8，排序-2

- 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

归并排序

- 归并是一种序列操作：把两个或更多有序序列合并为一个有序序列
- 基于归并的思想，可以实现排序，例如：
  - 初始时把待排序的 n 个记录看成 n 个有序子序列，长度均为 1
  - 把序列集合里的有序子序列两两归并，完成一遍，序列集合的规模减半，集合中的子序列长度加倍
  - 重复上面操作，最终得到一个长度为 n 的有序序列

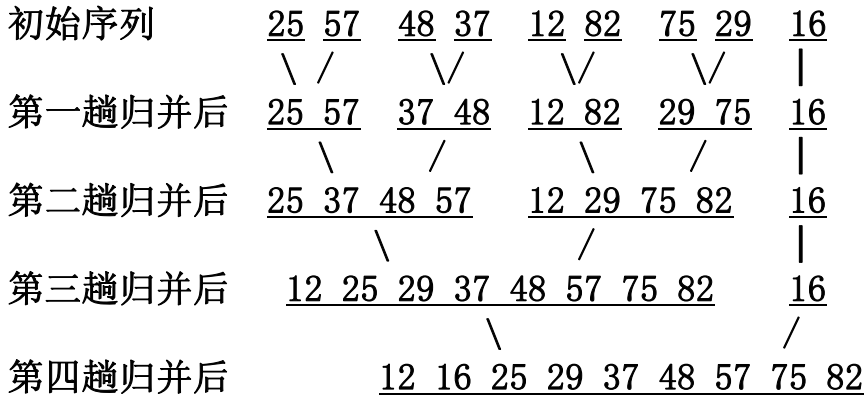
  这种方法称为二路归并排序，也可考虑三路归并或多路归并
- 归并操作特别适合用于处理外存的大量数据
  - 归并是一种顺序处理过程（无论是数据的输入还是输出），与外部数据存储的特点匹配，可以有效处理外存数据
  - 归并中对数据的访问具有局部性，符合外存数据交换的特点
  - 因此归并方法被大量用于做外排序

归并排序

下面讨论顺序表上的二路归并算法。先看一个例子：

假设初始记录序列为 25, 57, 48, 37, 12, 82, 75, 29, 16

用二路归并排序法完成序列的排序



排序后的结果为：**12, 16, 25, 29, 37, 48, 57, 75, 82**

归并排序

- 问题：一次归并结果的序列放在哪里？
- 放在原顺序表里的处理方式可称为“原地归并”
  - 人们提出了一些技术，但比较复杂（有兴趣请自己查资料）
  - 下面的算法把归并结果放到另一块存储区（**O(n)** 辅助空间）
- 下面的表归并排序算法分三层实现（用一个同样大的辅助表）：
  1. 最下层：实现表中相邻的一对有序序列的归并，归并结果存入另一个顺序表里的相同位置
  2. 第二层：基于 **1** 实现对整个表里各对有序序列的归并，所有归并结果顺序存入另一个顺序表
  3. 最高层：在两个顺序表之间往复执行操作 **2**；完成一遍归并后交换两个表的地位，然后再重复操作 **2** 的工作；直至整个表里只有一个有序序列时排序完成
- 一般情况中表长度不是**2**的幂，需要特别考虑表最后的不规则情况

归并排序

```
# from_lst[low..m] 和 from_lst[m+1..high] 是有序段
# 把它们归并为一个有序段 to_lst[low..high]

def merge(from_lst, to_lst, low, m, high):
    i, j, k = low, m+1, low
    while i <= m and j <= high:  # 反复复制两段首记录中较小的
        if from_lst[i].key <= from_lst[j].key:
            to_lst[k] = from_lst[i]; i += 1
        else:
            to_lst[k] = from_lst[j]; j += 1
        k += 1
    while i <= m:     # 复制第一段剩余记录
        to_lst[k] = from_lst[i]
        i += 1; k += 1
    while j <= high: # 复制第二段剩余记录
        to_lst[k] = from_lst[j]
        j += 1; k += 1
```


归并排序

```
def merge_pass(from_lst, to_lst, llen, slen):
    i = 1
    while i + 2*slen - 1 <= llen: #归并长len的两段
        merge(from_lst, to_lst, i, i+slen-1, i+2*slen-1)
        i += 2*slen
    if i + slen - 1 < n: #剩下两段，后段长度小于 slen
        merge(from_lst, to_lst, i, i + slen-1, llen)
    else:  #只剩下一段，复制到数组to
        for j in range(i, n): to_lst[j] = from_lst[j]

def merge_sort(lst):
    slen, llen = 1, len(lst)
    templst = [None for i in range(llen)]
    while slen < llen:
        merge_pass(lst, templst, llen, slen)
        slen *= 2
        merge_pass(templst, lst, llen, slen) # 结果存回原位
        slen *= 2
```

排序完成时，得到的结果可能在 templst 里，这时执行一次 merge_pass 就能把结果拷贝回表 lst

归并排序：算法分析

- 考虑时间复杂度：
  - 做了第 **k** 遍归并后，有序子序列的长度为 2^k
  - 因此完成排序需要做不多于 **log n + 1** 遍归并
  - 每遍归并做 **O(n)** 次比较，总比较和移动次数都为 $O(n \log_2 n)$
- 空间复杂度：
  - 这里给出的算法使用了和待排记录序列等量的空间（**O(n)**）
  - 人们在减少归并排序算法的辅助空间方面做了些研究
- 给出的二路归并算法是稳定的（很容易做到，这里关注了归并相同排序码的记录时的选择顺序），但没有适应性
- 归并技术常用于实现基于归并的外存排序算法
  - 这方面的情况不介绍了

Python 系统的 list 排序

- **Python** 系统里采用了一种混成式的排序算法，称为 **Timsort**
  - 这是一种基于归并的稳定排序算法，其中结合了归并排序和插入排序的技术，最坏时间复杂度是 **O(n log n)**
  - 该算法具有适应性，在被排序的数组元素接近排好序的情况下，它的时间复杂度可能远远小于 **O(n log n)**，达到线性
  - 最坏情况下 **Timsort** 算法需要 **n/2** 工作空间，因此其空间复杂度是 **O(n)**。但在较有利的情况下只需要很少临时存储空间
  - 这一算法比较适合许多实际情况，特别是数据序列分段有序或者基本有序，但其中也有些非有序元素的情况
- **Timsort** 算法是 **Python** 开发过程中由 **Tim Peters** 在**2002**年设计，由于在实际使用中表现很好，已被另外一些重要软件采纳。例如
  - **Java SE7** 不再用原算法，改用 **Timsort**（与原算法不完全兼容）
  - **Android** 平台和 **Gnu** 的开源数值语言 **Octave** 等

排序算法：理论总结

- 下表总结了一些排序算法的时间、空间复杂性，以及稳定性（Shell 排序在我们的课程中没讲）
- 已证明，基于关键码比较的排序时间复杂性下界为 O(n log n)。理想的排序算法是 O(n log n) 时间，O(1) 空间，稳定。还没找到！

排序方法	最坏时间复杂度	平均时间复杂度	辅助空间	稳定性	适应性
直接插入排序	$O(n^2)$	$O(n^2)$	O(1)	Y	Y
二分插入排序	$O(n^2)$	$O(n^2)$	O(1)	Y	Y
表插入排序	$O(n^2)$	$O(n^2)$	O(1)	Y	Y
~~Shell排序*~~	~~$O(n^{1.3})$~~	~~$O(n^{1.3})$~~	~~O(1)~~	~~N~~	~~Y~~
直接选择排序	$O(n^2)$	$O(n^2)$	O(1)	N	N
堆选择排序	O(n log n)	O(n log n)	O(1)	N	N
起泡排序	$O(n^2)$	$O(n^2)$	O(1)	Y	Y
快速排序	$O(n^2)$	O(n log n)	O(log n)	N	N
归并排序	O(n log n)	O(n log n)	O(n)	Y	N

排序算法：实用总结

- 平均时间：实践中快速排序法的速度最快，但最坏情况是平方，不如归并和堆排序；n 较大时归并可能比堆排序快，但需大辅助空间
- 朴素排序算法中直接插入排序最简单，序列中记录“基本有序”或n较小时应优先选用直接插入排序。它也常与其他排序方法结合使用（如一些实用快速排序算法在划分出很短的分段时转用插入排序）
- 序列接近有序时，直接插入排序速度很快，起泡排序也较快（但可能受到“小记录”与最终位置距离的影响）
- 稳定性：简单排序法多是稳定的，归并排序稳定的，但大部分时间性能好的排序都不稳定，如快速排序、堆排序等。一般来说，排序中的比较是在相邻的记录关键字之间进行的排序方法是稳定的
- 记录的主关键字（如各种唯一标示码，学号、身份证编号等）通常具有唯一性。按主关键字排序时排序方法是否稳定就无关紧要
- 按记录的次关键字（其他，如姓名，籍贯，年龄，成绩等）排序时，应根据问题所需慎重选择方法，有时需要稳定算法。如果用了不稳定的排序算法，可能还需要对具有相同关键码的记录段再排序