数据结构与算法B

11-排序及基本排序算法

内容提要

- 排序的基本概念
- 排序算法介绍
- 各种排序策略的比较

排序的基本概念

- 记录(Record)：进行排序的基本单位
- 关键码(Key)：唯一确定记录的一个或多个域
- 排序码(Sort Key)：记录中作为排序运算依据的一个或多个域
- 序列(Sequence)：线性表，由记录组成的集合
- 排序(Sorting) — 将序列中的记录按照排序码特定的顺序排列起来，即排序码域的值具有不减(或不增)的顺序

排序的基本概念

- 给定一个序列R =$\{r_1, r_2, \ldots, r_n\}$，其排序码分别为k =$\{k_1, k_2, \ldots, k_n\}$，排序的目的就是将R中的记录按照特定的顺序重新排列，形成一个新的有序序列R'= $\{r'_1, r'_2, \ldots, r'_n\}$
  - 相应排序码为k' =$\{k'_1, k'_2, \ldots, k'_n\}$
  - 其中$k'_1 \le k'_2 \le \ldots \le k'_n$或$k'_1 \ge k'_2 \ge \ldots \ge k'_n$，前者称不减序（也称非递减序），后者称不增序（也称非递增序）

排序的基本概念

- 正序与逆序
  - “正序”序列 ：待排序序列**正好符合排序要求**
  - “逆序” 序列 ：把待排序序列逆转过来，正好符合排序要求

  - 譬如，需要得到一个不减序序列？

- **正序：**

11	19	23	55	80	97

- **逆序：**

97	80	55	23	19	11

排序的基本概念

- 排序的稳定性：

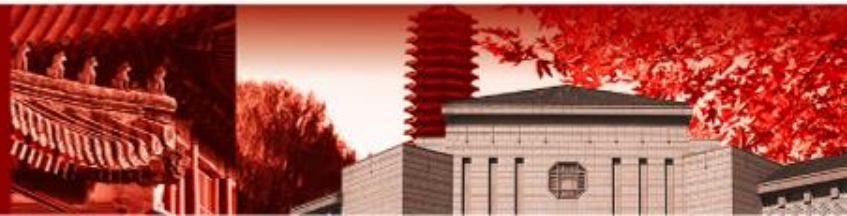
  在待排序的文件中，若存在多个排序码相同的记录，经过排序后记录的相对次序保持不变，则这种排序方法称为是“稳定的”；否则，是“不稳定的”

譬如，对下列数据进行不减序排序

23	19	55	97	19’	80

正确但不稳定的排序结果

19’	19	23	55	80	97



排序的种类

- **按排序方法**
  - 插入排序
  - 选择排序
  - 交换排序
  - 分配排序
  - 归并排序

- **按排序中涉及的存储器**
  - **内排序**：待排序的记录在排序过程中全部存放在内存
  - **外排序**：如果排序过程中需要使用外存的

本章讨论的都是内排序的方法，
虽然有些方法(归并排序)也可以用于外排序

排序算法的评价

- 算法的时间复杂性
  - 一般按最坏情况或平均情况估算（在应用时要根据情况计算实际的开销，选择合适的算法）
  - 排序的两个基本操作：比较和交换

23	11	55	97	19	80

  - 比较两个排序码（如23和11）的大小
  - 交换两个记录（如23和11）的位置

- 算法的空间复杂性
  - 执行排序算法所需的附加空间一般不大，一般只给出结论

排序算法介绍

插入排序	1 直接插入排序 2 二分法插入排序 3 Shell排序
选择排序	4 直接选择排序 5 堆排序
交换排序	6 冒泡排序 7 快速排序
分配排序	8 基数排序
归并排序	9 二路归并排序

排序算法介绍

插入排序	1 直接插入排序 2 二分法插入排序 3 Shell排序
选择排序	4 直接选择排序 5 堆排序
交换排序	6 冒泡排序 7 快速排序
分配排序	8 基数排序
归并排序	9 二路归并排序

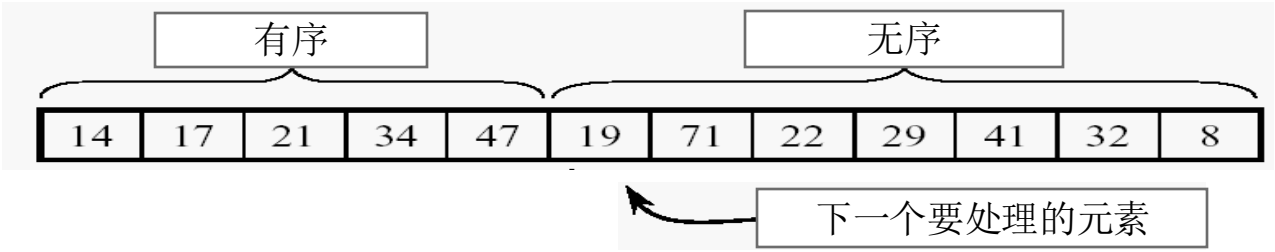
插入排序

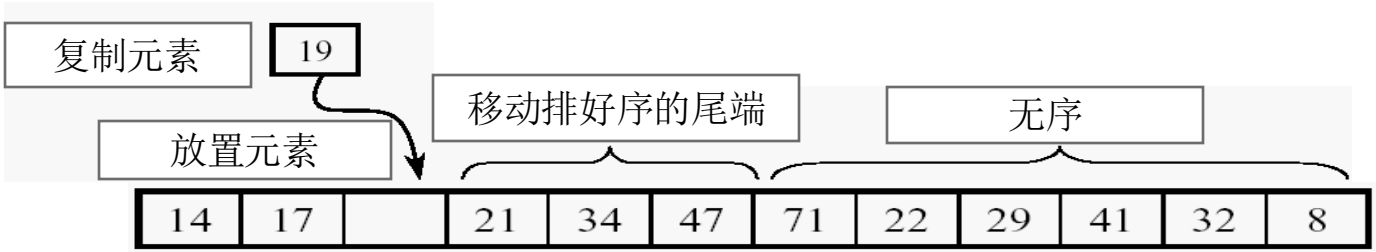
每一步将一个待排序的记录，按其排序码大小插到前面已经排序文件中的适当位置，直到全部插入完为止。

- **直接插入排序**
- 二分法插入排序
- shell排序

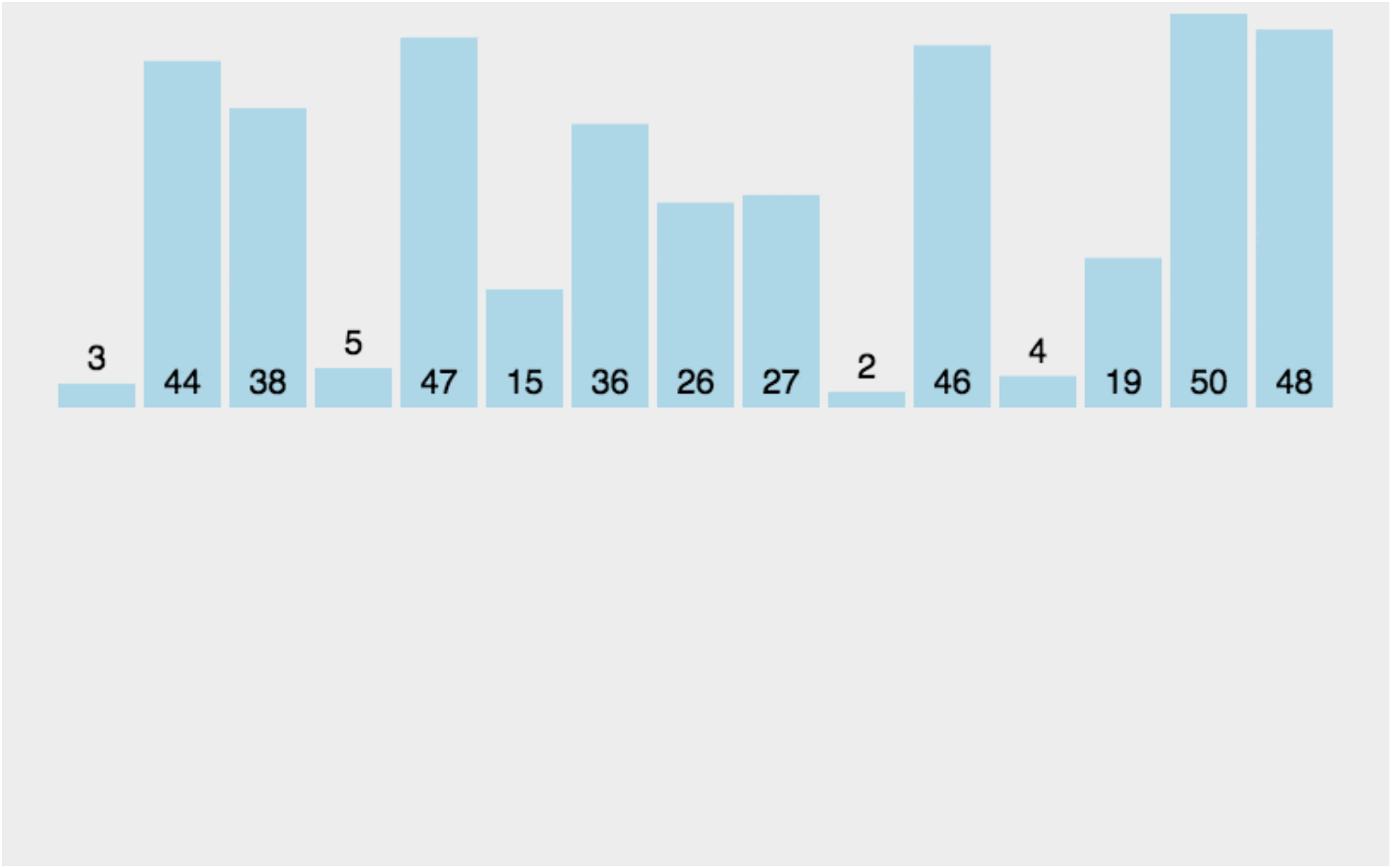
直接插入排序

- 基本思想：
  - 将待排序序列分为**已排序**和**未排序**两部分。初始时，已排序部分仅包含第一个元素，未排序部分包含剩余元素。
  - 逐个处理待排序的记录。每步将一个待排序的元素R_i按其排序码K_i大小插入到前面已排序表中的适当位置，直到全部插入完为止。





直接插入排序—示例



直接插入排序—代码

新项/插入项

比对、移动

插入新项

```
def insertion_sort(alist): /* 按不减序进行直接插入排序
    for index in range(1, len(alist)):
        current_value = alist[index]
        position = index
        while position > 0 and alist[position - 1] > current_value:
            alist[position] = alist[position - 1]
            position = position - 1
        if position != index:
            alist[position] = current_value
```

直接插入排序算法评价

- 算法是稳定的
- 空间代价：O(1)，交换操作需要一个辅助空间
- 时间代价
  - 最佳情况（正序）：n-1次比较，0次交换，O(n)复杂度
  - 最差情况（逆序）：比较和交换次数为$O(n^2)$

$$\sum_{i=1}^{n-1} i = n(n-1)/2$$

  - 平均情况：$O(n^2)$

实验表明：当记录数量n较小时，直接插入排序是一种高效的排序算法！

插入排序

每一步将一个待排序的记录，按其排序码大小插到前面已经排序文件中的适当位置，直到全部插入完为止。

- 直接插入排序
- **二分法插入排序**
- shell排序

二分法插入排序

- 基本思想：

  在直接插入排序的基础上减少比较的次数，即在插入R_i时改用二分法找插入位置。

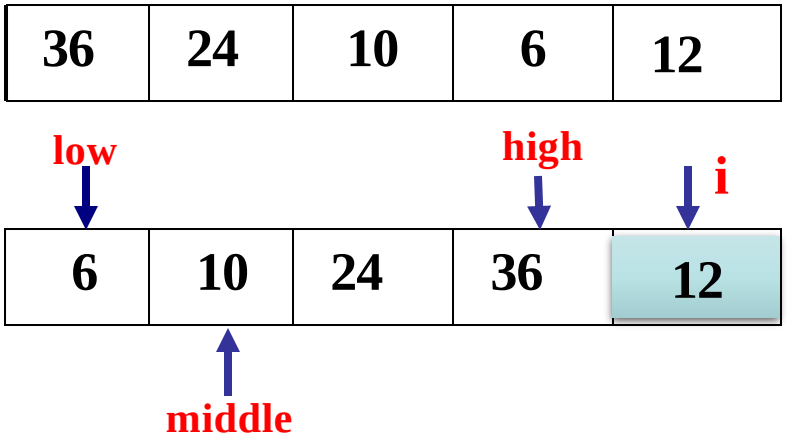
  –插入记录R_i时，记录集合中子区间$\{R_0,R_1,\ldots,R_{i-1}\}$已经有序

  –low =0; high = i-1; mid= (low+high)/2 带入二分检索

  - 找出应该插入的位置；
  - 将原位置的记录向后顺移，将记录R_i插入。

- 二分插入排序采用顺序存储结构

二分法插入排序——示例



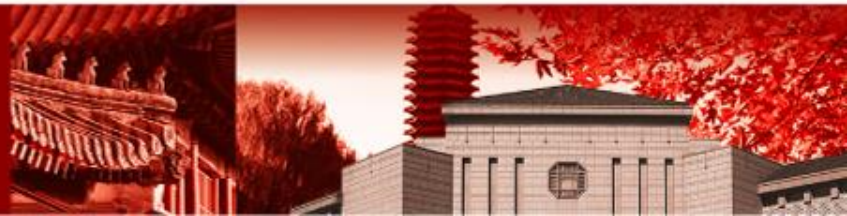
1）移动条件：<middle，high左移（middle-1）；
≥middle，low右移（middle+1）。

2）结束条件：low>high;

3）插入位置：low

二分法插入排序算法—代码

```
def binary_insertion_sort(alist):
    for i in range(1, len(alist)):
        key = alist[i]    #当前需要插入的元素
        low = 0           #二分查找的左边界
        high = i - 1      #二分查找的右边界
        while low <= high:
            middle = (low + high) // 2
            if alist[middle] < key:
                low = middle + 1    # 在右半部分继续查找
            else:
                high = middle - 1   # 在左半部分继续查找
        j = i - 1
        while j >= low:
            alist[j + 1] = alist[j]  # 向右移动元素，为插入腾出空间
            j -= 1
        alist[low] = key    # 插入当前元素到正确位置
    return alist
```

二分插入排序分析

- 算法是稳定的
- 空间代价：O(1)，算法中有一个辅助空间
- 时间代价
  - 比较次数降为nlogn量级：插入每个记录需O(log *i*)次比较
  - 移动次数仍为n^2量级：每插入一个记录最多移动i+1次
  - 因此，最佳情况下总时间代价为O(nlog *n*) ，最差和平均情况下仍为$O(n^2)$

二分插入排序比较次数

- 二分插入排序的比较次数与待排序记录的初始状态无关，仅依赖于记录的个数，插入第 i 个记录时，
  - 如果$i = 2^j (0 \le j \le \lfloor \log_2 n \rfloor)$，则无论排序码的大小，都恰好经过j次比较才能确定插入位置；
  - 如果 $2^j < i \le 2^{j+1}$，则比较次数为j+1。
  - 因此，将$n\ (n = 2^k)$个记录排序的总比较次数为

$$\sum_{i=1}^{n} \lceil \log_2 i \rceil = 0 + 1 + 2 + 2 + \ldots + k + k + \ldots + k$$

$$= n \log_2 n - n + 1 \approx n \log_2 n$$

插入排序

每一步将一个待排序的记录，按其排序码大小插到前面已经排序文件中的适当位置，直到全部插入完为止。

- 直接插入排序
- 二分法插入排序
- Shell排序

Shell排序

- Shell排序的提出基于直接插入排序的2个性质
  - –在待排序序列较短情形下效率高
  - –在整体有序的情形下时间代价低
- Shell排序又称为“缩小增量排序”
  - –1959年由D.L.Shell提出

Shell排序

如何利用这两个性质？

1. 在初始无序时，进行等间隔的小序列分割

- 先将待排序序列转化为若干小序列，在这些小序列内进行直接插入排序

2. 在整个序列趋向有序后，逐步扩大序列规模

- 逐渐扩大小序列的规模，而减少小序列个数，使得待排序序列逐渐处于更有序的状态

3. 最后，对整个序列进行一次完整的插入排序

Shell排序实现方法

1) 选定一个间隔增量序列（$n>d_1>d_2>\ldots>d_t=1$）

   - n：文件长度，d_i：间隔增量，t：排序趟数

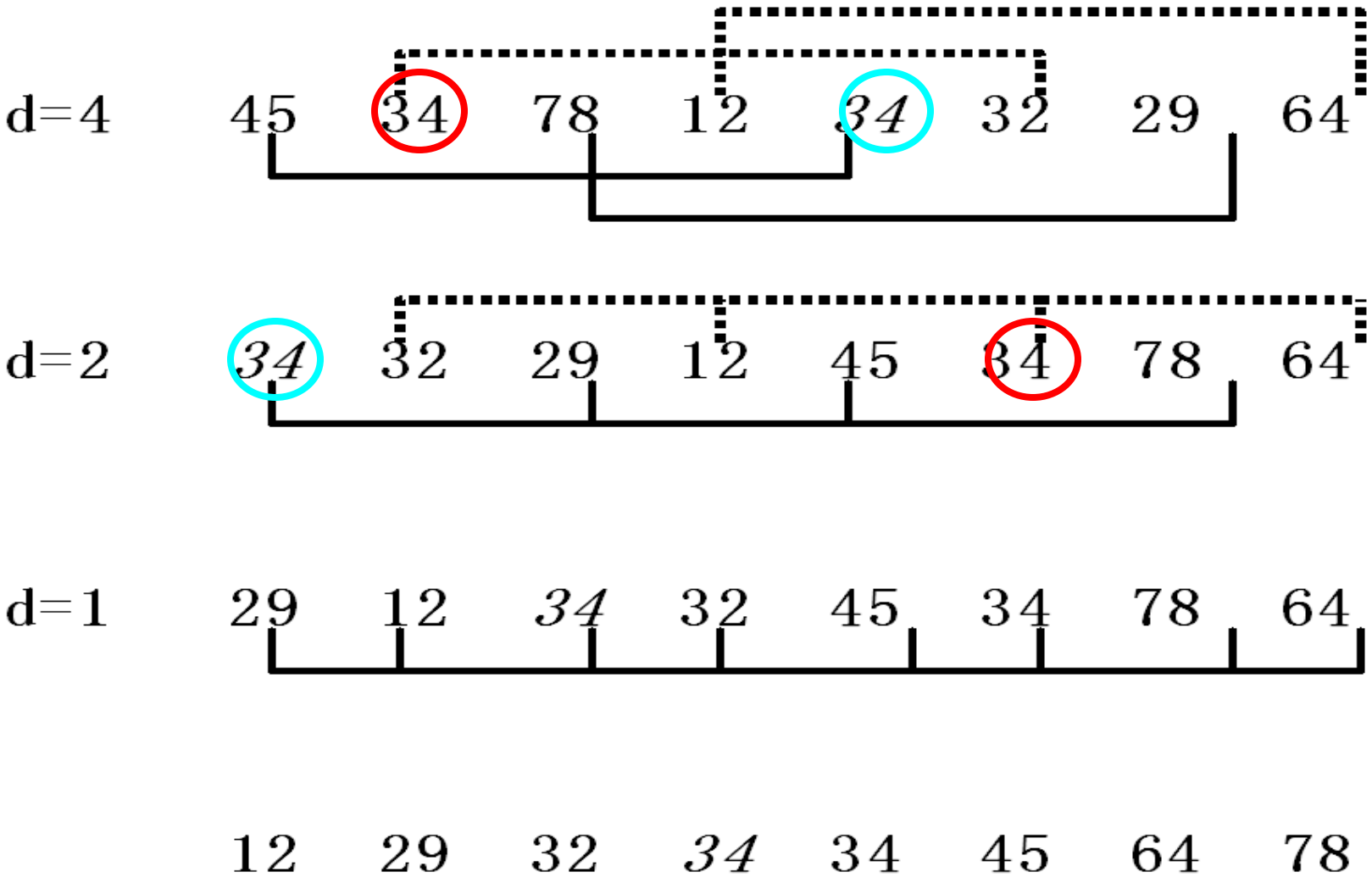
2) 将文件按 d_1 分组（彼此相距 d_1 的记录划为一组），在各组内采用直接插入法进行排序。

3) 分别按 $d_2, \ldots d_t$ 重复上述分组和排序工作。

- **Shell 最初提出的增量序列是**

$$d_1 = \lfloor n/2 \rfloor,\ d_{i+1} = \lfloor d_i/2 \rfloor$$

Shell排序过程


d=4
45 34 78 12 34 32 29 64
d=2
34 32 29 12 45 34 78 64
d=1
29 12 34 32 45 34 78 64
12 29 32 34 34 45 64 78

Shell排序算法

间隔设定

子列表排序

间隔缩小

```
def shell_sort(alist):
    sublist_count = len(alist) // 2
    while sublist_count > 0:
        for start_position in range(sublist_count):
            gap_insertion_sort(alist, start_position, sublist_count)
        print("After increments of size", sublist_count, "The list is", alist)
        sublist_count = sublist_count // 2


def gap_insertion_sort(alist, start, gap):
    for i in range(start + gap, len(alist), gap): # 从start+gap开始，以gap为增量遍历
        current_value = alist[i]    # 当前待插入元素
        position = i       # 当前位置
        while position >= gap and alist[position - gap] > current_value:
            alist[position] = alist[position - gap]    # 元素后移
            position = position - gap      # 向前比较
        alist[position] = current_value    # 插入元素
```

Shell排序算法性能分析

- Shell排序算法的速度比直接插入排序快，其时间复杂度分析比较复杂，Shell排序的平均比较次数和平均移动次数都为$n^{1.3}$左右

- Shell排序算法中增加了一个辅助空间，因此算法的辅助空间为S(n)=O(1)

- Shell排序是不稳定的

插入排序小结

- 直接插入排序思想最简单，容易实现，但时间效率低。
- 其它插入排序方法都是从减少比较次数、移动次数出发对直接插入排序进行改进。
  - –直接：顺序检索确定位置，记录移动实现插入；
  - –折半：减少比较次数（折半检索确定位置）；
  - –Shell排序：改变增量（来自：当待排序序列基本有序，并且n较小时，提高了直接插入排序效率）
- 除Shell外，其它插入排序算法的时间复杂度为$O(n^2)$，并且是稳定的。

排序算法介绍

插入排序	1 直接插入排序 2 二分法插入排序 3 Shell排序
选择排序	4 直接选择排序 5 堆排序
交换排序	6 冒泡排序 7 快速排序
分配排序	8 基数排序
归并排序	9 二路归并排序

选择排序

每趟从待排序的记录序列中选择关键字最小/大的记录放置到已排序表的最前位置，直到全部排完。

1. 直接选择排序
2. 堆排序

直接选择排序

- 基本思想
  - 每一趟在后面n-i个待排记录中选取最小记录和第i个记录互换。
- 具体过程
  - 首先，在n个记录中选择最小者与r[0]互换；
  - 然后，从剩余的n-1个记录中选择最小者与r[1]互换；
  - …如此下去，直到全部有序为止。
- 优点：实现简单
- 缺点：每趟只能确定一个元素，表长为n时需要n-1趟

直接选择排序算法

找出未排序最小元素的索引

把找到的最小元素与第一个未排序元素交换

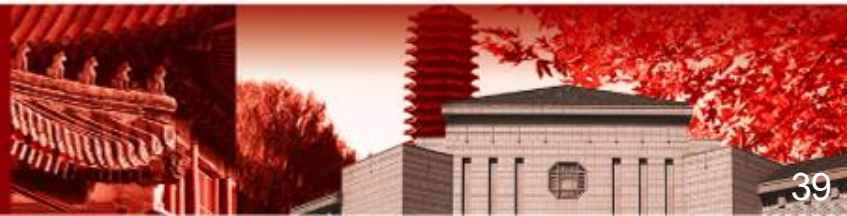
```
def selection_sort(alist):
    for i in range(len(alist)):
        min_index = i
        for j in range(i + 1, len(alist)):
            if alist[j] < alist[min_index]:
                min_index = j
        if min_index != i:
            alist[i], alist[min_index] = alist[min_index], alist[i]
```

直接选择排序——示例

例：关键字序列T= （21，25，49，25*，16，08），
请给出直接选择排序的具体实现过程。

原始序列：	21，25，49，25*，16，08	
第1趟	08，25，49，25*，16，21	最小值 08 与r[0]交换位置
第2趟	08，16，49，25*，25，21	不稳定操作发生
第3趟	08，16，21，25*，25，49	
第4趟	08，16，21，25*，25，49	
第5趟	08，16，21，25*，25，49	

北京大学



直接选择排序性能分析

- 直接选择排序的比较次数与记录的初始状态无关。
  - 第i趟排序：从第i个记录开始，顺序比较选择最小关键码记录需要n-i次比较。
  - 总的比较次数：$\sum_{i=1}^{n-1}(n-i)=\frac{n(n-1)}{2}$

- 时间复杂度：$T(n)=O(n^2)$，
- 辅助空间1个记录单位： $S(n)=O(1)$
- 稳定性：不稳定的排序。

选择排序

每趟从待排序的记录序列中选择关键字最小/大的记录放置到已排序表的最前位置，直到全部排完。

1. 直接选择排序
2. 堆排序

堆排序

- 动机：

  在选择过程中如何利用前面的比较结果来减少比较次数？

- 发明：

  由计算机先驱奖获得者、斯坦福大学计算机科学系教授Robert W．Floyd 和J．Williams在1964年共同发明

堆的定义

1、堆的定义

当n个排序码序列$K=\{k_0,k_1,k_2,\ldots,k_{n-1}\}$，当且仅当满足如下条件时，称之为堆

$k_i \le k_{2i+1}$ 且 $k_i \le k_{2i+2}$ 或 $k_i \ge k_{2i+1}$ 且 $k_i \ge k_{2i+2}$ $(i=0,1,2,\ldots,\lfloor n/2 \rfloor -1)$

如果堆中根结点的排序码最小，则称为最小堆

如果堆中根结点的排序码最大，则称为最大堆

2、堆的完全二叉树表示

堆可以用一棵完全二叉树表示，则 K_{2i+1} 是 K_i 的左孩子；K_{2i+2} 是 K_i 的右孩子。

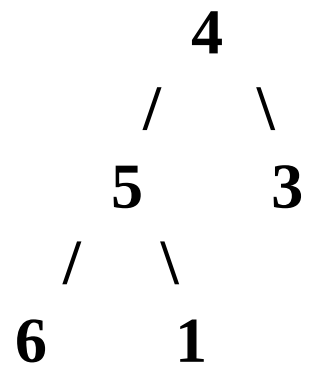
堆排序的基本思想

1. 对所有记录建立最大堆（O(n)）
2. 取出堆顶的最大记录移到数组末端，放在下标n-1的位置；重新将剩下的n-1个记录建堆（O(logn)），再取新堆顶最大的记录，放到数组第n-2位；…；不断重复这一操作，直到堆为空。
3. 这时数组正好是按从小到大排序

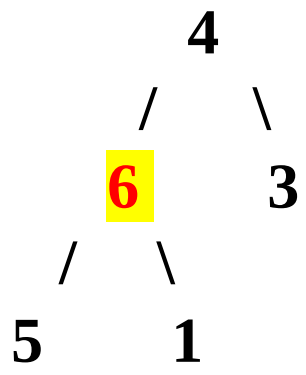
堆排序 —示例

初始序列： 4，5，3，6，1

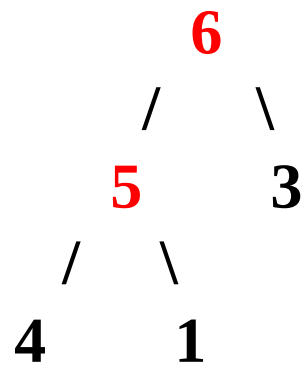
(1)初始建堆



[1] 初始完全二叉树



[2] 以序号⌊n/2⌋-1为根的子树建堆

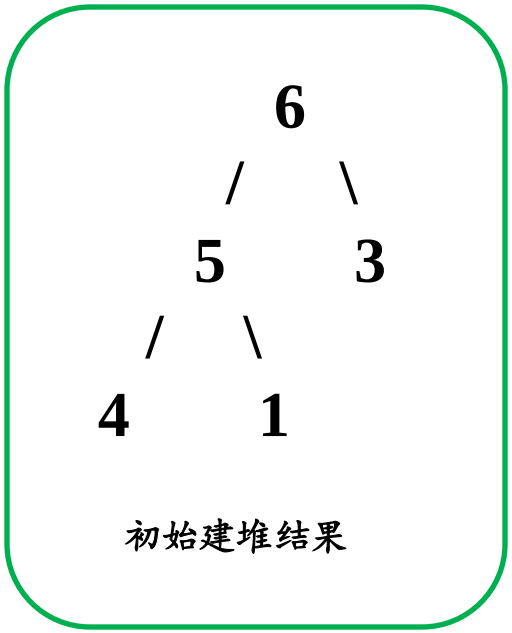


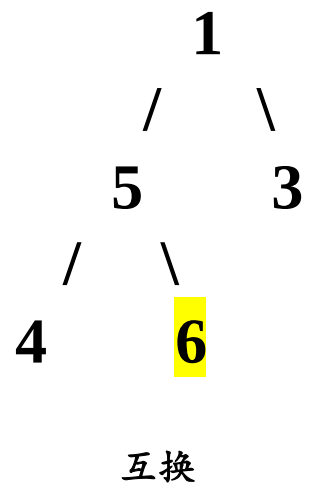
[2]以序号0 为根的子树建堆

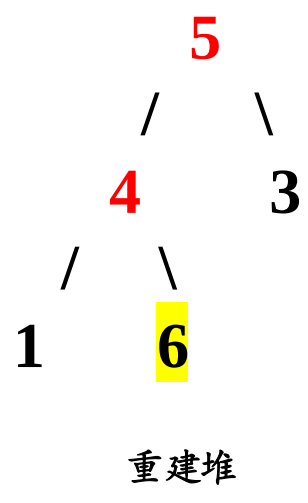
堆排序 —示例

初始序列： 4，5，3，6，1

(2)将序号为0的元素与序号为n-1的元素互换，重建堆



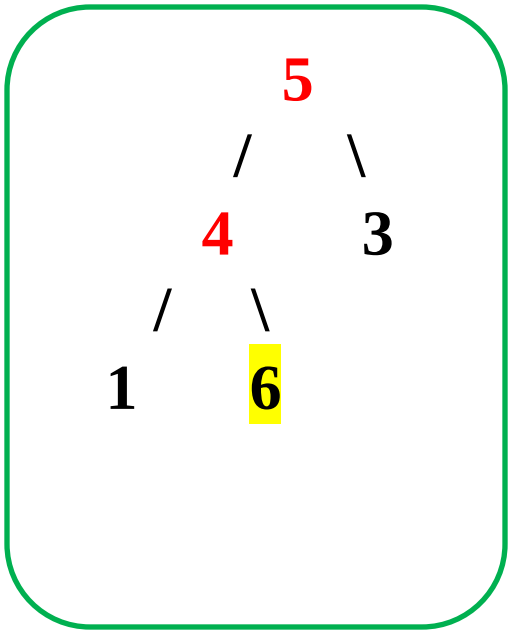


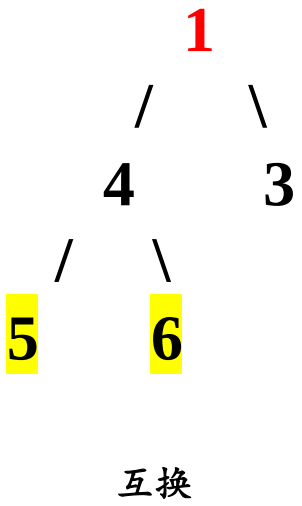


堆排序 —示例

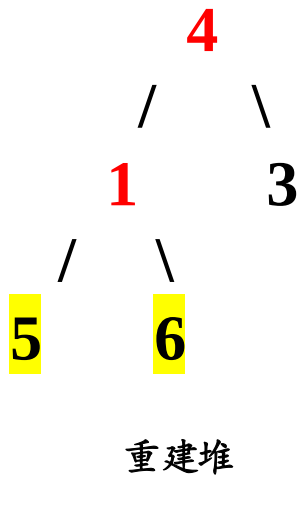
初始序列： 4，5，3，6，1

(2)将序号为0的元素与序号为n-2的元素互换，重建堆





互换

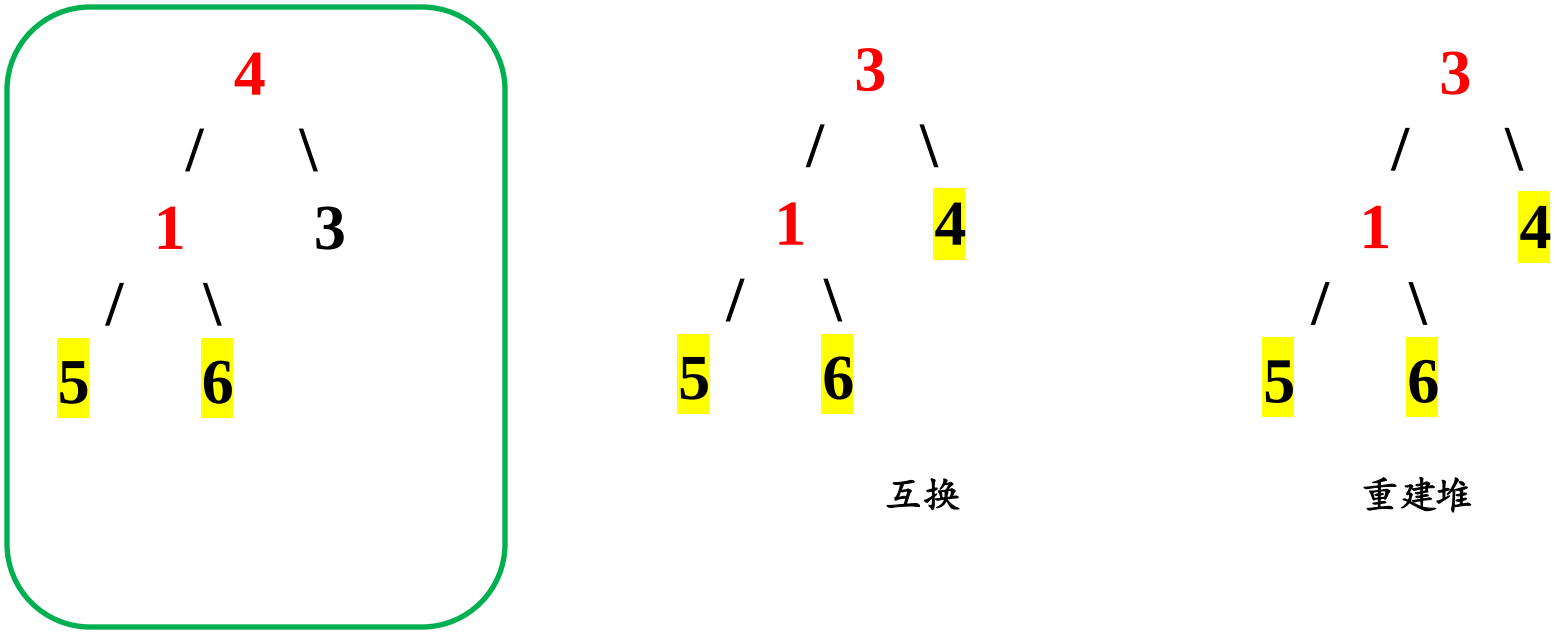


重建堆

堆排序 —示例

初始序列： 4，5，3，6，1

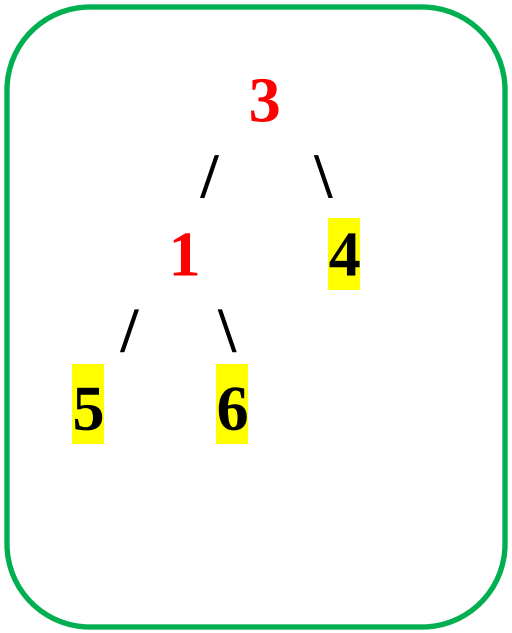
(2)将序号为0的元素与序号为n-3的元素互换，重建堆

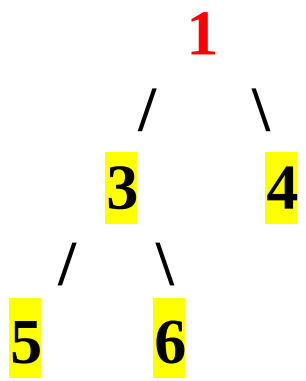


堆排序 —示例

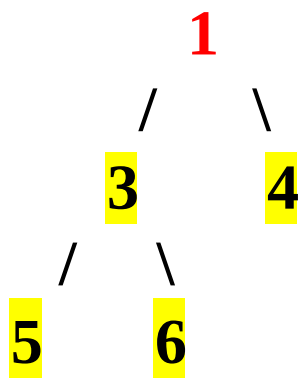
初始序列： 4，5，3，6，1

将序号为0的元素与序号为n-4的元素互换，重建堆





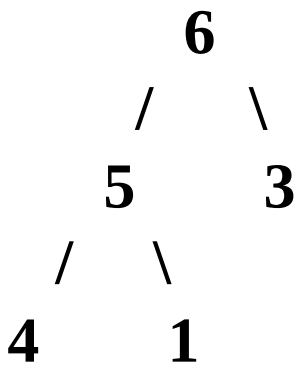
互换 【排序完成】



重建堆【排序完成】

关键问题-1

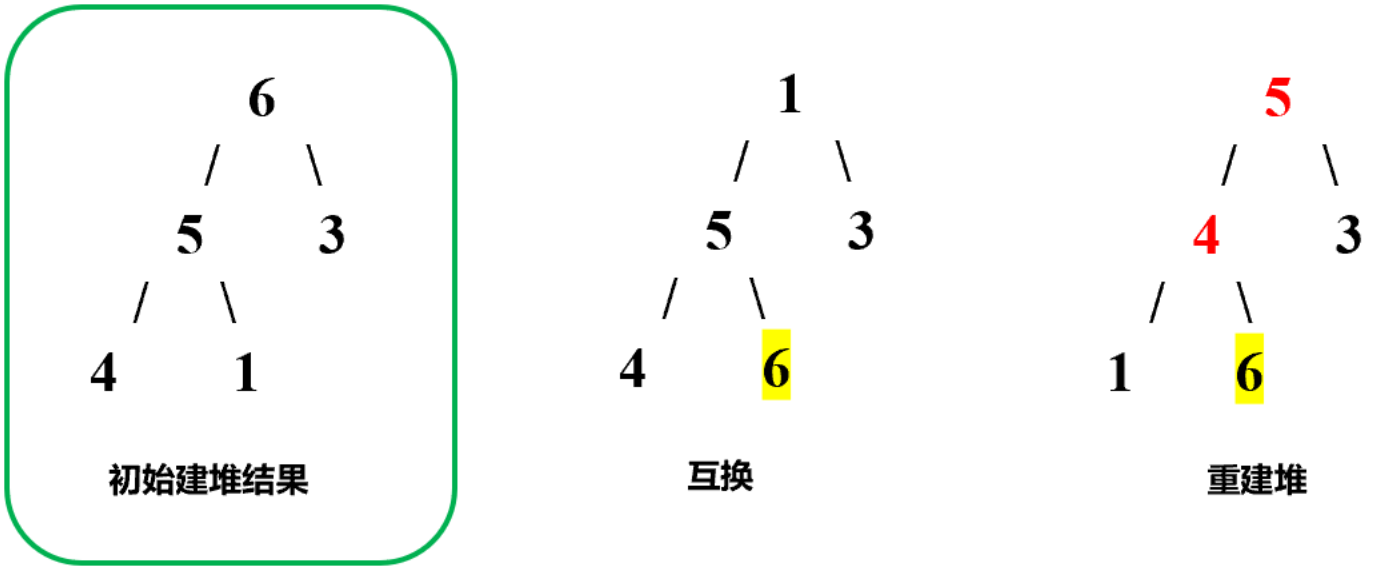
- 如何将原始序列构成初始堆？



- 初始完全二叉树中，序号为⌊n/2⌋，⌊n/2⌋+1 …，n-1的结点为叶子，以其为根的子树必然为堆。
- 因此，初始堆建立时，只需要将所有非终端结点为根的子树调整为堆。

关键问题-2

- 初始建堆以后，把序列的第1个元素与第n个元素交换，丢掉最大值后如何构造新堆？



关键问题-2

- 采用“筛选法”建堆
  - 为以R_i 为根的完全二叉树建堆，这时R_i 的左、右子树都是堆，可以把R_i 与其左、右子树根结点R_{2i+1}、R_{2i+2}中最大者交换位置。
  - 若交换位置后破坏了子树的堆特性，则再对这棵子树重复交换过程，直到以R_i 为根结点的子树成为堆。

堆排序算法

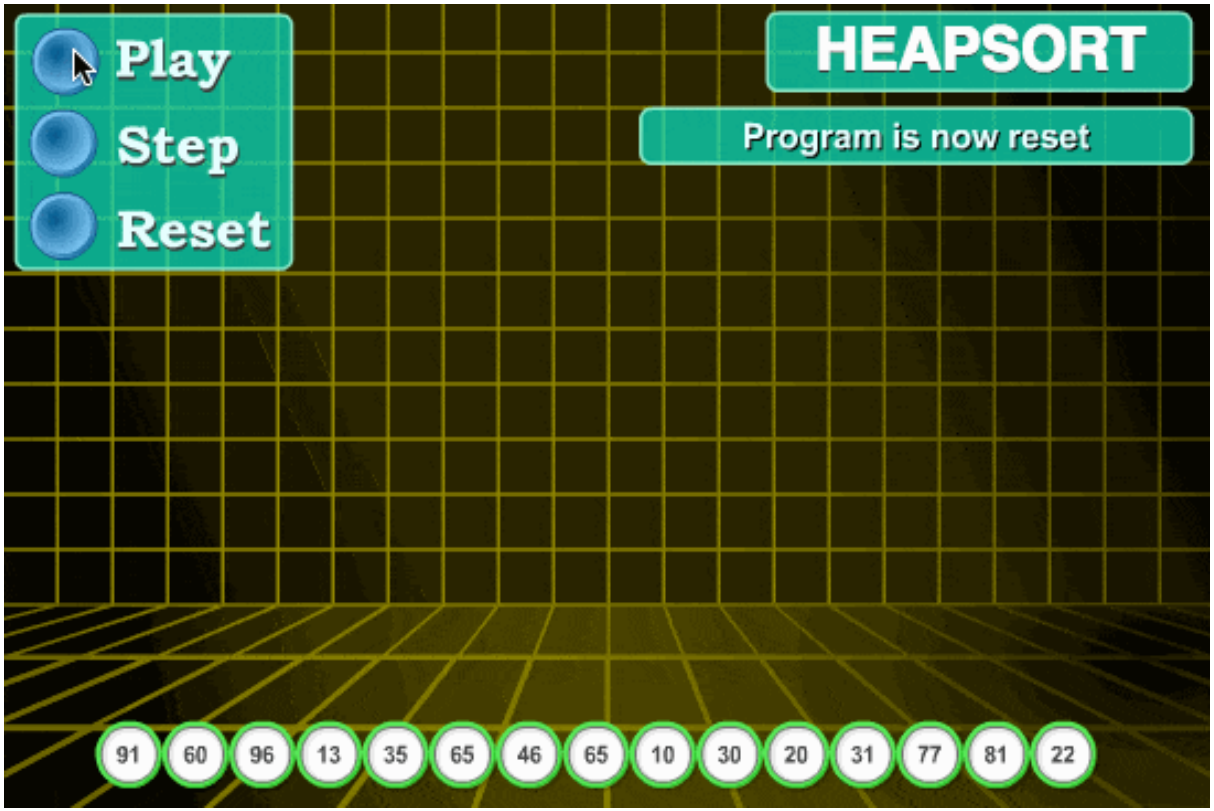
```
def heap_sort(alist):
    def sift_down(start, end):
        root = start          # 当前子树的根节点
        while True:
            child = 2 * root + 1     # 左子节点位置
            if child > end:          # 如果没有子节点
                break
            if child + 1 <= end and alist[child] < alist[child + 1]:
                child += 1     # 如果有右子节点且右子节点更大，选择较大的子节点
            if alist[root] < alist[child]:
                alist[root], alist[child] = alist[child], alist[root]
                root = child   # 继续向下检查
            else:
                break     # 堆性质已满足，退出

    for start in range((len(alist) - 2) // 2, -1, -1):    #从最后一个非叶子节点开始
        sift_down(start, len(alist) - 1)
    for end in range(len(alist) - 1, 0, -1):
        alist[0], alist[end] = alist[end], alist[0]    # 将最大值(堆顶)移到末尾
        sift_down(0, end - 1)     # 对剩余元素重新构建最大堆
```

堆内调整

初始建堆

构造新堆

堆排序算法—动画演示


Play
Step
Reset
HEAPSORT
Program is now reset
91
60
96
13
35
65
46
65
10
30
20
31
77
81
22

时间效率评价

- 分析
  - 建初始堆比较次数C_1：O(n)
  - 重新建堆比较次数C_2：O(nlog_2n)
  - 总比较次数=C_1+C_2
  - 移动次数小于比较次数，因此，

- 堆排序算法的时间复杂性为：O（nlog_2n）
- 堆排序算法的空间复杂性为：O（1）
- 堆排序算法是不稳定的。
- 适用于n值较大的情况。

排序算法介绍

插入排序	1 直接插入排序 2 二分法插入排序 3 Shell排序
选择排序	4 直接选择排序 5 堆排序
交换排序	6 冒泡排序 7 快速排序
分配排序	8 基数排序
归并排序	9 二路归并排序

交换排序

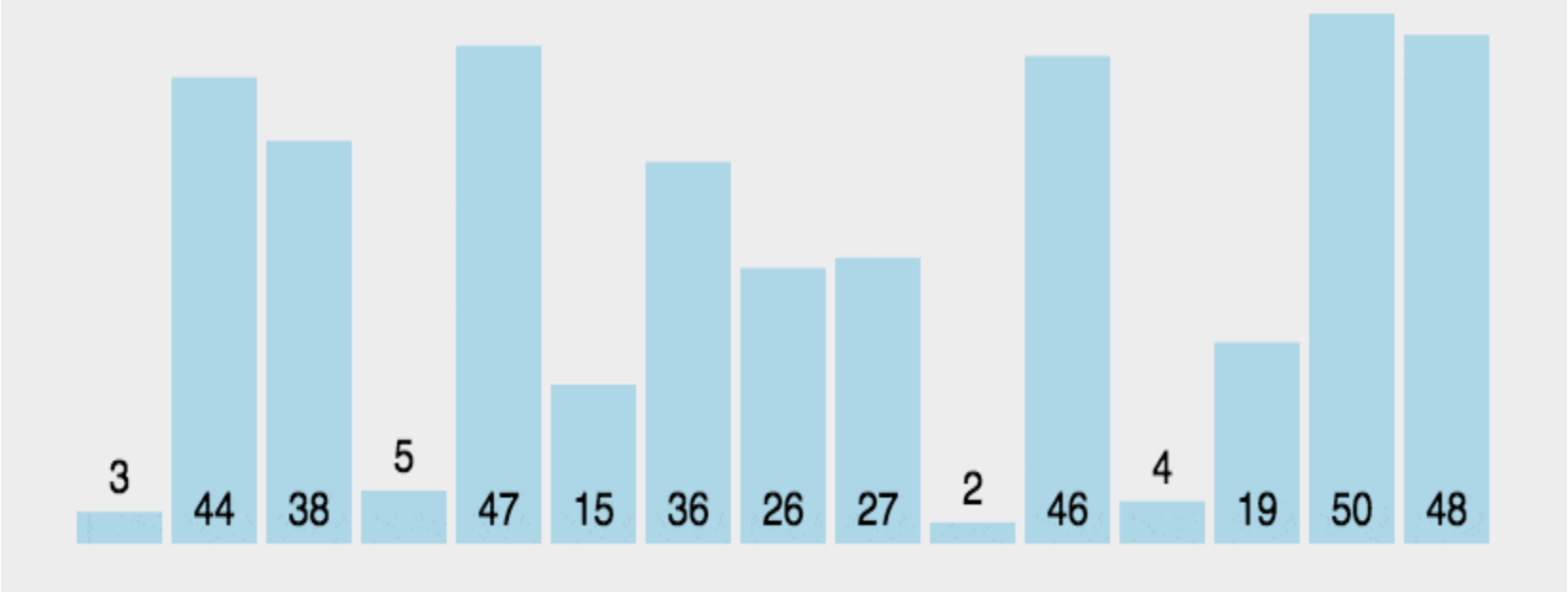
两两比较待排序记录的排序码，交换不满足顺序要求的偶对，直到全部满足为止。

1. 冒泡排序
2. 快速排序

冒泡排序

- 基本思想
  - 不停地比较相邻的记录，如果不满足排序要求，就交换相邻记录，直到所有的记录都已经排好序
  - 原理
    - 若序列中有 n 个元素，通常进行 n - 1 趟。
      - 第1趟，针对第r[0]至r[n-1]个元素进行。
      - 第2趟，针对第 r[0]至r[n-2] 个元素进行。……
      - 第n-1趟，针对第r[0]至r[1]个元素进行。
    - 第i趟进行的过程：
      - 针对r[0]至r[n-i] 元素进行，比较两个相邻的元素。若相邻的元素的相对位置不正确，则进行交换；否则继续比较下面两个相邻的元素。

冒泡排序【向后冒泡】示意



冒泡排序算法分析

- 算法是稳定的
- 空间代价：O(1)的临时空间
- 时间代价
  - 比较次数
  - 交换次数最多为$O(n^2)$，最少为0，平均为$O(n^2)$。
  - 最大，最小，平均时间代价均为$O(n^2)$。

冒泡排序方法改进

- 改进:

  可以设置一个标志no_swap表示本次冒泡是否有记录交换，如果没有交换则表示整个排序过程完成

冒泡排序算法【向后冒泡】

```
def bubble_sort(alist):
    for i in range(len(alist) - 1, 0, -1):
        no_swap = True
        for j in range(i):
            if alist[j] > alist[j + 1]:
                alist[j], alist[j + 1] = alist[j + 1], alist[j]
                no_swap = False
        if no_swap:
            break          # 如果没有发生交换，说明数组已有序，提前结束
```

外层循环控制排序轮数，i表示每轮需要比较的最后一个元素位置

内层循环进行相邻元素比较

冒泡排序的算法评价

- 若文件初状为正序，则一趟冒泡就可完成排序，排序码的比较次数为n-1，且没有记录交换，时间复杂度是O(n)
- 若文件初态为逆序，则需要n-1趟冒泡，每趟进行n-i次排序码的比较，且每次比较都需要交换，比较和交换次数均达到最大值:

$$C_{\max} = \sum_{i=1}^{n-1}(n-i) = n(n-1)/2 = O(n^2)$$

冒泡排序的算法评价(续)

- 冒泡排序最坏时间复杂度为： $O(n^2)$
- 冒泡排序平均时间复杂度为：$O(n^2)$
- 冒泡排序算法中增加一个辅助空间，辅助空间为：$S(n)=O(1)$
- 冒泡排序算法是稳定的

选择排序

两两比较待排序记录的排序码，交换不满足顺序要求的偶对，直到全部满足为止。

1. 冒泡排序
2. 快速排序

快速排序

- 快速排序算法是20世纪十大算法之一，1962年提出

17. 计算的美丽 – 1980 年图灵奖获得者 Tony Hoare

C. Antony R. Hoare (1/11/1934–)

图灵奖获得时间：

1980 年。 第十五位图灵奖(1980 年)获得者。

图灵奖引用(Turing Award Citation) :

For his fundamental contributions to the definition and design of programming languages.

笔者注：

Hoare 对程序设计语言的主要贡献为：Hoare Logic, 快速排序算法(Quicksort)和 CSP(Communication Sequential Processes)

Charles Antony Richard Hoare(Tony Hoare和CAR Hoare)出生于 1934 年 1 月 11 日于斯里兰卡。其父母为英国人。Hoare于 1956 年从牛津大学(http://www.ox.ac.uk/)获得其学士学位。后来Hoare前往原苏联并在莫斯科州立大学学习自然语言的计算机转换。1960 年，Hoare回到英国并工作于Elliott Brothers公司。在Elliott Brothers，Hoare实现了ALGOL 60 编译器。1968 年，Hoare获得了University of Belfast(http://www.qub.ac.uk/)的教授职务。1977 年，Hoare回到牛津大学担任计算机科学程序语言研究小组的教授。除了其在牛津大学的教职，Hoare也在微软 Microsoft Inc.在英国的研究所出任研究员的职位，可参见：http://research.microsoft.com/users/thoare/

在 2000 年，Hoare 由于其在计算机科学和教育方面的杰出贡献被英国皇家授予爵士爵位。

快速排序是对冒泡排序的改进

- **冒泡排序**

  在相邻两个记录间比较和交换，每次交换只能上移或下移一个位置，导致总的比较与移动次数增多

- **如何改进？**

基于“分治”思想的快速排序

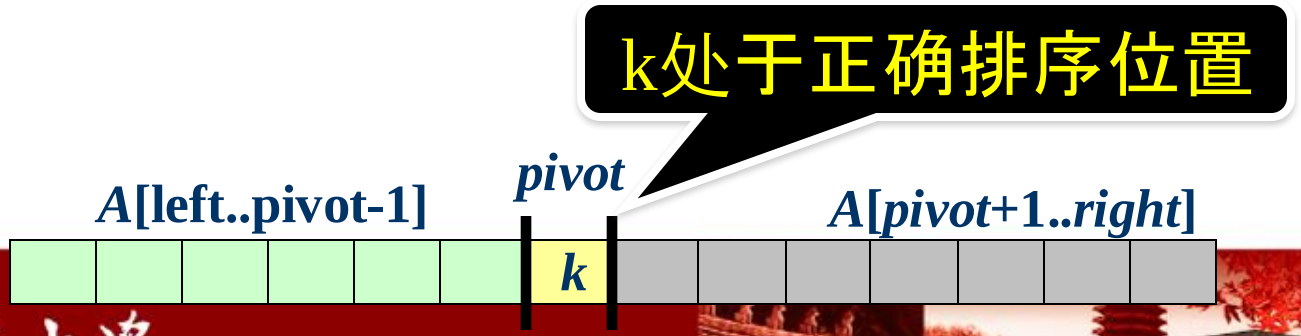
- 分治思想

- 基于分治思想的快速排序

1.轴值选择：从待排序列中选择轴值k(参考点)为划分基准;

2.序列划分：划分为子序列L和R，L中记录都“≤”k，R中记录都“>” k;

3.递归排序：对子序列进行递归划分，直到仅含1或0个元素



快速排序是对冒泡排序的改进

- **冒泡排序**

  在相邻两个记录间比较和交换，每次交换只能上移或下移一个位置，导致总的比较与移动次数增多

- **快速排序**又称**分区交换排序**

  设待排序的n个记录{$R_0, R_1, \ldots R_{n-1}$}，选取第一个记录R_0为划分基准，寻找R_0的最终位置（一趟快速排序）：

  – **{R[0], R[1], …,R[i-1]} 存放的为小于R_0的记录**

  –**{R[i+1], R[i+2],…,R[n]}存放的为大于R_0的记录**

  –**R[i] 为R_0的最终位置**

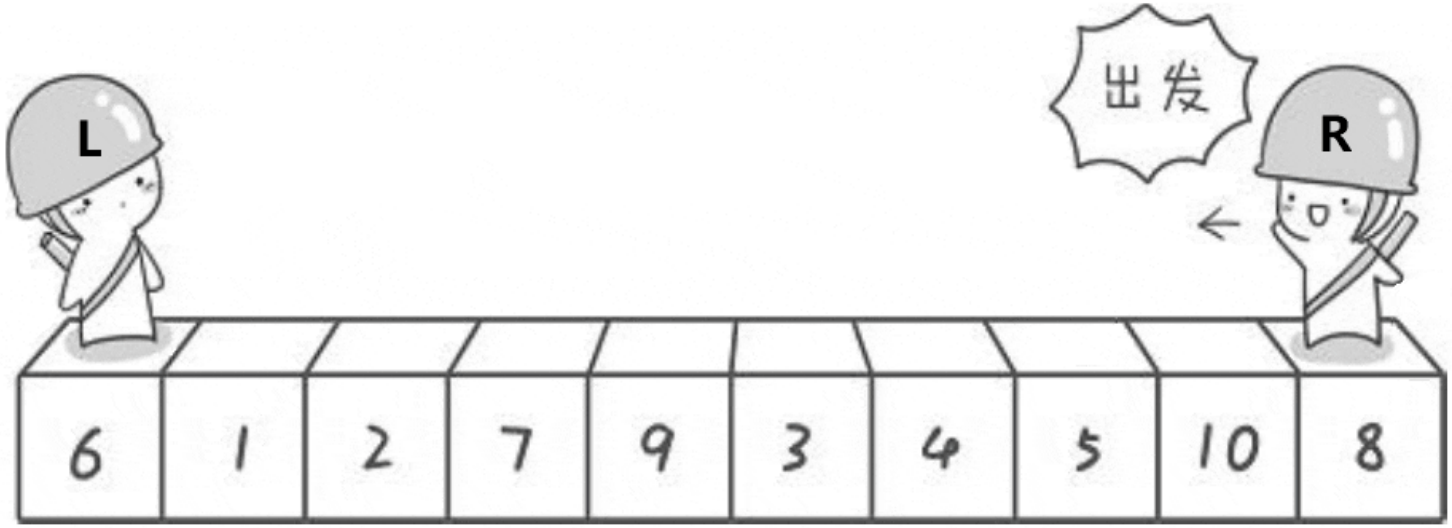
一趟快速排序

- 设置变量i = 0，变量j = n-1；
- 保存记录temp=R_0， R_0为空出的位置（空位在前一区）；
- 令j向前扫描，寻找小于R_0的记录，找到小于R_0的记录R[j]，将记录R[j]移到当前空位中。这时R[j]为新空位（空位在后一区）；
- i自i+1起向后扫描，寻找大于R_0的记录，找到大于R_0的记录R[i]，将记录R[i]移到当前空位中，空位又到了前一区；
- 如此交替改变扫描方向，从两端向中间靠拢，直到i=j，这时i所指的位置为R_0的最终位置

快速排序演示—— 一趟快速排序

先将第一个数据存放在临时变量 **key** ***中，形成一个坑位***

key =



快速排序—示例

- 序列{49， 38， 65， 97，76，13，27，49’}在快速排序过程中各趟排序后的状态

[27	38	13]	49	[76	97	65	49’]
[13]	27	[38]	49	[76	97	65	49’]
13	27	[38]	49	[76	97	65	49’]
13	27	38	49	[76	97	65	49’]
13	27	38	49	[49’	65]	76	[97]
13	27	38	49	49’	[65]	76	[97]
13	27	38	49	49’	65	76	[97]
13	27	38	49	49’	65	76	97

快速排序算法

从右向左扫描

从左向右扫描

```
def partition(start,end):
    pivot = alist[start]
    low = start
    high = end
    while low < high:
        while low < high and alist[high] >= pivot:
            high -= 1
        alist[low] = alist[high]
        while low < high and alist[low] <= pivot:
            low += 1
        alist[high] = alist[low]
    alist[low] = pivot
    return low
```

快速排序算法

```
def _quick_sort(start, end):
    if start < end:
        pivot = partition(start, end)
        _quick_sort(start, pivot - 1)
        _quick_sort(pivot + 1, end)
```

处理左区间

处理右区间

快速排序算法性能分析

- 当待排序记录已经排序时，算法的执行时间最长
  - 第一趟经过n-1次比较，将第一个记录定位在原来的位置上，并得到一个包括n-1个记录的子文件
  - 第二趟经过n-2次比较，将第二个记录定位在原来的位置上，并得到一个包括n-2个记录的子文件；…
  - 这样总比较次数为

$$C_{\max} = \sum_{i=1}^{n-1}(n-i) = \frac{n}{2}(n-1) \approx \frac{n^2}{2}$$

- 快速排序的最坏时间复杂度应为$O(n^2)$

快速排序算法性能分析

- **最好情况下**，每次划分使两个子区的长度大致相等
  - 对长度为n的文件快速排序所需的比较次数 C(n)等于
    - 对长度为n的无序区进行划分所需的比较次数n-1

      加上
    - 递归对所得左、右两个无序子区(长度≤n/2)快速排序的比较次数
  - 快速排序的记录移动次数不大于比较次数
- 快速排序最好时间复杂度为$O(n\log_2 n)$

快速排序算法性能分析

- 快速排序的平均时间复杂度是$T(n)=O(n\log_2 n)$
- 算法需要一个栈空间实现递归
  - 栈的大小取决于递归调用的深度，最多不超过n，若每次都选较大的部分进栈，处理较短的部分，则递归深度最多不超过$\log_2 n$，所以快速排序的辅助空间为$S(n)=O(\log_2 n)$

- 快速排序算法是不稳定的

排序算法介绍

插入排序	1 直接插入排序 2 二分法插入排序 3 Shell排序
选择排序	4 直接选择排序 5 堆排序
交换排序	6 冒泡排序 7 快速排序
分配排序	8 基数排序
归并排序	9 二路归并排序

分配排序

- 把排序码分解成若干部分，然后通过对各部分排序码的分别排序，最终达到整个排序码的排序。

分配排序的例子

- 扑克牌排序
  - –花色：梅花<方块<红心<黑桃),
  - –面值：2<3<…<10<J<Q<K<A),
  - **–规定：花色的地位高于面值。**

- 扑克牌排序有以下两种方法：
  - –先按花色分成4堆，然后将每堆按面值从小到大排序，最后按花色从小到大迭在一起。（每一堆分别进行排序）
  - –先将牌按面值大小分成13堆，然后从小到大把它们收集起来；再按花色分成4堆，最后顺序地收集起来。（分配—收集—分配）

分配排序

- 多关键码排序条件分析

  假设有n个记录$(R_0,R_1,\ldots R_{n-1})$，

  每个记录R_i中含有d个关键码$(k_i^0, k_i^1,\ldots, k_i^{d-1})$，

  则该n个记录对关键码$(k^0,k^1,\ldots,k^{d-1})$有序是指：

  任意两个记录 R_i 和 R_j 满足有序关系

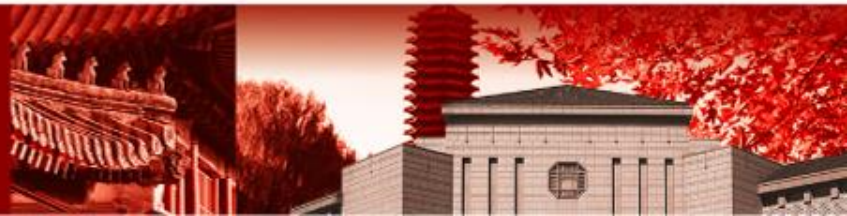
  $$(k_i^0, k_i^1,\ldots, k_i^{d-1}) < (k_j^0, k_j^1,\ldots, k_j^{d-1})$$

  k^0称为最高位关键码，k^{d-1}称为最低位关键码

分配排序

- 分配排序：实现多关键码排序的方法
  - 高位优先法
    - 将文件逐层分割成若干子文件，各子文件独立排序；
  - 低位优先法
    - 对每个关键码都是所有记录参加排序，可通过若干次“分配”和“收集”实现排序。

基数排序



基数排序

- 把每个排序码看成是一个d元组：$K_i=(K_i^0, K_i^1,\ldots, K_i^{d-1})$

  其中每个K_i都是集合$\{C_0，C_1，\ldots，C_{r-1}\}$中的值

  即$C_0\le K_i^j\le C_{r-1}(0\le i\le n-1,0\le j\le d-1)$，其中r称为基数。

- 基数排序的基本思想
  - 排序时先按K_i^{d-1}从小到大将记录分配到r个堆中；
  - 然后依次收集，再按K_i^{d-2}分配到r个堆中…
  - 如此反复，直到对K_i^0分配、收集，便得到最终排序序列

基数排序—例子

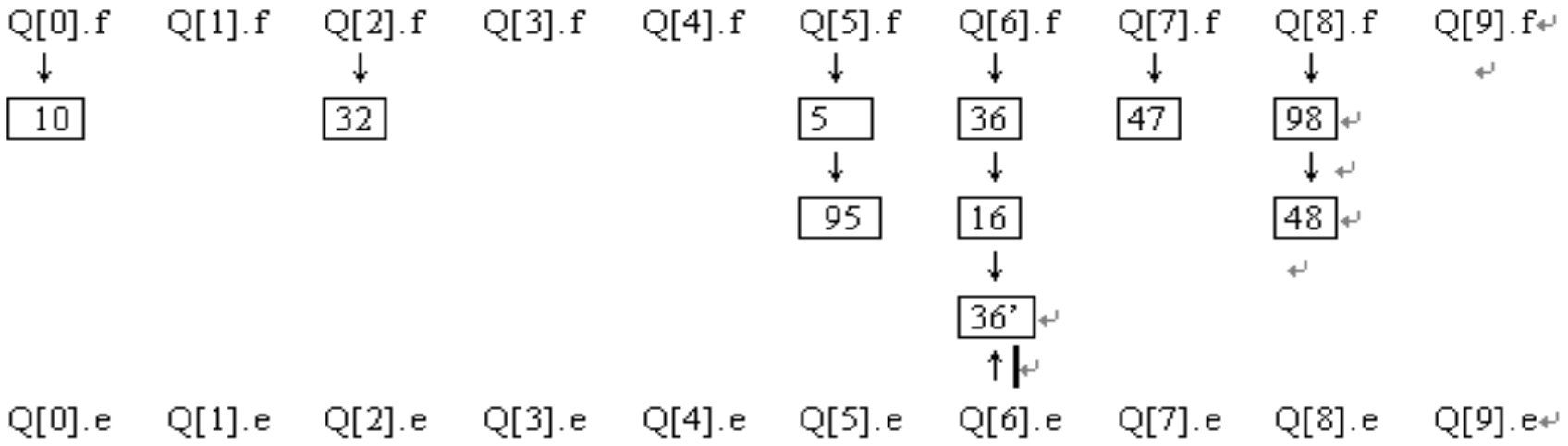
- 数列{ 36, 5, 16, 98, 95, 47, 32, 36’, 48, 10 }， 请用基数排序法排序。

(1)初始状态

36 →5 →16 →98 →95 →47 →32 →36’ →48 →10

基数排序—例子

(2)第一趟分配后

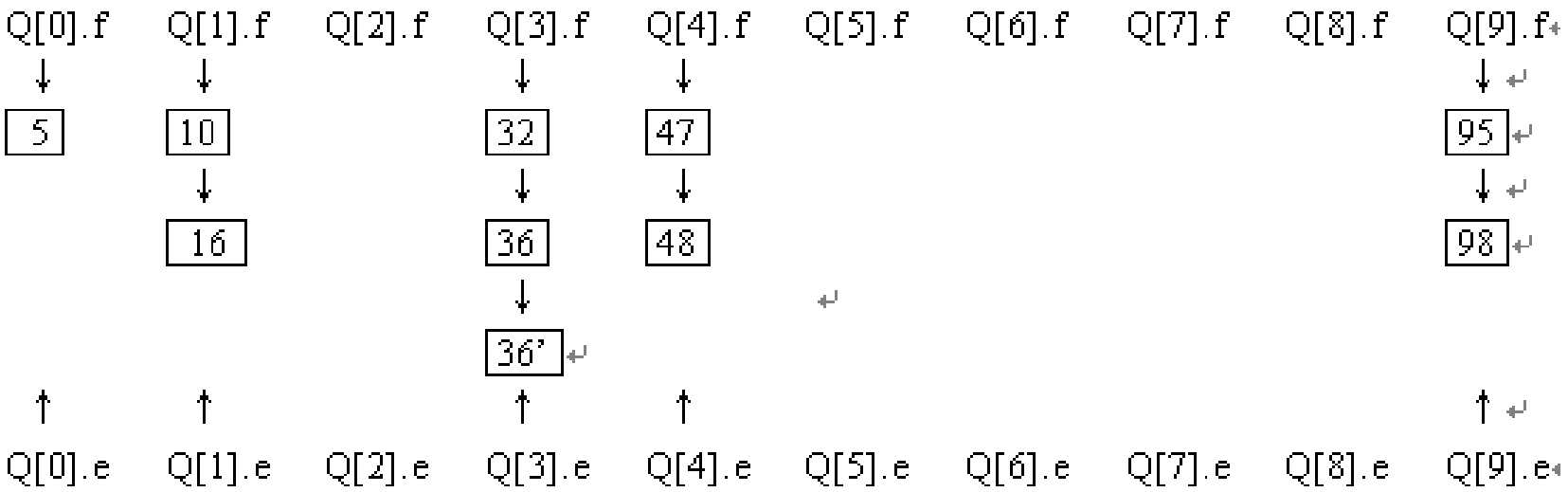


(3)第一趟收集后

10 →32 →5 →95 →36 →16 →36’ →47 →98 →48

基数排序—例子

(4)第二趟分配后



(5)第二趟收集后

5 →10 →16 →32 →36 →36’ →47 →48→95→98

基数排序的数据结构定义

```python
def radix_sort(nums):
    # 获取最大数的位数
    max_num = max(nums)
    max_digits = len(str(max_num))

    # 进行每一位的排序(从个位到最高位)
    for digit in range(max_digits):
        # 创建0-9的桶(列表)
        buckets = [[] for _ in range(10)]

        # 将数字放入对应的桶中
        for num in nums:
            # 获取当前位的数字
            current_digit = (num // 10**digit) % 10
            buckets[current_digit].append(num)

        # 将桶中的数字重新放回原数组
        nums = [num for bucket in buckets for num in bucket]

    return nums
```

基数排序算法性能分析

- 基数排序的时间复杂度T(n)=O(d*(r+n))
  - –每趟排序中，将n个记录分配到队列的时间为O(n)，收集的时间为O(r)，因此一趟排序的时间为O(r+n)；
  - –共要进行d趟排序
- 当n较大、d较小，特别是记录的信息量较大时，基数排序非常有效。

基数排序算法性能分析

- 基数排序的空间复杂度S(n)=O(n+r)
- 基数排序是稳定的

排序算法介绍

插入排序	1 直接插入排序 2 二分法插入排序 3 Shell排序
选择排序	4 直接选择排序 5 堆排序
交换排序	6 冒泡排序 7 快速排序
分配排序	8 基数排序
归并排序	9 二路归并排序

归并排序

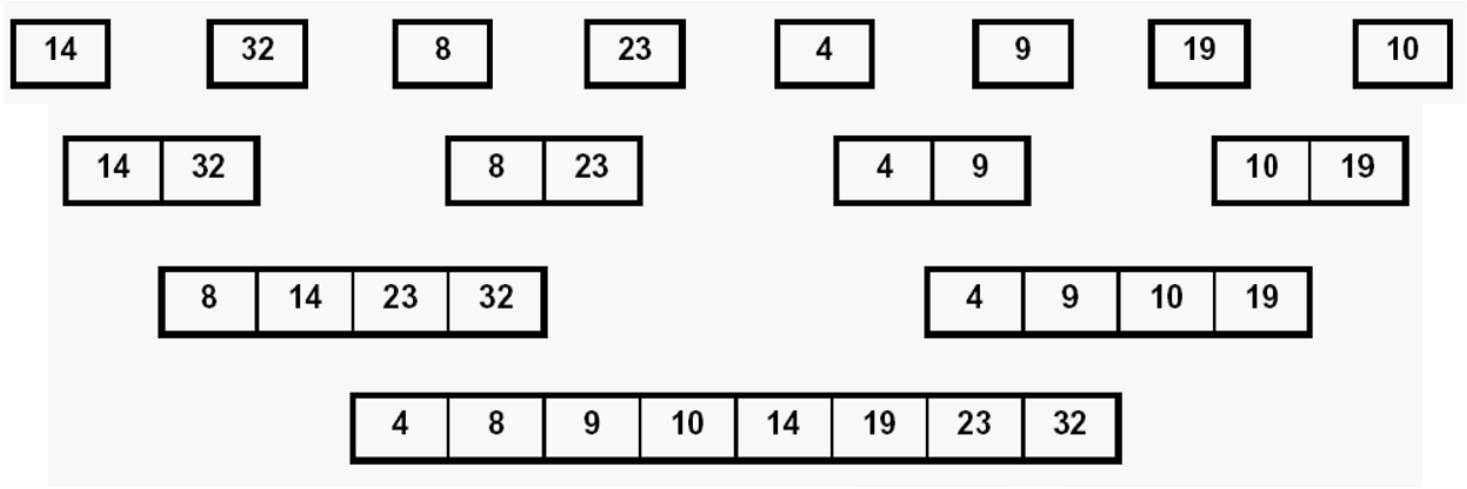
- 把待排序的文件分成若干子文件，每个子文件内排序；再将已排序的子文件合并，得到完全排序的文件。

二路归并排序

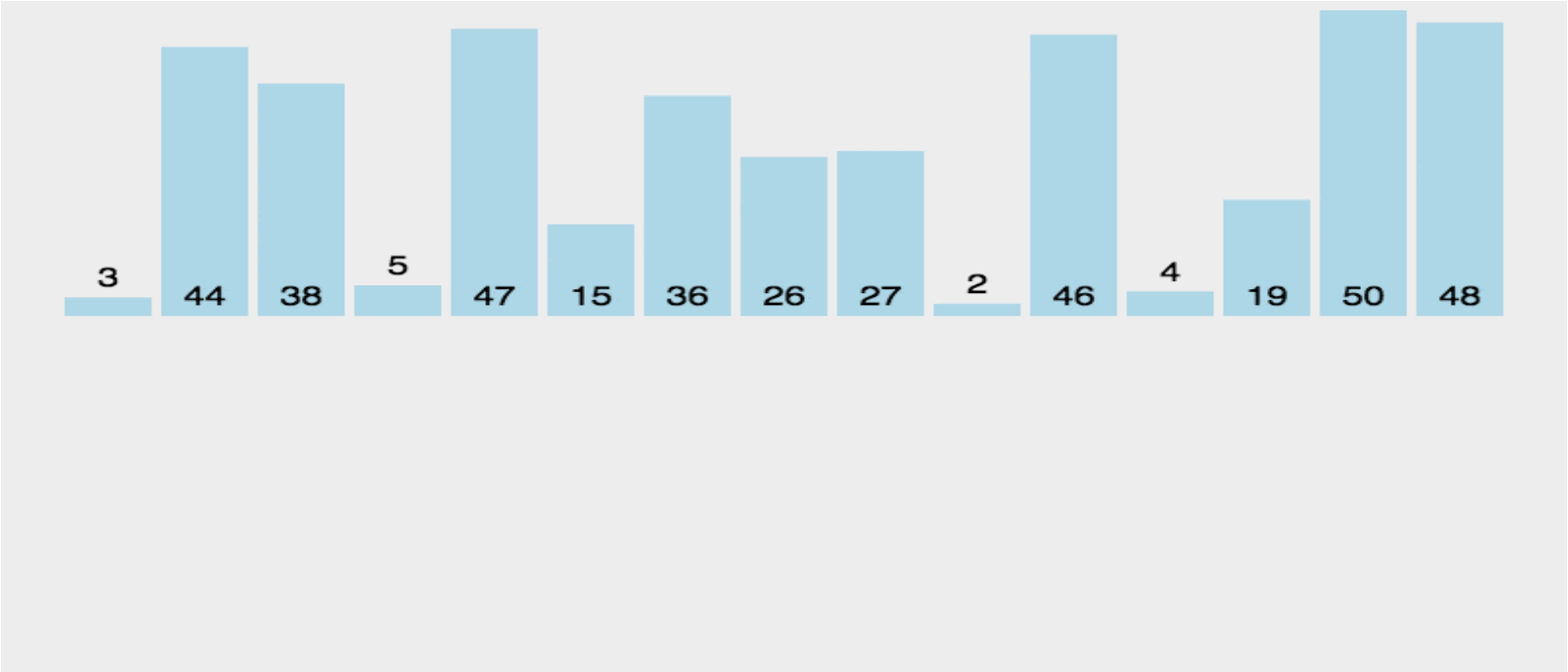
① 设文件中有n个记录可以看成n个子文件，每个文件只有一个记录；

② 将每两个子文件归并，得到n/2个、每个包含2个记录的子文件；

③ 将n/2个子文件归并，如此反复，直到得到一个文件。



二路归并排序演示



两组归并的算法merge

比较左右子数组的元素，按序放入result

分成两组

```
def merge_sort(alist):
    def merge(left, right):   # 1.定义合并两个有序列表的函数
        result = []           # 存储合并结果
        i = j = 0             # 初始化左右子列表的序号
        while i < len(left) and j < len(right):
            if left[i] <= right[j]:
                result.append(left[i])
                i += 1
            else:
                result.append(right[j])
                j += 1
        result += left[i:]    # 将剩余元素直接追加到result末尾
        result += right[j:]
        return result
# 2. 递归终止条件：子列表长度为0或1时直接接返回
    if len(alist) <= 1:
        return alist
    mid = len(alist) // 2
    left = merge_sort(alist[:mid]) # 递归排序左半部分
    right = merge_sort(alist[mid:])# 递归排序右半部分
    return merge(left, right)
```

算法评价

- 二路归并排序算法的时间复杂度为$T(n)=O(n\log_2 n)$
  - 第i次归并以后，有序记录的长度为2^i。因此n个记录的文件排序，必须做$\lceil \log_2 n \rceil$趟归并
  - 每一趟归并所花费的时间是O(n)

- 二路归并排序算法的辅助空间为$S(n)=O(n)$
  - 算法中增加了一个数组**result[ ]**

- 二路归并排序是稳定的

各种排序方法的比较

- 排序算法之间的比较主要考虑以下几个方面：
  - 算法的时间复杂度
  - 算法的辅助空间
  - 排序的稳定性
  - 算法结构的复杂性
  - 参加排序的数据的规模

- 各种排序算法的时间复杂度、辅助空间及算法的稳定性

各种排序算法的比较

分类	算法	最大时间	平均时间	最小时间	空间代价	稳定性
插入排序	直接插入	$O(n^2)$	$O(n^2)$	$O(n)$	$O(1)$	稳定
	二分插入	$O(n^2)$	$O(n^2)$	$O(n\log n)$	$O(1)$	稳定
	Shell排序	$O(n^{3/2})$	$O(n^{3/2})$	$O(n^{3/2})$	$O(1)$	不稳定
选择排序	直接选择	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	不稳定
	堆排序	$O(n\log n)$	$O(n\log n)$	$O(n\log n)$	$O(1)$	不稳定

各种排序算法的比较

分类	算法	最大时间	平均时间	最小时间	空间代价	稳定性
交换排序	冒泡排序	$O(n^2)$	$O(n^2)$	$O(n^2)$	O(1)	稳定
	快速排序	$O(n^2)$	O(nlog n)	O(nlog n)	O(log n)	不稳定
分配排序	桶式排序	O(n+m)	O(n+m)	O(n+m)	O(m)	稳定
	基数排序	O(d·(n+r))	O(d·(n+r))	O(d·(n+r))	O(n+r)	稳定
归并排序	二路归并	O(nlog n)	O(nlog n)	O(nlog n)	O(n)	稳定

算法评价

- 各种排序方法各有优缺点，可适用于不同的场合，应当根据具体情况，选择合适的排序算法。
- 从数据的规模n来看，
  - 当数据规模n较小时，n^2和$n\log_2 n$的差别不大，则采用简单的排序方法比较合适
    - 如直接插入排序或直接选择排序等
  - 当数据规模n较大时，应选用速度快的排序算法
    - Shell排序、堆排序、快速排序及归并排序的排序速度较快

算法评价(续)

- 从算法结构的简单性看，
  - –速度慢的排序算法比较简单、直接
  - –Shell排序法，快速排序法、堆排序法及归并排序法可以看作是对某一种排序方法的改进，算法结构一般都比较复杂

算法评价(续)

- 从文件的初态来看，
  - 当文件的初态已基本有序时，可选择简单的排序方法，如直接插入排序或冒泡排序等
  - 快速排序有可能出现最坏情况，时间复杂度为$O(n^2)$，且此时递归深度为n，即所需栈空间为O(n)
  - 堆排序不会出现象快速排序那样的最坏情况，且堆排序所需的辅助空间比快速排序少
    - 但这两种算法都是不稳定的，如果要求排序是稳定的，则可以选择归并排序方法

算法评价(续)

- 基数排序法所需的辅助空间较大，
  - 其时间复杂度可简化成O(dn)；当排序码的位数d较少时，可进一步简化成O(n)，能达到较快的速度。
  - 但是基数排序只适用于像字符串和整数这类有明显结构特征的排序码，当排序码的取值范围为某个无穷集合时，则无法使用
  - 因此，当n较大，记录的排序码位数较少且可以分解时，采用基数排序方法较好

- 归并排序法可以用于内排序，也可用于外排序。

判断题

- 如果某种算法是不稳定的,则该方法没有实际应用价值
- 对于n个记录进行冒泡排序，所需要的平均时间是O(n）
- 对于n个记录进行快速排序，坏情况下所需要的平均时间是$O(n^2)$
- 对n个元素的序列进行冒泡排序时,最小的比较次数是n-1

选择题

- 下述几种排序方法中,平均时间复杂性最小的是(　　)。

A.直接插入排序
B.选择排序
C.快速排序
D.冒泡排序

C

选择题

- 下述几种排序方法中，一般情况下要求内存量（空间复杂性）最大的是(　)。

A.插入排序

B.选择排序

C.快速排序

D.归并排序

选择题

- 在以下排序方法中，关键字比较的次数与记录的初始排列次序无关的是（　）

A shell排序

B 冒泡排序

C 直接插入排序

D 直接选择排序

选择题

- 对n个不同的排序码的元素进行不减序冒泡排序，在（A）情况下比较的次数最少，其比较次数为（B）；在（C）情况下比较的次数最多，其比较次数为（D）。

- **供选择的答案**

A，C： 1.从大到小排列好的　　2.从小到大排列好的

3.元素无序　　4.元素基本有序逻辑结构

B，D： 1.n+1　2.n　　3.n-1　　4.n(n-1)/2

选择题

- 排序方法有很多种，其中（A）法从未排序序列中依次取出元素，与已排序序列中的元素做比较，将其放入正确位置上；（B）法从未排序序列中挑选元素，并将其放入已排序序列的一端；而交换排序法则是对序列中元素进行一系列比较，当被比较的元素为逆序时进行交换。（C）和（D）是基于这类方法的两种具体排序方法，且D是比C效率更高。

- 供选择的答案

  1，选择排序 2，快速排序 3，插入排序 4，冒泡排序

  5，归并排序 6，折半排序 7，基数排序

选择题

- 在内排序的过程中，采取不同排序方法会产生不同的排序中间结果。设要将序列**{Q，H，C，Y，P，A，M，S，R，D，F，X}**按字母序的升序重新排列，则
  - （A）是冒泡排序的一次扫描结果，
  - （B）是初始步长为4的希尔排序的一次扫描结果，
  - （C）是二路归并排序的一次扫描结果，
  - （D）是以第一个元素为分界元素的快速排序的一次扫描结果

- 供选择的答案 A~E:
  - 1，F，H，C，D，P，A，M，Q，R，S，Y，X
  - 2，P，A，C，S，Q，D，F，X，R，H，M，Y
  - 3，A，D，C，R，F，Q，M，S，Y，P，H，X
  - 4，H，C，Q，P，A，M，S，R，D，F，X，Y
  - 5，H，Q，C，Y，A，P，M，S，D，R，F，X

Any Questions?