一、 冒泡排序（Bubble Sort）

1. 原理图解

1）比较 54 与 26，54>26，进行互换

54	26	93	17	77	31

2）比较 54 与 26，54<93，不进行互换

26	54	93	17	77	31

3）比较 93 与 17，93>17，进行互换

26	54	93	17	77	31

4）第一次循环下来

26	54	17	77	31	93

5）进行第二次循环，直到排序完毕

26	54	17	77	31	93

- 注意，第二次循环只需要比较至[31]的位置，因为 93 在第一次循环已经被确定是最大的数。

2. Python 程序

1）算法实现

```python
def bubble_sort(a_list):
    for pass_num in range(len(a_list)-1,0,-1):
        for i in range(pass_num):
            if a_list[i] > a_list[i+1]:
                temp = a_list[i]
                a_list[i] =a_list[i+1]
                a_list[i+1] = temp
```

2）测试程序

```python
a_list = [54,26,93,17,77,31]
bubble_sort(a_list)
print(a_list)
```

3）测试结果

```
[17, 26, 31, 54, 77, 93]
```

3. 算法复杂度

循环	执行次数
1	n-1
2	n-2
3	n-3
…	…
n-1	1

总执行次数为 $\frac{1}{2}n^2 - \frac{1}{2}n$

算法复杂度为 $O(n^2)$

4. 算法改进

1) 思路：每次循环时，进行检查，该循环是否有交换发生。如果没有交换发生，说明排序已经完成，无须进行后续的循环。

2) 程序实现

```
def short_bubble_sort(a_list):
    exchanges = True
    pass_num = len(a_list)-1
    while pass_num > 0 and exchanges:
        # 如果此次循环未发生互换，则exchanges = False
        exchanges = False
        for i in range(pass_num):
            if a_list[i] > a_list[i+1]:
                exchanges = True
                temp = a_list[i]
                a_list[i] = a_list[i+1]
                a_list[i+1] = temp
        pass_num = pass_num - 1
    # 查看代码在第几次循环停止
    print(len(a_list)-1-pass_num)

a_list = [20,30,40,90,50,60,70,80,100,110]
short_bubble_sort(a_list)
print(a_list)
```

3) 程序结果

```
2
[20, 30, 40, 50, 60, 70, 80, 90, 100, 110]
```

二、 选择排序（Selection Sort）

1. 原理图解

1）第一次循环：找到最大的 93，将其移至队尾

54	26	93	17	77	31

2）第二次循环：找到最大的 77，将其移至队尾-1

54	26	17	77	31	93

3）以此类推，直到排序完毕

2. Python 程序

1）算法实现

```
def selection_sort(a_list):
    for fill_slot in range(len(a_list)-1, 0,-1):
        pos_of_max = 0
        for location in range(1,fill_slot+1):
            if a_list[location] > a_list[pos_of_max]:
                pos_of_max = location

        temp = a_list[fill_slot]
        a_list[fill_slot] = a_list[pos_of_max]
        a_list[pos_of_max] = temp
```

2）测试程序

```
a_list = [54,26,93,17,77,31]
selection_sort(a_list)
print(a_list)
```

3）测试结果

```
[17, 26, 31, 54, 77, 93]
```

3. 算法复杂度

选择算法复杂度仍为 O(n^2)，但是与冒泡法相比，少了交换的步骤。

三、 插入排序（Insert Sort）

1．原理图解

1）第一次循环：假设 54 是一个独立的子列表 SubList，里面只有一个元素 54

54	26	93	17	77	31

2）第二次循环：将 26 与 54 对比，发现 26<54，发生互换

26	54	93	17	77	31

3）第三次循环：将 93 与 54 对比，发现 93>54，不发生互换

26	54	93	17	77	31

4）以此类推，直至排序完毕

2．Python 实现

1）算法实现

```
def insertion_sort(a_list):
    for index in range(1,len(a_list)):
        current_value = a_list[index]
        position = index

        # 寻找插入的位置
        while position > 0 and a_list[position - 1] > current_value:
            a_list[position] = a_list[position - 1]
            position = position - 1

        a_list[position] = current_value
```

2）测试程序

```
a_list = [54,26,93,17,77,31]
insertion_sort(a_list)
print(a_list)
```

3）测试结果

```
[17, 26, 31, 54, 77, 93]
```

3．算法复杂度

算法复杂度仍为 O(n^2)，但是在最理想的情况下，只需要进行 N-1 次对比，N 为列表中元素的个数。同理，在一个排序性比较明显的原列表中，使用插入排序法更加快捷。

四、 希尔排序（Shell Sort）

1. 原理图解

1） 将一个列表分为若干个子列表。这里我们将原列表分为三个相等的子列表

54	26	93	17	77	41	44	55	20
54	26	93	17	77	41	44	55	20
54	26	93	17	77	41	44	55	20

2）使用插入排序法在子列表内进行排序

17	26	93	44	77	41	54	55	20
54	26	93	17	55	41	44	77	20
54	26	20	17	77	31	44	55	93

3）将子列表合并形成一个新列表

17	26	20	44	55	31	54	77	93

4） 继续使用插入法将新列表进行排序，直至排序完毕。

5） 有时候，数据较多，我们将数组分为 n 个小组，再将小组合成一个更大的小组进行插入排序，最终将更大的小组合成整个列表进行插入排序。

2. Python 实现

1）算法实现

```
def shell_sort(a_lsit):
    # 分为 n//2 个子列表，分别进行插入排序
    sublist_count = len(a_lsit) // 2
    while sublist_count > 0 :
        for start_positon in range(sublist_count):
            gap_insertion_sort(a_lsit,start_positon,sublist_count)
        print("After increments of size",sublist_count,"The list is",
a_lsit)
        # 将 n//个子列表组合成 n//4,n//8,n//16...1 个列表进行插入排序
        sublist_count = sublist_count // 2

def gap_insertion_sort(a_list, start, gap):
    for i in range(start+gap,len(a_list),gap):
        current_value = a_list[i]
        position = i
        while position >= gap and a_list[position-gap] >
current_value:
            a_list[position] = a_list[position-gap]
            position = position - gap
            a_list[position] = current_value
```

2）测试程序

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

3）测试结果

```
After increments of size 4 The list is [20, 26, 44, 17, 54, 31, 93, 55, 77]
After increments of size 2 The list is [20, 17, 44, 26, 54, 31, 77, 55, 93]
After increments of size 1 The list is [17, 20, 26, 31, 44, 54, 55, 77, 93]
[17, 20, 26, 31, 44, 54, 55, 77, 93]
```

3. 算法复杂度

直觉上来看希尔排序法似乎不优于插入排序法，因为希尔排序法频繁使用了插入排序法。但实际上，希尔排序法的每一次插入排序都比较简单，因为子列表比较短。另外，前面的步骤生成了一个排序性比较好的新列表，减少了插入排序的计算次数。这使得最后一步插入排序十分高效。

与前面的排序方法相比，Shell 的算法复杂度在 O(n)与 O(n^2)之间，与分组方式有关，数学计算不设讨论。

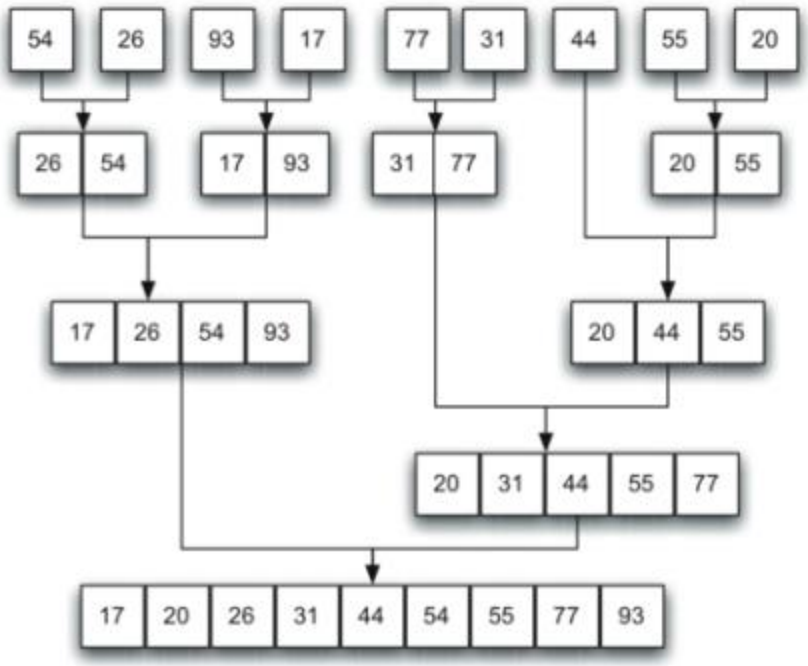
五、 归并排序（Merge Sort）

1. 原理图解

1）将一组列表[54,26,93,17,77,41,44,55,20]两等分为左列表[17,26,54,93]和右列表[77,41,44,55,20]，左列表两等分为左左列表[54,26]和左右列表[17,93]。同样让右列表两等分。

54	26	93	17	77	41	44	55	20
54	26	93	17	77	41	44	55	20
54	26	93	17	77	41	44	55	20

2）将 54 与 26 对比，93 与 17 对比，得到[26,54],[17,93]。然后在[26,54,17,93]内对比，顺序是这样子的：① 26 与 17 对比，发现 17 小，于是有[17] ② 26 与 93 比，发现 26 小，于是有[17,26] ③ 54 与 93 比，发现 54 小，有[17,26,54] ④ 剩下 93，有[17,26,54,93]。



3）以此类推，直至得出结果

2. Python 实现

1）算法实现

```python
def merge_sort(a_list):
    print("Splitting ",a_list)
    if len(a_list) > 1 :
        mid = len(a_list)//2
        left_half = a_list[:mid]
        right_half = a_list[mid:]

        # 递归
        merge_sort(left_half)
        merge_sort(right_half)

        i = 0
        j = 0
        k = 0

        # 组内排序
        while i < len(left_half) and j < len(right_half):
            if left_half[i] < right_half[j]:
                a_list[k] = left_half[i]
                i = i + 1
            else:
                a_list[k] = right_half[j]
                j = j + 1
            k = k + 1

        while i < len(left_half):
            a_list[k] = left_half[i]
            i = i + 1
            k = k + 1

        while j < len(right_half):
            a_list[k] = right_half[j]
            j = j + 1
            k = k + 1
    print("Merging ",a_list)
```

2）测试程序

```python
a_list = [54,26,93,17,77,31,44,55,20]
merge_sort(a_list)
print(a_list)
```

3）测试结果

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

3. 算法复杂度

归并排序的时间复杂度由拆分+合并两部分组成。拆分为二分法，时间复杂度为 O(log n)；合并每层代价为 n，共有 log n 层，的时间复杂度为 O(n log n)。相加的时间复杂度仍为 O(n log n)。

六、 快速排序（Quick Sort）

1. 原理图解

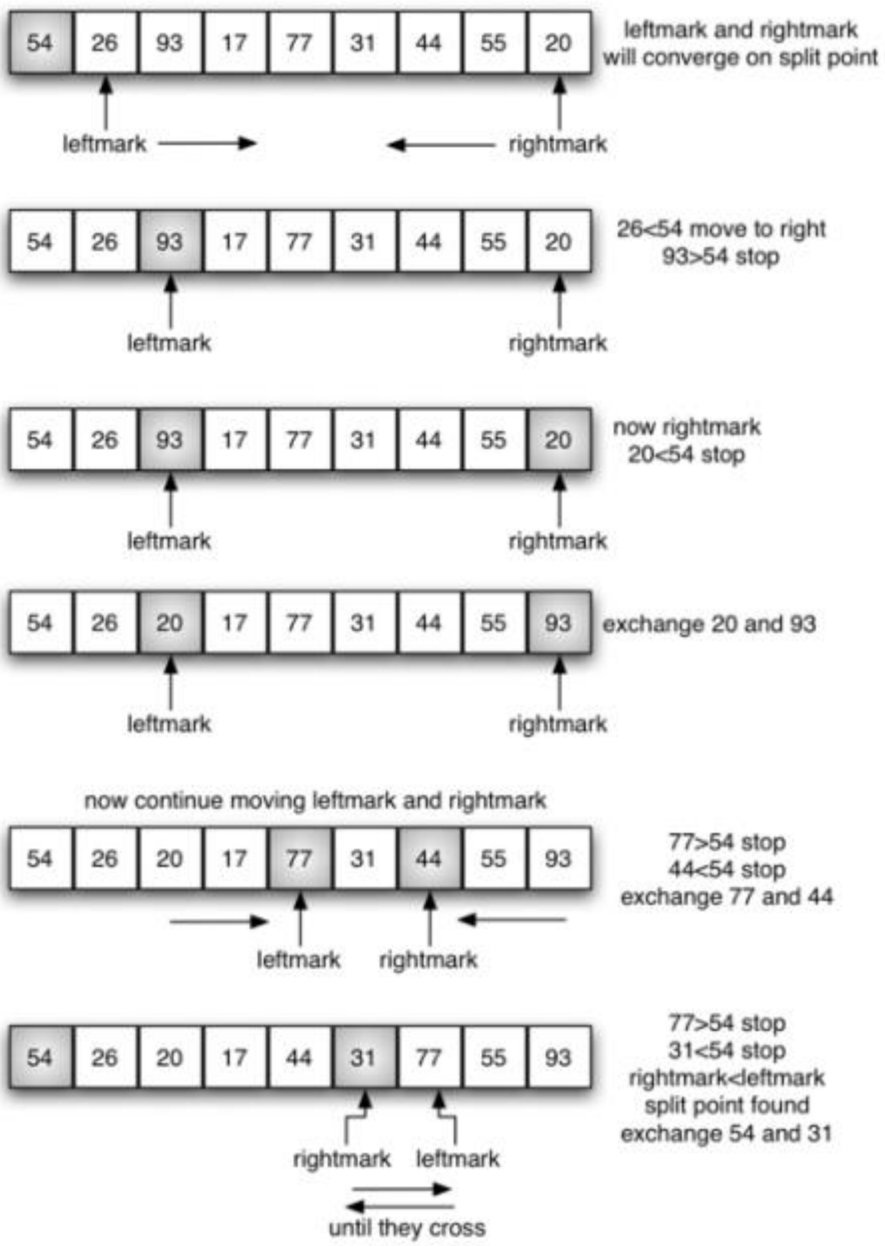
1）选取第一个元素 54 作为分离点，并使左边标记位置为 26，右边标记位置为 20。

① 左边标记，26<54,向前移动至 93，此时 93>54,暂停

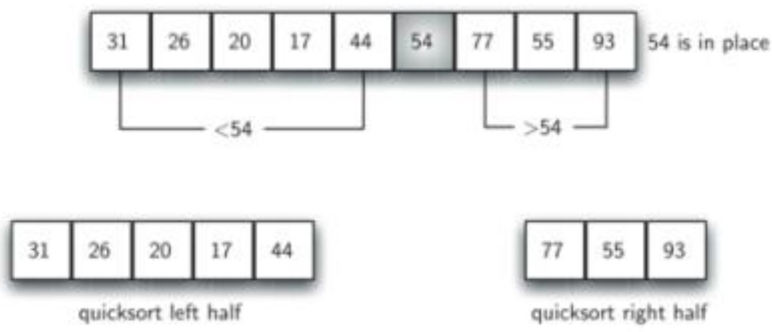
② 右边标记，20<54，暂停

③ 交换右边标记与右边标记的值

④ 以此类推，直到左边标记的位置>右边标记的位置



2）将 31 与 54 交换。对左右两边的子列表[31,26,20,17,44],[77,55,93]继续按照第 1 步的步骤进行快速排序。



2. Python 实现

1）算法实现

```
def quick_sort(a_lsit):
    quick_sort_helper(a_lsit,0,len(a_lsit)-1)

def quick_sort_helper(a_list,first,last):
    # 递归2 分、4 分、8 分...
    if first < last:
        split_point = partition(a_list,first,last)
        quick_sort_helper(a_list,first,split_point - 1)
        quick_sort_helper(a_list,split_point + 1, last)

def partition(a_list,first,last):
    pivot_value = a_list[first]
    left_mark = first + 1
    right_mark = last
    done = False
    while not done:
        # 寻找左边标记暂停点
        while left_mark <= right_mark and a_list[left_mark] <=
pivot_value:
            left_mark = left_mark + 1
        # 寻找右边标记暂停点
        while a_list[right_mark] >=pivot_value and right_mark >=
left_mark:
            right_mark = right_mark - 1

        # 右边标记超过左边标记，停止
        if right_mark < left_mark:
            done = True
        # 交换左右标记暂停点的值
        else:
```

```
            temp = a_list[left_mark]
            a_list[left_mark] = a_list[right_mark]
            a_list[right_mark] = temp

    # 将右边标记的值与分离点的值交换
    temp = a_list[first]
    a_list[first] = a_list[right_mark]
    a_list[right_mark] = temp

    return right_mark
```

2）测试程序

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

3）测试结果

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

3. 算法复杂度

一般来说，分离点的数值为列表中的中值，此时类似于二分法，一共需要共有 log n 层，每层代价为 n，时间复杂度为 O(n log n)。不过在最糟糕的情况下，分离点的数值处于列表的最左边或者最右边，此时时间复杂度为 O(n^2)。

七、总结

算法	期望时间复杂度
冒泡排序（Bubble Sort）	O(n^2)
选择排序（Selection Sort）	O(n^2)
插入排序（Insert Sort）	O(n^2)
希尔排序（Shell Sort）	O(n)~O(n^2)
归并排序（Merge Sort）	O(n log n)
快速排序（Quick Sort）	O(n log n)

*除了时间复杂度，有时候还要考虑空间复杂度，使用了二分法的排序方法是用了以空间换取时间的原理。