类别	排序方法	时间复杂度			空间复杂度	稳定性
		平均情况	最好情况	最坏情况	辅助存储	
插入排序	直接插入	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$	稳定
	Shell排序	$O(n^{1.3})$	$O(n)$	$O(n^2)$	$O(1)$	不稳定
选择排序	直接选择	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	不稳定
	堆排序	$O(nlog_2n)$	$O(nlog_2n)$	$O(nlog_2n)$	$O(1)$	不稳定
交换排序	冒泡排序	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$	稳定
	快速排序	$O(nlog_2n)$	$O(nlog_2n)$	$O(n^2)$	$O(nlog_2n)$	不稳定
归并排序		$O(nlog_2n)$	$O(nlog_2n)$	$O(nlog_2n)$	$O(n)$	稳定
基数排序		$O(d(r+n))$	$O(d(n+rd))$	$O(d(r+n))$	$O(rd+n)$	稳定

注：基数排序的复杂度中，r代表关键字的基数，d代表长度，n代表关键字的个数。

归并、直接插入、冒泡排序是稳定的

1. 冒泡排序（从大到小）

算法原理：

S1：从待排序序列的起始位置开始，从前往后依次比较各个位置和第一个位置比较。

S2：如果当前位置的值大于第一个位置，就把他们两个的值交换。

```
    public static void bubbleSort(int [] array) {
        int len = array.length;
        for(int i = 0; i < len; i ++) {
            for(int j = i + 1; j < len; j++) {
                if(array[i] <= array[j]) {
                    int tmp = array[i];
                    array[i] = array[j];
                    array[j] = tmp;
                }
            }
        }
    }
```

2. 选择排序

选择排序是冒泡排序算法的一种改进，基本思想：第一趟，在待排序记录r[1]~r[n]中选出最小的记录，将它与r[1]交换；第二趟，在待排序记录r[2]~r[n]中选出最小的记录，将它与r[2]进行交换，以此类推。

```
public static void select_Sort(int [] array) {
    int tmp;
    for (int i = 0; i < array.length; i++) {
        int index = i;
        for(int j = i + 1; j < array.length; j++) {
            if(array[index] > array[j]) {
                index = j;
            }
        }
        if(index != i) {
            tmp = array[index];
            array[index] = array[i];
            array[i] = tmp;
        }
    }
}
```

3. 快速排序

思想：通过每一趟快速排序，将要排序的数据分割为独立的两部分，找出一个中心数据，其中一部分的所有数据都要比这个小，另一部分所有数据都要比这个数据要大，然后按照此方法对这两部分数据进行快速排序，整个排序过程可以递归进行，以此实现整个数据变成有序序列。

```
public static void quickSort(int [] array, int left, int right) {
    if(left < right) {
        int pivot = array[left];
        int low = left;
        int high = right;
        while (low < high) {
            while(low < high && array[high] >= pivot) {
                high --;
            }
            array[low] = array[high];
            while(low < high && array[low] <= pivot) {
                low ++;
            }
            array[high] = array[low];
        }
        array[low] = pivot;
        quickSort(array, left, low-1);
        quickSort(array, low + 1, right);
    }
}
```

4. 直接插入排序

思想：插入排序的基本方法是每步将一个待排序序列按数据大小插到前面已经排好的序列中的适当位置，直到全部数据插入完毕为止。

1. 将这个序列的第一个元素R0视为一个有序序列。

2. 依次把后面的元素插入到前面来。

```
public static void insert_sort(int [] array) {
    for (int i = 1; i < array.length; i++) {
        if(array[i] < array[i -1]) {
            int tmp = array[i];
            int j;
            for(j = i - 1; j >=0 && tmp < array[j]; j--) {
                array[j + 1] = array[j];
            }
            array[j+1] = tmp;
        }
    }
}
```

5. 希尔排序

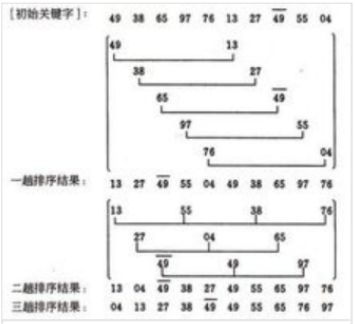
先取一个小于n的整数d1作为第一个增量，把要分的纪录分组。所有距离为d1的倍数的数放在同一个组中。先在各组内进行直接插入排序；然后，取第二个增量d2<d1重复上述的分组和排序。直至d = 1.

假设待排序文件有10个记录，其关键字分别是：

49，38，65，97，76，13，27，49，55，04。

增量序列的取值依次为：

5，3，1



希尔排序实例

```
public static void shell_sort(int [] array) {

    for(int increment = array.length / 2; increment > 0; increment /= 2) {
        //下面为直接插入排序的部分，只是在直接插入排序中increment = 1
        for(int i = increment; i < array.length; i ++) {
            if(array[i] < array[i - increment]) {
             int tmp = array[i];
             int j;
             for(j = i - increment; j >= 0 && tmp < array[j]; j = j -increment) {
                 array[j + increment] = array[j];
                }
                array[j + increment] = tmp;
            }
        }
    }


}
```

6. 归并排序

在算法中分治算法中，每次将整体数据分成两部分，这样就可以利用二分法来实现排序。

```
  public static void merge_sort(int [] array, int low, int high) {
      int mid = (high + low) / 2;
      if(low < high) {//二分法，low到mid是一组，mid+1到high是一组
          merge_sort(array, mid + 1, high);
          merge_sort(array, low, mid);
          merge(array, low, mid, high);
      }
  }

  public static void merge(int [] array, int low, int mid, int high) {
      int [] tmp = new int [high - low + 1];
      int i = low;
      int j = mid + 1;
      int k = 0;
      while(i <= mid && j <= high) { //归并原则
          if(array[i] < array[j]){
              tmp[k++] = array[i++];
          }else {
              tmp[k++] = array[j++];
          }
      }
      while(i <= mid) { //把左边剩余的数移入数组中
          tmp[k++] = array[i++];
      }
      while(j <= high) {//把右边剩余的数移入数组中
          tmp[k++] = array[j++];
      }
      for(int x=0;x<tmp.length;x++){ // 把新数组中的数覆盖nums数组
          array[x+low] = tmp[x];
      }
  }
```

7. 堆排序

```java
public class Heap_sort {
    public static void heapSort(int [] array, int n) {
        int lastIndex = n - 1;
        buildMaxHeap(array, lastIndex);  //建立大顶堆
        while (lastIndex > 0) {
            swap(array, 0, lastIndex); //对一次调整后，将根节点与最后一个节点进行互换
            if(--lastIndex == 0)    //只剩一个元素时，不需要调整堆，排序结束
               break;
            adjustHeap(array, 0, lastIndex);//在上述步骤的基础上，继续调整，得到剩下的序列。
        }
    }

    public static void buildMaxHeap(int [] array, int lastIndex) {
        int j = (lastIndex - 1) / 2; //第一个非叶子节点开始
        while (j >= 0) {
            int rootIndex = j;
            adjustHeap(array, rootIndex, lastIndex); //对每个非叶子节点进行调整。
            j--;
        }
    }

    public static void adjustHeap(int [] array, int rootIndex, int lastIndex) {//从根节点开始往下调整
        int biggerIndex = rootIndex;
        int leftChildIndex = rootIndex * 2 + 1;
        int rightChildIndex = rootIndex * 2 + 2;
        if(rightChildIndex <= lastIndex) {//如果右孩子存在，则左孩子一定存在
            if(array[rightChildIndex] > array[rootIndex] || array[leftChildIndex] > array[rootIndex]) {
                biggerIndex = array[rightChildIndex] > array[leftChildIndex]?rightChildIndex:leftChildIndex;
            }
        }else if (leftChildIndex <= lastIndex) {//保证左孩子存在，且不能越界。
            if(array[leftChildIndex] > array[rootIndex]) {
                biggerIndex = leftChildIndex;
            }
        }
        if (biggerIndex != rootIndex) {
            swap(array, biggerIndex, rootIndex);    //交换元素
            adjustHeap(array, biggerIndex, lastIndex); //对每一个非叶子节点都要循环调整。
        }
    }


    public static void swap(int [] array, int biggerIndex, int rootIndex) {
        int temp = array[rootIndex];
        array[rootIndex] = array[biggerIndex];
        array[biggerIndex] = temp;
    }

    public static void main(String [] args) {
          int [] array = new int [] {9,7,4,5,2,1,3,8,6,0};
          heapSort(array,10);
          for(int i = 0; i < array.length; i++) {
              System.out.println(array[i]);
          }
    }
}
```

堆排序

堆排序求升序用大顶堆，求降序用小顶堆。

本例用求降序的小顶堆来解析。

堆排序步骤如下：

1、我们将数据（49、38、65、97、76、13、27、50）建立一个数组$arr；

2、用数组$arr建立一个小顶堆（主要步骤，会在代码注释里解释，下图是用一个数组建立小顶堆的过程）；

3、将堆的根（最小的元素）与最后一个叶子交换，并将堆长度减一，跳到第二步；

4、重复2-3步，直到堆中只有一个结点，排序完成。