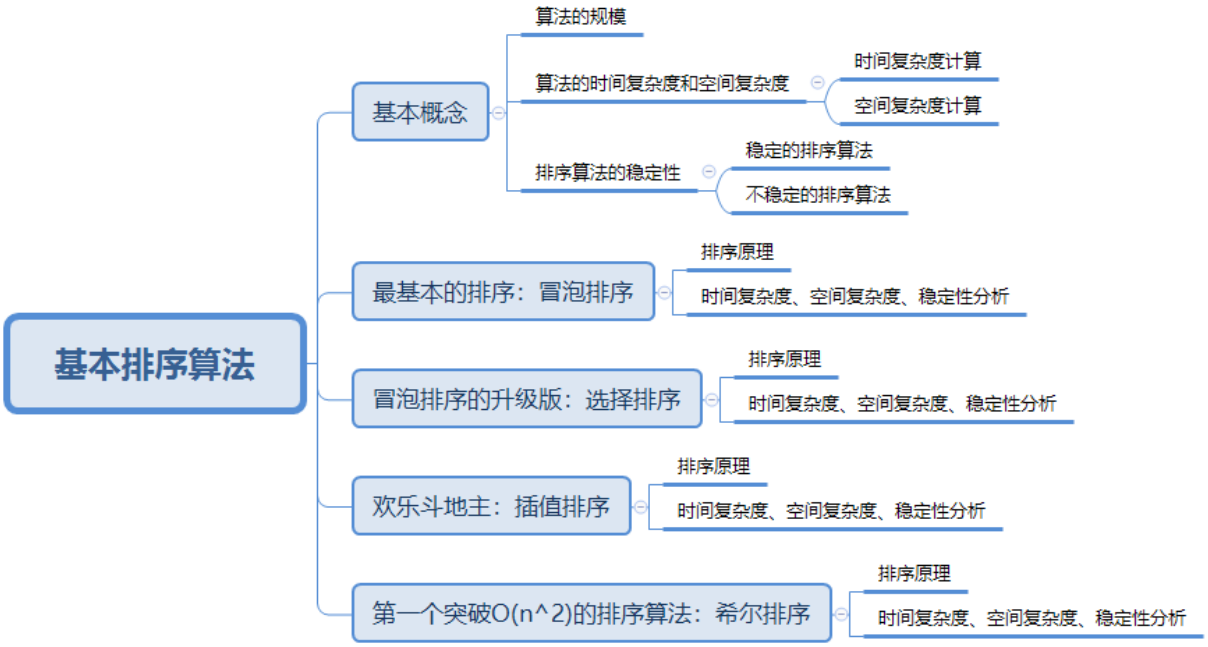


1.基本概念普及

①算法问题的规模

所谓算法问题的规模，实际上就是对算法处理问题大小的一种抽象，我们一般习惯使用n、m等字符表示一个问题的规模

举一个比较实际的例子：例如一个排序算法是针对数组进行排序的，那么这个数组中的元素数量就可以认为是这个排序问题的规模

再比如说：使用二分搜索算法在一个ArrayList集合中查询一个指定元素的下标，那么这个集合的长度就可以看做是这个二分搜索问题的规模

所以：算法的规模实际上就是这个算法解决的问题中，处理数据多少的一种描述

②算法的时间复杂度和空间复杂度

俗话说：算法没有优劣之分，只有快慢之别

不同的算法适用于不同的场景，算法之间本身没有好与坏的区别，有的只是在处理相同规模问题的时候，谁快谁慢

哪一种算法占用辅助空间更少的区别

算法的时间复杂度和空间复杂度是用来衡量一个算法快慢与否以及运行时占用辅助空间大小的一种计算标准，一般用O()表示

这里需要特殊强调的是：算法的时间复杂度一般是无法精确计算的，因为一个算法在运行时消耗的时间是以毫秒为单位来统计的

但是因为计算机硬件配置的不同，所以同一个算法在不同计算机上，即使计算的是同一组数据，那么使用的时间也是有很大差异的

例如：同样是使用冒泡排序对10万个随机正整数进行排序操作，在一台单核CPU的计算机上和在一台i7多核CPU的计算机上进行计算

其运算时间一定是具有很大差异的

所以我们得出一个结论：算法的时间复杂度是不能精确计算的，所有算法的时间复杂度只不过是对算法运行时间和处理问题规模之间关系的一种估算描述

1.时间复杂度

时间复杂度是用来大致描述算法运行时间和算法处理问题规模之间关系的一种衡量标准

一个算法的时间复杂度越高，那么也就说明这个算法在处理问题的时候所花费的时间也就越长

但是正如前面我们说过的，一个算法的时间复杂程度并不能被精确计算出来，我们计算算法时间复杂程度，也只不过是粗略的估算

那么，在一个算法的运算过程用时公式被计算出来之后，我们遵从如下“忽略”标准描述其时间复杂度：

1.忽略公式中的常数项

2.忽略公式中的低次幂项，仅保留公式中的最高次幂项

3.忽略公式中最高次幂项的常数系数

4.如果一个公式中所有的项都是常数项，那么这个算法的时间复杂度统一表示为O(1)

下面我们来举几个相关的案例进行说明：

例1：算法1的运算过程用时公式是：$2n^2 + 5n + 6$，则这个算法的时间复杂度是$O(n^2)$

例2：算法2的运算过程用时公式是：nlogn + 5n + 2，则这个算法的时间复杂度是O(nlogn)

例3：算法3的运算过程用时公式是：2n + 7，则这个算法的时间复杂度是O(n)

例4：元素交换算法只有3个步骤，其运算过程用时公式为：1+1+1，则其时间复杂度是O(1)

在排序算法中，常见的时间复杂度有3种，分别是：$O(n^2)$、O(nlogn)、O(n)

其中logn表示以2为底n的对数，这个值到底是怎么计算出来的，在快速排序算法中我们会详细进行分析

上述的3种时间复杂度之间的大小关系是：$O(n^2) > O(nlogn) > O(n)$

也就是说，时间复杂度为O(n^2)的排序算法运行效率最低，也就是最慢；时间复杂度为O(n)的排序算法运行效率最高，也就是最快

2.空间复杂度

空间复杂度是用来衡量一个算法在运行过程当中，在除了保存原始数据的空间之外，还需要额外消耗多少空间的一种衡量标准

举个例子：冒泡排序在执行过程中，只需要消耗一个临时变量，用来交换两个反序的元素即可，所以冒泡排序的空间复杂度就是O(1)

空间复杂度和时间复杂度一样，也是用来描述算法问题规模和运算时消耗额外空间之间关系的一种表达式，并不是用来详细计算数值的公式

排序算法中空间复杂度常见的也是有3种：O(1)、O(n)、O(logn)

而空间复杂度和时间复杂度相似的是，空间复杂度越高，就表示这个算法在运行过程中所需要消耗的额外空间也就越多

从这个角度来讲，上面的三种空间复杂度之间的关系可以看做：O(n) > O(logn) > O(1)

也就是说，在空间复杂度层面来讲，O(1)是最小的空间复杂度

③排序算法的稳定性

排序算法的稳定性指的是，在一个排序算法处理的数组或者集合中，如果存在取值相同的元素，

那么在排序完成前后，这些取值相同的元素之间的相对顺序有没有发生变化

如果排序之后，取值相同元素之间的相对顺序没有发生变化，那么这个排序算法就是稳定的

如果排序之后，取值相同元素之间的相对顺序发生了变化，那么这个排序算法就是不稳定的

具体案例我们可以看如下图片：

假设在待排序序列中存在两个取值相同的2，那么我们将这两个2分别定义为2a和2b

排序之前，他们的相对顺序如下：

3　2a　5　1　2b　7　6

现在对上述序列进行排序操作

稳定排序结果：

1　2a　2b　3　5　6　7

不稳定排序结果：

1　2b　2a　3　5　6　7

上面我们已经解释了什么是算法的时间复杂度、空间复杂度和排序算法的稳定性问题

为了让大家一目了然的记住我们之后所要讲解的8种排序算法的相关概念取值，特列出如下表格：

序号	排序算法名称	时间复杂度	空间复杂度	排序算法稳定性
1	冒泡排序	$O(n^2)$	O(1)	稳定排序
2	选择排序	$O(n^2)$	O(1)	不稳定排序
3	插值排序	$O(n^2)$	O(1)	稳定排序
4	希尔排序	$O(n^k)$ (1.3 <= k <= 2)	O(1)	不稳定排序
5	归并排序	O(nlogn)	O(n)	稳定排序
6	快速排序	O(nlogn)	O(logn)	不稳定排序
7	堆排序	O(nlogn)	O(1)	不稳定排序
8	桶排序	O(n+m)（m为排序元素最大值)	O(m)	稳定排序

2.最基本的排序：冒泡排序

①排序原理

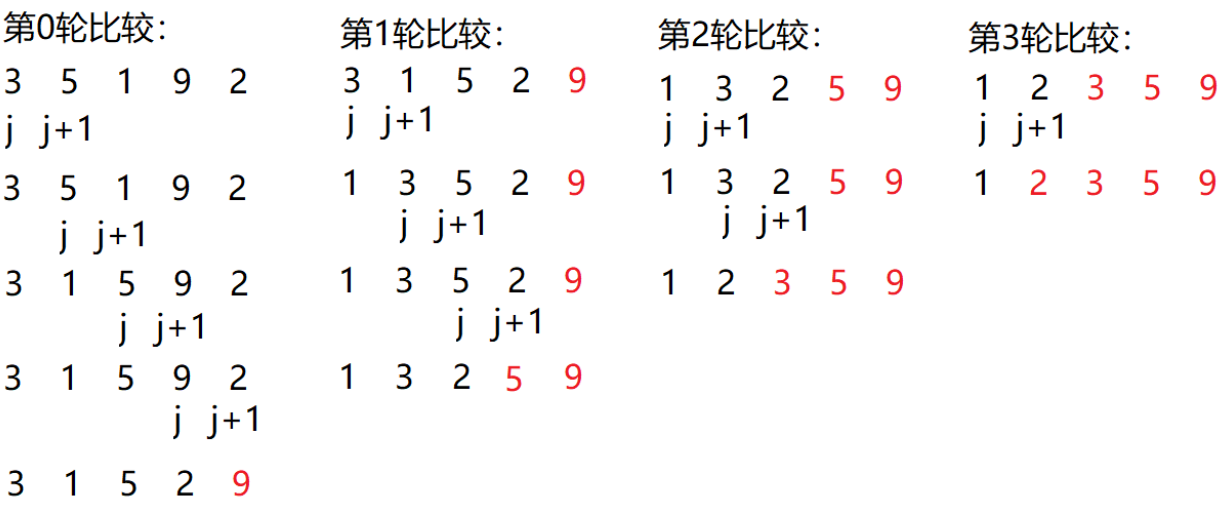
冒泡排序作为排序算法中最为基础的一种排序算符，他的排序原理也是最为简单的：相邻位比较，反序则互换

所谓的相邻位，实际上就是排序数组中下标紧挨着的两个元素，用数组下标进行说明，就是下标为j的元素和下标为j+1的元素之间的关系

反序则互换指的是，如果两个相邻元素之间的取值，和约定的数组顺序（例如升序或者降序）是相反的

那么数组中这两个位置上的元素要通过一个临时变量来完成一次交换

下面我们通过一个图片来说明冒泡排序（升序排序）的流程：



从上述流程图我们可以总结出如下结论，通过这些结论我们有能够进一步整理出冒泡排序算法的代码：

1.每进行一轮冒泡排序，数组的待排序序列中的最大值都会到达待排序序列的最后，我们将这个动作称之为元素的“归位”

2.在进行第n轮比较的时候（n的取值从0开始），那么数组中有n个元素是有序的

3.当数组中n个元素是有序的时候，这n个元素将不会参与到本轮排序中，因为这些元素已经归位了，到达了它们在数组中应该所在的位置

4.如果数组中存在m个元素，那么整体的比较轮次，只要进行m-1轮即可，因为在m-1轮排序之后，数组中只剩下1个元素没有经过排序

但是一个元素本身就是有序的，不需要再和其他元素进行比较，那么这个元素也就自然而然的归位了

根据上述特性，我们可以得到如下冒泡排序算法代码：

```java
public void bubbletSort(int[] array) {

    /*
     * [3]用最外层循环i控制比较的轮次，
     * 第i轮比较的时候，就有i个元素已经有序，则这i个元素是不需要进行排序的，
     * 而在整个长度为n的数组中，只要进行n-1轮排序即可
     */
    for(int i = 0; i < array.length-1; i++) {
        /*
         * [2]内层循环控制两个相邻位下标的变化，
         * 在第i轮比较中，数组的后i个元素就是有序的，所以不需要参与比较
         */
        for(int j = 0; j+1 <= array.length-1-i; j++) {
            //[1]相邻位比较，反序则互换
            if(array[j] > array[j+1]) {
                int tmp = array[j];
                array[j] = array[j+1];
                array[j+1] = tmp;
            }
        }
    }

}
```

上述的算法代码我们是采用“由内到外”的思想去编写的：先解决内部比较简单的步骤，凡是重复的步骤，只要采用循环就能够处理

所以，有的时候我们反过来思考一个问题，可能会变得更简单

②时间复杂度、空间复杂度、稳定性分析

1.冒泡排序的时间复杂度

在进行冒泡排序的时候，假设数组中存在n个元素：

第0轮排序：n个元素比较n-1次

第1轮排序：n-1个元素比较n-2次

第2轮排序：n-2个元素比较n-3次

...

第n-3轮排序：3个元素比较2次

第n-2轮排序：2个元素比较1次

那么将上面比较的次数相加：(n-1) + (n-2) + (n-3) + ... + 3 + 2 + 1 = n*(n-1)/2 = (n^2-n)/2

根据之前讲解的计算算法时间复杂度的方式，我们可以知道，上述公式中的最高次幂项是$n^2/2$

所以冒泡排序的时间复杂度是$O(n^2)$

2.冒泡排序的空间复杂度

整个冒泡排序在运算过程中，仅仅使用了一个可以重复利用，用来交换相邻位元素的临时变量

所以冒泡排序的空间复杂度是O(1)

3.冒泡排序的稳定性

在冒泡排序进行相邻位比较的时候，如果相邻的两个元素大小是相等的，那么这两个元素并不会进行互换

并且，冒泡排序算法每一轮比较都是从数组的最开始，一步一步向后进行比较

所以并不会出现相同取值的两个元素之间互换的情况

所以冒泡排序是一种稳定的排序算法

3.冒泡排序的升级版：选择排序

①排序原理

选择排序的原理与冒泡排序相比更加容易理解：选定一个标准位，将待排序序列中的元素与标准位元素逐一比较，反序则互换

其中所谓的标准位实际上也是数组中的一个下标位，在选定了这个下标位之后，在一轮排序中这个标准位将不再移动，

然后将待排序序列——也就是这个标准位之后所有元素组成的序列——中的元素逐个和标准位上的值进行比较

如果某一个待排序序列上的值比标准位上的值更小（升序排序）或者更大（降序排序），那么就将这个元素和标准位上的元素进行互换即可

标准位的选择一般选取待排序序列的第一个下标作为标准位使用

下面我们还是通过一个图片来理解选择排序的运行过程

使用变量i来表示当前标准位的下标位置，使用变量j来遍历待排序序列

第0轮排序：

3	1	9	2	8
i				j
3	1	9	2	8
i			j	
2	1	9	3	8
i		j		
2	1	9	3	8
i	j			
1	2	9	3	8

第1轮排序：

1	2	9	3	8
	i			j
1	2	9	3	8
	i		j	
1	2	9	3	8
	i	j		
1	2	9	3	8

第2轮排序

1	2	9	3	8
		i		j
1	2	8	3	9
		i	j	
1	2	3	8	9

第3轮排序：

1	2	3	8	9
			i	j
1	2	3	8	9

从上图中我们依然能够总结出一些结论，用于创建我们的代码：

1.和冒泡排序相似的是，在第n轮排序（n从0开始取值）进行的时候，整个数组中有n个元素是有序的，这n个元素是不参与到排序中的

2.每一轮排序过后，标准位上的值都是整个待排序序列中的最小值（升序排序）或者最大值（降序排序）

实际上，选择排序还可以做另一种解释：使用标准位作为一个存储空间，然后不断地在待排序序列中选择最小值或者最大值，放在这个存储空间中

根据上述规律的总结，我们可以得到如下代码去实现一个选择排序：

```
public void selecttionSort(int[] array) {

    /*
     * [1]使用外层循环指定标记位，
     * 然后将待排序序列中的元素一个一个的和标记位上的值进行比较，反序则互换；
     * 比较依然是进行n-1次即可
     */
    for(int i = 0; i < array.length-1; i++) {
        /*
         * [2]使用内层循环j循环倒序遍历整个待排序序列，
         * 使用array[j]和标准位的array[i]进行大小比较
         */
        for(int j = array.length-1; j > i; j--) {
            /*
             * [3]标准位上的元素和待排序序列中的元素进行比较，反序则互换
             */
            if(array[i] > array[j]) {
                int tmp = array[i];
                array[i] = array[j];
```

```
                array[j] = tmp;
            }
        }
    }

}
```

②时间复杂度、空间复杂度、稳定性分析

1.选择排序的时间复杂度

从上述原理图看来，选择排序每一轮比较的情况如下：

第0轮排序：使用后n-1个元素与标准位比较

第1轮排序：使用后n-2个元素与标准位比较

第2轮排序：使用后n-3个元素与标准位比较

...

第n-3轮排序：使用后2个元素与标准位比较

第n-2轮比较：使用最后一个元素与标准位比较

将上述比较的次数相加得到：(n-1) + (n-2) + (n-3) + ... + 3 + 2 + 1 = n*(n-1)/2 = n^2-n/2

这个时间复杂度的公式与冒泡排序的时间复杂度公式是相同的

所以，选择排序的时间复杂度依然是O(n^2)

2.选择排序的空间复杂度

在选择排序中，在使用待排序序列中的元素与标准位上的元素进行比较的时候

如果发现两个元素是反序的，那么仅需要使用一个可以重复利用的临时变量进行元素交换即可

这一点依然与冒泡排序中的情况相同，所以，选择排序的空间复杂度依然是O(1)

3.选择排序的稳定性

从图中的流程我们可以看出，选择排序每一轮次的比较都是从待排序序列的最后，逐个向前进行比较的

这就导致如果待排序序列中如果存在两个取值相同的元素，那么后面的元素一定会首先与标准位上的值进行互换

而前面的值则不会与标准位进行互换，见下图：

3 2a 2b 5 7 2b首先与标准位上的3进行交换
i j

2b 2a 3 5 7 在2a与标准位进行比较的时候，标准位此时的取值是2b，与2a同值，不发生交换
i j

所以，选择排序是一种不稳定的排序算法

注意：有的同学可能已经发现：在遍历待排序序列的时候，如果不是倒序遍历，

而是从i+1位下标开始，逐个向后正序遍历，那么与标准位首先进行互换的同值元素就是2a，也就是说这种选择排序变成了一种稳定的排序算法

是的，这一情况是没错的，但是从实践运行的角度出发我们发现：倒序遍历待排序序列的方式，能够在一定程度上提升选择排序算法的效率

比正序遍历待排序序列要快上一些，所以在这里我们依旧选择使用倒序的方式遍历待排序序列

4.欢乐斗地主：插值排序

①排序原理

在学习插值排序之前，首先让我们回忆一下我们曾经玩过的扑克牌游戏的抓牌的过程：

假设现在我的手里有4张牌，分别是♥A，♦3，♥8和❀9，并且我从牌堆中又抽取了一张牌，这张牌是♠7

那么，我们应该如何确定这张♠7应该放在我手牌中的什么位置，才能够保证手牌是有序（仅比较大小，忽略牌的花色）的呢?

实际上很简单，首先我们发现：我的手牌现在本身就是有序的，然后我们可以将这张♠7首先和❀9进行比较，

我们发现，♠7小于❀9，那么我们将这张♠7放在❀9的前面，与它前面的一张牌继续比较

❀9的前面是♥8，♠7又小于♥8，所以我们将♠7放在♥8的前面，继续和前一张牌进行比较

♥8的前面是♦3，♠7不小于♦3，那么此时♠7就找到了自己在手牌中的位置，不需要继续向前移动并进行比较了

上面这一系列抓牌的流程，实际上就是一个完整的插值排序的流程，

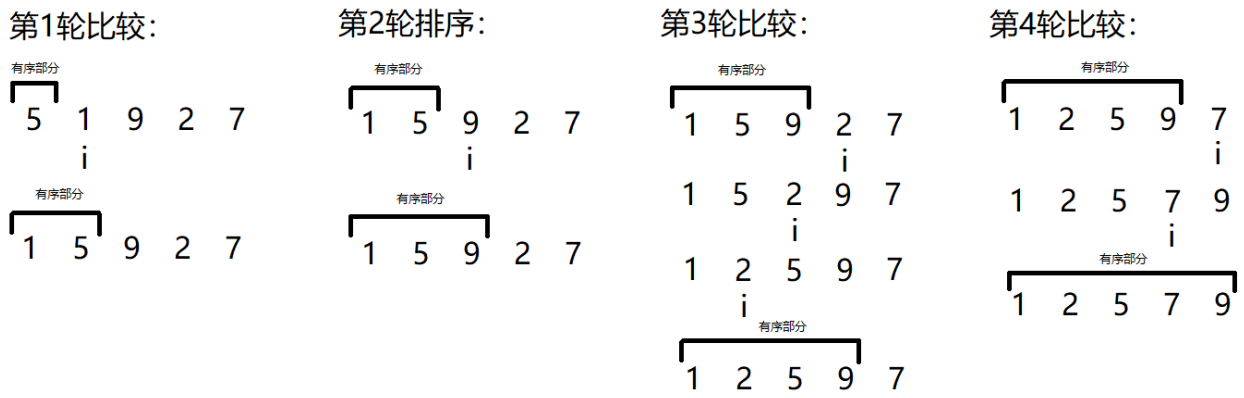
所以我们可以将插值排序的流程总结成为：将待排序序列（牌堆）中的第一个元素（抓到的牌）与前面有序序列（手牌）中的元素逐一比较

反序则互换并继续向前比较，否则停止比较，元素归位

实际上，插值排序的思想得益于“前面的n个元素都是有序”的这一思想，也就是说：新元素之所以下不小于某一个元素之后就可以停止比较和互换

是因为如果他不小于第m个元素，那么在有序序列中他也一定不小于前m-1个元素

下面我们使用一张图片对插值排序的流程进行演示：



从上图中我们依然能够总结出一些规律：

1.在开始首轮排序比较的时候，我们一般都是选择数组中下标为1的元素作为抽到的牌

而保留下标为0的元素作为已有的一张手牌，也就是说：在第n轮排序过程中（n从1开始取值），我们使用下标为n的数组元素作为待比较的元素

2.并不是每一轮排序，我们都要将抽取到的新牌与手中有序的m张牌都比较一遍，这也是插值排序相比起前两种排序稍微快一些的原因

3.因为我们首先认定数组中下标为0的元素就是初始的一张手牌，所以如果数组中有n元素，我们依然只要进行n-1轮排序

就能够保证数组中所有的元素有序

根据上面的理论，我们可以得到如下代码实现的插值排序代码：

```
public void insertionSort(int[] array) {

    /*
     * [1]使用外层排序控制从“牌堆”中抽取的“手牌”
     * 手牌从数组中的第2个元素开始抽，也就是下标为1的元素开始，
     * 下标为0的元素当做已有手牌
     */
    for(int i = 1; i < array.length; i++) {
        /*
         * [2]使用内层循环控制抽到的牌和手牌之间的比较和排序，
         * 只要手牌更小，就一直向前交换，直到找到合适位置，或者数组到头为止
         */
        for(int j = i; j-1 >= 0 && array[j] < array[j-1]; j--) {
            /*
```

```
15                 * [3]直接进行交换，因为比较的步骤已经定义在循环条件中了
16                 */
17                int tmp = array[j];
18                array[j] = array[j-1];
19                array[j-1] = tmp;
20            }
21        }
22
23    }
```

②时间复杂度、空间复杂度、稳定性分析

1.插值排序的时间复杂度

在上面的内容中我们也已经说过：插值排序并不是每一轮排序都要进行n-1次比较，但是如果在最坏情况下——也就是将数组元素逆序的情况下

我们依然需要将待比较元素和前面有序的元素全部进行比较，那么在一个长度为n的数组中：

第1轮排序：比较1次

第2轮排序：比较2次

第3轮排序：比较3次

...

第n-2轮排序：比较n-2次

第n-1轮排序：比较n-1次

将上述比较次数相加：1 + 2 + 3 + ... + (n-2) + (n-1)，所得到的的是和冒泡排序和选择排序一样的n*(n-1)/2

也就是说，插值排序的时间复杂度同样是$O(n^2)$

但是，如果一个序列本身就是近似有序的，那么插值排序的效率就会更高，这一点我们会在后面的希尔排序中进行说明

2.插值排序的空间复杂度

从上述原理我们可以看出，插值排序也只是在元素进行交换的时候使用了一个可以重复使用的临时变量空间

那么，插值排序的空间复杂度同样是O(1)

3.插值排序的稳定性

如果我们在牌堆中抽到了一张我们手中已有取值的牌，因为我们是从手牌的末尾开始逐一向前比较这张牌的

所以在遇到取值相同的牌的时候，我们并不会将两张牌进行交换，而是直接同值比较，结束这一轮排序

所以插值排序是一种稳定的排序算法

5.第一个突破O(n^2)的排序算法：希尔排序

①排序原理

希尔排序算法（Shell's Sort）是排序算法中第一个时间复杂度突破O(n^2)的排序算法，

也就是说：前面我们学过的冒泡排序、选择排序和插值排序的理论实践复杂度都是O(n^2)，也就是说时间复杂度较高，比较慢一些

而希尔排序是第一个从理论计算到实际操作中，时间复杂度都小于O(n^2)的一种排序算法，所以希尔排序在排序算法界具有很重要的意义

希尔排序的实际上是一种变形的插值排序。它将整个序列按照规定的步长（step）进行分部分取值，并将这些值进行插值排序；

在一轮排序之后，再将步长折半，以此类推，当步长取值为1的时候，整个希尔排序也就退化成为了一个单纯的插值排序

但是，正如前面我们说过的，如果一个序列本身就是近似有序的，那么插值排序的效率将会大大提升

所以，实际上希尔排序前面进行的分步和按照步长排序，都是在为这一点打下基础，最终提升插值排序的效率

希尔排序的这种“步步紧逼”的做法，我们称之为“缩小增量法”

下面我们使用一张图对希尔排序算法进行说明：

希尔排序的规律总结如下：

1.希尔排序总共进行几轮，取决于步长的取值，当步长取值为1的时候，就是最后一轮排序

2.希尔排序的步长，在每一轮中都是上一轮步长的一半，如果不能整除，则向下取整

3.希尔排序的每一轮比较，实际上都是将数组中间隔为step的诸多元素进行插值排序

所以在每一轮比较之后，数组整体都更接近有序状态

4.关于希尔排序的初始步长选择，我们一般定义为待排序序列长度的一半，然后每进行一轮希尔排序，步长折半

从上述步骤的演示我们可以推断出如下的代码实现步骤：

```
/**
 * 希尔排序
 * @param array
 */
public void shellSort(int[] array) {
    int step = array.length / 2;  //定义初始步长是数组长度的一半
    while(step > 0) {
        //使用一个循环，控制步长内的元素全部进行一次插值排序
        for(int start = 0; start < step; start++) {
            insertionSort(array, start, step);  //内部使用间隔为步长的插值排序算
        }
        step /= 2;  //步长变成原来的一半
    }
}

/**
```

```
 * 希尔排序是一种基于插值排序的算法
 * 这个方法是希尔排序内部使用的一种带有步长，并指定比较起点的插值排序算法
 * @param array 待排序数组
 * @param start 比较起点
 * @param step 步长
 */
private void insertionSort(int[] array, int start, int step) {
    for(int i = start + step; i < array.length; i += step) {
        for(int j = i; j-step >= 0 && array[j] < array[j-step]; j -= step) {
            int tmp = array[j];
            array[j] = array[j-step];
            array[j-step] = tmp;
        }
    }
}
```

②时间复杂度、空间复杂度、稳定性分析

1.希尔排序的时间复杂度

希尔排序的时间复杂度我们计算起来比较复杂，所以在这里我们直接记住这个结论即可：

在数组整体有序的情况下，希尔排序的时间复杂度是接近$O(n^{1.3})$的

在数组整体逆序的情况下，希尔排序的时间复杂度和插值排序接近，是$O(n^2)$

所以，希尔排序的时间复杂度我们进行如下表示：

希尔排序的时间复杂度：$O(n^k)$ $(1.3 <= k <= 2)$

注意：有的同学认为，希尔排序在代码实现的时候，相当于嵌套了4层循环结构，那它的时间复杂程度应该是$O(n^4)$

实际上这种认知是错误的。通过算法的循环嵌套层数推断算法的事件复杂程度的确是一种技巧，但是并不是百试百灵的

一个算法的时间复杂程度，依然要通过分析算法的基本流程本身才能够得到

2.希尔排序的空间复杂度

希尔排序本身是基于插值排序的，所以在进行排序的过程中我们需要一个临时变量去进行元素交换

同时我们还需要一个额外的变量保存当前的步长，所以这样一来我们就需要两个额外空间进行希尔排序的运算

但是，之前我们说过，只要所需的辅助空间数量是常量级别的，那么空间复杂度都是O(1)

所以，希尔排序的空间复杂度依然是O(1)

3.希尔排序的稳定性

希尔排序虽然是基于插值排序的，并且插值排序也是一种稳定的排序算法，

但是我们在分布对数组进行排序的时候，何有可能将两个具有相同取值的元素的相对位置进行改变

所以，希尔排序是一种不稳定的排序算法

6.4种基本排序算法的效率比较

上面我们已经学习的4种基本排序算法：冒泡排序、选择排序、插值排序和希尔排序

下面我们通过一个实验案例来进行上述4种排序算法效率的直观比较

实验内容：创建4个具有相同初始化长度、初始化元素内容和元素顺序的，长度为100000的正整数数组

数组中填充的全部都是取值范围在[0, 100000]之间的随机正整数

并且为了消除数据之间差异导致排序效果上带来的影响，我们选择首先创建一个这样的数组，并将这个数组复制4份

分别交给对应的排序算法进行排序，同时使用时间戳的方式，记录每一种排序算法的运行时间。

实验代码如下：

```
public class TestSpeed {

    public static void main(String[] args) {

        BubbleSort bs = new BubbleSort();
        SelectionSort ss1 = new SelectionSort();
        InsertionSort is = new InsertionSort();
        ShellSort ss2 = new ShellSort();

        System.out.println("创建一个100000整数级随机数数组，分别进行排序，比较4种

        long start = 0;
        long end = 0;

```

```
        int[] originArray = new int[100000];  //创建一个原始数组
        //填充随机数组
        for(int i = 0; i < originArray.length; i++) {
            originArray[i] = (int)(Math.random() * 100000);  //原始数组的每一个
        }

        //为了公平起见，我们将原始数组拷贝4份，分别使用4种排序算法进行排序计算事件，
        int[] array1 = Arrays.copyOf(originArray, originArray.length);
        int[] array2 = Arrays.copyOf(originArray, originArray.length);
        int[] array3 = Arrays.copyOf(originArray, originArray.length);
        int[] array4 = Arrays.copyOf(originArray, originArray.length);

        //开始比较
        System.out.println("冒泡排序开始……");
        start = System.currentTimeMillis();
        bs.bubbletSort(array1);
        end = System.currentTimeMillis();
        System.out.println("冒泡排序结束，用时" + (end - start) + "毫秒");

        System.out.println("----------");

        System.out.println("选择排序开始……");
        start = System.currentTimeMillis();
        ss1.selecttionSort(array2);
        end = System.currentTimeMillis();
        System.out.println("选择排序结束，用时" + (end - start) + "毫秒");

        System.out.println("----------");

        System.out.println("插值排序开始……");
        start = System.currentTimeMillis();
        is.insertionSort(array3);
        end = System.currentTimeMillis();
        System.out.println("插值排序结束，用时" + (end - start) + "毫秒");

        System.out.println("----------");

        System.out.println("希尔排序开始……");
        start = System.currentTimeMillis();
        ss2.shellSort(array4);
        end = System.currentTimeMillis();
        System.out.println("希尔排序结束，用时" + (end - start) + "毫秒");

```

```
    }

}
```

上述代码的运行结果：

```
创建一个100000整数级随机数数组，分别进行排序，比较4种基本排序算法用时
冒泡排序开始……
冒泡排序结束，用时11118毫秒
----------
选择排序开始……
选择排序结束，用时5285毫秒
----------
插值排序开始……
插值排序结束，用时2594毫秒
----------
希尔排序开始……
希尔排序结束，用时12毫秒
```

从上述程序运行的结果来看，在这4种基本排序算法中：

1.冒泡排序的效率最低

2.冒泡排序、选择排序和插值排序虽然理论上的时间复杂度都是$O(n^2)$，但是选择排序和插值排序的实际表现会更好一些

3.希尔排序的效率远远高于前面的3种排序算法，甚至所用时间和前面的3种算法都不在一个数量级上

但是，后面我们还会学到更加强大的4种高级排序算法，他们的运行效率会更加强大。