# 一、简单排序

在我们的程序中，排序是非常常见的一种需求，提供一些数据元素，把这些数据元素按照一定的规则进行排序。比如查询一些订单，按照订单的日期进行排序；再比如查询一些商品，按照商品的价格进行排序等等。所以，接下来我们要学习一些常见的排序算法。

在java的开发工具包jdk中，已经给我们提供了很多数据结构与算法的实现，比如List，Set，Map，Math等等，都是以API的方式提供，这种方式的好处在于一次编写，多处使用。我们借鉴jdk的方式，也把算法封装到某个类中，那如果是这样，在我们写java代码之前，就需要先进行API的设计，设计好之后，再对这些API进行实现。

就比如我们先设计一套API如下：

| 类名 | ArrayList |
| --- | --- |
| 构造方法 | ArrayList()：创建ArrayList对象 |
| 成员方法 | 1.boolean add(E e)：向集合中添加元素<br>2.E remove(int index):从集合中删除指定的元素 |

然后再使用java代码去实现它。以后我们讲任何数据结构与算法都是以这种方式讲解

## 1.1 Comparable接口介绍

由于我们这里要讲排序，所以肯定会在元素之间进行比较，而Java提供了一个接口Comparable就是用来定义排序规则的，在这里我们以案例的形式对Comparable接口做一个简单的回顾。

**需求：**

1.定义一个学生类Student，具有年龄age和姓名username两个属性，并通过Comparable接口提供比较规则；

2.定义测试类Test，在测试类Test中定义测试方法Comparable getMax(Comparable c1,Comparable c2)完成测试

```
//学生类
public class Student implements Comparable<Student>{

    private String username;
    private int age;

    public String getUsername() {
        return username;
    }

    public void setUsername(String username) {
        this.username = username;
    }

    public int getAge() {
        return age;
    }
```

```
18
19      public void setAge(int age) {
20          this.age = age;
21      }
22
23      @Override
24      public String toString() {
25          return "Student{" +
26                  "username='" + username + '\'' +
27                  ", age=" + age +
28                  '}';
29      }
30      //定义比较规则
31      @Override
32      public int compareTo(Student o) {
33          return this.getAge()-o.getAge();
34      }
35  }
36
37  //测试类
38  public class Test {
39      public static void main(String[] args) {
40
41          Student stu1 = new Student();
42          stu1.setUsername("zhangsan");
43          stu1.setAge(17);
44
45          Student stu2 = new Student();
46          stu2.setUsername("lisi");
47          stu2.setAge(19);
48
49          Comparable max = getMax(stu1, stu2);
50          System.out.println(max);
51      }
52      //测试方法，获取两个元素中的较大值
53      public static Comparable getMax(Comparable c1,Comparable c2){
54          int cmp = c1.compareTo(c2);
55          if (cmp>=0){
56              return c1;
57          }else{
58              return c2;
59          }
60      }
61  }
```

## 1.2 冒泡排序

冒泡排序（Bubble Sort），是一种计算机科学领域的较简单的排序算法。

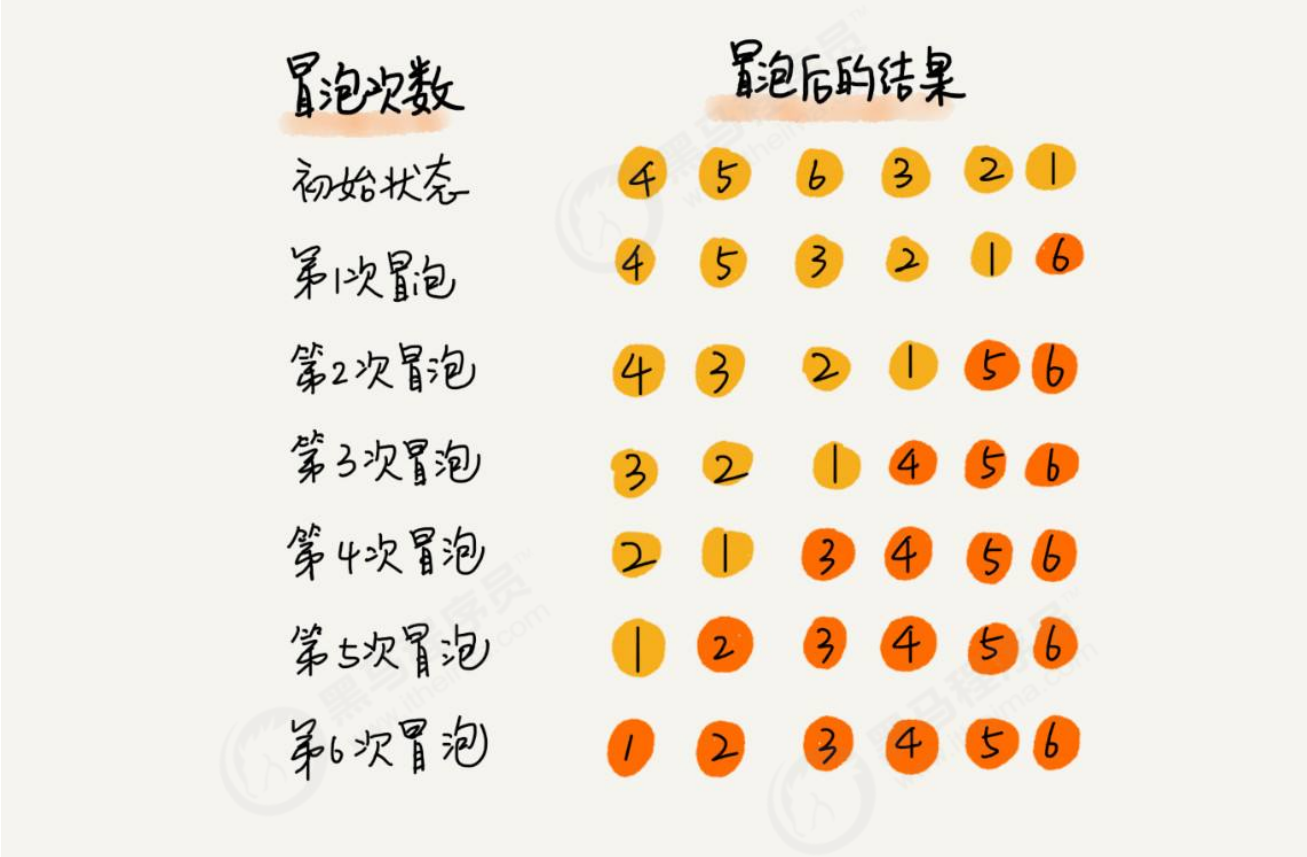
**需求：**

排序前：{4,5,6,3,2,1}

排序后：{1,2,3,4,5,6}

**排序原理：**

1. 比较相邻的元素。如果前一个元素比后一个元素大，就交换这两个元素的位置。
2. 对每一对相邻元素做同样的工作，从开始第一对元素到结尾的最后一对元素。最终最后位置的元素就是最大值。




**冒泡排序API设计：**

| 类名 | Bubble |
| --- | --- |
| 构造方法 | Bubble()：创建Bubble对象 |
| 成员方法 | 1.public static void sort(Comparable[] a)：对数组内的元素进行排序<br>2.private static boolean greater(Comparable v,Comparable w):判断v是否大于w<br>3.private static void exch(Comparable[] a,int i,int j)：交换a数组中，索引i和索引j处的值 |

**冒泡排序的代码实现：**

```
//排序代码
public class Bubble {
    /*
        对数组a中的元素进行排序
     */
    public static void sort(Comparable[] a){
```

```
        for(int i=a.length-1;i>0;i--){
            for (int j = 0; j <i; j++) {
                if (greater(a[j],a[j+1])){
                    exch(a,j,j+1);
                }
            }
        }
    }

    /*
        比较v元素是否大于w元素
     */
    private static  boolean greater(Comparable v,Comparable w){
        return v.compareTo(w)>0;
    }

    /*
    数组元素i和j交换位置
     */
    private static void exch(Comparable[] a,int i,int j){
        Comparable t = a[i];
        a[i]=a[j];
        a[j]=t;
    }

}

//测试代码
public class Test {
    public static void main(String[] args) {
        Integer[] a = {4, 5, 6, 3, 2, 1};
        Bubble.sort(a);
        System.out.println(Arrays.toString(a));
    }
}
```

**冒泡排序的时间复杂度分析** 冒泡排序使用了双层for循环，其中内层循环的循环体是真正完成排序的代码，所以，我们分析冒泡排序的时间复杂度，主要分析一下内层循环体的执行次数即可。

在最坏情况下，也就是假如要排序的元素为{6,5,4,3,2,1}逆序，那么：

元素比较的次数为：

(N-1)+(N-2)+(N-3)+...+2+1=((N-1)+1)*(N-1)/2=N^2/2-N/2;

元素交换的次数为：

(N-1)+(N-2)+(N-3)+...+2+1=((N-1)+1)*(N-1)/2=N^2/2-N/2;

总执行次数为：

(N^2/2-N/2)+(N^2/2-N/2)=N^2-N;

按照大O推导法则，保留函数中的最高阶项那么最终冒泡排序的时间复杂度为O(N^2).

# 1.3 选择排序

选择排序是一种更加简单直观的排序方法。

**需求：**

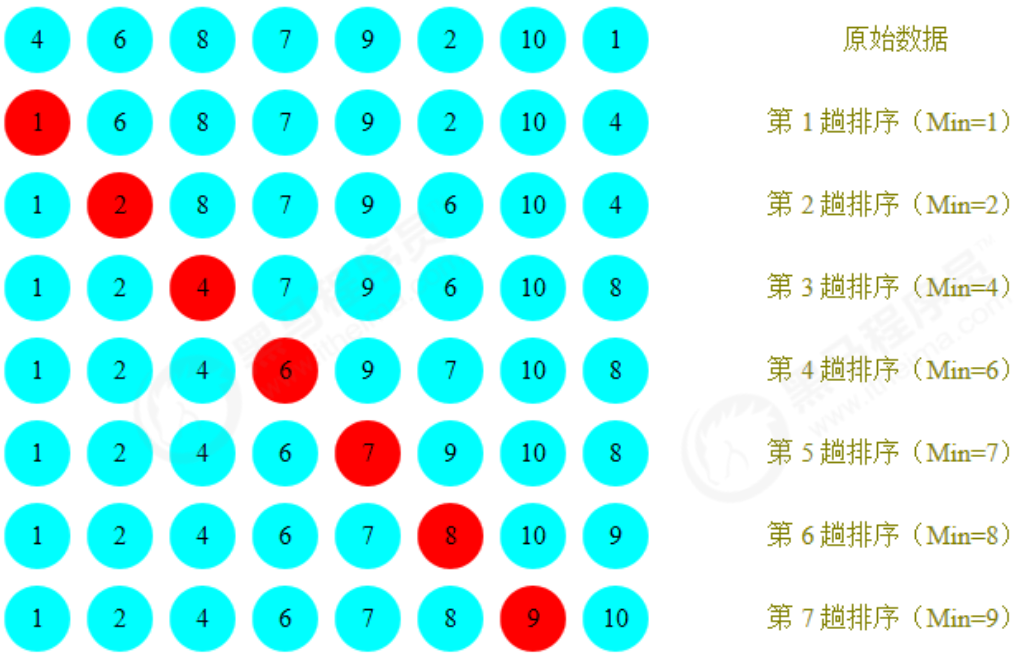
排序前：{4,6,8,7,9,2,10,1}

排序后：{1,2,4,5,7,8,9,10}

**排序原理：**

1.每一次遍历的过程中，都假定第一个索引处的元素是最小值，和其他索引处的值依次进行比较，如果当前索引处的值大于其他某个索引处的值，则假定其他某个索引出的值为最小值，最后可以找到最小值所在的索引

2.交换第一个索引处和最小值所在的索引处的值




**选择排序API设计：**

| 类名 | Selection |
| --- | --- |
| 构造方法 | Selection()：创建Selection对象 |
| 成员方法 | 1.public static void sort(Comparable[] a)：对数组内的元素进行排序<br>2.private static boolean greater(Comparable v,Comparable w):判断v是否大于w<br>3.private static void exch(Comparable[] a,int i,int j)：交换a数组中，索引i和索引j处的值 |

**选择排序的代码实现：**

```
1  //排序代码
```

```
public class Selection {
    /*
        对数组a中的元素进行排序
     */
    public static void sort(Comparable[] a){
        for (int i=0;i<=a.length-2;i++){
            //假定本次遍历，最小值所在的索引是i
            int minIndex=i;
            for (int j=i+1;j<a.length;j++){
                if (greater(a[minIndex],a[j])){
                    //跟换最小值所在的索引
                    minIndex=j;
                }
            }
            //交换i索引处和minIndex索引处的值
            exch(a,i,minIndex);
        }
    }

    /*
        比较v元素是否大于w元素
     */
    private static  boolean greater(Comparable v,Comparable w){
        return v.compareTo(w)>0;
    }

    /*
    数组元素i和j交换位置
     */
    private static void exch(Comparable[] a,int i,int j){
        Comparable t = a[i];
        a[i]=a[j];
        a[j]=t;
    }

}

//测试代码
public class Test {
    public static void main(String[] args) {
        Integer[] a = {4,6,8,7,9,2,10,1};
        Selection.sort(a);
        System.out.println(Arrays.toString(a));
    }
}
```

**选择排序的时间复杂度分析：**

选择排序使用了双层for循环，其中外层循环完成了数据交换，内层循环完成了数据比较，所以我们分别统计数据交换次数和数据比较次数：

数据比较次数：

(N-1)+(N-2)+(N-3)+...+2+1=((N-1)+1)*(N-1)/2=N^2/2-N/2;

数据交换次数：

N-1

时间复杂度：N^2/2-N/2+（N-1）=N^2/2+N/2-1;

根据大O推导法则，保留最高阶项，去除常数因子，时间复杂度为O(N^2);

# 1.4 插入排序

插入排序（Insertion sort）是一种简单直观且稳定的排序算法。

插入排序的工作方式非常像人们排序一手扑克牌一样。开始时，我们的左手为空并且桌子上的牌面朝下。然后，我们每次从桌子上拿走一张牌并将它插入左手中正确的位置。为了找到一张牌的正确位置，我们从右到左将它与已在手中的每张牌进行比较，如下图所示：

**需求：**

排序前：{4,3,2,10,12,1,5,6}

排序后：{1,2,3,4,5,6,10,12}

**排序原理：**

1.把所有的元素分为两组，已经排序的和未排序的；

2.找到未排序的组中的第一个元素，向已经排序的组中进行插入；

3.倒叙遍历已经排序的元素，依次和待插入的元素进行比较，直到找到一个元素小于等于待插入元素，那么就把待插入元素放到这个位置，其他的元素向后移动一位；


**插入排序API设计：**

| 类名 | Insertion |
| --- | --- |
| 构造方法 | Insertion()：创建Insertion对象 |
| 成员方法 | 1.public static void sort(Comparable[] a)：对数组内的元素进行排序<br>2.private static boolean greater(Comparable v,Comparable w):判断v是否大于w<br>3.private static void exch(Comparable[] a,int i,int j)：交换a数组中，索引i和索引j处的值 |

**插入排序代码实现：**

```
public class Insertion {
    /*
        对数组a中的元素进行排序
     */
    public static void sort(Comparable[] a){
        for (int i=1;i<a.length;i++){
            //当前元素为a[i],依次和i前面的元素比较，找到一个小于等于a[i]的元素
            for (int j=i;j>0;j--){
                if (greater(a[j-1],a[j])){
                    //交换元素
                    exch(a,j-1,j);
                }else {
                    //找到了该元素，结束
                    break;
                }

            }
```

```
        }
    }

    /*
        比较v元素是否大于w元素
     */
    private static  boolean greater(Comparable v,Comparable w){
        return v.compareTo(w)>0;
    }

    /*
    数组元素i和j交换位置
     */
    private static void exch(Comparable[] a,int i,int j){
        Comparable t = a[i];
        a[i]=a[j];
        a[j]=t;
    }
}
```

**插入排序的时间复杂度分析**

插入排序使用了双层for循环，其中内层循环的循环体是真正完成排序的代码，所以，我们分析插入排序的时间复杂度，主要分析一下内层循环体的执行次数即可。

最坏情况，也就是待排序的数组元素为{12,10,6,5,4,3,2,1}，那么：

比较的次数为：

(N-1)+(N-2)+(N-3)+...+2+1=((N-1)+1)*(N-1)/2=N^2/2-N/2;

交换的次数为：

(N-1)+(N-2)+(N-3)+...+2+1=((N-1)+1)*(N-1)/2=N^2/2-N/2;

总执行次数为：

(N^2/2-N/2)+(N^2/2-N/2)=N^2-N;

按照大O推导法则，保留函数中的最高阶项那么最终插入排序的时间复杂度为O(N^2).

# 二、高级排序

之前我们学习过基础排序，包括冒泡排序，选择排序还有插入排序，并且对他们在最坏情况下的时间复杂度做了分析，发现都是O(N^2)，而平方阶通过我们之前学习算法分析我们知道，随着输入规模的增大，时间成本将急剧上升，所以这些基本排序方法不能处理更大规模的问题，接下来我们学习一些高级的排序算法，争取降低算法的时间复杂度最高阶次幂。

## 2.1希尔排序

希尔排序是插入排序的一种，又称“缩小增量排序”，是插入排序算法的一种更高效的改进版本。

前面学习插入排序的时候，我们会发现一个很不友好的事儿，如果已排序的分组元素为{2,5,7,9,10}，未排序的分组元素为{1,8}，那么下一个待插入元素为1，我们需要拿着1从后往前，依次和10,9,7,5,2进行交换位置，才能完成真正的插入，每次交换只能和相邻的元素交换位置。那如果我们要提高效率，直观的想法就是一次交换，能把1放到更前面的位置，比如一次交换就能把1插到2和5之间，这样一次交换1就向前走了5个位置，可以减少交换的次数，这样的需求如何实现呢？接下来我们来看看希尔排序的原理。

**需求：**

排序前：{9,1,2,5,7,4,8,6,3,5}

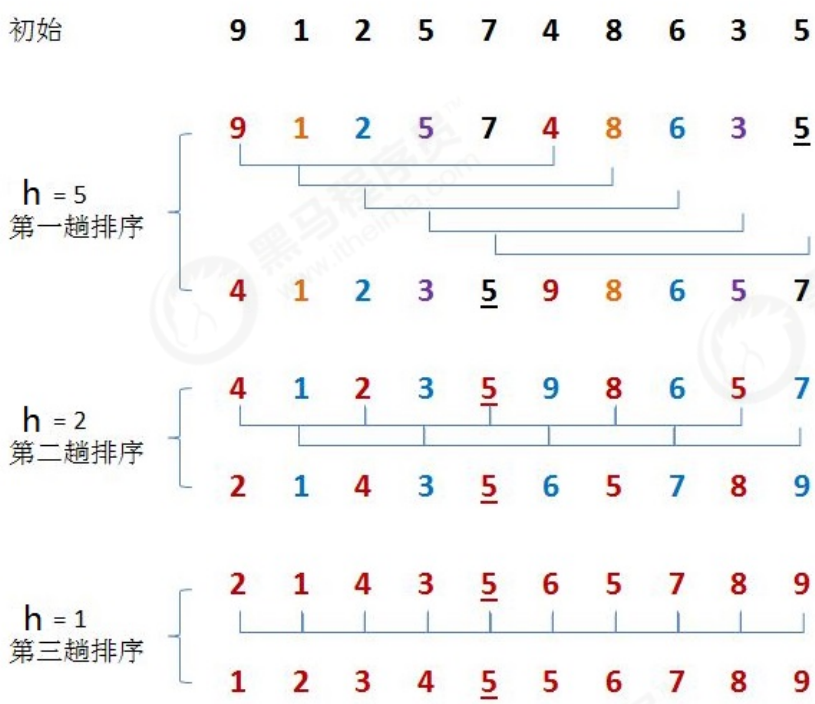
排序后：{1,2,3,4,5,5,6,7,8,9}

**排序原理：**

1.选定一个增长量h，按照增长量h作为数据分组的依据，对数据进行分组；

2.对分好组的每一组数据完成插入排序；

3.减小增长量，最小减为1，重复第二步操作。




增长量h的确定：增长量h的值每一固定的规则，我们这里采用以下规则：

```
int h=1
while(h<5){
    h=2h+1；//3,7
}
//循环结束后我们就可以确定h的最大值；
h的减小规则为：
    h=h/2
```

**希尔排序的API设计：**

| 类名 | Shell |
| --- | --- |
| 构造方法 | Shell()：创建Shell对象 |
| 成员方法 | 1.public static void sort(Comparable[] a)：对数组内的元素进行排序<br>2.private static boolean greater(Comparable v,Comparable w):判断v是否大于w<br>3.private static void exch(Comparable[] a,int i,int j)：交换a数组中，索引i和索引j处的值 |

**希尔排序的代码实现：**

```
//排序代码
public class Shell {
    /*
            对数组a中的元素进行排序
        */
    public static void sort(Comparable[] a){
        int N = a.length;
        //确定增长量h的最大值
        int h=1;
        while(h<N/2){
            h=h*2+1;
        }


        //当增长量h小于1，排序结束
        while(h>=1){
            //找到待插入的元素
            for (int i=h;i<N;i++){
                //a[i]就是待插入的元素
                //把a[i]插入到a[i-h],a[i-2h],a[i-3h]...序列中
                for (int j=i;j>=h;j-=h){
                    //a[j]就是待插入元素，依次和a[j-h],a[j-2h],a[j-3h]进行比较，如果a[j]小，那么交换位置，如果不小于，a[j]大，则插入完成。
                    if (greater(a[j-h],a[j])){
                        exch(a,j,j-h);
                    }else{
                        break;
                    }
                }

            }
```

```
            h/=2;
        }

    }

    /*
        比较v元素是否大于w元素
     */
    private static  boolean greater(Comparable v,Comparable w){
        return v.compareTo(w)>0;
    }

    /*
    数组元素i和j交换位置
     */
    private static void exch(Comparable[] a,int i,int j){
        Comparable t = a[i];
        a[i]=a[j];
        a[j]=t;
    }
}

//测试代码
public class Test {
    public static void main(String[] args) {
        Integer[] a = {9,1,2,5,7,4,8,6,3,5} ;
        Shell.sort(a);
        System.out.println(Arrays.toString(a));
    }
}
```

**希尔排序的时间复杂度分析**

在希尔排序中，增长量h并没有固定的规则，有很多论文研究了各种不同的递增序列，但都无法证明某个序列是最好的，对于希尔排序的时间复杂度分析，已经超出了我们课程设计的范畴，所以在这里就不做分析了。

我们可以使用事后分析法对希尔排序和插入排序做性能比较。

在资料的测试数据文件夹下有一个reverse_shell_insertion.txt文件，里面存放的是从100000到1的逆向数据，我们可以根据这个批量数据完成测试。测试的思想：在执行排序前前记录一个时间，在排序完成后记录一个时间，两个时间的时间差就是排序的耗时。

**希尔排序和插入排序性能比较测试代码：**

```
public class SortCompare {
    public static void main(String[] args) throws Exception{
        ArrayList<Integer> list = new ArrayList<>();
        //读取reverse_arr.txt文件
        BufferedReader reader = new BufferedReader(new InputStreamReader(new
FileInputStream("reverse_shell_insertion.txt")));
        String line=null;
        while((line=reader.readLine())!=null){

            //把每一个数字存入到集合中
```

```
            list.add(Integer.valueOf(line));
        }
        reader.close();
        //把集合转换成数组
        Integer[] arr = new Integer[list.size()];
        list.toArray(arr);

        testInsertion(arr);//使用插入排序耗时：20859
//        testShell(arr);//使用希尔排序耗时：31

    }
    public static void testInsertion(Integer[] arr){
        //使用插入排序完成测试
        long start = System.currentTimeMillis();
        Insertion.sort(arr);
        long end= System.currentTimeMillis();
        System.out.println("使用插入排序耗时："+(end-start));
    }

    public static void testShell(Integer[] arr){
        //使用希尔排序完成测试
        long start = System.currentTimeMillis();
        Shell.sort(arr);
        long end = System.currentTimeMillis();
        System.out.println("使用希尔排序耗时："+(end-start));
    }
}
```

通过测试发现，在处理大批量数据时，希尔排序的性能确实高于插入排序。

# 2.2 归并排序

## 2.2.1 递归

正式学习归并排序之前，我们得先学习一下递归算法。

**定义：**

定义方法时，在方法内部调用方法本身，称之为递归.

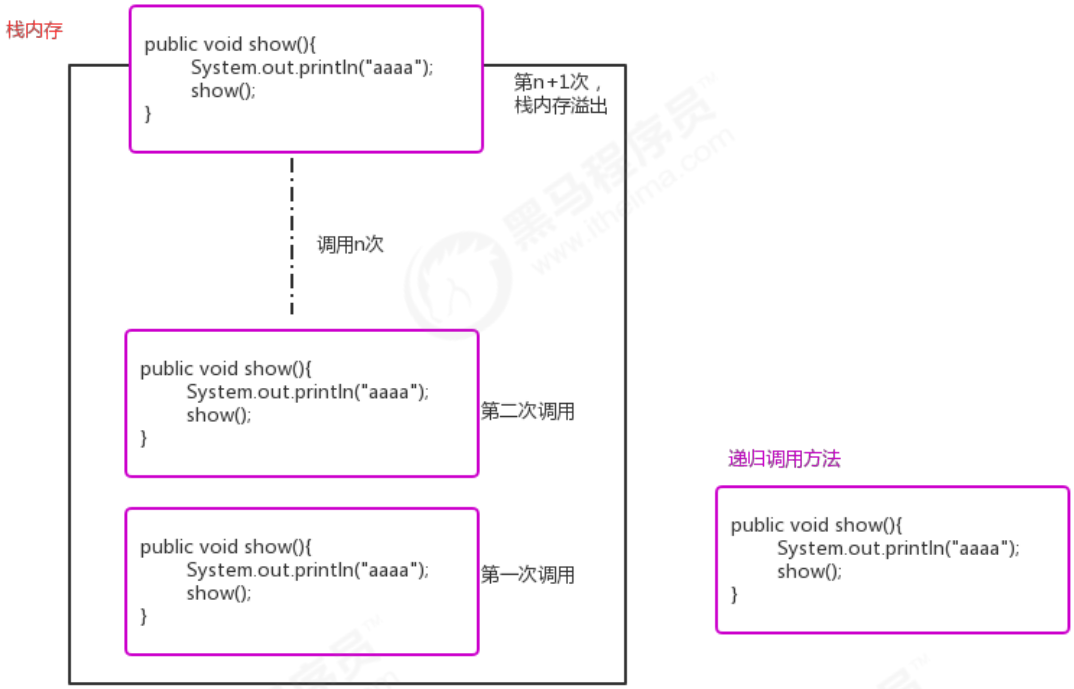
```
public void show(){
    System.out.println("aaaa");
    show();
}
```

**作用：**

它通常把一个大型复杂的问题，层层转换为一个与原问题相似的，规模较小的问题来求解。递归策略只需要少量的程序就可以描述出解题过程所需要的多次重复计算，大大地减少了程序的代码量。

**注意事项：**

在递归中，不能无限制的调用自己，必须要有边界条件，能够让递归结束，因为每一次递归调用都会在栈内存开辟新的空间，重新执行方法，如果递归的层级太深，很容易造成栈内存溢出。




**需求：**

请定义一个方法，使用递归完成求N的阶乘；

```
分析：

1!:     1
2!:     2*1=2*1!
3!:     3*2*1=3*2!
4!:     4*3*2*1=4*3!
...
n!:     n*(n-1)*(n-2)...*2*1=n*(n-1)!

所以，假设有一个方法factorial(n)用来求n的阶乘，那么n的阶乘还可以表示为n*factorial(n-1)
```

**代码实现：**

```java
public class Test {
    public static void main(String[] args) throws Exception {
        int result = factorial(5);
        System.out.println(result);
    }

    public static int factorial(int n){
        if (n==1){
            return 1;
        }
        return n*factorial(n-1);
    }
}
```

## 2.2.2 归并排序

归并排序是建立在归并操作上的一种有效的排序算法，该算法是采用分治法的一个非常典型的应用。将已有序的子序列合并，得到完全有序的序列；即先使每个子序列有序，再使子序列段间有序。若将两个有序表合并成一个有序表，称为二路归并。

**需求：**

排序前：{8,4,5,7,1,3,6,2}

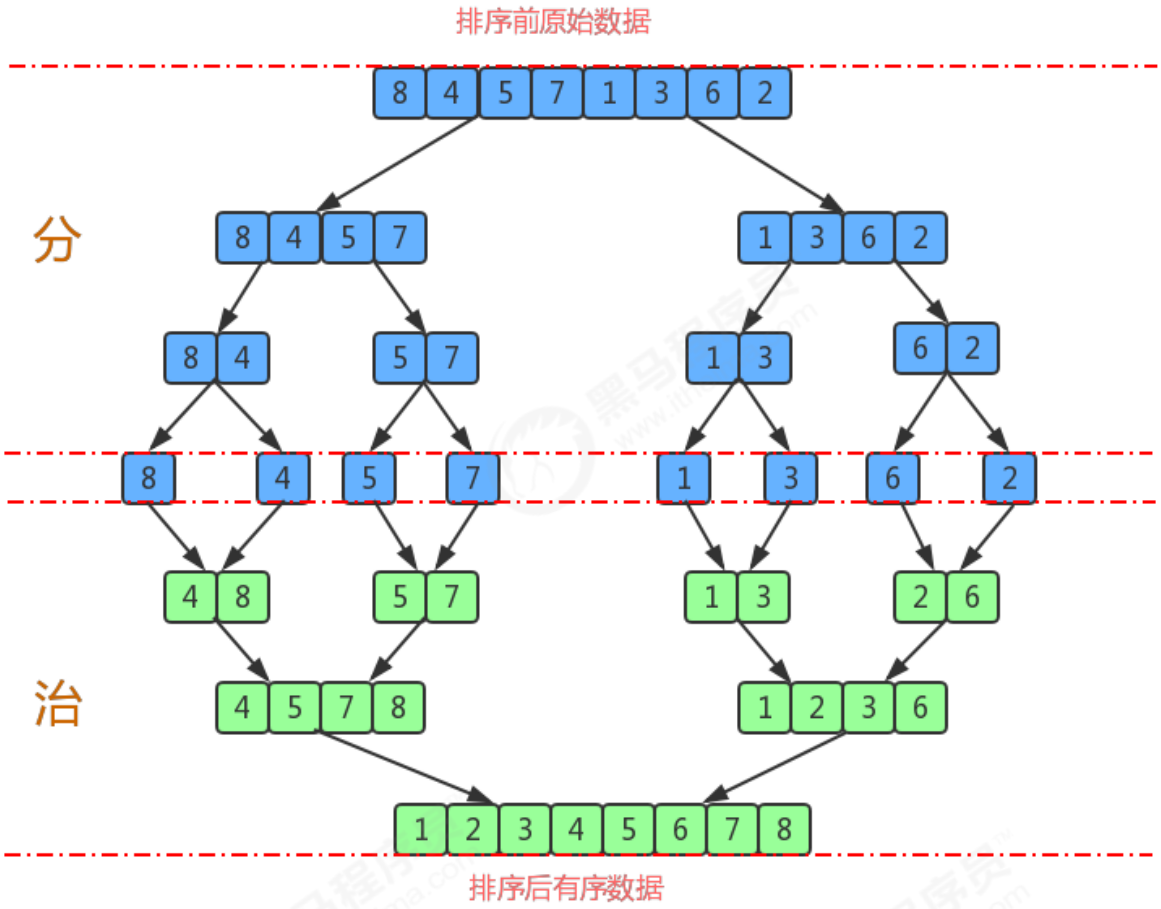
排序后：{1,2,3,4,5,6,7,8}

**排序原理：**

1.尽可能的一组数据拆分成两个元素相等的子组，并对每一个子组继续拆分，直到拆分后的每个子组的元素个数是1为止。

2.将相邻的两个子组进行合并成一个有序的大组；

3.不断的重复步骤2，直到最终只有一个组为止。




**归并排序API设计：**

| 类名 | Merge |
| --- | --- |
| 构造方法 | Merge()：创建Merge对象 |
| 成员方法 | 1.public static void sort(Comparable[] a)：对数组内的元素进行排序<br>2.private static void sort(Comparable[] a, int lo, int hi)：对数组a中从索引lo到索引hi之间的元素进行排序<br>3.private static void merge(Comparable[] a, int lo, int mid, int hi):从索引lo到所以mid为一个子组，从索引mid+1到索引hi为另一个子组，把数组a中的这两个子组的数据合并成一个有序的大组（从索引lo到索引hi ）<br>4.private static boolean less(Comparable v,Comparable w):判断v是否小于w<br>5.private static void exch(Comparable[] a,int i,int j)：交换a数组中，索引i和索引j处的值 |
| 成员变量 | 1.private static Comparable[] assist：完成归并操作需要的辅助数组 |

**归并原理：**

前提：

int lo=0;

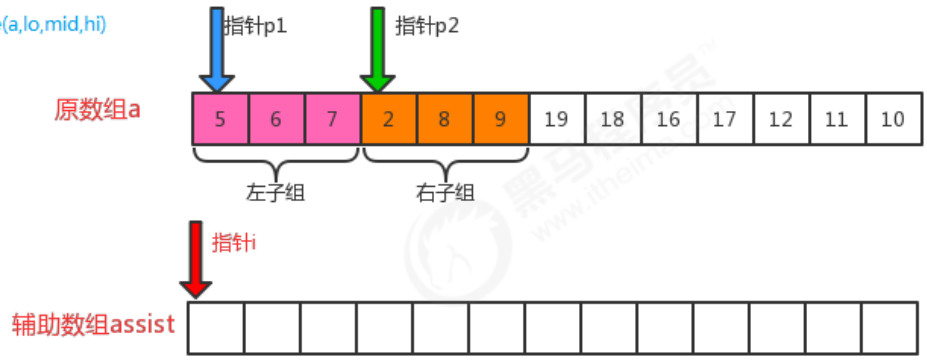
int hi=5;

int mid=lo+(hi-lo)/2=2

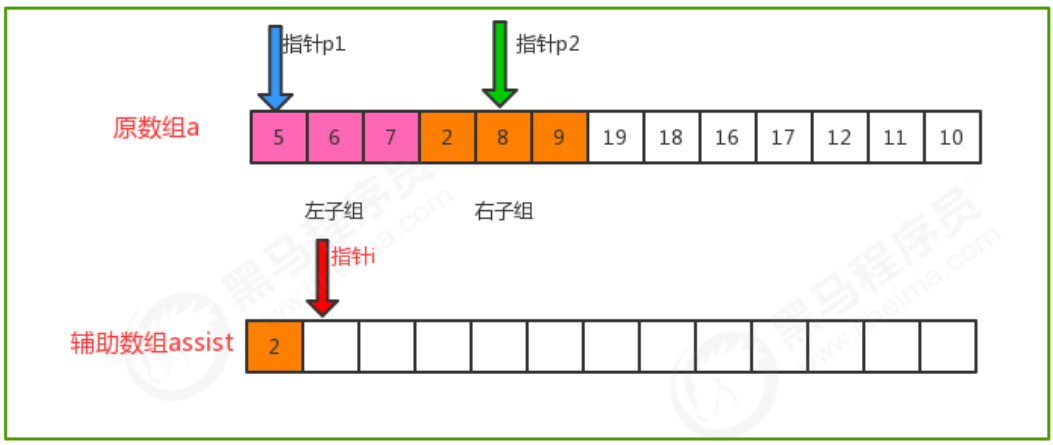
sort ( a,lo,mid )

sort(a,mid+1,hi);

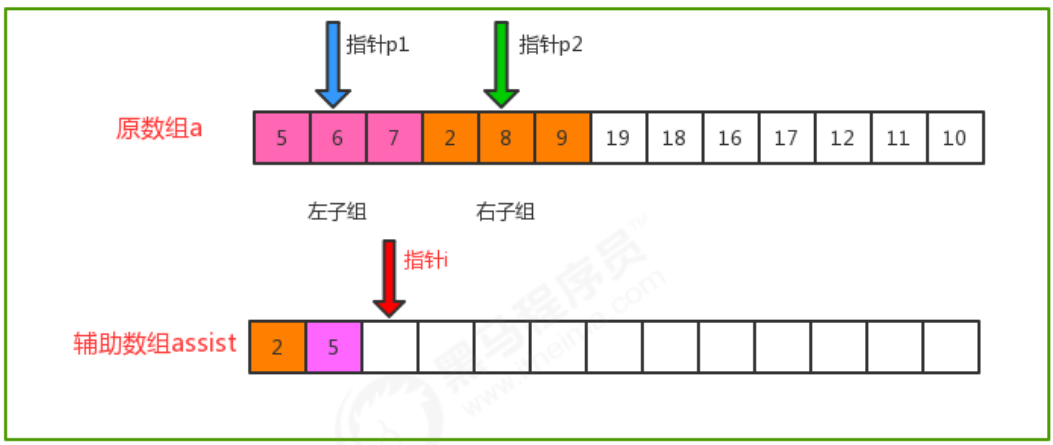
调用merge(a,lo,mid,hi)



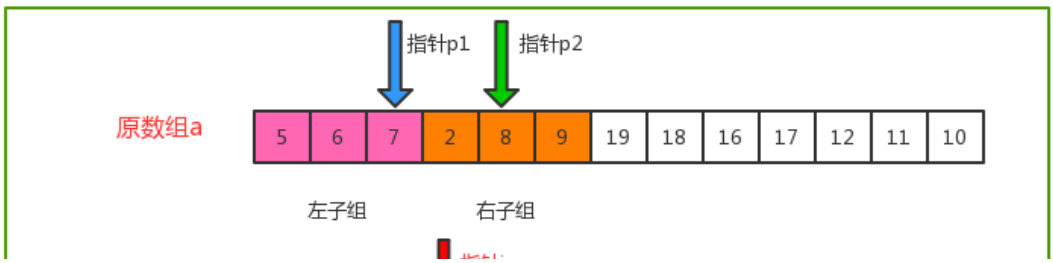

第一次填充

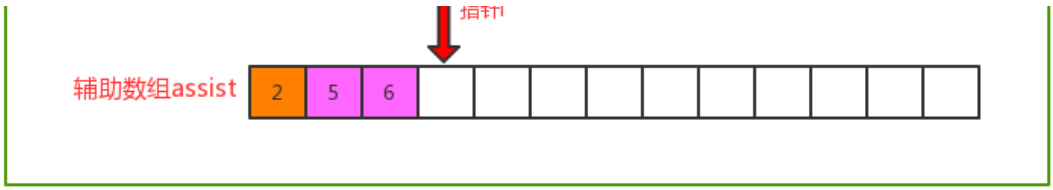



第二次填充

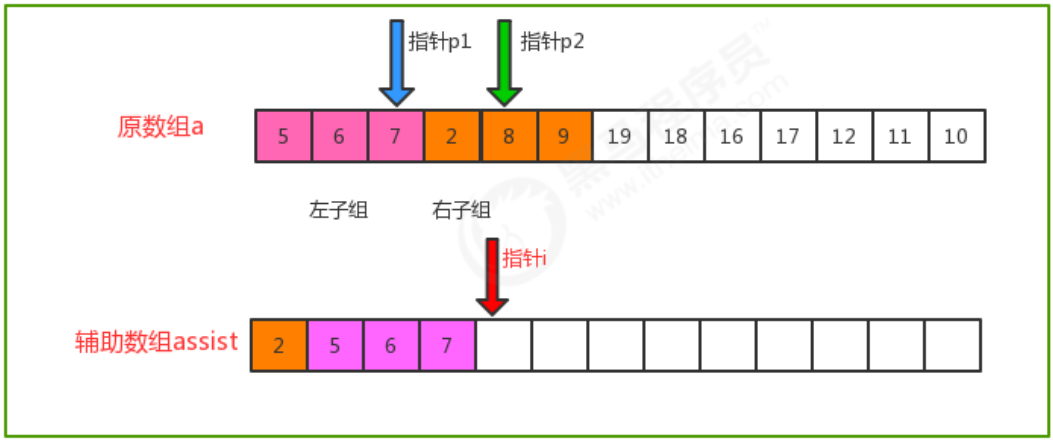



第三次填充

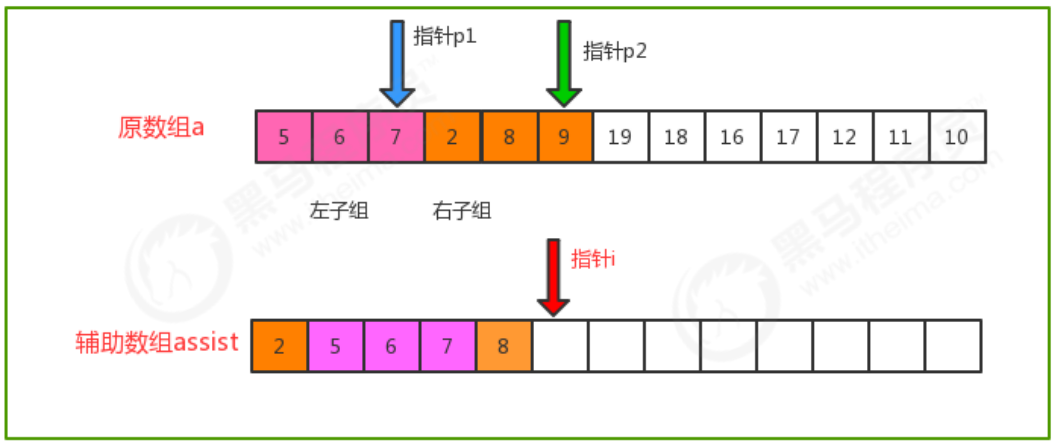




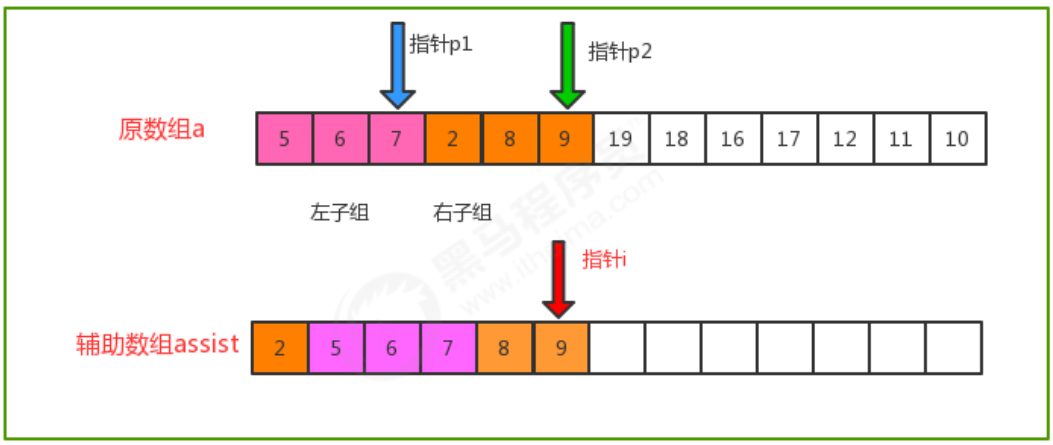

第四次填充



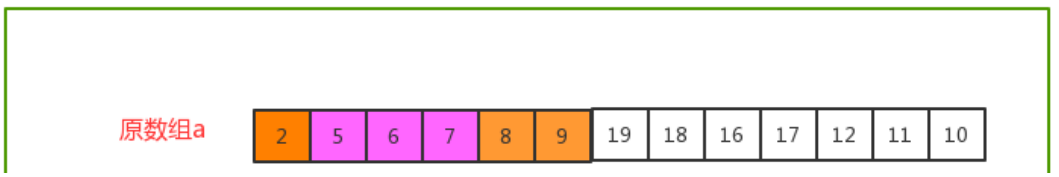

第五次填充

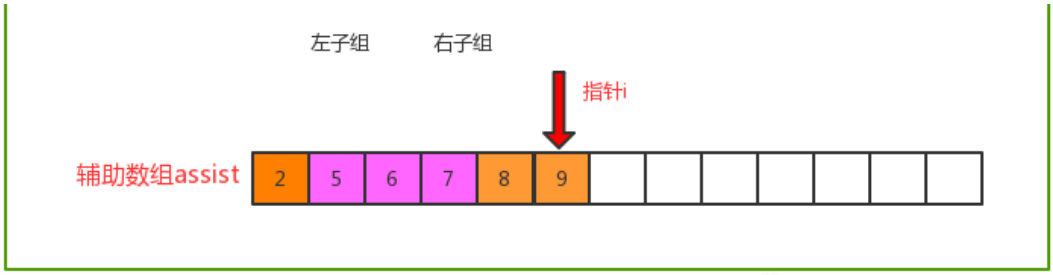



第六次填充




把辅助数组中排序好的元素拷贝到原数组中






**归并排序代码实现：**

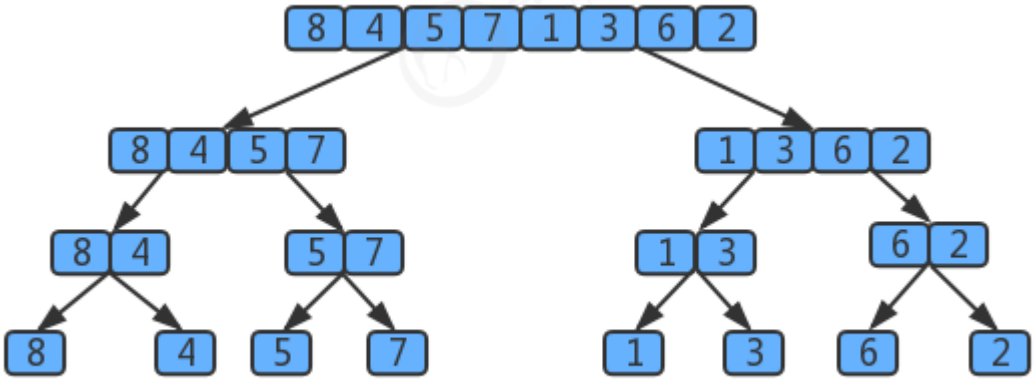
```
//排序代码
public class Merge {
    private static Comparable[] assist;//归并所需要的辅助数组

    /*
           对数组a中的元素进行排序
        */
    public static void sort(Comparable[] a) {
        assist = new Comparable[a.length];
        int lo = 0;
        int hi = a.length-1;
        sort(a, lo, hi);
    }

    /*
    对数组a中从lo到hi的元素进行排序
     */
    private static void sort(Comparable[] a, int lo, int hi) {
        if (hi <= lo) {
            return;
        }

        int mid = lo + (hi - lo) / 2;

        //对lo到mid之间的元素进行排序；
        sort(a, lo, mid);
        //对mid+1到hi之间的元素进行排序；
        sort(a, mid+1, hi);
        //对lo到mid这组数据和mid到hi这组数据进行归并
        merge(a, lo, mid, hi);
    }

    /*
    对数组中，从lo到mid为一组，从mid+1到hi为一组，对这两组数据进行归并
     */
    private static void merge(Comparable[] a, int lo, int mid, int hi) {
        //lo到mid这组数据和mid+1到hi这组数据归并到辅助数组assist对应的索引处
        int i = lo;//定义一个指针，指向assist数组中开始填充数据的索引
        int p1 = lo;//定义一个指针，指向第一组数据的第一个元素

        int p2 = mid + 1;//定义一个指针，指向第二组数据的第一个元素
```

```
41
42
43          //比较左边小组和右边小组中的元素大小，哪个小，就把哪个数据填充到assist数组中
44          while (p1 <= mid && p2 <= hi) {
45              if (less(a[p1], a[p2])) {
46                  assist[i++] = a[p1++];
47              } else {
48                  assist[i++] = a[p2++];
49              }
50          }
51
52          //上面的循环结束后，如果退出循环的条件是p1<=mid，则证明左边小组中的数据已经归并完毕，如果退出循环的条件是p2<=hi,则证明右边小组的数据已经填充完毕；
53          //所以需要把未填充完毕的数据继续填充到assist中,//下面两个循环，只会执行其中的一个
54          while(p1<=mid){
55              assist[i++]=a[p1++];
56          }
57          while(p2<=hi){
58              assist[i++]=a[p2++];
59          }
60
61          //到现在为止，assist数组中，从lo到hi的元素是有序的，再把数据拷贝到a数组中对应的索引处
62
63          for (int index=lo;index<=hi;index++){
64              a[index]=assist[index];
65          }
66      }
67
68
69      /*
70          比较v元素是否小于w元素
71       */
72      private static boolean less(Comparable v, Comparable w) {
73          return v.compareTo(w) < 0;
74      }
75
76      /*
77      数组元素i和j交换位置
78       */
79      private static void exch(Comparable[] a, int i, int j) {
80          Comparable t = a[i];
81          a[i] = a[j];
82          a[j] = t;
83      }
84  }
85
86  //测试代码
87  public class Test {
88      public static void main(String[] args) throws Exception {
89          Integer[] arr = {8, 4, 5, 7, 1, 3, 6, 2};
90          Merge.sort(arr);
91          System.out.println(Arrays.toString(arr));
92      }
```

```
93  }
94
```

**归并排序时间复杂度分析：**

归并排序是分治思想的最典型的例子，上面的算法中，对a[lo...hi]进行排序，先将它分为a[lo...mid]和a[mid+1...hi]两部分，分别通过递归调用将他们单独排序，最后将有序的子数组归并为最终的排序结果。该递归的出口在于如果一个数组不能再被分为两个子数组，那么就会执行merge进行归并，在归并的时候判断元素的大小进行排序。




用树状图来描述归并，如果一个数组有8个元素，那么它将每次除以2找最小的子数组，共拆log8次，值为3，所以树共有3层,那么自顶向下第k层有2^k个子数组，每个数组的长度为2^(3-k)，归并最多需要2^(3-k)次比较。因此每层的比较次数为 2^k * 2^(3-k)=2^3,那么3层总共为 3*2^3。

假设元素的个数为n，那么使用归并排序拆分的次数为log2(n),所以共log2(n)层，那么使用log2(n)替换上面3*2^3中的3这个层数，最终得出的归并排序的时间复杂度为：log2(n)* 2^(log2(n))=log2(n)*n,根据大O推导法则，忽略底数，最终归并排序的时间复杂度为O(nlogn);

**归并排序的缺点：**

需要申请额外的数组空间，导致空间复杂度提升，是典型的以空间换时间的操作。

**归并排序与希尔排序性能测试：**

之前我们通过测试可以知道希尔排序的性能是由于插入排序的，那现在学习了归并排序后，归并排序的效率与希尔排序的效率哪个高呢？我们使用同样的测试方式来完成一样这两个排序算法之间的性能比较。

在资料的测试数据文件夹下有一个reverse_arr.txt文件，里面存放的是从1000000到1的逆向数据，我们可以根据这个批量数据完成测试。测试的思想：在执行排序前前记录一个时间，在排序完成后记录一个时间，两个时间的时间差就是排序的耗时。

**希尔排序和插入排序性能比较测试代码：**

```
1  public class SortCompare {
2      public static void main(String[] args) throws Exception{
3          ArrayList<Integer> list = new ArrayList<>();
4          //读取a.txt文件
5          BufferedReader reader = new BufferedReader(new InputStreamReader(new
```

```
FileInputStream("reverse_merge_shell.txt")));
        String line=null;
        while((line=reader.readLine())!=null){
            //把每一个数字存入到集合中
            list.add(Integer.valueOf(line));
        }
        reader.close();
        //把集合转换成数组
        Integer[] arr = new Integer[list.size()];
        list.toArray(arr);

//        testMerge(arr);//使用归并排序耗时：1200
        testShell(arr);//使用希尔排序耗时：1277

    }
    public static void testMerge(Integer[] arr){
        //使用插入排序完成测试
        long start = System.currentTimeMillis();
        Merge.sort(arr);
        long end= System.currentTimeMillis();
        System.out.println("使用归并排序耗时："+(end-start));
    }

    public static void testShell(Integer[] arr){
        //使用希尔排序完成测试
        long start = System.currentTimeMillis();
        Shell.sort(arr);
        long end = System.currentTimeMillis();
        System.out.println("使用希尔排序耗时："+(end-start));
    }
}
```

通过测试，发现希尔排序和归并排序在处理大批量数据时差别不是很大。

## 2.3 快速排序

快速排序是对冒泡排序的一种改进。它的基本思想是：通过一趟排序将要排序的数据分割成独立的两部分，其中一部分的所有数据都比另外一部分的所有数据都要小，然后再按此方法对这两部分数据分别进行快速排序，整个排序过程可以递归进行，以此达到整个数据变成有序序列。

**需求：**

排序前:{6, 1, 2, 7, 9, 3, 4, 5, 8}

排序后:{1, 2, 3, 4, 5, 6, 7, 8, 9}

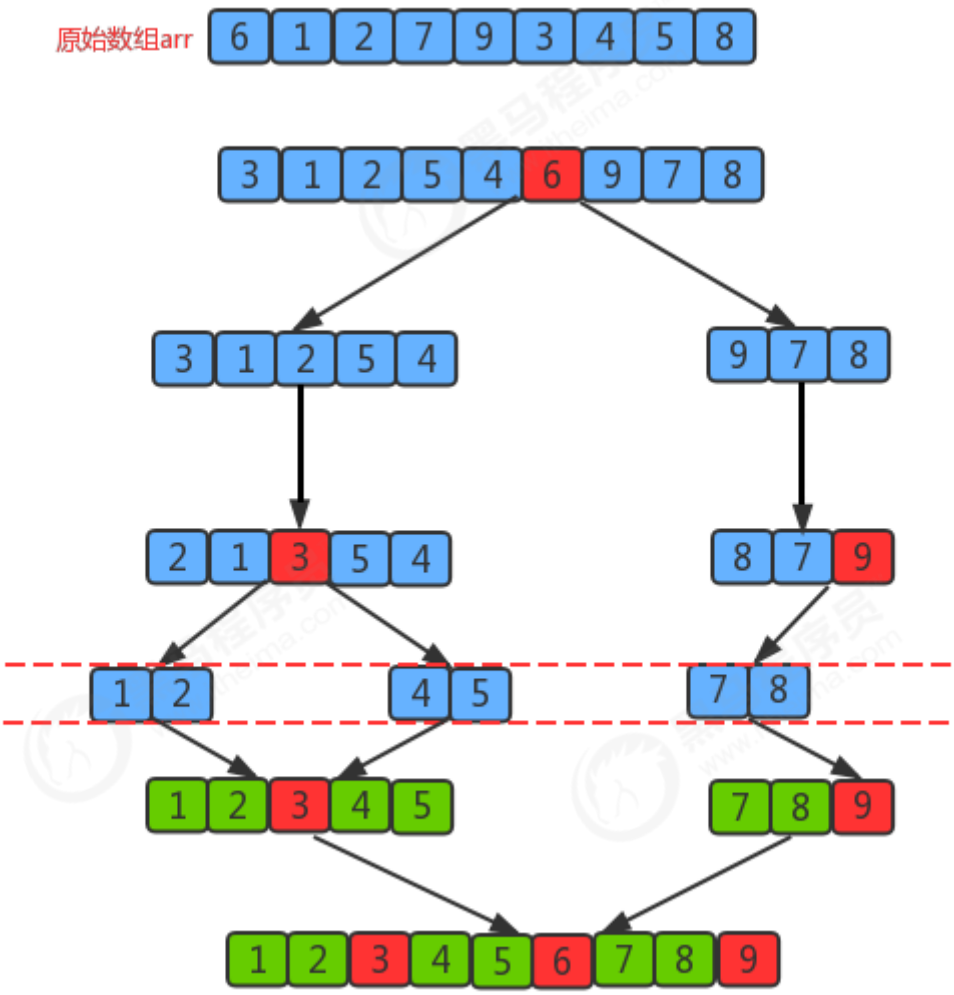
**排序原理：**

1.首先设定一个分界值，通过该分界值将数组分成左右两部分；

2.将大于或等于分界值的数据放到到数组右边，小于分界值的数据放到数组的左边。此时左边部分中各元素都小于或等于分界值，而右边部分中各元素都大于或等于分界值；

3.然后，左边和右边的数据可以独立排序。对于左侧的数组数据，又可以取一个分界值，将该部分数据分成左右两部分，同样在左边放置较小值，右边放置较大值。右侧的数组数据也可以做类似处理。

4.重复上述过程，可以看出，这是一个递归定义。通过递归将左侧部分排好序后，再递归排好右侧部分的顺序。当左侧和右侧两个部分的数据排完序后，整个数组的排序也就完成了。




**快速排序API设计:**

| 类名 | Quick |
| --- | --- |
| 构造方法 | Quick()：创建Quick对象 |
| 成员方法 | 1.public static void sort(Comparable[] a)：对数组内的元素进行排序<br>2.private static void sort(Comparable[] a, int lo, int hi)：对数组a中从索引lo到索引hi之间的元素进行排序<br>3.public static int partition(Comparable[] a,int lo,int hi):对数组a中，从索引 lo到索引 hi之间的元素进行分组，并返回分组界限对应的索引<br>4.private static boolean less(Comparable v,Comparable w):判断v是否小于w<br>5.private static void exch(Comparable[] a,int i,int j)：交换a数组中，索引i和索引j处的值 |

**切分原理：**

把一个数组切分成两个子数组的基本思想：

1.找一个基准值，用两个指针分别指向数组的头部和尾部；

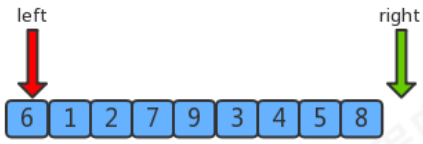
2.先从尾部向头部开始搜索一个比基准值小的元素，搜索到即停止，并记录指针的位置；

3.再从头部向尾部开始搜索一个比基准值大的元素，搜索到即停止，并记录指针的位置；

4.交换当前左边指针位置和右边指针位置的元素；

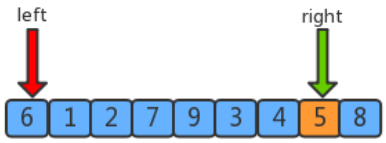
5.重复2,3,4步骤，直到左边指针的值大于右边指针的值停止。


第一次切分，取基准值为int key=arr[0]=6，left初始值为0，ringht初始值为9



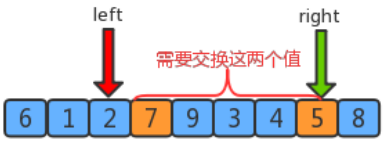

先从右往左搜索，搜索一个比基准值小的元素，arr[--right]>key,暂时没有找到


arr[--right]<key,找到一个比基准值小的元素，此时right=7

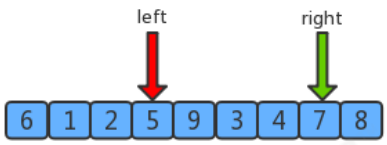



再从左往右搜索，搜索一个比基准值大的元素，arr[++left]<key,暂时没有找到

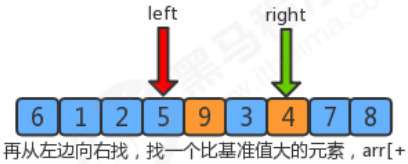

arr[++left]<key,暂时没有找到



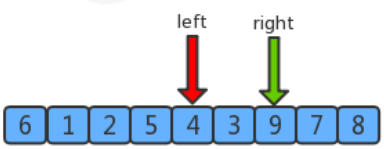

arr[++left]>key,找到一个比基准值大的元素，此时left=3，那么交换left索引处的值和right索引处的值



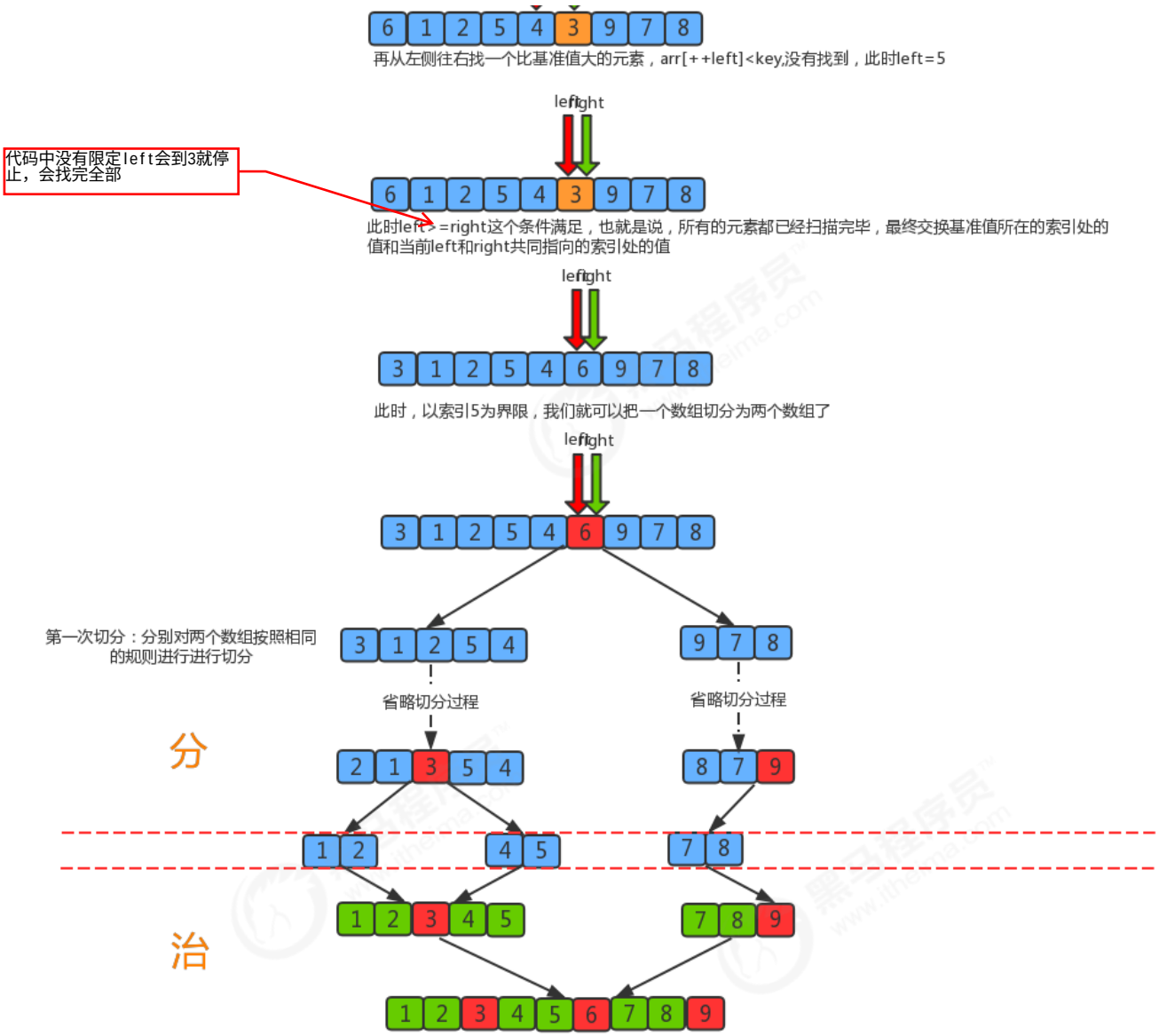

继续从右侧往左找一个比基准值小的元素，arr[--right]<key,找到了，此时right=6




再从左边向右找，找一个比基准值大的元素，arr[++left]>key,找到了,此时left=4，交换left索引处和right索引处的值




继续从右侧往左找一个比基准值小的元素，arr[--right]<key,找到了，此时right=5




**快速排序代码实现：**

```
//排序代码
public class Quick {
    public static void sort(Comparable[] a) {
        int lo = 0;
        int hi = a.length - 1;
        sort(a, lo,hi);
    }

    private static void sort(Comparable[] a, int lo, int hi) {
        if (hi<=lo){
            return;
        }
        //对a数组中，从lo到hi的元素进行切分
        int partition = partition(a, lo, hi);
        //对左边分组中的元素进行排序

        //对右边分组中的元素进行排序
```

```
        sort(a,lo,partition-1);
        sort(a,partition+1,hi);
    }

    public static int partition(Comparable[] a, int lo, int hi) {
        Comparable key=a[lo];//把最左边的元素当做基准值
        int left=lo;//定义一个左侧指针，初始指向最左边的元素
        int right=hi+1;//定义一个右侧指针，初始指向左右侧的元素下一个位置
        //进行切分
        while(true){
            //先从右往左扫描，找到一个比基准值小的元素
            while(less(key,a[--right])){//循环停止，证明找到了一个比基准值小的元素
                if (right==lo){
                    break;//已经扫描到最左边了，无需继续扫描
                }
            }

            //再从左往右扫描，找一个比基准值大的元素
            while(less(a[++left],key)){//循环停止，证明找到了一个比基准值大的元素
                if (left==hi){
                    break;//已经扫描到了最右边了，无需继续扫描
                }
            }

            if (left>=right){
                //扫描完了所有元素，结束循环
                break;
            }else{
                //交换left和right索引处的元素
                exch(a,left,right);
            }
        }

        //交换最后rigth索引处和基准值所在的索引处的值
        exch(a,lo,right);
        return right;//right就是切分的界限
    }

    /*
    数组元素i和j交换位置
     */
    private static void exch(Comparable[] a, int i, int j) {
        Comparable t = a[i];
        a[i] = a[j];
        a[j] = t;
    }

    /*
       比较v元素是否小于w元素
    */
    private static boolean less(Comparable v, Comparable w) {
        return v.compareTo(w) < 0;

    }
```

```
70
71  }
72
73  //测试代码
74  public class Test {
75      public static void main(String[] args) throws Exception {
76          Integer[] arr = {6, 1, 2, 7, 9, 3, 4, 5, 8};
77          Quick.sort(arr);
78          System.out.println(Arrays.toString(arr));
79      }
80  }
```

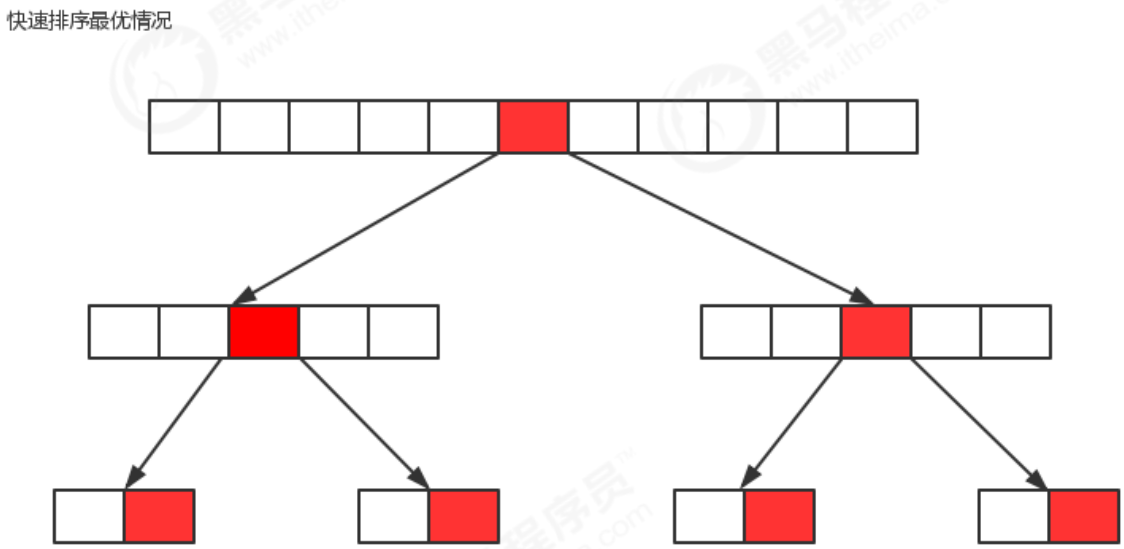
**快速排序和归并排序的区别：**

快速排序是另外一种分治的排序算法，它将一个数组分成两个子数组，将两部分独立的排序。快速排序和归并排序是互补的：归并排序将数组分成两个子数组分别排序，并将有序的子数组归并从而将整个数组排序，而快速排序的方式则是当两个数组都有序时，整个数组自然就有序了。在归并排序中，一个数组被等分为两半，归并调用发生在处理整个数组之前，在快速排序中，切分数组的位置取决于数组的内容，递归调用发生在处理整个数组之后。

**快速排序时间复杂度分析：**

快速排序的一次切分从两头开始交替搜索，直到left和right重合，因此，一次切分算法的时间复杂度为O(n),但整个快速排序的时间复杂度和切分的次数相关。

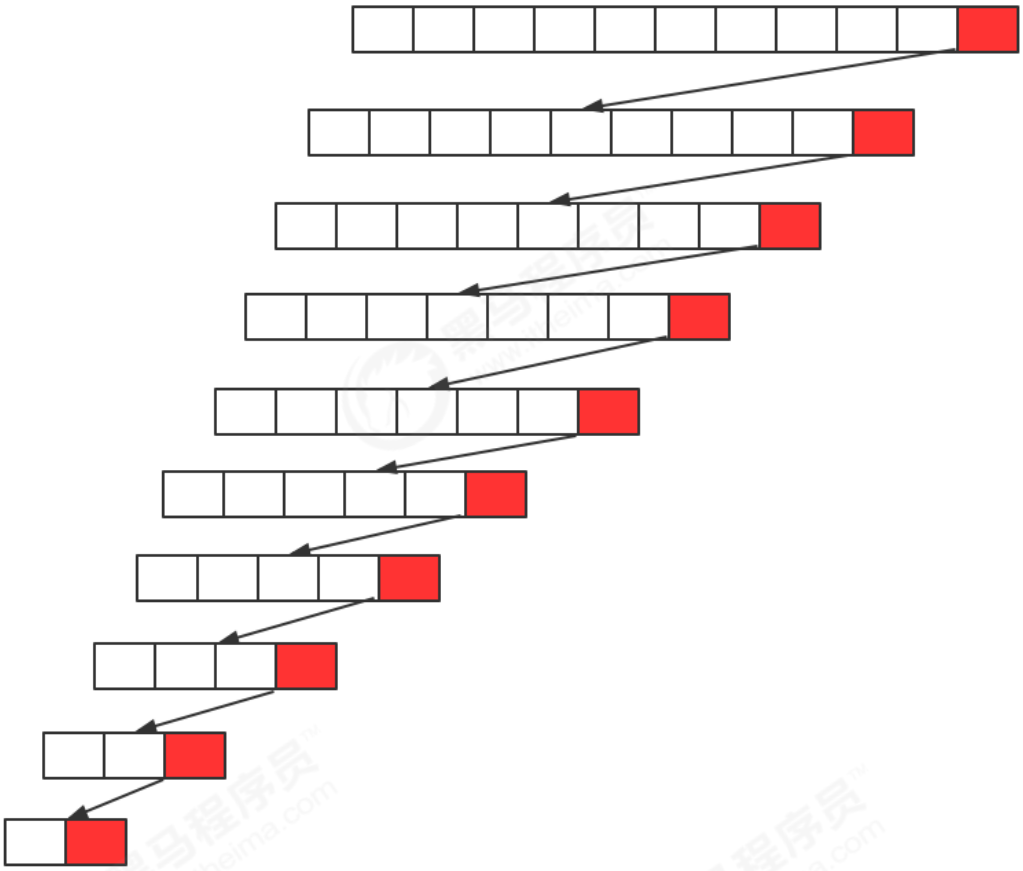
最优情况：每一次切分选择的基准数字刚好将当前序列等分。




如果我们把数组的切分看做是一个树，那么上图就是它的最优情况的图示，共切分了logn次，所以，最优情况下快速排序的时间复杂度为O(nlogn);

最坏情况：每一次切分选择的基准数字是当前序列中最大数或者最小数，这使得每次切分都会有一个子组，那么总共就得切分n次，所以，最坏情况下，快速排序的时间复杂度为O(n^2);

快速排序最差情况

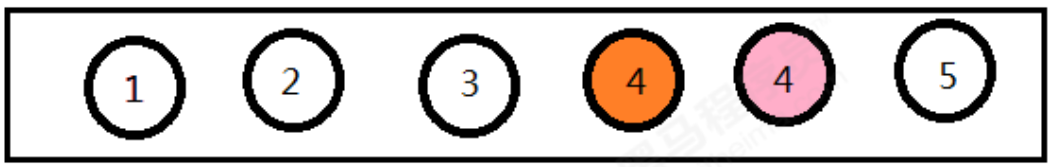


平均情况：每一次切分选择的基准数字不是最大值和最小值，也不是中值，这种情况我们也可以用数学归纳法证明，快速排序的时间复杂度为O(nlogn),由于数学归纳法有很多数学相关的知识，容易使我们混乱，所以这里就不对平均情况的时间复杂度做证明了。

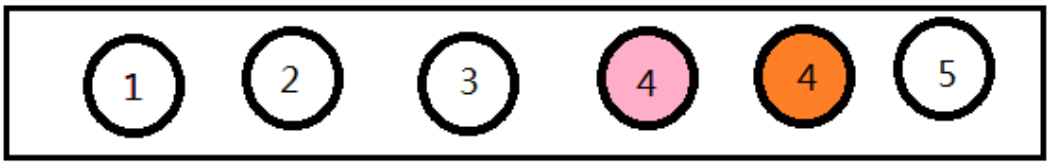
## 2.4 排序的稳定性

**稳定性的定义：**

数组arr中有若干元素，其中A元素和B元素相等，并且A元素在B元素前面，如果使用某种排序算法排序后，能够保证A元素依然在B元素的前面，可以说这个该算法是稳定的。


稳定




不稳定

**稳定性的意义：**

如果一组数据只需要一次排序，则稳定性一般是没有意义的，如果一组数据需要多次排序，稳定性是有意义的。例如要排序的内容是一组商品对象，第一次排序按照价格由低到高排序，第二次排序按照销量由高到低排序，如果第二次排序使用稳定性算法，就可以使得相同销量的对象依旧保持着价格高低的顺序展现，只有销量不同的对象才需要重新排序。这样既可以保持第一次排序的原有意义，而且可以减少系统开销。

第一次按照价格从低到高排序：

| 商品名称 | 价格 | 销量 |
| --- | --- | --- |
| 三星Note9 | 3999 | 21 |
| 华为mate30 | 4999 | 65 |
| 华为p30 | 5999 | 65 |
| Iphone 11 | 6899 | 32 |

第二次按照销量进行从高到低排序：

| 商品名称 | 价格 | 销量 |
| --- | --- | --- |
| 华为mate30 | 4999 | 65 |
| 华为p30 | 5999 | 65 |
| Iphone 11 | 6899 | 32 |
| 三星Note9 | 3999 | 21 |

**常见排序算法的稳定性：**

冒泡排序：

只有当arr[i]>arr[i+1]的时候，才会交换元素的位置，而相等的时候并不交换位置，所以冒泡排序是一种稳定排序算法。

选择排序:

选择排序是给每个位置选择当前元素最小的,例如有数据{5(1)，8 ，5(2)， 2， 9 },第一遍选择到的最小元素为2，所以5(1)会和2进行交换位置，此时5(1)到了5(2)后面，破坏了稳定性，所以选择排序是一种不稳定的排序算法。

插入排序：

比较是从有序序列的末尾开始，也就是想要插入的元素和已经有序的最大者开始比起，如果比它大则直接插入在其后面，否则一直往前找直到找到它该插入的位置。如果碰见一个和插入元素相等的，那么把要插入的元素放在相等元素的后面。所以，相等元素的前后顺序没有改变，从原无序序列出去的顺序就是排好序后的顺序，所以插入排序是稳定的。

希尔排序：

希尔排序是按照不同步长对元素进行插入排序,虽然一次插入排序是稳定的，不会改变相同元素的相对顺序，但在不同的插入排序过程中，相同的元素可能在各自的插入排序中移动，最后其稳定性就会被打乱，所以希尔排序是不稳定的。

归并排序：

归并排序在归并的过程中，只有arr[i]<arr[i+1]的时候才会交换位置，如果两个元素相等则不会交换位置，所以它并不会破坏稳定性，归并排序是稳定的。

快速排序：

快速排序需要一个基准值，在基准值的右侧找一个比基准值小的元素，在基准值的左侧找一个比基准值大的元素，然后交换这两个元素，此时会破坏稳定性，所以快速排序是一种不稳定的算法。