数据结构与算法
作业报告

第一次

姓名	郝飞洋
班级	软件 2202 班
学号	2221411150
电话	13383536923
Email	958015679@qq. com
日期	2023-11-27

目录

任务 1 2
任务 2 2
题目 2
数据设计 2
算法设计 2
主干代码说明 3
运行结果展示 5
总结和收获 5
任务 3 5
题目 5
数据设计 5
算法设计 6
主干代码说明 8
运行结果展示（仅选一种算法测试结果展示） 15
总结和收获 15
任务 4 15
题目 15
数据设计 15
算法设计 16
主干代码说明 16
运行结果展示 19
规律阐述 20
任务 5 21
题目：数据分布对排序算法的影响 21
数据设计 21
算法设计 22
主干代码说明 22
运行结果展示 23
规律阐述 23
任务 6 23
题目：快速排序的再探讨和应用 23
附录：每个题的源代码 24
任务 2 24
任务 3 25
任务 4 29
任务 5 34
任务 6 34

任务 1

附于整个文档最后

任务 2

题目

已知一张图片是对某个事物横截面的扫描结果图，该图片的宽度是 m，高度是 n，图片的每一个像素只会由两种颜色之一构成：要么是蓝色，要么是白色。图片中的每一列的颜色分布有如下两种情形：

① 整列所有像素的颜色全是白色；

② 列中像素可以是白色或者蓝色，但在这种情况下，要么所有蓝色像素集中在从上到下，要么所有蓝色像素集中在从下到上，也就是说不会出现蓝色和白色相间的情形。

如果定义每一列的长度为蓝色像素的数量，那么如何求解图片中长度最大的列的长度呢？朴素的算法的时间复杂度是 O(mn)，但该任务要求完成的算法的时间复杂度必须满足 O(mlogn)。[2]

在随实验的附件中有一个 tomography.png 图片，同学们可以使用这张图片做为测试数据，图片的大小是 1200*1600，该图片中最大列长是 1575。对图片的处理可以继续使用在面向对象程序设计课程中构建的 Picture 类型。

数据设计

利用面向对象课程中构建的 Picture 类型对图片进行处理

将每一列都视作一个排好序的线性表，通过索引来确定其“长度”

设置一个记录最大值的变量来对所求值进行存储

算法设计

题目中要求算法的时间复杂度必须满足 O(mlogn)，所以自然想到用分治+递归。具体地说，就是用二分法来确定蓝白交界处的位置，并通过其索引来判断该列的长度。

具体算法如下：

长度计算算法：

判断首尾颜色是否相同，

若是，则长度为 0，算法结束

有两个指针分别指向首元素和尾元素

无条件的循环：

判断首元素和中间元素是否相等：

若是，首元素指针 <-- 中间元素指针

若否，尾元素指针 <-- 中间元素指针

判断两个指针是否指向相邻位置：

若是，

判断这列中首元素的颜色是否是蓝色：

若是，则长度=首元素指针+1，算法结束

若否，则长度=图片的高度-尾元素指针，算法结束

最大列长算法：

初始化 max=0，遍历所有的列，调用列长计算算法，将各个列的列长与 max 比较，若大于 max，则将其值赋给 max。

主干代码说明

```
1. import java.awt.*;
2. import java.io.IOException;
```

导入一些必要的包

```
3. public class MaxColumnLength {
4.     public static void main(String[] args) throws IOExcepti
   on {
5.         Picture testPicture = new Picture("F:\\1 学校课程
   \\3\\数据结构\\数据结构实验 1\\homework1(1)\\tomography.png");
```

设置 max 初始值为 0

```
6.          int max = 0;
```

遍历图片的宽度方向：

```
7.          for(int c = 0; c < testPicture.getWidth(); c++){
```

根据前述算法进行实现：

```
8.              int len = 0;
9.              if(testPicture.getColor(c,0).equals(testPicture
   .getColor(c, testPicture.getHeight()-1))){
10.                 continue;
11.             }
12.             int low = 0;
13.             int high = testPicture.getHeight()-1;
14.             while(true){
15.                 int mid = low + (high - low) / 2;
16.                 if(testPicture.getColor(c,low).equals(testP
   icture.getColor(c,mid))){
17.                     low = mid;
18.                 }else{
19.                     high = mid;
20.                 }
21.                 if(high - low <= 1){
22.                     if(testPicture.getColor(c,0).equals(Col
   or.white)){
23.                         len = testPicture.getHeight() - hig
   h;
24.                         break;
25.                     }else{
26.                         len = low + 1;
27.                         break;
28.                     }
29.                 }
30.             }
```

判断是否大于已知的最大值：

```
31.             if(len > max){
32.                 max = len;
33.             }
34.         }
```

输出结果：

```
35.         System.out.println(max);
36.     }
```

```
37.     public int getLength(Picture pic){
38.         return 0;
39.     }
40.
41.}
```

运行结果展示

```
1575
```

总结和收获

通过自己设计算法，体验到设计算法的技巧和其中需要的细致思考。

任务 3

题目

排序算法的实现

参照 Insertion 类的实现方式，为其他四个排序算法实现相对应的类类型。这些类类型中有可能需要相配合的成员方法，请同学们灵活处理。其中 Shell 排序中的间隔递减序列采用如下函数：

$$\begin{cases} h_1 = 1 \\ h_i = h_{i-1} * 3 \end{cases}$$

要求：

- 每个排序算法使用课堂上所讲授的步骤，不要对任何排序算法进行额外的优化；
- 对每个排序算法执行排序之后的数据可以调用 SortAlgorithm 类型中的成员方法 isSorted 进行测试，检查是否排序成功。

数据设计

使用提供的模板进行算法编写。

排序对象为 Comparable 类的数组，在排序过程中的辅助变量用合适的数据类型进行表示。

算法设计

选择排序:

外层循环（从 0 到 N）:

设 min 的初始值为循环变量 i

内层循环（从 i 到 N）:

如果当前扫描到的元素比 min 对应的元素小,

则将当前下标赋给 min

交换 i 对应的值和 min 对应的值

希尔排序:

第一层循环（用于计算增量，使用题目中给出的增量序列）:

第二层循环（选择每一组开始的元素）:

第三层循环（从每组的第二个元素开始遍历）:

第四层循环（比较该元素与组内前一个元素的大小，若小于前一个元素，则向前交换，不断向前直到不能交换为止）

快速排序:

需要用两个辅助变量 high 和 low 来标识目前正在处理的数组的首尾

若 high==low，排序结束，算法停止。

根据“三选一法”选择轴值

通过双指针法把数组分成两部分:

初始状态两个指针分别指向首尾

循环:

左指针：只要指向的值比轴值小就向右移

右指针：只要指向的值比轴值大且没有移动到数组左端就向左移

判断两指针位置是否交叉：

若否，将两指针指向的值互换

若是，将左指针所指的值与轴值交换

调用快速排序对 low 到左指针-1 的部分进行排序

调用快速排序对左指针+1 到 high 的部分进行排序

归并排序:

若数组元素个数不为 0 或 1:

将数组从中间一分为二

将两个数组分别进行归并排序

将两个数组合并

合并数组的具体方法：

将原数组 1 拷贝到一个临时数组中

在下面的过程中，指针指向的是临时数组中的元素，归并结果存放在原来的位置上。

指针 1、2 分别指向数组 1、2 的头部

比较两指针所指元素的大小，取较小值加入新数组中，并把这个指针右移

重复上一步，直到有一个指针已经到头：

将另一个指针所指的元素及之后的元素拷贝到新数组中

主干代码说明

快速排序：

代码说明已经以注释形式写在代码之中

这里使用了多种方法实现未优化的快速排序

```
1. public class Quick extends SortAlgorithm {
2.     public void sort(Comparable[] objs) {
3.         int N = objs.length;
4.         quickSort(objs, 0, N - 1);
5.     }
6.
7.     public static void main(String[] args){
8.         Double[] testData = GenerateData.getRandomData(5);
9.         Double[] test = new Double[6];
10.        for (int i = 0; i < 5; i++) {
11.            test[i] = testData[i];
12.            System.out.print(test[i] + " ");
13.        }
14.        System.out.println();
15.        SortAlgorithm alg = new Quick();
16.        alg.sort(testData);
17.        for (int i = 0; i < 5; i++) {
18.            System.out.print(testData[i] + " ");
19.        }
20.        System.out.println(test.equals(testData));
21.        System.out.println(alg.isSorted(testData));
22.    }
23.
24.    private Comparable choosePivot(Comparable[] objs, int l
   ow, int high){
25.        int mid = low + (high - low) / 2;
26.        Comparable pivot = objs[high];
27.        if (!(less(objs[low], objs[high]) ^ less(objs[mid],
    objs[low]))) {
28.            pivot = objs[low];
29.            exchange(objs, low, high);
30.        } else if (!(less(objs[mid], objs[low]) ^ less(objs
   [high], objs[mid]))) {
31.            pivot = objs[mid];
32.            exchange(objs, mid, high);
33.        }
```

```
34.         return pivot;
35.     }
36.
37.     /**
38.      * 老师上课讲的快速排序方法
39.      * @param objs
40.      * @param low
41.      * @param high
42.      */
43.     public void quickSort(Comparable[] objs, int low, int h
    igh) {
44.         //基准情形
45.         if (low >= high) {
46.             return;
47.         }
48.         //选择轴值
49.         Comparable pivot = choosePivot(objs, low, high);
50.         //相向双指针法的操作和递归
51.         int left = low;
52.         int right = high - 1;
53.         while (true) {
54.             while (less(objs[left], pivot)) {
55.                 left++;
56.             }
57.             while (less(pivot, objs[right]) && right > 0) {
58.                 right--;
59.             }
60.             if (left < right) {
61.                 exchange(objs, left, right);
62.             } else {
63.                 exchange(objs, left, high);
64.                 if (left > 1) {
65.                     quickSort(objs, low, left - 1);
66.                 }
67.                 quickSort(objs, left + 1, high);
68.                 return; //一定不要忘记这个return，否则会一直循
    环退不出去!
69.             }
70.         }
71.     }
72.
73.     /**
74.      * 将 partition 函数分离出来的快速排序算法
75.      * @param objs
76.      * @param low
```

```
77.      * @param high
78.      */
79.     public void quick(Comparable[] objs, int low, int high)
    {
80.         //基准情形
81.         if (low >= high) {
82.             return;
83.         }
84.         int pivotIndex = partition_2(objs, low, high);
85.         quick(objs, low, pivotIndex-1);
86.         quick(objs, pivotIndex+1, high);
87.     }
88.
89.     /**
90.      * 单路划分（两指针同向运动）
91.      * @param objs
92.      * @param low
93.      * @param high
94.      * @return
95.      */
96.     private int partition_1(Comparable[] objs, int low, int
     high){
97.         Comparable pivot = choosePivot(objs, low, high);
98.         int i = low; //i 相当于是一个“栈”的top 指针，指向目前已
    扫描到的最后一个比轴值小的元素的下一个位置。
99.         int j = low; //j 往右遍历搜索比轴值小的元素，找到后就放
    到i 左边的那个“栈”里去
100.          //刚开始时i 和j 在同一个位置上
101.          while(j < high){
102.              if(less(objs[j],pivot)){ //j 找到了比轴值小的元
    素
103.                  if(i != j){
104.                      exchange(objs,i,j);
105.                  }
106.                  //当一个比轴值小的元素放到左边的“栈”中时，i
    就会向后移动一位。
107.                  i++;
108.              }
109.              j++;
110.          }
111.          //此时i 指向的是比轴值小的元素的后一个元素（也就是第一
    个比轴值大的元素）
112.          exchange(objs,i,high);
113.          //将i 指向的值和轴值交换位置，完成划分。
114.          return i;
```

```
115.          //返回轴值最后所在的位置。
116.      }
117.
118.      /**
119.       * 普通双路划分（两指针相向运动）
120.       * @param objs
121.       * @param low
122.       * @param high
123.       * @return
124.       */
125.      public int partition_2(Comparable[] objs, int low, in
   t high){
126.          Comparable pivot = choosePivot(objs, low, high);
127.          int left = low;
128.          int right = high - 1;
129.          while(left < right){
130.              //其实只要让 left 指针和 right 指针碰面即可，就算
   交错了也必定会是交错到对方已经排过的领域的第一个值.
131.              //下面的两个循环的顺序是有讲究的，left 先动，最后
   指向的值是第一个大于轴值的值；right 先动，最后指向的是最后一个小
   于轴值的值。
132.              while(left < right && less(objs[left], pivot)
   ){ //这两个循环中的 left<right 都是有必要的!
133.                  left++;
134.              }
135.              while(left < right && less(pivot, objs[right]
   )){ //这两个循环中的比较都是严格小于，这样做可以使排序后等于
   pivot 的元素平均划分到两个区域中，为下一次划分奠定基础
136.                  right--;
137.              }
138.              exchange(objs,left,right);
139.          }
140.          exchange(objs,left,high);
141.          return left;
142.      }
143.  }
```

归并排序:

```
1. public class Merge extends SortAlgorithm {
2.     public void sort(Comparable[] objs) {
3.         int N = objs.length;
4.         mergeSort(objs,0,N-1);
5.     }
```

以下是 mergeSort 的主干代码：

```
6.      public void mergeSort(Comparable[] objs, int low, int h
   igh){
```

递归的基本情况：

```
7.          if(low == high){
8.              return;
9.          }
10.         int mid = low + (high - low) / 2;
```

递归调用：

```
11.         mergeSort(objs, low, mid);
12.         mergeSort(objs, mid + 1, high);
```

调用 merge 函数：

```
13.         merge(objs, low, mid + 1, high);
14.     }
```

这里将 merge 单独拿出来，方便 mergeSort 本身的递归调用。

```
15.     private void merge(Comparable[] objs, int start, int st
   art2, int end){
16.         int len1 = start2 - start;
17.         Comparable[] tmp = new Comparable[len1];
18.         for (int i = 0; i < len1; i++) {
19.             tmp[i] = objs[i];
20.         }
21.
22.         int p1 = 0;
23.         int p2 = start2;
24.         for(int i = start; i <= end; i++){
25.             if(less(tmp[p1],objs[p2])){
26.                 objs[i] = tmp[p1];
27.                 p1++;
28.                 if(p1 == len1){
29.                     break;
30.                 }
31.             }else{
32.                 objs[i] = objs[p2];
33.                 p2++;
34.                 if(p2>end){
35.                     while(p1<len1){
36.                         i++;
37.                         objs[i] = tmp[p1];
38.                         p1++;
```

```
                    }
                }
            }
        }
    }
```

在主函数中写辅助验证的代码:

```
    public static void main(String[] args){
        Double[] testData = GenerateData.getRandomData(5);
        Double[] test = new Double[6];
        for (int i = 0; i < 5; i++) {
            test[i] = testData[i];
        }
        SortAlgorithm alg = new Merge();
        alg.sort(testData);
        System.out.println(test.equals(testData));
        System.out.println(alg.isSorted(testData));
    }
}
```

选择排序:

```
public class Selection extends SortAlgorithm {
    public void sort(Comparable[] objs) {
        int N = objs.length;
        for (int i = 0; i < N; i++) {
            int min = i;
            for (int j = i; j < N; j++) {
                if(less(objs[j],objs[min])){
                    min = j;
                }
            }
            exchange(objs,i,min);
        }
    }
    public static void main(String[] args){
        Double[] testData = GenerateData.getRandomData(5);
        Double[] test = new Double[6];
        for (int i = 0; i < 5; i++) {
            test[i] = testData[i];
        }
        SortAlgorithm alg = new Insertion();
        alg.sort(testData);
        System.out.println(test.equals(testData));
```

```
        System.out.println(alg.isSorted(testData));
    }
}
```

希尔排序:

```
public class Shell extends SortAlgorithm {
    public void sort(Comparable[] objs){
        int N = objs.length;
        //计算每一轮的增量
        for(int i = N / 3; i > 0; i /= 3){
            //选择每一组的开始元素
            for( int j = 0; j < i; j ++){
                //开始插入排序，由于下面是和前一个元素比较，所以要从每组的第二个元素开始遍历
                for (int k = i + j; k < N; k += i) {
                    //将每一个元素放到该放的位置上去
                    for (int l = k; l >= i && less(objs[l], objs[l-i]); l -= i) {
                        exchange(objs,l,l-i);
                    }
                }
            }
        }
    }

    public static void main(String[] args){
        Double[] testData = GenerateData.getRandomData(5);
        Double[] test = new Double[6];
        for (int i = 0; i < 5; i++) {
            test[i] = testData[i];
        }
        SortAlgorithm alg = new Shell();
        alg.sort(testData);
        System.out.println(test.equals(testData));
        System.out.println(alg.isSorted(testData));
    }
}
```

运行结果展示（仅选一种算法测试结果展示）

```
true
```

总结和收获

通过对排序算法的实现，对各个排序算法有了更深的理解，并对算法实现过程中需要注意的地方更加熟悉了。

任务 4

题目

排序算法性能测试和比较

完成对每一个排序算法在数据规模为：2^8、2^9、2^{10}、……、2^{16} 的均匀分布的随机数据序列、正序序列和逆序序列的排序时间统计。

要求：

- 在同等规模的数据量和数据分布相同下，要做 T 次运行测试，用平均值做为此次测试的结果，用以排除因数据的不同和机器运行当前的状态等因素造成的干扰；（在 SortTest 类型的 test 方法参数中有对每次数据规模下的测试次数的指定）
- 将所有排序算法的运行时间结果用图表的方式进行展示，X 轴代表数据规模，Y 轴代表运行时间。（如果因为算法之间运行时间差异过大而造成显示上的问题，可以通过将运行时间使用取对数的方式调整比例尺）
- 对实验的结果进行总结：从一个算法的运行时间变化趋势和数据规模的变化角度，从同样的数据规模和相同数据分布下不同算法的时间相对差异上等角度进行阐述。

数据设计

使用提供的模板进行算法编写。

利用任务 3 中写好的排序算法进行完成。

通过对运行时间取对数的方法调整比例尺。

利用 java 提供的类来产生随机数据。

算法设计

通过理解给定作图模板并进行运用来画出图像。

主干代码说明

通过使用给定的画图模板来利用 jfreechart 画图

```
1. import org.jfree.chart.ChartFactory;
2. import org.jfree.chart.ChartPanel;
3. import org.jfree.chart.JFreeChart;
4. import org.jfree.chart.plot.PlotOrientation;
5. import org.jfree.chart.plot.XYPlot;
6. import org.jfree.chart.renderer.xy.XYLineAndShapeRenderer;
7. import org.jfree.chart.ui.ApplicationFrame;
8. import org.jfree.chart.ui.RectangleInsets;
9. import org.jfree.data.xy.XYDataset;
10.import org.jfree.data.xy.XYSeries;
11.import org.jfree.data.xy.XYSeriesCollection;
12.
13.import java.awt.*;
14.
15.public class LineXYDemo extends ApplicationFrame {
16.    // 该构造方法中完成了数据集、图表对象和显示图表面板的创建工作
17.    public LineXYDemo(String title){
18.        super(title);
19.        XYDataset dataset = createDataset();              //
    创建记录图中坐标点的数据集
20.        JFreeChart chart = createChart(dataset);          //
    使用上一步已经创建好的数据集生成一个图表对象
21.        ChartPanel chartPanel = new ChartPanel(chart);    //
    将上一步已经创建好的图表对象放置到一个可以显示的Panel 上
22.        // 设置 GUI 面板 Panel 的显示大小
23.        chartPanel.setPreferredSize(new Dimension(500, 270)
   );
24.        setContentPane(chartPanel);                       //
    这是 JavaGUI 的步骤之一，不用过于关心，面向对象课程综合训练的视频
    中进行了讲解。
25.    }
26.
```

```
27.     private JFreeChart createChart(XYDataset dataset) {
28.         // 使用已经创建好的dataset 生成图表对象
29.         // JFreechart 提供了多种类型的图表对象，本次实验是需要使用XYLine 型的图表对象
30.         JFreeChart chart = ChartFactory.createXYLineChart(
31.                 "Comparison of sorting algorithm performance(Random Data)",      // 图表的标题
32.                 "Data scale",                                   // 横轴的标题名
33.                 "log10(running time)",                          // 纵轴的标题名
34.                 dataset,                                        // 图表对象中使用的数据集对象
35.                 PlotOrientation.VERTICAL,                       // 图表显示的方向
36.                 true,                                           // 是否显示图例
37.                 false,                                          // 是否需要生成tooltips
38.                 false                                           // 是否需要生成urls
39.         );
40.         // 下面所做的工作都是可选操作，主要是为了调整图表显示的风格
41.         // 同学们不必在意下面的代码
42.         // 可以将下面的代码去掉对比一下显示的不同效果
43.         chart.setBackgroundPaint(Color.WHITE);
44.         XYPlot plot = (XYPlot)chart.getPlot();
45.         plot.setBackgroundPaint(Color.lightGray);
46.         plot.setAxisOffset(new RectangleInsets(5.0, 5.0, 5.0, 6.0));
47.         plot.setDomainGridlinePaint(Color.WHITE);
48.         plot.setRangeGridlinePaint(Color.WHITE);
49.         XYLineAndShapeRenderer renderer = (XYLineAndShapeRenderer) plot.getRenderer();
50.         renderer.setDefaultShapesVisible(true);
51.         renderer.setDefaultShapesFilled(true);
52.         return chart;
53.     }
54.
55.     private double[][] ArrayLog10(double[][] rawdata){
56.         int row = rawdata.length;
57.         int column = rawdata[0].length;
58.         double[][] result = new double[row][column];
59.         for (int i = 0; i < row; i++) {
```

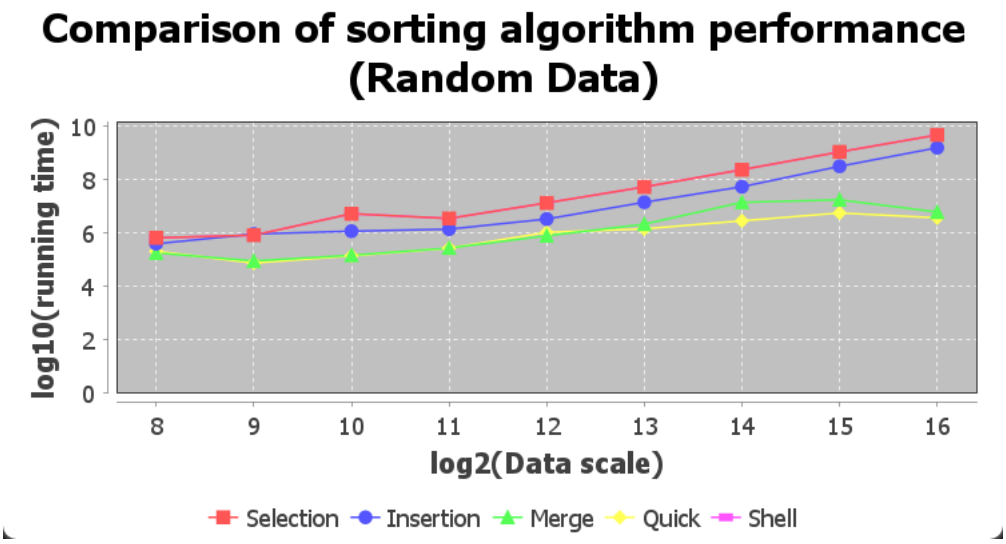
```
60.             for(int j = 0; j < column; j++){
61.                 result[i][j] = Math.log10(rawdata[i][j]);
62.             }
63.         }
64.         return result;
65.     }
66.     private XYDataset createDataset() {
67.         // 本样例中想要显示的是三组数据的变化图
68.         // X 数组是三组数据共同拥有的x 坐标值；Y1、Y2 和Y3 数组分别存储了三组数据对应的y 坐标值
```

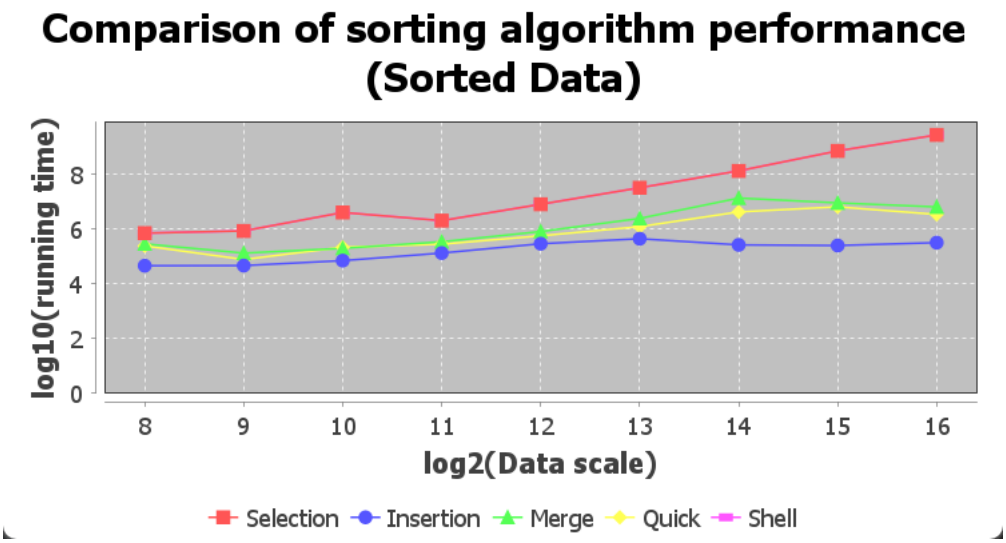
此处输入数据:

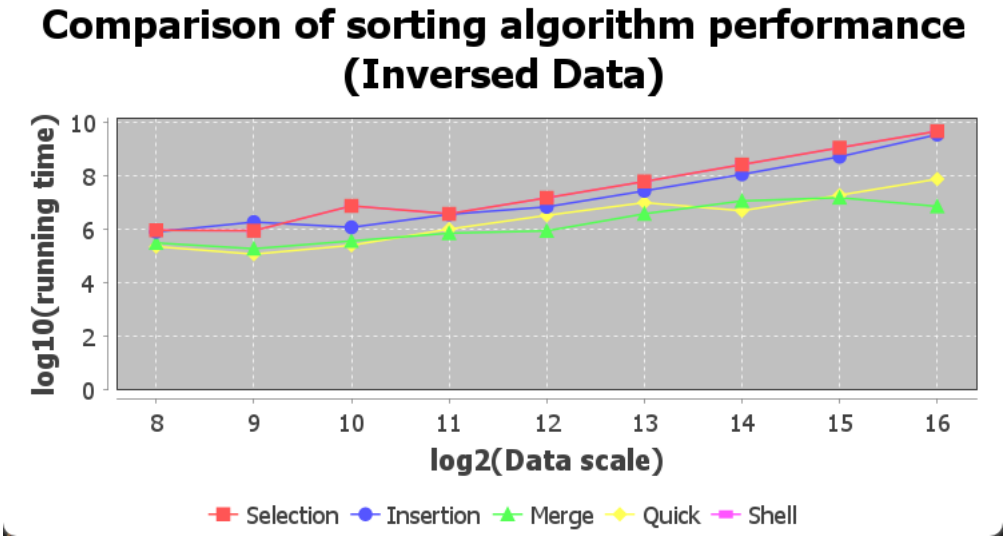
```
69.         double[] X = {1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0};
70.         double[] Y1 = {698440.0000,817360.0000,5733380.0000,5270680.0000,14846220.0000,59159340.0000,273607600.0000,1177341760.0000,4901339440.0000};
71.         double[] Y2 = {268400.0000,558500.0000,1187359.8000,2837539.6000,3360580.0000,12852500.2000,52058420.0000,321808119.8000,1413004639.8000};
72.         double[] Y3 = {210940.0000,86920.0000,178140.0000,293240.0000,596619.8000,1314800.2000,10481939.8000,12115399.8000,6555679.6000};
73.         double[] Y4 = {251300.0000,72000.0000,133200.0000,212540.0000,690520.0000,1040740.0000,2148280.0000,4855980.0000,10460420.0000};
74.         double[] Y5 = {163880.0000,132240.0000,298420.0000,184160.0000,694540.0000,1258120.0000,1564240.0000,2983220.0000,6451060.0000};
75.         double[][] Y_raw = {Y1, Y2, Y3, Y4, Y5};
76.         double[][] Y = ArrayLog10(Y_raw);
77.         // jfreechart 中使用XYSeries 对象存储一组数据的(x,y)的序列，因为有三组数据所以创建三个XYSeries 对象
78.         XYSeries[] series = {new XYSeries("Selection"), new XYSeries("Insertion"), new XYSeries("Merge"), new XYSeries("Quick"), new XYSeries("Shell")};
79.         int N = X.length;
80.         int M = series.length;
81.         for(int i = 0; i < M; i++)
82.             for(int j = 0; j < N; j++)
83.                 series[i].add(X[j], Y[i][j]);
84.         // 因为在该图表中显示的数据序列不止一组，所以在jfreechart 中需要将多组数据序列存放到一个XYSeriesCollection 对象中
```

```
85.         XYSeriesCollection dataset = new XYSeriesCollection
    ();
86.         for(int i = 0; i < M; i++)
87.             dataset.addSeries(series[i]);
88.
89.         return dataset;
90.     }
91.
92.     public static void main(String[] args) {
93.         LineXYDemo demo = new LineXYDemo("Comparison of sor
    ting algorithm performance");
94.         demo.pack();
95.         demo.setVisible(true);
96.     }
97. }
```

运行结果展示







规律阐述

选择排序:
时间复杂度为 O(n^2)，比较次数较多，数据交换次数较少。选择排序在处理数据时，只有在发现当前位置的数据比后面的位置上的数据大时，才会进行数据交换，因此交换次数相对较少。在数据量较大时，选择排序的时间效率较低。
插入排序:
时间复杂度为 O(n^2)，比较次数和数据交换次数均较多。插入排序在处理数据时，需要不断地将新数据插入到已排序好的数列中，并保持数列的有序性。因此，插入排序在处理数据量较大、数据分布无规律或数据本身无序的情况下，效率较低。
快速排序:
时间复杂度为 O(nlogn)，比较次数和数据交换次数均较多。快速排序是一种分治算法，通过选择一个基准元素将待排序的数列分成两部分，然后对这两部分分别进行排序。在处理数据时，快速排序能够利用分治的思想，将一个较大的数列分成若干个小数列，然后对每个小数

列进行递归处理。因此，快速排序在处理数据量较大、数据分布无规律或数据本身无序的情况下，效率较高。

归并排序:

时间复杂度为 O(nlogn)，比较次数和数据交换次数均较多。归并排序是一种基于分治思想的排序算法，通过将待排序的数列分成若干个子序列，然后对每个子序列进行递归处理，最终合并得到有序的序列。归并排序在处理数据时，需要将数据进行分解和合并操作，因此需要额外的空间存储临时数据。但是，归并排序具有较好的稳定性，即相同值的元素在排序后保持原有的相对顺序。

希尔排序:

希尔排序的运行时间随着数据量的增加而增加，但相对于冒泡排序和插入排序等算法，其运行时间的变化趋势较为平缓。这是因为希尔排序在每次迭代时都会将数据分成更多的子序列，从而在每个子序列上进行更精细的排序操作。因此，当数据量较大时，希尔排序的效率相对较高。希尔排序在处理不同规模的数据时，其性能表现也不同。对于较小的数据集，希尔排序的效率可能不如其他一些算法（如插入排序），因为希尔排序的额外操作（即划分和合并子序列）会使得运行时间较长。然而，随着数据规模的增加，希尔排序的优势逐渐显现，因为它能够更好地利用缓存和减少内存访问的开销。在相同的数据规模和相同的数据分布下，希尔排序相对于其他一些排序算法（如冒泡排序、插入排序）具有更快的运行速度。这是因为希尔排序在处理数据时，能够更快地缩小数据的范围，从而减少需要比较和交换的元素数量。

任务 5

题目：数据分布对排序算法的影响

完成了任务 3 和任务 4 之后，现要求为 GenerateData 类型再增加一种数据序列的生成方法，该方法要求生成分布不均匀数据序列：1/2 的数据是 0，1/4 的数据是 1，1/8 的数据是 2 和 1/8 的数据是 3。对这种分布不均匀的数据完成如同任务 4 的运行测试，检查这样的数据序列对于排序算法的性能影响。要求同任务 4。（此时，可以将任务 4、任务 5 的运行测试结果做一个纵向比较，用以理解数据序列分布的不同对同一算法性能的影响，如果能从每个排序算法的过程去深入分析理解则更好。）

数据设计

继承上一个题的设计思想。

算法设计

通过计算数组长度*比例来计算赋值的个数，赋值之后再用 shuffle 函数来将数组打乱。

测试过程与上一个任务大致相同，不再赘述。

主干代码说明

```
1. public static Double[] getNovelData(int N){
```

创建数组：

```
2.          Double[] numbers = new Double[N];
3.          for(int i = 0; i < N/2; i++){
4.              numbers[i] = 0.0;
5.          }
```

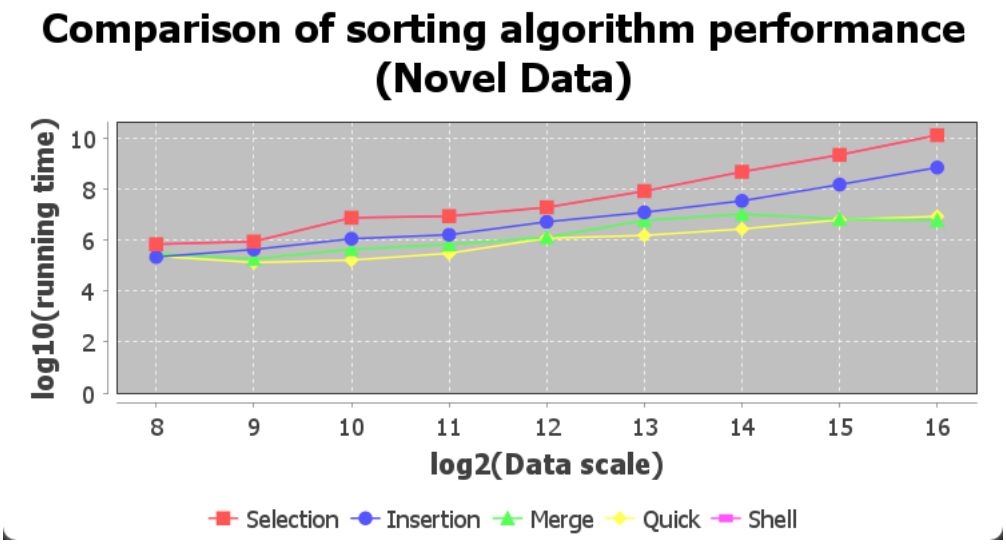
进行赋值：

```
6.          for(int i = N/2; i < 3 * N / 4; i++){
7.              numbers[i] = 1.0;
8.          }
9.          for(int i = 3 * N / 4; i < 7 * N / 8; i++){
10.             numbers[i] = 2.0;
11.         }
12.         for(int i = 7 * N / 8; i < N; i++){
13.             numbers[i] = 3.0;
14.         }
15.         shuffle(numbers, 0, numbers.length - 1);
```

打乱数组：

```
16.         return numbers;
17.     }
```

运行结果展示



规律阐述

在不同数据分布下，每个排序算法的表现都会有所变化。分布不均匀数据可能对插入排序、选择排序等有较大影响，而对于快速排序、归并排序等影响相对较小。每个算法对数据分布的敏感性不同，这种比较有助于选择适当的算法来处理特定类型的数据。

任务 6

题目：快速排序的再探讨和应用

附录：每个题的源代码

任务 2

```
1. import java.awt.*;
2. import java.io.IOException;
3.
4. public class MaxColumnLength {
5.     public static void main(String[] args) throws IOExcepti
   on {
6.         Picture testPicture = new Picture("F:\\1 学校课程
   \\3\\数据结构\\数据结构实验 1\\homework1(1)\\tomography.png");
7.         int max = 0;
8.         for(int c = 0; c < testPicture.getWidth(); c++){
9.             int len = 0;
10.            if(testPicture.getColor(c,0).equals(testPicture
   .getColor(c, testPicture.getHeight()-1))){
11.                continue;
12.            }
13.            int low = 0;
14.            int high = testPicture.getHeight()-1;
15.            while(true){
16.                int mid = low + (high - low) / 2;
17.                if(testPicture.getColor(c,low).equals(testP
   icture.getColor(c,mid))){
18.                    low = mid;
19.                }else{
20.                    high = mid;
21.                }
22.                if(high - low <= 1){
23.                    if(testPicture.getColor(c,0).equals(Col
   or.white)){
24.                        len = testPicture.getHeight() - hig
   h;
25.                        break;
26.                    }else{
27.                        len = low + 1;
28.                        break;
29.                    }
30.                }
31.            }
32.            if(len > max){
```

```
                max = len;
            }
        }
        System.out.println(max);
        System.out.println(testPicture.getHeight());
    }
    public int getLength(Picture pic){
        return 0;
    }

}
```

任务 3

```
/**
     * 对 objs[left..right]进行归并排序（公用 temp 数组版本）
     *
     * @param objs
     * @param left
     * @param right
     * @param temp
     */
    public void mergeSort(Comparable[] objs, int left, int right, Comparable[] temp){
        if(right - left < 16){ //取 THRESHOLD=16
            Insertion insertionSort = new Insertion();
            insertionSort.insertionSort(objs,left,right);
            return;
        }
        int mid = (left + right) / 2;
        mergeSort(objs, left, mid, temp);
        mergeSort(objs, mid + 1, right, temp);

        if (!less(objs[mid+1],objs[mid])){
            return;
        }
        //合并两个有序区间
        for (int i = left; i <= right; i++) {
            temp[i] = objs[i];
        }
        //这里给出了一种在递归函数中让某个内容“不递归”的方法：作为函数的参数传进去。
        //比如这里的 temp 数组，是一个全局使用的数组，我们就可以不在函数中创建它，而是在调用处创建之后作为参数传入。
```

```
28.         //数组全局使用有两个好处：
29.         //1. 避免了将元素复制时产生的下标偏移问题，在调用时，取哪块用哪块
30.         //2. 如果每一次归并都创建一个新的数组，此时创建和回收数组的性能开销会比较大
31.         int i = left;
32.         int j = mid + 1;
33.         for (int k = left; k <= right; k++) {
34.             if(i == mid+1){
35.                 objs[k] = temp[j];
36.                 j++;
37.             } else if (j == right + 1) {
38.                 objs[k] = temp[i];
39.             } else if (less(temp[j],temp[i])){//此处应考虑归并排序算法的稳定性，要使得两元素相等时 i 优先放入。
40.                 objs[k] = temp[j];
41.                 j++;
42.             }else{
43.                 objs[k] = temp[i];
44.                 i++;
45.             }
46.         }
47.     }
48.
49.     //归并排序的优化 1：在小区间内使用插入排序
50.     //归并排序的优化 2：在归并两个有序数组之前检查是否满足“左最大>=右最小”
51.
52.     /*关于分治算法：
53.         分解、解决、合并
54.         应用：归并排序、快速排序、树、回溯算法、动态规划（记忆化递归）
55.      */
```

```
1. /**
2.       * 将 partition 函数分离出来的快速排序算法
3.       * @param objs
4.       * @param low
5.       * @param high
6.       */
7.     public void quick(Comparable[] objs, int low, int high) {
8.         //基准情形
```

```
9.          if (low >= high) {
10.             return;
11.         }
12.         int pivotIndex = partition(objs, low, high);
13.         quick(objs, low, pivotIndex-1);
14.         quick(objs, pivotIndex+1, high);
15.     }
16.
17.     /**
18.      * 单路划分（两指针同向运动）
19.      * @param objs
20.      * @param low
21.      * @param high
22.      * @return
23.      */
24.     private int partition_1(Comparable[] objs, int low, int
    high){
25.         Comparable pivot = choosePivot(objs, low, high);
26.         int i = low; //i 相当于是一个“栈”的top 指针，指向目前已
    扫描到的最后一个比轴值小的元素的下一个位置。
27.         int j = low; //j 往右遍历搜索比轴值小的元素，找到后就放
    到i 左边的那个“栈”里去
28.         //刚开始时i 和j 在同一个位置上
29.         while(j < high){
30.             if(less(objs[j],pivot)){ //j 找到了比轴值小的元素
31.                 if(i != j){
32.                     exchange(objs,i,j);
33.                 }
34.                 //当一个比轴值小的元素放到左边的“栈”中时，i 就会
    向后移动一位。
35.                 i++;
36.             }
37.             j++;
38.         }
39.         //此时i 指向的是比轴值小的元素的后一个元素（也就是第一个
    比轴值大的元素）
40.         exchange(objs,i,high);
41.         //将i 指向的值和轴值交换位置，完成划分。
42.         return i;
43.         //返回轴值最后所在的位置。
44.     }
45.
46.     /**
47.      * 普通双路划分（两指针相向运动）
48.      * @param objs
```

```
49.      * @param low
50.      * @param high
51.      * @return
52.      */
53.     public int partition_2(Comparable[] objs, int low, int high){
54.         Comparable pivot = choosePivot(objs, low, high);
55.         int left = low;
56.         int right = high - 1;
57.         while(left < right){
58.             //其实只要让 left 指针和 right 指针碰面即可，就算交错了也必定会是交错到对方已经排过的领域的第一个值.
59.             //下面的两个循环的顺序是有讲究的，left 先动，最后指向的值是第一个大于轴值的值；right 先动，最后指向的是最后一个小于轴值的值。
60.             while(left < right && less(objs[left], pivot)){ //这两个循环中的 left<right 都是有必要的!
61.                 left++;
62.             }
63.             while(left < right && less(pivot, objs[right])){ //这两个循环中的比较都是严格小于，这样做可以使排序后等于 pivot 的元素平均划分到两个区域中，为下一次划分奠定基础
64.                 right--;
65.             }
66.             exchange(objs,left,right);
67.         }
68.         exchange(objs,left,high);
69.         return left;
70.     }
71.     private int partition(Comparable[] objs, int left, int right){
72.         Comparable pivot = choosePivot(objs,left,right);
73.         //循环不变量:
74.         //[left..i) < pivot
75.         //(j..right-1] > pivot
76.         int i = left;
77.         int j = right-1;
78.         while(i <= j){
79.             while(i<=j && less(objs[i],pivot)){ // i 找大于等于轴值的值，因此在小于轴值时要向右移动
80.                 i++;
81.             }
82.             while(i<=j && less(pivot,objs[j])){ // j 找小于等于轴值的值，因此在大于轴值时要向左移动
83.                 j--;
```

```
            }
            if(i<=j){
                exchange(objs,i,j);
                i++;
                j--;
            }
        }
        exchange(objs,i,right);
        return i;
    }
```

其他请参见主干代码分析部分

任务 4

```
import org.jfree.chart.ChartFactory;
import org.jfree.chart.ChartPanel;
import org.jfree.chart.JFreeChart;
import org.jfree.chart.plot.PlotOrientation;
import org.jfree.chart.plot.XYPlot;
import org.jfree.chart.renderer.xy.XYLineAndShapeRenderer;
import org.jfree.chart.ui.ApplicationFrame;
import org.jfree.chart.ui.RectangleInsets;
import org.jfree.data.xy.XYDataset;
import org.jfree.data.xy.XYSeries;
import org.jfree.data.xy.XYSeriesCollection;

import java.awt.*;

public class LineXYDemo extends ApplicationFrame {
    // 该构造方法中完成了数据集、图表对象和显示图表面板的创建工作
    public LineXYDemo(String title){
        super(title);
        XYDataset dataset = createDataset();              //
    创建记录图中坐标点的数据集
        JFreeChart chart = createChart(dataset);          //
    使用上一步已经创建好的数据集生成一个图表对象
        ChartPanel chartPanel = new ChartPanel(chart);    //
    将上一步已经创建好的图表对象放置到一个可以显示的Panel 上
        // 设置GUI 面板Panel 的显示大小
        chartPanel.setPreferredSize(new Dimension(500, 270)
    );
```

```
24.         setContentPane(chartPanel);                    // 这是JavaGUI的步骤之一，不用过于关心，面向对象课程综合训练的视频中进行了讲解。
25.     }
26.
27.     private JFreeChart createChart(XYDataset dataset) {
28.         // 使用已经创建好的dataset生成图表对象
29.         // JFreechart提供了多种类型的图表对象，本次实验是需要使用XYLine型的图表对象
30.         JFreeChart chart = ChartFactory.createXYLineChart(
31.                 "Comparison of sorting algorithm performance(Repetitive Data)",      // 图表的标题
32.                 "Data Repetitiveness",          // 横轴的标题名
33.                 "running time",                 // 纵轴的标题名
34.                 dataset,                        // 图表对象中使用的数据集对象
35.                 PlotOrientation.VERTICAL,       // 图表显示的方向
36.                 true,                           // 是否显示图例
37.                 false,                          // 是否需要生成tooltips
38.                 false                           // 是否需要生成urls
39.         );
40.         // 下面所做的工作都是可选操作，主要是为了调整图表显示的风格
41.         // 同学们不必在意下面的代码
42.         // 可以将下面的代码去掉对比一下显示的不同效果
43.         chart.setBackgroundPaint(Color.WHITE);
44.         XYPlot plot = (XYPlot)chart.getPlot();
45.         plot.setBackgroundPaint(Color.lightGray);
46.         plot.setAxisOffset(new RectangleInsets(5.0, 5.0, 5.0, 6.0));
47.         plot.setDomainGridlinePaint(Color.WHITE);
48.         plot.setRangeGridlinePaint(Color.WHITE);
49.         XYLineAndShapeRenderer renderer = (XYLineAndShapeRenderer) plot.getRenderer();
50.         renderer.setDefaultShapesVisible(true);
51.         renderer.setDefaultShapesFilled(true);
52.         return chart;
53.     }
54.
```

```
55.     private double[][] ArrayLog10(double[][] rawdata){
56.         int row = rawdata.length;
57.         int column = rawdata[0].length;
58.         double[][] result = new double[row][column];
59.         for (int i = 0; i < row; i++) {
60.             for(int j = 0; j < column; j++){
61.                 result[i][j] = Math.log10(rawdata[i][j]);
62.             }
63.         }
64.         return result;
65.     }
66.     private XYDataset createDataset() {
67.         // 本样例中想要显示的是三组数据的变化图
68.         // X 数组是三组数据共同拥有的x 坐标值；Y1、Y2 和Y3 数组
    分别存储了三组数据对应的y 坐标值
69.         double[] X = {50, 60, 80, 100};
70.
71.         double[] Ysr = {646480.0000,810360.0000,5216780.000
    0,3451820.0000,13326260.0000,53072580.0000,236708440.0000,1
    101155780.0000,4808753200.0000};
72.         double[] Yss = {673840.0000,815200.0000,3853560.000
    0,1939400.0000,7704600.0000,31027140.0000,128925740.0000,69
    3407720.0000,2670950720.0000};
73.         double[] Ysi = {880000.0000,844980.0000,7165340.000
    0,3632840.0000,14365020.0000,58608360.0000,252736780.0000,1
    084954120.0000,4502642480.0000};
74.         double[] Ysn = {709960.0000,902620.0000,7810680.000
    0,8972540.0000,19912840.0000,86534640.0000,498192580.0000,2
    317368140.0000,13764761740.0000};
75.
76.         double[] Yir = {386280.0000,898000.0000,1149020.000
    0,1370900.0000,3291480.0000,14115640.0000,53964540.0000,316
    715600.0000,1586506900.0000};
77.         double[] Yis = {43180.0000,43720.0000,66320.0000,12
    5640.0000,276080.0000,421140.0000,250280.0000,237340.0000,3
    02640.0000};
78.         double[] Yii = {764640.0000,1783020.0000,1127500.00
    00,3463940.0000,6567880.0000,26136200.0000,108678180.0000,4
    95460960.0000,3352051980.0000};
79.         double[] Yin = {222700.0000,438700.0000,1159760.000
    0,1667620.0000,5377860.0000,12731920.0000,35900020.0000,156
    989740.0000,733090540.0000};
80.
```

```
81.         double[] Yshr = {175020.0000,148880.0000,361760.000
    0,322840.0000,784160.0000,982520.0000,2201940.0000,2746840.
    0000,9321320.0000};
82.         double[] Yshs = {170160.0000,183120.0000,370640.000
    0,405920.0000,191160.0000,442780.0000,320280.0000,644640.00
    00,1467720.0000};
83.         double[] Yshi = {141660.0000,134840.0000,268200.000
    0,355320.0000,886880.0000,917120.0000,1243420.0000,1090480.
    0000,2299960.0000};
84.         double[] Yshn = {173480.0000,119220.0000,227160.000
    0,488580.0000,632280.0000,1114120.0000,971400.0000,1399820.
    0000,3818280.0000};
85.
86.         double[] Ymr = {174420.0000,87320.0000,142320.0000,
    266360.0000,769800.0000,2113820.0000,13971980.0000,17393940
    .0000,6029860.0000};
87.         double[] Yms = {272340.0000,125000.0000,186920.0000
    ,325680.0000,770900.0000,2332820.0000,12997100.0000,8645920
    .0000,6153220.0000};
88.         double[] Ymi = {293540.0000,179160.0000,348220.0000
    ,682320.0000,833840.0000,3623400.0000,11007900.0000,1459210
    0.0000,6858920.0000};
89.         double[] Ymn = {299440.0000,188580.0000,446000.0000
    ,706540.0000,1305500.0000,6132160.0000,10919980.0000,687910
    0.0000,6172660.0000};
90.
91.         double[] Yqr = {201260.0000,72520.0000,135620.0000,
    263440.0000,1036060.0000,1405140.0000,2822540.0000,5558600.
    0000,3651920.0000};
92.         double[] Yqs = {229600.0000,73060.0000,205320.0000,
    265880.0000,538740.0000,1166500.0000,4042660.0000,6187060.0
    000,3265560.0000};
93.         double[] Yqi = {214460.0000,111620.0000,240120.0000
    ,968260.0000,3171600.0000,9499700.0000,4730680.0000,1809410
    0.0000,72704400.0000};
94.         double[] Yqn = {241980.0000,134880.0000,168280.0000
    ,311020.0000,1216200.0000,1576560.0000,2813200.0000,6417160
    .0000,8831980.0000};
95.
96.         double[] Yqur = {338080.0000,100760.0000,107120.000
    0,182340.0000,708040.0000,1132960.0000,1726960.0000,8226760
    .0000,4730540.0000};
97.         double[] Yqus = {205060.0000,146660.0000,81340.0000
    ,132840.0000,413460.0000,597480.0000,1273940.0000,5646500.0
    000,5287160.0000};
```

```
98.         double[] Yqui = {183880.0000,147940.0000,250580.000
    0,985520.0000,3103540.0000,9707160.0000,4694080.0000,183595
    60.0000,72049020.0000};
99.         double[] Yqun = {225360.0000,144640.0000,199820.000
    0,249480.0000,1009000.0000,1080220.0000,2061620.0000,105932
    60.0000,5792220.0000};
100.
101.          double[] Yqu = {12475340.0000, 11279740.0000, 115
    30620.0000, 8851100.0000};
102.          double[] Yqt = {960.0000, 1000.0000, 1060.0000, 1
    040.0000};
103.
104.  //          double[][] Y_raw = {Yqr, Yqur};
105.  //          double[][] Y = ArrayLog10(Y_raw);
106.          double[][] Y = {Yqu, Yqt};
107.          // jfreechart 中使用XYSeries 对象存储一组数据的(x,y)
    的序列，因为有三组数据所以创建三个XYSeries 对象
108.          XYSeries[] series = {new XYSeries("quick_update")
    , new XYSeries("quick_triple")};
109.  //          XYSeries[] series = {new XYSeries("Selection"),
     new XYSeries("Insertion"), new XYSeries("Merge"), new XYSe
    ries("Quick"), new XYSeries("Shell")};
110.          int N = X.length;
111.          int M = series.length;
112.          for(int i = 0; i < M; i++)
113.              for(int j = 0; j < N; j++)
114.                  series[i].add(X[j], Y[i][j]);
115.          // 因为在该图表中显示的数据序列不止一组，所以在
    jfreechart 中需要将多组数据序列存放到一个XYSeriesCollection 对
    象中
116.          XYSeriesCollection dataset = new XYSeriesCollecti
    on();
117.          for(int i = 0; i < M; i++)
118.              dataset.addSeries(series[i]);
119.
120.          return dataset;
121.      }
122.
123.      public static void main(String[] args) {
124.          LineXYDemo demo = new LineXYDemo("Comparison of s
    orting algorithm performance");
125.          demo.pack();
126.          demo.setVisible(true);
127.      }
128.  }
```

任务 5

```
1. public static Double[] getNovelData(int N){
2.         Double[] numbers = new Double[N];
3.         for(int i = 0; i < N/2; i++){
4.             numbers[i] = 0.0;
5.         }
6.         for(int i = N/2; i < 3 * N / 4; i++){
7.             numbers[i] = 1.0;
8.         }
9.         for(int i = 3 * N / 4; i < 7 * N / 8; i++){
10.            numbers[i] = 2.0;
11.        }
12.        for(int i = 7 * N / 8; i < N; i++){
13.            numbers[i] = 3.0;
14.        }
15.        shuffle(numbers, 0, numbers.length - 1);
16.        return numbers;
17.    }
```

任务 6

```
1.     private void quickSortTriple_rightPivot(Comparable[] ob
   js, int left, int right){
2.         if(left <= right){
3.             return;
4.         }
5.
6.         Comparable pivot = choosePivot(objs,left,right);
7.         //循环不变量
8.         //[left..lt) < pivot
9.         //[lt..i) == pivot
10.        //(gt..right-1] >pivot
11.        int lt = left;
12.        int gt = right - 1;
13.        int i = left;
14.
15.        while(i <= gt){
16.            if(less(objs[i],pivot)){
17.                exchange(objs, lt, i);
18.                lt++;
```

```
                i++;
            } else if (less(pivot, objs[i])) {
                exchange(objs, gt, i);
                gt--;
            } else {
                i++;
            }
        }
        //轴值和第一个比轴值大的元素交换
        exchange(objs,right, gt+1);
        quickSortTriple_rightPivot(objs,left,lt-1);
        quickSortTriple_rightPivot(objs,gt+2,right);
    }
```

```
 public Comparable kthsmallest_double(Comparable[] objs, int k){
        int len = objs.length;
        int target = k - 1;
        int left = 0;
        int right = len - 1;
        while(true){
            int pivotIndex = partition(objs, left, right);
            if(pivotIndex == target){
                return objs[pivotIndex];
            } else if (pivotIndex < target) {
                left = pivotIndex + 1;
            }else {
                right = pivotIndex - 1;
            }
        }
    }

 private int partition(Comparable[] objs, int left, int right){
        Comparable pivot = choosePivot(objs,left,right);
        //循环不变量:
        //[left..i) < pivot
        //(j..right-1] > pivot
        int i = left;
        int j = right-1;
        while(i <= j){
            while(i<=j && less(objs[i],pivot)){
            // i 找大于等于轴值的值，因此在小于轴值时要向右移动
                i++;
            }
```

```
            while(i<=j && less(pivot,objs[j])){
            // j 找小于等于轴值的值，因此在大于轴值时要向左移动
                j--;
            }
            if(i<=j){
                exchange(objs,i,j);
                i++;
                j--;
            }
        }
        exchange(objs,i,right);
        return i;
    }
```

```
    public Comparable kthsmallest_triple(Comparable[] objs, int k, int left, int right){
        // base case
        if ( right - left <= 1 ){
            if ( left == right ){
                return objs[left];
            } else {
                if (less(objs[right],objs[left])){
                    exchange(objs, left, right);
                }
                if ( k==1 ){
                    return objs[left];
                } else {
                    return objs[right];
                }
            }
        }
        Comparable pivot = choosePivot(objs,left,right);
        //循环不变量
        //[left..lt) < pivot
        //[lt..i) == pivot
        //(gt..right-1] >pivot
        int lt = left;
        int gt = right - 1;
        int i = left;

        while(i <= gt){
            if(less(objs[i],pivot)){
                exchange(objs, lt, i);
                lt++;
```

```
30.                 i++;
31.             } else if (less(pivot, objs[i])) {
32.                 exchange(objs, gt, i);
33.                 gt--;
34.             } else {
35.                 i++;
36.             }
37.         }
38.         //轴值和第一个比轴值大的元素交换
39.         exchange(objs,right, gt+1);
40.         // 每轮排序完成后，[left, lt-
    1] < pivot, [gt+2, right] > pivot
41.
42.         if (k-1 <= lt-left-1){
43.             return kthsmallest_triple(objs,k,left,lt-1);
44.         } else if (k-1 > lt-left-1 && k-1 < gt-left+2) {
45.             return pivot;
46.         } else {
47.             return kthsmallest_triple(objs, k-gt-
    2, gt+2, right);
48.         }
49.     }
```