

冒泡排序

选择排序:

插入排序:

希尔排序

快速排序:

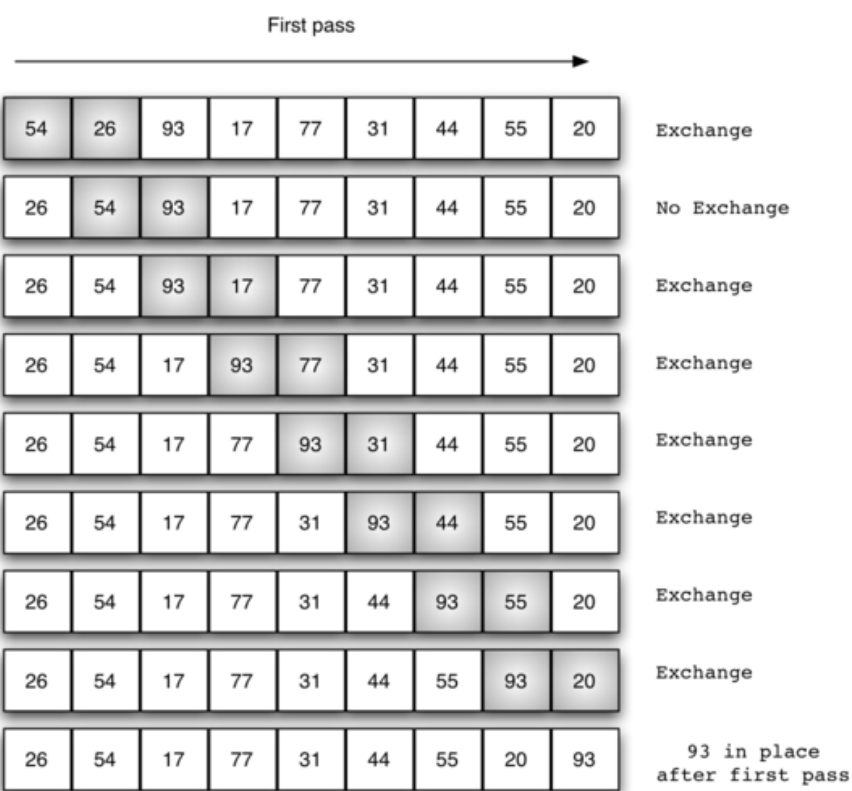
归并排序

常见排序算法效率比较

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。



```

1 def sort(alist):
2     '''
3     思路:
4     2-取出第一个, 去跟第二个比较, 如果第一个大, 就换位置,
5     3-小的话就不变, 直到比较完, 它左边没有比他大的, 右边
6     4-for 循环 i 指针不停往下+1,
7     '''
8     n = len(alist)
9     for j in range(0,n-1):
10         # 多个循环, 每次从头找, 但是因为最后一个找到最大的了, 就不用去遍历了, 所以-j
11         for i in range(0,n-1-j):
12             # 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
13             if alist[i] > alist[i+1]:
14                 alist[i], alist[i+1] = alist[i+1], alist[i] # 交换位置
15             # 如果后面的大就不需要换, 并且, i指针会加一, 继续往下
16
17
18 if __name__ == '__main__':
19     alist = [54, 26, 93, 17, 77, 31, 44, 55, 20]
20     sort(alist)
21     print(alist)

```

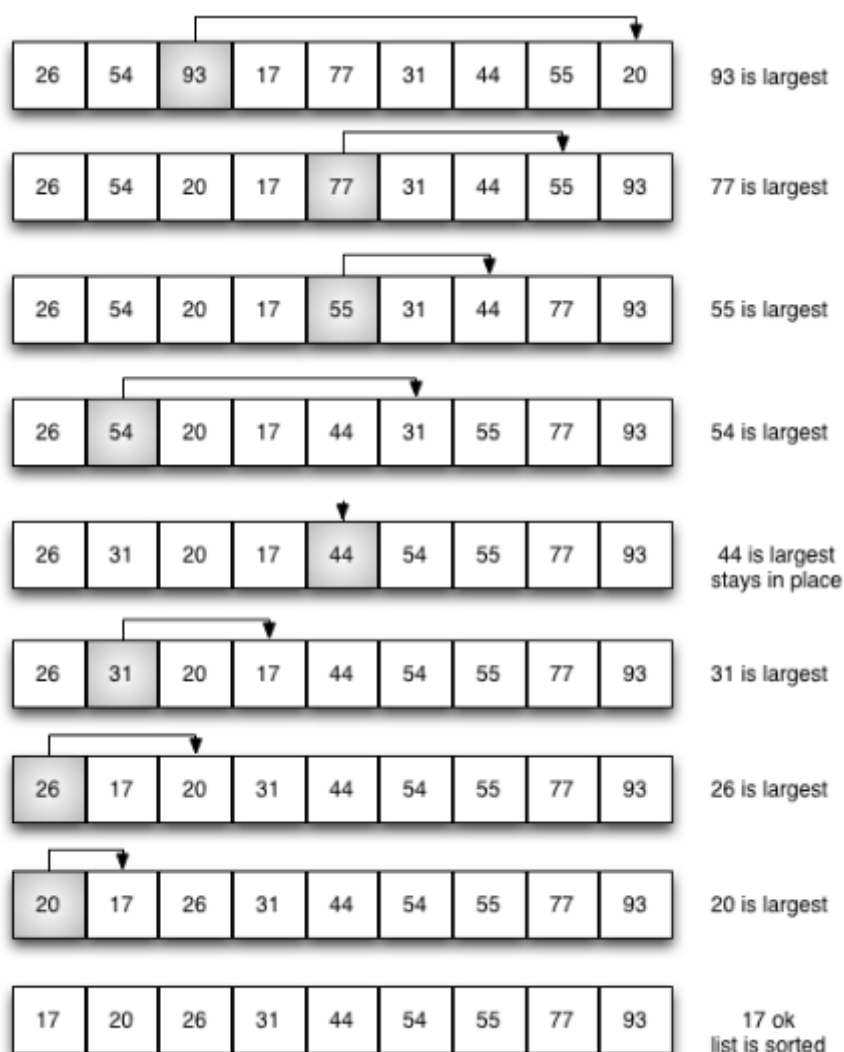
时间复杂度

- 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： **$O(n^2)$**
- 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。



```
1 def select_sort(alist):
2     # 选择排序
3     # --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
4     # --外层---取每个都去对比，始终把最小的放在最左边
5     n = len(alist)
6     for j in range(0,n-1):
7         #
8         # 先假设第一个 min_index = 0 可以算出内层循环
9         min_index = j
```

```

10         for i in range(j+1,n): # 优化j
11             if alist[min_index] > alist[i]:
12                 # 发现是 alist[i] 更小, 指针调换, 直到找到最后
13                 min_index = i
14             # 最后执行调换
15         alist[min_index],alist[j] =alist[j] , alist[min_index]
16
17 alist = [30,24,50,60,10,1]
18 select_sort(alist)
19 print(alist)
20

```

时间复杂度

- 最优时间复杂度: $O(n^2)$
- 最坏时间复杂度: $O(n^2)$
- 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

```

1 def insert_sorted(alist):
2     '''
3     思路:
4     1-选择第一个作为有序的部分
5     2-其他作为无序部分
6     3-把无序部分每一个遍历放进去,
7     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
8     5-若是比前面的小, 就交换位置
9
10    ====
11    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
12    '''
13    # 选择第一个开始--驱动下面多个进性

```

```
13     for i in range(1,len(alist)):
14         # 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
15         while i > 0:
16             if alist[i] < alist[i-1] : # 如果前面大于后面
17                 alist[i-1] ,alist[i] = alist[i] , alist[i-1] # 交换位置
18                 i -= 1 # 继续执行比较
19             # 如果遇到了 前面的小于后面，就停止循环了
20         else:
21             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： $O(n^2)$
- 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

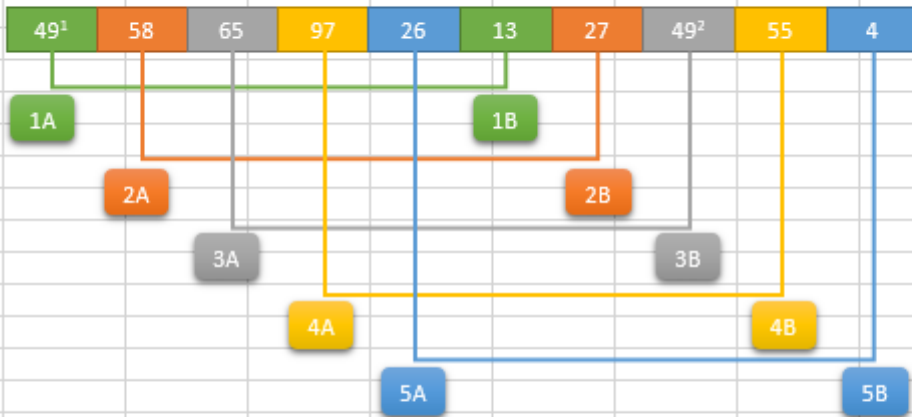
希尔排序(Shell Sort)是插入排序的一种。也称缩小增量排序，是直接插入排序算法的一种更高效的改进版本。希尔排序是非稳定排序算法。

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。

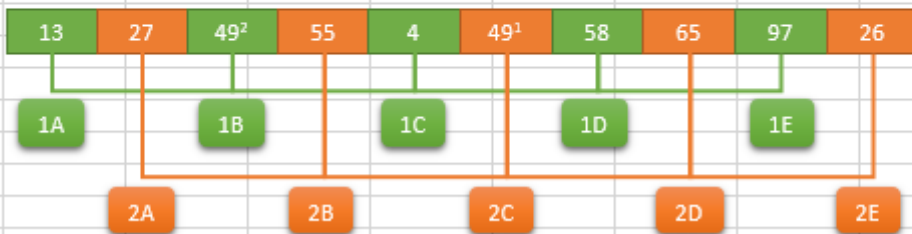
待排序数组：

49 ¹	58	65	97	26	13	27	49 ²	55	4
-----------------	----	----	----	----	----	----	-----------------	----	---

第一次 $\text{gap} = 10 / 2 = 5$



第二次 $\text{gap} = 5 / 2 = 2$



```
1 def shell_sort(alist):
2     n = len(alist)
3     # 初始步长
4     gap = n / 2
5     while gap > 0:
6         # 按步长进行插入排序
7         for i in range(gap, n):
8             j = i
9             # 插入排序
10            while j >= gap and alist[j-gap] > alist[j]:
11                alist[j-gap], alist[j] = alist[j], alist[j-gap]
12                j -= gap
13            # 得到新的步长
14            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： $O(n^2)$
- 稳定性：不稳定

快速排序：

总结思路：

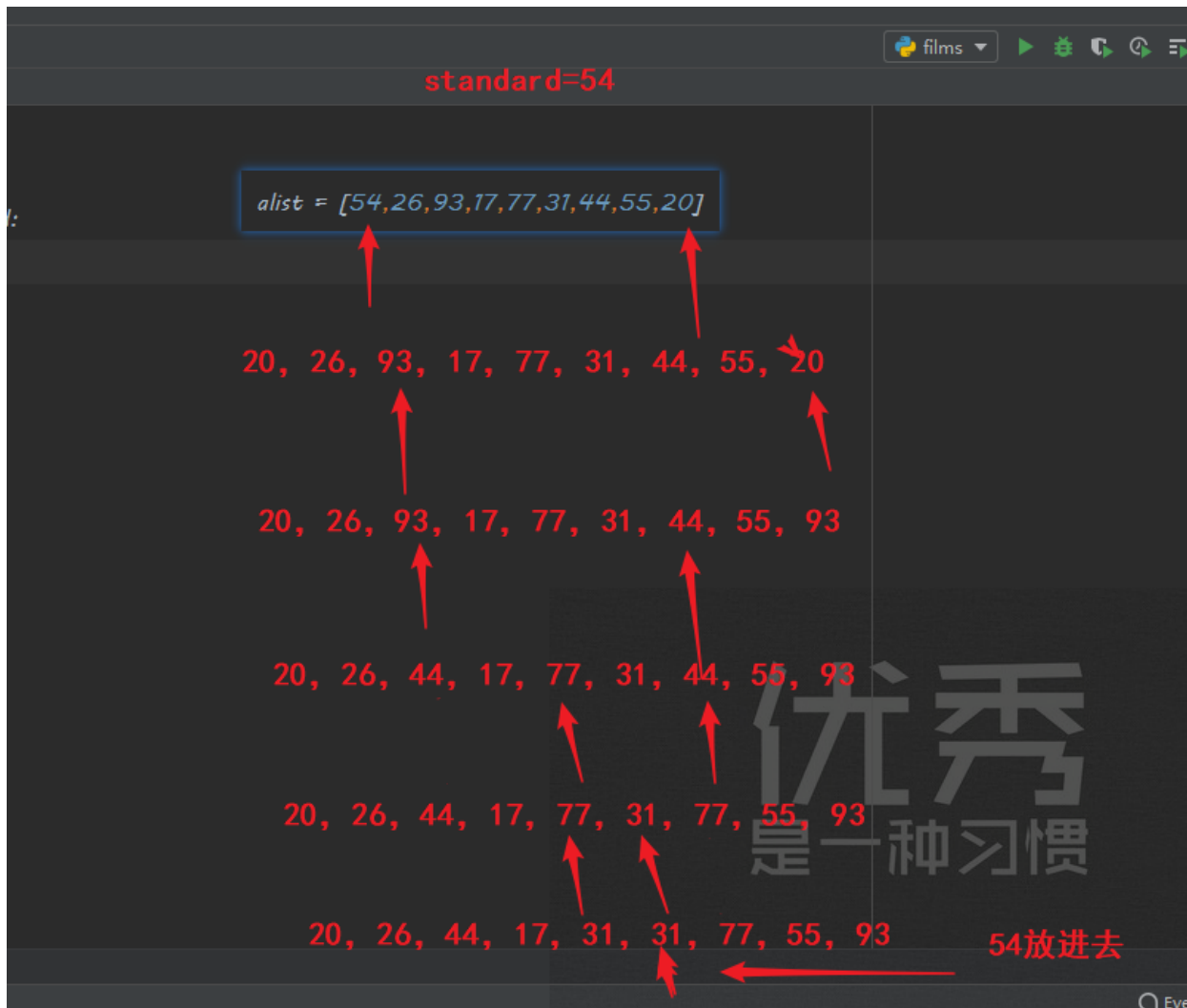
通过一趟排序将要排序的数据分割成独立的两部分，其中一部分的所有数据都比另外一部分的所有数据都要小，然后再按此方法对这两部分数据分别进行快速排序，整个排序过程可以递归进行，以此达到整个数据变成有序序列。

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。



```
1 def quick_sort(alist, start, end):
2     '''
3     快速排序
4     1- 选取基准值
5     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
6     3-指针重合，证明排好，把基准值插进去
7     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
8     4-分到没有得分结束，就是只有一个元素时候
9     :param alist:
10    :return: sorted_alist
11    '''
12    # 递归推出条件
13    if start >= end:
14        return
15    # 假设从第一个开始
16    standard = alist[start]
17
```



```

18     # 定义指针
19     left = start
20     right = end
21     while left < right:
22         # 右指针前行并且没有重合
23         while left < right and alist[right] >= standard :
24             # 前行
25             right -= 1
26         # 右边指针值小于标准值
27         alist[left] = alist[right]
28
29         # 左指针前行
30         while left < right and alist[left] <= standard:
31             left +=1
32         # 左边指针值大于标准值
33         alist[right] = alist[left]
34     # 重合后，插入基准值
35     alist[left] = standard
36     print(alist)
37     # 进行递归
38     # 递归左分区
39     quick_sort(alist,start,left-1) # 从基准值左边递归回去
40     # 递归右分区
41     quick_sort(alist,left+1,end) # 从基准值右边递归回去
42     # 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
43     # 传入的是alist 对象，所以都是在原来上修改
44
45
46 if __name__ == '__main__':
47     alist = [54,26,93,17,77,31,44,55,20]
48     quick_sort(alist,0,len(alist)-1)
49     print("res:",alist)

```

时间复杂度

- 最优时间复杂度：O(nlogn)

- 最坏时间复杂度： $O(n^2)$
- 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

```
1 def merge_sort(alist):
2     if len(alist) <= 1:
3         return alist
4     # 二分分解
5     num = len(alist)/2
6     left = merge_sort(alist[:num]) # 递归调用到拆分到一个一个
7     right = merge_sort(alist[num:]) # 也是递归调用拆分到一个一个
8     # 合并
9     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
10
11 def merge(left, right):
12     '''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'''
13     #left与right的下标指针
14     l, r = 0, 0
15     result = []
16     while l<len(left) and r<len(right):
17         if left[l] <= right[r]:
18             result.append(left[l])
19             l += 1
20         else:
21             result.append(right[r])
```

```
22         r += 1
23     result += left[l:]
24     result += right[r:]
25     return result
26
27 alist = [54,26,93,17,77,31,44,55,20]
28 sorted_alist = mergeSort(alist)
29 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	$O(n \log n) \sim O(n^2)$	$O(n^{1.3})$	$O(n^2)$	$O(1)$	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	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	$O(\log n) \sim O(n)$	不稳定