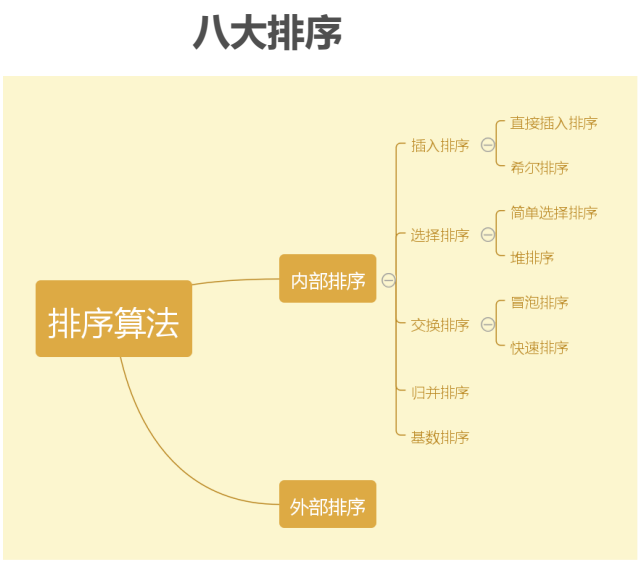
用java实现八大排序算法

2019年03月25日 19:53:44 在IT中穿梭旅行 阅读数 41 更多



一、直接插入排序

插入排序由于操作不尽相同，可分为 直接插入排序，折半插入排序(又称二分插入排序)，

链表插入排序，希尔排序 。我们先来看下直接插入排序。

1、基本思想

直接插入排序的基本思想是：将数组中的所有元素依次跟前面已经排好的元素相比较，

如果选择的元素比已排序的元素小，则交换，直到全部元素都比较过为止。

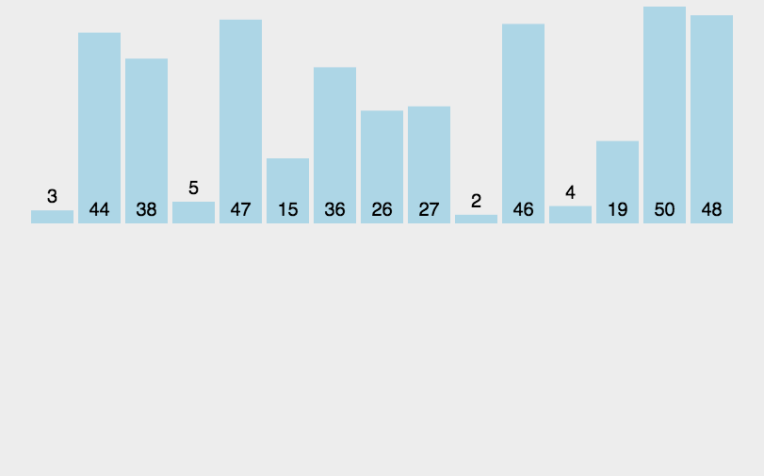
6 5 3 1 8 7 2 4

2、算法描述

一般来说，插入排序都采用in-place在数组上实现。具体算法描述如下：

①. 从第一个元素开始，该元素可以认为已经被排序
②. 取出下一个元素，在已经排序的元素序列中从后向前扫描
③. 如果该元素（已排序）大于新元素，将该元素移到下一位置
④. 重复步骤3，直到找到已排序的元素小于或者等于新元素的位置
⑤. 将新元素插入到该位置后

⑥. 重复步骤②~⑤



3、代码实现

```
import java.util.Arrays;
import java.util.Random;
public class InsertSort {
    public static void main(String[] args) {

        Random ran = new Random();
        int[] arr = new int[20];
        for (int i = 0; i < 20; i++) {
            arr[i] = ran.nextInt(100) + 1;
        }
        System.out.println(Arrays.toString(arr));
        sort(arr);
        System.out.println(Arrays.toString(arr));
    }
    public static void sort(int[] arr){
        int i, tmp;
        for( i=1;i<arr.length;i++){  //从第二个数字开始循环,
            tmp =arr[i];
             int  j =i-1;
             while(j>=0&&arr[j]>tmp){
                 arr[j+1] = arr[j];
                j--;
             }
            arr[j+1]= tmp;
            System.out.println("第" + i + "次" + Arrays.toString(arr));
        }
    }
}
```


平均时间复杂度	最好情况	最坏情况	空间复杂度
O(n²)	O(n)	O(n²)	O(1)

二、希尔排序（Shell Sort)

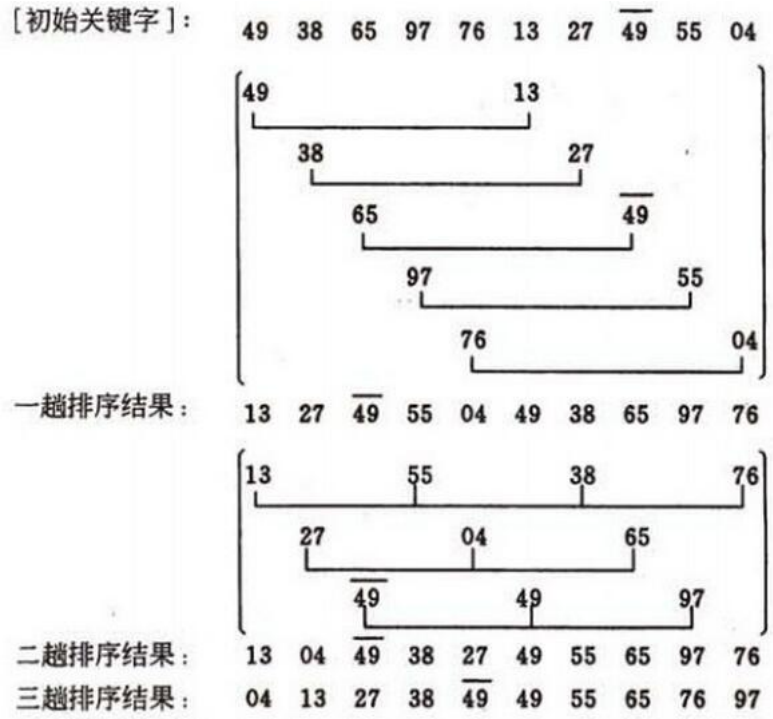
第一个突破O(n^2)的排序算法；是简单插入排序的改进版；

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。

希尔排序是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，

待整个序列中的记录“基本有序”时，再对全体记录进行依次直接插入排序。

1、基本思想



将待排序数组按照步长gap进行分组，然后将每组的元素利用直接插入排序的方法进行排序；

每次再将gap折半减小，循环上述操作；当gap=1时，利用直接插入，完成排序。

2、算法描述

1.选择一个增量序列t1，t2，…，tk，其中ti>tj，tk=1；（一般初次取数组半长，

之后每次再减半，直到增量为1）

2.按增量序列个数k，对序列进行k 趟排序；

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。

3、代码实现

```
1
2  import java.util.Arrays;
3  import java.util.Random;
4
5  /**
6   *@Author:759057893@qq.com Lyz
7   *@Date: 2019/4/2 15:51
8   *@Description:
9   **/
10
11 /*
12 希尔排序
13  */
14 public class ShellSort {
15
16     public static void main(String[] args) {
17         Random ran = new Random();
18         int[] arr = new int[10];
19         for (int i = 0; i < 10; i++) {
20             arr[i] = ran.nextInt(100) + 1;
21         }
22         System.out.println(Arrays.toString(arr));
23         shell_sort(arr);
```

```
        System.out.println(Arrays.toString(arr));
    }


    public static void shell_sort(int[] arr) {
        int gap = 1, i, j, len = arr.length;
        int temp;
        while (gap < len / 3)
            gap = gap * 3 + 1;      // <O(n^(3/2)) by Knuth,1973>: 1, 4, 13, 40, 121, ...
        for (; gap > 0; gap /= 3) {
            for (i = gap; i < len; i++) {
                temp = arr[i];
                for (j = i - gap; j >= 0 && arr[j] > temp; j -= gap)
                    arr[j + gap] = arr[j];
                arr[j + gap] = temp;
            }
        }
    }
}
```

平均时间复杂度	最好情况	最坏情况	空间复杂度
O(nlog2 n)	O(nlog2 n)	O(nlog2 n)	O(1)

三、选择排序（Selection Sort）

	8
	5
	2
	6
	9
	3
	1
	4
	0
	7

1、基本思想

选择排序的基本思想：比较 + 交换。

2、算法描述

①. 从待排序序列中，找到关键字最小的元素；
②. 如果最小元素不是待排序序列的第一个元素，将其和第一个元素互换；
③. 从余下的 N - 1 个元素中，找出关键字最小的元素，重复①、②步，直到排序结束。

3、代码实现

```
/*
选择排序
 */
public class SelectSort {
    public static void main(String[] args) {
        Random ran = new Random();
        int arr[] = new int[20];
        for(int i= 0 ; i<20;i++){
            arr[i] = ran.nextInt(100)+1;
        }
        System.out.println(Arrays.toString(arr));

        SelectSort.sort(arr);
```

```
        System.out.println(Arrays.toString(arr));
    }


    public static void sort(int[] arr){
        int mixindex = 0;
        int temp = 0;
        for (int i = 0; i < arr.length - 1; i++) {
            mixindex = i;//假设最小元素的下标
            for (int j = i + 1; j < arr.length; j++) {
                if (arr[j] < arr[mixindex]) {
                    mixindex = j;
                }
            }
            if (mixindex != i) {
                temp = arr[i];
                arr[i] = arr[mixindex];
                arr[mixindex] = temp;
            }
            System.out.println("第" + i + "次" + Arrays.toString(arr));
        }
    }
}
```

平均时间复杂度	最好情况	最坏情况	空间复杂度
O(n²)	O(n²)	O(n²)	O(1)

四、堆排序（Heap Sort）

堆的定义如下：n个元素的序列{k1,k2,···,kn}，当且仅当满足下关系时，称之为堆。

ki <= k(2i) 或 ki >= k(2i)
ki <= k(2i+1) ki >= k(2i+1)

堆的含义就是：**完全二叉树中任何一个非叶子节点的值均不大于（或不小于）**

其左，右孩子节点的值。

堆分为大顶堆和小顶堆，

满足Key[i]>=Key[2i+1]&&key>=key[2i+2]称为**大顶堆**，

满足 Key[i]<=key[2i+1]&&Key[i]<=key[2i+2]称为**小顶堆**。

1、基本思想

此处以大顶堆为例，堆排序的过程就是将待排序的序列构造成一个堆，

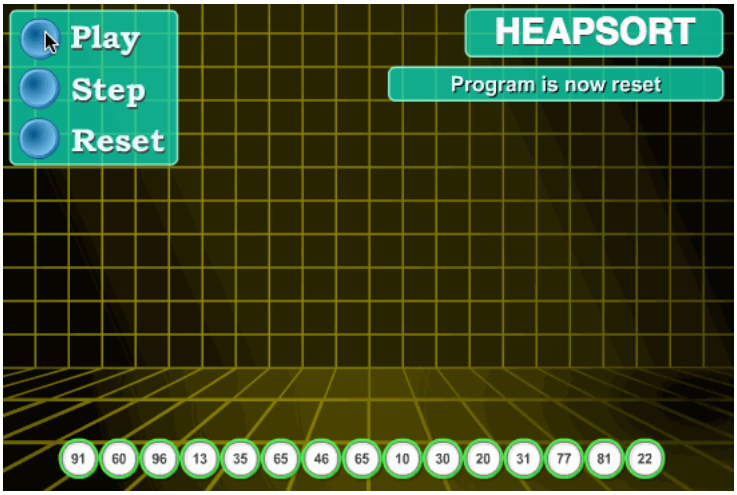
选出堆中最大的移走，再把剩余的元素调整成堆，找出最大的再移走，重复直至有序

使用大顶堆进行升序排序，使用小顶堆进行降序排序

2、算法描述

- ①. 先将初始序列K[1..n]建成一个大顶堆, 那么此时第一个元素K1最大, 此堆为初始的无序区.
- ②. 再将关键字最大的记录K1 (即堆顶, 第一个元素)和无序区的最后一个记录 Kn 交换,
- 由此得到新的无序区K[1..n-1]和有序区K[n], 且满足K[1..n-1].keys <= K[n].key
- ③. 交换K1 和 Kn 后, 堆顶可能违反堆性质, 因此需将K[1..n-1]调整为堆. 然后重复步骤②,
- 直到无序区只有一个元素时停止.

动图效果如下所示：



3、代码实现

- 从算法描述来看，堆排序需要两个过程，一是建立堆，二是堆顶与堆的最后一个元素交换位置。
- 所以堆排序有两个函数组成。
- 一是建堆函数，
- 二是反复调用建堆函数以选择出剩余未排元素中最大的数来实现排序的函数。

总结起来就是定义了以下几种操作：

- 最大堆调整（Max_Heapify）：将堆的末端子节点作调整，使得子节点永远小于父节点
- 创建最大堆（Build_Max_Heap）：将堆所有数据重新排序
- 堆排序（HeapSort）：移除位在第一个数据的根节点，并做最大堆调整的递归运算

对于堆节点的访问：

- 父节点i的左子节点在位置：**(2*i+1);**
- 父节点i的右子节点在位置：**(2*i+2);**
- 子节点i的父节点在位置：**floor((i-1)/2);**


```
/**
 * 堆排序
 *
 * 1. 先将初始序列K[1..n]建成一个大顶堆, 那么此时第一个元素K1最大, 此堆为初始的无序区.
 * 2. 再将关键字最大的记录K1 (即堆顶, 第一个元素)和无序区的最后一个记录 Kn 交换,
 *由此得到新的无序区K[1..n-1]和有序区K[n], 且满足K[1..n-1].keys≤K[n].key
 * 3. 交换K1 和 Kn 后, 堆顶可能违反堆性质, 因此需将K[1..n-1]调整为堆. 然后重复步骤2,
 *直到无序区只有一个元素时停止.
 * @param arr  待排序数组
 */
public static void heapSort(int[] arr){
    for(int i = arr.length; i > 0; i--){
        max_heapify(arr, i);

        int temp = arr[0];      //堆顶元素(第一个元素)与Kn交换
        arr[0] = arr[i-1];
        arr[i-1] = temp;
    }
}

private static void max_heapify(int[] arr, int limit){
    if(arr.length <= 0 || arr.length < limit) return;
    int parentIdx = limit / 2;

    for(; parentIdx >= 0; parentIdx--){
        if(parentIdx * 2 >= limit){
            continue;
        }
        int left = parentIdx * 2;       //左子节点位置
```

```
        int right = (left + 1) >= limit ? left : (left + 1);    //右子节点位置，如果没有右节点，默认为左节点位置
        int maxChildId = arr[left] >= arr[right] ? left : right;
        if(arr[maxChildId] > arr[parentIdx]){   //交换父节点与左右子节点中的最大值
            int temp = arr[parentIdx];
            arr[parentIdx] = arr[maxChildId];
            arr[maxChildId] = temp;
        }
    }
    System.out.println("Max_Heapify: " + Arrays.toString(arr));
}
```

注: x>>1 是位运算中的右移运算, 表示右移一位, 等同于x除以2再取整, 即 x>>1 == Math.floor(x/2) .

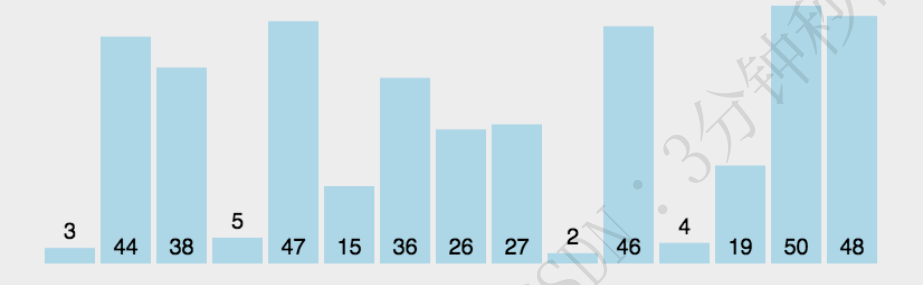
以上,
①. 建立堆的过程, 从length/2 一直处理到0, 时间复杂度为O(n);
②. 调整堆的过程是沿着堆的父子节点进行调整, 执行次数为堆的深度, 时间复杂度为O(lgn);
③. 堆排序的过程由n次第②步完成, 时间复杂度为O(nlgn).

平均时间复杂度	最好情况	最坏情况	空间复杂度
$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n\log_2 n)$	O(1)

五、冒泡排序（Bubble Sort）

1、基本思想

冒泡排序（Bubble Sort）是一种简单的排序算法。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
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。



2、算法描述

冒泡排序算法的运作如下：

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①~③，直到没有任何一对数字需要比较。

3、代码实现

```
import java.util.Arrays;
import java.util.Random;

/**
 *@Author:759057893@qq.com Lyz
 *@Date: 2019/3/28 17:58
 *@Description:
 **/

/*
    冒泡排序
 */
public class BubblingSort {

    public static void main(String[] args) {
        Random ran = new Random();
        int[] arr = new int[5];
```

```
        for (int i = 0; i < 5; i++) {19            arr[i] = ran.nextInt(50) + 1;
        }
        System.out.println(Arrays.toString(arr));

        bubblingSort(arr);
        System.out.println(Arrays.toString(arr));
    }

    public static void bubblingSort(int[] arr){
        int mub;
        for (int i = 0; i < arr.length - 1; i++) {
            for (int j = 0; j < arr.length - i - 1; j++) {
                if (arr[j + 1] > arr[j]) {
                    mub = arr[j + 1];
                    arr[j + 1] = arr[j];
                    arr[j] = mub;
                }
            }
            System.out.println("第" + i + "次" + Arrays.toString(arr));
        }

    }
}
```

平均时间复杂度	最好情况	最坏情况	空间复杂度
O(n²)	O(n)	O(n²)	O(1)

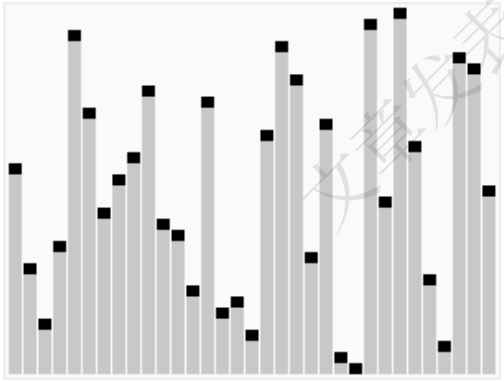
六、快速排序（Quick Sort）

快速排序（Quicksort）是对冒泡排序的一种改进，借用了分治的思想，由C. A. R. Hoare在1962年提出。

1、基本思想

快速排序的基本思想：**挖坑填数+分治法。**

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。



2、算法描述

快速排序使用分治策略来把一个序列（list）分为两个子序列（sub-lists）。步骤为：

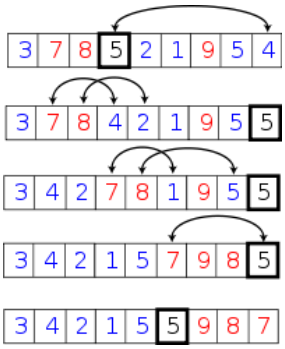
①. 从数列中挑出一个元素，称为”基准”（pivot）。
②. 重新排序数列，所有比基准值小的元素摆放在基准前面，所有比基准值大的元素摆在基准后面（相同的数可以到任一边）。在这个分区结束之后，i数列的中间位置。这个称为分区（partition）操作。
③. 递归地（recursively）把小于基准值元素的子数列和大于基准值元素的子数列排序。

递归到最底部时，数列的大小是零或一，也就是已经排序好了。这个算法一定会结束，因为在每次的迭代（iteration）中，它至少会把一个元素摆到它去。

3、代码实现

用伪代码描述如下：

①. **i = L; j = R;** 将基准数挖出形成第一个坑**a[i]**。
②. **j--**，由后向前找比它小的数，找到后挖出此数填前一个坑a[i]中。
③. **i++**，由前向后找比它大的数，找到后也挖出此数填到前一个坑**a[j]**中。
④. 再重复执行②，③二步，直到**i==j**，将基准数填入**a[i]**中



快速排序具体介绍：https://blog.csdn.net/code_AC/article/details/74158681

```
/* 递归*/

private static void quickSort(int[] arr,int left,int right){
       if(arr==null||arr.length<2||left>right){
           return;
       }
       int l = left;
       int r = right;
       int key = arr[left]; //以第一个数作为基准
        while(l<r){
            while(l<r&&arr[r]>=key){
                r--;
            }
            arr[l] = arr[r];
            while(l<r&&arr[l]<=key){
                l++;
            }
            arr[r] = arr[l];
        }
        arr[l] = key;
        quickSort(arr,left,l-1);
        quickSort(arr,l+1,right);
    }
```

上面是递归版的快速排序：通过把基准temp插入到合适的位置来实现分治，并递归地对分治后的两个划分继续快排。

那么非递归版的快排如何实现呢?

因为递归的本质是栈，所以我们非递归实现的过程中，可以借助栈来保存中间变量就可以实现非递归了。

在这里中间变量也就是通过Pritation函数划分区间之后分成左右两部分的首尾指针，只需要保存这两部分的首尾指针即可。

```
/**
 * 快速排序（非递归）
 *
 * ①. 从数列中挑出一个元素，称为"基准"（pivot）。
 * ②. 重新排序数列，所有比基准值小的元素摆放在基准前面，所有比基准值大的元素摆在基准后面
 *（相同的数可以到任一边）。在这个分区结束之后，该基准就处于数列的中间位置。
 * 这个称为分区（partition）操作。
 * ③. 把分区之后两个区间的边界（low和high）压入栈保存，并循环①、②步骤
 * @param arr    待排序数组
 */
public static void quickSortByStack(int[] arr){
    if(arr.length <= 0) return;
    Stack<Integer> stack = new Stack<Integer>();

    //初始状态的左右指针入栈
```

```
    stack.push(0);
    stack.push(arr.length - 1);
    while(!stack.isEmpty()){
        int high = stack.pop();     //出栈进行划分
        int low = stack.pop();

        int pivotIdx = partition(arr, low, high);

        //保存中间变量
        if(pivotIdx > low) {
            stack.push(low);
            stack.push(pivotIdx - 1);
        }
        if(pivotIdx < high && pivotIdx >= 0){
            stack.push(pivotIdx + 1);
            stack.push(high);
        }
    }
}

private static int partition(int[] arr, int low, int high){
    if(arr.length <= 0) return -1;
    if(low >= high) return -1;
    int l = low;
    int r = high;

    int pivot = arr[l];    //挖坑1：保存基准的值
    while(l < r){
        while(l < r && arr[r] >= pivot){  //坑2：从后向前找到比基准小的元素，插入到基准位置坑1中
            r--;
        }
        arr[l] = arr[r];
        while(l < r && arr[l] <= pivot){   //坑3：从前往后找到比基准大的元素，放到刚才挖的坑2中
            l++;
        }
        arr[r] = arr[l];
    }
    arr[l] = pivot;   //基准值填补到坑3中，准备分治递归快排
    return l;
}
```

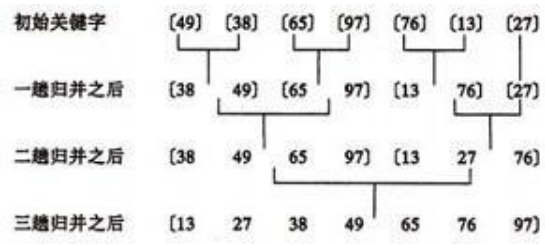
快速排序是通常被认为在同数量级（O(nlog2n)）的排序方法中平均性能最好的。但若初始序列按关键码有序或基本有序时，

快排序反而蜕化为冒泡排序。为改进之，通常以“三者取中法”来选取基准记录，

即将排序区间的两个端点与中点三个记录关键码居中的调整为支点记录。**快速排序是一个不稳定的排序方法。**
快速排序算法复杂度:

平均时间复杂度	最好情况	最坏情况	空间复杂度
O(nlog₂n)	O(nlog₂n)	O(n²)	O(1)（原地分区递归版）

七、归并排序（Merging Sort）



归并排序是建立在归并操作上的一种有效的排序算法，1945年由约翰·冯·诺伊曼首次提出。

该算法是采用分治法（Divide and Conquer）的一个非常典型的应用，且各层分治递归可以同时进行。

1、基本思想

归并排序算法是将两个（或两个以上）有序表合并成一个新的有序表，即把待排序序列分为若干个子序列，每个子序列是有序的。然后再把有序子序列合并为整体有序序列。

6 5 3 1 8 7 2 4

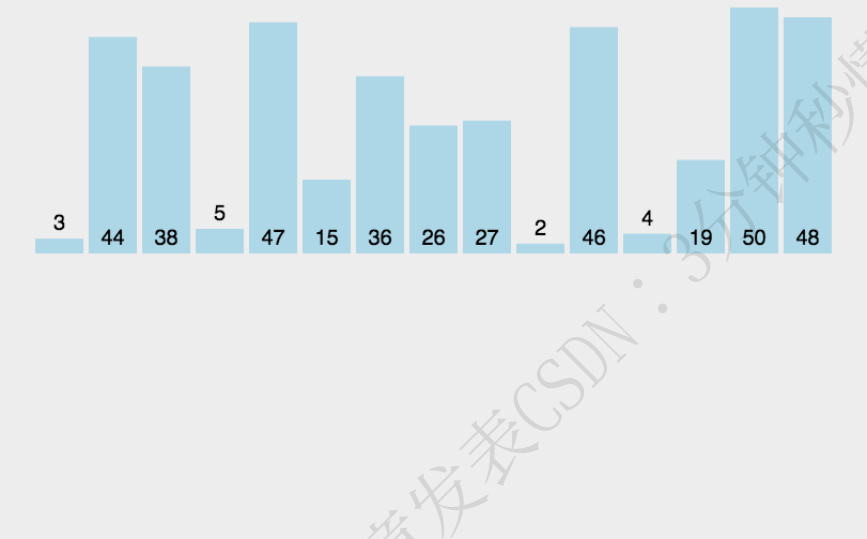
2、算法描述

归并排序可通过两种方式实现：

- 自上而下的递归
- 自下而上的迭代

一、递归法（假设序列共有n个元素）：

①. 将序列每相邻两个数字进行归并操作，形成 floor(n/2)个序列，排序后每个序列包含两个元素；
②. 将上述序列再次归并，形成 floor(n/4)个序列，每个序列包含四个元素；
③. 重复步骤②，直到所有元素排序完毕。



二、迭代法

①. 申请空间，使其大小为两个已经排序序列之和，该空间用来存放合并后的序列
②. 设定两个指针，最初位置分别为两个已经排序序列的起始位置
③. 比较两个指针所指向的元素，选择相对小的元素放入到合并空间，并移动指针到下一位置
④. 重复步骤③直到某一指针到达序列尾
⑤. 将另一序列剩下的所有元素直接复制到合并序列尾

3、代码实现

归并排序其实要做两件事：

- 分解：将序列每次折半拆分
- 合并：将划分后的序列段两两排序合并

因此，归并排序实际上就是两个操作，拆分+合并

如何分解?

在这里，我们采用递归的方法，首先将待排序列分成A,B两组；然后重复对A、B序列分组；直到分组后组内只有一个元素，此时我们认为组内所有元素有序，则分组结束。

如何合并?

L[first...mid]为第一段，L[mid+1...last]为第二段，并且两端已经有序，现在我们要将两端合成达到L[first...last]并且也有序。

首先依次从第一段与第二段中取出元素比较，将较小的元素赋值给temp[]
重复执行上一步，当某一段赋值结束，则将另一段剩下的元素赋值给temp[]
此时将temp[]中的元素复制给L[]，则得到的L[first...last]有序

```
/**
 * 归并排序（递归）
 *
 * ①. 将序列每相邻两个数字进行归并操作，形成 floor(n/2) 个序列，排序后每个序列包含两个元素;
 * ②. 将上述序列再次归并，形成 floor(n/4) 个序列，每个序列包含四个元素;
 * ③. 重复步骤②，直到所有元素排序完毕。
 * @param arr     待排序数组
 */

    public static void main(String[] args) {
        int[] arrays = {9, 2, 5, 1, 3, 2, 9, 5, 2, 1, 8};
        mergeSort(arrays, 0, arrays.length - 1);

        System.out.println("公众号：Java3y" + arrays);


    }

    /**
     * 归并排序
     *
     * @param arrays
     * @param L       指向数组第一个元素
     * @param R       指向数组最后一个元素
     */
    public static void mergeSort(int[] arrays, int L, int R) {

        //如果只有一个元素，那就不用排序了
        if (L == R) {
            return;
        } else {

            //取中间的数，进行拆分
            int M = (L + R) / 2;

            //左边的数不断进行拆分
            mergeSort(arrays, L, M);

            //右边的数不断进行拆分
            mergeSort(arrays, M + 1, R);

            //合并
            merge(arrays, L, M + 1, R);

        }
    }


    /**
     * 合并数组
     *
     * @param arrays
     * @param L       指向数组第一个元素
     * @param M       指向数组分隔的元素
     * @param R       指向数组最后的元素
     */
    public static void merge(int[] arrays, int L, int M, int R) {

        //左边的数组的大小
        int[] leftArray = new int[M - L];

        //右边的数组大小
```

```
        int[] rightArray = new int[R - M + 1];

        //往这两个数组填充数据
        for (int i = L; i < M; i++) {
            leftArray[i - L] = arrays[i];
        }
        for (int i = M; i <= R; i++) {
            rightArray[i - M] = arrays[i];
        }


        int i = 0, j = 0;
        // arrays数组的第一个元素
        int  k = L;


        //比较这两个数组的值，哪个小，就往数组上放
        while (i < leftArray.length && j < rightArray.length) {

            //谁比较小，谁将元素放入大数组中,移动指针，继续比较下一个
            if (leftArray[i] < rightArray[j]) {
                arrays[k] = leftArray[i];

                i++;
                k++;
            } else {
                arrays[k] = rightArray[j];
                j++;
                k++;
            }
        }

        //如果左边的数组还没比较完，右边的数都已经完了，那么将左边的数抄到大数组中(剩下的都是大数字)
        while (i < leftArray.length) {
            arrays[k] = leftArray[i];

            i++;
            k++;
        }
        //如果右边的数组还没比较完，左边的数都已经完了，那么将右边的数抄到大数组中(剩下的都是大数字)
        while (j < rightArray.length) {
            arrays[k] = rightArray[j];

            k++;
            j++;
        }
    }
```


八、基数排序（Radix Sort）

基数排序（Radix sort）是一种非比较型整数排序算法，

其原理是**将整数按位数切割成不同的数字，然后按每个位数分别比较。**由于整数也可以表达字符串（比如名字或日期）

和特定格式的浮点数，所以基数排序也不是只能使用于整数

1、基本思想

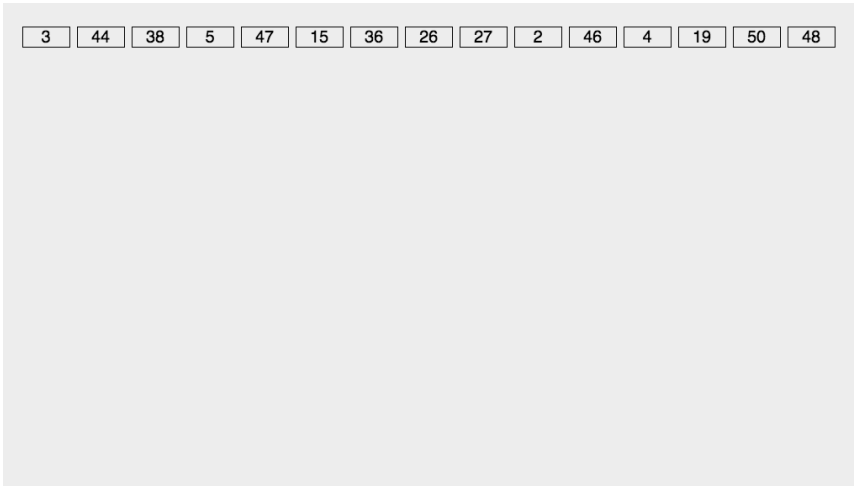
它是这样实现的：将所有待比较数值（正整数）统一为同样的数位长度，数位较短的数前面补零。

然后，从最低位开始，依次进行一次排序。这样从最低位排序一直到最高位排序完成以后，数列就变成一个有序序列。

基数排序按照优先从高位或低位来排序有两种实现方案：

MSD（Most significant digital）从最左侧高位开始进行排序。先按k1排序分组, 同一组中记录, 关键码k1相等, 再对各组按k2排序分成子组, 之后, 对
码继续这样的排序分组, 直到按最次位关键码kd对各子组排序后. 再将各组连接起来, 便得到一个有序序列。MSD方式适用于位数多的序列。

LSD （Least significant digital）从最右侧低位开始进行排序。先从kd开始排序，再对kd-1进行排序，依次重复，直到对k1排序后便得到一个有序序
式适用于位数少的序列。



2、算法描述

我们**以LSD为例**，从最低位开始，具体算法描述如下：

①. 取得数组中的最大数，并取得位数；
②. arr为原始数组，从最低位开始取每个位组成radix数组；
③. 对radix进行计数排序（利用计数排序适用于小范围数的特点）；

3、代码实现

基数排序：通过序列中各个元素的值，对排序的N个元素进行若干趟的“分配”与“收集”来实现排序。

- **分配：**我们将L[i]中的元素取出，首先确定其个位上的数字，根据该数字分配到与之序号相同的桶中
- **收集：**当序列中所有的元素都分配到对应的桶中，再按照顺序依次将桶中的元素收集形成新的一个待排序列L[]。
- 对新形成的序列L[]重复执行分配和收集元素中的十位、百位…直到分配完该序列中的最高位，则排序结束

```
/**
 * 基数排序（LSD 从低位开始）
 *
 * 基数排序适用于：
 *  (1)数据范围较小，建议在小于1000
 *  (2)每个数值都要大于等于0
 *
 * ①. 取得数组中的最大数，并取得位数；
 * ②. arr为原始数组，从最低位开始取每个位组成radix数组；
 * ③. 对radix进行计数排序（利用计数排序适用于小范围数的特点）；
 * @param arr     待排序数组
 */
public static void radixSort(int[] arr){
    if(arr.length <= 1) return;

    //取得数组中的最大数，并取得位数
    int max = 0;
    for(int i = 0; i < arr.length; i++){
        if(max < arr[i]){
            max = arr[i];
        }
    }
    int maxDigit = 1;
    while(max / 10 > 0){
        maxDigit++;
        max = max / 10;
    }
    System.out.println("maxDigit: " + maxDigit);

    //申请一个桶空间
    int[][] buckets = new int[10][arr.length-1];
    int base = 10;

    //从低位到高位，对每一位遍历，将所有元素分配到桶中
```

```
    for(int i = 0; i < maxDigit; i++){
        int[] bktLen = new int[10];        //存储各个桶中存储元素的数量

        //分配：将所有元素分配到桶中
        for(int j = 0; j < arr.length; j++){
            int whichBucket = (arr[j] % base) / (base / 10);
            buckets[whichBucket][bktLen[whichBucket]] = arr[j];
            bktLen[whichBucket]++;
        }

        //收集：将不同桶里数据挨个捞出来,为下一轮高位排序做准备,由于靠近桶底的元素排名靠前,因此从桶底先捞
        int k = 0;
        for(int b = 0; b < buckets.length; b++){
            for(int p = 0; p < bktLen[b]; p++){
                arr[k++] = buckets[b][p];
            }
        }

        System.out.println("Sorting: " + Arrays.toString(arr));
        base *= 10;
    }
}
```

基数排序算法复杂度，其中k为最大数的位数：

平均时间复杂度	最好情况	最坏情况	空间复杂度
O(d*(n+r))	O(d*(n+r))	O(d*(n+r))	O(n+r)

其中，d 为位数，r 为基数，n 为原数组个数。

在基数排序中，因为没有比较操作，所以在复杂上，最好的情况与最坏的情况在时间上是一致的，**均为 O(d*(n + r))。**

基数排序更适合用于对时间, 字符串等这些整体权值未知的数据进行排序。

Tips: 基数排序不改变相同元素之间的相对顺序，因此它是稳定的排序算法。

基数排序 vs 计数排序 vs 桶排序

这三种排序算法都利用了桶的概念，但对桶的使用方法上有明显差异：

1. 基数排序：根据键值的每位数字来分配桶
2. 计数排序：每个桶只存储单一键值
3. 桶排序：每个桶存储一定范围的数值

总结

八大排序算法耗时比较

排序类型	平均情况	最好情况	最坏情况	辅助空间	稳定性
冒泡排序	$O(n^2)$	O(n)	$O(n^2)$	O(1)	稳定
选择排序	$O(n^2)$	$O(n^2)$	$O(n^2)$	O(1)	不稳定
直接插入排序	$O(n^2)$	O(n)	$O(n^2)$	O(1)	稳定
折半插入排序	$O(n^2)$	O(n)	$O(n^2)$	O(1)	稳定
希尔排序	O(n^1.3)	O(nlogn)	$O(n^2)$	O(1)	不稳定
归并排序	$O(nlog_2n)$	$O(nlog_2n)$	$O(nlog_2n)$	O(n)	稳定
快速排序	$O(nlog_2n)$	$O(nlog_2n)$	$O(n^2)$	$O(nlog_2n)$	不稳定
堆排序	$O(nlog_2n)$	$O(nlog_2n)$	$O(nlog_2n)$	O(1)	不稳定
计数排序	O(n+k)	O(n+k)	O(n+k)	O(k)	稳定
桶排序	O(n+k)	O(n+k)	$O(n^2)$	O(n+k)	(不)稳定
基数排序	O(d(n+k))	O(d(n+k))	O(d(n+kd))	O(n+kd)	稳定

常用排序算法							
类别	排序方法	时间复杂度			空间复杂度	稳定性	备注
		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	n小时较好
	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	n小时较好
	堆排序	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(1)$	不稳定	n大时较好
交换排序	冒泡排序	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$	稳定	n小时较好
	快速排序	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n^2)$	$O(n\log_2 n)$	不稳定	n大时较好
归并排序		$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(1)$	稳定	n大时较好
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							