 可知比较和移动次数约  $n^2/4$
- 是稳定的排序方法



## 希尔排序的思想

对于元素数量较少，或者基本有序的待排序序列，直接插入排序的效率不错。

- 根据增量  $d$  分割出子序列
- 对子序列进行直接插入排序
- 增量  $d$  的选择
  - \* Shell 最初的方案:  $d = n/2, d = d/2, \dots, d = 1$
  - \* Knuth 的方案:  $d = d/3 + 1$
  - \* 其它:  $d$  为奇数;  $d$  互质  $\dots$



# 希尔排序举例 I



第 1 趟:  $d_1 = 5$

49	38	65	97	76	13	27	49	55	04
13					49				
	27					38			
		49					65		
			55					97	
				04					76

- 观察 04: 跳跃式的往前移
- 观察 49: 希尔排序不稳定

## 希尔排序举例 II



第 2 趟:  $d_2 = 2$

13	27	49	55	04	49	38	65	97	76
04		13		38		49		97	
	27		49		55		65		76



# 希尔排序分析

- 希尔排序的时间性能优于直接插入排序!
- 希尔排序开始时增量  $d$  较大 (这使得分组较多, 每组记录少), 故各组内直接插入较快, 后来增量  $d$  渐小 (各组记录渐多), 但组内元素已经比较接近有序状态, 所以新的一趟排序过程也比较快。
- 希尔排序的复杂度分析很复杂
  - \* 在特定情况下可以准确估算比较、移动次数, 但是考虑与增量之间的依赖关系, 并给出完整的数学分析, 目前还做不到
  - \* 在增量序列为  $\delta[k] = 2^{t-k+1}$  时, 希尔排序的时间复杂度为  $O(n^{3/2})$
  - \* Knuth 的统计结论是, 平均比较次数和对象平均移动次数在  $n^{1.25}$  与  $1.6 \cdot n^{1.25}$  之间。

### 希尔排序增量序列的取法

目前尚未有工作求得一种最好的增量序列。但需注意的是：应使增量序列中的值没有除 1 之外的公因子，并且最后一个增量必须为 1。

# 练习



分析以下关键字的希尔排序过程：

0	1	2	3	4	5	6	7
47	38	47	9	78	13	27	66

## 练习



分析以下关键字的希尔排序过程：

0	1	2	3	4	5	6	7
47	38	47	9	78	13	27	66

47 13 27 9 78 38 47 66

# 练习



分析以下关键字的希尔排序过程：

0	1	2	3	4	5	6	7
47	38	47	9	78	13	27	66

47 13 27 9 78 38 47 66

27 9 47 13 47 38 78 66

## 练习



分析以下关键字的希尔排序过程：

0	1	2	3	4	5	6	7
47	38	47	9	78	13	27	66

47 13 27 9 78 38 47 66

27 9 47 13 47 38 78 66

9 13 27 38 47 47 66 78



## 交换排序

- 冒泡排序
- 快速排序

# 冒泡排序 (Bubble Sort) I



## 算法思想

比较相邻两个结点, 较大的结点往下移, 较小的往上移, 使得每轮比较结束后最大的沉到最后.



## 冒泡排序 (Bubble Sort) II

49	38	38	38	38	13	13
38	49	49	49	13	27	27
65	65	65	13	27	38	38
97	76	13	27	49	49	
76	13	27	49	49		
13	27	49	65			
27	49	76				
49	97					

第 1 列为原始序列；第 2 列为第 1 趟排序后的结果，第 3 列为第 2 趟排序后结果 …

最后一个数字为本趟排序挑选出的最大数字。



# 冒泡排序算法分析

- 最好情况下, 初始状态是递增有序的
  - \* 扫描一趟, 关键字的比较次数为  $n - 1$ , 记录移动 0 次
- 最坏情况下, 初始状态是反序的
  - \* 扫描  $n - 1$  趟, 第  $i$  趟扫描要进行  $n - i$  次关键字的比较, 每次比较后进行记录移动, 故有:

$$\text{比较次数: } \sum_{i=1}^{n-1} (n - i) = \frac{n \times (n - 1)}{2}$$

$$\text{移动次数: } \sum_{i=1}^{n-1} 3(n - i) = \frac{3 \times n \times (n - 1)}{2}$$

- 稳定的



# 快速排序—冒泡法的改进




## 算法思想

通过一趟排序将待排序记录分割成独立的两部分，其中一部分的关键字均小于另一部分的关键字，然后分别对这两部分记录继续分别进行排序即可。

0	1	2	3	4	5	6	7
27	38	13	49	76	97	65	49

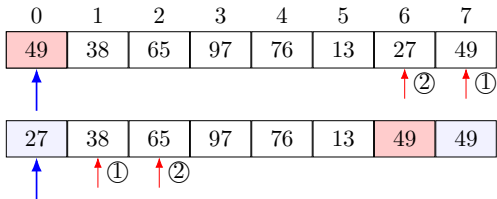
SEE: <https://www.itcodemonkey.com/article/11276.html>

## 快速排序举例 (单次划分过程)

0	1	2	3	4	5	6	7
49	38	65	97	76	13	27	49
						 ②	 ①

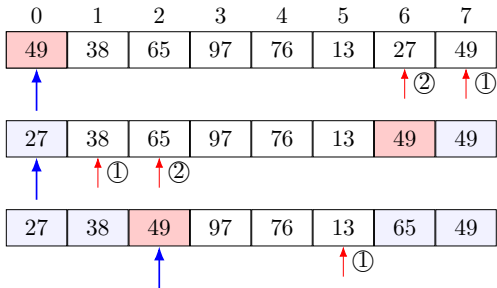
其中：带圆圈的数字表示比较的次序；蓝色箭头表示枢轴元素的位置；红色箭头表示要比较的元素位置。

## 快速排序举例 (单次划分过程)



其中：带圆圈的数字表示比较的次序；蓝色箭头表示枢轴元素的位置；红色箭头表示要比较的元素位置。

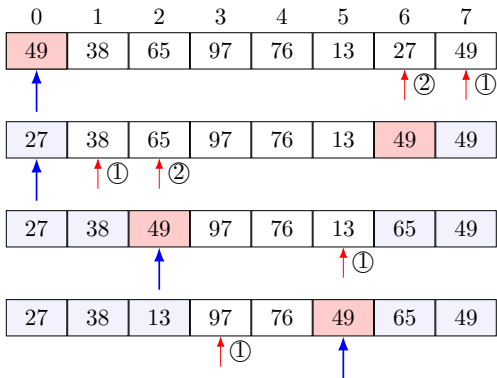
## 快速排序举例 (单次划分过程)



其中：带圆圈的数字表示比较的次序；蓝色箭头表示枢轴元素的位置；红色箭头表示要比较的元素位置。

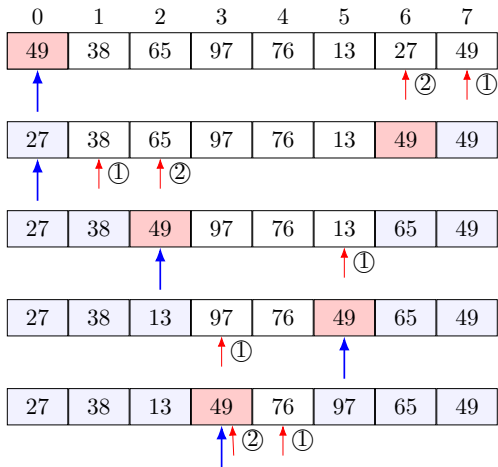


## 快速排序举例 (单次划分过程)



其中：带圆圈的数字表示比较的次序；蓝色箭头表示枢轴元素的位置；红色箭头表示要比较的元素位置。

## 快速排序举例 (单次划分过程)



其中：带圆圈的数字表示比较的次序；蓝色箭头表示枢轴元素的位置；红色箭头表示要比较的元素位置。



# 快速排序算法分析

- 空间效率

- \* 快速排序是递归的, 递归调用层次数与上述二叉树的深度一致。因而存储开销在理想情况下为  $O(\log_2 n)$ , 即树的高度; 最坏情况下为  $O(n)$ , 即二叉树是一个单链

- 时间效率

- \* 对于  $n$  个记录的待排序列, 一次划分需要  $n - 1$  次比较, 时效为  $O(n)$ , 若设  $T(n)$  为对  $n$  个记录的待排序列进行快速排序所需时间
- \* 理想情况下: 每次划分正好将分成两个等长的子序列

$$T(n) \leq cn + 2T(n/2) \cdots = O(n \log_2 n)$$

- 最坏情况下: 每次划分只得到一个子序列, 时效为  $O(n^2)$



# 名副其实的快速排序

- 快速排序的平均时间复杂度为  $O(n\log_2 n)$ 。并且在该数量级的排序方法中, 快速排序的平均性能最好。快速排序目前被认为是最好的内部排序方法。
- 若初始序列按关键字基本有序, 快速排序蜕化为起泡排序, 其时间复杂度为  $O(n^2)$
- 属于不稳定的排序算法

- 对给定的 2 个待排序列进行快速排序, 显示其执行时间, 注意 pivotkey 选子序列的第一个元素;
- 选子序列的首、尾、中间元素的中值作为 pivotkey, 重新进行快速排序, 观察执行时间的变化。
- 例如以下关键字序列, 选择不同枢轴值有无差异

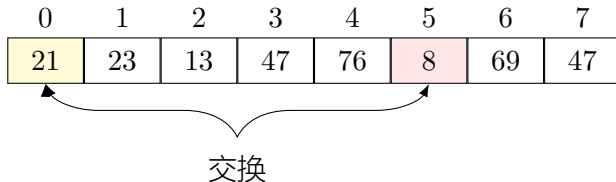
0	1	2	3	4	5	6	7
47	38	47	9	78	13	27	66

## 选择排序

- 简单选择排序
- 堆排序

## 算法思想

1. 第 1 轮排序从  $1 \sim n$  个数中找出最小的数, 然后将它与第 1 个数交换。第 1 个数则是最小的数。
2. 第 2 轮排序从  $2 \sim n$  个数中找出最小的数, 然后将它与第 2 个数交换。第 2 个数则是次小的数。
3. 经过  $n - 1$  轮处理, 完成全部  $n$  个数排序。





# 简单选择排序算法分析

- 空间复杂度:

- \* 一个辅助空间,  $O(1)$

- 时间复杂度

- \* 最好情况下, 序列为正序

第  $i$  趟需做  $n - i$  次比较, 故比较次数如下, 移动次数为 0

$$\sum_{i=1}^{n-1} (n - i) = \frac{n \times (n - 1)}{2} = O(n^2)$$

- \* 最坏情况下, 序列为反序

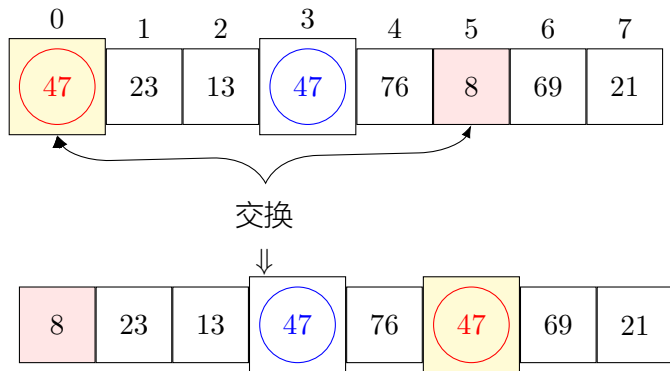
比较次数同上, 每趟排序均要执行交换操作, 移动次数为  $3(n - 1)$

- 是不是稳定的排序算法? 举例说明





# 简单选择排序不稳定



两个 47 的位置发生了交换

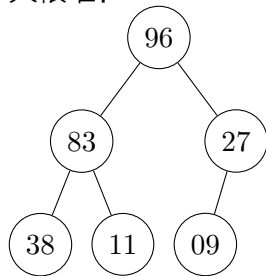
∴ 不稳定。



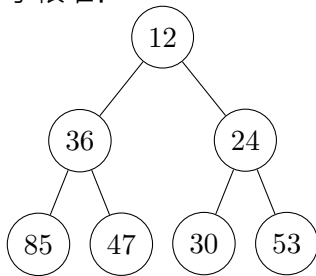
# 堆排序 (heap sort)

- 堆可以看成一棵完全二叉树, 且所有非叶结点的值均不大于 (不小于) 其左、右孩子结点的值。那么堆顶元素是最小值 (最大值)。
- 如何从一个无序序列建成一个堆?
- 在输出堆顶元素之后, 如何调整剩余部分成为一个新的堆?

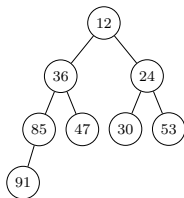
大根堆:



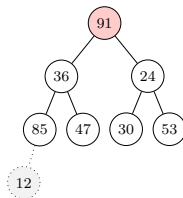
小根堆:



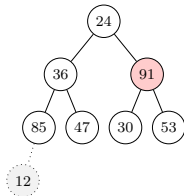
# 堆的调整举例



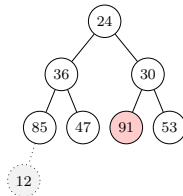
a. 输出堆顶元素 12



b. 以最后的 91 代替



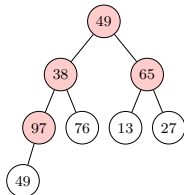
c. 选择 36, 24 中的最小值与 91 交换



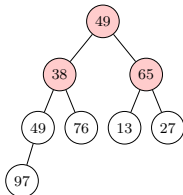
d. 选择 30, 53 中的最小值与 91 交换



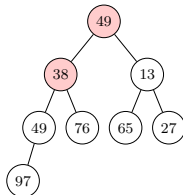
# 堆的建立: [49, 38, 65, 97, 76, 13, 27, 49]



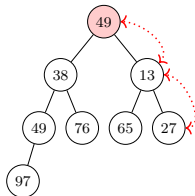
a. 初始序列



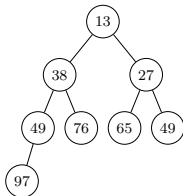
b. 97 被筛选



c. 65 被筛选



d. 49 被筛选



e. 49 再次被筛选



# 堆排序算法分析

- 空间复杂度: 一个辅助空间
- 时间复杂度为  $O(n \cdot \log_2 n)$ 
  - \*  $n$  较小时不提倡使用堆排序, 但  $n$  较大时还是很有效的。因为其运行时间主要耗费在建初始堆和调整上。
- 不稳定的.

## 归并排序

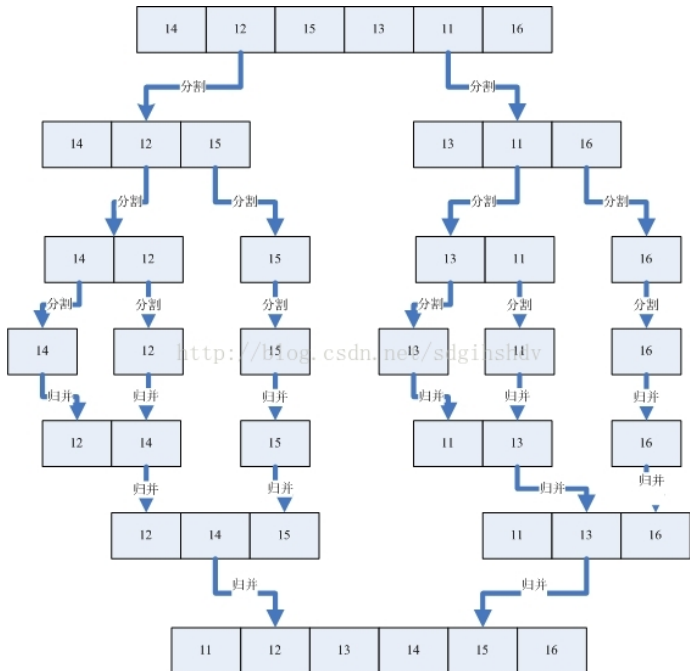
- 二路归并排序



## 算法思想

将若干个已排好序的部分合并成一个新的有序部分称为归并. 归并排序包括两个步骤:

- 划分子表
- 合并子表







# 归并排序算法分析

- 空间复杂度:  $O(n)$
- 在第  $i$  趟归并后, 有序子文件长度为  $2^i$ , 因此, 因此, 对于具有  $n$  个记录的序列来说, 需要  $\lceil \log_2 n \rceil$  趟归并。每趟归并的时间复杂度为  $O(n)$ 。因此, 归并算法的时间复杂度为  $O(n \cdot \log_2 n)$
- 稳定的排序方法



# 归并排序算法分析

- 对于内存排序来说, 归并排序的性能不如快速排序那么好, 而且它的编程一点也不简单。但归并排序是外部排序的中心思想。
- 在有些情况下, 数据集太大而不能完全存储在内存中被处理, 此时硬盘读取时间远大于内存处理时间。硬盘读取成为制约效率的关键!

## 基数排序

- 计数排序
- 基数排序



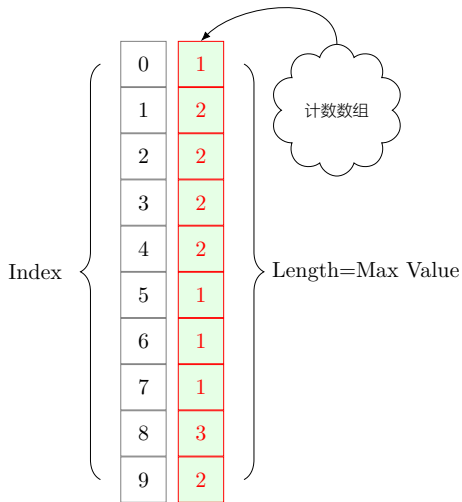
## 算法思想

构建一个足够大的数组，数组大小需要保证能够把所有元素都包含在这个数组上。

在对序列  $T$  进行排序时，依次读取序列  $T$  中的元素，并修改数组中该元素值对应位置上的信息即可。

一位数的序列对基数排序来说就是一个计数排序。

示例: [5, 8, 9, 1, 4, 2, 9, 3, 7, 1, 8, 6, 2, 3, 4, 0, 8]





## 算法思想

不需要直接对元素进行相互比较，也不需要将元素相互交换，而是对元素进行“分类”。

如果对效率有所要求，而不太关心空间的使用时，可以选择用计数排序。

# Reference



- 各种排序算法的在线动画演示:  
<https://www.cs.usfca.edu/~galles/visualization/ComparisonSort.html>  
<http://www.atool.org/sort.php>
- 搜索排序漫画系列  
<https://www.itcodemonkey.com/article/11276.html>



- 编程实现插入排序、希尔排序、冒泡排序、快速排序、选择排序、堆排序和二路归并排序。
- 能够对相同的一组输入，在一个地方（例如主函数里面）里面分别调用不同的方法，并记录每种排序算法各自的运行时间。
- 测试序列：27, 38, 1, 10, 27, 38, 3, 6, 9, 11, 18, 27, 9, 5, 100000, 1000001, 0, 10002220011, 11, 56, 79, 78, 5, 12, 8, 13, 9, 12, 10, 12