数据结构与算法
实验报告

第 一 次

姓名	DBQDSS
班级	数学强基 22
学号	*****
电话	*****
Email	*****
日期	2023-12-16

目录

任务 1：证明

1） 使用 O、Ω和Θ的定义，证明下面每一个等式：

a) $2\sqrt{n}+6=O(\sqrt{n})$
b) $n^2=\Omega(n)$
c) $log_2(n)=\Theta(\ln(n))$
d) $4^n\neq O(2^n)$

证明：$a)$令$T(n)=2\sqrt{n}+6$, 对于$\forall n>36, T(n)\leqslant 3\sqrt{n}$, 于是$T(n)=2\sqrt{n}+6=O(\sqrt{n})$

$b)$令$T(n)=n^2$, 对于$\forall n>2, T(n)>n$, 于是$T(n)=n^2=\Omega(n)$

$c)$令$T(n)=log_2(n)$, 由换底公式 $log_2(n)=\dfrac{ln(n)}{ln(2)}$, 令$k=\dfrac{1}{ln(2)}>0$

于是$T(n)=kln(n), \forall n>1, kln(n)\leqslant T(n)\leqslant kln(n)$

故$T(n)=log_2(n)=\Theta(ln(n))$

$d)$反证法：令$T(n)=4^n$, 若存在常数c, N_0, 满足$c>0, N_0\in\mathbb{N}$,

有：对于$\forall n>N_0, 4^n\leqslant c\cdot 2^n$恒成立

但 $4^n\leqslant c\cdot 2^n\Leftrightarrow 2^n\leqslant c\Leftrightarrow n\leqslant log_2 c$, 于是取 n 充分大即可导致原命题矛盾

于是 $T(n)=4^n\neq O(2^n)$

2） 使用数学归纳法证明$T(n)=\Omega(nlog(n))$。T(n)的定义式如下：

$$T(n)=\begin{cases}2T\left(\left\lceil\frac{n}{2}\right\rceil\right)+\frac{n}{2}, & n>1\\ 1, & n=1\end{cases}$$

证明：对 n 归纳，分奇偶两类来证明（同时考虑 $2n-1$ 与 $2n$ 的情形）

1) 当 $n=1$ 时，取 $c=1$即可，此时有：

$T(1)=1>1\cdot log1, \quad T(2)=2T(1)+1=3>2log2=2$

2) 假设当 $n\leqslant k-1$ 时，均存在正常数 c，

有 $T(2n-1)\geqslant c(2n-1)log(2n-1)$ 与 $T(2n)\geqslant c(2n)log(2n)$ 成立，

当 $n=k$ 时，

$$\begin{aligned}T(2k-1)&=2T(k)+\frac{2k-1}{2}=2T(k)+k-\frac{1}{2}\\ &\geqslant 2c_k klogk+k-1=c_k(2k)log(2k)+(1-2c_k)k-1\\ &\geqslant c_k(2k-1)log(2k-1)+(1-2c_k)k-1\geqslant c_{2k-1}(2k-1)log(2k-1)\end{aligned}$$

$$\begin{aligned}T(2k)&=2T(k)+\frac{2k}{2}=2T(k)+k\\ &\geqslant 2c_k klogk+k=c_k(2k)log(2k)+(1-2c_k)k\\ &\geqslant c_{2k}(2k)log(2k)\end{aligned}$$

故当 $n=k$ 时，原命题仍成立。

于是，由数学归纳法的原理，原命题成立。

任务 2：排序算法的实现

一、题目

请根据课上所讲，编程实现插入排序、选择排序、希尔排序、快速排序、归并排序。其中 Shell 排序中的间隔递减序列采用如下函数：

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

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结果编写测试程序检查是否排序成功。
- 进行上述 5 种排序算法的对比分析，总结个人收获。

二、算法设计

（一）插入排序（Insertion）

插入排序是一种最简单的排序方法，它的基本思想是将一个记录插入到已经排好序的有序表中，从而一个新的、记录数增 1 的有序表。在其实现过程使用双层循环，外层循环对除了第一个元素之外的所有元素，内层循环对当前元素前面有序表进行待插入位置查找，并进行移动。

此算法**具有稳定性。**

（二）选择排序（Selection）

选择排序的主要流程是：首先在未排序序列中找到最小（大）元素，存放到排序序列的起始位置，然后，再从剩余未排序元素中继续寻找最小（大）元素，然后放到已排序序列的末尾。以此类推，直到所有元素均排序完毕。

此算法**不具有稳定性。**

（三）希尔排序（Shell）

希尔排序是插入排序的一种改进，通过设置间隔 h 来比较和交换元素，然后逐渐缩小间隔，在每个间隔内递归执行插入排序方法。

在本任务中，我们采取后一间隔为前一间隔的 1/3 的方法进行操作。

此算法**不具有稳定性**。

（四）快速排序（Quicksort）

快速排序采用的是分治思想，即在一个无序的序列中选取一个任意的基准元素 pivot，利用 pivot 将待排序的序列分成两部分，前面部分元素均小于或等于基准元素，后面部分均大于或等于基准元素，然后采用递归的方法分别对前后两部分重复上述操作，直到将无序序列排列成有序序列。

此算法**不具有稳定性。**

（五）归并排序（Mergesort）

归并排序是建立在归并操作上的一种有效，稳定的排序算法，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。

此算法**具有稳定性。**

三、主干代码说明

（一）插入排序（Insertion）

```java
package sort;

2 个用法
public class Insertion extends SortAlgorithm{
    //插入排序
    public void sort(Comparable[] objs){
        int N = objs.length;
        for(int i = 1; i < N; i++){
            // 当 j > 0 或者 objs[j] < objs[j-1] 时，执行循环
            // objs[j] >= objs[j-1] 时，进入下一轮循环（因为0-j已经按照从小到大的顺序排好）
            for(int j = i; j > 0 && less(objs[j], objs[j-1]); j--)
                exchange(objs, j, j-1);
        }
    }
}
```

（二）选择排序（Selection）

```java
package sort;

1 个用法
public class Selection extends SortAlgorithm{

    // 选择排序（Selection）
    public void sort(Comparable[] objs) {
        for (int i = 0; i < objs.length - 1; i++) {
            int index = i;
            for (int j = i; j < objs.length; j++) {
                if (less(objs[j], objs[index])) {
                    index = j;
                }
            }
            exchange(objs, i, index);
        }
    }
}
```

（三）希尔排序（Shell）

```java
package sort;

1 个用法
public class Shell extends SortAlgorithm {

    // 希尔排序（Shell）
    @Override
    public void sort(Comparable[] objs) {

        // 引入数组长度
        int N = objs.length;

        // 初始化 h，并求出最大的 h。
        int h = 1;
        for (; h < N;) {
            h = 3 * h + 1;
        }

        // 循环进行希尔排序
        while (h >= 1) {
            for (int i = h; i < N; i++) {
                Comparable temp = objs[i];
                // 对间隔为 h 的子数组进行插入排序
                for (int j = i; j >= h && less(temp, objs[j - h]); j -= h)
                    exchange(objs, j, j-h);
            }
            // 缩小插入排序的间隔
            h = (h - 1) / 3;
        }
    }
}
```

（四）快速排序（Quicksort）

```java
package sort;

13 个用法  3 个继承者
public class Quicksort extends SortAlgorithm{

    public void sort(Comparable[] objs) {
        // 调用qsort方法进行快速排序
        qsort(objs, i: 0, j: objs.length - 1);
    }

    // 递归地将数组分为两部分并进行排序
    7 个用法  2 个重写
    public void qsort(Comparable[] objs, int i, int j) {
        // 查找轴点元素索引
        int pivotIndex = findpivot(objs, i, j);
        // 将轴点元素交换到末尾
        exchange(objs, pivotIndex, j);
        // 划分数组，返回划分的边界
        int k = partition(objs, i: i - 1, j, objs[j]);
        // 将轴点元素交换到划分边界
        exchange(objs, k, j);
        // 对左边部分排序
        if ((k - i) > 1) qsort(objs, i, j: k-1);
        // 对右边部分排序
        if ((j - k) > 1) qsort(objs, i: k+1, j);
    }

    // 寻找轴点元素
    4 个用法
    public int findpivot(Comparable[] objs, int i, int j) {
        if (less(objs[i], objs[j]) && less(objs[(i + j) / 2], objs[i])) {
            return i;
        } else if (less(objs[j], objs[i]) && less(objs[(i + j) / 2], objs[j])) {
            return j;
        } else {
            return (i + j) / 2;
        }
    }

    // 以轴点元素为基准，将数组划分为两部分
    4 个用法
    public int partition(Comparable[] objs, int i, int j, Comparable pivot) {
        do {
            while (less(objs[++i], pivot)); // 从左向右找到第一个大于等于轴点的元素
            while ((j != 0) && less(pivot, objs[--j])); // 从右向左找到第一个小于等于轴点的元素
            exchange(objs, i, j); // 交换找到的两个元素
        } while (i < j); // 重复这一过程，直到两个索引相遇
        exchange(objs, i, j); // 最后一次交换导致元素位置错误，所以需要再进行一次交换
        return i; // 返回轴点元素的最终位置
    }
}
```

（五）归并排序（Mergesort）

```java
package sort;

1 个用法
public class Mergesort extends SortAlgorithm {

    @Override
    public void sort(Comparable[] objs) {
        // 创建一个临时数组，存储排序过程中的元素
        Comparable[] temp = new Comparable[objs.length];
        // 调用mergesort方法进行归并排序
        mergesort(objs, low: 0, high: objs.length - 1, temp);
    }

    // 分治算法的主体-递归进行归并排序
    3 个用法
    public void mergesort(Comparable[] objs, int low, int high, Comparable[] temp) {
        // 若子数组的长度为1或0，则返回
        if (low >= high) {
            return;
        }
        // 计算出中点，将数组分为两部分
        int mid = low + (high - low) / 2;
        // 对左半部分归并排序
        mergesort(objs, low, mid, temp);
        // 对右半部分归并排序
        mergesort(objs, low: mid + 1, high, temp);
        // 合并两部分
        merge(objs, low, mid, high, temp);
    }

    // 合并操作
    1 个用法
    public void merge(Comparable[] objs, int low, int mid, int high, Comparable[] temp){
        // 将原数组元素复制到临时数组
        for(int i = low; i <= high; i++){
            temp[i] = objs[i];
        }
        // 重置两个索引，分别指向两个子数组的开头
        int i = low;
        int j = mid + 1;

        // 按顺序取出子数组中最小的元素并放回原数组
        for(int p = low; p <= high; p++) {
            if(i == mid + 1) {
                // 若左数组取完，则直接取右数组
                objs[p] = temp[j++];
            }else if (j > high) {
                // 若右数组取完，则取左数组
                objs[p] = temp[i++];
            }else if(less(temp[j], temp[i])) {
                // 若两数组都没有取完，较小的是j，则取j放回原数组
                objs[p] = temp[j++];
            }else {
                // 反之，则取i
                objs[p] = temp[i++];
            }
        }
    }
}
```

四、运行结果展示

IsSorted 类代码如下：

```java
package sort;
import static java.util.Arrays.sort;

public class IsSorted {

    public static void main(String[] args) {
        SortAlgorithm[] algs = new SortAlgorithm[]{new Insertion(), new Selection(), new Shell(), new Quicksort(), new Mergesort()};
        Double[] arr = GenerateData.getRandomData( N: 1000);

        //分别输出五种排序的结果
        //插入排序
        algs[0].sort(arr);
        System.out.println("插入排序的测试结果为:" + algs[0].isSorted(arr));
        //选择排序:
        algs[1].sort(arr);
        System.out.println("选择排序的测试结果为:" + algs[1].isSorted(arr));
        //希尔排序:
        algs[2].sort(arr);
        System.out.println("希尔排序的测试结果为:" + algs[2].isSorted(arr));
        //快速排序:
        algs[3].sort(arr);
        System.out.println("快速排序的测试结果为:" + algs[3].isSorted(arr));
        //归并排序:
        algs[4].sort(arr);
        System.out.println("归并排序的测试结果为:" + algs[4].isSorted(arr));
    }
}
```

运行结果为：

```
插入排序的测试结果为:true
选择排序的测试结果为:true
希尔排序的测试结果为:true
快速排序的测试结果为:true
归并排序的测试结果为:true

进程已结束，退出代码为 0
```

说明五种排序方法均已成功实现！

五、总结与收获

在本任务中，我对插入排序、选择排序、希尔排序、快速排序和归并排序这五种常见的排序方法，结合题目中提供的 SortAlgorithm 类和 GenerateData 类，进行了 Java 语言的实现，对它们的基本概念与代码过程进行了更加深入地思考与学习。

依照题目要求，为保证这些排序程序的可对比性，我每个排序算法均使用的是课堂上所讲授的步骤，没有对任何排序算法进行额外的优化，并调用 SortAlgorithm 类中的 isSorted 方法实现了对排序后对数组的有序性的检验。

特别地，在希尔排序的代码实现过程中，我采用了题目所要求方式的递减的间隔序列，能够提升排序过程的效率。

本任务不仅加强了我对于各类排序算法的理论知识，也使我对希尔排序算法的理解更加深刻了。

任务 3：排序算法性能测试和比较

一、题目

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（报告中请对 T 的取值进行说明）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；
- 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。

二、算法设计

使用 LineXY 类进行绘图，将五种排序算法的运行时间与数据规模的关系在同一张图中呈现，X 轴代表数据规模，Y 轴代表运行时间。

在图中绘制五条折线，包括插入排序(Insertion)，选择排序(Selection)，希尔排序(Shell)，快速排序(Quicksort)以及归并排序(Mergesort)。

由于不同算法之间可能会因为运行时间差距过大而造成显示上的问题，可以在纵坐标上取 10 的对数来平衡纵坐标的分布；同时，由于 2 的各次幂之间差距较大，因此横轴数据量取 2 的对数来平衡横坐标的分布。

其中，此处选用 T = 20。

三、主干代码说明

```java
package sort;

import org.jfree.chart.ChartFactory;
import org.jfree.chart.ChartPanel;
import org.jfree.chart.JFreeChart;
import org.jfree.chart.axis.NumberAxis;
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
    1 个用法
    public LineXYDemo(String title, SortAlgorithm[] algorithms, int[] dataSizes, int testIterations) {
        super(title);
        XYDataset dataset = createDataset(algorithms, dataSizes, testIterations);              // 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( width: 1500,  height: 500));
        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。
    }
```

```java
private JFreeChart createChart(XYDataset dataset) {
    // 使用已经创建好的dataset生成图表对象
    // 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
    JFreeChart chart = ChartFactory.createXYLineChart(
            title: "Sorting Algorithm Performance",      // 图表的标题
            "Data Size",                            // 横轴的标题名
            "Time (nanoseconds)",                                // 纵轴的标题名
            dataset,                        // 图表对象中使用的数据集对象
            PlotOrientation.VERTICAL,       // 图表显示的方向
            true,                           // 是否显示图例
            false,                          // 是否需要生成tooltips
            false                           // 是否需要生成urls
    );
    // 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
    // 同学们不必在意下面的代码
    // 可以将下面的代码去掉对比一下显示的不同效果
    chart.setBackgroundPaint(Color.WHITE);
    XYPlot plot = (XYPlot) chart.getPlot();

    NumberAxis rangeAxis = (NumberAxis) plot.getRangeAxis();
    rangeAxis.setRange( lower: 4, rangeAxis.getUpperBound()); // 设置纵轴的起始值为 4.0


    plot.setBackgroundPaint(Color.lightGray);
    plot.setAxisOffset(new RectangleInsets( top: 5.0,  left: 5.0,  bottom: 5.0,  right: 6.0));
    plot.setDomainGridlinePaint(Color.WHITE);
    plot.setRangeGridlinePaint(Color.WHITE);
    XYLineAndShapeRenderer renderer = (XYLineAndShapeRenderer) plot.getRenderer();
    renderer.setDefaultShapesVisible(true);
    renderer.setDefaultShapesFilled(true);
    return chart;
}
```

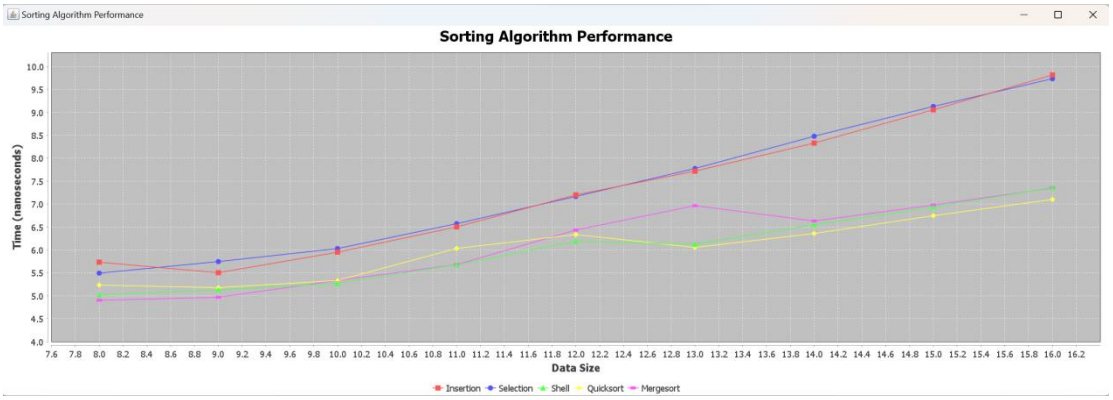
```java
private XYDataset createDataset(SortAlgorithm[] algorithms, int[] dataSizes, int testIterations) {
    XYSeriesCollection dataset = new XYSeriesCollection();

    for (SortAlgorithm algorithm : algorithms) {
        XYSeries series = new XYSeries(algorithm.getClass().getSimpleName());
        for (int dataSize : dataSizes) {
            Double[] data = GenerateData.getSortedData(dataSize);
            double averageTime = (SortTest.testForAcsending(algorithm, dataSize, testIterations));
            //将横坐标做二的对数间隔（log2）；纵坐标作十的对数间隔处理（lg）
            series.add( x: Math.log(dataSize) / Math.log(2), Math.log10(averageTime));
        }
        dataset.addSeries(series);
    }

    return dataset;
}
```

```java
public static void main(String[] args) {

    //绘制五条图线，包括插入排序，选择排序，希尔排序，快速排序以及归并排序）
    SortAlgorithm[] algorithms = {new Insertion(), new Selection(), new Shell(), new Quicksort(), new Mergesort()};
    //横坐标取2的八次方到16次方共9个数
    int[] dataSizes = new int[9];
    for (int i = 8; i <= 16; i++) {
        dataSizes[i - 8] = (int) Math.pow(2, i);
    }
    int testIterations = 30;
    LineXYDemo demo = new LineXYDemo( title: "Sorting Algorithm Performance", algorithms, dataSizes, testIterations);
    demo.pack();
    demo.setVisible(true);
}
```

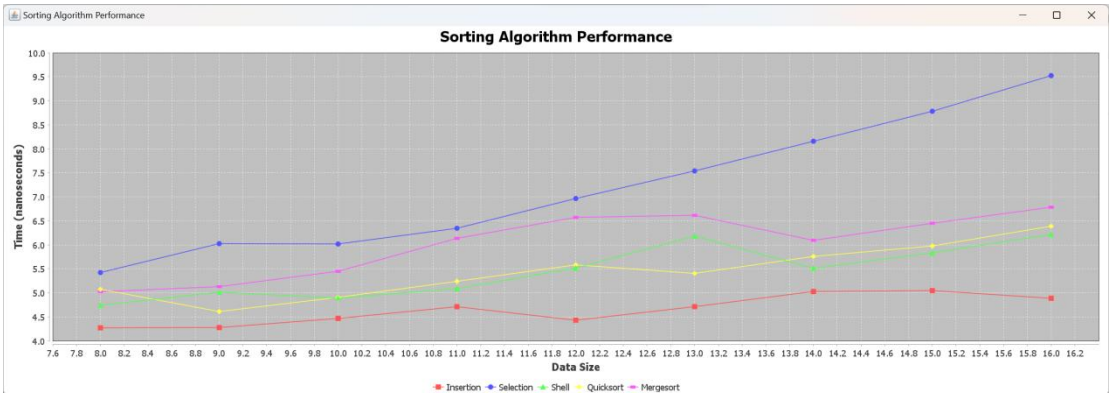
四、运行结果展示

下列三张图为上一题中五种排序算法分别在数据规模为 $2^8,2^9,...,2^{16}$ 上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：

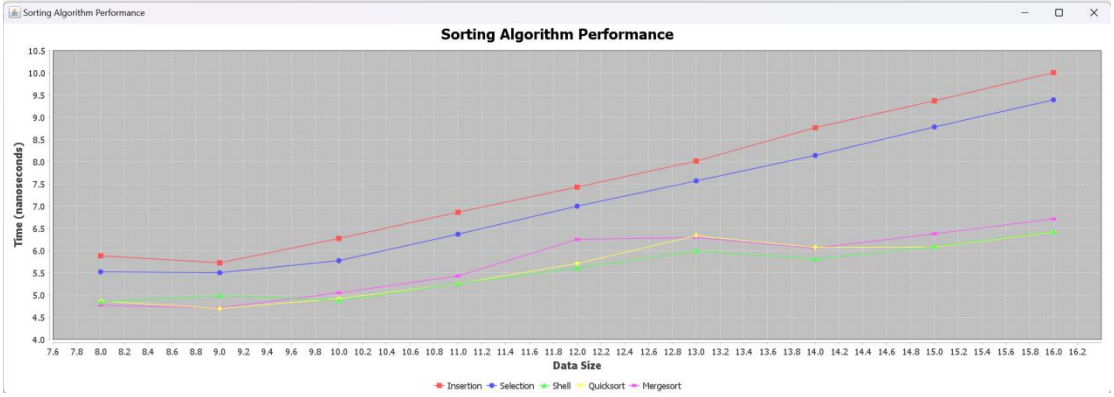
（一）均匀分布的随机数据：



（二）正序数据：



（三）逆序数据：



五、对结果的分析

（一）从一个算法的运行时间变化趋势和数据规模的变化角度：

①插入排序（Insertion）

插入排序在小规模数据上表现较好，其运行时间随着数据规模的增加呈现二次增长趋势。

因为需要大量的比较和移动操作，插入排序在正序数据上表现最好，而在逆序数据上表现最差。

②选择排序（Selection）

选择排序的运行时间随着数据规模的增加呈现明显的二次增长趋势，性能较差。

无论在规模多大的数据集上，因为要进行大量的比较和交换，选择排序的表现均是五种排序中最差的。

③希尔排序（Shell）

希尔排序的运行时间随着数据规模的增加呈现复杂的趋势，这取决于间隔递减序列的选择。

在大规模数据上，希尔排序的性能相对较好，尤其在适当选择间隔递减序列的情况下。

④快速排序（Quicksort）

随着数据规模的增加，快速排序表现出明显的优势，运行时间呈现近似线性的增长趋势。

此外，在不同数据分布下，对于随机数据、正序数据和逆序数据，快速排序的表现相对一致，这展现了它的普适性与高效性。

⑤归并排序（Mergesort）

归并排序的运行时间随着数据规模的增加呈现近似线性的增长趋势，性能相对较好。

此外，在不同数据分布下，对于随机数据、正序数据和逆序数据，归并排序的表现相对一致，这展现了它的普适性与高效性。

特别地，其在较小、较大数据集上优势明显，但在中等规模数据集上优势不明显。

结论：

在大规模数据集上，**快速排序**和**希尔排序**通常是较为优秀的选择，因为它们的运行时间呈线性或近似线性增长。

在小规模数据集上，**插入排序**和**选择排序**可能更适用，因为它们在较小规模上的性能相对较好。

归并排序在较小、较大规模数据集上的性能具有一定优势，但在中等规模数据上不如快速排序与希尔排序，同时，需要注意递减序列的选择。

（二）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：

①均匀分布的随机数据

快速排序与**希尔排序**在处理大规模随机数据时表现良好，因为它们平均时间复杂度较低。

归并排序在中等规模的数据集上表现不错，对于其间隔递减序列的选取可以提升排序效率。

因为它们的时间复杂度较高，**选择排序**和**插入排序**在大规模数据上相对较慢。

②正序数据

插入排序在正序数据上表现较好，因为它只需要少量的比较和移动操作。

快速排序在正序数据上仍然保持较好的性能，其稳定性也得到体现。

希尔排序的间隔序列在正序数据上可能影响其性能，但在中等规模数据上仍然比较高效。

③逆序数据

因为需要大量的比较和移动操作，插入排序和选择排序算法在此数据集之上运行时间最长。

快速排序和希尔排序在逆序数据上表现相对较好，尤其是希尔排序的分治法对逆序数据具有较好的适应性

结论：

在大规模数据集上，希尔排序和快速排序效果相差不大。

插入排序和选择排序适用于小规模数据，但在大规模数据上性能较差。

归并排序的性能介于插入排序和快速排序之间，在较小与较大规模数据上有一定优势，甚至性能优于快速排序与归并排序，中等规模反而不占优势。

六、总结与收获

通过对五种不同排序算法在不同数据规模、分布下的运行时间统计，可以注意到，随着数据规模增加，快速排序和希尔排序表现出较为线性的增长趋势，尤其在大规模数据上；相比之下，插入排序和选择排序在小规模数据上表现相对较好；而归并排序则在较小与较大规模数据上有一定优势，中等规模反而不占优势。

本任务深化了我对此五种排序算法性能与数据规模、分布关系的理解，使我对算法的性能有了更直观的感觉，为我未来在学习工作中选择与运用合适的算法提供了参考与经验。

任务 4：快速排序的再探讨与应用

一、题目

快速排序算法被誉为 20 世纪科学和工程领域的十大算法之一。前面的任务只是对快速排序的初识，下面从几个方面再更深入了解它：

① 优化快速排序：当待排序的数据量小于某个阈值时将递归的快速排序调用改为直接插入排序调用，按照这种策略的优化的快速排序算法进行实现、测试排序结果，并与任务 2 中没有优化的快速排序算法的执行效率进行对比；

② 在实际应用中经常会出现含有大量重复元素的数组，例如可能需要将大量人员资料按照生日排序，或者按照性别排序。给出使用①中完成的快速排序在数据规模为 2^{16} 的情况下，数据的重复率分别为 50%、60%、80%和 100%的运行时间的变化趋势图。结合①中的运行数据，给出观察结果；

③ 当遇到②中的重复性很高的数据序列时，快速排序还有很大的改进空间，方法是改进快速排序的划分方法，即将原有的二路划分（划分为比轴值大和不小于轴值）变成三路划分（划分为比轴值小，等于轴值和比轴值大）。三路划分的思路来自于经典的荷兰国旗问题。现要求自行查找三路划分的逻辑并实现之（高效的三路划分算法可以参考 J.Bentley 和 D.Mcllroy 的实现）。另外，用新的划分算法实现的快速排序重新对②完成实验并比较。

④ 在 N 个数据中找第 k 小元素($1 \leqslant k \leqslant N$)的问题可以有若干方法，请给出你能想到的方法，并简要分析每个方法的时间复杂度。在若干方法中，可以利用快速排序思想高效实现，请尽量独立思考，并最终给出设计思想、具体实现、测试以及时间复杂度分析。

二、数据结构设计

数据规模：

选用的数据规模为 $2^8 - 2^{16}$

数据分布：

根据任务要求，有三种数据分布：均匀分布的随机数据、正序数据和逆序数据。

用 N 来表示重复实验的次数

对于均匀分布的随机数据，可以使用 GenerateData.getRandomData(N)方法生成。

对于正序数据，可以使用 GenerateData.getSortedData(N)方法生成。

对于逆序数据，可以使用 GenerateData.getInversedData(N)方法生成。

分布不均匀数据，使用 getSortedSome、getRandomSome 和 getInversedSome 方法生成

对于重复数据，在 getRandomRepeat、getSortedRepeat、getInversedRepeat 方法生成

此处沿用上一题的 T = 20（在本任务的第四问中，取 T = 100）

三、算法设计

（一）优化快速排序（Optimized QuickSort）

采用分治策略，当数据规模小于某个阈值时，切换到直接插入排序，减少递归深度。在实验中，通过调整阈值，观察对快速排序性能的影响。

（二）重复元素的优化快速排序

考虑含有大量重复元素的情况，优化快速排序的划分策略。可以采用三路划分，将数组划分为小于、等于和大于枢纽元素的三个部分。

（三）三路划分的快速排序（Three-Way QuickSort）

使用三路划分，将数组划分为小于、等于和大于轴值的三个部分。

本算法适用于处理含有大量重复元素的数组，对于分布不均匀的数据，性能可能有显著提升。

（四）查找第 k 小元素的方法

利用快速排序的思想，通过划分数组找到第 k 小的元素，通过随机打乱数组顺序，避免最坏情况发生。

快速选择算法（QuickSelect）

1、算法思想：

快速选择是一种基于快速排序的思想，但只关心找到第 k 小的元素。快速选择通过在每次划分后选择一个子数组进行进一步处理，而不是递归处理两个子数组，从而减小问题的规模。

2、具体实现：

在 Quickselect 类中，采用 selectK 方法，接受一个数组 objs 和一个整数 k，并返回第 k 小的元素。它使用了快速选择的思想，在每次选择后递归调用其中一个子数组。

3、算法测试：

首先，编写测试函数，与暴力查找法的结果比较验证该算法的正确性

其次，计算不同数据量下两种算法的运行时间，进行性能比较

4、时间复杂度分析：

在最坏情况下，每次划分都只能排除一个元素，即数组每次只减少一个元素的规模。在这种情况下，时间复杂度为 $O(n^2)$，其中 n 是数组的长度。最坏情况通常发生在每次选择的基准元素都是当前子数组中的最大或最小元素时。在平均情况下，每轮划分平均能够将数组规模减半，类似于二分查找。因此，平均时间复杂度为 O(n)。

证明：设总共元素个数为 n

左右分区元素个数可能为：0 与 n-1 ，1 与 n-2，⋯ ，n-1 与 1

上述各组合出现概率相等，记 T_L、T_R 分别为左、右分区运行时间

于是有 $T(n) = T_L + T_R$

左分区有 $T_L = T_i + T_j + O(n)$

右分区有 $T_R = T_{n-i-1} + T_{n-j-1} + O(n)$

在平均情形下，$T_L = T_R = T(\frac{n}{2})$

综合上述两式有 $T(n) = T_L + T_R = T_i + T_j + O(n) + T_{n-i-1} + T_{n-j-1}$

由于两个子问题的平均状态下时间复杂度为 $T_i + T_{n-i-1} = O(n)$

于是有 $T(n) = 2T(\frac{n}{2}) + O(n)$

递归之，有 $T(n) = O(n) + O(log(n))$

于是，快速选择算法的平均时间复杂度为 $O(n)$

四、主干代码说明

（一）优化快速排序（Optimized QuickSort）

①优化快速排序算法（QuickSortO）

```java
package sort;

3 个用法  1 个继承者
public class Quicksort0 extends Quicksort{

    //优化快速排序算法的阀值
    3 个用法
    public Quicksort0(int gap){
        this.gap = gap;
    }

    7 个用法  1 个重写
    public void qsort(Comparable[] objs, int i, int j){
        if(j - i < 3){
            insertsort(objs,i,j);
        }else {
            int pivoindex = findpivot(objs,i,j);
            exchange(objs,pivoindex,j);
            int k = partition(objs, i: i-1,j,objs[j]);
            exchange(objs,k,j);
            if((k - i) > 1)qsort(objs,i, j: k-1);
            if((j - k) > 1)qsort(objs, i: k+1,j);}
    }


    2 个用法
    public void insertsort(Comparable[] objs,int low,int high){
        for(int i = low + 1; i <= high; i++){
            for(int j = i; j > low && less(objs[j], objs[j-1]); j--){
                exchange(objs, j, j: j-1);
            }
        }
    }


}
```

②画图（以随机数据为例）

```java
package sort;

import org.jfree.chart.ChartFactory;
import org.jfree.chart.ChartPanel;
import org.jfree.chart.JFreeChart;
import org.jfree.chart.axis.NumberAxis;
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.*;
```

```java
public LineXYDemo(String title, Quicksort[] algorithms, int[] dataSizes, int testIterations) {
    super(title);
    XYDataset dataset = createDatasetNewQuick(algorithms, dataSizes, testIterations);            // 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( width: 1500,  height: 500));
    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。
}
```

```java
private XYDataset createDatasetNewQuick(Quicksort[] algorithms, int[] dataSizes, int testIterations) {
    XYSeriesCollection dataset = new XYSeriesCollection();

    for (Quicksort algorithm : algorithms) {
        XYSeries series = new XYSeries( key: algorithm.getClass().getSimpleName() + algorithm.gap);
        for (int dataSize : dataSizes) {
            double averageTime = (SortTest.testForRandom(algorithm, dataSize, testIterations));
            //将横坐标取 log2，纵坐标取 lg
            series.add( x: Math.log(dataSize) / Math.log(2), Math.log10(averageTime));
        }
        dataset.addSeries(series);
    }
    for (Quicksort algorithm : algorithms) {
        XYSeries series = new XYSeries( key: algorithm.getClass().getSimpleName() + algorithm.gap + "Uneven");
        for (int dataSize : dataSizes) {
            double averageTime = (SortTest.testForRandomSome(algorithm, dataSize, testIterations));
            //将横坐标取 log2，纵坐标取 lg
            series.add( x: Math.log(dataSize) / Math.log(2), Math.log10(averageTime));
        }
        dataset.addSeries(series);
    }

    return dataset;
}
```

```java
        //绘制五条图线，包括未优化 与 阈值为3,7,15,31的优化快速排序算法
        Quicksort[] algorithms = new Quicksort[5];
        algorithms[0] = new Quicksort();
        int j = 1;
        for (int i = 2; i < 6; i++) {
            algorithms[j++] = new QuicksortO( gap: (int) Math.pow(2, i) - 1);
        }

        //重复比例
        double[] dataScales = new double[4];
        dataScales[0] = 0.5;
        dataScales[1] = 0.6;
        dataScales[2] = 0.8;
        dataScales[3] = 1.0;

        //横坐标取2的八次方到16次方共9个数
        int[] dataSizes = new int[9];
        for (int i = 8; i <= 16; i++) {
            dataSizes[i - 8] = (int) Math.pow(2, i);
        }

        int testIterations = 5;
        LineXYDemo demo = new LineXYDemo( title: "Sorting Algorithm Performance", algorithms, dataSizes, testIterations);
        demo.pack();
        demo.setVisible(true);


    }
```

（二）重复元素的优化快速排序

①生成有重复数据的数据

```java
    //生成重复的序列
    //用 0 <= proportion <= 1 表示重复率
    //随机重复
    1 个用法
    public static Double[] getRandomRepeat(int N,double proportion){
        Double[] numbers=getSortedData(N);
        for(int i = 0;i < N * proportion; i++){
            numbers[i] = -1.0;
        }
        shuffle(numbers, left: 0, numbers.length);
        return numbers;
    }

    //正序重复
    1 个用法
    public static Double[] getSortedRepeat(int N,double proportion){
        Double[] numbers = getSortedData(N);
        for(int i = 0; i < N * proportion; i++){
            numbers[i] = -1.0;
        }
        return numbers;
    }

    //逆序重复
    1 个用法
    public static Double[] getInversedRepeat(int N,double proportion){
        Double[] numbers = getInversedData(N);
        for(int i = (int) (N * proportion); i < N; i++){
            numbers[i] = 9999.0;
        }
        return numbers;
    }
```

```java
    // 将数组numbers中的[left,right)范围内的数据随机打乱
    3 个用法
    private static void shuffle(Double[] numbers, int left, int right){
        int N = right - left;
        Random rand = new Random();
        for(int i = 0; i < N; i++){
            int j = i + rand.nextInt( bound: N-i);
            exchange(numbers, i: i+left, j: j+left);
        }
    }
```

②画图

```java
package sort;

import org.jfree.chart.ChartFactory;
import org.jfree.chart.ChartPanel;
import org.jfree.chart.JFreeChart;
import org.jfree.chart.axis.NumberAxis;
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 {
```

```
    //绘制五条图线，包括未优化 与 阈值为3,7,15,31的优化快速排序算法
    Quicksort[] algorithms = new Quicksort[5];
    algorithms[0] = new Quicksort();
    int j = 1;
    for (int i = 2; i <= 5; i++) {
        algorithms[j++] = new QuicksortO( gap: (int) Math.pow(2, i) - 1);
    }

    //重复率
    double[] RepetitionRate = new double[4];
    RepetitionRate[0] = 0.5;
    RepetitionRate[1] = 0.6;
    RepetitionRate[2] = 0.8;
    RepetitionRate[3] = 1.0;

    int testIterations = 50;
    LineXYDemo demo = new LineXYDemo( title: "Sorting Algorithm Performance", algorithms, RepetitionRate, testIterations);
    demo.pack();
    demo.setVisible(true);
```

```
@ private XYDataset createDatasetForOptimizedQuickRepeat(Quicksort[] algorithms, double[] dataScales, int testIterations) {
    XYSeriesCollection dataset = new XYSeriesCollection();

    for (Quicksort algorithm : algorithms) {
        XYSeries series = new XYSeries( key: (algorithm.getClass().getSimpleName()) + algorithm.gap);
        for (double dataScale : dataScales) {
            double averageTime = (SortTest.testForRandomRepeat(algorithm, (int) Math.pow(2, 16), testIterations, dataScale));
            //将横坐标取 log2，纵坐标取 lg
            series.add(dataScale, Math.log10(averageTime));
        }
        dataset.addSeries(series);
    }

    return dataset;
}
```

```
    public LineXYDemo(String title, Quicksort[] algorithms, double[] dataScales, int testIterations) {
        super(title);
        XYDataset dataset = createDatasetForOptimizedQuickRepeat(algorithms, dataScales, testIterations);            //
        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( width: 1500,  height: 500));
        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
    }
```

③不同阈值的优化快速排序算法

```
package sort;

3 个用法  1 个继承者
public class QuicksortO extends Quicksort{

    //优化快速排序算法的阈值
    3 个用法
    public QuicksortO(int gap){
        this.gap = gap;
    }

    7 个用法  1 个重写
    public void qsort(Comparable[] objs, int i, int j){
        if(j - i < 3){
            insertsort(objs,i,j);
        }else {
            int pivoindex = findpivot(objs,i,j);
            exchange(objs,pivoindex,j);
            int k = partition(objs, i: i-1,j,objs[j]);
            exchange(objs,k,j);
            if((k - i) > 1)qsort(objs,i, j: k-1);
            if((j - k) > 1)qsort(objs, i: k+1,j);}
    }


    2 个用法
    public void insertsort(Comparable[] objs,int low,int high){
        for(int i = low + 1; i <= high; i++){
            for(int j = i; j > low && less(objs[j], objs[j-1]); j--){
                exchange(objs, j,  j: j-1);
            }
        }
    }


}
```

（三）三路划分的快速排序（Three-Way QuickSort）

①三路划分优化快速排序

```java
package sort;

2 个用法
public class QuicksortOO extends QuicksortO{

    2 个用法
    public QuicksortOO(int gap){
        super(gap);

    }
    7 个用法
    @Override
    public void qsort(Comparable[] objs,int i,int j){
        if(j-i<3){
            insertsort(objs,i,j);
        }else{
            int pivoindex = findpivot(objs,i,j);

            int[] edge = findedge2Ways(objs,i,j,pivoindex);
            int leftedge = edge[0];
            int rightedge = edge[1];

            if((leftedge - i) > 1)
                qsort(objs,i, j: leftedge-1);
            if((j - rightedge) > 1)
                qsort(objs, i: rightedge+1,j);
        }
    }
```

```java
    public int[] findedge2Ways(Comparable[] objs,int left,int right,int pivoindex){
        int lt = left;
        int gt = right;
        int index = left + 1;
        Comparable pivot = objs[pivoindex];
        exchange(objs,left,pivoindex);

        while (index <= gt) {
            int cmp = objs[index].compareTo(pivot);
            if (cmp < 0) {
                exchange(objs, lt++, index++);
            } else if (cmp > 0) {
                exchange(objs, index, gt--);
            } else {
                index++;
            }
        }


        return new int[]{lt, gt};

    }
}
```

②三种快速排序算法的正确性检验

```java
package sort;

import static java.util.Arrays.sort;

public class IsSorted {

    public static void main(String[] args){
        SortAlgorithm[] algs = new SortAlgorithm[]{new Quicksort(),new QuicksortO( gap: 7),new QuicksortOO( gap: 7)};
        Double[] arr=GenerateData.getRandomData( N: 1000);

        //分别输出三种排序的结果
        //插入排序
        algs[0].sort(arr);
        System.out.println("未优化的快速排序的测试结果为:" + algs[0].isSorted(arr));
        //选择排序;
        algs[1].sort(arr);
        System.out.println("一次优化快速排序的测试结果为:" + algs[1].isSorted(arr));
        //希尔排序;
        algs[2].sort(arr);
        System.out.println("三路划分快速排序的测试结果为:" + algs[2].isSorted(arr));
    }

}
```

③输出“不同阀值的优化快速排序在不同重复率的时间开销”

```java
public LineXYDemo(String title, Quicksort[] algorithms, double[] dataScales, int testIterations) {
    super(title);
    XYDataset dataset = createDatasetForOptimizedQuickRepeat(algorithms, dataScales, testIterations);
    JFreeChart chart = createChartRepeat(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( width: 1500,  height: 500));
    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。
}
```

```java
private XYDataset createDatasetForOptimizedQuickRepeat(Quicksort[] algorithms, double[] dataScales, int testIterations) {
    XYSeriesCollection dataset = new XYSeriesCollection();

    for (Quicksort algorithm : algorithms) {
        XYSeries series = new XYSeries( key: (algorithm.getClass().getSimpleName()) + algorithm.gap);
        for (double dataScale : dataScales) {
            double averageTime = (SortTest.testForRandomRepeat(algorithm, (int) Math.pow(2, 16), testIterations, dataScale));
            //将横坐标取 log2，纵坐标取 lg
            series.add(dataScale, Math.log10(averageTime));
        }
        dataset.addSeries(series);
    }

    return dataset;
}
```

```java
        //绘制五条图线，包括未优化 与 阈值为3,7,15,31的优化快速排序算法
        Quicksort[] algorithms = new Quicksort[5];
        algorithms[0] = new Quicksort();
        int j = 1;
        for (int i = 2; i <= 5; i++) {
            algorithms[j++] = new QuicksortOO( gap: (int) Math.pow(2, i) - 1);
        }

        //重复率
        double[] RepetitionRate = new double[4];
        RepetitionRate[0] = 0.5;
        RepetitionRate[1] = 0.6;
        RepetitionRate[2] = 0.8;
        RepetitionRate[3] = 1.0;

        int testIterations = 20;
        LineXYDemo demo = new LineXYDemo( title: "Sorting Algorithm Performance", algorithms, RepetitionRate, testIterations);
        demo.pack();
        demo.setVisible(true);
```

```java
private JFreeChart createChartRepeat(XYDataset dataset) {
    // 使用已经创建好的dataset生成图表对象
    // 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
    JFreeChart chart = ChartFactory.createXYLineChart(
            title: "Sorting Algorithm Performance",      // 图表的标题
            "Repetition rate",                           // 横轴的标题名
            "Time (nanoseconds)",                        // 纵轴的标题名
            dataset,                        // 图表对象中使用的数据集对象
            PlotOrientation.VERTICAL,       // 图表显示的方向
            true,                           // 是否显示图例
            false,                          // 是否需要生成tooltips
            false                           // 是否需要生成urls
    );
    // 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
    // 同学们不必在意下面的代码
    // 可以将下面的代码去掉对比一下显示的不同效果
    chart.setBackgroundPaint(Color.WHITE);
    XYPlot plot = (XYPlot) chart.getPlot();

    NumberAxis rangeAxis = (NumberAxis) plot.getRangeAxis();
    rangeAxis.setRange( lower: 4, rangeAxis.getUpperBound()); // 设置纵轴的起始值为 4.0


    plot.setBackgroundPaint(Color.lightGray);
    plot.setAxisOffset(new RectangleInsets( top: 5.0,  left: 5.0,  bottom: 5.0,  right: 6.0));
    plot.setDomainGridlinePaint(Color.WHITE);
    plot.setRangeGridlinePaint(Color.WHITE);
    XYLineAndShapeRenderer renderer = (XYLineAndShapeRenderer) plot.getRenderer();
    renderer.setDefaultShapesVisible(true);
    renderer.setDefaultShapesFilled(true);
    return chart;
}
```

④输出“比较同一阀值(7)的未优化、一次优化、三路划分优化在不同重复率下的性能”，调用的方法与②中的完全一致

```java
        Quicksort[] algorithms = new Quicksort[3];
        algorithms[0] = new Quicksort();
        algorithms[1] = new QuicksortO( gap: 7);
        algorithms[2] = new QuicksortOO( gap: 7);

        //重复率
        double[] RepetitionRate = new double[4];
        RepetitionRate[0] = 0.5;
        RepetitionRate[1] = 0.6;
        RepetitionRate[2] = 0.8;
        RepetitionRate[3] = 1.0;

        int testIterations = 20;
        LineXYDemo demo = new LineXYDemo( title: "Sorting Algorithm Performance", algorithms, RepetitionRate, testIterations);
        demo.pack();
        demo.setVisible(true);
    }
```

（四）查找第 k 小元素的方法

①快速选择算法代码（QuickSelect）

```java
package sort;

8 个用法
public class Quickselect extends Quicksort{
    2 个用法
    public Comparable selectK(Comparable[] objs, int k){
        if(k < 1 || k > objs.length){
            throw new IllegalArgumentException("OUT OF THE SCOPE");
        } else{
            return select(objs, k,  i: 0,  j: objs.length-1);
        }
    }
```

```java
package sort;

8 个用法
public class Quickselect extends Quicksort{
    2 个用法
    public Comparable selectK(Comparable[] objs, int k){
        if(k < 1 || k > objs.length){
            throw new IllegalArgumentException("OUT OF THE SCOPE");
        } else{
            return select(objs, k, i: 0, j: objs.length-1);
        }
    }
    3 个用法
    public Comparable select(Comparable[] objs, int k, int i, int j){
        int pivoindex = findpivot(objs, i, j);
        exchange(objs, pivoindex, j);
        int p = partition(objs, i: i - 1, j, objs[j]);
        exchange(objs, p, j);
        if(p == k)
            return objs[p];
        return p < k ? select(objs, k, i: p + 1, j) : select(objs, k, i, j: p - 1);
    }

    2 个用法
    public Comparable selectKDerect(Comparable[] objs,int k){
        if( k < 1 || k > objs.length){
            throw new IllegalArgumentException("OUT OF THE SCOPE");
        } else{
            return selectDerect(objs,k, i: 0, j: objs.length-1);
        }
    }

    1 个用法
    public Comparable selectDerect(Comparable[] objs,int k,int i,int j){
        sort(objs);
        return objs[k];
    }
}
```

②检测快速选择算法正确性

```java
public static double[] testDerect(Quickselect sel, int length, int T)
{
    Double[][] numbers =new Double[][]{GenerateData.getRandomData(length),GenerateData.getSortedData(length),
            GenerateData.getInversedData(length),GenerateData.getRandomSome(length)};

    double[] totalTime = new double[numbers.length];
    for(int j=0;j< numbers.length;j++){
        for(int i = 0; i < T; i++){
            totalTime[j] += timeDerect(sel, numbers[j], K: numbers.length/2);}
        totalTime[j]/=T;
    }
    return totalTime;
}


public static void main(String[] args) {


    System.out.println("验证快速选择算法的正确性");
    Integer[] array = {3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5};
    Quickselect quicksel = new Quickselect();

    // Test QuickSelect
    Comparable resultQuickSelect = quicksel.selectK(array, k: 5);
    System.out.println("QuickSelect Result: " + resultQuickSelect);

    // Test Direct Sorting
    Comparable resultDirectSort = quicksel.selectKDerect(array, k: 5);
    System.out.println("Direct Sorting Result: " + resultDirectSort);
```

```java
package sort;

import static java.util.Arrays.sort;

public class SelectTest extends SortTest{


    1个用法
    public static double timeQuick(Quickselect sel, Double[] numbers,int K){
        double start = System.nanoTime();
        sel.selectK(numbers,K);
        double end = System.nanoTime();
        return end - start;
    }
    1个用法
    public static double timeDerect(Quickselect sel, Double[] numbers,int K){
        double start = System.nanoTime();
        sel.selectKDerect(numbers,K);
        double end = System.nanoTime();
        return end - start;
    }

    0个用法
    public static double[] testQuick(Quickselect sel, int length, int T)
    {
        Double[][] numbers =new Double[][]{GenerateData.getRandomData(length),GenerateData.getSortedData(length),
                GenerateData.getInversedData(length),GenerateData.getSortedSome(length)};

        double[] totalTime = new double[numbers.length];
        for(int j=0;j< numbers.length;j++){
            for(int i = 0; i < T; i++){
                totalTime[j] += timeQuick(sel, numbers[j], K: numbers.length/2);}
            totalTime[j]/=T;
        }
        return totalTime;
    }
```

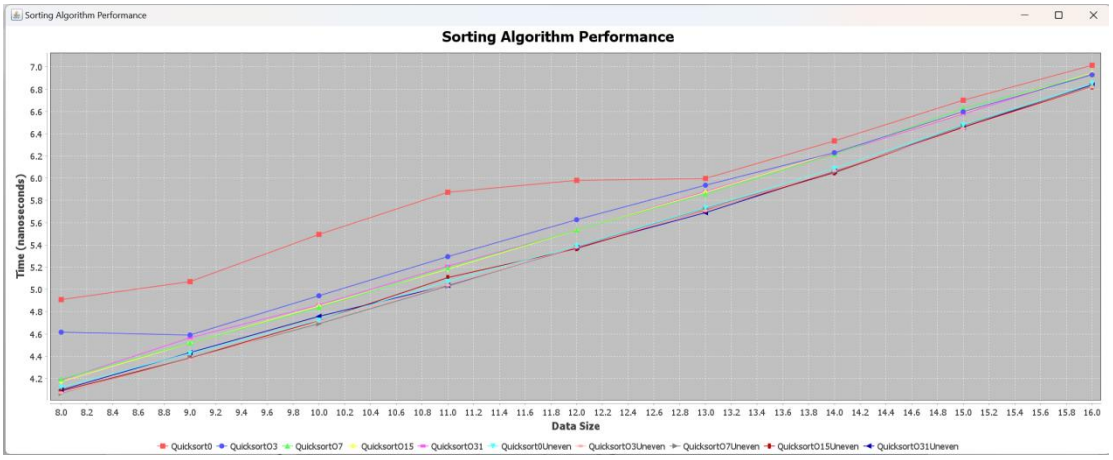
③比较不同数据类型、规模下，快速选择法与暴力查找法的时间开销

```java
    public static void main(String[] args) {

        System.out.println("以下每行分别是数据量在100, 1000, 10000, 100000时程序运行的时间开销");
        System.out.println();

        //数据量
        int[] dataLength = {100, 1000, 10000, 100000};
        double[][] elapsedTime = new double[dataLength.length][];
        String[] arr = {"随机数据", "正序数据", "逆序数据", "重复数据"};
        Quickselect quickselect = new Quickselect();
        for(int i = 0; i < dataLength.length; i++)
            elapsedTime[i] = testQuick(quickselect, dataLength[i], T: 100);
        System.out.println("快速选择法的时间开销：");
        for(double time[]: elapsedTime) {
            for (int k = 0; k < time.length; k++) {
                System.out.print(arr[k] + " ");
                System.out.printf("%6.4f ", time[k]);
            }
            System.out.println();
        }
        System.out.println();

        System.out.println("暴力查找法的时间开销");
        for(int i = 0; i < dataLength.length; i++)
            elapsedTime[i] = testDerect(quickselect, dataLength[i], T: 100);
        for(double time[]: elapsedTime) {
            for (int k = 0; k < time.length; k++) {
                System.out.print(arr[k] + " ");
                System.out.printf("%6.4f ", time[k]);
            }
            System.out.println();
        }
```

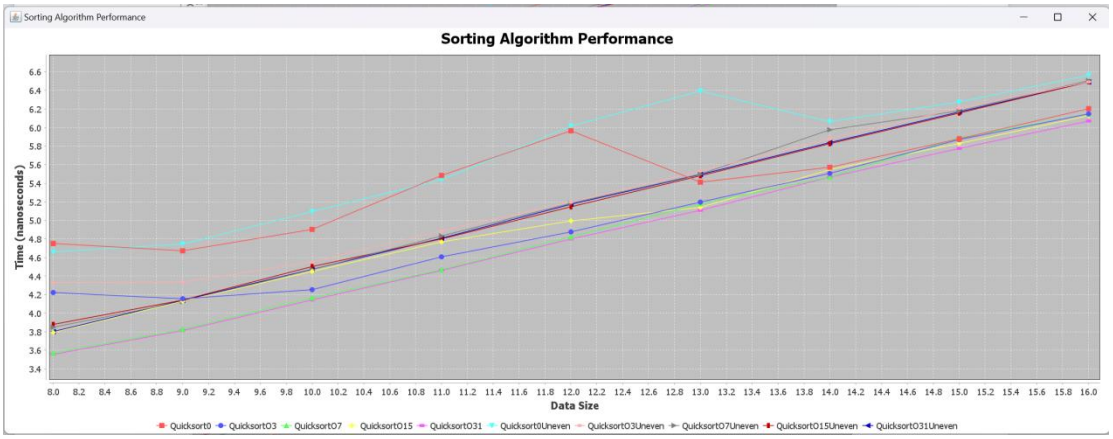
五、运行结果展示

（一）优化快速排序（Optimized QuickSort）

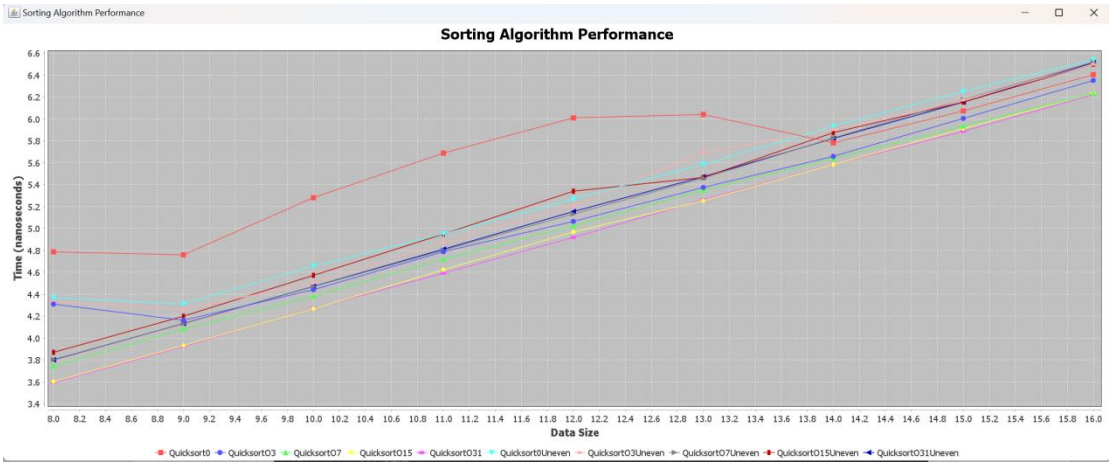
①分布均匀的随机数据



②正序数据



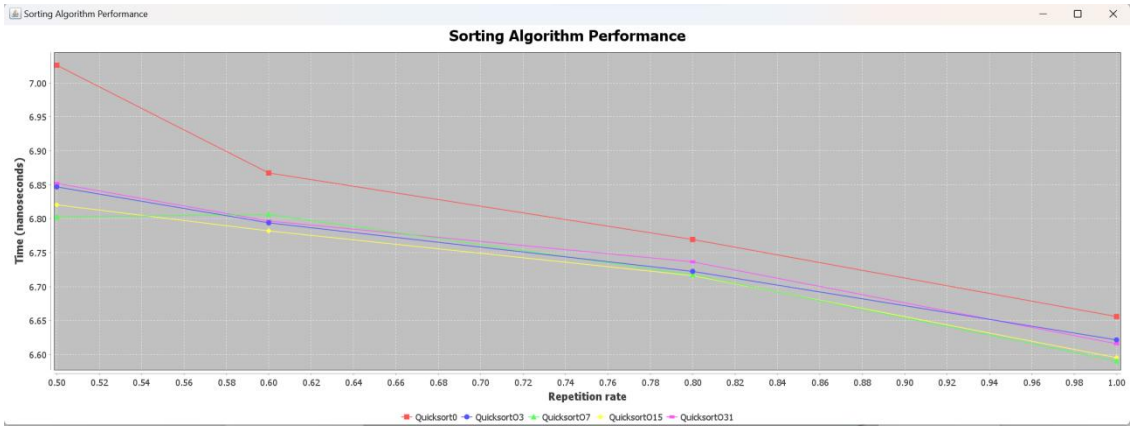
③逆序数据



其中，横坐标的 Uneven 代表为不均匀分布的数据。对于一次优化后的快速排序算法 QuickSortO（Optimized）采取不同的阈值进行测试。

（二）重复元素的优化快速排序

①不同阀值的优化快速排序在不同重复率的时间开销

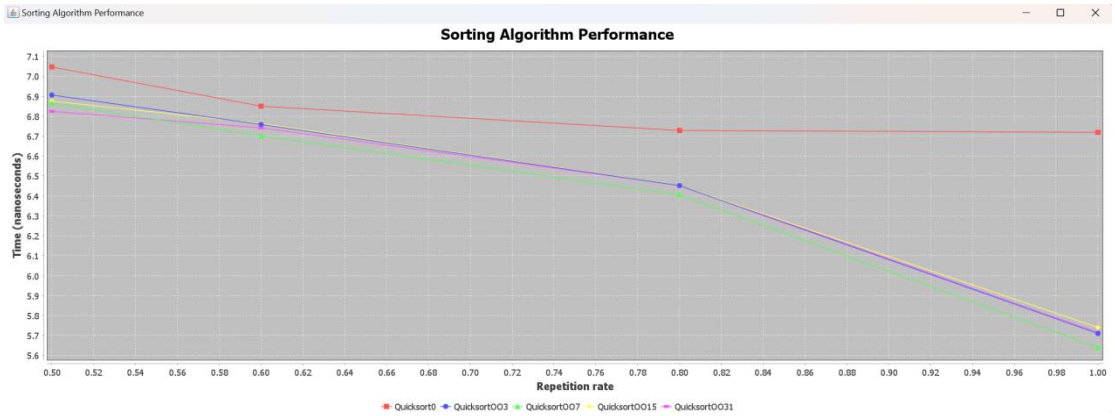


（三）三路划分的快速排序（Three-Way QuickSort）

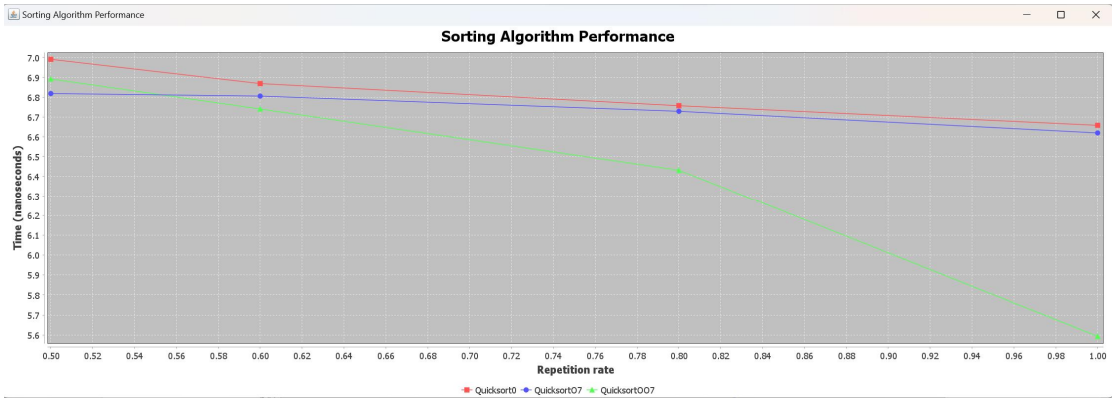
①三种快速排序方法的正确性检验

```
未优化的快速排序的测试结果为:true
一次优化快速排序的测试结果为:true
三路划分快速排序的测试结果为:true
```

②三路划分快速排序在不同阀值下的时间开销



③比较同一阀值(7)的未优化、一次优化、三路划分优化在不同重复率下的性能



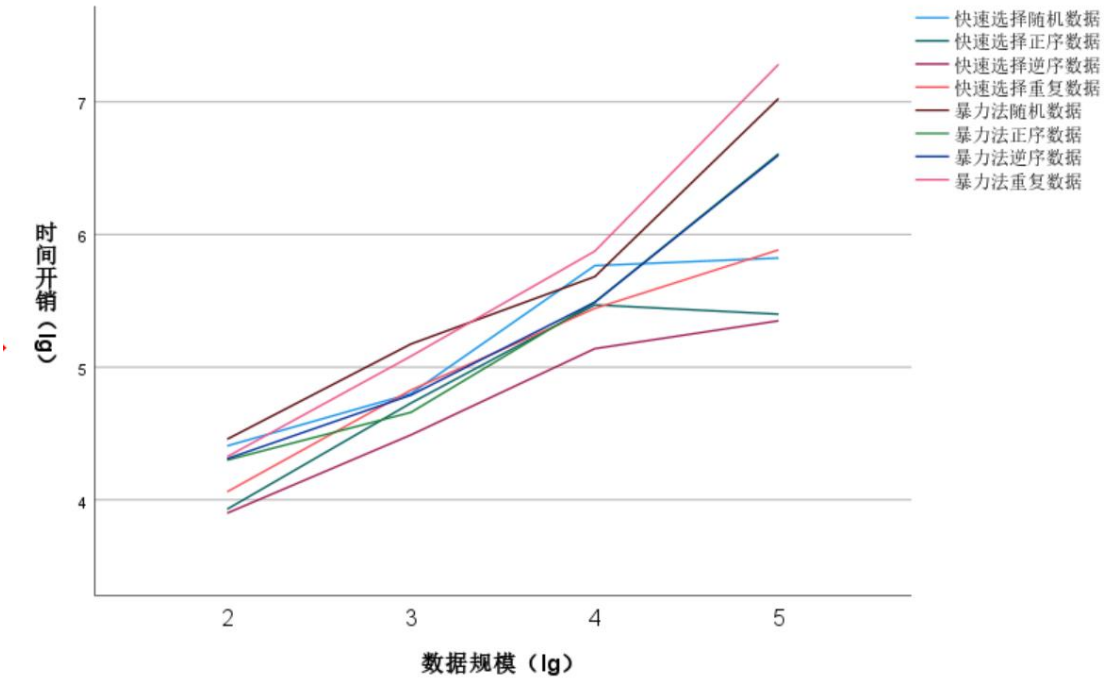
（四）查找第 k 小元素的方法

①检验快速选择算法的正确性

```
验证快速选择算法的正确性
QuickSelect Result: 4
Direct Sorting Result: 4
```

②比较不同数据类型、规模下，快速选择法与暴力查找法的时间开销

```
以下每行分别是数据量在100, 1000, 10000, 100000时程序运行的时间开销

快速选择法的时间开销：
随机数据 25517.9900 正序数据 8551.0000 逆序数据 7952.0100 重复数据 11512.0300
随机数据 63288.0200 正序数据 53817.0300 逆序数据 30986.9800 重复数据 67253.0200
随机数据 583299.0300 正序数据 296713.0700 逆序数据 136893.9900 重复数据 277672.0300
随机数据 665645.9600 正序数据 253401.9500 逆序数据 224850.0700 重复数据 766668.9300

暴力查找法的时间开销
随机数据 28669.0100 正序数据 19895.9700 逆序数据 20575.9800 重复数据 21152.0100
随机数据 149732.0000 正序数据 45615.9900 逆序数据 61509.0100 重复数据 121148.0300
随机数据 481671.0000 正序数据 309361.0100 逆序数据 309500.0300 重复数据 749939.0200
随机数据 10624600.9300 正序数据 4080160.1100 逆序数据 3937866.0000 重复数据 19208969.9000
```



结论：

①问题一

我们可以直观地在运行时间的图像中观察到，无论数据序列是随机、正序还是逆序，引入直接插入排序优化后的 QuickSortO 无论取任一种阈值，相比未优化的 QuickSort 都有明显提升。

针对不同的数据量，阈值的选择对算法的提升程度具有较强的相关性：通过引入直接插入排序使得快速排序的递归深度减小，从而减少了递归过程中的一些开销。这通常在处理较

小规模的数据时会带来性能提升，因为直接插入排序在小规模数据上的表现可能比快速排序更好。

综上，优化程度好坏，阈值的选择是关键，过小的阈值可能导致频繁的切换，而过大的阈值可能无法充分发挥插入排序的优势。

②问题二

通过图像中我们可以注意到，随着数据的重复度的提高，不同阈值的快速排序优化算法的运行时间都有一定程度的减少。此外，我们可以发现阈值取为 7 时的 QuickSortO 在该数据量下普遍表现更为优异。

通过观察纵轴的运行时长，我们可以发现在数据重复度较大时，未优化的 QuickSort 与优化后的 QuickSortO 差距减小，快速排序的优势被降低。

这是因为，随着数据重复率的增加，快速排序的性能可能会受到影响。高重复率的数据可能导致划分不均匀，使得快速排序的优势减弱。

③问题三

应用三路划分优化后的快速排序算法 QuickSortOO 对于问题②中重复数据的排序任务，同样是当阈值取 7 时的性能最为优异，于是在后续比较中对 QuickSortOO 取阈值为 7。

然后再用重复数据任务来测试未优化 QuickSort、一次优化 QuickSort 与三路划分优化 QuickSortOO，我们可以注意到，经过三路划分优化后的快速排序算法 QuickSortOO 在重复率较高的数据集中性能得到飞跃式地提高。一次优化的 QuickSortO 也有一定程度上的性能提升，与问题②的运行结果相符合。

采用三路划分的快速排序在处理包含大量重复元素的数据时性能可能更好。它通过减少元素交换的次数，更有效地处理了相同元素的情况，减少了不必要的交换。从而提高了性能。尤其在处理有大量相同元素的数据时，这一改进可能带来显著的性能提升。这个优化策略在处理实际应用中可能会更加高效。

④问题四

首先，编写测试方法，经检验，暴力查找与快速选择算法的查找结果一致，说明快速查找 QuickSelect 具有正确性。同时，在不同规模的数据上，快速选择算法均比暴力查找性能更好。

其次，可以注意到，当数据量较大时，快速选择算法优越性更加明显。而且，快速选择算法对于数据规模的增大所增加的时间开销也较少。

在实际应用中，选择子数组的策略也会影响快速选择的性能。例如，通过三路划分，可以有效处理有大量重复元素的情况，提高算法的效率。因此，快速选择在平均情况下具有较好的性能，但在最坏情况下的性能较差。

六、总结与收获

在本任务中，我学到了关于快速排序算法的更深入的知识。通过引入不同的阈值进行优化，当数据规模较小时，转而使用直接插入排序，有效减小递归深度，提高了排序算法的性能。此外，通过使用三路划分的思想来优化快速排序，解决了高重复率数据的排序问题，进一步提升了算法的效率。

这次实验使我更深入地理解了快速排序的原理和应用，并提高了对算法性能进行分析和优化的能力。