实验课 05

实验5-1 插入排序

插入排序（Insertion Sort）是一种简单直观的排序算法。它的工作原理是通过构建有序序列，对于未排序数据，在已排序序列中从后向前扫描，找到相应位置并插入。

一般来说，插入排序都采用in-place在数组上实现。具体算法描述如下：

1. 从第一个元素开始，该元素可以认为已经被排序
2. 取出下一个元素，在已经排序的元素序列中从后向前扫描
3. 如果该元素（已排序）大于新元素，将该元素移到下一位置
4. 重复步骤3，直到找到已排序的元素小于或者等于新元素的位置
5. 将新元素插入到该位置后
6. 重复步骤2~5

我们用Python尝试实现插入排序的算法，并给出一个小规模的数组进行测试。

```
def insertion_sort(arr):
    for i in range(1, len(arr)):  # 下标为0元素视作有序，所以从下标为1的元素开始排序（步骤1，步骤6）
        origin = arr[i]
        j = i
        while j > 0:  # 从后向前扫描（步骤2）
            if arr[j-1] > origin:  # 如果有序的某元素大于待排序元素，则该元素向后移动一个位置（步骤3）
                arr[j] = arr[j-1]
            else:  # 否则变量j的值就是待排序元素的插入位置（步骤4）
                break
            j -= 1
        arr[j] = origin  # 如果用内部循环采用for形式，一定要预先定义j的值，否则此语句中j的值未知
```

```
arr_1 = [7,9,0,-1,6,4,3,1,2,5]
insertion_sort(arr_1)
print(arr_1)
```

我们用一个小动画帮助大家理解插入排序的执行过程。

实验5-2 选择排序

选择排序（Selection Sort）是一种简单直观的排序算法。它的工作原理如下。首先在未排序序列中找到最小（大）元素，存放到排序序列的起始位置，然后，再从剩余未排序元素中继续寻找最小（大）元素，然后放到已排序序列的末尾。以此类推，直到所有元素均排序完毕。


```
def selection_sort(arr):
    for i in range(len(arr) - 1):  # 如果前n-1个元素已经有序，则整体也已经有序，否则内部循环变量j会越界
        minIndex = i
        for j in range(i + 1, len(arr)):
            if arr[minIndex] > arr[j]:
                minIndex = j
        if i != minIndex:
            arr[i], arr[minIndex] = arr[minIndex], arr[i]  # 将后未排序元素中的最小值移动至已排序元素的末尾
```

```
arr_2 = [7,9,0,-1,6,4,3,1,2,5]
selection_sort(arr_2)
print(arr_2)
```

实验5-3 快速排序

快速排序（Quick Sort），又称分区交换排序（Partition-exchange Sort），简称快排，一种排序算法，最早由东尼·霍尔提出。在平均状况下，排序n个项目要$O(n\ log\ n)$次比较，在最坏情况下需要$O(n^2)$次比较。

快速排序使用分治策略来把一个序列分为较小和较大的两个子序列，然后递归地排序两个子序列。

步骤为：

1. 挑选基准值：从数列中挑出一个元素，称为“基准”（pivot）
2. 分割：重新排序数列，所有比基准值小的元素摆放在基准前面，所有比基准值大的元素摆在基准后面（与基准值相等的数可以到任何一边）。在这个分割结束之后，对基准值的排序就已经完成，
3. 递归排序子序列：递归地将小于基准值元素的子序列和大于基准值元素的子序列排序。

递归到最底部的判断条件是数列的大小是零或一，此时该数列显然已经有序。

```
def quick_sort(arr, left, right):
    if left >= right:
        return
    pivot = arr[left]
    i = left
    j = right
    while i < j:
        while i < j and arr[j] >= pivot:
            j -= 1
        arr[i] = arr[j]
        while i < j and arr[i] < pivot:
            i += 1
        arr[j] = arr[i]
    arr[i] = pivot
    quick_sort(arr, left, i-1)
    quick_sort(arr, i+1, right)
```

```
arr_3 = [7,9,0,-1,6,4,3,1,2,5]
quick_sort(arr_3, 0, len(arr_3)-1)
print(arr_3)
```

实验5-4 程序性能测试

时间就是金钱！ 如何评测计算机程序的运行效率呢？

一方面，计算机芯片制造商研发出的CPU和GPU性能越来越强，对于同一款大型程序在不同架构的处理器上运行，运行时间会有些许差异，衡量处理器运行性能的常见计量单位是IPS（每秒指令）和FLOPS（每秒浮点运算指令），此外TOP500是针对全球已知最强大的超级计算机系统做出排名与详细介绍的项目。

另一方面，算法设计也是影响计算机程序运行效率的重要因素，前面我们学习了一些**排序算法**，不同的算法有着不同的时间复杂度。小规模的数组排序可能看不出算法运行效率之间的差别，我们需要利用大规模的数据来测试不同算法之间的效率差异。

首先我们导入random和time两个库，并且编写插入排序和快速排序两个函数，然后利用随机数生成一个长度为10,000的数组。

```
import random
```

```
import time


def insertion_sort(arr):
    for i in range(1, len(arr)):  # 下标为0元素视作有序，所以从下标为1的元素开始排序（步骤1，步骤6）
        origin = arr[i]
        j = i
        while j > 0:  # 从后向前扫描（步骤2）
            if arr[j - 1] > origin:  # 如果有序的某元素大于待排序元素，则该元素向后移动一个位置（步骤3）
                arr[j] = arr[j - 1]
            else:  # 否则变量j的值就是待排序元素的插入位置（步骤4）
                break
            j -= 1
        arr[j] = origin  # 如果用内部循环采用for形式，一定要预先定义j的值，否则此语句中j的值未知


def quick_sort(arr, left, right):
    if left >= right:
        return
    pivot = arr[left]
    i = left
    j = right
    while i < j:
        while i < j and arr[j] >= pivot:
            j -= 1
        arr[i] = arr[j]
        while i < j and arr[i] < pivot:
            i += 1
        arr[j] = arr[i]
    arr[i] = pivot
    quick_sort(arr, left, i - 1)
    quick_sort(arr, i + 1, right)


arr1 = []
for i in range(10000):
    arr1.append(random.random())
arr2 = arr1.copy()
```

接下来我们使用time库中`perf_counter()`方法计算两个排序算法的执行时间。

```
t_a = time.perf_counter()
insertion_sort(arr1)
t_b = time.perf_counter()
print(t_b - t_a)
```

```
t_a = time.perf_counter()
quick_sort(arr2,0,len(arr2)-1)
t_b = time.perf_counter()
print(t_b - t_a)
```

两种算法的执行效率高下立判！插入排序的平均时间复杂度为$O(n^2)$，而快速排序的平均时间复杂度为$O(n\ log\ n)$.

`perf_counter()` 方法返回计时器的精准时间（系统的运行时间），包含整个系统的睡眠时间，具有最高的可用分辨率。由于返回值的基准点是未定义的，所以，只有连续调用的结果之间的差才是有效的。

此外，`process_time()` 方法返回当前进程执行 CPU 的时间总和，**不包含睡眠时间。**由于返回值的基准点是未定义的，所以，只有连续调用的结果之间的差才是有效的。

当然，该例只是粗略计算程序运行时间，实际运行时还要考虑快速排序中递归调用带来的影响等客观因素。

实验练习05

1. 请用Python编写冒泡排序的算法；
2. 请对两种解决同一问题（比如求函数导数、查找算法等等）但是时间复杂度不同的算法进行效率比较。