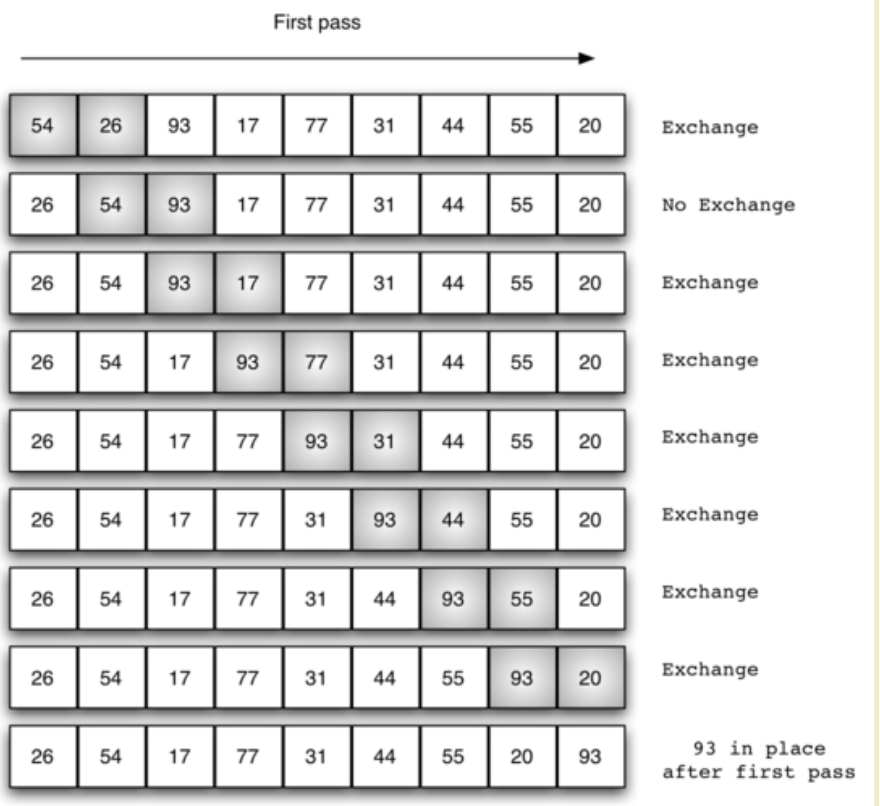
冒泡排序

冒泡排序算法的运作如下:

- 比较相邻的元素。如果第一个比第二个大（升序），就交换他们两个。
- 对每一对相邻元素作同样的工作，从开始第一对到结尾的最后一对。这步做完后，最后的元素会是最大的数。
- 针对所有的元素重复以上的步骤，除了最后一个。
- 持续每次对越来越少的元素重复上面的步骤，直到没有任何一对数字需要比较。



```python
def sort(alist):
    '''
    思路:
    2-取出第一个，去跟第二个比较，如果第一个大，就换位置，
    3-小的话就不变，直到比较完，它左边没有比他大的，右边
    4-for 循环 i 指针不停往下+1，
    '''
    n = len(alist)
    for j in range(0,n-1):
        # 多个循环，每次从头找，但是因为最后一个是找到最大的了，就不用去遍历了,所以-j
        for i in range(0,n-1-j):
            # 从指针第一个开始，如果大于前面的，就换位置，小的就不换，因为大需要排到最后
            if alist[i] > alist[i+1]:
                alist[i] , alist[i+1] =alist[i+1] , alist[i]  # 交换位置
            # 如果后面的大就不需要换，并且，i指针会加一，继续往下


if __name__ == '__main__':
    alist = [54, 26, 93, 17, 77, 31, 44, 55, 20]
    sort(alist)
    print(alist)
```

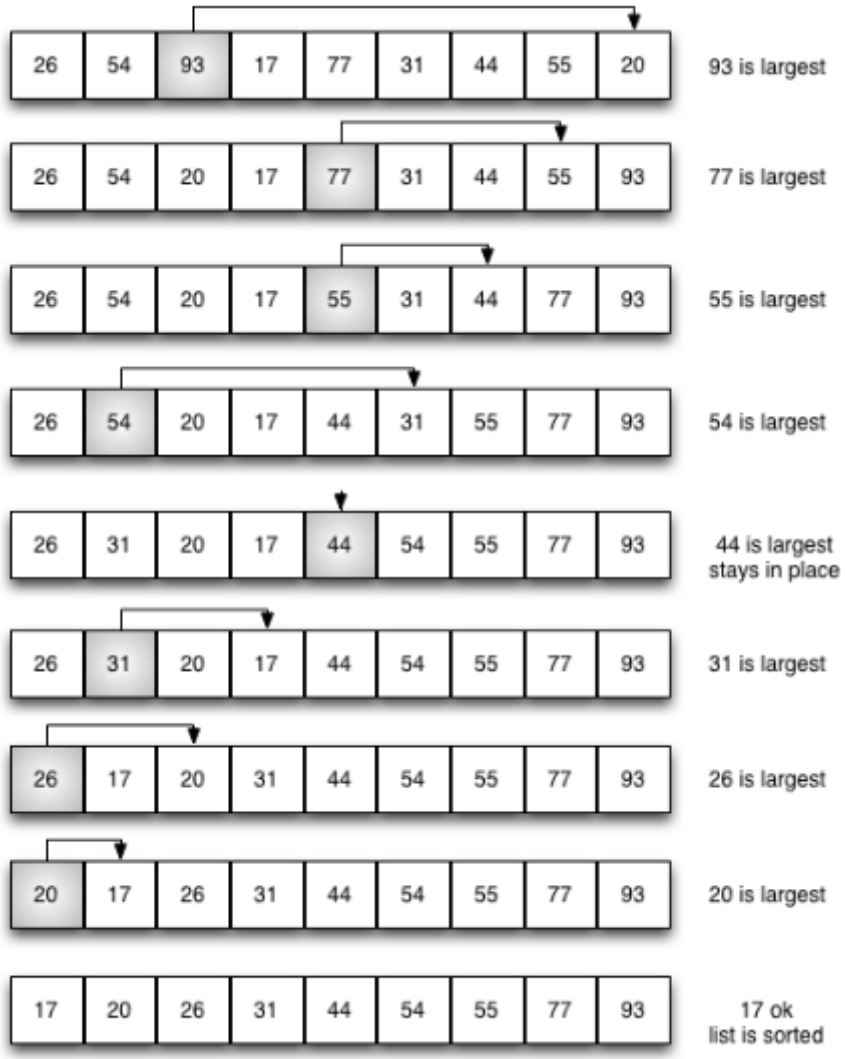
时间复杂度

- 最优时间复杂度：O(n) （表示遍历一次发现没有任何可以交换的元素，排序结束。）
- 最坏时间复杂度：**O(n2)**
- 稳定性：稳定---相同不会交换位置

选择排序：

- 思路：从第一位开始，依次比较找到最小的，最小跟第一位替换，然后再从第二位开始找

选择排序的主要优点与数据移动有关。如果某个元素位于正确的最终位置上，则它不会被移动。选择排序每次交换一对元素，它们当中至少有一个将被移到其最终位置上，因此对n个元素的表进行排序总共进行至多n-1次交换。在所有的完全依靠交换去移动元素的排序方法中，选择排序属于非常好的一种。




```python
def select_sort(alist):
    # 选择排序
    # --内层---1-先取第一个，指针指着，然后对比每个找到最小值，把最小值换过来
    # --外层---取每个都去对比，始终把最小的放在最左边
    n = len(alist)
    for j in range(0,n-1):
        #
        # 先假设第一个 Min_index = 0 可以算出内层循环
        min_index = j
```

```
        for i in range(j+1,n):  # 优化j
            if alist[min_index] > alist[i]:
                # 发现是 alist[i] 更小，指针调换，直到找到最后
                min_index = i
            # 最后执行调换
        alist[min_index] ,alist[j] =alist[j] , alist[min_index]

alist = [30,24,50,60,10,1]
select_sort(alist)
print(alist)

```

时间复杂度

- **最优时间复杂度：O(n2)**
- **最坏时间复杂度：O(n2)**
- 稳定性：不稳定（考虑升序每次选择最大的情况）

插入排序：

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（不断在有序中比较），找到相应位置并插入

```
def insert_sorted(alist):
    '''
    思路：
    1-选择第一个作为有序的部分
    2-其他作为无序部分
    3-把无序部分每一个遍历放进去，
    4-放进去时候都会跟有序部分比较，从左往右，如果新插入的大就停住，相当于在有序中最右边
    5-若是比前面的小，就交换位置
    ====
    代码表现：--外层循环驱动轮流插入进去--内层循环引起插入时候不断进性比较
    '''
    # 选择第一个开始--驱动下面多个进性
```

```
    for i in range(1,len(alist)):
        # 还需要内层循环，第一次插入一个比较了一次，第二次插入就要比较两次
        while i > 0:
            if alist[i] < alist[i-1] : # 如果前面大于后面
                alist[i-1] ,alist[i] = alist[i] , alist[i-1] # 交换位置
                i -= 1 # 继续执行比较
            # 如果遇到了 前面的小于后面，就停止循环了
            else:
                break
```

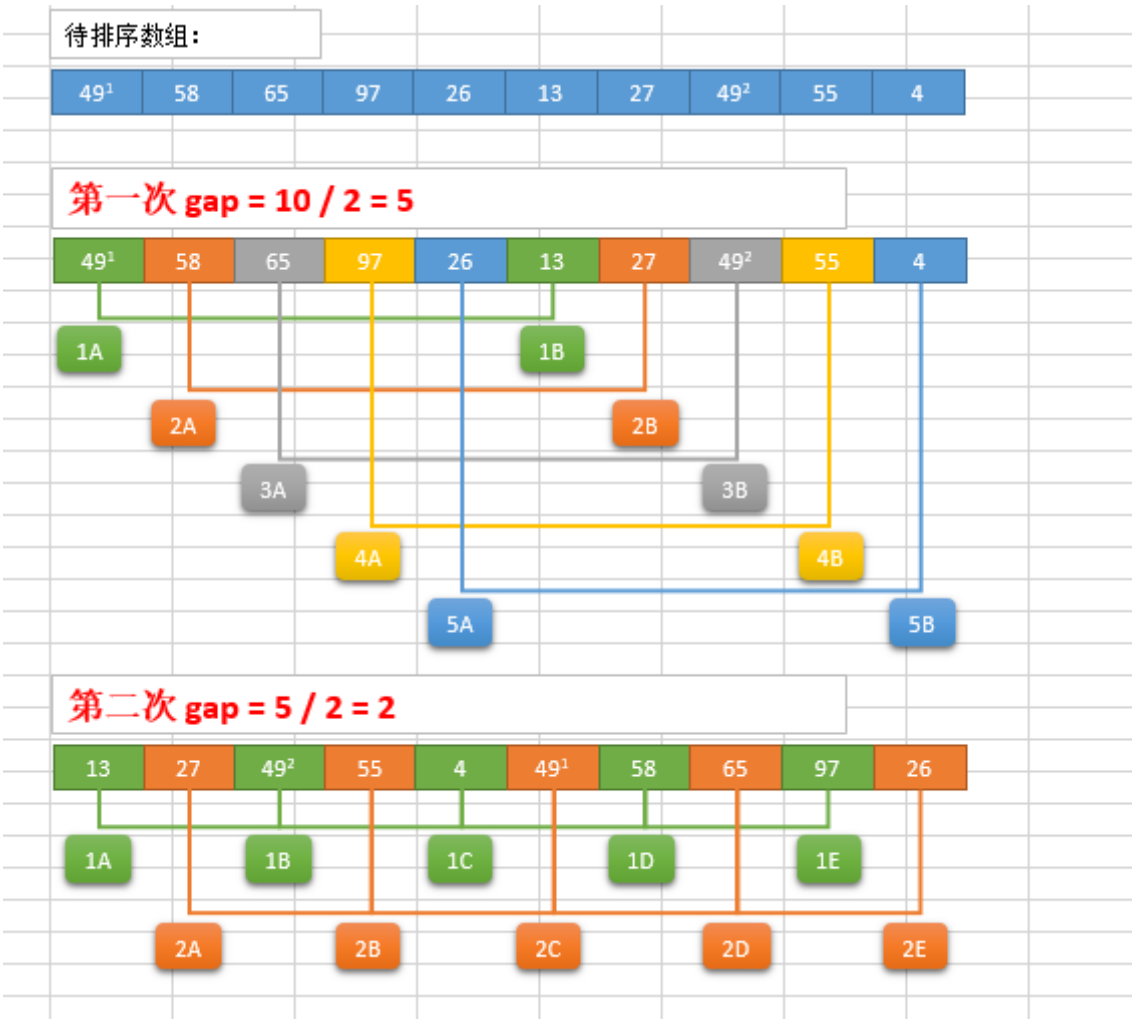
时间复杂度

- 最优时间复杂度：O(n) （升序排列，序列已经处于升序状态）--你
- 最坏时间复杂度：O(n2)
- 稳定性：稳定--list里面前后有个 77，77 插入时候还是保持前后顺序，没有变，所有就稳定

希尔排序

希尔排序(Shell Sort)是插入排序的一种。也称缩小增量排序，是直接插入排序算法的一种更高效的改进版本。希尔排序是非稳定排序算法。

希尔排序是把记录按下标的一定增量分组，对每组使用直接插入排序算法排序；随着增量逐渐减少，每组包含的关键词越来越多，当增量减至1时，整个文件恰被分成一组，算法便终止。




```python
def shell_sort(alist):
    n = len(alist)
    # 初始步长
    gap = n / 2
    while gap > 0:
        # 按步长进行插入排序
        for i in range(gap, n):
            j = i
            # 插入排序
            while j>=gap and alist[j-gap] > alist[j]:
                alist[j-gap], alist[j] = alist[j], alist[j-gap]
                j -= gap
        # 得到新的步长
        gap = gap / 2
```

时间复杂度

- 最优时间复杂度：根据步长序列的不同而不同

- 最坏时间复杂度：O(n2)
- 稳定性：不稳定

快速排序：

总结思路：

通过一趟排序将要排序的数据分割成独立的两部分，其中一部分的所有数据都比另外一部分的所有数据都要小，然后再按此方法对这两部分数据分别进行快速排序，整个排序过程可以递归进行，以此达到整个数据变成有序序列。

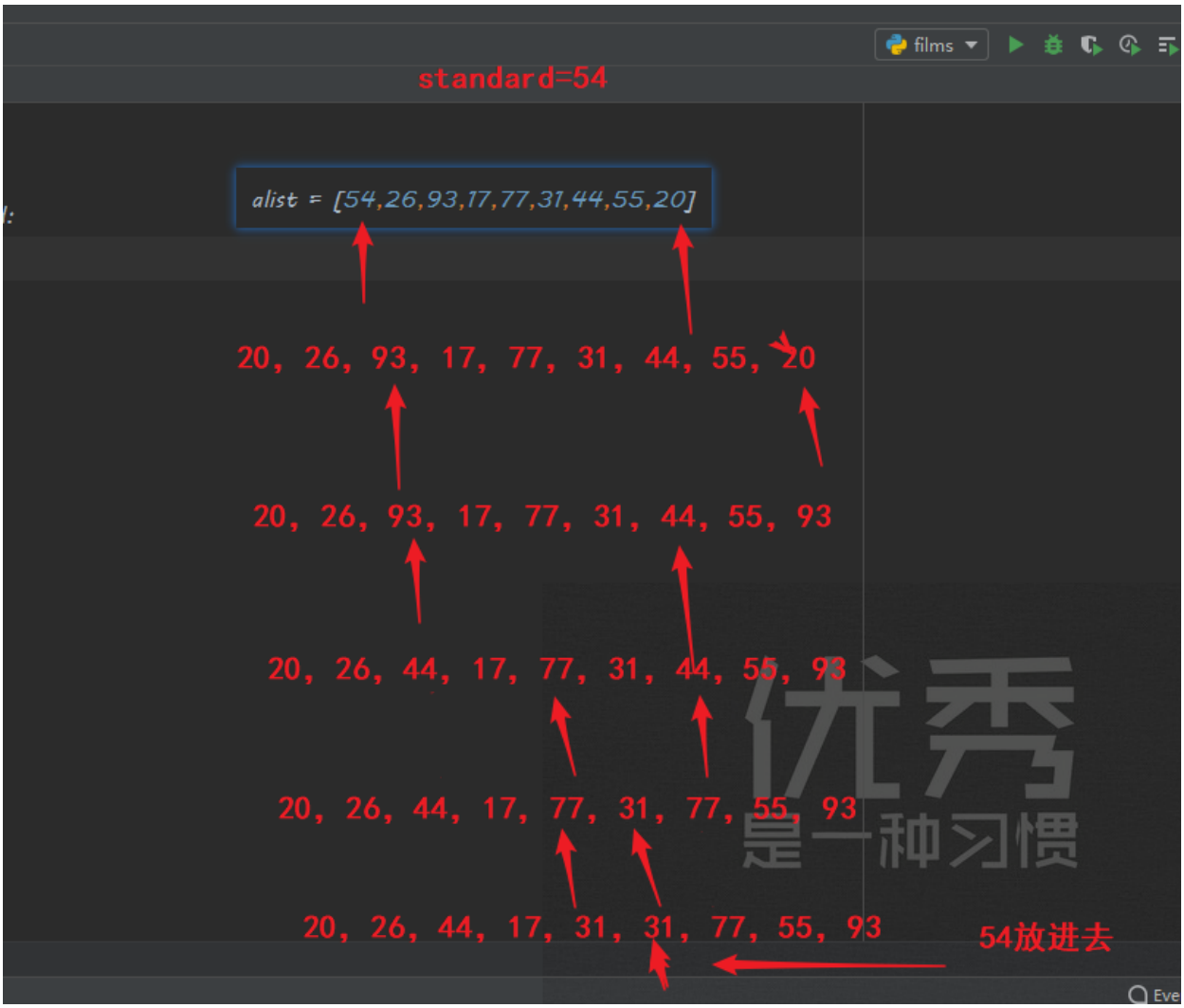
步骤为：

1. 从数列中挑出一个元素，称为**"基准"**（pivot），

2. 重新排序数列，所有元素比基准值小的摆放在基准前面，所有元素比基准值大的摆在基准的后面（相同的数可以到任一边）。在这个分区结束之后，该基准就处于数列的中间位置。这个称为**分区（partition）操作**。

3. 递归地（recursive）把小于基准值元素的子数列和大于基准值元素的子数列排序。

递归的最底部情形，是数列的大小是零或一，也就是永远都已经被排序好了。虽然一直递归下去，但是这个算法总会结束，因为在每次的迭代（iteration）中，它至少会把一个元素摆到它最后的位置去。




```python
def quick_sort(alist,start,end):
    '''
    快速排序
    1- 选取基准值
    2-把左右分区，有左指针和右指针出现，比较对应大小，保证小的在基准值左边，大的在基准值右边
    3-指针重合，证明排好，把基准值插进去
    3-递归继续把左区按照2步骤继续分，右区按步骤2继续分
    4-分到没有得分结束，就是只有一个元素时候
    :param alist:
    :return: sorted_alist
    '''
    # 递归推出条件
    if start >= end:
        return
    # 假设从第一个开始
    standard = alist[start]

```

```
    # 定义指针
    left = start
    right = end
    while left < right:
        # 右指针前行并且没有重合
        while left < right and alist[right] >= standard :
            # 前行
            right -= 1
        # 右边指针值小于标准值
        alist[left] = alist[right]

        # 左指针前行
        while left < right and alist[left] <= standard:
            left +=1
        # 左边指针值大于标准值
        alist[right] = alist[left]
    # 重合后，插入基准值
    alist[left] = standard
    print(alist)
    # 进行递归
    # 递归左分区
    quick_sort(alist,start,left-1) # 从基准值左边递归回去
    # 递归右分区
    quick_sort(alist,left+1,end)  # 从基准值右边递归回去
    # 当递归传入 左指针等于右指针位置时候，就证明只有一个元素了，就退出递归
    # 传入的是alist 对象，所以都是在原来上修改


if __name__ == '__main__':
    alist = [54,26,93,17,77,31,44,55,20]
    quick_sort(alist,0,len(alist)-1)
    print("res:",alist)
```

时间复杂度

- 最优时间复杂度：O(nlogn)

- 最坏时间复杂度：O(n2)
- 稳定性：不稳定

归并排序

归并排序是采用分治法的一个非常典型的应用。归并排序的思想就是**先递归分解数组，再合并数组。**

将数组分解最小之后，然后合并两个有序数组，基本思路是**比较两个数组的最前面的数，谁小就先取谁，取了后相应的指针就往后移一位。**然后再比较，直至一个数组为空，最后把另一个数组的剩余部分复制过来即可。

归并排序的分析

```
def merge_sort(alist):
    if len(alist) <= 1:
        return alist
    # 二分分解
    num = len(alist)/2
    left = merge_sort(alist[:num]) # 递归调用到拆分到一个一个
    right = merge_sort(alist[num:]) # 也是递归调用拆分到一个一个
    # 合并
    return merge(left,right)  # 合并把左右两个列表合并排序，然后一层一层返回

def merge(left, right):
    '''合并操作，将两个有序数组left[]和right[]合并成一个大的有序数组'''
    #left与right的下标指针
    l, r = 0, 0
    result = []
    while l<len(left) and r<len(right):
        if left[l] <= right[r]:
            result.append(left[l])
            l += 1
        else:
            result.append(right[r])
```

```
            r += 1
    result += left[l:]
    result += right[r:]
    return result

alist = [54,26,93,17,77,31,44,55,20]
sorted_alist = mergeSort(alist)
print(sorted_alist)
```

时间复杂度

- 最优时间复杂度：O(nlogn)
- 最坏时间复杂度：O(nlogn)
- 稳定性：稳定

常见排序算法效率比较

排序方法	平均情况	最好情况	最坏情况	辅助空间	稳定性
冒泡排序	$O(n^2)$	$O(n)$	$O(n^2)$	O(1)	稳定
选择排序	$O(n^2)$	$O(n^2)$	$O(n^2)$	O(1)	不稳定
插入排序	$O(n^2)$	$O(n)$	$O(n^2)$	O(1)	稳定
希尔排序	$O(n \log n)$~ $O(n^2)$	$O(n^{1.3})$	$O(n^2)$	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
归并排序	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$	稳定
快速排序	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	$O(\log n)$~ $O(n)$	不稳定