

# 一、冒泡排序

排序算法 (Sorting algorithm) 是一种能将一串数据依照特定顺序进行排列的一种算法。

**排序算法的稳定性** 稳定性：稳定排序算法会让原本有相等键值的纪录维持相对次序。也就是如果一个排序算法是稳定的，当有两个相等键值的纪录R和S，且在原本的列表中R出现在S之前，在排序过的列表中R也将会是在S之前。

当相等的元素是无法分辨的，比如像是整数，稳定性并不是一个问题。然而，假设以下的数对将要以他们的第一个数字来排序。

```
(5, 3) (2, 1) (2, 7) (4, 6)
```

这个状况下，有可能产生两种不同的结果，一个是让相等键值的纪录维持相对的次序，而另外一个则没有：

```
(2, 1) (2, 7) (4, 6) (5, 3) (维持次序)  
(2, 7) (2, 1) (4, 6) (5, 3) (次序被改变)
```

不稳定排序算法可能会在相等的键值中改变纪录的相对次序，但是稳定排序算法从来不会如此。不稳定排序算法可以被特别地实现为稳定。作这件事情的一个方式是人工扩充键值的比较，如此在其他方面相同键值的两个对象间之比较，（比如上面的比较中加入第二个标准：第二个键值的大小）就会被决定使用在原先数据次序中的条目，当作一个同分决赛。然而，要记住这种次序通常牵涉到额外的空间负担。


冒泡排序 (Bubble Sort) 是一种简单的排序算法。它重复地遍历要排序的数列，一次比较两个元素，如果他们的顺序错误就把他们交换过来。遍历数列的工作是重复地进行直到没有再需要交换，也就是说该数列已经排序完成。这个算法的名字由来是因为越小的元素会经由交换慢慢“浮”到数列的顶端。

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.冒泡排序的分析

第一次交换过程

									
22	67	34	23	97	89	56	13	71	未交换
22	34	67	23	97	89	56	13	71	交换
22	34	23	67	97	89	56	13	71	交换
22	34	23	67	97	89	56	13	71	未交换
22	34	23	67	89	97	56	13	71	交换
22	34	23	67	89	56	97	13	71	交换
22	34	23	67	89	56	13	97	71	交换
22	34	23	67	89	56	13	71	97	交换

我们需要进行 $n-1$ 次冒泡过程，每次对应的比较次数：

冒泡次数	比较次数
1	$n-1$
2	$n-2$
3	$n-3$
...	...
$n-1$	1

## 2.冒泡排序的实现

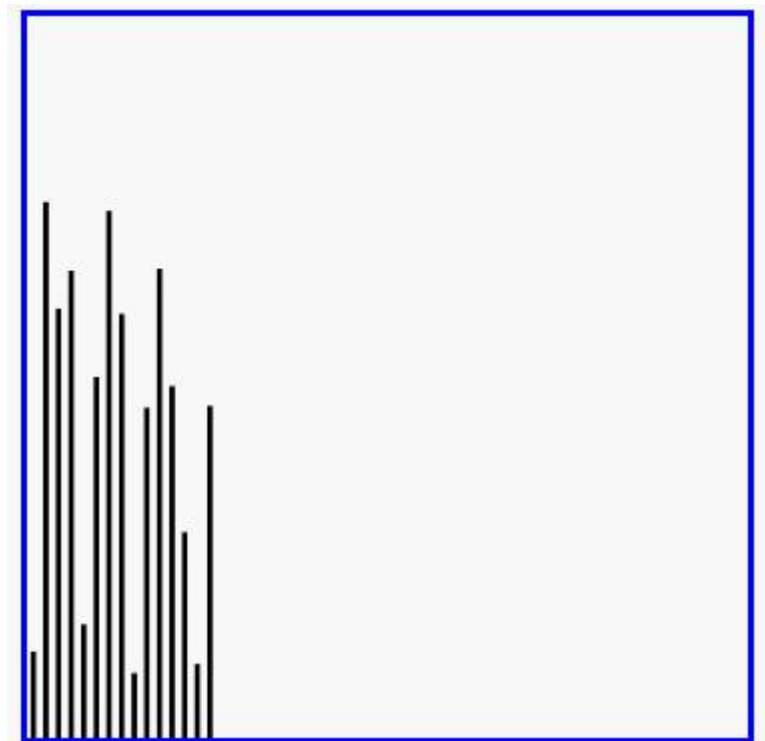
```
def bubble_sort(lis):
    n = len(lis)
    for j in range(n-1):
        for i in range(j+1,n):
            if lis[j] > lis[i]:
                lis[i],lis[j] = lis[j], lis[i]

if __name__ == '__main__':
    list1 = [22,67,34,23,97,89,56,13,71]
    bubble_sort(list1)
    print(list1)
```

### 3.冒泡排序的时间复杂度

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



## 二、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。插入排序在实现上，在从后向前扫描过程中，需要反复把已排序元素逐步向后挪位，为最新元素提供插入空间。

1.插入排序分析

排序过程

22	67	34	23	97	89	56	13	71
22	67	34	23	97	89	56	13	71
22	34	67	23	97	89	56	13	71
22	23	34	67	97	89	56	13	71
22	23	34	67	97	89	56	13	71
22	23	34	67	89	97	56	13	71
22	23	34	56	67	89	97	13	71
13	22	23	34	56	67	89	97	71
13	22	23	34	56	67	71	89	97

位置不变

比67小比22大

比22大

位置不变

比97小比67大

比67小比34大

比前面所有数小

比89小比67大

排序完成

6 5 3 1 8 7 2 4

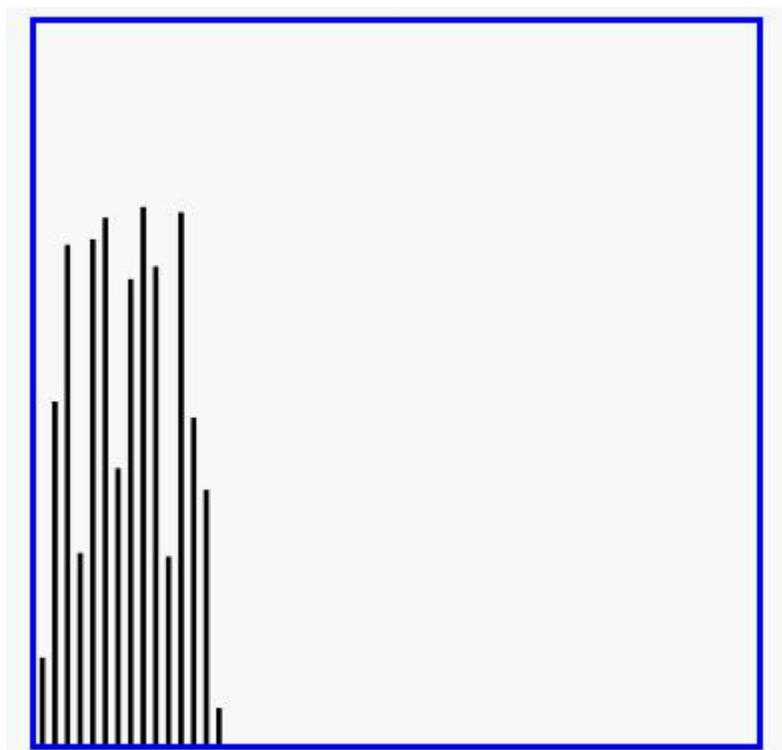
2.插入排序的实现

```
def insertion_sort(lis):
    n = len(lis)
    for i in range(n-1):
        for j in range(i+1,0,-1):
            if lis[j] < lis[j-1]:
                lis[j],lis[j-1] = lis[j-1],lis[j]

if __name__ == '__main__':
    list1 = [22,67,34,23,97,89,56,13,71]
    insertion_sort(list1)
    print(list1)
```

### 3.插入排序的时间复杂度

- 最优时间复杂度： $O(n)$ （升序排列，序列已经处于升序状态）
- 最坏时间复杂度： $O(n^2)$
- 稳定性：稳定



## 三.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。

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.选择排序分析

## 排序过程

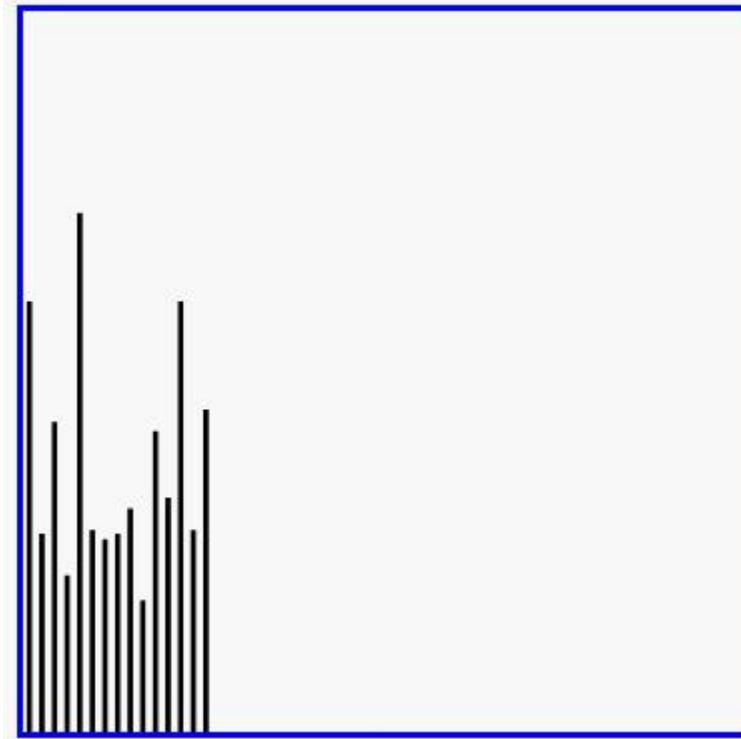
22	67	34	23	97	89	56	13	71	13最小
13	22	67	34	23	97	89	56	71	22最小
13	22	67	34	23	97	89	56	71	23最小
13	22	23	67	34	97	89	56	71	34最小
13	22	23	34	67	97	89	56	71	56最小
13	22	23	34	56	67	97	89	71	67最小
13	22	23	34	56	67	97	89	71	71最小
13	22	23	34	56	67	71	97	89	89最小
13	22	23	34	56	67	71	89	97	排序完成

## 2.选择排序实现

```
def selection_sort(list):  
    for i in range(0, len(list)): #第一趟循环排出有序数列  
        min = i #设定暂时最小值为无序区间第一个元素  
        for j in range(i+1, len(list)): #第二趟排序让min去和无序数列的数作比较找出真正最小值  
            if list[min] > list[j]:  
                min = j  
        list[min], list[i] = list[i], list[min]  
    return list  
  
if __name__ == '__main__':  
    list1 = [22, 67, 34, 23, 97, 89, 56, 13, 71]  
    selection_sort(list1)  
    print(list1)
```

### 3.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）



## 四、快速排序

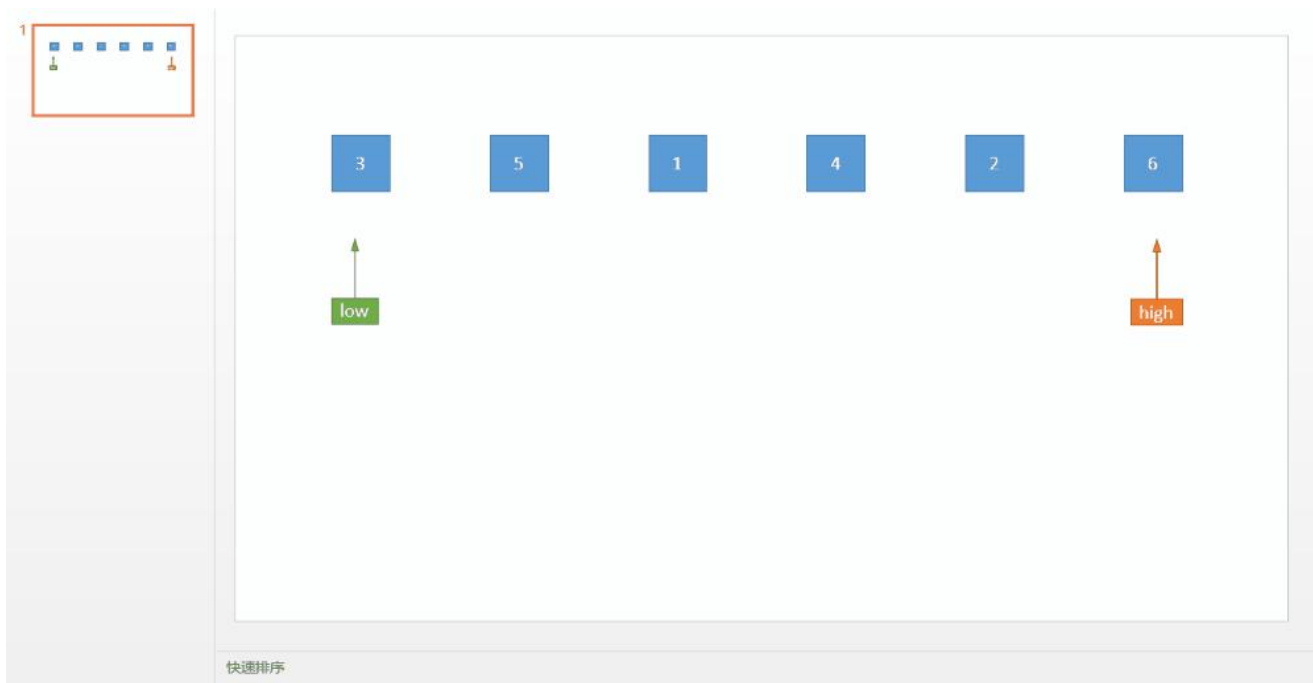
快速排序（英语：Quicksort），又称划分交换排序（partition-exchange sort），通过一趟排序将要排序的数据分割成独立的两部分，其中一部分的所有数据都比另外一部分的所有数据都要小，然后再按此方法对这两部分数据分别进行快速排序，整个排序过程可以递归进行，以此达到整个数据变成有序序列。步骤为：

- 从数列中挑出一个元素，称为“基准”（pivot），
- 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。
- 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.快速排序的分析





## 2.快速排序的实现

```
def quick_sort(lis, start, end):  
    '''快速排序'''  
    #递归的退出条件  
    if start >= end:  
        return  
  
    #设定起始元素为要确定位置的基准元素  
    flag = lis[start]  
  
    #low为左边向右移动的游标  
    low = start  
  
    #high为右边向左移动的游标  
    high = end  
  
    while low < high:  
        #如果low与high未重合，high指向的元素不小于基准元素，则high向左移动  
        while low < high and lis[high] >= flag:  
            high -= 1  
        #将high指向的元素放到low的位置上  
        lis[low] = lis[high]  
  
        #如果low与high未重合，low指向的元素小于基准元素，则low向右移动  
        while low < high and lis[low] < flag:  
            low += 1  
        #将low指向的元素放到high的位置上  
        lis[high] = lis[low]  
  
    #退出循环后，low与high重合，此时所指向的位置即为基准元素flag的正确位置  
    lis[low] = flag  
    #对基准元素左边的子序列进行快速排序  
    quick_sort(lis, start, low-1)
```

```
#对基准元素右边的子序列进行快速排序
```

```
quick_sort(list,low+1,end)
```

```
if name == 'main':
```

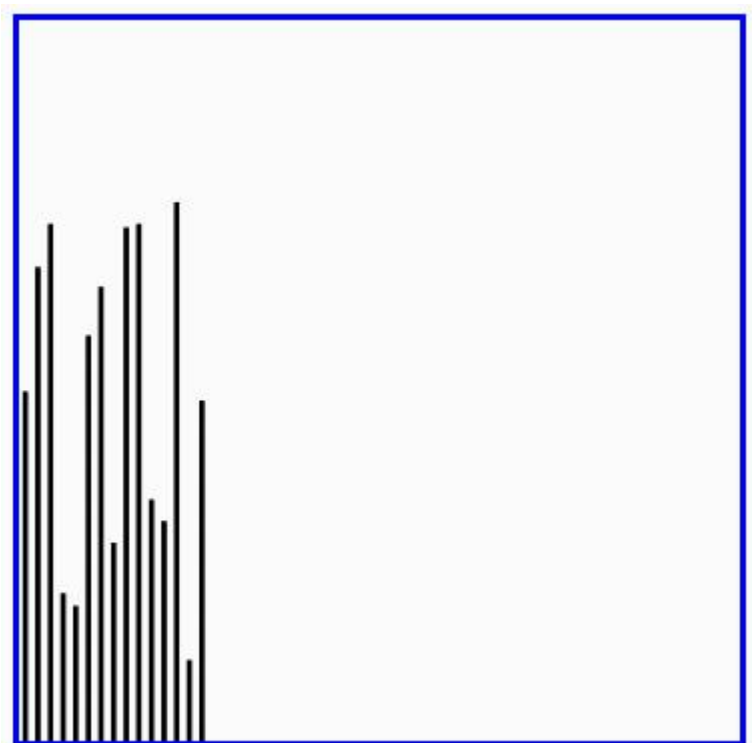
```
list1 = [22,67,34,23,97,89,56,13,71]
```

```
quick_sort(list1,0,len(list1)-1)
```

```
print(list1)
```

### 3.快速排序的时间复杂度

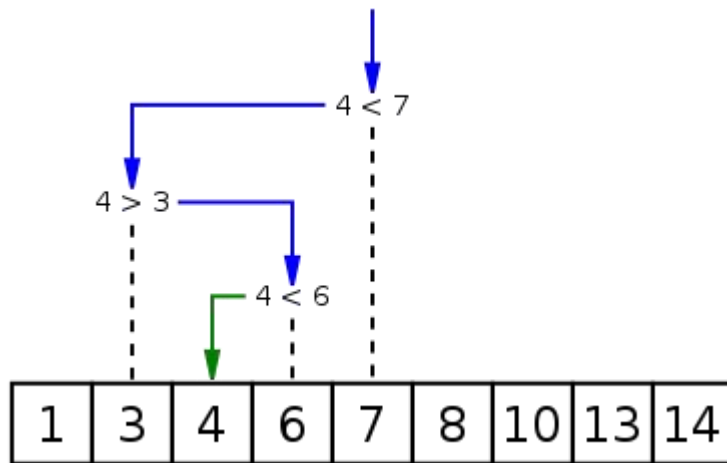
- 最优时间复杂度： $O(n\log n)$
- 最坏时间复杂度： $O(n^2)$
- 稳定性：不稳定



## 五、二分法查找

搜索是在一个项目集合中找到一个特定项目的算法过程。搜索通常的答案是真的或假的，因为该项目是否存在。搜索的几种常见方法：顺序查找、二分法查找、二叉树查找、哈希查找。

二分查找又称折半查找，优点是比较次数少，查找速度快，平均性能好；其缺点是要求待查表为有序表，且插入删除困难。因此，折半查找方法适用于不经常变动而查找频繁的有序列表。首先，假设表中元素是按升序排列，将表中间位置记录的关键字与查找关键字比较，如果两者相等，则查找成功；否则利用中间位置记录将表分成前、后两个子表，如果中间位置记录的关键字大于查找关键字，则进一步查找前一子表，否则进一步查找后一子表。重复以上过程，直到找到满足条件的记录，使查找成功，或直到子表不存在为止，此时查找不成功。



## 1.非递归实现

```
def binary_search(lis, item):
    start = 0
    end = len(lis)-1
    while start <= end:
        mid = (start + end) // 2
        if lis[mid] == item:
            return True
        elif item < lis[mid]:
            end = mid-1
        else:
            start = mid+1
    return False

if name == ' main ':
    list1 = [13, 22, 23, 34, 56, 67, 71, 89, 97]
    print(binary_search(list1,13))
```

Copy

## 2.递归实现

```
def binary_search(lis, item):
    if not len(lis):
        return False
    else:
        mid = len(lis) // 2
        if lis[mid] == item:
            return True
        else:
            if item < lis[mid]:
                return binary_search(lis[:mid], item)
            else:
                return binary_search(lis[mid+1:], item)

if name == ' main ':
    list1 = [13, 22, 23, 34, 56, 67, 71, 89, 97]
```

```
print(binary_search(list1,13))
```