

排序

```
public class Sorts {  
    // 冒泡排序  
    // 冒泡排序是稳定的原地排序， 主要思想是两个相邻的数字进行比较交换  
    // 时间复杂度:  $O(n^2)$ , 空间复杂度:  $O(1)$   
    public static void bubbleSort(int[] data){  
  
        if(data.length<=1) return ;  
  
        for(int i=0; i<data.length; i++){  
            boolean flag = false;  
            for(int j=0; j<data.length-i-1; j++){  
                if(data[j]>data[j+1]){  
                    int tmp = data[j];  
                    data[j] = data[j+1];  
                    data[j+1] = tmp;  
                    flag = true;  
                }  
            }  
            if(!flag) break;  
        }  
    }  
  
    // 插入排序  
    // 插入排序是稳定的原地排序， 主要的思想是选取第一个数字为基准点， 按大小往基准有序区进行插入  
    // 时间复杂度:  $O(n^2)$ , 空间复杂度:  $O(1)$   
    public static void insertionSort(int[] data){  
  
        if(data.length<=1) return;  
  
        for(int i=1; i<data.length; i++){  
            int value = data[i];  
  
            int j= i-1;  
  
            for( ; j>=0; j--){
```

```

        if(data[j]>value){
            data[j+1] =data[j];
        }else break;
    }
    data[j+1] = value;
}
}

```

// 选择排序

// 选择排序是不稳定的原地排序， 主要思想是查找为排序区中的最小值进行交换

// 时间复杂度: $O(n^2)$, 空间复杂度: $O(1)$

```

public static void selectionSort(int[] data){

```

```

    if(data.length <=1) return;

```

```

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

```

```

        for(int j=i+1 ; j<data.length; j++){
            if(data[minIndex] > data[j]){
                minIndex = j;
            }
        }

```

```

    }
    int tmp = data[minIndex];
    data[minIndex] = data[i];
    data[i] = tmp;
}
}

```

// 归并排序

// 归并排序是稳定的非原地排序， 主要是思想是用分治和递归方法把数组分为多个数组， 对长度小的数组进行排序， 最后进行合并

// 时间复杂度: $O(n\log n)$, 空间复杂度: $O(n)$

```

public static void mergeSort(int[] data, int left, int right){

```

```

    if(left>=right) return ;

```

```

    int q = (right + left)/2;
    mergeSort(data,left,q);

```

```

mergeSort(data,q+1, right);
merge(data,left,q,right);
}

public static void merge(int[] data, int left, int q, int right){
    int[] leftArray = new int[q-left+2];
    int[] rightArray = new int[right-q+1];

    // 把数据导入到左
    for(int i=0; i<q-left+1; i++){
        leftArray[i] = data[left+i];
    }
    // 在左数组中设置最大值的哨兵
    leftArray[q-left+1] = Integer.MAX_VALUE;

    // 把数组导入到右数组中
    for(int i=0; i<right-q; i++){
        rightArray[i] = data[q+i+1];
    }
    // 在右数组中设置最大值的哨兵
    rightArray[right-q] = Integer.MAX_VALUE;

    int i = 0;
    int j = 0;
    int k = left;
    while(k<=right){
        // 当左边数组到达哨兵值时, i不再增加, 直到右边数组读取完剩余值, 同理右边
        // 数组也一样
        if(leftArray[i]<=rightArray[j]){
            data[k++] = leftArray[i++];
        }else{
            data[k++] = rightArray[j++];
        }
    }
}

// 快速排序, data是数组
public static void quickSort(int[] data){
    quickSortInterval(data, 0, data.length-1);
}

```

```

}
// 快速排序递归函数, p,r为下标
public static void quickSortInterval(int[] data, int p, int r){
    if(p >= r) return;

    int q = partition(data, p, r); // 获取分区点
    quickSortInterval(data, p, q-1);
    quickSortInterval(data, q+1, r);
}

public static int partition(int[] data, int p, int r){
    // 基准值
    int pivot = data[r];
    int i = p;

    for(int j=p; j<r; j++){
        if( data[j] < pivot){
            if(i == j){
                i++;
            }else{
                int tmp = data[i];
                data[i++] = data[j];
                data[j] = tmp;
            }
        }
    }

    int tmp = data[r];
    data[r] = data[i];
    data[i] = tmp;

    return i;
}
}

```