选择排序（SelectionSort）

实现思路：每轮循环通过比较找出数组中的最小值，然后和当轮循环的轮数进行交换（每轮循环找最小值，然后一个个往前放置）

```
public static void sort(Comparable[] arr) {
        for (int i = 0; i < arr.length; i++) {
            int minIndex = i;
            for (int j = i + 1; j < arr.length; j++) {
                if (arr[minIndex].compareTo(arr[j]) > 0) {
                    minIndex = j;
                }
            }
            SortTestHelper.swap(arr, i, minIndex);
        }
    }
```

插入排序（InsertionSort）

实现思路：每轮循环的第i个数和前一位进行比较如果比前面小，交换位置，直到前面的数比这一轮的比较数小，停止比较（数组最前端已经排好序，可以提前结束内层循环）

```
public static void sort(Comparable[] arr) {
        for (int i = 0; i < arr.length-1; i++) {
            for (int j = i + 1; j > 0&&arr[j-1].compareTo(arr[j]) > 0 ; j--) {
                 SortTestHelper.swap(arr, j, j-1);
            }
        }
    }
```

冒泡排序（BubbleSort）

实现思路：每轮 i 和 i+1 进行比较如果， 如果比旁边数大交换位置。

```
public static void sort(Comparable[] arr) {
        for (int i = 0; i < arr.length; i++) {
            for(int j = 1; j < arr.length - i; j++ ) {     // -i 是因为后面的数
```

```
都比当前比较的数字要大，所以直接不用比较提高性能
                if(arr[j-1].compareTo(arr[j]) > 0) {
                    SortTestHelper.swap(arr, j, j-1);
                }
            }
        }
    }
```

希尔排序（ShellSort || 最小增量排序）

实现思路：希尔排序是插入排序的升级版，是利用希尔排序的在几乎有序的数组排序速度极快的特性创建的算法，即通过一个增量，然后通过这个增量进行两个相隔的数字交换位置，通过调整增量让整的数组尽量有序，当增量为1 的时候，希尔排序则变成一个几乎有序的插入排序

```
public static void sort(Comparable[] arr) {

        int n = arr.length;

        int step = 0;

        //获取增进量
        while (step  < n/3 ) {
            step = 3 * step + 2;
        }

        while (step >= 1) {
//              让每个元素都要出来进行比较
            for (int i = step; i < n; i++) {
                Comparable e = arr[i];
                int j = i;
                for ( ; j >= step&&e.compareTo(arr[j-step]) < 0; j-=step) {
                    arr[j] = arr[j-step];
                }
                arr[j] = e;
            }
            step /= 2;
        }

    }
```

归并排序 （mergeSort）

实现思路，通过将两个有序数组进行归并成一个有序数组，在通过递归的方式深入，逐步让整个数组有序。

```
/**
     * 把两个数组进行merge
     * 1 3 5 7|2 4 6 8
     * 1 2 3 4 5 6 7 8
     * [l,r]
     * @param arr 要传入排序的数组
     * @param l   要进行归并的左界
     * @param m   归并排序的中间点,前一个数组的最后一个位置
     * @param r   要归并排序的右界
     * @return
     */
    private static void mergeArr(Comparable[] arr, int l, int m, int r) {

        Comparable[] resArr = new Comparable[r-l+1];
        for(int i = 0; i <= r-l; i++) {
            resArr[i] =  arr[l+i];
        }

        int i = 0;  //第一个数组比较元素
        int j = m-l+1; //第二个数组比较元素位置

        for (int k = l; k <= r; k++ ) {
            if(i+l > m) {
//                i值越界 数组2直接灌入
                arr[k] = resArr[j];
                j++;
            } else if (j+l > r) {
//                j值越界，数组1直接灌入
                arr[k] = resArr[i];
                i++;
            } else if(resArr[i].compareTo(resArr[j]) < 0) {
                arr[k] = resArr[i];
                i++;
            } else {
                arr[k] = resArr[j];
                j++;
            }
        }
    }
```

```
40
41      /**
42       * 归并排序算法
43       * 2 3 1 5 8 2 4| 6 4 8 1 0 3
44       * @param arr    要传入的乱序数组
45       * @param l      数组的左边界
46       *               [l r]
47       * @param r      length-1数组的右边界
48       */
49      public static void sort(Comparable[] arr, int l, int r) {
50          if(l  >= r) {
51  //            ShellSort.sort(arr);
52              return;
53          }
54          int m = (l+r)/2;
55          sort(arr, l, m);
56          sort(arr, m+1, r);
57          mergeArr(arr, l,m,r);
58      }
```

快速排序（quickSort)

随机选择一个中间数，将比中间数大的数字放右边，比中间数字小的数字放在左边，这个过程叫做partition， 然后将两边的数据通过递归再次进行partition，知道每个partition的长度为1为止

快排又分为一路快排，二路快排和三路快排，应对有大量冗余数据应该使用三路快排。

```
1   /**
2        * 对部分数组进行 partition 操作(核心操作)
3        * partition：将数组的首位元素拿出，然后比遍历数组中的每个元素，
4        * 如果比首位元素大，往前放
5        * 4 | 2 1 3 | 6 5 7 8
6        * 缺点：如果遇到有大量冗余数据，排序性能会急剧下降
7        *
8        * @param arr 排序数组
9        * @param l   左边界 l
10       * @param r   右边界 r
11       * @return 返回中间值所在位置
12       * [l, r]
13       */
14      private static int partition(Comparable[] arr, int l, int r) {
15
```

```
//        partition操作的中间元素 变为一个随机的位置的数字
        int m = random.nextInt(r - l + 1) + l;
        SortTestHelper.swap(arr, m, l);
        Comparable e = arr[l];
        int j = l;
        for (int i = l + 1; i <= r; i++) {
            if (arr[i].compareTo(e) < 0) {
                j++;
                SortTestHelper.swap(arr, j, i);
            }
        }
        SortTestHelper.swap(arr, j, l);
        return j;
    }

 private static void quickSort(Comparable[] arr, int l, int r) {

        if (l >= r) {
//          如果 r和l之间的元素少于15个 直接使用 O(n^2) 级别排序算法，经过测试在只是简单排数的场景下，
//          如果添加其他类型的排序，会降低排序性能
//            ShellSort.sort(arr);
            return;
        }
        int p = partition(arr, l, r);
        quickSort(arr, l, p - 1);
        quickSort(arr, p + 1, r);
```

堆排序

堆的本质就是一颗数，将这个数的索引关联到对树上就形成了一个堆，通过使用 shiftUp 和 shiftDown操作来实现一个最大二叉树，从而保证树顶元素一直都是最大值或者最小值。