排序算法的稳定性

排序算法（英语：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。

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。

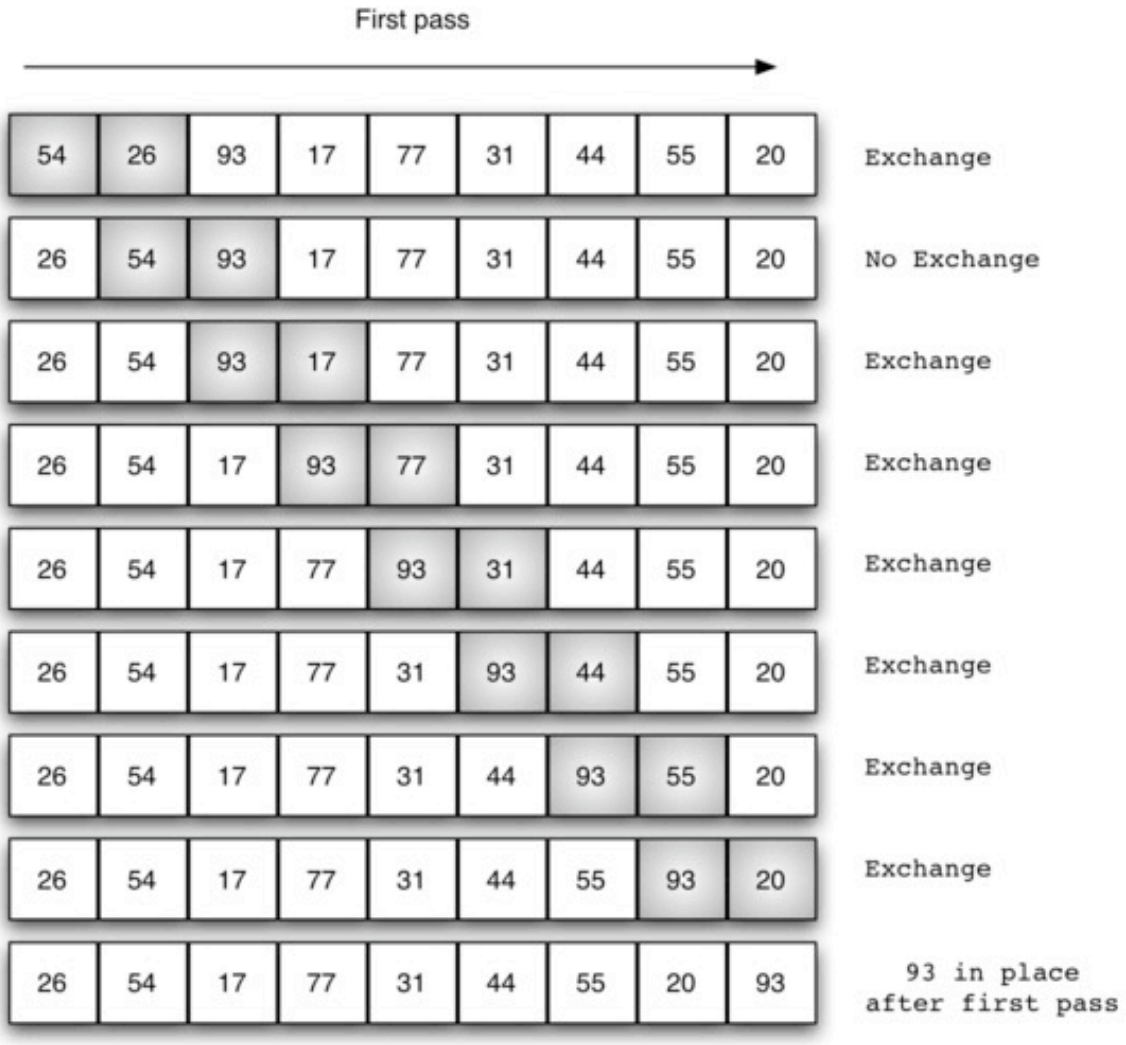
常见排序算法效率比较

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

冒泡排序 Bubble Sort

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 bubble_sort(alist):
  #自己写的!
    for j in range(len(alist)-1, 0, -1):
      for i in range(j):
        if alist[i] > alist[i+1]:
          alist[i], alist[i+1] = alist[i+1], alist[i]

li = [54,26,93,17,77,31,44,55,20]
bubble_sort(li)
print(li)
```

```
[17, 20, 26, 31, 44, 54, 55, 77, 93]
```

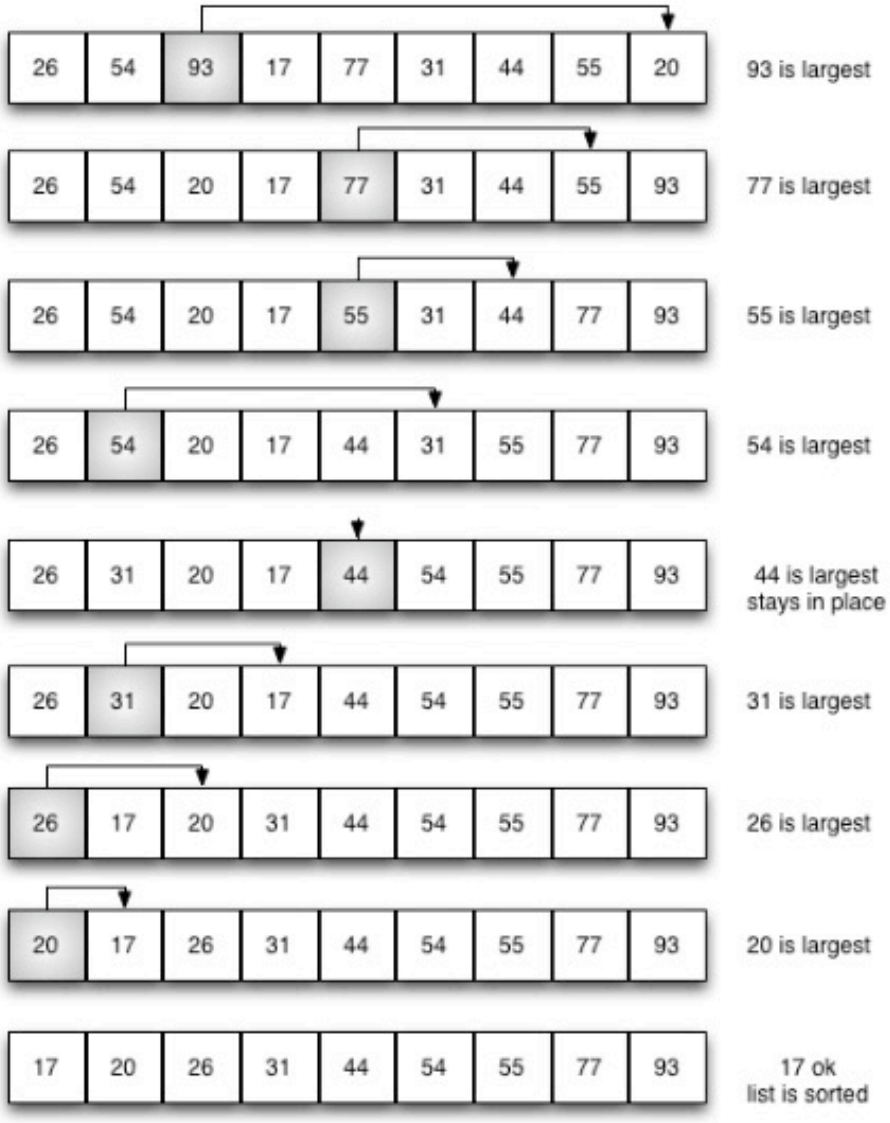
```python
#优化：如果给的就是从大到小排好的数列
def bubble_sort(alist):
    for j in range(len(alist)-1,0,-1):
        # j表示每次遍历需要比较的次数，是逐渐减小的
        count = 0
        for i in range(j):
            if alist[i] > alist[i+1]:
                alist[i], alist[i+1] = alist[i+1], alist[i]
                count += 1
        if count == 0:
          return alist


li = [1,2,3,4,5]
bubble_sort(li)
print(li)
```

```
[1, 2, 3, 4, 5]
```

时间复杂度

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

选择排序（Selection sort）



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(alist):
    n = len(alist)
    # 需要进行n-1次选择操作
    for i in range(n-1):
        # 记录最小位置
        min_index = i
        # 从i+1位置到末尾选择出最小数据
        for j in range(i+1, n):
            if alist[j] < alist[min_index]:
```

```
                min_index = j
        # 如果选择出的数据不在正确位置，进行交换
        if min_index != i:
            alist[i], alist[min_index] = alist[min_index], alist[i]

alist = [54,226,93,17,77,31,44,55,20]
selection_sort(alist)
print(alist)
```

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）

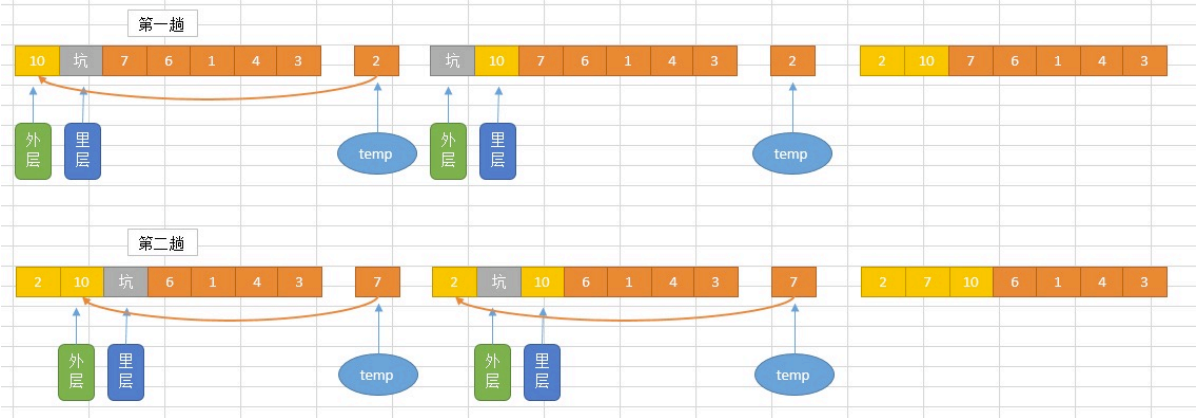
li=[10_1, 4, 5, 7, 10_2, 9] 每次选最大值排到队尾

排完：
li=[4, 5, 7, 9, 10_2,10_1]

10_1, 10_2位置和原来的位置不一样了，故不稳定

插入排序（Insertion Sort）

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。



6 5 3 1 8 7 2 4

```python
def insert_sort(alist):
  for i in range(1, len(alist)):
  # 从第二个位置，即下标为1的元素开始向前插入
    for j in range(i, 0, -1):
    # 从第i个元素开始向前比较，如果小于前一个元素，交换位置：
    # i, i-1,i-2...1
      if alist[j] < alist[j-1]:
        alist[j], alist[j-1]  = alist[j-1], alist[j]

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

```
[17, 20, 26, 31, 44, 54, 55, 77, 93]
```

```python
# 优化
def insert_sort(alist):
  for i in range(1, len(alist)):
  # 从第二个位置，即下标为1的元素开始向前插入
```

```
    j = i
    while j > 0:
    # 从第i个元素开始向前比较，如果小于前一个元素，交换位置：
    # i, i-1,i-2...1
      if alist[j] < alist[j-1]:
        alist[j], alist[j-1]  = alist[j-1], alist[j]
        j -= 1
      else:
        break

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

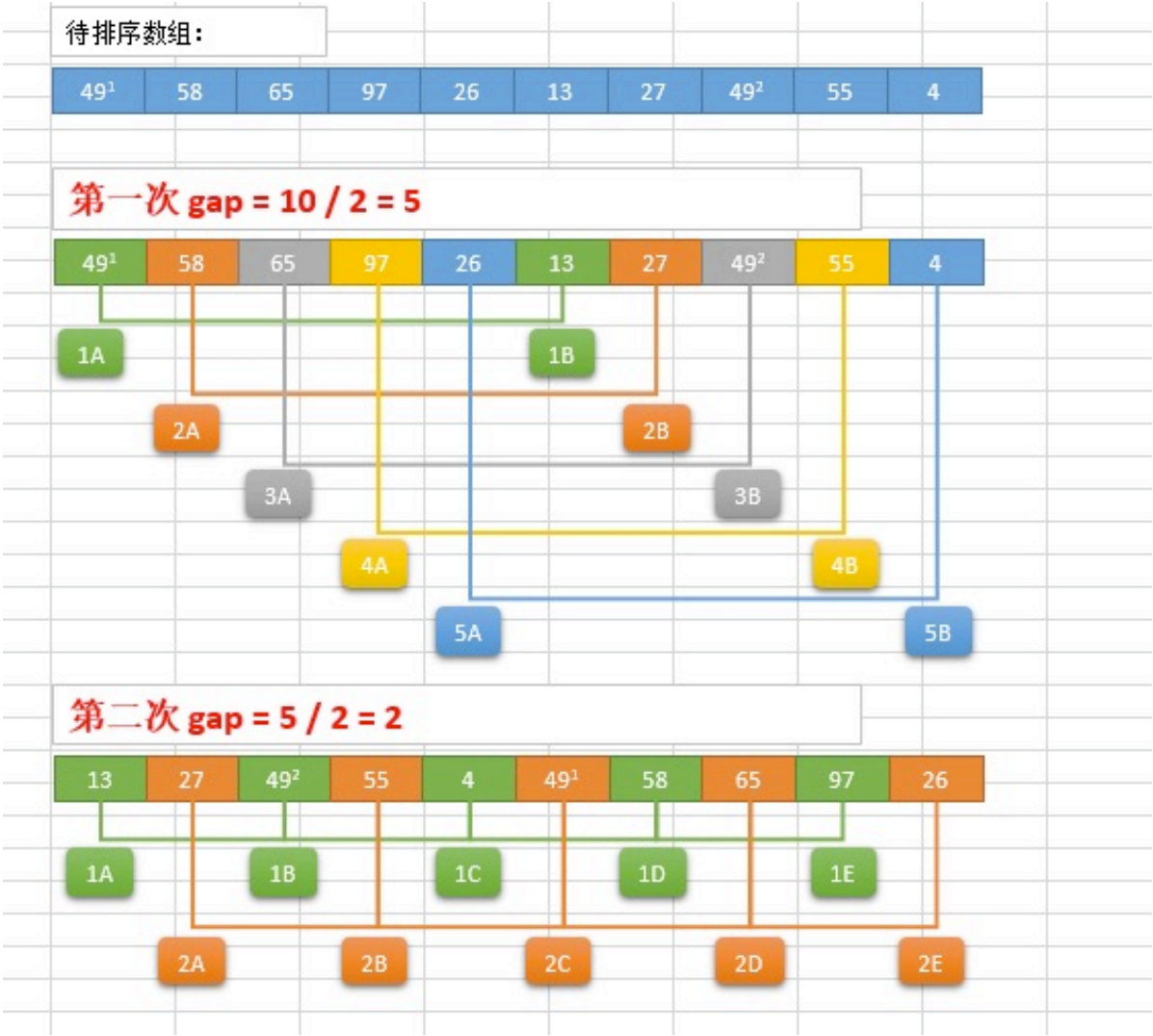
```
[17, 20, 26, 31, 44, 54, 55, 77, 93]
```

时间复杂度

- 最优时间复杂度：O(n) 升序排列，有序升序数列
- 最坏时间复杂度：O(n^2)
- 稳定性：稳定

希尔排序(Shell Sort)

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):
  "希尔排序"
  n = len(alist)
  gap = n // 2
  while gap > 0: #gap==1 插入排序
    #希尔和插入算法非常像，把i换为gap
    for j in range(gap, n):
      i = j
      while i>0:
        if alist[i] < alist[i - gap]:
          alist[i], alist[i-gap] = alist[i-gap], alist[i]
          i -= gap
        else:
          break
  #缩短gap步长
```

```
  gap //= 2  #/是精确除法，//是向下取整除法，%是求模

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

```

```
[17, 20, 26, 31, 44, 54, 55, 77, 93]
```

```
def insert_sort(alist):
  "插入排序"
  n = len(alist)
  # 从右边无序序列去多少个元素执行这样的过程
  for j in range(1, n):
  # j = [1,2,3...,n-1],
  # i 内层循环起始值
    i = j
    # 从第i个元素开始向前比较，如果小于前一个元素，交换位置：
    while i >0:
      if alist[i] < alist[i-1]:
        alist[i], alist[i-1]  = alist[i-1], alist[i]
        i -= 1
      else:
         break

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

```
[17, 20, 26, 31, 44, 54, 55, 77, 93]
```

时间复杂度

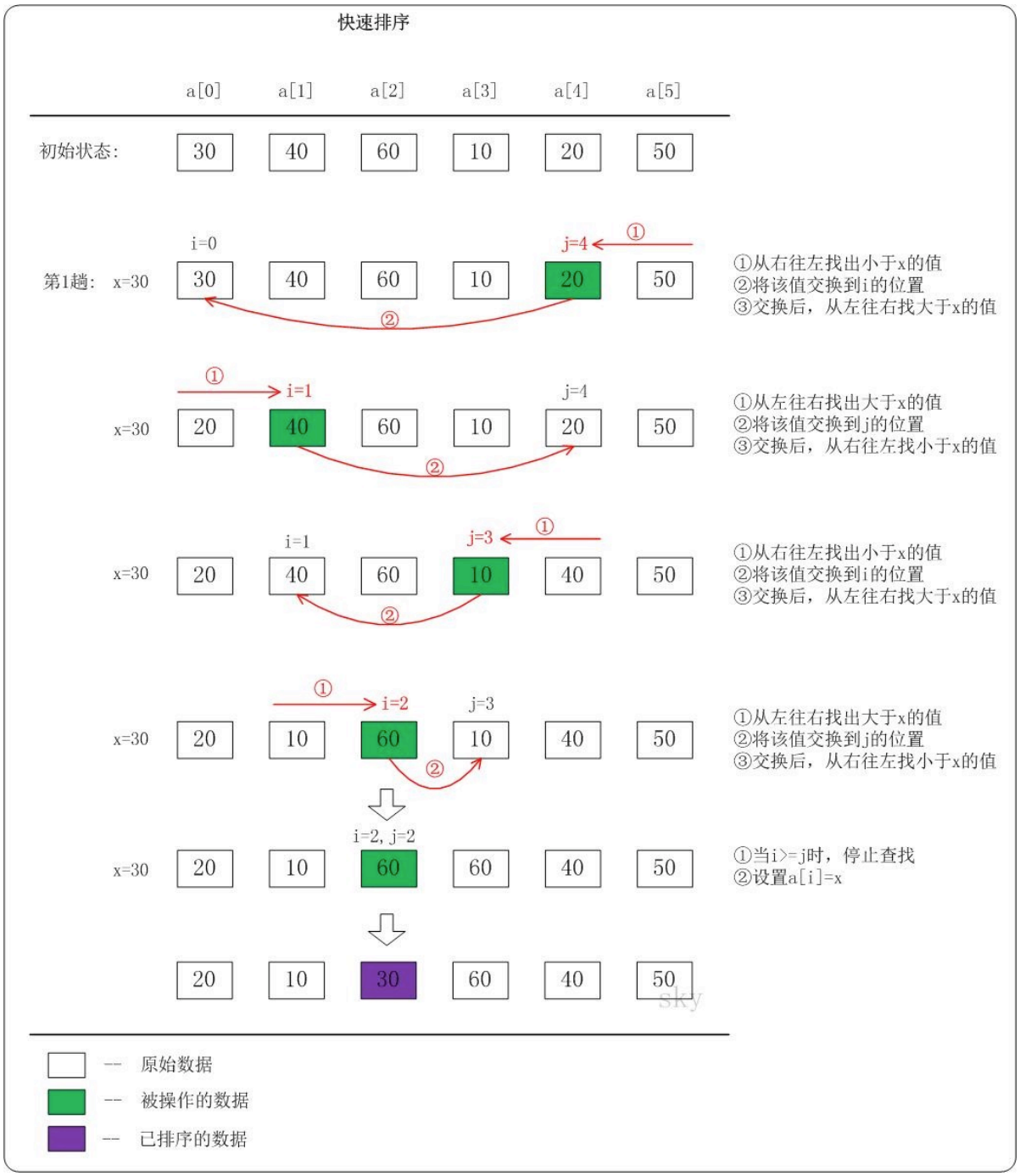
- 最优时间复杂度：跟序列的不同而不同而不同
- 最坏时间复杂度：O(n^2)，即gap为1，插入排序
- 稳定性：不稳定

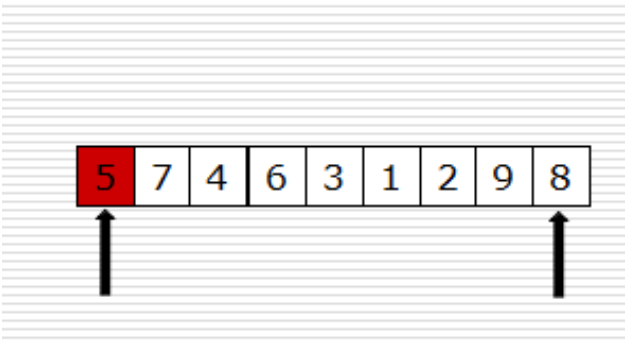
快速排序（Quicksort）必须掌握

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。


快速排序
a[0] a[1] a[2] a[3] a[4] a[5]
初始状态: 30 40 60 10 20 50
第1趟: x=30 i=0 j=4 ①
30 40 60 10 20 50
②
①从右往左找出小于x的值
②将该值交换到i的位置
③交换后，从左往右找大于x的值
① i=1 j=4
x=30 20 40 60 10 20 50
②
①从左往右找出大于x的值
②将该值交换到j的位置
③交换后，从右往左找小于x的值
i=1 j=3 ①
x=30 20 40 60 10 40 50
②
①从右往左找出小于x的值
②将该值交换到i的位置
③交换后，从左往右找大于x的值
① i=2 j=3
x=30 20 10 60 10 40 50
②
①从左往右找出大于x的值
②将该值交换到j的位置
③交换后，从右往左找小于x的值
i=2, j=2
x=30 20 10 60 60 40 50
①当i>=j时，停止查找
②设置a[i]=x
20 10 30 60 40 50
sky
— 原始数据
— 被操作的数据
— 已排序的数据


5 7 4 6 3 1 2 9 8

```python
def quick_sort(alist, start, end):
    """快速排序"""

    # 递归的退出条件
    if start >= end:
        return

    # 设定起始元素为要寻找位置的基准元素
    mid = alist[start]

    # low为序列左边的由左向右移动的游标
    low = start

    # high为序列右边的由右向左移动的游标
    high = end

    while low < high:
        # 如果low与high未重合，high指向的元素不比基准元素小，则high向左移动
        while low < high and alist[high] >= mid:
            high -= 1
        # 将high指向的元素放到low的位置上
        alist[low] = alist[high]

        # 如果low与high未重合，low指向的元素比基准元素小，则low向右移动
        while low < high and alist[low] < mid:
            low += 1
        # 将low指向的元素放到high的位置上
        alist[high] = alist[low]

    # 退出循环后，low与high重合，此时所指位置为基准元素的正确位置
    # 将基准元素放到该位置
    alist[low] = mid

    # 对基准元素左边的子序列进行快速排序
    quick_sort(alist, start, low-1)

    # 对基准元素右边的子序列进行快速排序
    quick_sort(alist, low+1, end)


alist = [54,26,93,17,77,31,44,55,20]
quick_sort(alist,0,len(alist)-1)
print(alist)
```

```
[17, 20, 26, 31, 44, 54, 55, 77, 93]
```

时间复杂度

- 最优时间复杂度：O($nlogn$)

  横向n，纵向logn($2^x = n, x = logn$)
- 最坏时间复杂度：O(n^2)

  横向n，纵向n
- 稳定性：不稳定

归并排序 merge sort

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。


```python
def merge_sort(alist):
    "归并排序"
    if len(alist) <= 1:
        return alist
    # 二分分解
    num = len(alist)//2
    left = merge_sort(alist[:num])
    right = merge_sort(alist[num:])
    # 合并
    return merge(left,right)
```

```python

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]:#加上=情况，就稳定了
            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 = merge_sort(alist)
print(sorted_alist)
```

```
[17, 20, 26, 31, 44, 54, 55, 77, 93]
```

时间复杂度

- 最优时间复杂度：O($nlogn$)

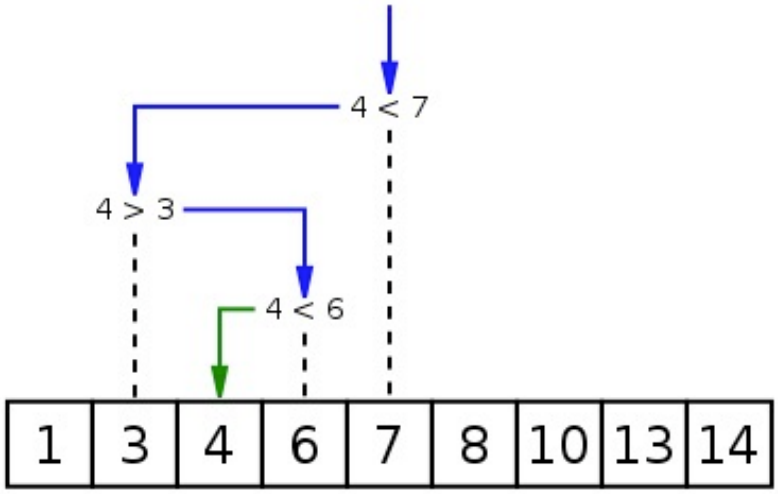
  横向logn($2^x = n, x = logn$)，纵向n
- 最坏时间复杂度：O($nlogn$)

  需要额外 空间
- 稳定性：稳定

搜索 二分法

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。

只能作用在有序的顺序表




```python
def binary_search(alist, item):
  "递归实现"
  n = len(alist)

  if n >0:
    mid =  n//2
    if alist[mid] == item:
      return True

    elif alist[mid]>item:
      return binary_search(alist[:mid], item)
    else:
      return binary_search(alist[mid+1:], item)
  return False

testlist = [0, 1, 2, 8, 13, 17, 19, 32, 42]
print(binary_search(testlist, 10))
print(binary_search(testlist, 42))

```

```
False
True
```

```python
def binary_search(alist, item):
  "非递归实现"
```

```
  n = len(alist)
  first = 0
  last = n-1

  while first<=last:
    mid =  (first+last)//2
    # 不能放在while外层
    if alist[mid] == item:
        return True
    elif alist[mid]>item:
        last = mid-1
    else:
        first = mid+1
  return False


testlist = [0, 1, 2, 8, 13, 17, 19, 32, 42]
print(binary_search(testlist, 10))
print(binary_search(testlist, 42))
```

```
False
True
```

```
def binary_search(alist, item):
      first = 0
      last = len(alist)-1
      while first<=last:
          midpoint = (first + last)//2
          if alist[midpoint] == item:
              return True
          elif item < alist[midpoint]:
              last = midpoint-1
          else:
              first = midpoint+1
      return False
testlist = [0, 1, 2, 8, 13, 17, 19, 32, 42,]
print(binary_search(testlist, 3))
print(binary_search(testlist, 13))
```

```
False
True
```

时间复杂度

- 最优时间复杂度：O(1)
- 最坏时间复杂度：O(logn)