1、冒泡排序

步骤

1. 比较相邻元素，如果第一个比第二个大，交换两个的位置；
2. 对每一个相邻的元素做步骤1的工作，这样最后一个就是最大的元素；
3. 从头开始重复步骤1、2，最后一个除外；
4. 重复步骤1~3，直到只有一对

算法代码

```
/**
*冒泡排序
*/
    public static int[] bubbleSort(int[] array){
        if(array.length==0)
            return array;
        for(int i=0;i<array.length;i++){
            for(int j=0;j<array.length-1-i;j++){
                if(array[j+1]<arrya[j]){
                    int temp=array[j];
                    array[j]=array[j+1];
                    array[j+1]=temp;
                }
            }
        }
        return array;
    }
```

算法分析

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

2、选择排序

选择排序是最稳定的算法之一，无论什么数据都是O(n2)的时间复杂度。

步骤

1. 在未排序序列中找到最小（最大）元素，放在排序序列的起始位置；
2. 从剩余未排序的元素中找到最小（最大）元素，放到已排序序列的末尾；
3. 重复步骤2，直到排序结束

算法代码

```
/**
*选择排序
*/
    public int[] selectionSort(int[] array){
```

```
        if(array.length==0)
            return array;
        for(int i=0;i<array.length;i++){
            int minIndex=i;
            for(int j=i;j<array.length;j++){
                if(array[j]<array[minIndex]){
                    minIndex=j;
                }
            }
            int temp = array[minIndex];
            array[minIndex]=array[i];
            array[i]=temp;
        }
        return array;
    }
```

算法分析

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

3、插入排序

插入排序通过构建有序序列，对于未排序数据，从已排序序列中向前扫描，找到相应的位置并插入。需要反复把已排序的元素逐步向后挪位，为最新元素提供插入空间。

步骤

1. 将第一个元素当成已排序元素；
2. 取出下一个元素，在已排序的元素序列由后向前扫描；
3. 如果已排序的元素大于新元素，将该已排序元素移动到下一个位置；
4. 重复步骤3，直到找到已排序的元素小于等于新元素的位置；
5. 将新元素插入到该位置后；
6. 重复步骤2~5。

算法代码

```
/**
*插入排序
*/
    public int[] insertionSort(int[] array){
       if(array.length==0)
           return array;
        int current;
        for(int i=0;i<array.length-1;i++){
            current=array[i+1];
            preIndex=i;
            while(preIndex>=0&&current<array[preIndex]){
                array[preIndex+1]=array[preIndex];
                preIndex--;
            }
            array[preIndex+1]=current;
        }
        return array;
```

```
    }
```

算法分析

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

4、希尔排序

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，同时该算法是冲破O(n2）的第一批算法之一。它与插入排序的不同之处在于，它会优先比较距离较远的元素。希尔排序又叫缩小增量排序。

希尔排序是把记录按下表的一定增量分组，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。

步骤

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：

- 步骤1：选择一个增量序列t1，t2，…，tk，其中ti>tj，tk=1；
- 步骤2：按增量序列个数k，对序列进行k 趟排序；
- 步骤3：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。

算法代码

```
/**
     * 希尔排序
     *
     * @param array
     * @return
     */
    public static int[] shellSort(int[] array){
        if(array.length==0)
            return array;
        int len=array.length;
        int current,gap=len/2;
        while(gap>0){
            for(int i=gap;i<array.length;i++){
                current=array[i];
                int preIndex=i-gap;
                while(preIndex>=0&&array[preIndex]>current){
                    array[preIndex+gap]=array[preIndex];
                    preIndex-=gap;
                }
                array[preIndex+gap]=current;
            }
            gap/=2;
        }
```

```
    }
```

算法分析

- 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)

5、归并排序

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。

通过递归的方式将大的数组一直分割，直到数组的大小为 1，此时只有一个元素，那么该数组就是有序的了，之后再把两个数组大小为1的合并成一个大小为2的，再把两个大小为2的合并成4的 ..... 直到全部小的数组合并起来。

步骤

- 步骤1：把长度为n的输入序列分成两个长度为n/2的子序列；
- 步骤2：对这两个子序列分别采用归并排序；
- 步骤3：将两个排序好的子序列合并成一个最终的排序序列。

算法代码

```
public class MergeSort {

    public static int[] mergeSort(int[] arr, int left, int right) {
        if (left < right) {
            int mid = (right + left) / 2;
            arr = mergeSort(arr, left, mid);
            arr = mergeSort(arr, mid + 1, right);
            merge(arr, left, mid, right);
        }
        return arr;
    }

    public static void merge(int[] arr, int left, int mid, int right) {
        int[] a = new int[right - left + 1];
        int i = left;
        int j = mid + 1;
        int k = 0;
        while (i <= mid && j <= right) {
            if (arr[i] < arr[j])
                a[k++] = arr[i++];
            else
                a[k++] = arr[j++];
        }
        while (i <= mid) a[k++] = arr[i++];
        while (j <= right) a[k++] = arr[j++];
        for (i = 0; i < a.length; i++) {
            arr[left++] = a[i];
        }
    }
}
```

6、快速排序

我们从数组中选择一个元素，我们把这个元素称之为**中轴元素**吧，然后把数组中所有小于中轴元素的元素放在其左边，所有大于或等于中轴元素的元素放在其右边，显然，此时中轴元素所处的位置的是**有序的**。也就是说，我们无需再移动中轴元素的位置。

从中轴元素那里开始把大的数组切割成两个小的数组(两个数组都不包含中轴元素)，接着我们通过递归的方式，让中轴元素左边的数组和右边的数组也重复同样的操作，直到数组的大小为1，此时每个元素都处于**有序的位置**。

算法实现

```
import java.util.Arrays;

public class QuickSort {
    public static void main(String[] args) {
        int[] arr={3,1,2,3,5,56,65,5};
        quickSort(arr,0,arr.length-1);
        System.out.println(Arrays.toString(arr));
    }
    public static int[] quickSort(int[] arr, int left, int right) {
        if (left < right) {
            int mid = partition(arr, left, right);
            arr = quickSort(arr, left, mid - 1);
            arr = quickSort(arr, mid + 1, right);
        }
        return arr;
    }

    public static int partition(int[] arr, int left, int right) {
        int pivot = arr[left];
        int i = left;
        int j = right;
        while (i < j) {
            while (arr[j] >= pivot && i < j) j--;
            while (arr[i] <= pivot && i < j) i++;
            int temp = arr[i];
            arr[i] = arr[j];
            arr[j] = temp;
        }
        arr[left] = arr[j];
        arr[j] = pivot;
        return j;
    }

}
```

算法分析：

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

7、堆排序

堆排序就是将堆顶元素与最后一个元素交换后，剩余元素再次转变为一个新的堆，然后再依次将堆顶和第二、第三...个元素进行交换，直到剩余一个元素。

步骤

- 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。

算法实现

```
public class HeapSort {
    public static void main(String[] args) {
        int[] arr = {5, 1, 7, 3, 1, 6, 9, 4};
//        int[] arr = {16, 7, 3, 20, 17, 8};

        heapSort(arr);

        for (int i : arr) {
            System.out.print(i + " ");
        }
    }

    public static void heapSort(int[] arr) {
        for (int i = (arr.length - 1) / 2; i >= 0; i--) {
            adjustHeap(arr, i, arr.length);
        }
        for (int i = arr.length - 1; i > 0; i--) {
            int temp = arr[i];
            arr[i] = arr[0];
            arr[0] = temp;
            adjustHeap(arr, 0, i);
        }
    }

    public static void adjustHeap(int[] arr, int parent, int length) {
        int temp = arr[parent];
        int child = 2 * parent + 1;
        while (child < length) {
            if (child + 1 < length && arr[child + 1] > arr[child]) {
                child++;
            }
            if (temp >= arr[child]) {
                break;
            }
            arr[parent] = arr[child];
            parent = child;
            child = 2 * child + 1;
        }
        arr[parent] = temp;
    }
```

```
}
```

算法分析

- 时间复杂度：O(nlogn)
- 空间复杂度：O(1)
- 非稳定排序
- 原地排序

8、计数排序

计数排序是一种适合于最大值和最小值的差值不是不是很大的排序。

基本思想：就是把数组元素作为数组的下标，然后用一个临时数组统计该元素出现的次数，例如 temp[i] = m, 表示元素 i 一共出现了 m 次。最后再把临时数组统计的数据从小到大汇总起来，此时汇总起来是数据是有序的。

代码实现

```
public class CountSort {
    public static void main(String[] args) {
        int[] arr = {5, 1, 7, 3, 1, 6, 9, 4};
//        int[] arr = {16, 7, 3, 20, 17, 8};

        arr=countSort(arr);

        for (int i : arr) {
            System.out.print(i + " ");
        }
    }
    public static int[] countSort(int[] arr) {
        if (arr == null && arr.length < 2) {
            return arr;
        }
        int length = arr.length;
        int min = arr[0];
        int max = arr[0];
        for (int i = 0; i < length; i++) {
            if (arr[i] > max)
                max = arr[i];
            if (arr[i] < min)
                min = arr[i];

        }
        int d =max-min+1;
        int[] temp=new int[d];
        for (int i = 0; i < length; i++) {
           temp[arr[i]-min]++;
        }
        int k=0;
        for(int i=0;i<temp.length;i++){
            for (int j=temp[i];j>0;j--){
                arr[k++]=i+min;
            }
        }
```

```
        return arr;
    }
}
```

9、基数排序

先将数组按个位数将数组元素放在对应的十个整数桶（比如个位数1的放在1的桶）中，然后再依次取出，然后依次按十位、百位放入后取出，则可实现排序。

代码实现

```
import java.util.Arrays;

/**
 * 基数排序
 */

public class RadixSort {
    public static void main(String[] args) {
//        int[] arr = {1, 3, 5, 7, 8, 4, 6, 7, 8};
        int[] arr = {16, 7, 3, 20, 17, 8};
        radixSort(arr);
        System.out.println(Arrays.toString(arr));
    }

    public static void radixSort(int[] arr) {
        int max = arr[0];
        for (int i : arr) {
            if (max < i) {
                max = i;
            }
        }
        for (int i = 1; max / i > 0; i *= 10) {
            countingSort(arr, i);

        }
    }

    public static void countingSort(int[] arr, int exp) {
        int[] bucket = new int[10];
        for (int i = 0; i < arr.length; i++) {
            bucket[(arr[i] / exp) % 10]++;
        }
        for (int i = 1; i < bucket.length; i++) {
            bucket[i] += bucket[i - 1];
        }
        int[] temp = new int[arr.length];
        for (int i = arr.length - 1; i >= 0; i--) {
            temp[bucket[(arr[i] / exp) % 10] - 1] = arr[i];
            bucket[(arr[i] / exp) % 10]--;
        }
        for (int i = 0; i < temp.length; i++) {
            arr[i] = temp[i];
        }
```

```
    }
}
```