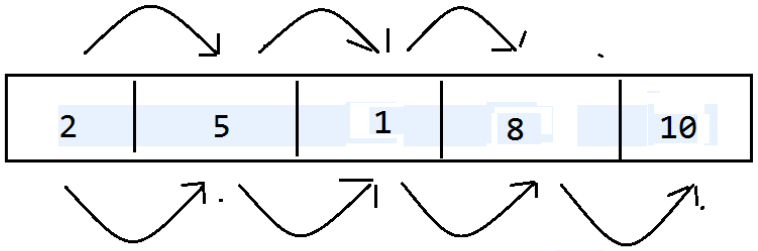
八大排序算法

- 不稳定的排序算法：快速排序、希尔排序、堆排序、选择排序**(简记：快些选堆)**
- 所需辅助空间最多：归并排序
- 所需辅助空间最少：堆排序
- 平均速度最快：快速排序
- 当n较大，则应采用时间复杂度为O(nlogn)的排序方法：快速排序、堆排序或归并排序。
- 时间复杂度：冒泡排序=选择排序=插入排序=O(N的平方)；其他都是O(NlogN)，但是并不是绝对的。

1.冒泡排序

- 原理：由大到小排序--相邻两个元素进行比较，大的元素放左边，小的元素放右边，一轮循环后，最小的元素在最右边。
- **稳定排序算法：能保证排序前后，2个相等的数其在序列的前后位置顺序和排序后它们两个的前后位置顺序相同**



思路： 相邻的两个元素比较，符合条件交换位置。

一轮 大的放在右边，小的放在左边

```
//把最大值放在最后一个位置
for(int i = 0 ; i<arr.length-1 ; i++){ // 4
      //相邻的两个元素比较
      if(arr[i] >arr[i+1]){
            int temp =arr[i];
            arr[i] = arr[i+1];
            arr[i+1] = temp;
      }
}
```

```
for(int i = 0 ; i<arr.length-1-1 ; i++){
            //相邻的两个元素比较
            if(arr[i] >arr[i+1]){
                  int temp =arr[i];
                  arr[i] = arr[i+1];
                  arr[i+1] = temp;
            }
      }
```

- 实现

```
    // 冒泡排序--由大到小
    public static void bubbleSort(int[] arr) {
        for (int i = 0; i < arr.length-1; i++) {
```

```
            for (int j = 1; j < arr.length - i; j++) {
                if (arr[j-1] < arr[j]) {
                    int temp = arr[j];
                    arr[j] = arr[j-1];
                    arr[j-1] = temp;
                }
            }
        }
    }
```

- 时间复杂度：O(N的平方)，较慢，稳定算法，最坏情况和最好情况都是一样的复杂度

2.选择排序(直接排序)

- 原理：使用索引为0的元素与其他的元素比较，如果有比索引0大的元素，则记录该元素下标，一轮比较完成后，下标值是最大值，然后进行一次比较，通过一轮循环，最大的元素在索引为0的位置。然后使用索引为1的元素和其余元素进行比较，依次循环。
- 比较次数和冒泡一样，但是交换次数变少。每一轮循环只交换一次。
- 选择排序对冒泡排序进行了改进，将元素的交换次数从O(N的平方)降为O(N)，但是比较次数没有变化，依然是O(N的平方)。
- 它与冒泡排序的比较次数相同，但选择排序的交换次数少于冒泡排序。冒泡排序是在每次比较之后，若比较的两个元素顺序与待排序顺序相反，则要进行交换，而选择排序在每次遍历过程中只记录下来最小的一个元素的下标，待全部比较结束之后，将最小的元素与未排序的那部分序列的最前面一个元素交换，这样就降低了交换的次数，提高了排序效率。
- 实现
  #
  // 选择排序（直接排序）--从大到小排序
  public static void selectSort(int[] arr) {

```
    //每次循环，比较次数和冒泡一样，但是只交换一次，提高效率
    for (int i = 0; i < arr.length-1; i++) {
        int lowIndex = i;

        for (int j = i + 1; j < arr.length; j++) {
            if (arr[lowIndex] < arr[j]) {
                lowIndex = j;
            }
        }
        // 将当前第一个元素与它后面序列中的最大的元素交换，也就是将最大的元素放在最前端
        int temp = arr[i];
        arr[i] = arr[lowIndex];
        arr[lowIndex] = temp;
    }
```

  }

- 时间复杂度：O(N的平方)，不稳定算法，最坏情况和最好情况都是一样的复杂度

3.插入排序

- 原理：非常类似于平时打牌时候插入牌的过程。开始摸牌时，我们的左手是空的，接着一次从桌上摸起一张牌，并将它插入到左手的正确位置，为了找到这张牌的正确位置，要将它与手中已有的牌从左到右进行比较，无论什么时候手中的牌都是排序好的。
- 分析：对于一个给定要排序的数组，以大牌来分析，当摸第一张牌时候(下标为0的元素)，不用排序，直接插入；当摸第二张牌时候（下标为1的元素），将这张牌和第一张牌进行比较，如果更大，则将第1张牌右移一个位置，也就是第一张牌现在空了一个位置出来，然后第二张牌插入到右边，这样就完成了目前所有牌的排序了，后面每次来一张牌都要和已排序的牌进行比较，然后原来的牌要空出一个位置让新来的牌有位置插入。
- 插入排序法的排序思想就是从数组的第二个元素开始,将数组中的每一个元素按照规则插入到已排好序的数组中以达到排序的目的.一般情况下将数组的第一个元素作为起始元素,从第二个元素开始依次插入.由于要插入到的数组是已经排好序的,所以只是要从右向左找到比插入点小(对升序而言)的第一个数组元素就插入到其后面.直到将最后一个数组元素插入到数组中,整个排序过程就算完成。

- 不管何种排序方式，查找比较规则都是从右到左
- 一般情况下， 该算法比冒泡排序快一倍，比选择排序要快一些
- 对于有序或者基本有序的数据，插入排序非常快，可以达到O(N)
- 如果数据是逆序的，那么插入排序比冒泡排序还慢
- 实现

  #

  //插入排序--由大到小

  public static void insertSort(int[] arr){

```
int j;
int key;
for (int i = 1; i < arr.length; i++) {
    key=arr[i];//key就是我们每次新摸到的牌，现在需要找到插入的位置并空出位置
    j=i-1;
    while (j>=0 && arr[j]<key) {//将已排好序的所有比key小的元素右移一个位置，空出一个位置，方便key插入
        arr[j+1]=arr[j];
        j--;
    }
    arr[j+1]=key;//空出位置后，然后插入
}
```

  }
- 时间复杂度：O(N的平方)，稳定算法
- 就上面三种算法比较的话，插入排序是最好的，适合小数据量。
- 以上三种称为简单排序算法
- 最优复杂度：当输入数组就是排好序的时候，复杂度为O(n)，而快速排序在这种情况下会产生O(n^2)的复杂度
- 最差复杂度：当输入数组为倒序时，复杂度为O(n^2)
- 插入排序比较适合用于“少量元素的数组”

4.希尔排序

- 原理：是基于插入排序的改进算法，插入排序每次移动都是一个一个移动，当逆序时候，移动次数要很多，希尔排序改进了移动步长，可以实现大跨度移动，减少移动时间。希尔排序也叫做缩小增量排序法
- 将整个无序数据分割成若干小的子序列分别进行插入排序
- 在希尔排序开始时增量较大，分组较多，每组的记录数目少，故各组内直接插入较快，后来增量di逐渐缩小，分组数逐渐减少，而各组的记录数目逐渐增多，但由于已经按di-1作为距离排序，使文件较接近于有序状态，所以新的一趟排序过程也较快。
- 在插入算法中，如果有一个最小的数在数组的最后面，用插入算法就会从最后一个位置移动到第一个，这样就会浪费很大，使用这个改进的希尔排序可以实现数据元素的大跨度的移动。也就是这个算法的优越之处。
- 先取一个小于n的整数d1作为一个增量，把表的全部记录分成d1个组，所有距离为d1的倍数的记录放在同一个组中，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，直至所取的dt=1,即所有记录放在同一组中进行直接插入排序为止。
- 举例：要排序的数组是[3 5 4 12 45 3 -32 7 5 8 34 -4],长度为12，首先确定最大增量，增量的值选取对运行效率影响很大，这里面有很多间隔划分算法。我们这里选择的增量为长度/2，这里=6，故开始分组 每隔6个元素一组，分组如下：第一组【3 -32】第二组【5 7】第三组【4 5】第四组【12 8】第五组【45 34】 第六组【3 -4】，然后对每一组进行插入排序，从大到小，经过第一次排序后，变为【3 7 5 12 45 3 -32 5 4 8 34 -4】，然后再次修改增量，这次增量要缩小，我们再次是上次增量/2=3；然后开始第二次分组：第一组【3 12 -32 8】第二组【7 45 5 34】第三组【5 3 4 -4】，然后对每组使用插入排序，排序后合成，则数组变为【12 45 5 8 34 4 3 7 3 -32 5 -4】，从这里可以看出，已经基本有序了。最后增量/2=1;表示不需要再分组了，对上述数据进行一次插入排序即可。
- 实现

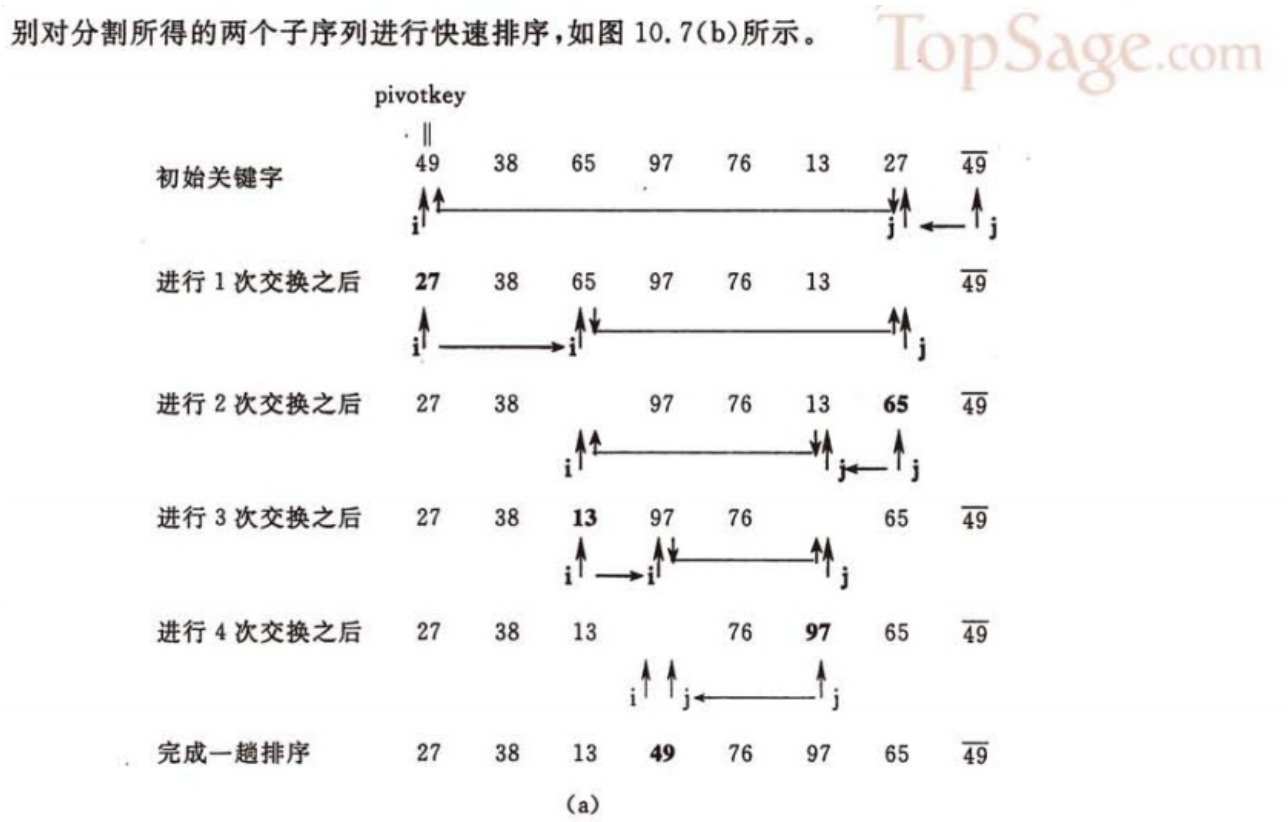
```
//希尔排序--由大到小
public static void shellSort(int[] arr) {
    int size = arr.length;
    //这层for是分组
    for(int gap = size/2; gap>=1; gap /= 2) {
        //这层for是对每组进行插入排序
        for(int i=gap; i<size; i++) {
```

```
            int temp = arr[i];
            int k=i-gap;
            while (k>=0 && arr[k]<temp) {
                arr[k+gap]=arr[k];
                k=k-gap;
            }
            arr[k+gap] = temp;
        }
    }
}
```

- 时间复杂度： O(nlog n)，不稳定算法，该算法复杂度并不是一直是这样，有可能和插入排序一样。

5.快速排序--重点

- 原理：通过一趟排序将待排序记录分割成独立的两部分，其中一部分记录的关键字均比另一部分关键字小，则分别对这两部分继续进行排序，直到整个序列有序。
- 分治法+挖坑填数法
- 介绍的比较好的博客：http://blog.csdn.net/wangkuifeng0118/article/details/7286332
- 简图：



- 对上图分析--由小到大排序
  - 开始，选择第一个数49作为基准数，把它挖出来，那么这个位置就是空的。i=0,j=6,X=49
  - 第一步，将剩下的数从右往左，找到第一个小于基准49的数，是27，那么将27填入到上次挖掉的坑中即a[0]位置，此时a[0]填了数了，从而a[6]是空的。i=i+1=1,j=6,X=49 【27 38 65 97 76 13 空】
  - 第二步,将新的数从新的i和j开始，左到右找到一个大于49的数，是a[2]=65,将该数填入上一个坑中a[6],此时a[2]为空，i=1,j=j-1=5,X=49,【27 38 空 97 76 13 65】
  - 第三步，重复第一步，使用新的i和j进行运算，可得【27 38 13 97 76 空 65】 i=2,j=5,X=49
  - 第四步，重复第二步，使用新的i进而j进行运算，可得【27 36 13 空 76 97 65】，i=2,j=4,X=49
  - 重复第一步，可得【27 36 13 76 空 97 65】，i=3,j=4;X=49
  - 重复第二步，可得【27 36 13 空 76 97 65】，i=3,j=3,X=49
  - 此时i=j，循环退出，然后将49填入空位置可得【27 36 13 49 76 97 65】可以看出，基准数左边全是小于49的，右边全是大于49的，

    - 然后以目前的i为分解线，分成2组采用递归方式进行排序
- 实现如下

```
//快速排序--由大到小
public static void quick_sort(int arr[], int left, int right) {
        // 该算法选取第一个数作为基准数，如果想用中间数为基准数，那么将第一个数和中间数交换一次即可
        if (left < right) {

            // Swap(s[l], s[(l + r) / 2]); //将中间的这个数和第一个数交换
            int i = left, j = right, x = arr[left];
            while (i < j) {
                // 从右向左找第一个大于x的数
                while (i < j && arr[j] < x) {//注意：如果进入循环，表示没有找到，继续找
                    j--;//如果没找到，那么j--，继续找，找到则跳出循环
                }
                if (i < j) {
                    //当从右向左找到后，i要加1
                    arr[i++] = arr[j];//核心是i++
                }

                //从左向右找第一个小于等于x的数
                while (i < j && arr[i] >= x) {//注意：如果进入循环，表示没有找到，继续找
                    i++;//如果没找到，i++，继续找，找到则跳出循环
                }

                if (i < j) {
                    //当从左向右找到后，j要减1
                    arr[j--] = arr[i];//核心是j--
                }

            }
            arr[i] = x;//将基准数填入空位置
            quick_sort(arr, left, i - 1); // 递归调用
            quick_sort(arr, i + 1, right);
        }
    }

    调用方法： MySortUtils.quick_sort(arrays, 0, arrays.length-1);
```

- 时间复杂度：O(N logN)，不稳定的排序算法
- 最坏情况为O(N*N)

6.归并排序

- 原理比较简单：采用分治法。
- 首先考虑下如何将二个有序数列合并。这个简单，只要从比较二个数列的第一个数，谁小就先取谁，取了后就在对应数列中删除这个数。然后再进行比较，如果有数列为空，那直接将另一个数列的数据依次取出即可。具体间函数mergearray
- 归并排序，其的基本思路就是将数组分成二组A，B，如果这二组组内的数据都是有序的，那么就可以很方便的将这二组数据进行排序。如何让这二组组内数据有序了？可以将A，B组各自再分成二组。依次类推，当分出来的小组只有一个数据时，可以认为这个小组组内已经达到了有序，然后再合并相邻的二个小组就可以了。这样通过先递归的分解数列，再合并数列就完成了归并排序。
- 详细分析见：http://blog.csdn.net/morewindows/article/details/6678165
- 实现

```
//归并排序--由大到小
public static void mergesort(int[] a, int first, int last, int[] temp) {
        if (first < last) {
            int mid = (first + last) / 2;
            mergesort(a, first, mid, temp); // 左边有序
            mergesort(a, mid + 1, last, temp); // 右边有序
            mergearray(a, first, mid, last, temp); // 再将二个有序数列合并
        }
```

```
    }



    //将二个有序数列a[first...mid]和a[mid...last]合并
    private static void mergearray(int[] a, int first, int mid, int last,
            int[] temp) {
        int i = first, j = mid + 1;
        int m = mid, n = last;
        int k = 0;

        while (i <= m && j <= n) {
            if (a[i] >= a[j]) //由大到小
                temp[k++] = a[i++];
            else
                temp[k++] = a[j++];
        }

        while (i <= m) //将剩下的A数组中的数填充到C中，如果有的话
            temp[k++] = a[i++];

        while (j <= n) //将剩下的B数组中的数填充到C中，如果有的话
            temp[k++] = a[j++];

        for (i = 0; i < k; i++)
            a[first + i] = temp[i];
    }


//外部调用方法
int[] tmp=new int[arrays.length];
MySortUtils.mergesort(arrays, 0, arrays.length-1, tmp);
```

- 时间复杂度：O(NlogN),是稳定排序算法，需要两倍于快速排序的空间
- 最坏时间复杂度和最好复杂度一样

7.基数排序

- 原理非常简单，基数一般都是取10，容易计算。
  - 根据数据项个位上的值，把所有数据项分为10组，第一组，个位全是0，第二组个位全是1，第十组个位全是9，并且相同组内顺序不变
  - 根据数据项十位上的值，把所有数据项分为10组，第一组，十位全是0，第二组十位全是1，第十组十位全是9，并且相同组内顺序保持上一个的顺序，因为这是稳定排序算法
  - 依次类推，当最高位的数据项都分好组后，那么最后一个分组就已经排好序了
- 举例说明
  - 原始数据：421 240 35 532 305 46 43 124
  - 第一步，为了好观察，全部补0：421 240 035 532 305 046 043 124
  - 第一轮：｛240｝，｛421｝，｛532｝，｛043｝，｛124｝，｛035，305｝，｛046｝
  - 第二轮：｛305｝，｛421，124｝，｛532，035｝，｛240，043，046｝
  - 第三轮：｛043，046｝，｛124｝，｛240｝，｛305｝，｛421｝，｛532｝
  - 此时可以看出，已经排好序了
- 一般实现，是采用10个链表，而不是10个数组，链表可以根据需要扩展和放缩。
- 算法非常简单，但是实现起来效果不佳
- 实现

```
/**
 * 基数排序算法思想： 先找出最大的数有几位数，这样就确定了要进行几次排序， 然后建立10个数组进行存放第n位数为0,1,2...9的数字，
 * 存放完之后要进行收集10个数组中的数字到原来的数组中， 然后在进行排序 第i位数为0,1,2...9的计算方法：array[j] %
 * Math.pow(10, i+1) / Math.pow(10,i)
 * 即第一位数直接除以10的余数就行了，第二位数，则除以100得余数，再除以10得商即为第二位数，以此类推。
 *
```

```
 * @param array 要进行基数排序的数组
 */
public static void radixSort(int[] arrays) {
    /**
     * 为了能够计算对正负数进行排序，需要将正数和负数分成两个数组
     * 负数数组进行全部取绝对值，然后进行基数排序，排好后，全部位置反向，符号变为符
     * 正数数组不用处理
     * 最后两个排行序的数组进行何必即可。感觉好麻烦呀
     */
    //计算有多少个负数
    int neg_count = 0;
    for (int i = 0; i < arrays.length; i++) {
        if (arrays[i] < 0) {
            neg_count++;
        }
    }
    //分成两个数组
    int[] neg_list = new int[neg_count];
    int j = 0, k = 0;
    int[] pos_list = new int[arrays.length - neg_count];
    for (int i = 0; i < arrays.length; i++) {
        if (arrays[i] < 0) {
            neg_list[j++] = Math.abs(arrays[i]);
        } else {
            pos_list[k++] = arrays[i];
        }
    }
    //对负数数组进行基数排序
    MySortUtils.radixSort_part(neg_list);
    //反转
    for (int startIndex = 0, endIndex = neg_list.length - 1; startIndex < endIndex; startIndex++, endIndex--) {
        int temp = neg_list[startIndex];
        neg_list[startIndex] = neg_list[endIndex];
        neg_list[endIndex] = temp;
    }
    //变为负数
    for (int i = 0; i < neg_list.length; i++) {
        neg_list[i]=0-neg_list[i];
    }
    //对正数数组进行基数排序
    MySortUtils.radixSort_part(pos_list);
    //两个数组合并
    System.arraycopy(neg_list, 0, arrays, 0, neg_count);
    System.arraycopy(pos_list, 0, arrays, neg_count, arrays.length
            - neg_count);
}

/**
 * 这个函数只能进行全部是正数的基数排序
 *
 * @param array
 */
private static void radixSort_part(int[] array) {
    int max, time;
    // 确定最大的数有几位数，确定排序的次数
    max = array[0];
    for (int i = array.length - 1; i > 0; i--) {
        if (max < array[i]) {
            max = array[i];
        }
    }
    time = 0;
    while (max > 0) {
        max /= 10;
        time++;
    }
    // 建立10个数组
```

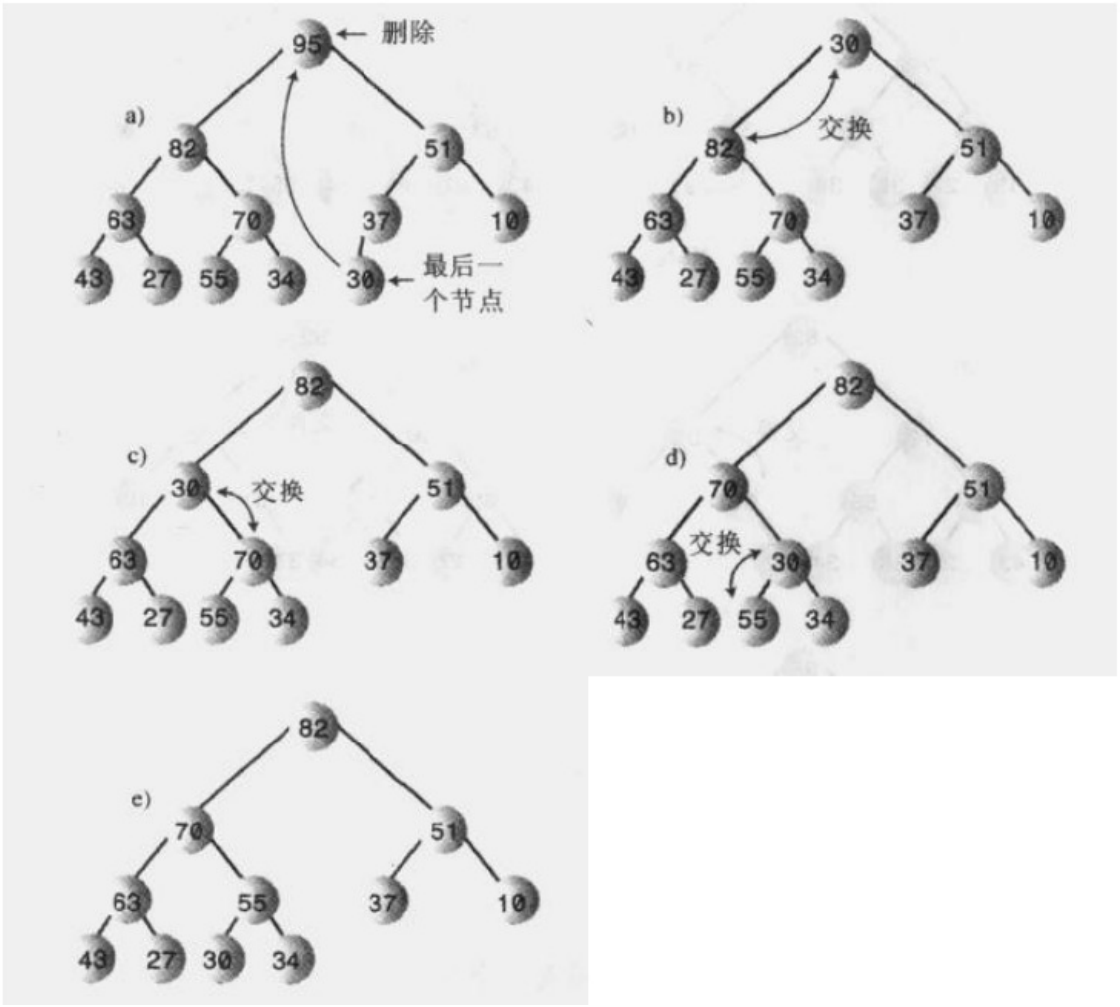
```
    List<ArrayList<Integer>> queue=new ArrayList<ArrayList<Integer>>(10);
    for (int i = 0; i < 10; i++) {
        ArrayList<Integer> queue1 = new ArrayList<Integer>();
        queue.add(queue1);
    }

    // 分别进行time次分配和收集数组，根据不同的位数进行分配到数组中，再收集到原来的数组中
    for (int i = 0; i < time; i++) {
        for (int j = 0; j < array.length; j++) {
            //Math.pow(10, i + 1)表示10的i+1次幂
            int index = (int) (array[j] % Math.pow(10, i + 1) / Math.pow(10, i));
             ArrayList<Integer> queue2=queue.get(index);
             queue2.add(array[j]);
             queue.set(index, queue2);
        }
        // 收集完了之后进行分配
        int count = 0;
        for (int k = 0; k < 10; k++) {
            while (queue.get(k).size() > 0) {
                ArrayList<Integer> queue3=queue.get(k);
                array[count] = queue3.remove(0);
                count++;
            }
        }
    }
}
```

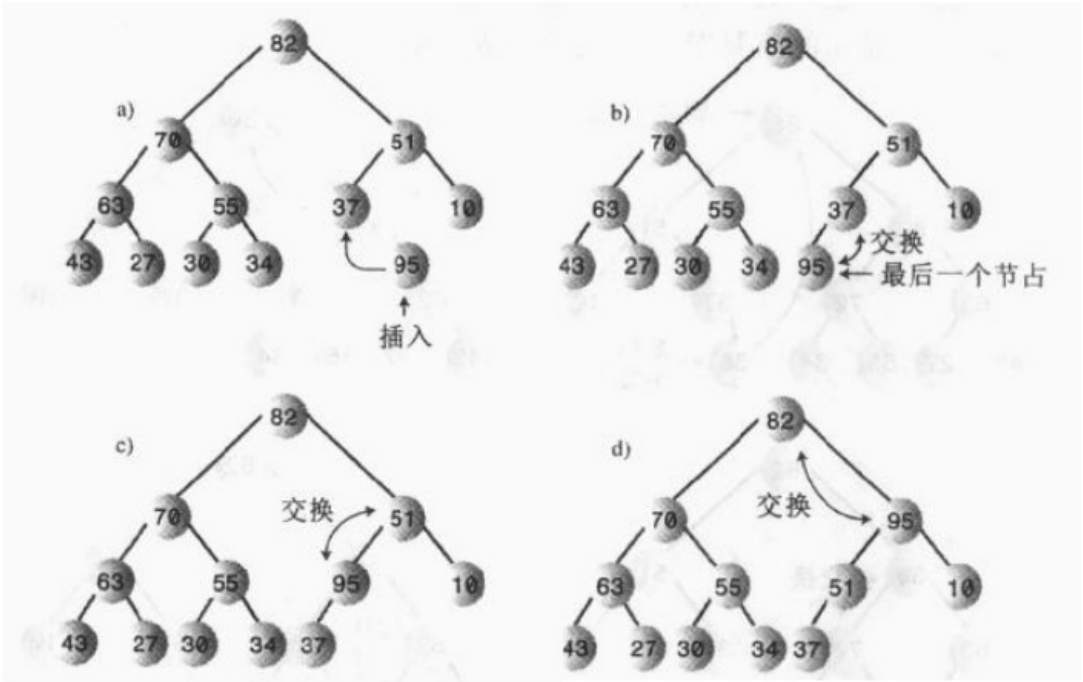
- 时间复杂度：O(NlogN)，稳定排序算法
- 适合并行计算

8.堆排序

- 堆：是完全二叉树即除了树的最后一层节点步满足满的，其他层从左到右都是满的；常常用一个数组实现；堆中的每一个节点都满足堆的条件即每一个节点的父节点的关键字都大于或等于这个节点的子节点的关键字(大顶堆)，还有小顶堆则相反。
- 注意堆和二叉搜索树的区别：堆是二叉搜索树的弱序，二叉搜索树中所有节点的左子孙的关键字都小于右子孙的关键字，而堆没有这个要求。
- 由于堆的这些性质，在堆中进行遍历节点，查找指定关键字都比较麻烦，一般步进行这些操作
- 堆的主要用途是进行**快速移除最大节点操作和快速插入新节点操作**。对于这两个操作，对于堆这种顺序结构是足够的，不需要实现二叉搜索树即可快速完成。常常用于堆排序操作。
- 堆的操作就是上面的两个，同时堆排序也就是利用上面的两个操作进行排序，效率较高。
- 对于一个已经是堆的数据，以最大堆为例，移除操作非常简单，但是移除顶部节点后，需要重新调整为堆结构，步骤如下：
  - 移除根
  - 把最后一个节点移动到根的位置
  - 一直向下筛选这个节点，直到它在一个大于它的节点子下，小于它的节点之上为止。
  - 特别注意：筛选交换的原则是，对于最大堆：交换原则是和下一层最大的孩子进行交换；对于最小堆：交换原则是和下一层最小的孩子进行交换
- 具体操作见图



- 对于一个已经是堆的数据，以最大堆为例，插入操作更简单，原理和移除差不多，但是是反向的，而且不需要比较左右孩子节点，只需要比较父节点即可
  - 将要插入的节点放置在堆的最后面
  - 采用向上筛选算法，直到它在一个大于它的节点子下，小于它的节点之上为止。
  - 注意：该节点只需要和父节点比较大小即可
- 具体操作见图



- 如数组中节点的索引是i，那么它的父节点的索引是(i-1)/2;它的左孩子节点索引是2*i*+1;它的右孩子节点的索引是2*i*+2;
- 堆排序的实现如下

```
/**
 * 堆排序--由大到小(建立最小堆)
 * 堆排序 --由小到大(建立最大堆)
 * 虽然感觉好像不太对劲，按照常理，由大到小，应该是建立最大堆，但是实际上是没错的
 * 主要原因是我们没有开辟新的空间，而是在原始空间基础上完成的，需要反向赋值。
 * @param heapArray
 */
public static void headSort(int[] heapArray) {
    int size = heapArray.length;
    int current=size;
    //初始化将数据调整为最小堆结构
    //注意：j=size/2-1和(size-1)/2本质上没有区别，有些人写成了后面的形式
    //但是实际上前者更好，size/2-1是从底部开始，第一个有子节点的节点，然后从后开始进行调整，构造最小堆
    for (int j = size / 2 - 1; j >= 0; j--) {
        //j表示调整j节点及其子孙节点为堆结构，其余节点不管
        trickleDown(j,heapArray,current);
     }

    //堆排序
    for (int j = size - 1; j >= 0; j--) //
    {
          //最小堆：root永远是最小的
          //最大堆：root永远是最大的
          int root = heapArray[0];
          heapArray[0] = heapArray[--current];
          //沉降法调整为堆结构，0表示调整整个树
          trickleDown(0,heapArray,current);
          //反向赋值
```

```
            heapArray[j] = root;
        }
    }


//目前里面实现的是：最小堆-由大到小排序
//如果需要由小到大排序--最大堆，那么改两个地方“>”改为“<”即可
private static void trickleDown(int index, int[] heapArray,int current) {
    int largerChild;
    int top = heapArray[index]; //保存目前节点-是父节点
    while (index < current / 2) //判断是否有孩子节点，如果没有，不需要调整
    {
        int leftChild = 2 * index + 1;//左孩子节点下标
        int rightChild = leftChild + 1;//右孩子节点下标

        //最小堆：找出左右孩子节点中，最小的那个
        //最大堆：找出左右孩子节点中，最大的那个
        if (rightChild < current &&
                heapArray[leftChild] > heapArray[rightChild])
            largerChild = rightChild;
        else
            largerChild = leftChild;

        //最小堆：如果父节点比孩子节点中最小的那个还小，那么不需要调整，否则孩子节点和父节点交互
        //最大堆：如果父节点比孩子节点中最大的那个还大，那么不需要调整，否则孩子节点和父节点交互
        if (top <= heapArray[largerChild])
            break;
        //孩子节点和父节点交换位置
        heapArray[index] = heapArray[largerChild];
        index = largerChild; //下一步运算需要
    }

    //整个堆调整完成，将当前节点填入空位置，完成一个节点的最大堆或者最小堆操作
    heapArray[index] = top;
}
```

- 时间复杂度：O(NlogN),但是依然没有快速排序快
- 优点：对初始数据的分布不敏感，最坏情况也可以维持在O(NlogN),但是快速排序在某些时候，复杂度可能降为O(N的平方)
- 插入节点时间复杂度为O(logn)
- 删除节点时间复杂度为O(logn)
- 查询最小元素的复杂度是O(1)
- 合并两个堆的复杂度是O(lgn)

总结

- 快速排序方法在(要排序的数据已基本有序)情况下最不利于发挥其长处
- 在排序方法中，元素比较次数与元素的初始排列无关的是选择排序，因为每次都要找出最大/最小值，必须全部比较一次，和初始排序无关
- 精简排序：插入排序和归并排序
- In-place sort（不占用额外内存或占用常数的内存）：插入排序、选择排序、冒泡排序、堆排序、快速排序。
- Out-place sort：归并排序、计数排序、基数排序、桶排序。

各种常用排序算法						
类别	排序方法	时间复杂度			空间复杂度	稳定性
		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		$O(nlog_2n)$	$O(nlog_2n)$	$O(nlog_2n)$	$O(1)$	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						

-

黄海安著