**排序**

---------------------------------------------

术语说明

·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；

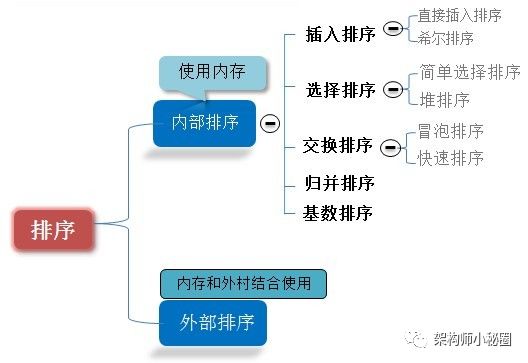
·内排序：所有排序操作都在内存中完成；

·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。





**比较和非比较的区别**

常见的快速排序、归并排序、堆排序、冒泡排序等属于比较排序。在排序的最终结果里，元素之间的次序依赖于它们之间的比较。每个数都必须和其他数进行比较，才能确定自己的位置。

在冒泡排序之类的排序中，问题规模为n，又因为需要比较n次，所以平均时间复杂度为O(n²)。在归并排序、快速排序之类的排序中，问题规模通过分治法消减为logN次，所以时间复杂度平均O(nlogn)。

比较排序的优势是，适用于各种规模的数据，也不在乎数据的分布，都能进行排序。可以说，比较排序适用于一切需要排序的情况。

计数排序、基数排序、桶排序则属于非比较排序。非比较排序是通过确定每个元素之前，应该有多少个元素来排序。针对数组arr，计算arr[i]之前有多少个元素，则唯一确定了arr[i]在排序后数组中的位置。

非比较排序只要确定每个元素之前的已有的元素个数即可，所有一次遍历即可解决。算法时间复杂度O(n)。

非比较排序时间复杂度底，但由于非比较排序需要占用空间来确定唯一位置。所以对数据规模和数据分布有一定的要求。

1. **冒泡排序（Bubble Sort）**

**--------------------------------------------------------------**

**最佳情况：T(n) = O(n) 最差情况：T(n) = O(n2) 平均情况：T(n) = O(n2)**

·比较相邻的元素。如果第一个比第二个大，就交换它们两个；

·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，这样在最后的元素应该会是最大的数；

·针对所有的元素重复以上的步骤，除了最后一个；

·重复步骤1~3，直到排序完成。

for (int i = 0; i < array.length; i++)

for (int j = 0; j < array.length - 1 - i; j++)

if (array[j + 1] < array[j]) {

int temp = array[j + 1];

array[j + 1] = array[j];

array[j] = temp;

}

1. **选择排序（Selection Sort）**

**--------------------------------------------------------------**

**最佳情况：T(n) = O(n2) 最差情况：T(n) = O(n2) 平均情况：T(n) = O(n2)**

表现最稳定的排序算法之一，因为无论什么数据进去都是O(n2)的时间复杂度，所以用到它的时候，数据规模越小越好。唯一的好处可能就是不占用额外的内存空间了吧。

**工作原理：**首先在未排序序列中找到最小（大）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（大）元素，然后放到已排序序列的末尾。以此类推，直到所有元素均排序完毕。

for (int i = 0; i < array.length; i++) {

int minIndex = i;

for (int j = i; j < array.length; j++) {

if (array[j] < array[minIndex]) //找到最小的数

minIndex = j; //将最小数的索引保存

}

int temp = array[minIndex];

array[minIndex] = array[i];

array[i] = temp;

}

1. **插入排序（Insertion Sort）**

**--------------------------------------------------------------**

**最佳情况：T(n) = O(n) 最坏情况：T(n) = O(n2) 平均情况：T(n) = O(n2)**

·从第一个元素开始，该元素可以认为已经被排序；

·取出下一个元素，在已经排序的元素序列中从后向前扫描；

·如果该元素（已排序）大于新元素，将该元素移到下一位置；

·重复步骤3，直到找到已排序的元素小于或者等于新元素的位置；

·将新元素插入到该位置后；

·重复步骤2~5。

for (int i = 0; i < array.length - 1; i++) {

current = array[i + 1];

int preIndex = i;

while (preIndex >= 0 && current < array[preIndex]) {

array[preIndex + 1] = array[preIndex];

preIndex--;

}

array[preIndex + 1] = current;

}

1. **希尔排序（Shell Sort）**

**--------------------------------------------------------------**

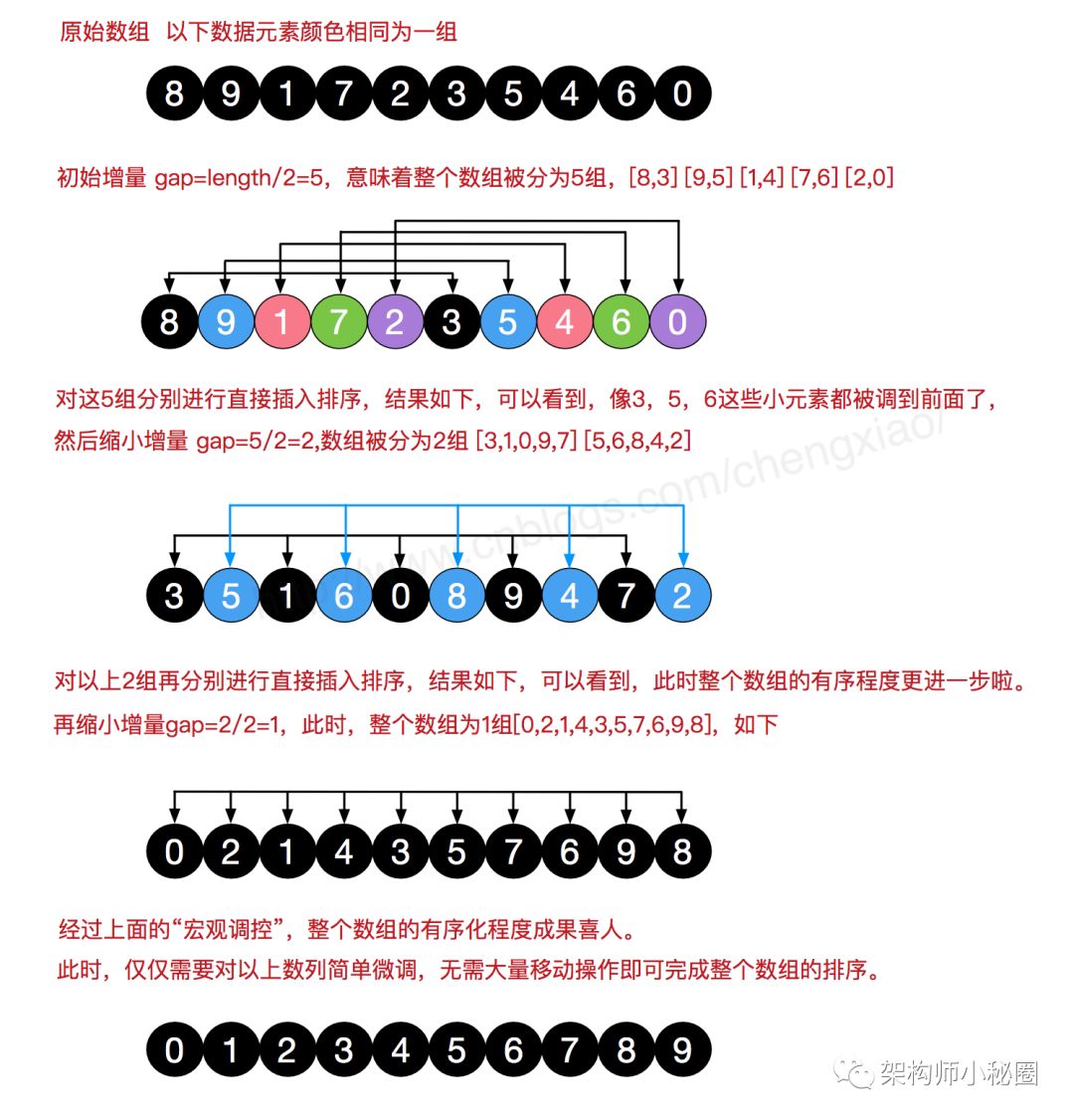
**最佳情况：T(n) = O(nlog2 n) 最坏情况：T(n) = O(nlog2 n) 平均情况：T(n) =O(nlog2n)**

先将整个待排序的记录序列分割成为若干子序列分别进行直接插入排序，具体算法描述：

·选择一个增量序列t1，t2，…，tk，其中ti>tj，tk=1；

·按增量序列个数k，对序列进行k 趟排序；

·每趟排序，根据对应的增量ti，将待排序列分割成若干长度为m 的子序列，分别对各子表进行直接插入排序。仅增量因子为1 时，整个序列作为一个表来处理，表长度即为整个序列的长度。



while (gap > 0) {

for (int i = gap; i < len; i++) {

temp = array[i];

int preIndex = i - gap;

while (preIndex >= 0 && array[preIndex] > temp) {

array[preIndex + gap] = array[preIndex];

preIndex -= gap;

}

array[preIndex + gap] = temp;

}

gap /= 2;

}

return array;

1. **归并排序（Merge Sort）**

**--------------------------------------------------------------**

**最佳情况：T(n) = O(n) 最差情况：T(n) = O(nlogn) 平均情况：T(n) = O(nlogn)**

和选择排序一样，归并排序的性能不受输入数据的影响，但表现比选择排序好的多，因为始终都是O(n log n）的时间复杂度。代价是需要额外的内存空间。

·把长度为n的输入序列分成两个长度为n/2的子序列；

·对这两个子序列分别采用归并排序；

·将两个排序好的子序列合并成一个最终的排序序列

for (int index = 0, i = 0, j = 0; index < result.length; index++) {

if (i >= left.length)

result[index] = right[j++];

else if (j >= right.length)

result[index] = left[i++];

else if (left[i] > right[j])

result[index] = right[j++];

else

result[index] = left[i++];

}

return result;

1. **快速排序（Quick Sort）**

**--------------------------------------------------------------**

**最佳情况：T(n) = O(nlogn) 最差情况：T(n) = O(n2) 平均情况：T(n) = O(nlogn)**

·从数列中挑出一个元素，称为 “基准”（pivot）；

·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）。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；

·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。

public static int partition(int[] array, int start, int end) {

int pivot = (int) (start + Math.random() \* (end - start + 1));

int smallIndex = start - 1;

swap(array, pivot, end);

for (int i = start; i <= end; i++)

if (array[i] <= array[end]) {

smallIndex++;

if (i > smallIndex)

swap(array, i, smallIndex);

}

return smallIndex;

}

public static void swap(int[] array, int i, int j) {

int temp = array[i];

array[i] = array[j];

array[j] = temp;

}

1. **堆排序（Heap Sort）**

**--------------------------------------------------------------**

**最佳情况：T(n) = O(nlogn) 最差情况：T(n) = O(nlogn) 平均情况：T(n) = O(nlogn)**

·将初始待排序关键字序列(R1,R2….Rn)构建成大顶堆，此堆为初始的无序区；

·将堆顶元素R[1]与最后一个元素R[n]交换，此时得到新的无序区(R1,R2,……Rn-1)和新的有序区(Rn),且满足R[1,2…n-1]<=R[n]；

·由于交换后新的堆顶R[1]可能违反堆的性质，因此需要对当前无序区(R1,R2,……Rn-1)调整为新堆，然后再次将R[1]与无序区最后一个元素交换，得到新的无序区(R1,R2….Rn-2)和新的有序区(Rn-1,Rn)。不断重复此过程直到有序区的元素个数为n-1，则整个排序过程完成。

public static int[] HeapSort(int[] array) {

len = array.length;

if (len < 1) return array;

//1.构建一个最大堆

buildMaxHeap(array);

//2.循环将堆首位（最大值）与末位交换，然后在重新调整最大堆

while (len > 0) {

swap(array, 0, len - 1);

len--;

adjustHeap(array, 0);

}

return array;

}

// 建立最大堆

public static void buildMaxHeap(int[] array) {

//从最后一个非叶子节点开始向上构造最大堆

for (int i = (len - 1) / 2; i >= 0; i--) {

adjustHeap(array, i);

}

}

//调整使之成为最大堆

public static void adjustHeap(int[] array, int i) {

int maxIndex = i;

//如果有左子树，且左子树大于父节点，则将最大指针指向左子树

if (i \* 2 < len && array[i \* 2] > array[maxIndex])

maxIndex = i \* 2;

//如果有右子树，且右子树大于父节点，则将最大指针指向右子树

if (i \* 2 + 1 < len && array[i \* 2 + 1] > array[maxIndex])

maxIndex = i \* 2 + 1;

//如果父节点不是最大值，则将父节点与最大值交换，并且递归调整与父节点交换的位置。

if (maxIndex != i) {

swap(array, maxIndex, i);

adjustHeap(array, maxIndex);

}

}

1. **计数排序（Counting Sort）**

**--------------------------------------------------------------**

**最佳情况：T(n) = O(n+k) 最差情况：T(n) = O(n+k) 平均情况：T(n) = O(n+k)**

计数排序(Counting sort)是一种稳定的排序算法。计数排序使用一个额外的数组C，其中第i个元素是待排序数组A中值等于i的元素的个数。然后根据数组C来将A中的元素排到正确的位置。它只能对整数进行排序。

·找出待排序的数组中最大和最小的元素；

·统计数组中每个值为i的元素出现的次数，存入数组C的第i项；

·对所有的计数累加（从C中的第一个元素开始，每一项和前一项相加）；

·反向填充目标数组：将每个元素i放在新数组的第C(i)项，每放一个元素就将C(i)减去1。

1. **桶排序（Bucket Sort）**

**--------------------------------------------------------------**

**最佳情况：T(n) = O(n+k) 最差情况：T(n) = O(n+k) 平均情况：T(n) = O(n2)**

桶排序 (Bucket sort)的工作的原理：假设输入数据服从均匀分布，将数据分到有限数量的桶里，每个桶再分别排序（有可能再使用别的排序算法或是以递归方式继续使用桶排序进行排

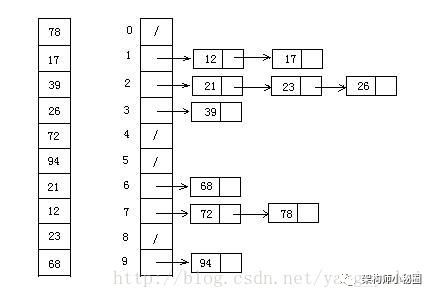
·人为设置一个BucketSize，作为每个桶所能放置多少个不同数值（例如当BucketSize==5时，该桶可以存放｛1,2,3,4,5｝这几种数字，但是容量不限，即可以存放100个3）；

·遍历输入数据，并且把数据一个一个放到对应的桶里去；

·对每个不是空的桶进行排序，可以使用其它排序方法，也可以递归使用桶排序；

·从不是空的桶里把排好序的数据拼接起来。

注意，如果递归使用桶排序为各个桶排序，则当桶数量为1时要手动减小BucketSize增加下一循环桶的数量，否则会陷入死循环，导致内存溢出。



public static ArrayList<Integer> BucketSort(ArrayList<Integer> array, int bucketSize) {

if (array == null || array.size() < 2)

return array;

int max = array.get(0), min = array.get(0);

// 找到最大值最小值

for (int i = 0; i < array.size(); i++) {

if (array.get(i) > max)

max = array.get(i);

if (array.get(i) < min)

min = array.get(i);

}

int bucketCount = (max - min) / bucketSize + 1;

ArrayList<ArrayList<Integer>> bucketArr = new ArrayList<>(bucketCount);

ArrayList<Integer> resultArr = new ArrayList<>();

for (int i = 0; i < bucketCount; i++) {

bucketArr.add(new ArrayList<Integer>());

}

for (int i = 0; i < array.size(); i++) {

bucketArr.get((array.get(i) - min) / bucketSize).add(array.get(i));

}

for (int i = 0; i < bucketCount; i++) {

if (bucketCount == 1)

bucketSize--;

ArrayList<Integer> temp = BucketSort(bucketArr.get(i), bucketSize);

for (int j = 0; j < temp.size(); j++)

resultArr.add(temp.get(j));

}

return resultArr;

}

1. **基数排序（Radix Sort）**

**--------------------------------------------------------------**

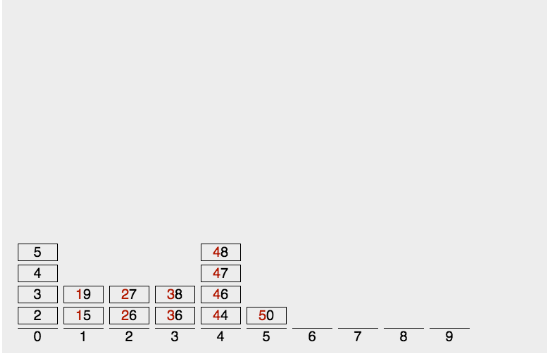
**最佳情况：T(n) = O(n \* k) 最差情况：T(n) = O(n \* k) 平均情况：T(n) = O(n \* k)**

基数排序是按照低位先排序，然后收集；再按照高位排序，然后再收集；依次类推，直到最高位。有时候有些属性是有优先级顺序的，先按低优先级排序，再按高优先级排序。最后的次序就是高优先级高的在前，高优先级相同的低优先级高的在前。基数排序基于分别排序，分别收集，所以是稳定的。

·取得数组中的最大数，并取得位数；

·arr为原始数组，从最低位开始取每个位组成radix数组；

·对radix进行计数排序（利用计数排序适用于小范围数的特点）；



基数排序有两种方法：

MSD 从高位开始进行排序 LSD 从低位开始进行排序