8，排序-I

- 排序的基本概念
- 插入算法：简单插入排序；二分法插入排序
- 选择排序：简单选择排序；堆排序
- 起泡排序
- 快速排序
- 归并和归并排序
- **Python** 系统的排序
- 排序算法的比较和总结
- 理论结果和实际情况

排序：概念

- 排序是整理数据序列使其中元素按特定顺序排列的操作

  排序过程中序列里的数据元素保持不变，但其排列顺序可能改变
- 排序是非常有意义的操作
  - 人们日常生活中经常做各种排序工作
  - 排序是计算中最重要最常要做的工作，许多算法里都需要做排序
  - 统计说明，在计算机数据处理中很大比例的工作是做某些排序
- 排序可以使数据更具结构性，有利于处理。对一些信息处理工作，排序的数据更容易用，排序也是许多算法的重要组成部分。例如
  - 排序后的序列可以采用二分法查找（效率高）
  - 许多算法里需要对计算中使用的数据排序，例如
    - **Kruskal** 算法需要对图中的边按权排序
    - 最佳二叉树生成算法需要数据项按关键码排序

排序

- 一个排序，总是针对一个数据集合的元素组成的序列进行，基于该数据集上的一种序关系。一类常见关系是该集合上的全序
- 集合 **S** 上的全序关系 ⩽，是集合 **S** 上的一种自反，传递，反对称关系

  而且对 **S** 的任意元素 **e, e'**，都有 **e** ⩽ **e'** 或者 **e'** ⩽ **e** 成立

  例：整数集合上的小于等于，字符串的字典序，等等
- 实际排序的情况也有许多变化，例如
  - 如果排序基于的序是简单全序，也就是说，两个元素按序相等当且仅当它们是同一个元素，那么一组数据元素就有唯一确定的顺序
  - 实际情况也常不是这样，常见情况是所考虑的“序”把被排序数据归结为一组（有序的）等价类，同一类的数据元素都认为“相等”
  - 实际例子很多，例如要求按年龄给一个学校的学生排序（很多人同一年出生，甚至同年同月同日出生），按所发薪金给一个公司的雇员排序（一些人工资相同）

排序

- 排序的定义：
  - 假设考虑的数据集合是 **S**，**S** 元素上有一个序关系 ⩽
  - 一个排序算法 **sort** 是从 **S** 的元素序列到 **S** 的元素序列的映射
  - 对 **S** 的任意元素序列 **s**
    - **s' = sort(s)** 是 **s** 的一个排列（元素不变，顺序可能调整）
    - 而且对 **s'** 中任意相邻的元素 **e, e'**，都有 **e** ⩽ **e'**
- 显见：在排序后的序列 **s'** 里，**loc(e)** ⩽ **loc(e')** 当且仅当 **e** ⩽ **e'**，这里的 **loc(e)** 表示 **e** 在 **s'** 里的位置。前一个 ⩽ 表示整数（位置）的小于关系，后一个 ⩽ 表示这一排序所用的数据集上的序
- 显然，同一集数据上可能有多种不同的都有意义的序，在不同时刻，可能需要对同一个数据序列做不同的排序

  因此，每个排序都要明确说明用那种序。另一方面，对一些典型数据集有一些常用的典型序，如整数的小于等于，字符串的字典序等

排序：意义

- 可以抽象讨论排序问题，我们更关心计算机的排序方法（排序算法）
  - 由于排序的重要性，人们提出了许多排序算法

    不同算法的基本想法差别很大

    但都能完成排序工作
  - 有些算法很直观朴素，描述简单，但通常效率较低
  - 有些算法更深刻地反映了排序问题的某些本质

    因此效率较高，但通常也更复杂一些
- 由于排序是计算中最重要工作之一，排序算法的研究一直很受重视
  - 不断有新的研究结果出现，包括经典算法的调整和实现方法
  - 运行环境进步带来的新问题，也促使人们重新考虑和调整已有的算法。如在现代新型硬件结构上（例如多层次存储器上）优化排序算法，在多核系统、多处理器系统、分布式系统中实现排序算法等

基于比较的排序

- 下面主要考虑基于数据元素中的关键码及其序关系比较的排序，这是排序的一类常见情况。其他排序的情况也都与此类似
  - 考虑的是某种数据记录，记录里有一个或几个支持排序的关键码
  - 这些关键码相对简单，有易于判断的序关系，例如整数或字符串
  - 基于关键码排序，就是根据记录中某个关键码（可称之为排序码）的序关系整理记录序列，使之成为按关键码排序的序列
- 针对某个关键码（排序码）对数据记录的序列排序，可能需要按排序码递增的顺序，也可能需要按其递减的顺序
- 在排序过程中，如果待排序记录全部保存在内存，这种工作称为内排序；排序中使用外存（磁盘、磁带等）的排序工作称为外排序

  有些算法更适合用于内排序，有些可能适合处理外排序问题

  下面讨论的都是内排序算法，其中归并排序是多数外排序算法的基础
- 如果数据本身没有自然的序，可考虑用 **hash** 函数将其映射到有序集

基本操作，性质和评价

- 首先考虑排序中的基本操作，主要是两种：
  - 比较关键码，确定序关系（元素比较）
  - 移动数据记录（调整记录的位置和/或顺序）
- 评价排序算法的主要标准是：
  - 执行算法所需的时间（时间复杂性，基于基本操作描述）
  - 执行算法所需要的附加空间（空间复杂性）
    - 在考虑排序算法时，通常不计记录序列本身所占用的空间，因为这部分空间原来就存在和使用着，总是需要的
    - 只考虑排序操作中需要的临时性辅助空间
  - 算法本身的复杂程度也是一个需要考虑的因素。但算法的实现只需要做一次，因此这是一个次要因素

  这些是对任何算法时都需要考虑的性质

排序的性质

排序算法特有的一些性质：

- 稳定性：是排序算法的一种重要性质，稳定的算法可能更有用
  - 待排序序列里可能出现不同记录 R_i 和 R_j $(0 \leqslant i < j \leqslant n-1)$ 但 $K_i = K_j$ 的情况，也就是说：两个不同记录的排序码相等
  - 如果某个排序算法能保证：对待排序序列里任何排序码相同的记录对 (R_i, R_j)，排序都不会改变 R_i 与 R_j 前后顺序，这种排序算法就是稳定的（算法维持序列中排序码相同记录的相对位置）
  - 如果一个排序算法不能保证上述条件，就说它是不稳定的

  原序列的顺序可能隐含一些信息，稳定的排序算法维持这些信息
- 适应性：是排序算法的另一有价值性质
  - 如实际待排序序列较接近排好序的形式，算法能否更快完成工作
  - 具有适应性的算法有时工作得更快（实际效率可能更高），因为实际中常常需要处理接近排序的序列

排序：排序算法分类

- 一些书籍里把被排序的记录序列称为文件
- 排序方法很多，可能按不同方式分类。不少书籍把它们分为如下几类：

  一、插入排序　　二、选择排序　　三、交换排序

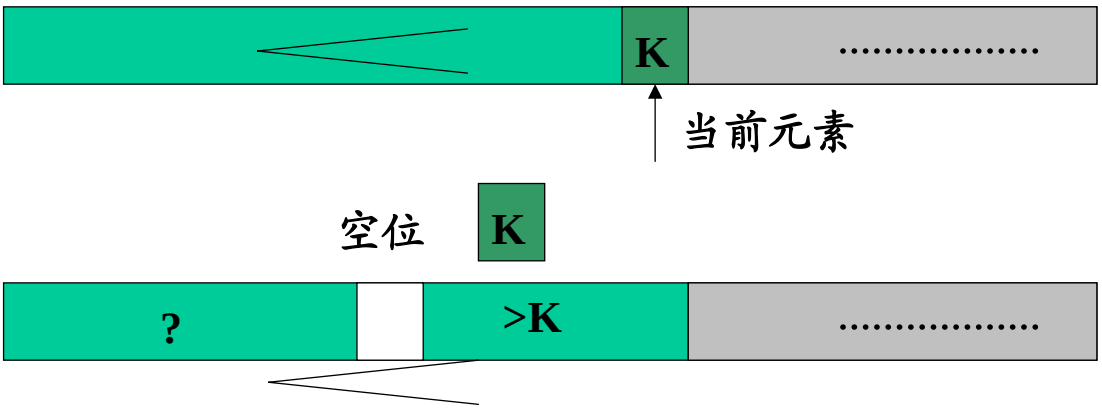
  四、分配排序　　五、归并排序　　六、外部排序
- 下面也采用类似方式讨论其中的一些算法

  一些简单排序只做简单介绍，有些排序前面已经介绍，如堆排序。排序算法很多，这里介绍的只是一些最经典的算法
- 下面排序算法里使用的示例数据结构就是一个表，表中的元素是下面定义的 record 对象：

```
class record:
    def __init__(self, key, datum):
        self.key = key
        self.datum = datum
```


插入排序

- 排序的一种基本方法是维护一个已排序的子序列：
  - 每步把一个待排序记录按关键字插入已排序部分序列中适当位置
  - 一个记录的序列是排序的（作为排序工作的出发点）
  - 当所有记录都插入排序序列时，排序工作完成
- 先考虑一种直接插入排序的简单算法，其基本思想如下图所示：



插入排序

- 待排序的 n 个记录 $\{R_0, R_2, \ldots, R_{n-1}\}$ 保存在表里
  - 在考虑插入记录 R_i (i=1,2…n-1) 时序列分为两个半区 $[R_0, \ldots, R_{i-1}]$ 和 $[R_i, R_{n-1}]$，前半区间里的记录已排好序，后半区尚未排序
  - 处理 R_i 就是在前半区间检索 R_i 的目标位置；用排序码 K_i 依次与 K_{i-1}，K_{i-2}，… 比较，大元素后移，直至找出插入 R_i 的正确位置
  - 对 i = 1, 2, ..., n-1 做这一动作，直至所有元素都插入已排序序列
- 简单插入法的算法很简单

```
def insert_sort(lst):
    for i in range(1, len(lst)):
        x = lst[i]
        for j in range(i, -1, -1):
            if lst[j-1].key > x.key: lst[j] = lst[j-1]
            else: break
        lst[j] = x
```


插入排序：算法分析

- 空间复杂性：只需要一个记录的辅助空间。
- 时间效率：
  - 比较关键码的次数：最小 n-1次，最大 n(n-1)/2次
  - 移动记录的次数：最小为 2*(n-1)，最大：(n+2)(n-1)/2
- 平均比较次数 $\sum_{j=1}^{i-1} p_j c_j = \frac{1}{i-1} \sum_{j=1}^{i-1} j = \frac{i}{2}$
- 总比较次数 $\sum_{i=2}^{n} \frac{i}{2} = O(n^2)$
- 平均情况：比较 $O(n^2)$，移动 $O(n^2)$，算法的时间复杂性是 $O(n^2)$
- 如果被排序序列已经排好序，简单插入排序只需要线性时间，对接近排序的序列工作得更快（适应性）。本算法稳定

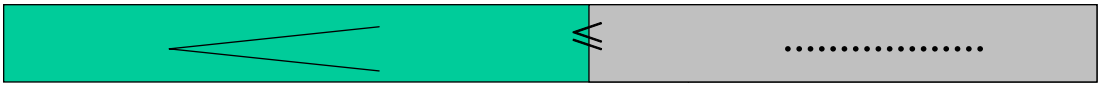
插入排序

- 在插入排序中需要查找元素插入位置
  - 由于是在排序序列里查找，可以用二分法减少比较的次数。在 **k** 个元素的已排序序列里检索，只需做 **log k** 次比较。
  - 但所确定位置之后的元素仍需要后移，要做线性次元素移动。这一做法有效减少排序中关键码比较次数，但不减少元素移动次数
  - 二分法查找插入排序的时间和空间复杂性与简单插入一样。通过适当安排，也可以得到稳定性
- 自我练习题：写一个采用二分检索的稳定的插入排序算法，用一些数据做实验，确定算法的性质
- 另外，对链接表也可以实现插入排序算法（前面已介绍过）
- 一些算法以插入排序作为基础算法：
  - **shell** 排序，见张老师教材
  - 有些高效排序算法在表很短时改用插入排序，形成一种混合方法

选择排序

- 基本思想：
  - 维护最小的 **i** 个记录的已排序序列
  - 每次从剩余未排序的记录中选取关键码最小的记录，存放到已排序序列之后，作为序列的第 **i +1** 个记录，使已排序序列增长
  - 空序列作为排序开始；被选序列只剩一个元素时（它必然为最大），整个排序过程完成

  如果需要从大到小排序，只需每次选取最大元素
- 直接选择算法：以空排序序列开始；每次从未排序记录中选排序码最小的记录，与未排序段的第一个记录交换；直到所有记录排好序
- 直接选择排序算法执行过程中被排序表的状态：



直接选择排序，已排序段中的最大元素小于等于所有未排序元素

选择排序

- 选择元素的基本工作：
  - 用一个内层循环顺序比较，维护已找到的最小记录下标
  - 循环结束时得到未排序段最小记录的下标。将其到已排序段之后
  - 未排序段剩下一个元素时就不必再选择（工作完成）
- 算法也很简单：

```
def select_sort(lst):
    for i in range(len(lst)-1):
        k = i
        for j in range(i, len(lst)):
            if lst[j].key < lst[k].key: k = j
        if i != k:
            lst[i], lst[k] = lst[k], lst[i]
```


选择排序

- 直接选择排序的比较次数与文件初始状态无关
- 直接选择排序的时间复杂度：
  - 记录复制：最好 **0**（增加判断 **i = k** 时不交换），最坏 **2×(n-1)**
  - 比较：**n(n-1)/2** （总是这样）
  - 总的时间复杂度：**$O(n^2)$**
- 稳定性：不稳定
  - 找到小元素后交换，是导致不稳定的根源
  - 如果找到元素后逐个移动前面尚未排序的元素，腾出最前面的空位后存入，这样修改后的算法就是稳定的
- 直接选择排序没有适应性，对任何序列都需要 **$O(n^2)$** 次比较。

  实际试验说明其平均排序效率低于插入排序算法，实际中不常用

选择排序

- 选择排序的低效在于顺序比较
  - 每次选择元素，都是从头开始比较
  - 整个排序中，做了很多重复的比较工作
- 树形选择可能在 **log n** 时间里选出一个元素
- 堆排序是一种高效的选择排序算法，基于堆的概念
  - 其高效的原因是在堆里积累了做过的比较操作得到的信息
  - 建立初始堆需要 **O(n)** 时间，选择一个元素用 **O(log n)** 时间，堆排序算法的时间复杂性是 **O(n log n)**
  - 堆排序在原表中进行，只需要几个工作变量，空间复杂性是 **O(1)**
  - 堆排序不稳定，无适应性（取出最小元素后用最后面元素筛选）
    - 数据沿二叉完全树的分支路径移动，与表的线性结构脱节
    - 很难做出稳定的堆排序算法

排序和交换

- 一种基本观点：一个序列没排好序，那么其中一定有逆序存在
  - 如果交换所发现的逆序记录，得到的序列将更接近排序序列
  - 通过不断减少序列中的逆序，最终可以得到排序序列
- 不同的确定逆序方式和交换方式，可以得到不同排序方法
  - 起泡排序是一种典型的通过交换“逆序对”实现排序的方法

    基本操作是比较相邻元素，遇到相邻的逆序对时交换它们

    反复比较和交换，最终完成排序
  - 后面介绍的快速排序的实现也采用发现逆序和交换数据记录的方法

    但快速排序中最基本的想法是“划分”：按某种标准区分小元素和大元素，通过不断的划分最终到达排序的序列
- 下面先介绍起泡排序，而后再介绍快速排序的思想和算法

起泡排序

- 基本想法：顺序比较序列里相邻的记录，一旦发现逆序记录对就交换它们。通过不断比较和交换，最终得到排好序的序列
- 显然：如果一个序列中每对相邻记录的顺序正确（前一记录不大于后一记录，假定要求按上升序排序），整个序列就是一个排序序列
- 算法，设待排序记录序列是 $(R_1, R_2, ..., R_n)$，需要从小到大排序：
  - 顺序对各个 i 比较相邻记录项 R_i 和 R_{i+1}。发现逆序时交换两个记录
  - 通过一次完整扫描，保证能把一个最大的元素移到最后
  - n-1 次扫描，每次缩短一个项的扫描范围，保证能完成排序

```
def bubble_sort(lst):
    for i in range(len(lst)):
        for j in range(1,len(lst)-i):
            if lst[j-1].key > lst[j].key:
                lst[j-1], lst[j] = lst[j], lst[j-1]
```


起泡排序

- 如果一次扫描没发现逆序就说明排序已完成，可以提前结束
- 改进算法：
  - 用一个辅助变量，内循环开始时赋 False，遇到逆序赋 True
  - 内层循环结束后检查这个变量，值为 False 就结束排序
  - 这样做可能提高效率，改进的算法具有适应性
- 起泡排序的性质：
  - 最坏时间复杂度为$O(n^2)$，平均时间复杂度也为$O(n^2)$，最好情况的时间复杂度为 O(n)（改进的方法，当序列元素已排好序）
  - 起泡排序算法中的辅助空间是 O(1)
  - 起泡排序算法的稳定性依赖于相等元素不交换

快速排序

- 在基于关键码比较的各种内排序算法中，快速排序是实践中平均速度最快的一种算法
  - 快速排序算法在 1960 年前后由英国计算机科学家 C.A.R. Hoare 提出，作为最早用递归方式（Algol 60 语言引进递归描述方式）描述的优美算法，展示了用递归方式描述算法的威力
  - 被认为是“20世纪最具影响力的十个算法”之一
- 快速排序的基本思想是“划分”：
  - 设法把被排序序列按某种标准分为大小两组，两组顺序已定
  - 而后采用同样方式递归地分别对两组记录排序
  - 划分到每个子部分最多包含一个记录时，整个序列的排序完成
- 快速排序算法有许多不同的实现方法，下面介绍其中一种实现

  可以写出链接表的快速排序算法，作为自我练习题

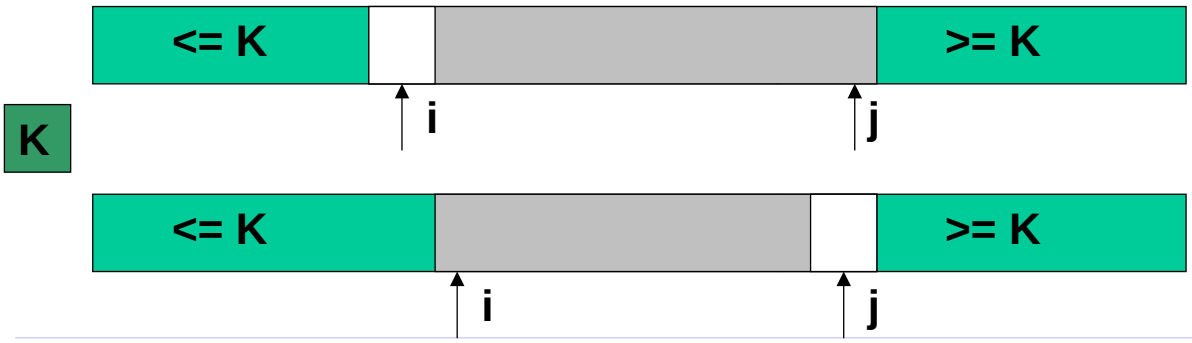
快速排序

- 快速排序的表实现：
  - 在表的内部完成排序，尽可能少使用辅助空间
  - 最简单的划分方式是取序列第一个记录，以其关键码为标准，把关键码小的记录移到数组一边，关键码大的记录移到另一边
  - 一次划分完成后，中间空位就是作为比较标准的记录的位置
  - 而后对两边的记录序列用同样方式分别处理（递归处理）
  - 当一个分段只有一个元素或没有元素时，其排序立即完成
  - 所有分段的排序完成也是整个数组的排序完成
- 可以采用不同的方法选择划分的标准和移动记录，不同做法形成了连续表上的快速排序的不同实现
- 不同书上可能给出不同的快速排序算法，但基本思想一样

  都是基于关键码对连续表中的记录做划分（得到两个分段），以及对分段的递归处理（也可以用循环的方式写出快速排序程序）

快速排序：算法梗概

- （一次）划分的一种实现方法：
  - 设指针 i 和 j，初值分别是序列第一个和最后记录的位置；取出第一个记录，设其排序码为 K，作为划分标准
  - (1), 从 j 所指位置起向前搜索，找到第一个关键字小于 K 的记录并将其存入前面空位；(2) 从 i 所指位置起向后搜索，找到第一个关键字大于 K 的记录并将其存入上一步留下的空位
  - 重复地交替进行上述两个动作，直到 i 不小于 j 为止
  - 将关键码 K 的记录存入空位（其位置已正确确定）



快速排序：算法

- 一次划分完成后对两边子序列按同样方式递归处理。由于要做两个递归，快速排序算法的执行形成了一种二叉树形式的递归
- 用一个主过程，其中调用一个递归定义的过程：

```
def quick_sort(lst):
    qsort_rec(lst, 0, len(lst)-1)
```

- 递归过程的框架：

```
def qsort_rec(lst, l, r):
    if l >= r: return           #无记录或一个记录
    i = l; j = r; pivot = lst[i] # lst[i] 为初始空位
    while i < j:   # 找 pivot 的最终位置
        ... ... # 用 j 向左扫描找小于 pivot 的记录并移到左边
        ... ... # 用 i 向右扫描找大于 pivot 的记录并移到右边
    lst[i] = pivot              # 将 pivot 存入其最终位置
    qsort_rec(lst, l, i-1)      # 递归处理左半区间
    qsort_rec(lst, i+1, r)      # 递归处理右半区间
```

快速排序：算法

```
def qsort_rec(lst, l, r):
    if l >= r: return   #无记录或一个记录
    i = l; j = r
    pivot = lst[i]      # lst[i] 是初始空位
    while i < j:        # 找 pivot 的最终位置
        while i < j and lst[j].key >= pivot.key:
            j -= 1      # 用 j 向左扫描找小于 pivot 的记录
        if i < j:
            lst[i] = lst[j]; i += 1 # 小记录移到左边
        while i < j and lst[i].key <= pivot.key:
            i += 1      # 用 i 向右扫描找大于 pivot 的记录
        if i < j:
            lst[j] = lst[i]; j -= 1 # 大记录移到右边
    lst[i] = pivot              # 将 pivot 存入其最终位置
    qsort_rec(lst, l, i-1)  # 递归处理左半区间
    qsort_rec(lst, i+1, r)  # 递归处理右半区间
```


快速排序：算法分析

- 快速排序工作中的记录移动次数不大于比较次数
  - 其最坏情况时间复杂度为$O(n^2)$，出现在待排序序列为有序时
  - 最好时间复杂度为$O(n \log_2 n)$，如果每次划分能把序列分为长度差不多的两段，就可以得到 $O(n \log_2 n)$
  - 平均时间复杂度是$T(n) = O(n \log_2 n)$
- 为减少最坏情况出现，可考虑修改判据

  例如“三者取中”规则：每趟划分前比较 lst[l]、lst[r] 和 lst[(l+r)/2] 的关键码，取值居中的记录与 lst[l] 交换，基于其关键码划分
- 算法用栈实现递归，栈大小取决于递归深度，不超过 n。若每次选较大半序列进栈，先处理较短的，递归深度将不超过 $\log_2 n$，所以快速排序的辅助空间为 $O(\log_2 n)$
- 常见（包括这里的）快速排序算法是不稳定的（但也有人研究并提出了一些稳定的快速排序算法）