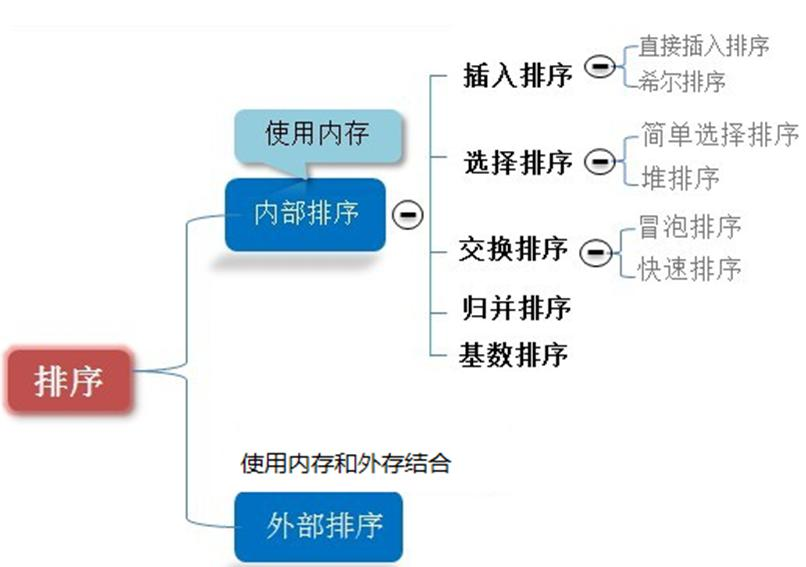
# 排序算法

## 1、简介

排序也称排序算法(Sort Algorithm)， 排序是将一组数据， 依指定的顺序进行排列的过程

### 排序算法的分类

* 内部排序：指将需要处理的所有数据都加载到内部存储器(内存)中进行排序。
* 外部排序法：数据量过大， 无法全部加载到内存中， 需要借助外部存储(文件等)进行排序。
* 常见的排序算法分类



## **2、算法的复杂度**

### **2.1、时间复杂度的计算方法**

#### 方法一：事后统计

存在两个问题：

1. 一想对设计的算法运行性能进行评测，需要实际运行程序
2. 二是所得时间的统计量依赖于计算机的硬件、 软件等环境因素, 这种方式， 要在同一台计算机的相同状态下运行， 才能比较哪个算法速度更快

#### 方法二：事前估算

过分析某个算法的时间复杂度来判断哪个算法更优

### 2.2、时间频度

#### 基本介绍

* 一个算法花费的时间与算法中语句的执行次数成正比例， 哪个算法中语句执行次数多， 它花费时间就多。
* 一个算法中的语句执行次数称为语句频度或时间频度。 记为 T(n)

### 2.3、时间复杂度

* 一般情况下， 算法中的基本操作语句的重复执行次数是问题规模 n 的某个函数， 用 T(n)表示， 若有某个辅助函数 f(n)， 使得当 n 趋近于无穷大时， T(n) / f(n) 的极限值为不等于零的常数， 则称 f(n)是 T(n)的同数量级函数。记作 T(n)=Ｏ ( f(n) )， 称Ｏ ( f(n) ) 为算法的渐进时间复杂度， 简称时间复杂度。
* T(n) 不同， 但时间复杂度可能相同。

如： T(n)=n²+7n+6 与 T(n)=3n²+2n+2 它们的 T(n) 不同， 但时间复杂度相同， 都为 O(n²)。

* 计算时间复杂度的方法：

1. 用常数 1 代替运行时间中的所有加法常数 T(n)=n²+7n+6 => T(n)=n²+7n+1
2. 修改后的运行次数函数中， 只保留最高阶项 T(n)=n²+7n+1 => T(n) = n²
3. 去除最高阶项的系数 T(n) = n² => T(n) = n² => O(n²)

### **2.4、常见的时间复杂度**

#### **2.4.1、常见时间复杂度概述**

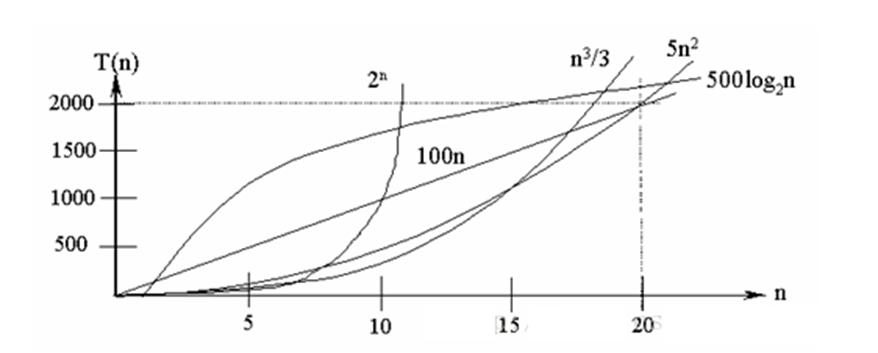
常见时间复杂度

* + 常数阶 O(1)
  + 对数阶 O(log2n)
  + 线性阶 O(n)
  + 线性对数阶 O(nlog2n)
  + 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)

结论：

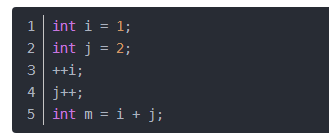
常见的算法时间复杂度由小到大依次为： Ο (1)＜Ο (log2n)＜Ο (n)＜Ο (nlog2n)＜Ο (n2)＜Ο (n3)＜ Ο (nk) ＜ Ο (2n) ， 随着问题规模 n 的不断增大， 上述时间复杂度不断增大， 算法的执行效率越低

从图中可见， 我们应该尽可能避免使用指数阶的算法

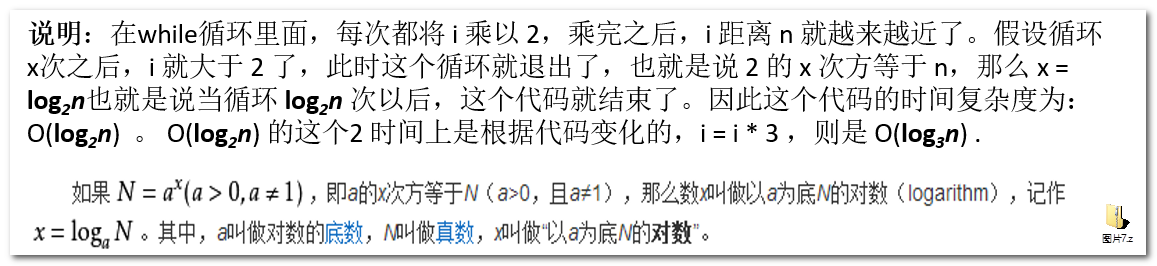


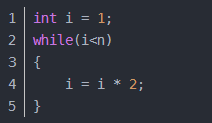
#### **2.4.2、常数阶 O(1)**

* 无论代码执行了多少行，只要是没有循环等复杂结构，那这个代码的时间复杂度就都是O(1)
* 代码在执行的时候，它消耗的时候并不随着某个变量的增长而增长，那么无论这类代码有多长，即使有几万几十万行，都可以用O(1)来表示它的时间复杂度。



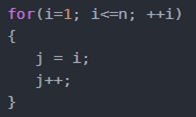
#### **2.4.3、对数阶 O(log2n)**





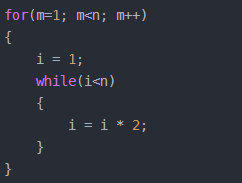
#### **2.4.4、线性阶 O(n)**

* 说明：这段代码，for循环里面的代码会执行n遍，因此它消耗的时间是随着n的变化而变化的，因此这类代码都可以用O(n)来表示它的时间复杂度



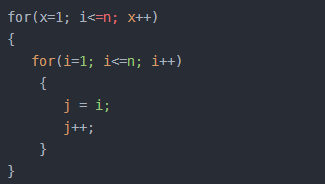
#### **2.4.5、线性对数阶 O(nlogN)**

线性对数阶O(nlogN) 其实非常容易理解，将时间复杂度为O(logn)的代码循环N遍的话，那么它的时间复杂度就是 n \* O(logN)，也就是了O(nlogN)



#### **2.4.6、平方阶 O(n²)**

平方阶O(n²) 就更容易理解了，如果把 O(n) 的代码再嵌套循环一遍，它的时间复杂度就是 O(n²)，这段代码其实就是嵌套了2层n循环，它的时间复杂度就是 O(n\*n)，即 O(n²) 如果将其中一层循环的n改成m，那它的时间复杂度就变成了 O(m\*n)



#### **2.4.7、其他阶**

* 立方阶 O(n³)、 K 次方阶 O(n^k)
* 说明： 参考上面的 O(n²) 去理解就好了， O(n³)相当于三层 n 循环， 其它的类似

### **2.5、平均和最坏时间复杂度**

* 平均时间复杂度是指所有可能的输入实例均以等概率出现的情况下， 该算法的运行时间。
* 最坏情况下的时间复杂度称最坏时间复杂度。 一般讨论的时间复杂度均是最坏情况下的时间复杂度。 这样做的原因是： 最坏情况下的时间复杂度是算法在任何输入实例上运行时间的界限， 这就保证了算法的运行时间不会比最坏情况更长。
* 平均时间复杂度和最坏时间复杂度是否一致， 和算法有关(如图)。



### **2.6、算法的空间复杂度**

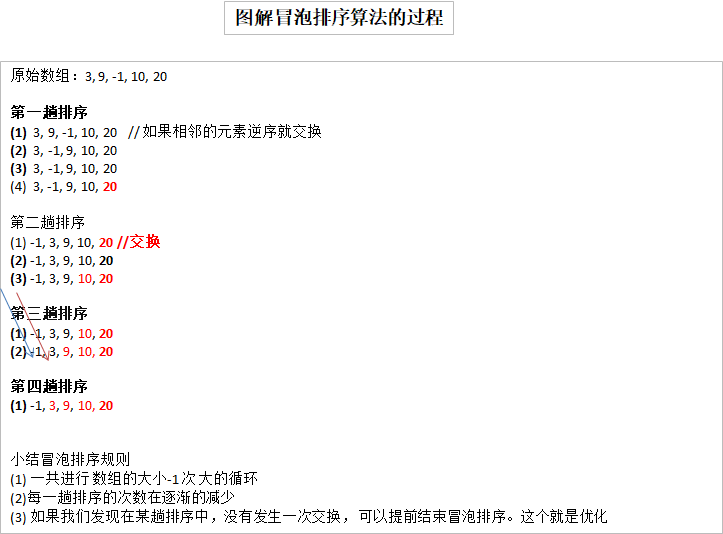
* 一个算法的空间复杂度(Space Complexity)定义为该算法所耗费的存储空间， 它也是问题规模 n 的函数
* 在做算法分析时， 主要讨论的是时间复杂度。 从用户使用体验上看， 更看重的程序执行的速度。 一些缓存产品(redis, memcache)和算法(基数排序)本质就是用空间换时间

## 3、冒泡排序

### 3.1、基本介绍

* 冒泡排序（Bubble Sorting） 的基本思想是： 通过对待排序序列从前向后（从下标较小的元素开始），依次比较相邻元素的值， 若发现逆序则交换， 使值较大的元素逐渐从前移向后部， 就象水底下的气泡一样逐渐向上冒。
* 优化：因为排序的过程中， 各元素不断接近自己的位置， 如果一趟比较下来没有进行过交换， 就说明序列有序， 因此要在排序过程中设置一个标志 flag 判断元素是否进行过交换。 从而减少不必要的比较。 (这里说的优化， 可以在冒泡排序写好后， 再进行)

### 3.2、冒泡排序图解



## **4、选择排序**

### **4.1、选择排序基本介绍**

* 选择式排序也属于内部排序法， 是从欲排序的数据中， 按指定的规则选出某一元素， 再依规定交换位置后达到排序的目的。

### **4.2、选择排序思想**

选择排序（select sorting） 也是一种简单的排序方法。 它的基本思想是（n 是数组大小）：

* 第一次从 arr[0]~arr[n-1]中选取最小值，与 arr[0] 交换
* 第二次从 arr[1]~arr[n-1]中选取最小值， 与 arr[1] 交换
* 第三次从 arr[2]~arr[n-1]中选取最小值， 与 arr[2] 交换， …，
* 第 i 次从 arr[i-1]~arr[n-1]中选取最小值， 与 arr[i-1] 交换， …，
* 第 n-1 次从 arr[n-2]~arr[n-1]中选取最小值，与 arr[n-2] 交换，
* 总共通过 n-1 次， 得到一个按排序码从小到大排列的有序序列。

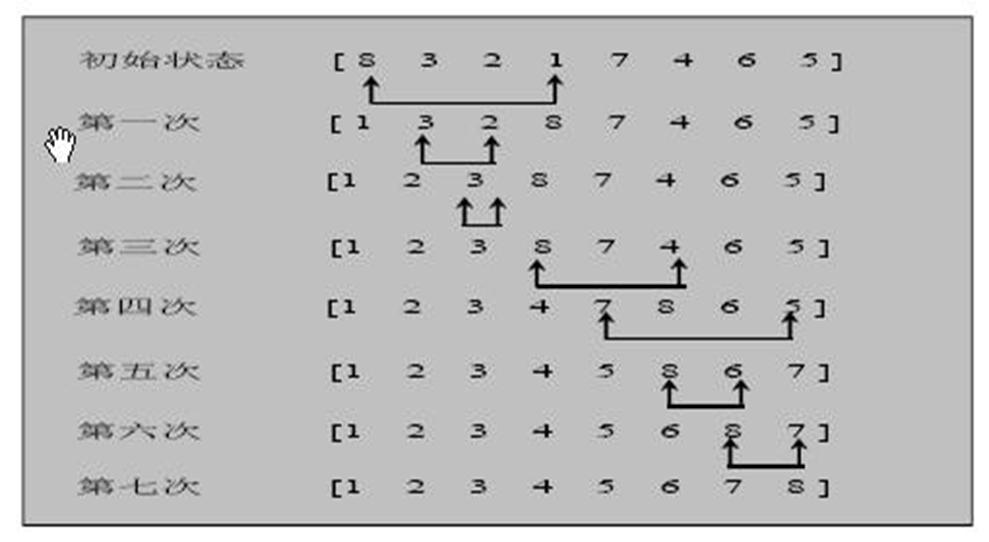
### **4.3、选择排序图解**

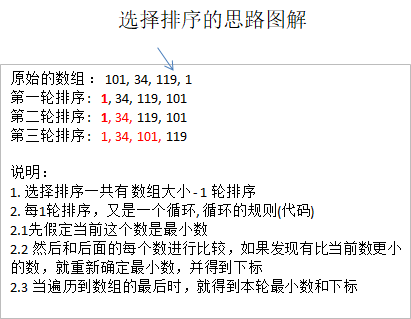
1. 选择排序流程：

* 第一次循环，默认 arr[0] 是最小的元素，将其与 arr[1]~arr[n-1] 进行比较，找到最小的元素，并与 arr[0] 的位置位置
* 第二次循环，默认 arr[1] 是最小的元素，将其与 arr[2]~arr[n-1] 进行比较，找到最小的元素，并与 arr[1] 的位置位置
* 第 i 次循环，默认 arr[i] 是最小的元素，将其与 arr[i+1]~arr[n-1] 进行比较，找到最小的元素，并与 arr[i] 的位置位置
* 直到循环执行 n - 1 次

1. 总结：两层 for 循环

* 第一层 for 循环控制走多少趟：for (int i = 0; i < arr.length - 1; i++) {
* 从数组第一个元素开始，因为每次都是拿当前元素 arr[j] 和其后一个元素 arr[j+1] 进行比较
* 到数组倒数第二个元素结束，将 arr[arr.length - 2] 与 arr[arr.length - 1] 进行比较后，数组就已经是有序数组
* 如果数组大小为 n ，那么执行完第 n - 1 趟时，数组就已经是有序数组
* 第二层 for 循环控制从第几个元素开始执行选择排序：for (int j = i + 1; j < arr.length; j++)
* 每次进入第二层 for 循环时，先假设当前元素 arr[i] 是最小的元素：min = arr[i]; ，并记录最小元素的下标：index = i;
* 然后依次和其后面的元素 arr[j] 比较，如果找到比 arr[i] 小的元素，则更新最小值和最小值的索引：min = arr[j]; index = j ;





### **4.5、总结**

* 由于选择排序算法在最内层的 for 循环中，满足 if (min > arr[j]) { 条件后，只需要记录最小值和最小值在数组中的索引，无需像冒泡排序那样每次都要执行交换操作，所以选择排序算法的执行速度比冒泡排序算法快一些

## **5、插入排序**

### **5.1、插入排序基本介绍**

插入式排序属于内部排序法， 是对于欲排序的元素以插入的方式找寻该元素的适当位置， 以达到排序的目的。

### **5.2、插入排序思想**

* 插入排序（Insertion Sorting） 的基本思想是： 把 n 个待排序的元素看成为一个有序表和一个无序表
* 开始时有序表中只包含一个元素， 无序表中包含有 n-1 个元素， 排序过程中每次从无序表中取出第一个元素， 把它的排序码依次与有序表元素的排序码进行比较， 将它插入到有序表中的适当位置， 使之成为新的有序表

### **5.3、插入排序图解**

1. 插入排序逻辑：

* 首先，将数组分为两个数组，前部分有序数组，后部分是无序数组，我们的目的就是一点一点取出无序数组中的值，将其放到有序数组中去
* 第一趟：arr[0] 作为有序数组的元素，arr[1] 作为无序数组中第一个元素，将 arr[1] 与 arr[0] 比较，目标是将 arr[1] 插入到有序数组中
* 第二趟：arr[0] 和 arr[1] 作为有序数组的元素，arr[2] 作为无序数组中第一个元素，将 arr[2] 与 arr[0] 和 arr[1] 比较，目标是将 arr[2] 插入到有序数组中
* 第 i 趟：arr[0]~arr[i] 作为有序数组的元素，arr[i+1] 作为无序数组中第一个元素，将 arr[i+1] 与 arr[0]~arr[i] 比较，目标是将 arr[i+1] 插入到有序数组中
* 第 n-1 趟：此时有序数组为 arr[0]~arr[n-2] ，无序数组为 arr[n-1] ，将无序数组中最后一个元素插入到有序数组中即可

**如何进行插入？**

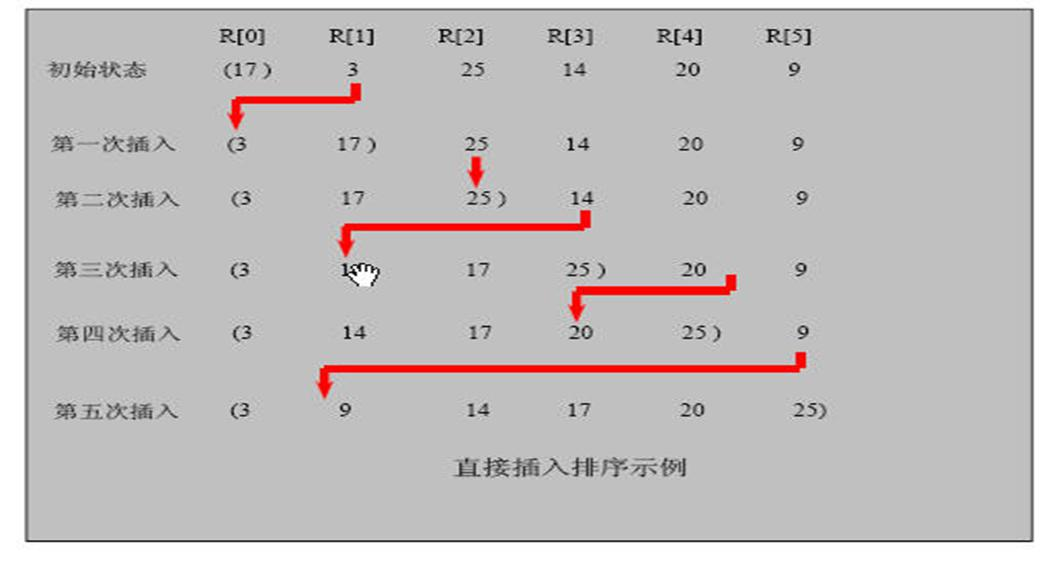
* 假设有个指针（index），指向无序数组中的第一个元素，即 arr[index] 是无序数组中的第一个元素，我们定义一个变量来存储该值：int insertVal = arr[index];，现在要将其插入到前面的有序数组中
* 将 index 前移一步，则指向有序数组最后一个元素，我们定义一个新的变量来存储该指针：insertIndex = index - 1; ，即 arr[insertIndex] 是有序数组最后一个元素
* 我们需要找到一个比 insertVal 小的值，并将 insertVal 插入在该值后面：
* 如果 insertVal > arr[insertIndex] ，执行插入
* 如果 insertVal < arr[insertIndex] ，将有序数组后移，腾出插入空间，insertIndex 指针前移，再看看前一个元素满不满足条件，直到找到插入位置
* 即循环终止条件为找到插入位置，又分为两种情况：

1. 在有序数组中间找到插入位置
2. insertVal 比有序数组中所有的数都小，插入在数组第一个位置（insertIndex = 0 的情况）

总结：两层循环

* for 循环控制走多少趟：for(int i = 1; i < arr.length; i++) { ，从数组第一个元素开始到数组最后一个元素结束
* while 循环不断将指针前移，在有序数组中寻找插入位置，并执行插入：

while (insertIndex >= 0 && insertVal < arr[insertIndex])



### **5.5、总结**

* 插入排序在寻找插入位置时，需要对数组元素进行整体挪位，所以效率比选择排序稍低

## **6、希尔排序**

### **6.1、简单插入排序问题**

* 我们看简单的插入排序可能存在的问题，数组 arr = { 2, 3, 4, 5, 6, 1 } 这时需要插入的数 1(最小)，简单插入排序的过程如下
* 结论: 当需要插入的数是较小的数时， 后移的次数明显增多， 对效率有影响

### **6.2、希尔排序基本介绍**

希尔排序是希尔（Donald Shell） 于 1959 年提出的一种排序算法。 希尔排序也是一种插入排序， 它是简单插入排序经过改进之后的一个更高效的版本， 也称为缩小增量排序。

### **6.3、希尔排序基本思想**

希尔排序按照增量将数组进行分组，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

### **6.4、希尔排序图解（交换法）**

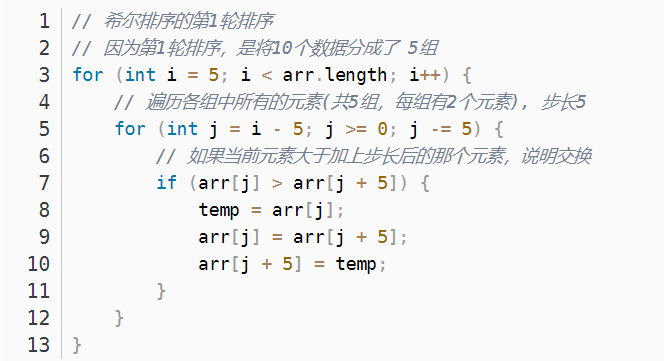
#### 第一次：gap = arr.length/5 = 5 ， 将数组分为五组，每个数组元素的索引相差 5

如何完成第一次的排序？

* + - 仔细想想，我们需要用一次循环将每组中的元素排序
    - 总共有五组，我们又需要一次循环
    - 所以完成每次排序，需要两层循环

程序代码如下，把 i ，j 都看作是辅助指针：

* i 与 j 配合使用，可以将指针从数组第一个元素，移动至最后一个元素，目的：把数组遍历一遍
* j 与 i 配合使用，每次都从数组索引 i 处往前遍历，每次向前移动 gap 个位置，然后进行交换（冒泡排序的意思）：看看前面的元素有没有比我的值大，如果前面的元素比我的值大，我就要和他交换位置，跑到前面去



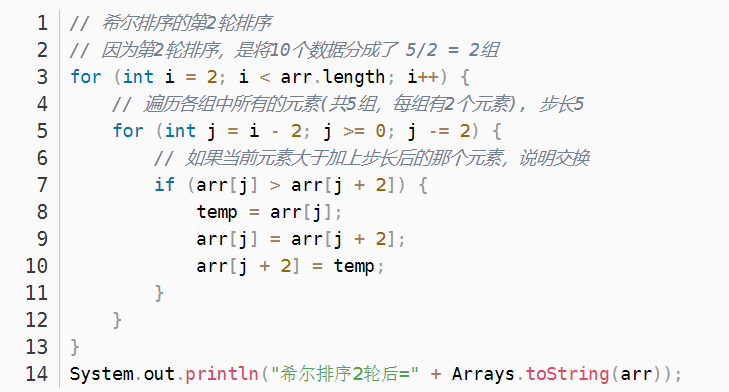
#### 第二次：gap = gap /2 = 2; ， 将数组分为两组，每个数组元素的索引相差 2

第一组：

1. i = 2 时，数组从索引 2 处往前遍历，间隔为 2 ：将 arr[0]、arr[2] 排序
2. i = 4 时，数组从索引 4 处往前遍历，间隔为 2 ：将 arr[0]、arr[2]、arr[4] 排序
3. i = 6 时，数组从索引 6 处往前遍历，间隔为 2 ：将 arr[0]、arr[2]、arr[4]、arr[6] 排序

第二组：

1. i = 3 时，数组从索引 3 处往前遍历，间隔为 2 ：将 arr[1]、arr[3] 排序
2. i = 5 时，数组从索引 5 处往前遍历，间隔为 2 ：将 arr[1]、arr[3]、arr[5] 排序
3. i = 7 时，数组从索引 7 处往前遍历，间隔为 2 ：将 arr[1]、arr[3]、arr[5]、arr[7] 排序



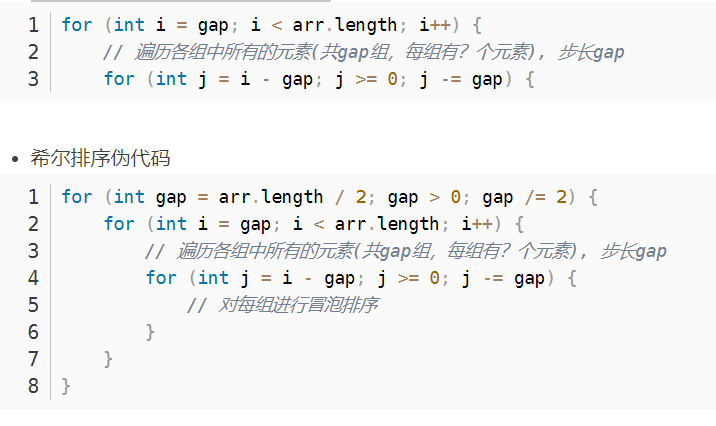
#### 第三次：gap = gap /2 = 1; ， 将数组分为一组，每个数组元素的索引相差 1 ，对于交换法而言，这就是异常冒泡排序

1. i = 1 时，数组从索引 1 处往前遍历，间隔为 1 ：将 arr[0]、arr[1] 排序
2. i = 2 时，数组从索引 2 处往前遍历，间隔为 1 ：将 arr[0]、arr[1]、arr[2] 排序
3. i = 3 时，数组从索引 3 处往前遍历，间隔为 1 ：将 arr[0]、arr[1]、arr[2]、arr[3] 排序



#### 总结：每次使用循环改变 gap 的值（初始值：数组大小/2 ，之后：gap = gap/2），然后在改变 gap 的循环中嵌套上面的双层 for 循环

* 改变 gap ：for (int gap = arr.length / 2; gap > 0; gap /= 2) {
* 内层循环：实现对每组数组的排序





分析：由于使用交换法实现希尔排序算法，所以基于交换法的希尔排序算法比简单选择排序算法更慢，所以我们一定要编写基于插入法的希尔排序算法

### 6.5、编写希尔排序（插入法）

编写基于插入法的希尔排序算法：

* 记录当前位置的元素值 int temp = arr[j]; ，从当前元素前一个位置开始，往前寻找，每次移动 gap 个距离

1. 如果 temp < arr[j - gap] ：
   1. 将数组元素后移，腾出插入空间：arr[j] = arr[j - gap];
   2. 然后继续往前找：j -= gap;
2. 如果 temp > arr[j - gap] ，找到插入位置，执行插入 arr[j] = temp; ，因为在上一步已经腾出了插入空间，并且将指针 j 前移，所以可直接插入
3. 如果 找到数组最前面还是没有找到插入位置：j - gap < 0 ，则证明 temp 需要插入在数组最前面

* 仅仅就是将之前交换法的冒泡操作替换成了插入操作

## **7、快速排序**

### **7.1、快排简介**

1. 快速排序是由东尼·霍尔所发展的一种排序算法。在平均状况下，排序 n 个项目要 Ο(nlogn) 次比较。在最坏状况下则需要 Ο(n2) 次比较，但这种状况并不常见。事实上，快速排序通常明显比其他 Ο(nlogn) 算法更快，因为它的内部循环（inner loop）可以在大部分的架构上很有效率地被实现出来。
2. 快速排序使用分治法（Divide and conquer）策略来把一个串行（list）分为两个子串行（sub-lists）。
3. **快速排序又是一种分而治之思想在排序算法上的典型应用。本质上来看，快速排序应该算是在冒泡排序基础上的递归分治法。**
4. 快速排序的名字起的是简单粗暴，因为一听到这个名字你就知道它存在的意义，就是快，而且效率高！它是处理大数据最快的排序算法之一了。
5. 虽然 Worst Case 的时间复杂度达到了 O(n²)，但是人家就是优秀，在大多数情况下都比平均时间复杂度为 O(n logn) 的排序算法表现要更好，可是这是为什么呢，我也不知道。好在我的强迫症又犯了，查了 N 多资料终于在《算法艺术与信息学竞赛》上找到了满意的答案：
6. 快速排序的最坏运行情况是 O(n²)，比如说顺序数列的快排。但它的平摊期望时间是 O(nlogn)，且 O(nlogn) 记号中隐含的常数因子很小，比复杂度稳定等于 O(nlogn) 的归并排序要小很多。所以，对绝大多数顺序性较弱的随机数列而言，快速排序总是优于归并排序。

### **7.2、代码思路**

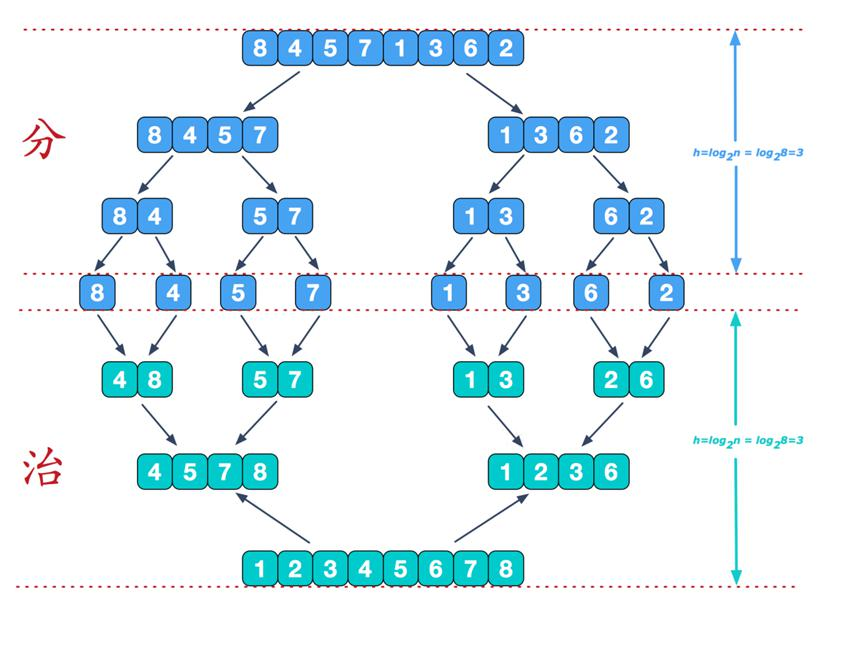
1. **从数列中挑出一个元素，称为 “基准”（pivot）;**
2. **重新排序数列，所有元素比基准值小的摆放在基准前面，所有元素比基准值大的摆在基准的后面（相同的数可以到任一边）。**
3. 在这个分区退出之后，该基准就处于数列的中间位置。这个称为分区（partition）操作；
4. 递归地（recursive）把小于基准值元素的子数列和大于基准值元素的子数列排序；

## **8、归并排序**

### **8.1、归并排序基本介绍**

* 归并排序（MERGE-SORT） 是利用归并的思想实现的排序方法， 该算法采用经典的分治（divide-and-conquer）策略
* ****分治法将问题分(divide)成一些小的问题然后递归求解， 而治(conquer)的阶段则将分的阶段得到的各答案"修补"在一起， 即分而治之****

### **8.2、归并排序思想**



### **8.3、归并排序代码思路**

* 合并时，其实是拿着原数组（arr）中两个相邻的子数组（arr1、arr2）进行合并，我们使用三个指针，来表示两个子数组在原数组中的位置

arr[left] ~ arr[mid] 为 arr1

arr[mid + 1] ~ arr[right] 为 arr2

* 如何合并？

首先，需要一个临时的 temp 数组，其大小与原数组 arr 一样

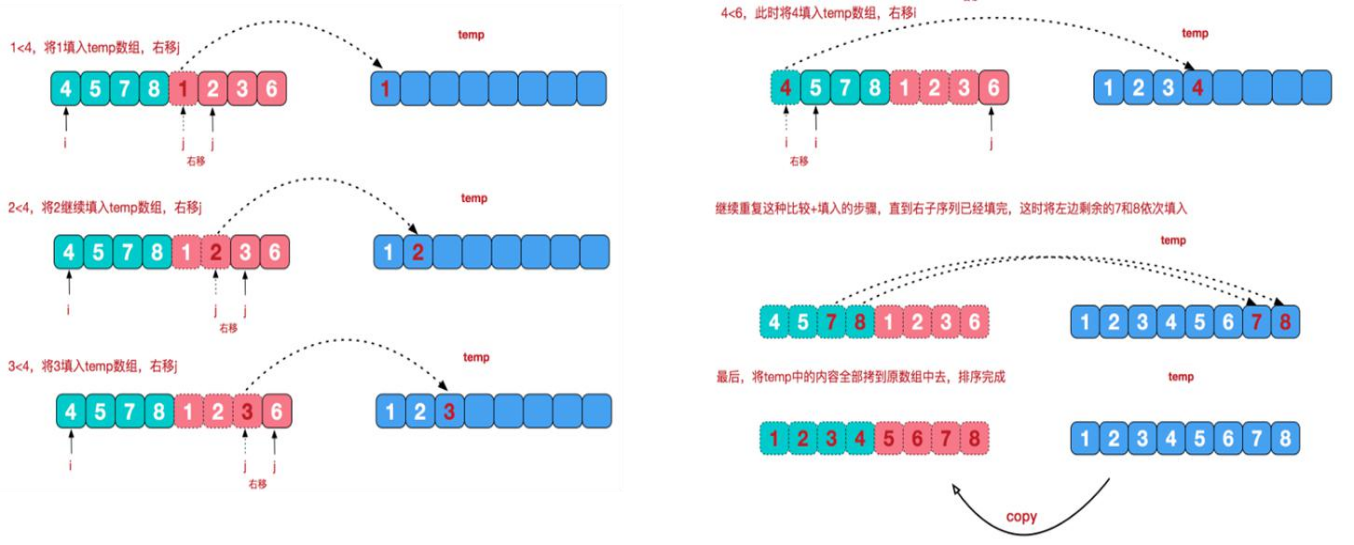
定义辅助指针 i 遍历 arr1 ，定义辅助指针 j 遍历 arr2 ，原则就是，把 arr1 和 arr2 中的数往 temp 中放，使得 temp[left] ~ temp[right] 是有序数组

最后把 temp 临时数组中的数据拷贝回原数组中（个人认为，最后一下次再拷贝回去就行。。。）

* 如何分？

向左递归拆分：mergeSort(arr, left, mid, temp);

向右递归拆分：mergeSort(arr, mid + 1, right, temp);

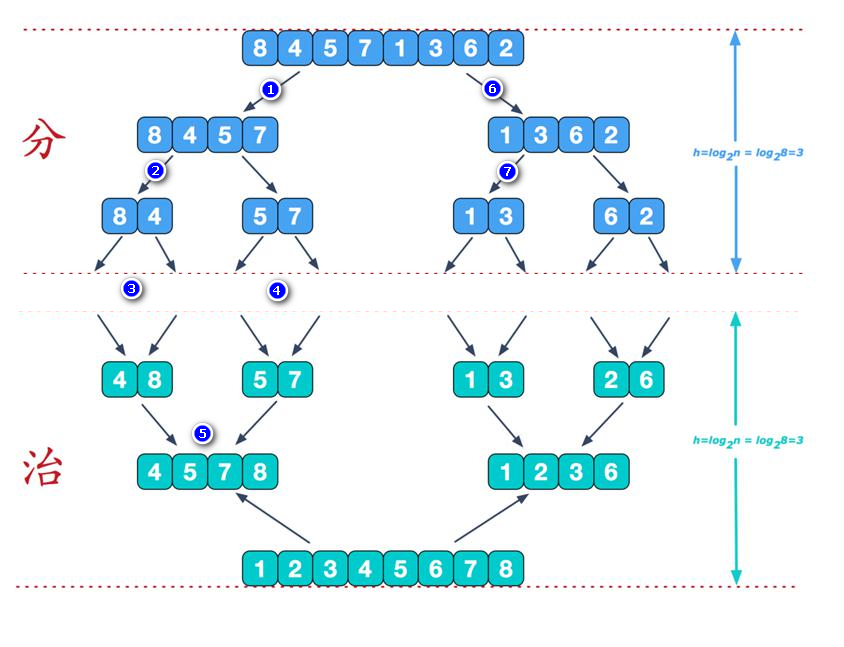


### **8.5、总结**

先将数组分为左右两半，先执行左半边递归：

1. 首先执行左递归到最深层，条件 if (left < right) 不满足，开始执行合并，合并 { 8, 4 } 到临时数组 temp 中，变为有序数组 { 4, 8 } ，再拷贝回原数组 arr 中
2. 然后执行最深层的右递归，条件 if (left < right) 不满足，开始执行合并，合并 { 5, 7 } 到临时数组 temp 中，变为有序数组 { 2, 7 } ，再拷贝回原数组 arr 中
3. 合并完后，递归回溯至上一节，开始执行合并，合并 { 4, 5, 7, 8 } 到临时数组 temp 中，变为有序数组 { 4, 5, 7, 8 } ，再拷贝回原数组 arr 中

右左半边的递归也是同样的道理



## **基数排序**

### **9.1、基数排序基本介绍**

* 基数排序（radix sort） 属于“分配式排序” （distribution sort） ， 又称“桶子法” （bucket sort） 或 bin sort， 顾名思义， 它是通过键值的各个位的值， 将要排序的元素分配至某些“桶” 中， 达到排序的作用
* 基数排序法是属于稳定性的排序， 基数排序法的是效率高的稳定性排序法
* 基数排序(Radix Sort)是桶排序的扩展
* 基数排序是 1887 年赫尔曼· 何乐礼发明的。 它是这样实现的： 将整数按位数切割成不同的数字， 然后按每个位数分别比较。

### **9.2、基数排序思想**

* 将所有待比较数值统一为同样的数位长度， 数位较短的数前面补零。
* 然后， ****从最低位开始， 依次进行一次排序。这样从最低位排序一直到最高位排序完成以后, 数列就变成一个有序序列。****

### **9.3、基数排序图解**

有 10 个桶，对应编号为 0~9

**步骤**

1. 第一步：根据原数组 arr 中每个元素的个位数，将其依次放入 0~9 号桶中（每个桶从前往后放），放置完毕后，再将桶中的数据依次取出（每个桶从前往后取），放回原数组 arr 中，这样原数组 arr 中个位数的元素就已经按照顺序排好了
2. 第二步：根据原数组 arr 中每个元素的十位数，将其依次放入 0~9 号桶中（每个桶从前往后放），放置完毕后，再将桶中的数据依次取出（每个桶从前往后取），放回原数组 arr 中，这样原数组 arr 中十位数 + 个位数的元素就已经按照顺序排好了
3. 第三步：根据原数组 arr 中每个元素的百位数，将其依次放入 0~9 号桶中（每个桶从前往后放），放置完毕后，再将桶中的数据依次取出（每个桶从前往后取），放回原数组 arr 中，这样原数组 arr 中百位数 + 十位数 + 个位数的元素就已经按照顺序排好了
4. …

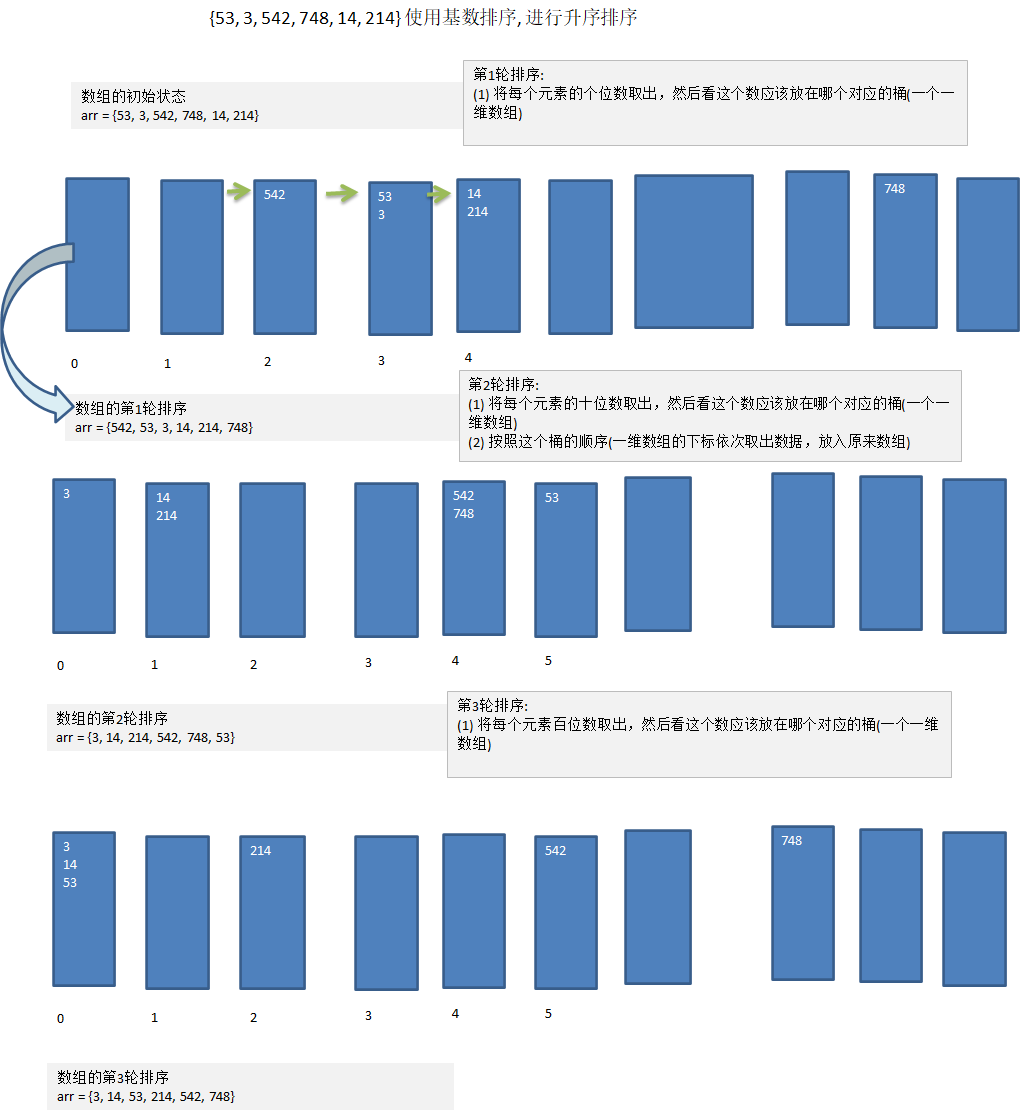
**何时排序完毕？当数组中最长位数的元素处理完毕，排序完成**

* 桶的容量如何确定？假设数组每个元素位数相同，那么单个桶最大容量即为数组容量，我们用一个二维数组来表示桶：int[][] bucket = new int[10][arr.length];
* 我们如何知道每桶中装了几个元素？这也需要记录，用一个一维数组来记录：

int[] bucketElementCounts = new int[10];

**总结：**

* 假设数组中元素的最长位数为 maxLength ，则处理完 maxLength 位数后，数组排序完毕：\*for(int i = 0 , n = 1; i < maxLength; i++, n = 10) {
* 使用一个 for 循环处理原一维数组 arr ，将其放入桶中
* for(int j = 0; j < arr.length; j++) {
* 使用两层 for 循环，处理 10 个 桶，将其中的元素放回原一维数组中
* for (int k = 0; k < bucketElementCounts.length; k++) {
* if (bucketElementCounts[k] != 0) {
* for (int l = 0; l < bucketElementCounts[k]; l++) {



### **9.5、基数排序的说明**

* 基数排序是对传统桶排序的扩展， 速度很快
* 基数排序是经典的空间换时间的方式， 占用内存很大，当对海量数据排序时， 容易造成 OutOfMemoryError 。
* 基数排序时稳定的。 [注：假定在待排序的记录序列中， 存在多个具有相同的关键字的记录， 若经过排序， 这些记录的相对次序保持不变， 即在原序列中， r[i]=r[j]， 且 r[i]在 r[j]之前， 而在排序后的序列中， r[i]仍在 r[j]之前，则称这种排序算法是稳定的； 否则称为不稳定的]

## **10、常用排序算法总结和对比**

### **10.1、排序算法的比较图**



### **10.2、相关术语解释**

稳定：如果 a 原本在 b 前面， 而 a=b， 排序之后 a 仍然在 b 的前面；

不稳定：如果 a 原本在 b 的前面， 而 a=b， 排序之后 a 可能会出现在 b 的后面；

内排序： 所有排序操作都在内存中完成；

外排序： 由于数据太大， 因此把数据放在磁盘中， 而排序通过磁盘和内存的数据传输才能进行；

时间复杂度： 一个算法执行所耗费的时间。

空间复杂度： 运行完一个程序所需内存的大小。

n: 数据规模

k: “桶” 的个数

In-place：不占用额外内存

Out-place：占用额外内存