排序算法小结（python实现）

对冒泡排序、选择排序、归并排序、插入排序、快速排序使用python简单实现，测试不同情况下的执行时间。

一、基本原理与实现代码

（一） 冒泡排序

1、 基本原理

1. 对数组从前到后扫描，比较相邻的两个数A、B，如果A>B，交换A和B。
2. 第一次扫描后，最大的数定位到最后一个，继续对前面n-1个数再次扫描，将第二大的数移动至倒数第二个。
3. 经过n-1次扫描，所有数据排序完毕。
4. 因为排序过程与气泡在水中上升很像，所以叫做冒泡排序
5. 缺点：复杂度较高
6. 优点：适应性强，不仅可用于数组，还可用于链表等数据结构

2、冒泡排序算法复杂度

1. 比较次数 $O(n^2)$：n-1 + n-2 + ... + 1。
2. 交换次数 $O(n^2)$：每次最好情况0，最坏情况与比较次数一致。
3. 短路优化，可以检测每次扫描的交换次数，如果没有交换，说明已经排好序，可以提前退出。

3、python实现冒泡排序

冒泡排序python代码：

```
def bubble_sort(a_list: list):
    for i in range(len(a_list) - 1, 0, -1):
        for j in range(i):
            if a_list[j] > a_list[j + 1]:
                # swap
                a_list[j], a_list[j + 1] = a_list[j + 1], a_list[j]
```

（二） 选择排序

1、基本原理

1. 与冒泡排序很类似，从前到后依次扫描数组。
2. 不同的是扫描时不直接交换数据，仅记录最大数所在位置，扫描完成后与最后位置数据交换。
3. 经过n-1次扫描，排序完成。
4. 排序过程是每次选择一个最大数，移动至最后，所以称选择排序。
5. 与冒泡排序相比，比较次数不变，交换次数减少。

2、选择排序算法复杂度

1. 比较次数 O(n^2): n-1 + n-2 + ... + 1
2. 交换次数 O(n)

3、python实现选择排序

选择排序python代码：

```
# 选择排序
def selection_sort(a_list: list):
    for i in range(len(a_list) - 1, 0, -1):
        pos_max = 0
        for j in range(1, i + 1):
            if a_list[j] > a_list[pos_max]:
                pos_max = j
        a_list[pos_max], a_list[i] = a_list[i], a_list[pos_max]
```

（三） 插入排序

1、基本思路

维护一个有序列表，对每一个元素，与它前方已排好的有序列表比较，插入到合适的位置

2、插入排序算法复杂度

1. 比较次数 O(n^2):每次扫描最好情况下比较1次，最差情况比较n-1次
2. 交换次数 O(n^2):每次最好情况移动0次，最差情况移动n-1次
3. 计算量与要排序的数据有关，有序性越好的数据花费越小

3、插入排序python实现

插入排序python代码：

```
# 插入排序
def insert_sort(a_list: list):
    for i in range(1, len(a_list)):
        tmp_num = a_list[i]
        done = False
        j = i
        while j > 0 and not done:
            if a_list[j - 1] > tmp_num:
                a_list[j] = a_list[j - 1]
                j -= 1
            else:
                a_list[j] = tmp_num
                done = True
```

```
        if not done:
            a_list[0] = tmp_num
```

（四） 归并排序

1、归并排序基本思路

1. 用递归的思想解决排序。将待排序数据分为两半，分别排序后合并。
2. 结束条件：细分到只有一个元素是自然有序。
3. 优点：时间复杂度低。稳定。
4. 缺点：需额外占用内存空间

2、归并排序算法复杂度

1. 每次规模缩小一半，共需经过$\log_2 n$ 次递归。
2. 每次递归比较次数最小n/2,最大n。
3. 总得复杂度为O(nlogn)。
4. 因为合并需要额外占用1倍的存储空间。

3、归并排序**python**实现

归并排序python代码：

```
#  归并排序
def merge_sort(a_list: list):
    if len(a_list) > 1:
        split_point = len(a_list) // 2
        left = a_list[:split_point]
        right = a_list[split_point:]
        # 左右分别排序
        merge_sort(left)
        merge_sort(right)
        # 合并左右两边
        i = j = k = 0
        while i < len(left) and j < len(right):
            if left[i] < right[j]:
                a_list[k] = left[i]
                i += 1
            else:
                a_list[k] = right[j]
                j += 1
            k += 1
        while i < len(left):
            a_list[k] = left[i]
            i += 1
            k += 1
        while j < len(right):
            a_list[k] = right[j]
```

```
            j += 1
            k += 1
```

（五） 快速排序

5.1 基本原理

1. 类似归并排序“分而治之”的思路，按照“中值”将数据分为左右两部分
2. 所有比“中值”小的放在左边，比“中值”大的放在右边，对左右两部分继续排序
3. 结束条件，分割到的元素少于等于1个时自然有序
4. 实现方法：维护两个游标，左游标往右，遇到大于中值的数停止，右游标往左，遇到小于中值的数停止；交换左右游标对应数字；继续直到左右游标汇合；此时右游标对应的位置就是中值位置，交换中值与右游标对应数据；从中值所在处分为左右两部分，递归快速排序。
5. 优点：不需要额外空间， 排序速度快。
6. 缺点：不稳定排序，速度受中值选取合理性影响较大，最差情况时间复杂度为 $O(n^2)$

5.2 快速排序复杂度分析

1. 受“中值”影响较大
2. 最好情况下，如果中值每次都选在差不多中间，数据每次从中间分隔，时间复杂度与归并排序类似，还节省了一些赋值的开销。
3. 最差情况下，每次有一边没有分到数据，都从头分开，时间复杂度就退化到冒泡排序的O(n2)，还增加了递归的开销

5.3 快速排序python实现

辅助递归函数：

```
def fast_sort_helper(a_list, left, right):
    if left < right:
        right_mark = partition(a_list, left, right)
        # 从分割点左右递归调用
        fast_sort_helper(a_list, left, right_mark - 1)
        fast_sort_helper(a_list, right_mark + 1, right)
```

分割函数：中值选择采用最简单的第一个元素作为中值。

```
def partition(a_list, left, right):
    # 仅简单使用第一个元素作为“中值”，最简单优化可从前中后各取一个数，选择最佳的数
    middle_value = a_list[left]
    left_mark = left + 1
    right_mark = right
    done = False
    while not done:
        # 左标往右走
```

```
        while left_mark <= right_mark and a_list[left_mark] <= middle_value:
            left_mark += 1
        # 右标往左
        while right_mark >= left_mark and a_list[right_mark] >= middle_value:
            right_mark -= 1
        if right_mark < left_mark:
            done = True
        else:
            # 交换左右数字
            a_list[left_mark], a_list[right_mark] = a_list[right_mark],
a_list[left_mark]
    # 交换“中间”数与右标所对数，右标就是新的分割点
    a_list[left], a_list[right_mark] = a_list[right_mark], a_list[left]
    return right_mark
```

二、实际运行时间对比

（一） 测试方法

创建一个随机整数列表，使用python模块timeit调用每个排序函数，获取时间。

```
timeit.update两个参数：
repeat代表重复测试次数，可以设置此数值获取平均时间以减小误差
number代表本次测试重复次数，第一次排好序的数据被传输至排序函数
```

在如下几种情况下分别测试排序时间：

数据量500： 列表大小500，重复3次，执行1次

排序方法	时间
python time:	7.246666666666666e-05
bubble sort:	0.02335633333333333
insert sort:	0.011658533333333337
merge sort:	0.0017039333333333333
fast nr sort:	0.0012762333333333348
fast sort:	0.000994666666666667

数据量5000： 列表大小500，重复3次，执行10次

排序方法	时间
python time:	0.00012353333333333332
bubble sort:	0.13095703333333333

排序方法	时间
insert sort:	0.011946099999999996
merge sort:	0.014976866666666663
fast nr sort:	0.1216999666666667
fast sort:	0.12206223333333333

数据量5000：列表大小5000，重复3次，执行1次

排序方法	时间
python time:	0.0009940333333333332
bubble sort:	2.450567
insert sort:	1.3227681333333328
merge sort:	0.023637333333333288
fast nr sort:	0.015928899999999874
fast sort:	0.01364209999999962

数据量15000：列表大小5000，重复3次，执行3次

排序方法	时间
python time:	0.0011597666666666665
bubble sort:	4.945499066666667
insert sort:	1.3075788666666657
merge sort:	0.06137263333333346
fast nr sort:	2.130640966666666
fast sort:	递归深度超出错误

结论：

- python自带排序快至少10倍以上
- 数据有序性对插入排序影响很大，越有续插入越快
- 中值选取对快速排序影响很大，选取不当甚至无法运行

（二） 数据有序时改变中值选取方法对快速排序影响

将使用最左边值作为"中值"简单优化：获取列表左、中、右三个位置的数，采用中间大小的数作为中值。

同样15000数据量：列表大小5000，重复3次，执行3次

排序方法	时间

排序方法	时间
python time:	0.0011853
bubble sort:	5.177140733333333
insert sort:	1.2944408333333328
merge sort:	0.06419519999999916
fast nr sort:	0.04581703333333328
fast sort:	0.03610246666666702

快速排序又快起来了