> 2018年12月7日~2018年12月14日

排序算法的内存消耗：可以用空间复杂度来衡量，对于空间复杂度为$O(1)$的排序算法，称之为`原地排序`。

排序算法的稳定性：如果待排序的序列中存在值相等的元素，经过排序后，相等元素之间的先后顺序不变。例如，对于序列：`6,8,6,2,3,9`，经过排序后为：`2,3,6,6,8,9`，若其中两个6的先后顺序没有改变，则其为稳定的排序算法。应用场景：给电商交易系统中的"订单"排序，按照金额大小对订单数据排序，对于相同金额的订单以下单时间早晚排序。用稳定排序算法可简洁地解决。先按照下单时间给订单排序，排序完成后用稳定排序算法按照订单金额重新排序。

[算法可视化动画]

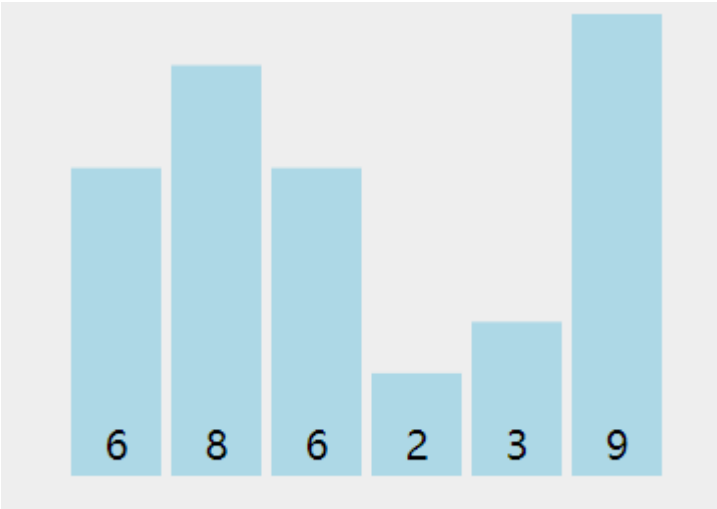
冒泡排序

1，算法思想

1. 比较相邻的元素。如果第一个比第二个大，就交换他们两个。
2. 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。
3. 针对所有的元素重复以上的步骤，除了最后一个。
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。

2，算法图解

对序列`6,8,6,2,3,9`进行冒泡排序：



3，算法实现

```
    public static <AnyType extends Comparable<? super AnyType>> void
bubbleSort(AnyType[] array) {
        boolean changed;
        for (int i = 0; i < array.length - 1; i++) {
            changed = false;
            for (int j = 0; j < array.length - i - 1; j++) {
                if (array[j].compareTo(array[j + 1]) > 0) {
                    AnyType temp = array[j];
```

```
                    array[j] = array[j + 1];
                    array[j + 1] = temp;
                    changed = true;
                }
            }
            if (!changed) {
                return;
            }
        }
    }
```

4，复杂度分析

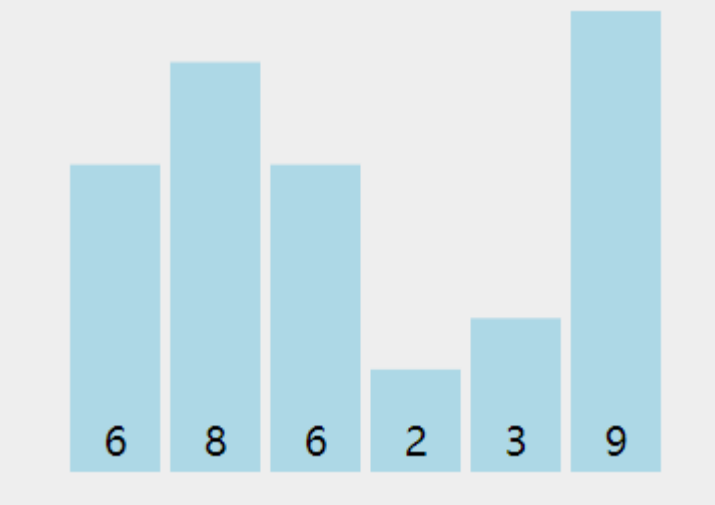
- 由于没有开辟新的空间，所以空间复杂度为$O(1)$，即为原地排序。
- 由于当两个元素大小相同时，冒泡排序不对其进行交换，因此大小相同的元素在排序前后不会改变顺序，因此冒泡排序时稳定的排序算法。
- 最好情况下，即待排序的序列是有序的，此时内层for循环一轮后便会结束，所以最好情况下的时间复杂度为$O(N)$。
- 最坏情况下，待排序的序列的逆序的，此时需要比较的次数为：$(N-1)+(N-2)+.....+1=\frac{(N-1)N}{2}$则最坏情况下的时间复杂度为$O(N^2)$。
- 平均情况下的时间复杂度用`有序度`与`逆序度`来分析，有序度是数组中具有有序关系的元素对个数，对于序列：`2,4,3,1,5,6`，其有序元素对有11个：(2,4),(2,3),(2,5),(2,6),(4,5),(4,6),(3,5),(3,6),(1,5),(1,6),(5,6)，因此这组数据的有序度为11。同理，对于一个逆序序列：`6,5,4,3,2,1`，其有序度为0，对于一个有序序列：`1,2,3,4,5,6`，其有序度为：$\frac{N(N-1)}{2}=15$，对于这种完全有序的序列的有序度称之为`满有序度`。那么逆序度即与有序度相反，可以通过以下公式求得：$逆序度=满有序度-有序度$排序的过程就是增加有序度，减少逆序度的过程，直至达到满有序度。冒泡排序有两个操作：比较与交换，每交换一次，有序度就加1。不管采用何种算法，交换次数总是确定的，交换次数即为逆序度。最坏情况下，初始有序度为0，此时要进行$\frac{N(N-1)}{2}$次交换。最好情况下，初始有序度为$\frac{N(N-1)}{2}$，此时不需进行交换。那么其交换的平均次数为$\frac{N(N-1)}{4}$，因为比较的次数比交换的次数要多，但是时间复杂度的上限为$O(N^2)$，所以平均时间复杂度为$O(N^2)$。

选择排序

1，算法思想

首先在未排序序列中找到最小元素，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元素，然后放到序列的第2个位置。以此类推，直到所有元素均排序完毕。

2，算法图解



3，算法实现

```
    public static <AnyType extends Comparable<? super AnyType>> void
selectionSort(AnyType[] array) {
        for (int i = 0; i < array.length; i++) {
            int min = i;
            for (int j = i + 1; j < array.length; j++) {
                if (array[j].compareTo(array[min]) < 0) {
                    min = j;
                }
            }
            if (min != i) {
                AnyType temp = array[min];
                array[min] = array[i];
                array[i] = temp;
            }
        }
    }
```

4， 复杂度分析

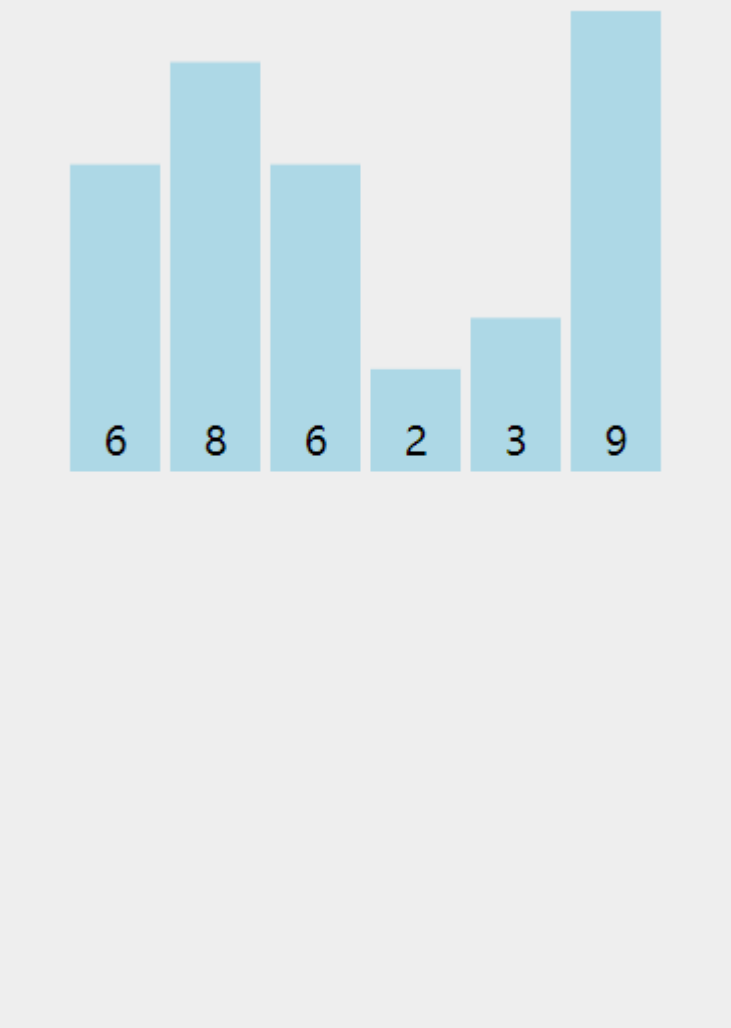
- 由于没有开辟新的空间，所以空间复杂度为$O(1)$，即为原地排序。
- 选择排序每次都要找剩余未排序元素中的最小值，并和前面的元素交换位置，破坏了稳定性，在2中图解中，对序列：`6,8,6,2,3,9`，其中两个6的位置发生了改变，因此选择排序时不稳定的。
- 由于无论是有序还是逆序，选择排序的比较次数都为：$(N-1)+(N-2)+.....+1=\frac{(N-1)N}{2}$即其最好与最坏时间复杂度为$O(N^2)$，那么其平均时间复杂度也为$O(N^2)$。

插入排序

1，算法思想

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。因而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。

2，算法图解



3，算法实现

```
    public static <AnyType extends Comparable<? super AnyType>> void
insertionSort(AnyType[] array) {
        int j;
        for (int i = 1; i < array.length; i++) {
            AnyType tmp = array[i];
            for (j = i; j > 0 && tmp.compareTo(array[j - 1]) < 0; j--) {
                array[j] = array[j - 1];
            }
            array[j] = tmp;
        }
    }
```

4，复杂度分析

- 由于没有开辟新的空间，所以空间复杂度为$O(1)$，即为原地排序。
- 在插入排序中，对于值相同的元素，将后面出现的元素，插入到前面出现元素的后面，保持原有的顺序不变，因此插入排序时稳定的排序算法。

- 最好情况下，待排序的序列是有序的，此时内循环的检测条件不成立而会终止，所以最好的时间复杂度为$O(N)$。
- 最坏情况下，待排序的序列是逆序的，此时数据需要插入的次数为$1+2+3+.....+N=\frac{N(N+1)}{2}$ 所以最坏情况下的时间复杂度为$O(N^2)$。
- 平均情况下，由于在数组中插入一个数据的平均时间复杂度为$O(N)$，那么对于插入排序，每次插入操作相当于在数组中插入一个数据，循环执行n次插入操作，因此平均时间复杂度为$O(N^2)$。

5，二分查找插入排序

在已排序序列中通过二分查找来确定插入的位置，以此减少比较次数提升算法效率，其算法实现如下：

```
    public static <AnyType extends Comparable<? super AnyType>> void
binaryInsertionSort(AnyType[] array) {
        for (int i = 1; i < array.length; i++) {
            AnyType tmp = array[i];
            //通过二分查找得出要插入的位置
            int index = getIndexByBinary(array, i - 1, tmp);
            //将index位置后面的元素整体往后挪动一位
            for (int j = i; j > index; j--) {
                array[j] = array[j - 1];
            }
            array[index] = tmp;
        }
    }

    public static <AnyType extends Comparable<? super AnyType>> int
getIndexByBinary(AnyType[] array, int i, AnyType tmp) {
        int index = 0;
        int end = i;
        while (index <= end) {
            int binary = index + (end - index) / 2;
            if (tmp.compareTo(array[binary]) < 0) {
                end = binary - 1;
            } else {
                index = binary + 1;
            }
        }
        return index;
    }
```

希尔排序

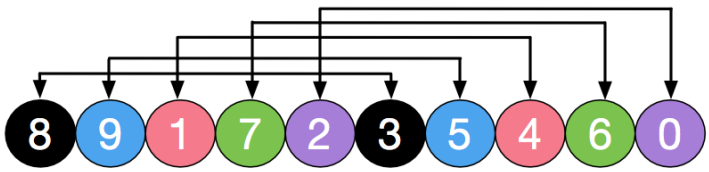
1，算法思想

希尔排序通过将比较的全部元素分为几个区域来提升插入排序的性能。这样可以让一个元素可以一次性地朝最终位置前进一大步。然后算法再取越来越小的步长进行排序，算法的最后一步就是普通的插入排序，但是到了这步，需排序的数据几乎是已排好的了（此时插入排序较快）。

2，算法图解

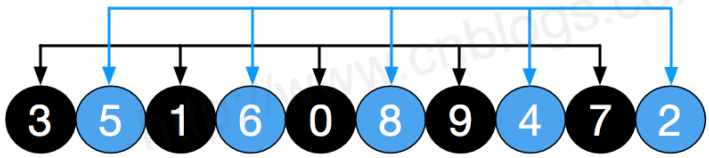
原始数组 以下数据元素颜色相同为一组

初始增量 gap=length/2=5，意味着整个数组被分为5组，[8,3] [9,5] [1,4] [7,6] [2,0]



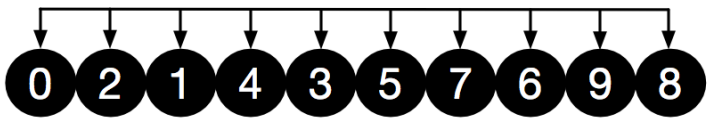
对这5组分别进行直接插入排序，结果如下，可以看到，像3，5，6这些小元素都被调到前面了，

然后缩小增量 gap=5/2=2,数组被分为2组 [3,1,0,9,7] [5,6,8,4,2]



对以上2组再分别进行直接插入排序，结果如下，可以看到，此时整个数组的有序程度更进一步啦。

再缩小增量gap=2/2=1，此时，整个数组为1组[0,2,1,4,3,5,7,6,9,8]，如下



经过上面的“宏观调控”，整个数组的有序化程度成果喜人。

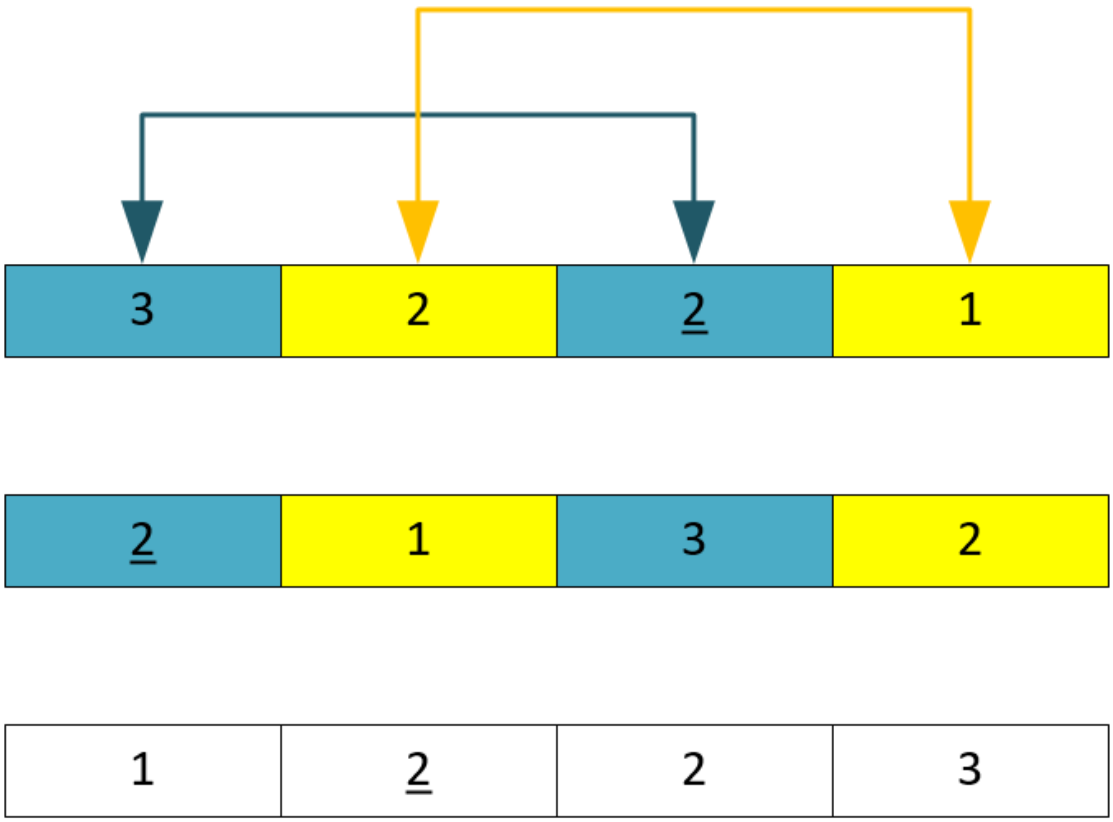
此时，仅仅需要对以上数列简单微调，无需大量移动操作即可完成整个数组的排序。

3，算法实现

```java
    public static <AnyType extends Comparable<? super AnyType>> void
shellSort(AnyType[] array) {
        int j;
        for (int gap = array.length / 2; gap > 0; gap /= 2) {
            for (int i = gap; i < array.length; i++) {
                AnyType tmp = array[i];
                for (j = i; j >= gap && tmp.compareTo(array[j - gap]) < 0; j -= gap) {
                    array[j] = array[j - gap];
                }
                array[j] = tmp;
            }
        }
    }
```

4，复杂度分析

- 由于没有开辟新的空间，所以空间复杂度为$O(1)$，即为原地排序。
- 希尔排序是 不稳定 的排序算法：



可以看出，两个2的前后顺序改变了。

总结

排序算法	空间复杂度	稳定性	最好时间复杂度	最坏时间复杂度	平均时间复杂度
冒泡	$O(1)$	√	$O(N)$	$O(N^2)$	$O(N^2)$
选择	$O(1)$	×	$O(N^2)$	$O(N^2)$	$O(N^2)$
插入	$O(1)$	√	$O(N)$	$O(N^2)$	$O(N^2)$