数据结构和算法（一）冒泡与选择排序

- 注:本篇文章会持续更新

介绍

- 注：里面内容部分图片跟描述引用自该公众号

排序算法可以分为内部排序和外部排序，内部排序是数据记录在内存中进行排序，而外部排序是因排序的数据很大，一次不能容纳全部的排序记录，在排序过程中需要访问外存。常见的内部排序算法有：插入排序、希尔排序、选择排序、冒泡排序、归并排序、快速排序、堆排序、基数排序等。用一张图概括:

排序算法	平均时间复杂度	最好情况	最坏情况	空间复杂度	排序方式	稳定性
冒泡排序	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$	In-place	稳定
选择排序	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	In-place	不稳定
插入排序	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$	In-place	稳定
希尔排序	$O(n \log n)$	$O(n \log^2 n)$	$O(n \log^2 n)$	$O(1)$	In-place	不稳定
归并排序	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$	Out-place	稳定
快速排序	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	$O(\log n)$	In-place	不稳定
堆排序	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(1)$	In-place	不稳定
计数排序	$O(n + k)$	$O(n + k)$	$O(n + k)$	$O(k)$	Out-place	稳定
桶排序	$O(n + k)$	$O(n + k)$	$O(n^2)$	$O(n + k)$	Out-place	稳定
基数排序	$O(n \times k)$	$O(n \times k)$	$O(n \times k)$	$O(n + k)$	Out-place	稳定

关于时间复杂度：

1. 平方阶 (O(n2)) 排序 各类简单排序：直接插入、直接选择和冒泡排序。
2. 线性对数阶 (O(nlog2n)) 排序 快速排序、堆排序和归并排序。
3. O(n1+§)) 排序，§ 是介于 0 和 1 之间的常数。 希尔排序。
4. 线性阶 (O(n)) 排序 基数排序，此外还有桶、箱排序。

关于稳定性：

稳定的排序算法：冒泡排序、插入排序、归并排序和基数排序。

不是稳定的排序算法：选择排序、快速排序、希尔排序、堆排序。

名词解释：

n：数据规模

k：“桶”的个数

In-place：占用常数内存，不占用额外内存

Out-place：占用额外内存

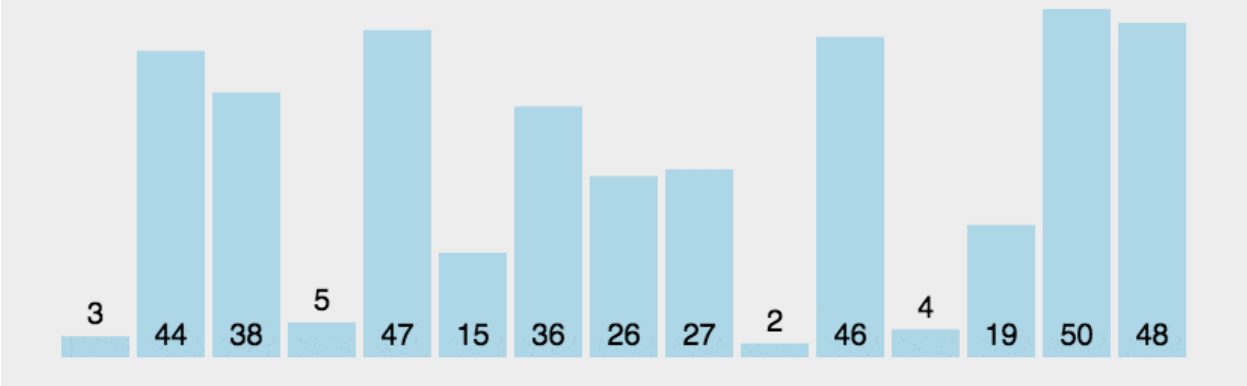
稳定性：排序后 2 个相等键值的顺序和排序之前它们的顺序相同

冒泡排序算法

冒泡排序（Bubble Sort）也是一种简单直观的排序算法。它重复地走访过要排序的数列，一次比较两个元素，如果他们的顺序错误就把他们交换过来。走访数列的工作是重复地进行直到没有再需要交换，也就是说该数列已经排序完成。这个算法的名字由来是因为越小的元素会经由交换慢慢“浮”到数列的顶端。

作为最简单的排序算法之一，冒泡排序给我的感觉就像 Abandon 在单词书里出现的感觉一样，每次都在第一页第一位，所以最熟悉。冒泡排序还有一种优化算法，就是立一个 flag，当在一趟序列遍历中元素没有发生交换，则证明该序列已经有序。但这种改进对于提升性能来说并没有什么太大作用。

- 动画演示



- 思想原理

1. 比较相邻的元素。如果第一个比第二个大，就交换他们两个。
2. 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。这步做完后，最后的元素会是最大的数。
3. 针对所有的元素重复以上的步骤，除了最后一个。
4. 持续每次对越来越少的元素重复上面的步骤，直到没有任何一对数字需要比较。

- 代码实现

```
public static void bubbleSort(int[] array){
    //3 1 5 8 2 9 4 6 7    n*(n-1)/2    n
    for(int i=array.length-1;i>0;i--) {
        boolean flag=true;
        for (int j = 0; j < i; j++) {
            if (array[j] > array[j + 1]) {
                int temp = array[j];
                array[j] = array[j + 1];
                array[j + 1] = temp;
                flag=false;
            }
        }

        if(flag){
```

```
                break;
            }
        }
    }
```

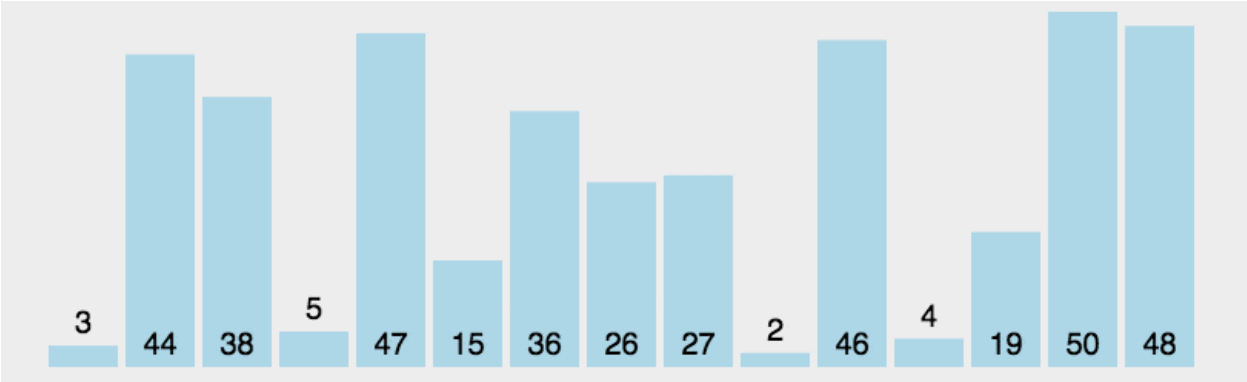
- 使用场景
  - 数据量足够小，比如斗牛游戏的牌面排序

选择排序算法

- 简介

  选择排序是一种简单直观的排序算法，无论什么数据进去都是 O(n^2) 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。

- 动画演示



- 算法步骤

  1. 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
  2. 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。
  3. 重复第二步，直到所有元素均排序完毕。

- 代码实现

```
/**
*选择排序
*/
public void selectSort(int[] arr) {
    for (int j = 0; j < arr.length - 1; j++) {
        //定义一个最小值
        int min = j;
        for (int i = j + 1; i < arr.length; i++) {
            if (arr[j] > arr[i]) {
                min = arr[i];
            }
        }
        //如果第一位已经是最小值了就不用替换了 一定保证程序达到最优;
        if (min != j) {
            int temp = arr[min];
            arr[min] = arr[j];
            arr[j] = temp;
        }
```

```
    }
}
```

- 使用场景

  快速排序的基础

参考例子

利用蛮力法给牌进行排序(冒泡排序)

- 编写 卡片 > 牌 数据 model

```
/**
* 牌的数据 Bean
*
*/
public class Cards implements Comparable{
    public int pokerColors;//花色
    public int cardPoints;//点数

    public Cards(int pokerColors, int cardPoints) {
        this.pokerColors = pokerColors;
        this.cardPoints = cardPoints;
    }
    //提供一个方法，用来比较对象的大小
    @Override
    public int compareTo(@NonNull Object o) {
        Cards c=(Cards)o;
        if(this.cardPoints>c.cardPoints){
            return 1;
        }else if(this.cardPoints<c.cardPoints){
            return -1;
        }
        if(this.pokerColors>c.pokerColors){
            return 1;
        }else if(this.pokerColors<c.pokerColors){
            return -1;
        }
        return 0;
    }

    @Override
    public String toString() {
        return "Cards{" +
                "pokerColors=" + pokerColors +
                ", cardPoints=" + cardPoints +
                '}';
    }


}
```

- 卡片进行排序

```
public void testCards() {
    Cards [] cards = {new Cards(3,9),new Cards(1,10),new Cards(2,6)};
    for (int i = cards.length - 1; i > 0; i--) {
        for (int j = 0; j < i; j++) {
            if (cards[j].compareTo(cards[j+ 1] ) > 0 ){
                Cards temp =cards[j];
                cards[j] = cards[j+1];
                cards[j+1] = temp;
            }
        }
    }
}
```