排序问题

3.1 排序概述

3.1.1 什么是排序

排序是计算机程序中的一种重要操作，排序就是将一组杂乱无章的数据按照一定的规律（升序或降序）组织起来，它的功能是将一个数据元素的任意序列，重新排列成一个按关键字有序的序列。排序算法是指一种能将一串记录序列按照某种特定的方式进行调整的方法。

如果是记录，则既可以按照记录的主关键字排序（主关键字唯一标识一条记录，如学生记录中的学号就是主关键字，学号不能重复，用来唯一标识一个学生），也可以按照记录的次关键字排序（如学生记录中的姓名、专业等都是次关键字，次关键字可以是重复的）。

试想一下，如何让计算机将 7,6,3,4,2,5,1 排成从小到大的序列？就算完全没有排序算法的概念，相信读者也能找到办法来解决这个问题。有些读者可能已经学过一些简单的排序方法，比如起泡排序、直接插入排序等，这些简单排序方法都可以很好地解决前面这种数据量比较小的排序问题。那么现在换个问题：要对 Google 的搜索关键词进行排序，选出十大热门关键词，该如何实现？对于这类数据量很大的问题，排序算法的效率就非常关键了。

3.1.2 排序的分类

由于待排序的记录数量不同，使得排序过程中涉及的存储器不同，根据排序过程中所有记录是否全部都存放在内存中，把排序方法分为内排序和外排序。

1. 内排序

内排序是指在排序的整个过程中，待排序的所有记录全部放在内存中；内排序的方法很多，但就全面性能而言，很难提出一种被认为最好的方法，每一种方法都有各自的优缺点，适合在不同的环境下使用。按照排序过程中依据的不同原则对内排序方法进行分类，大

致分为插入排序、交换排序、选择排序、归并排序4类。

(1) 插入排序。在一个已经有序的序列中,插入一个新的关键字,就好比军训排队,已经排好了一个纵队。这时,有人要临时加入到这个队列中,于是教官大声喊道:“新来的,迅速找到你的位置,入队!”于是新来的人“插入”到这个队伍的合适位置。这就是插入类的排序。属于这类排序的有直接插入排序、折半插入排序、希尔排序。

(2) 交换排序。交换排序的核心是交换,即每一趟排序,都通过一系列的交换动作,让一个关键字排到它最终的位置上。还是军训排队的例子,设想军训刚开始,一群学生要排队,教官说:“你比你旁边的高,你俩换一下。怎么换完还比下一个高?继续换……”最后这个人将被换到合适的位置。这就是交换排序。属于这类排序的有起泡排序(比高矮排队的例子)、快速排序。

(3) 选择排序。选择排序的核心是选择,即每一趟排序都选出一个最小(或最大)的关键字,把它和序列中第一个(或最后一个)关键字交换,直到最小(或最大)的关键字到位。继续以军训排队的例子说明。教官说:“你们都站着别动,我看谁个子最小。”然后教官选出个子最小的同学,说:“第一个位置是你的了,你和第一个同学换一下,剩下的同学我继续选。”这就是选择排序。属于这类排序的有简单选择排序、堆排序。

(4) 归并排序。所谓归并,就是将两个或两个以上的有序序列合并成一个新的有序序列,归并排序就是基于这种思想。继续以排队的例子说明。这次教官想了个特别的方法,他说:“你们每个人,先和旁边的人组成一个二人组,二人组内部先排好。”看到大家排好了,又说:“二人组和旁边的二人组继续组合成一个四人组,每个四人组内部排好队,动作要快!”这样不停排下去,最后全部学生都归并到了一个组中,同时也完成了排序,这就是归并排序。这个例子正是二路归并排序,特点是每次都把两个有序序列归并成一个新的有序序列。

2. 外排序

外排序是指待排序的记录数量很大,以至于内存一次不能容纳全部记录,整个排序过程需要在内外存之间多次交换数据才能得到排序的结果。本章集中讲解内排序的知识点。根据排序方法是否建立在关键字比较的基础上,排序方法分为以下几类。

(1) 基于比较:主要通过关键字之间的比较和记录的移动实现。

(2) 不基于比较:根据待排序数据的特点所采取的其他方法。

3.1.3 排序算法的性质与性能

1. 稳定性

所谓稳定性,是指当待排序序列中有两个或两个以上相同的关键字时,排序前和排序后这些关键字的相对位置,如果没有发生变化就是稳定的,否则是不稳定的。如果关键字 $K_i==K_j$,并且 $i<j$,则称关键字 K_i 在 K_j 之前。如果在排序之后,K_i 依然在 K_j 之前,则为稳定排序;反之为不稳定排序。例如,某序列中有两个关键字都是40,现在以40(a)和40(b)区分它们,用算法A对其进行排序,排序前40(a)在40(b)之前,如果排序后40(a)仍然在40(b)之前,则算法A是稳定的;如果能找到另外一种算法B,使排序后40(a)在40(b)之后,则算法B是不稳定的。

如果关键字不能重复,那么排序结果是唯一的。此时所选择的排序算法稳定与否就是

无关紧要的；如果关键字可以重复，则在选择排序算法时，应根据具体的需求来考虑选择稳定的还是不稳定的排序算法。

（1）稳定排序算法有起泡排序、插入排序、归并排序、基数排序。

（2）不稳定排序算法有快速排序、希尔排序、简单选择排序、堆排序。

2. 排序算法的性能评价

时间复杂度和空间复杂度是衡量一个排序算法好坏的主要标准，所以在写代码时，目标就是写出运行速度快、占用空间少的代码，这样的代码就是理想型代码。算法的时间复杂度表示常见的有常数阶 $O(1)$、对数阶 $O(\log n)$、线性阶 $O(n)$、线性对数阶 $O(n\log n)$、平方阶 $O(n^2)$、立方阶 $O(n^3)$……k 次方阶 $O(n^k)$、指数阶 $O(2^n)$ 和阶乘阶 $O(n!)$。常见的算法的时间复杂度之间的关系为：

$$O(1)<O(\log n)<O(n)<O(n\log n)<O(n^2)<O(2^n)<O(n!)<O(n^n)$$

排序算法的空间复杂度是对一个排序算法在运行过程中临时占用存储空间大小的一个量度（即除去原始序列大小的内存，在算法过程中用到的额外的存储空间）。在排序算法的时间复杂度分析中，通常会考虑排序算法在最好、最坏和平均情况下的时间复杂度。

3.2 插入排序

插入排序算法的思想是：每趟将一个元素按其关键字值的大小插入到它前面已排序的子序列中，如此重复，直到插入全部元素。

插入排序算法有直接插入排序和希尔排序。

3.2.1 直接插入排序

直接插入排序是将新的数据插入到已排好序的数列中，排序的基本方法是：每一步将一个待排序的元素，按其排序码的大小，插入到前面已经排好序的一组元素中的适当位置，直到元素全部插入为止。

1. 直接插入排序过程

直接插入排序就是从无序表中取出相应元素，把它插入到有序表的合适位置，使有序表仍然有序，其排序过程如下。

（1）第一趟比较前两个数，然后将第二个数按大小插入到有序表中。

（2）第二趟对第三个数据与前两个数从后向前扫描，把第三个数按大小插入到有序表中。

（3）如此进行下去，进行了 $n-1$ 趟扫描以后就完成了整个排序过程。

直接插入排序是由两层嵌套循环组成的。外层循环标识并决定待比较的数值。内层循环比较数值从而确定其最终位置。直接插入排序将待比较的数值与它的前一个数值进行比较，所以外层循环是从第二个数值开始的。在前一数值比待比较数值大的情况下继续循环比较，直到找到比待比较数值小的数值并将待比较数值置入其后的位置，结束该次循环。

【例 3-1】 数组数据为{50，70，30，20，10，70，40，60}，采用直接插入排序方法对其进行升序排列，其过程如表 3-1 所示。

表 3-1 直接插入排序过程

第一趟	50	70	30	20	10	70	40	60
第二趟	30	50	70	20	10	70	40	60
第三趟	20	30	50	70	10	70	40	60
第四趟	10	20	30	50	70	70	40	60
第五趟	10	20	30	50	70	70	40	60
第六趟	10	20	30	40	50	70	70	60
第七趟	10	20	30	40	50	60	70	70
第八趟	10	20	30	40	50	60	70	70

根据例 3-1 可以总结出直接插入排序的算法思想：每趟将一个待排序的关键字按照其值的大小插入到已经排好的部分有序序列的适当位置，直到所有待排关键字都被插入到有序序列中为止。

2. 直接插入排序算法代码

```
def insertSort(a, length):
    for i in range(1, length):
        if a[i-1] > a[i]:
        t = a[i]
        j = i-1
        while a[j] > t and j >= 0:
            a[j+1] = a[j]
            j = j-1
        a[j+1] = t
b = [ 70,50,30,20,10,70,40,60]
insertSort(b, len(b))
print(b)
```

运行结果：

```
[10,20,30,40,50,60,70,70]
```

3. 直接插入排序算法分析

衡量排序算法性能的重要指标是排序算法的时间复杂度和空间复杂度，排序算法的时间复杂度由算法执行中的元素比较次数和移动次数确定。

设数据序列有 n 个元素，直接插入排序算法执行 $n-1$ 趟，每趟的比较次数和移动次数与数据序列的初始排列有关。以下分 3 种情况分析直接插入排序算法的时间复杂度。

(1) 最好情况。一个排序的数据序列，如{1,2,3,4,5,6,}，每趟元素 a_i 与 a_{i-1} 比较 1 次，移动 2 次(keys[i]到 temp 再返回)，直接插入排序算法比较次数为 $n-1$，移动次数为 $2(n-1)$，时间复杂度为 $O(n)$。

(2) 最坏情况。一个反序排列的数据序列，如{6,5,4,3,2,1}，第 i 趟插入元素 a_i 比较 i 次，移动 $i+2$ 次。直接插入排序算法比较次数 C 和移动次数 M 的计算见式(3-1)和式(3-2)，时间复杂度为 $O(n^2)$。

$$C=\sum_{i=1}^{n-1} i=\frac{n(n-1)}{2}\approx\frac{n^2}{2} \tag{3-1}$$

$$M=\sum_{i=1}^{n-1}(i+2)=\frac{(n-1)\times(n+4)}{2}\approx\frac{n^2}{2} \tag{3-2}$$

（3）随机排列。一个随机排列的数据序列，第 i 趟插入元素 a_i，在等概率情况下，在子序列$\{a_0,a_1,\cdots,a_{i-1}\}$中查找 a_i 平均比较$(i+1)/2$次，插入 a_i 平均移动$i/2$次，直接插入排序算法比较次数 C 和移动次数 M 的计算见式(3-3)和式(3-4)，时间复杂度为 $O(n^2)$。

$$C=\sum_{i=1}^{n}\frac{i+1}{2}=\frac{1}{4}n^2+\frac{3}{4}n+1\approx\frac{n^2}{4} \tag{3-3}$$

$$M=\sum_{i=1}^{n}\frac{i}{2}=\frac{n\times(n+1)}{2}\approx\frac{n^2}{4} \tag{3-4}$$

总之，直接插入排序算法的时间效率为 $O(n)\sim O(n^2)$，数据序列的初始排列越接近有序，直接插入排序的时间效率越高。

直接插入排序算法中的 temp 占用一个存储单元，空间复杂度为 $O(1)$。直接插入排序算法是一种稳定的排序算法，具有以下特点：

（1）直接插入排序算法简单、容易实现，适用于待排序记录基本有序或待排序记录个数较少时。

（2）当待排序的记录个数较多时，大量的比较和移动操作使直接插入排序算法的效率降低。

3.2.2 希尔排序

希尔排序是 D. L. Shell 在 1959 年提出的，又称为缩小增量排序，其基本思想是分组的直接插入排序。由直接插入排序算法分析可知，若数据序列接近有序，则时间效率越高；再者，当 n 较小时，时间效率也较高。

1. 希尔排序算法介绍

希尔排序的本质也是插入排序，只不过是将待排序列按某种规则分成几个子序列，分别对这几个子序列进行直接插入排序。这个规则的差别在于增量的选取，如果增量为 1，就是直接插入排序。例如，先以增量 5 来分割序列，即将下标为 0、5、10、15、……的关键字分成一组，将下标为 1、6、11、16、……的关键字分成另一组等，然后分别对这些组进行直接插入排序，这就是一趟希尔排序。将上面排好序的整个序列，再以增量 2 分割，即将下标为 0、2、4、6、……的关键字分成一组，将下标为 1、3、5、7、9、……的关键字分成另一组等，然后分别对这些组进行直接插入排序，就完成了一趟希尔排序。最后以增量 1 分割整个序列，其实就是对整个序列进行一趟直接插入排序，从而完成希尔排序。

希尔排序的基本思想是：将整个待排序记录分割成若干个子序列，在子序列内分别进行直接插入排序，待整个序列中的记录基本有序时，对全体记录进行直接插入排序。

【例 3-2】 以序列{10，50，30，20，70，70，40，60}为例，该序列的希尔排序过程如表 3-2 所示。

表 3-2 希尔排序过程

第一趟	10	50	30	20	70	70	40	60
第二趟	10	20	30	50	40	60	70	70
第三趟	10	20	30	40	50	60	70	70
第四趟	10	20	30	40	50	60	70	70

2. 希尔排序算法代码

```
a = [10,50,30,20,70,70,40,60]
b = len(a)
gap = b // 2
while gap >= 1:
    for i in range (b):
        j = i
        while j >= gap and a[j - gap] > a[j]:
a[j],a[j - gap] = a[j - gap],a[j]
            j -= gap
gap = gap//2
print(a)
```

运行结果：

```
[10,20,30,40,50,60,70,70]
```

3. 希尔排序算法分析

1）时间复杂度

在最坏情况下，每两个数都要比较并交换一次，则最坏情况下的时间复杂度为 $O(n^2)$；在最好情况下，数组是有序的，不需要交换，只需要比较，则最好情况下的时间复杂度为 $O(n)$。经大量研究发现，希尔排序的平均时间复杂度为 $O(n^{1.3})$。

2）空间复杂度

因为只需要一个变量用于两数的交换，与 n 的大小无关，所以希尔排序的空间复杂度为 $O(1)$。

希尔排序算法在比较过程中，会错过关键字相等元素的比较。因此，希尔排序算法是不稳定的。

3.3 交换排序

交换排序的主要操作是交换，在待排序列中选两个记录，对它们的关键字进行比较，如果反序（即排列顺序与排序后的次序正好相反），则交换它们的存储位置。

3.3.1 起泡排序

起泡排序是交换排序中最简单的排序方式，其基本思想是：两两比较相邻元素的关键字，如果反序则交换，直到没有反序的元素时为止。

1. 起泡排序过程

（1）从第一个元素开始逐个比较相邻的元素。如果第一个比第二个大（$a[1]>a[2]$），就交换它们两个。

（2）对每一对相邻元素做同样的工作，从开始的第一对到结尾的最后一对。此时在这一点，最后的元素应该是最大的数，我们将一遍这样的操作称为“一趟起泡排序”。

（3）针对所有元素重复以上的步骤，每一趟起泡排序的最大值已放在最后，下一次操作则不需要将此最大值纳入计算过程。

（4）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，即

完成起泡排序。

【例 3-3】 以序列{70,50,30,20,10,70,40,60}为例，该序列的起泡排序过程如表 3-3 所示。

表 3-3 起泡排序过程

第一趟	70	50	30	20	10	70	40	60
第二趟	50	30	20	10	70	40	60	70
第三趟	30	20	10	50	40	60	70	70
第四趟	20	10	30	40	50	60	70	70
第五趟	10	20	30	40	50	60	70	70
第六趟	10	20	30	40	50	60	70	70
第七趟	10	20	30	40	50	60	70	70

2. 起泡排序算法代码

```
def bubbleSort(arr):
    n = len(arr)
    for i in range(n):
        # Last i elements are already in place
        for j in range(0, n - i - 1):
            if arr[j] > arr[j + 1] :
                arr[j], arr[j + 1] = arr[j + 1], arr[j]
arr = [64, 34, 25, 12, 22, 11, 90]
bubbleSort(arr)
print ("排序后的数组:")
for i in range(len(arr)):
print ("%d" %arr[i]),
```

运行结果如下：

```
排序后的数组:
11
12
22
25
34
64
90
```

3. 起泡排序算法分析

（1）最好情况。数据序列排序，只需一趟扫描，比较 n 次，没有数据移动，时间复杂度为 $O(n)$。

（2）最坏情况。数据序列随机排列和反序排列，需要 $n-1$ 趟扫描，比较次数和移动次数都是 $O(n^2)$，时间复杂度为 $O(n^2)$。

总之，数据序列越接近有序，起泡排序算法的时间效率越高，为 $O(n)\sim O(n^2)$。起泡排序需要一个辅助空间用于交换两个元素，空间复杂度为 $O(1)$。起泡排序算法是稳定的。其实起泡排序是八大排序算法中最简单及基础的排序，这个算法的名字由来是因为越大的元素会经由交换慢慢“浮”到数列的顶端（升序或降序排列），就如同碳酸饮料中二氧化碳的气泡最终会上浮到顶端一样，故得名“起泡排序”。

3.3.2 快速排序

快速排序是对起泡排序的一种改进。在起泡排序中，元素的比较和移动是在相邻单元中进行的，元素每次交换只能上移或下移一个单元，因而总的比较次数和移动次数较多。

快速排序的基本思想是：首先选一个基准值，通过一趟排序将待排序元素分割成独立的两部分，前一部分元素的关键字均小于或等于基准值，后一部分记录的关键字均大于或等于基准值，然后分别对这两部分重复上述方法，直到整个序列有序。

1. 快速排序过程

首先在数组中选择一个基准点，然后分别从数组的两端扫描数组，设两个指示标志(low 指向起始位置，high 指向末尾)，首先从后半部分开始，若发现有元素比该基准点的值小，则交换 low 和 high 位置的值；然后从前半部分开始扫描，若发现有元素大于基准点的值，则交换 low 和 high 位置的值，如此往复循环，直到 low＞＝high，然后把基准点的值放到 high 这个位置。一次排序就完成了。可以看出，在第四趟时已经达到顺序排列，但还是会继续计算几趟直到完成全部运算。

【例 3-4】 以序列{70，50，30，20，10，70，40，60}为例，该序列的快速排序过程如表 3-4 所示。

表 3-4 快速排序过程

第一趟	70	50	30	20	10	70	40	60
第二趟	60	50	30	20	10	40	70	70
第三趟	40	50	30	20	10	60	70	70
第四趟	10	20	30	40	50	60	70	70
第五趟	10	20	30	40	50	60	70	70

2. 快速排序算法代码

```
def partition(arr,low,high):
    i = ( low-1 ) # 最小元素索引
    pivot = arr[high]
    for j in range(low ,high):
        if arr[j] <= pivot:
            i = i+1
            arr[i],arr[j] = arr[j],arr[i]
    arr[i+1],arr[high] = arr[high],arr[i+1]
    return ( i+1 )
def quickSort(arr,low,high):
    if low < high:
        pi = partition(arr,low,high)
        quickSort(arr,low, pi-1)
        quickSort(arr, pi+1, high)
arr = [10,7, 8, 9, 1, 5]
n = len(arr)
quickSort(arr,0,n-1)
print ("排序后的数组:")
for i in range(n):
print ("%d" %arr[i]),
```

运行结果如下：

```
排序后的数组:
1
5
7
8
9
10
```

3. 快速排序算法分析

快速排序的执行时间与数据序列的初始排列及基准值的选取有关。

(1) 在最好情况下，每趟排序将序列分成长度相近的两个子序列，时间复杂度为 $O(n\times\log_2 n)$。

(2) 在最坏情况下，每趟将序列分成长度差异很大的两个子序列，时间复杂度为 $O(n^2)$。

例如，设一个排序数据序列有 n 个元素，若选取序列的第一个值作为基准值，则第一趟得到的两个子序列长度分别为 0 和 $n-1$，比较次数为

$$C=\sum_{i=1}^{n-1}(n-i)=\frac{n\times(n-1)}{2}\approx\frac{n^2}{2} \tag{3-5}$$

快速排序选择基准值还有其他多种方法，如可以选取序列的中间值等。由于序列的初始排列是随机的，所以不管如何选择基准值，总会存在最坏情况。此外，快速排序还要在执行递归函数的过程中花费一定的时间和空间，使用栈保存参数，栈所占用的空间与递归调用的次数有关，空间复杂度为 $O(\log_2 n)\sim O(n)$。

总之，当 n 较大且数据序列随机排列时，快速排序是“快速”的；当 n 很小或基准值选取不合适时，快速排序则较慢。快速排序算法是不稳定的。

3.4 选择排序

每趟排序在当前待排序序列中选出关键字最小的元素，添加到有序序列中。选择排序的特点是元素移动的次数较少。选择排序算法分为两种：简单选择排序和堆排序。

3.4.1 简单选择排序

简单选择排序的基本思想是：第 i 趟($1\leqslant i\leqslant n-1$)排序在待排序序列 $r[i]\sim r[n]$中选取最小元素，并和第 i 个元素交换。

1. 简单选择排序过程

(1) 设置两个元素 i 和 j，i 自数组第一个元素开始，j 自第 $i+1$ 个元素开始。

(2) 接着 j 遍历整个数组，选出整个数组最小的值，并让这个最小的值和 i 的位置交换(如果 i 选择的元素是最小的，则不需要交换)，这个过程称为一趟选择排序。

(3) i 选中下一个元素($i++$)，重复进行每一趟选择排序。

(4) 重复上述步骤，使得 i 到达 $n-1$ 处，即完成排序。

【例 3-5】 以序列{2,10,9,4,8,1,6,5}为例，该序列的简单选择排序过程如表 3-5 所示。

表 3-5 简单选择排序过程

开始数据	2	10	9	4	8	1	6	5
第一趟	1	10	9	4	8	2	6	5
第二趟	1	2	9	4	8	10	6	5
第三趟	1	2	4	9	8	10	6	5
第四趟	1	2	4	5	8	10	6	9
第五趟	1	2	4	5	6	10	8	9
第六趟	1	2	4	5	6	8	10	9
第七趟	1	2	4	5	6	8	9	10

2. 简单选择排序算法代码

```
def select_sort(origin_items, comp = lambda x, y: x < y ):
    '''简单选择排序'''
    items = origin_items[:]
    for i in range(len(items) - 1):
        min_index = i
        for j in range(i + 1, len(items)):
            if comp(items[j], items[min_index]):
                min_index = j
        items[i], items[min_index] = items[min_index], items[i]
    return items
print(select_sort([9,3,5,2,1,10,24,30]))
```

运行结果如下：

```
[1,2,3,5,9,10,24,30]
```

3. 选择排序算法分析

简单选择排序的比较次数与数据序列的初始排列无关，第 i 趟排序的比较次数是 $n-i$；移动次数与初始排列有关，正序排列的数据排序序列移动 0 次；反序排列的数据序列，每趟排序都要交换，移动 $3(n-1)$次。因此其最坏情况下的时间复杂度为 $O(n^2)$。简单选择排序的空间复杂度为 $O(1)$。简单选择排序算法是不稳定的。

3.4.2 堆排序

堆排序是由 1991 年计算机先驱奖获得者、斯坦福大学计算机科学系教授罗伯特·弗洛伊德和威廉姆斯在 1964 年共同提出的。

1. 什么是堆

堆是一种非线性的数据结构，可以把堆看作一个数组。堆是一种特殊的二叉树，也可以被看作一个完全二叉树。通俗来讲，堆其实就是利用完全二叉树的结构维护的一维数组，每个子节点的值总是小于(或者大于)它的父节点。相应地，按照堆的特点可以把堆分为大顶堆和小顶堆。

(1) 大顶堆：每个节点的值都大于或等于其左右孩子节点的值。

(2) 小顶堆：每个节点的值都小于或等于其左右孩子节点的值。

这种特性与二叉排序树很相似。这种特殊的数据结构便于快速访问到需要的值，如优先队列就使用堆进行处理。

堆排序是指利用这种数据结构设计的一种排序算法。

2. 堆排序的基本思想

把待排序的元素按照大小在二叉树位置上排列(使用数组模拟,不一定要使用二叉树),排序好的元素要满足：父节点的元素要大于或等于其子节点,这个过程叫作堆化过程。如果根节点存放的是最大的数,则叫作大顶堆。如果根节点存放的是最小的数,叫作小顶堆。根据这个特性(大顶堆根最大,小顶堆根最小),可以把根节点拿出来,然后进行堆化；再把根节点拿出来,一直循环到最后一个节点,就排好序了。

3. 堆排序过程

(1) 将待排序序列构造成一个大顶堆。

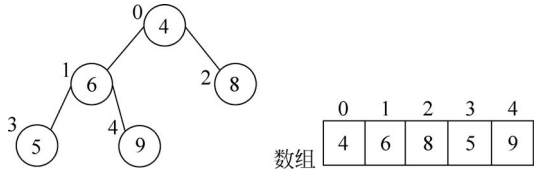
(2) 此时整个序列的最大值就是顶堆的根节点。

(3) 将其与末尾元素进行交换,此时末尾就为最大值。

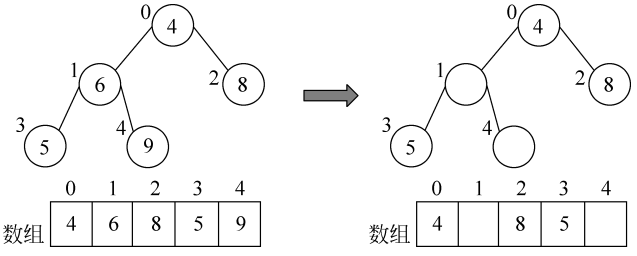
(4) 然后将剩余 $n-1$ 个元素重新构造成一个堆,这样会得到 n 个元素的次小值。如此反复执行,便能得到一个有序序列了。

以数组{4,6,8,5,9}为例,堆排序中的大顶堆排序过程如下：

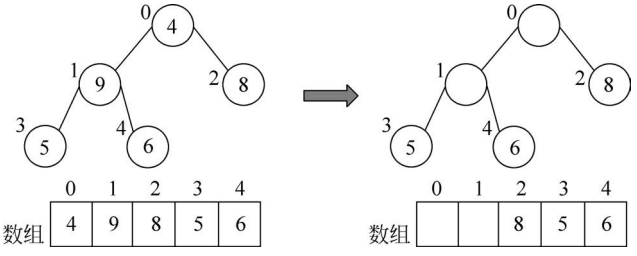
(1) 假设给定无序序列数组结构为



(2) 此时从最后一个非叶子节点开始(叶子节点自然不用调整,第一个非叶子节点 arr.length/2−1=5/2−1=1,也就是下面的节点),从左至右、从下至上进行调整,观察 6 的两个子节点,从右至左,9 大于 6 就和 6 互换。



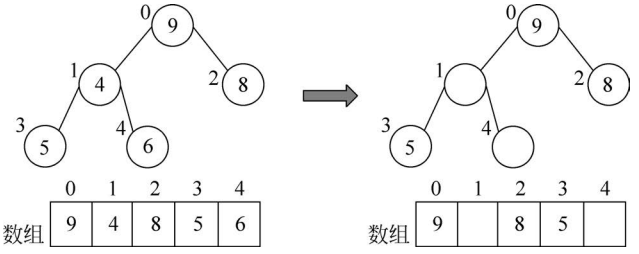
(3) 找到第二个非叶子节点 4,由于{4,9,8}中 9 元素最大,所以 4 和 9 交换。



(4) 此时,交换导致了子根{4,5,6}结构混乱,继续调整,{4,5,6}中 6 最大,交换 4 和 6。

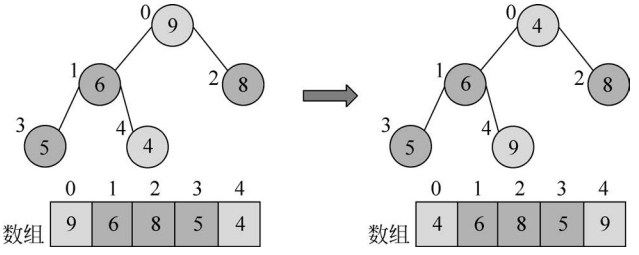
(5) 此时就将一个无序数组构造成了一个大顶堆。

以数组{4,6,8,5,9}为例,堆排序中的小顶堆排序过程如下：

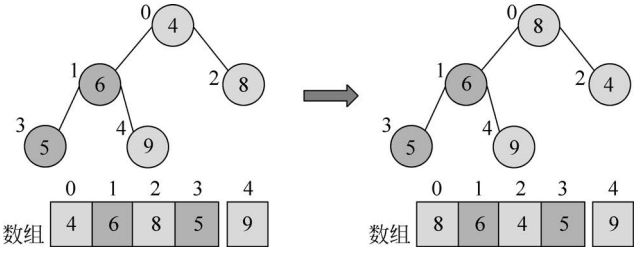


将堆顶元素与末尾元素进行交换，使末尾元素最大，然后继续调整堆，再将堆顶元素与末尾元素交换，得到第二大元素，如此反复进行交换—重建—交换的过程。

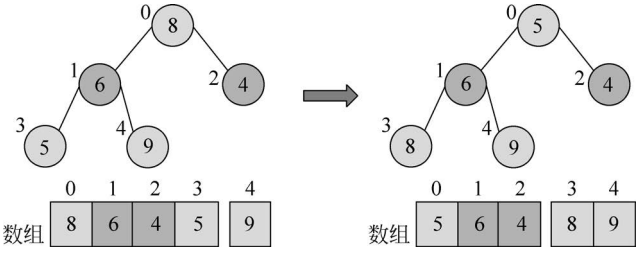
（1）将堆顶元素 9 和末尾元素 4 进行交换。



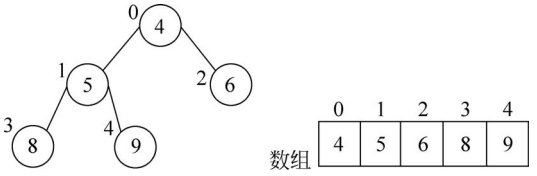
（2）重新调整结构，使其继续满足堆定义。



（3）再将堆顶元素 8 与末尾元素 5 进行交换，得到第二大元素 8。



（4）继续进行调整、交换，如此反复进行，最终使得整个序列有序。



（5）此时就将一个大顶堆构造成了一个小顶堆。

4. 堆排序算法代码

```
def heapify(arr, n, i):
    largest = i
    l = 2 * i + 1                                    # left = 2*i + 1
    r = 2 * i + 2                                    # right = 2*i + 2
    if l < n and arr[i] < arr[l]:
        largest = l
    if r < n and arr[largest] < arr[r]:
        largest = r
    if largest != i:
        arr[i],arr[largest] = arr[largest],arr[i]    # 交换
        heapify(arr, n, largest)
def heapSort(arr):
    n = len(arr)
    # Build a maxheap.
    for i in range(n, -1, -1):
        heapify(arr, n, i)
    # 一个个交换元素
    for i in range(n-1, 0, -1):
        arr[i], arr[0] = arr[0], arr[i]              # 交换
arr = [ 12, 11, 13, 5, 6, 7]
heapSort(arr)
n = len(arr)
print ("排序后")
for i in range(n):
print ("%d" %arr[i]),
```

运行结果如下：

```
排序后 5 6 7 11 12 13
```

5. 堆排序算法分析

堆排序的运行时间主要是消耗在构建堆和在重建堆时的反复筛选上。在构建堆的过程中，因为是从完全二叉树最下层的非叶子节点开始构建的，将它与其孩子节点进行比较和必要的互换，对于每个非叶子节点来说，其实最多会进行 2 次比较和互换，故初始化堆的时间复杂度为 $O(n)$。在正式排序时，第 i 次取堆顶记录和重建堆需要 $O(\log i)$ 的时间（完全二叉树的某个节点到根节点的距离为 $\log(2i+1)$），并且需要取 $n-1$ 次堆顶记录，因此重建堆的时间复杂度为 $O(n\log n)$。所以总的来说，堆排序的时间复杂度为 $O(n\log n)$。由于堆排序对元素的排序状态不敏感，因此堆排序的最好时间复杂度、最坏时间复杂度和平均时间复杂度均为 $O(n\log n)$。

堆排序的空间复杂度为 $O(1)$。堆排序算法是不稳定的。

3.5 归并排序

归并排序的主要思想是将若干有序序列逐步归并，最终归并为一个有序序列。二路归并排序是归并排序中最简单的排序方法。

二路归并排序的基本思想是：将一个具有 n 个待排序记录的序列看成是 n 个长度为 1 的有序序列，然后进行两两归并，得到 $n/2$ 个长度为 2 的有序序列，再进行两两归并，得到

$n/4$ 个长度为 4 的有序序列……直至得到一个长度为 n 的有序序列。

1. 归并排序过程

归并排序的核心思想是将两个有序的数列合并成一个大的有序的序列。通过递归，层层合并，即为归并，归并排序的算法效率仅次于快速排序，是一种稳定的算法。归并排序需要建立两倍的数组空间，一般适用于对总体而言无序，但是各子项又相对有序（并不是完全乱序）的情况。

（1）申请空间，使其大小为两个已经排序序列之和，该空间用来存放合并后的序列。

（2）设定两个指针，最初位置分别为两个已经排序序列的起始位置。

（3）比较两个指针所指向的元素，选择相对小的元素放入到合并空间，并将指针移动到下一位置。

（4）重复步骤（3），直到某一指针超出序列尾，将另一序列剩下的所有元素直接复制到合并序列尾。

【例 3-6】 以序列{70,50,30,20,10,70,40,60}为例，该序列的归并排序过程如表 3-6 所示。

表 3-6 归并排序过程

第一趟	50	70	20	30	10	70	40	60
第二趟	20	30	50	70	10	40	60	70
第三趟	10	20	30	40	50	60	70	70
第四趟	10	20	30	40	50	60	70	70

2. 归并排序算法代码

```
def merge_sort(alist):
    """归并排序算法"""
    n = len(alist)
    if n <= 1:
        return alist
    # 二分分解
    num = len(alist) // 2
    # left 采用归并排序后形成的有序的新列表
    left_li = merge_sort(alist[:num])
    # right 采用归并排序后形成的有序的新列表
    right_li = merge_sort(alist[num:])
    # 将两个有序的子序列合并成一个有序的整体
    left_pointer, right_pointer = 0, 0              #定义两个指针
    result = []                                     # 定义一个空列表用于存放有序序列
    while left_pointer < len(left_li) and right_pointer < len(right_li):
        if left_li[left_pointer] < right_li[right_pointer]:
            result.append(left_li[left_pointer])
            left_pointer += 1
        else:
            result.append(right_li[right_pointer])
            right_pointer += 1
    # 如果左右数组长度不对称，则直接将剩余的元素添加到 result 中
    result += left_li[left_pointer:]
    result += right_li[right_pointer:]
    return result
if __name__ == "__main__":
```

```
    alist = [54, 26, 93, 17, 77, 31, 44, 55, 20, 99]
    sorted_alist = merge_sort(alist)
print(sorted_alist)
```

运行结果如下：

```
[17,20,26,31,44,54,55,77,93,99]
```

3. 归并排序算法分析

n 个元素归并排序，每趟比较 $n-1$ 次，数据移动 $n-1$ 次，进行 $\log_2 n$ 趟排序，时间复杂度为 $O(n\log_2 n)$。归并排序需要 $O(n)$容量的附加空间，与数据序列的存储容量相同，空间复杂度为 $O(n)$。归并排序算法是稳定的。归并排序虽然看上去是稳定的而且时间复杂度不高，但在实际应用中，开辟大块的额外空间并且将两个数组的元素来回复制是比较耗时的，所以归并排序一般不用于内部排序。

3.6 排序算法的比较

本章介绍了 7 种不同的排序算法，其中插入排序包含直接插入排序、希尔排序，交换排序包含起泡排序、快速排序，选择排序包含简单选择排序、堆排序和归并排序，它们在时间复杂度、空间复杂度和稳定性上各有优势。并不存在绝对意义上的最优排序方法，应从时间复杂度、空间复杂度和是否稳定上比较这几种排序方法。

具有 $O(n^2)$时间复杂度的是简单选择排序、直接插入排序和起泡排序这 3 种排序算法。当元素规模 n 较小或基本有序时，它们是较好的排序方法。同时，由于相邻的两个元素总是进行比较，因此在比较两个关键字相等的元素时可以确定两者的相对位置，从而保证排序后它不会发生相对位置的变化，因此在理论上，这些时间复杂度为 $O(n^2)$的排序都是稳定的，然而简单选择排序在进行最小元素和第一个位置的元素的交换时，改变了被交换元素和其他元素的相对位置，因此简单选择排序是不稳定的，而直接插入排序和起泡排序是稳定的。

希尔排序是最早从 $O(n^2)$时间复杂度中提升的排序方法之一，它使用一个增量序列进行多次的规模逐渐变大的排序。在对规模较小序列的排序时使用直接插入排序将序列基本有序化，这样一来，在对规模较大序列的排序时就避免了过多的比较和交换，从而将时间复杂度减少到 $O(nd)$，其中 d 的取值同增量序列和排序对象的具体情况有关，在最差的情况下接近 2，即时间复杂度接近直接插入排序。由于希尔排序无法保证总是将相邻的两个元素进行比较，可能出现一个元素在排序过程中“跳跃”到和它等值且初始位置在前的另一个元素之前，因此，希尔排序是不稳定的。

在时间效率上表现较好的是快速排序、堆排序和归并排序 3 种排序算法，它们都使用分而治之的方法，将原序列分成两个部分，在排序过程中，这两个部分之间只进行复杂度为 $O(n)$的划分或归并操作，其他的比较或交换操作各自集中在两个部分内部，因此大大减少了比较或交换的次数。例如，堆排序在堆顶元素输出以后需要寻找下一个堆顶元素，在寻找的过程中不断地将问题规模减小，直到跳出循环；快速排序在寻找到基准后，序列被划分为两个部分，两个部分在内部各自进行比较交换，两个部分之间并没有进行比较。同样地，归并排序始终将规模减半再进行排序，在规模为 N 时再进行复杂度为 $O(n)$的归并操作。这

3种排序均实现了$O(n\log_2 n)$的时间复杂度。具体到实际的平均时间效率上，快速排序无疑是最佳的排序方法。

然而，在最坏情况下，快速排序的时间效率不如堆排序和归并排序，可能导致$O(n^2)$的最差结果。此外，快速排序需要$O(\log_2 n)$深度的栈空间，归并排序也需要$O(n)$的额外空间。堆排序在空间复杂度上表现出色，仅需要常数个额外空间。

在稳定性上，归并排序是稳定的，而堆排序和快速排序是不稳定的。

因此，每一种排序都有其自身优点，适用于不同的情况。应该根据具体的条件，选择相应的排序方法，甚至将两种以上的排序方法结合使用，排序算法性能比较如表3-7所示。

表3-7 排序算法性能比较

算法思路	排序算法	时间复杂度	最好情况	最坏情况	空间复杂度	稳定性
插入排序	直接插入排序	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$	稳定
	希尔排序	$O(n\log_2 n)$			$O(1)$	不稳定
交换排序	起泡排序	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$	稳定
	快速排序	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n^2)$	$O(\log_2 n)$	不稳定
选择排序	简单选择排序	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	不稳定
	堆排序	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(1)$	不稳定
归并排序	归并排序	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n)$	稳定

3.7 作业与思考题

1. 有序列{53,88,170,257,466,66,512,890,999,69}，写出采用希尔排序对该序列升序排序第二趟的结果。

2. 有序列{10,5,23,67,81,6,12,1,9}，使用堆排序算法进行升序排序，写出每一趟排序后的结果。

3. 有序列{28,4,6,46,87,58,51,24}，使用归并排序算法进行降序排序，写出第二趟排序后的结果。

4. 在本章学习的7种排序中，稳定排序方法有哪些？

5. 对n个关键字进行堆排序，最坏情况下的时间复杂度是多少？

6. 对序列{28,16,32,12,60,2,5,7}进行升序快速排序，写出第一趟排序后的结果。

7. 在本章学习的7种排序中，不稳定排序方法有哪些？

8. 对序列{15,9,7,8,20,−1,4}进行排序，进行一趟排序后关键字序列变为{9,15,7,8,20,−1,4}，采用的是什么算法？