# 渐进分析和排序算法作业报告

第一次



| | |
|---|---|
| 姓名 | 史佳鑫 |
| 班级 | 软件 2302 班 |
| 学号 | 2236115195 |
| 电话 | 17342941319 |
| Email | 2236115195@stu.xjtu.edu |
| 日期 | 2024-10-26 |

# 目录

## 任务 1 证明

1）证明等式：

$$a)2\sqrt{n}+6=O(\sqrt{n})$$

要证明 $2\sqrt{n}+6=O(\sqrt{n})$，

我们需要找到一个常数 $C$ 和一个值$n_0$，使得对于所有 $n \geq n_0$，

不等式 $2\sqrt{n}+6 \leq C\sqrt{n}$成立。

我们可以选择 $C=4$。对于 $n \geq 9$，我们有：

$$2\sqrt{n}+6 \leq 4\sqrt{n}$$

因此，$2\sqrt{n}+6=O(\sqrt{n})$

$$b)n^2=\Omega(n)$$

要证明$n^2=\Omega(n)$，我们需要找到一个常数 $C$ 和一个值$n_0$，

使得对于所有 $n \geq n_0$，不等式$n^2 \geq Cn$ 成立。

我们可以选择 $C=1$。对于 $n \geq 1$，

我们有：$n^2 \geq n$

因此，$n^2=\Omega(n)$

$$c)\log_2 n=\Theta(\ln n)$$

要证明$\log_2 n=\Theta(\ln n)$，我们需要证明存在常数$C_1$和$C_2$以及一个值$n_0$，

使得对于所有 $n \geq n_0$，不等式$C_1 \ln n \leq \log_2 n \leq C_2 \ln n$ 成立。

使用对数的换底公式，

我们有：$\log_2 n = \frac{\ln n}{\ln 2}$

由于 $\ln 2$ 是一个常数（大约为 $0.693$），

我们可以设置$C_1 = \frac{1}{2\ln 2}$和$C_2 = \frac{1}{\ln 2}$。

对于所有 $n \geq 1$，我们有：$\frac{1}{2\ln 2}\ln n \leq \frac{\ln n}{\ln 2} \leq \frac{1}{\ln 2}\ln n$

这简化为：$\frac{\ln n}{2\ln 2} \leq \log_2 n \leq \frac{\ln n}{\ln 2}$

因此，$\log_2 n = \Theta(\ln n)$

d)$4^n \neq O(2^n)$

要证明$4^n \neq O(2^n)$，我们需要证明不存在常数 $C$ 和一个值$n_0$，

使得对于所有 $n \geq n_0$，不等式$4^n \leq C \cdot 2^n$成立。由于$4^n = (2^2)^n = 2^{2n}$，

我们有：$2^{2n} \leq C \cdot 2^n$

这意味着：$2^n \leq C$ 这显然是不可能的，

因为$2^n$随着 $n$ 的增加而无限增长。

因此，$4^n \neq O(2^n)$。

2）使用数学归纳法证明 $T(n) = O(n^2)$

当 $n = 1$ 时，$T(1) = 3$。

我们需要证明 $3 \leq C \cdot 1^2$对于某个常数 $C$ 成立。

我们可以选择 $C = 3$，因此基础情况成立。

假设对于某个 $k \geq 1$，$T(k) \leq Ck^2$成立。我们需要证明

$T(k+1) \leq C(k+1)^2$。根据 $T(n)$的定义，我们有：

$T(k+1) = T(k) + 2(k+1)$根据归纳假设，$T(k) \leq Ck^2$，

因此：$T(k+1) \leq Ck^2 + 2(k+1)$

我们需要证明 $Ck^2 + 2(k+1) \leq C(k+1)^2$

展开右边，我们得到：$C(k+1)^2 = C(k^2 + 2k + 1) = Ck^2 + 2Ck + C$

我们需要找到一个常数 $C$，使得：$Ck^2 + 2(k+1) \leq Ck^2 + 2Ck + C$

这简化为：$2k + 2 \leq 2Ck + C$

对于足够大的 $k$，我们可以选择 $C = 2$，

因此：$2k + 2 \leq 4k + 2$ 这在 $k \geq 1$ 时成立。

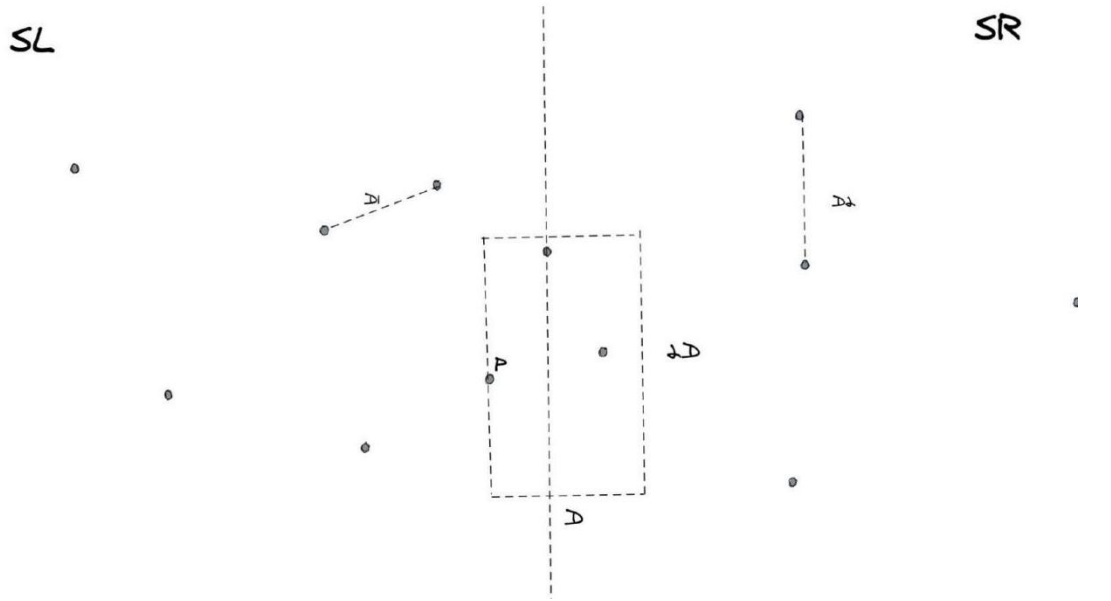
因此，根据数学归纳法，$T(n) = O(n^2)$

任务 2 设计算法要多思考

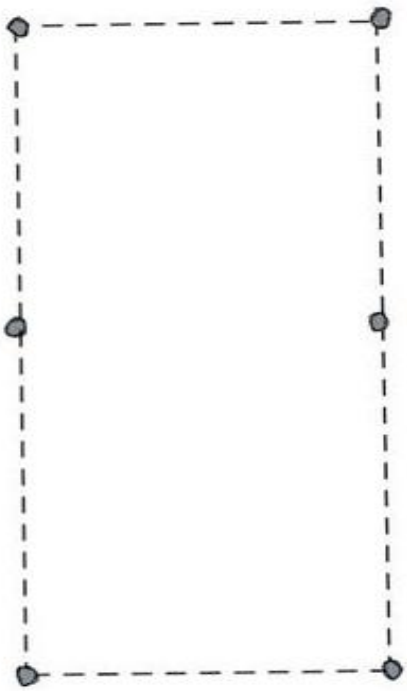
算法描述:

排序: 将所有点按照它们的 x 坐标进行排序。这一步可以使用快速排序或归并排序，时间复杂度为 O(nlogn)。

分治: 将排序后的点集分成两部分，每部分大约有 n/2 个点。拆分结束之后，我们只需要分别统计左边部分的最近点对、右边部分的最近点对，以及一个点在左边一个点在右边的最近点对即可。首先我们通过递归调用可以获得左边部分 SL 的最短距离 D1 以及右边部分 SR 的最短距离 D2。我们取 D=min(D1,D2)，也就是左右两边最短距离的最小值。对于一个点在左边一个点在右边的最近点对这种情况，我们可以通过已经算出的 D 进行限制筛选。实际上我们在左侧随便选择一个点 p，要想和 p 点构成最近点对，必须在下图这个虚线框起来的范围内。




这个虚线构成的框是一个长方形，它的宽是 D，长是 2D。这是怎么来的呢？其实很简单，对于 p 点来说，要想和他构成全局的最近点对，那么距离它的距离一定要小于目前的最优解 D。既然距离要小于 D，那么意味着它们的横纵坐标之差的绝对值必须也要小于 D。并且实际上虚线框内最多只有 6 个点（包括 P）



在上图当中，一共有 6 个点，这 6 个点两两之间的最短距离是 D，这是最极端的情况。无论我们如何往其中加入点，都一定会产生两个点之间的距离小于 D。

合并：在合并步骤中，我们需要考虑跨越两部分的点对。我们将点集分成左右两个部分之后，对右侧部分按照纵坐标进行排序，对于左侧的点(x, y)而言，我们只需要筛选出右侧满足纵坐标范围在(y - d, y + d)范围内的点，这样的点最多只有 6 个。我们可以利用二分法找到纵坐标大于 y - d 的最小的点，然后依次枚举之后的 6 个点即可。

更新最小距离：如果找到更小的距离，更新最小距离和对应的点对。

时间复杂度分析:

排序：代码开始时对所有点按照横坐标进行排序，这一步的时间复杂度是 O(nlogn)。

递归：由于每次递归都将问题规模减半，递归的深度是 logn，每层递归的时间复杂度是 O(n)（因为每层递归都需要处理 n 个点），所以递归部分的总时间复杂度是 O(nlogn)。

划分：在合并步骤中，代码将点集分成两个部分。这一步需要遍历所有点，时间复杂度是 O(n)。

右侧点集排序：对 right 点集按照纵坐标进行排序，这一步的时间复杂度是 O(mlogm)，其中 m 是 right 点集的大小，最坏情况下 m 可以接近 n/2，所以时间复杂度是 O(nlogn)。

二分查找：对于 left 中的每个点，使用二分查找在 right 中找到可能构成更小距离的点。二分查找的时间复杂度是 O(logm)，对于每个点

都需要进行一次二分查找，所以这部分的时间复杂度是 O(nlogm)，最坏情况下是 O(nlogn)。

检查点对：对于每个在二分查找中找到的点，检查其与 right 中接下来的 6 个点的距离。这一步对于 left 中的每个点都需要进行，时间复杂度是 O(6m)，最坏情况下是 O(6n)，即 O(n)。

综合以上步骤，算法的总时间复杂度是 O(nlogn) + O(n) + O(nlogn) + O(nlogn) + O(n) = O(nlogn)。这是因为 O(nlogn)项是主导项，而其他 O(n)项在渐进意义上可以忽略。

任务 3

1、 题目 排序算法的实现

2、 其他四种排序方法类的实现

Selection

```java
public class Selection extends SortAlgorithm {
    1 个用法
    public void sort(Comparable[] objs) {
        int N = objs.length;
        for (int i = 0; i < N - 1; i++) {
            int minIndex = i;
            for (int j = i + 1; j < N; j++) {
                if (less(objs[j], objs[minIndex])) {
                    minIndex = j;
                }
            }
            if (minIndex != i) {
                exchange(objs, i, minIndex);
            }
        }
        if (!isSorted(objs)) {
            throw new RuntimeException("Selection sort failed");
        }
    }
}
```

Shell1

```java
public class Shell1 extends SortAlgorithm {
    1 个用法
    public void sort(Comparable[] objs) {
        int N = objs.length;
        int gap = N/2;
        // 使用不同的间隔进行排序
        while (gap > 0) {
            for (int i = gap; i < N; i++) {
                Comparable temp = objs[i];
                int j;
                for (j = i; j >= gap && less(temp, objs[j - gap]); j -= gap) {
                    objs[j] = objs[j - gap];
                }
                objs[j] = temp;
            }
            gap = gap / 2; // 更新间隔
        }
        if (!isSorted(objs)) {
            throw new RuntimeException("Shell1 sort failed");
        }
    }
}
```

Shell2

```
0 个用法
public class Shell2 extends SortAlgorithm {
    1 个用法
    public void sort(Comparable[] objs) {
        int N = objs.length;
        // 计算Hibbard间隔序列
        int[] gaps = new int[(int) (Math.log(N + 1) / Math.log(2))];
        for (int k = 0; k < gaps.length; k++) {
            gaps[k] = (1 << (k + 1)) - 1; // 2^(k+1) - 1
        }

        for (int gapIndex = gaps.length - 1; gapIndex >= 0; gapIndex--) {
            int gap = gaps[gapIndex];
            // 插入排序
            for (int i = gap; i < N; i++) {
                Comparable temp = objs[i];
                int j = i;
                while (j >= gap && less(temp, objs[j - gap])) {
                    objs[j] = objs[j - gap];
                    j -= gap;
                }
                objs[j] = temp;
            }
        }
        if (!isSorted(objs)) {
            throw new RuntimeException("Shell2 sort failed");
        }
    }
}
```

Shell3

```java
import static java.lang.Math.pow;

0 个用法
public class Shell3 extends SortAlgorithm {
    1 个用法
    public void sort(Comparable[] objs) {
        int N = objs.length;
        // 计算Knuth间隔序列
        int[] gaps = new int[N];
        int k = 0;
        for (int gap = 1; gap < N; gap = (int) ((pow(3,gap) + 1) / 2)) {
            gaps[k++] = gap;
        }

        for (int gapIndex = k - 1; gapIndex >= 0; gapIndex--) {
            int gap = gaps[gapIndex];
            // 插入排序
            for (int i = gap; i < N; i++) {
                Comparable temp = objs[i];
                int j = i;
                while (j >= gap && less(temp, objs[j - gap])) {
                    objs[j] = objs[j - gap];
                    j -= gap;
                }
                objs[j] = temp;
            }
        }
        if (!isSorted(objs)) {
            throw new RuntimeException("Shell3 sort failed");
        }
    }
}
```

Quick

```java
public class Quick extends SortAlgorithm {
    // private static final int INSERTION_SORT_THRESHOLD = 9;

    1 个用法
    public void sort(Comparable[] objs) {
        sort(objs, left: 0, right: objs.length - 1);
        if (!isSorted(objs)) {
            throw new RuntimeException("Quick sort failed");
        }
    }

    3 个用法
    private void sort(Comparable[] objs, int left, int right) {
        while (left < right) {
         //  if (right - left < INSERTION_SORT_THRESHOLD) {
                // insertionSort(objs, left, right);
                // break; // 使用插入排序后直接返回
          //  }
            int pivotIndex = medianOfThree(objs, left, right);
            swap(objs, pivotIndex, right); // 将枢轴移到末尾
            int newPivotIndex = partition(objs, left, right);
            // 递归较小的部分
            if (newPivotIndex - left < right - newPivotIndex) {
                sort(objs, left, right: newPivotIndex - 1);
                left = newPivotIndex + 1; // 处理右边部分
            } else {
                sort(objs, left: newPivotIndex + 1, right);
                right = newPivotIndex - 1; // 处理左边部分
            }
        }
    }

    1 个用法
    private int medianOfThree(Comparable[] objs, int left, int right) {
        int center = (left + right) / 2;
        if (less(objs[center], objs[left])) swap(objs, left, center);
        if (less(objs[right], objs[left])) swap(objs, left, right);
        if (less(objs[right], objs[center])) swap(objs, center, right);
```

```
    1 个用法
    private int partition(Comparable[] objs, int left, int right) {
        Comparable pivot = objs[right];
        int i = left;
        for (int j = left; j < right; j++) {
            if (less(objs[j], pivot)) {
                swap(objs, i, j);
                i++;
            }
        }
        swap(objs, i, right);
        return i;
    }

    0 个用法
    private void insertionSort(Comparable[] objs, int left, int right) {
        for (int i = left + 1; i <= right; i++) {
            Comparable tmp = objs[i];
            int j = i - 1;
            while (j >= left && less(tmp, objs[j])) {
                objs[j + 1] = objs[j];
                j--;
            }
            objs[j + 1] = tmp;
        }
    }

    6 个用法
    private void swap(Comparable[] objs, int i, int j) {
        Comparable temp = objs[i];
        objs[i] = objs[j];
        objs[j] = temp;
    }


}
```

## Merge

```java
public class Merge extends SortAlgorithm {
    3 个用法
    private void sort(Comparable[] objs, int left, int right) {
        if (left < right) {
            int mid = (left + right) / 2;
            sort(objs, left, mid);
            sort(objs, left: mid + 1, right);
            merge(objs, left, mid, right);
        }
    }
    1 个用法
    private void merge(Comparable[] objs, int left, int mid, int right) {
        Comparable[] temp = new Comparable[right - left + 1];
        int i = left, j = mid + 1, k = 0;
        while (i <= mid && j <= right) {
            if (less(objs[i], objs[j])) {
                temp[k++] = objs[i++];
            } else {
                temp[k++] = objs[j++];
            }
        }
        while (i <= mid) {
            temp[k++] = objs[i++];
        }
        while (j <= right) {
            temp[k++] = objs[j++];
        }
        for (i = left, k = 0; i <= right; i++, k++) {
            objs[i] = temp[k];
        }
    }

    1 个用法
    public void sort(Comparable[] objs) {
        sort(objs, left: 0, right: objs.length - 1);
```

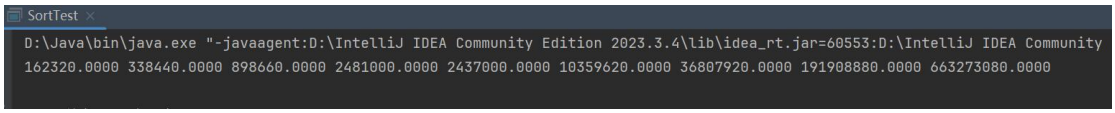
```java
    public void sort(Comparable[] objs) {
        sort(objs, left: 0, right: objs.length - 1);
        if (!isSorted(objs)) {
            throw new RuntimeException("Merge sort failed");
        }
    }
}
```

任务 4

1、 题目 排序算法性能测试和比较

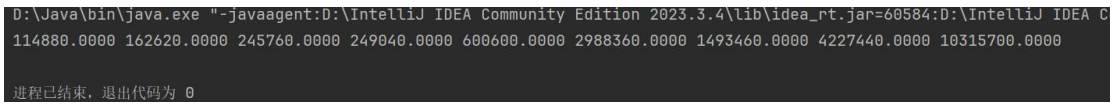
2、 分别对 Insertion、Selection、Shell1、Shell2，Shell3,Quicksort 和 Mergesort 进行不同数据规模的排序测试，结果如下：

Insertion

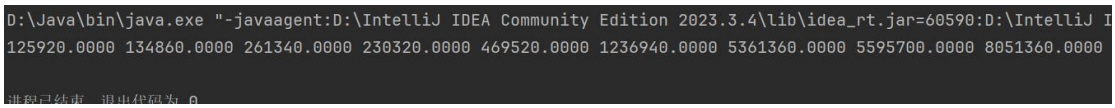



Selection

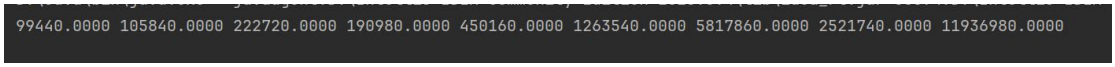

Shell1



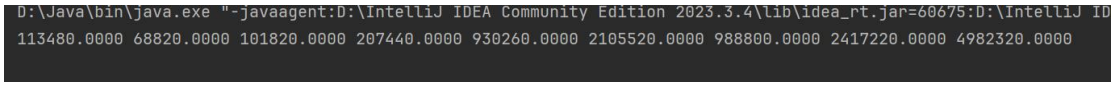

Shell2



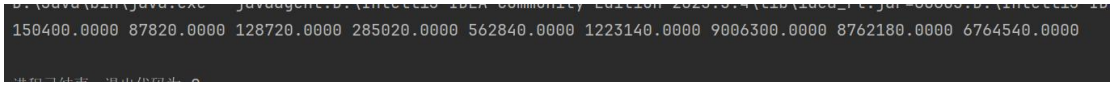

Shell3



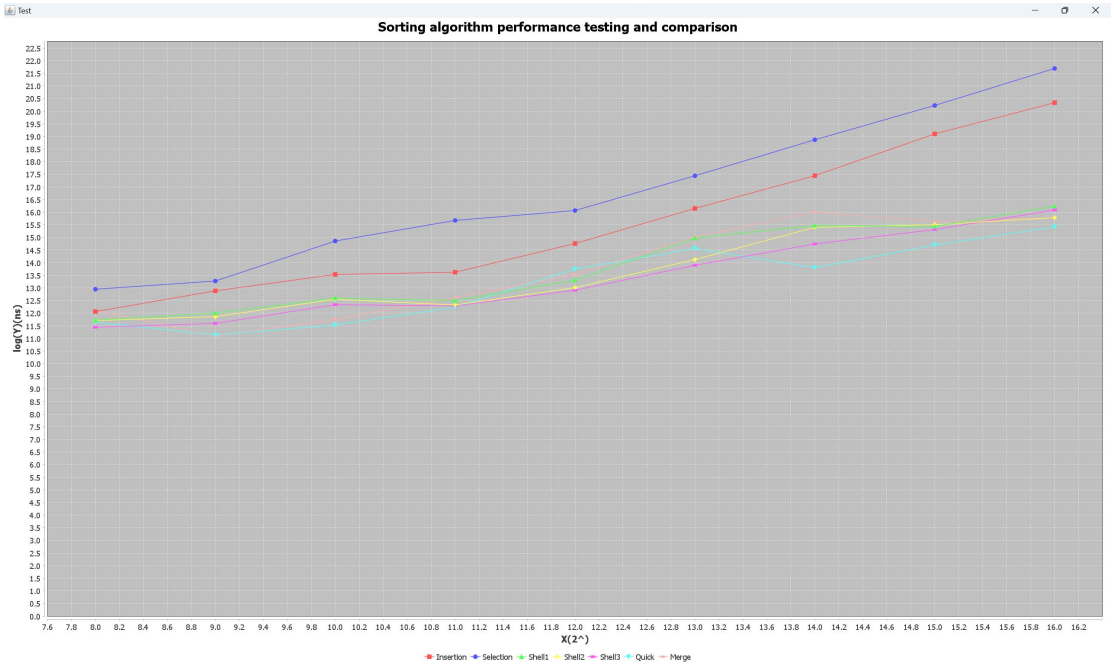

Quicksort




Mergesort




考虑到算法之间运行时间差异过大会造成显示上的问题，将运行时间使用取对数的方式调整比例尺，具体结果如下表：




3、 结果总结

1. 各算法的时间复杂度

首先，考虑各个算法的理论时间复杂度：

插入排序：平均和最坏情况为 $O(n^2)$，适合小规模数据。

选择排序：平均和最坏情况为 $O(n^2)$，性能较差。

希尔排序：依赖于增量选择，可达到 $O(n^{3/2})$，通常表现优于 $O(n^2)$ 的排序。

快速排序：平均情况 $O(n \log n)$，最坏情况 $O(n^2)$，但通常是非常高效的选择。

归并排序：稳定且具有 $O(n \log n)$ 的时间复杂度。

2. 数据规模的影响

根据运行时间数据，随着数据规模的增加，所有排序算法的运行时间均呈现出显著的增长趋势。尤其是 $O(n^2)$ 的排序算法（如插入和选

择排序）在较大数据规模（如 2^16）时，运行时间急剧增加，显示出其在处理大规模数据时的低效。

3. 数据分布对算法性能的影响

在均匀分布、正序序列和逆序序列的测试中，算法表现各异：

均匀分布：所有算法在此情况下表现相对一致，但 $O(n^2)$ 的算法时间依然较高。

正序序列：插入排序在此情况下表现最优，因为它的最佳情况是 $O(n)$。其他算法表现没有明显差异。

逆序序列：插入排序和选择排序表现最差，因为它们在最坏情况下的时间复杂度为 $O(n^2)$，而快速排序和归并排序在此情况下依然表现良好。

4. 不同算法之间的相对差异

根据实验结果：

快速排序和归并排序：在大部分情况下表现最好，尤其是在较大的数据规模上，能够处理得非常迅速。

希尔排序：在中等规模的数据上表现较好，但在更大的数据规模上效果逐渐下降，尤其是第二和第三种希尔排序。

插入排序和选择排序：表现最差，特别是在逆序序列中，显示出明显的劣势。

总结

总体来看，对于大规模数据，推荐使用快速排序或归并排序等 $O(n \log n)$ 的算法，尤其是在处理随机或逆序数据时。这些算法在时间复

杂度上具有优势，且在实际应用中能够有效减少排序时间。相对而言，插入排序和选择排序不适合处理大规模数据，尤其是在面对逆序或较复杂的数据分布时。

任务 5 子任务 1

1、 题目 探索快速排序的递归深度

2、 数据设计

增加一个整数类型的参数 depth，每递归调 用一次让 depth 加 1，每从递归中返回就让 depth 减 1，在整个执行过程中，depth 的最大 值即为快速排序的递归深度

3、 算法设计

递归深度追踪：在 sort 方法中，每次递归调用时将 depth 增加 1。

最大深度更新：在每次递归开始时更新 maxDepth，以便记录当前最大递归深度。

右边部分的处理：在递归时，不再使用 left 和 right 变量的更新逻辑，直接递归调用右部分，确保每次递归都增加深度

4、 主干代码说明

```java
public class Quicktest extends SortAlgorithm {
    3 个用法
    private int maxDepth = 0; // 用于记录最大深度

    1 个用法
    public void sort(Comparable[] objs) {
        sort(objs, left: 0, right: objs.length - 1, depth: 1); // 从1开始，因为初始深度为1
        if (!isSorted(objs)) {
            throw new RuntimeException("Quicktest sort failed");
        }
        System.out.println("Maximum recursion depth: " + maxDepth);
    }

    5 个用法
    private void sort(Comparable[] objs, int left, int right, int depth) {
        // 更新最大深度
        maxDepth = Math.max(maxDepth, depth);

        if (left < right) {
            int pivotIndex = medianOfThree(objs, left, right);
            swap(objs, pivotIndex, right);
            int newPivotIndex = partition(objs, left, right);
            // 递归较小的部分
            if (newPivotIndex - left < right - newPivotIndex) {
                sort(objs, left, right: newPivotIndex - 1, depth: depth + 1); // 递归左边
                sort(objs, left: newPivotIndex + 1, right, depth: depth + 1); // 递归右边
            } else {
                sort(objs, left: newPivotIndex + 1, right, depth: depth + 1); // 递归右边
```

```java
    1 个用法
    private int medianOfThree(Comparable[] objs, int left, int right) {
        int center = (left + right) / 2;
        if (less(objs[center], objs[left])) swap(objs, left, center);
        if (less(objs[right], objs[left])) swap(objs, left, right);
        if (less(objs[right], objs[center])) swap(objs, center, right);
        return center; // 返回中位数的索引
    }

    1 个用法
    private int partition(Comparable[] objs, int left, int right) {
        Comparable pivot = objs[right];
        int i = left;
        for (int j = left; j < right; j++) {
            if (less(objs[j], pivot)) {
                swap(objs, i, j);
                i++;
            }
        }
        swap(objs, i, right);
        return i;
    }

    6 个用法
    private void swap(Comparable[] objs, int i, int j) {
        Comparable temp = objs[i];
        objs[i] = objs[j];
        objs[j] = temp;
```

```java
import java.util.Scanner;

public class TicTacToe {
    19 个用法
    private static char[][] board = new char[3][3];
    18 个用法
    private static char currentPlayer = 'X';

    public static void main(String[] args) {...}
    //初始化棋盘，将所有位置设置为空位 ' '
    1 个用法
    private static void initializeBoard() {...}
    //打印当前棋盘的状态。
    2 个用法
    private static void printBoard() {...}
    //检查玩家的移动是否有效，即所选位置是否为空
    1 个用法
    private static boolean isValidMove(int row, int col) {...}
    //检查是否有玩家赢得游戏，通过检查棋盘上的行、列和对角线
    1 个用法
    private static boolean checkWinner() {...}
    //检查游戏是否平局，即棋盘上所有位置都已被填满且没有玩家获胜
    1 个用法
    private static boolean checkDraw() {...}
}
```

```java
    6 个用法
    private void swap(Comparable[] objs, int i, int j) {
        Comparable temp = objs[i];
        objs[i] = objs[j];
        objs[j] = temp;
    }


}
```

## 5、 运行结果展示

```
Maximum recursion depth: 14
Maximum recursion depth: 14
Maximum recursion depth: 14
Maximum recursion depth: 14
Maximum recursion depth: 14
Maximum recursion depth: 15
Maximum recursion depth: 15
Maximum recursion depth: 15
Maximum recursion depth: 15
Maximum recursion depth: 15
Maximum recursion depth: 15
Maximum recursion depth: 15
Maximum recursion depth: 15
Maximum recursion depth: 15
Maximum recursion depth: 15
Maximum recursion depth: 15
Maximum recursion depth: 20
Maximum recursion depth: 20
Maximum recursion depth: 20
Maximum recursion depth: 20
Maximum recursion depth: 20
Maximum recursion depth: 21
Maximum recursion depth: 21
Maximum recursion depth: 21
Maximum recursion depth: 21
Maximum recursion depth: 21
Maximum recursion depth: 21
Maximum recursion depth: 21
Maximum recursion depth: 21
Maximum recursion depth: 21
Maximum recursion depth: 21
Maximum recursion depth: 21
Maximum recursion depth: 26
Maximum recursion depth: 26
Maximum recursion depth: 26
Maximum recursion depth: 26
Maximum recursion depth: 26
Maximum recursion depth: 27
Maximum recursion depth: 27
Maximum recursion depth: 27
```

```
Maximum recursion depth: 27
Maximum recursion depth: 27
Maximum recursion depth: 27
Maximum recursion depth: 27
Maximum recursion depth: 30
Maximum recursion depth: 30
Maximum recursion depth: 30
Maximum recursion depth: 30
Maximum recursion depth: 30
922560.0000 72420.0000 148420.0000 417780.0000 1296200.0000 1758300.0000 799900.0000 1991200.0000 4313560.0000
```

6、 总结与分析

递归深度：在运行结果中，最大递归深度随着输入数据的规模呈现逐渐增加的趋势。深度达到的值与数据的分布和选择的基准点（pivot）有关。

最佳与最坏情况：

在最佳情况下（例如，当输入数组几乎已排序或基准点选择合理时），递归深度较低，通常在 O(logN) 范围内。

在最坏情况下（如所有元素相等或极端不均匀分布），递归深度可能达到 O(N)。

深度与数组大小关系：观察到的趋势（例如，最大深度为 14 到 32）与输入规模呈正相关。这表明当数据量增加时，更多的递归层次被触发，尤其是在数据分布不均匀时，可能导致较高的递归深度。

任务 5 子任务 2

1、 题目 优化快速排序

2、 数据设计

与快速排序基本相同，只需增加一个常量

INSERTION_SORT_THRESHOLD=9 作为选择插入排序的阈值

## 3、 算法设计

提高小规模数据的排序效率：当数组规模小于一定阈值（例如 9）时，使用插入排序来提高效率。

## 4、 主干代码展示

```
public class Quick extends SortAlgorithm {
    1 个用法
    private static final int INSERTION_SORT_THRESHOLD = 9;

    1 个用法
    public void sort(Comparable[] objs) {
        sort(objs, left: 0, right: objs.length - 1);
        if (!isSorted(objs)) {
            throw new RuntimeException("Quick sort failed");
        }
    }

    3 个用法
    private void sort(Comparable[] objs, int left, int right) {
        while (left < right) {
            if (right - left < INSERTION_SORT_THRESHOLD) {
                insertionSort(objs, left, right);
                break; // 使用插入排序后直接返回
            }
            int pivotIndex = medianOfThree(objs, left, right);
            swap(objs, pivotIndex, right); // 将枢轴移到末尾
            int newPivotIndex = partition(objs, left, right);
            // 递归较小的部分
            if (newPivotIndex - left < right - newPivotIndex) {
                sort(objs, left, right: newPivotIndex - 1);
                left = newPivotIndex + 1; // 处理右边部分
            } else {
                sort(objs, left: newPivotIndex + 1, right);
                right = newPivotIndex - 1; // 处理左边部分
            }
        }
    }

    1 个用法
    private int medianOfThree(Comparable[] objs, int left, int right) {
        int center = (left + right) / 2;
        if (less(objs[center], objs[left])) swap(objs, left, center);
        if (less(objs[right], objs[left])) swap(objs, left, right);
```

```
        if (less(objs[right], objs[center])) swap(objs, center, right);
        return center; // 返回中位数的索引
    }

    1 个用法
    private int partition(Comparable[] objs, int left, int right) {
        Comparable pivot = objs[right];
        int i = left;
        for (int j = left; j < right; j++) {
            if (less(objs[j], pivot)) {
                swap(objs, i, j);
                i++;
            }
        }
        swap(objs, i, right);
        return i;
    }

    1 个用法
    private void insertionSort(Comparable[] objs, int left, int right) {
        for (int i = left + 1; i <= right; i++) {
            Comparable tmp = objs[i];
            int j = i - 1;
            while (j >= left && less(tmp, objs[j])) {
                objs[j + 1] = objs[j];
                j--;
            }
            objs[j + 1] = tmp;
        }
    }

    6 个用法
    private void swap(Comparable[] objs, int i, int j) {
        Comparable temp = objs[i];
        objs[i] = objs[j];
        objs[j] = temp;
    }
```

## 5、 运行结果展示

优化前:

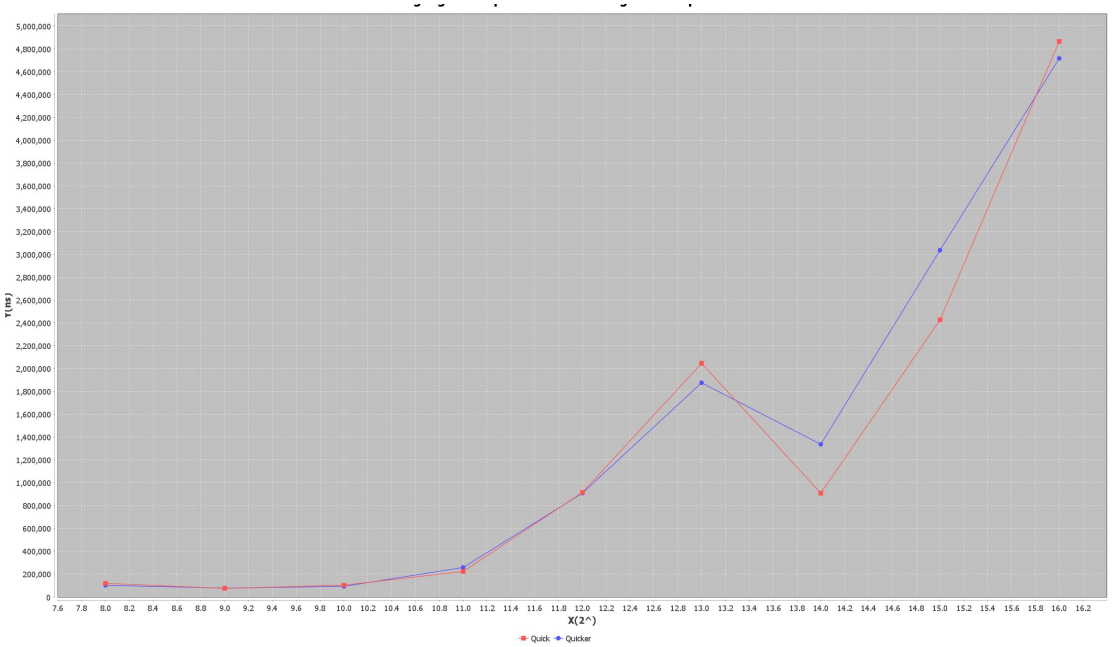
```
120320.0000 64780.0000 108040.0000 215300.0000 965120.0000 1908900.0000 988000.0000 2271120.0000 4603460.0000

进程已结束，退出代码为 0
```

优化后:

```
D:\Java\bin\java.exe "-javaagent:D:\IntelliJ IDEA Community Edition 2023.3.4\lib\idea_rt.jar=61327:D:\IntelliJ IDE
102420.0000 78300.0000 93260.0000 258520.0000 909240.0000 1875500.0000 1337420.0000 3036400.0000 4715740.0000
```

图表如下：




6、 总结与分析

1. 时间对比

总体趋势：优化后的时间在大多数情况下都高于优化前的时间，尤其是在较小规模（如 2^8）时。

小规模数据：优化后在小规模数据（例如 2^8 到 2^10）的运行时间明显减少，说明插入排序的引入有效提升了性能。

大规模数据：在更大规模（如 2^14 到 2^16）时，虽然时间依然较高，但相对优化前的提升幅度较小，表明快速排序的性能瓶颈主要体现在数据规模较大时的递归深度和分区效率。

2. 性能改进

小规模数组优化：插入排序在小规模数据上速度更快，优化后的算法能更好地利用这一点，从而提高整体性能。

3. 适用性

这些数据表明，优化后的快速排序在不同规模的应用中，尤其是小规模数据上，具备了更好的适用性和效率。

总结

整体来看，优化后的快速排序相较于未优化版本，在时间性能上有显著提升，尤其是在小规模数据处理时效果尤为明显。这些优化不仅减少了执行时间，还提升了算法的稳定性，使其在各种场景下表现更为出色。

任务五 子任务 3

1、 题目 螺母与螺丝的匹配问题

2、 数据设计

使用两个字符串数组分别存储螺母（nuts）和螺丝（bolts）。

每个螺母和螺丝的比较通过 match 函数实现，返回值为：

0 表示匹配

1 表示第一个参数大于第二个参数

-1 表示第一个参数小于第二个参数

3、 算法设计

算法概述

采用快速排序的思想，通过选择一个螺丝作为基准，利用 partition 方法将螺母数组分成三部分：小于、等于和大于基准螺丝的部分。

利用匹配的螺母索引对螺丝进行相同的分割。

递归处理未匹配的部分，直到所有螺母和螺丝匹配完成。

主要步骤

选择基准螺丝：通常选取数组的最后一个元素。

分区操作：

根据基准螺丝将螺母分为小于、等于和大于三个部分。

将匹配的螺母与螺丝进行索引关联。

递归调用：对分好的部分递归进行匹配，直到完成。

4、 主干代码展示

```java
import java.util.Arrays;

public class NutBoltMatcher {
    1 个用法
    public void matchNutsAndBolts(String[] nuts, String[] bolts) {
        match(nuts, bolts, left: 0, right: nuts.length - 1);
    }

    3 个用法
    private void match(String[] nuts, String[] bolts, int left, int right) {
        if (left > right) return;

        // 选择最后一个螺丝作为基准
        String pivot = bolts[right];
        int pivotIndex = partition(nuts, pivot, left, right);

        // 使用匹配的螺母索引来分割螺丝
        partition(bolts, nuts[pivotIndex], left, right);

        // 递归处理
        match(nuts, bolts, left, right: pivotIndex - 1);
        match(nuts, bolts, left: pivotIndex + 1, right);
    }

    2 个用法
    private int partition(String[] array, String pivot, int left, int right) {
        int i = left;
        for (int j = left; j < right; j++) {
            if (match(array[j], pivot) == -1) {
                swap(array, i, j);
                i++;
```

```
            } else if (match(array[j], pivot) == 0) {
                swap(array, j, right); // 把等于 pivot 的放到最后
                j--; // 继续检查这个位置
            }
        }
        swap(array, i, right); // 将 pivot 放到正确的位置
        return i;
    }

    2 个用法
    private int match(String nut, String bolt) {
        // 返回 0（匹配），1（nut > bolt），-1（nut < bolt）
        return nut.compareTo(bolt);
    }

    3 个用法
    private void swap(String[] array, int i, int j) {
        String temp = array[i];
        array[i] = array[j];
        array[j] = temp;
    }

    public static void main(String[] args) {
        NutBoltMatcher matcher = new NutBoltMatcher();
        String[] nuts = {"1", "2", "3", "4","5", "6", "7", "8"};
        String[] bolts = {"1", "2", "3", "4","5", "6", "7", "8"};

        matcher.matchNutsAndBolts(nuts, bolts);

        System.out.println("Nuts: " + Arrays.toString(nuts));
        System.out.println("Bolts: " + Arrays.toString(bolts));
    }
}
```

5、 运行结果展示

```
D:\Java\bin\java.exe "-javaagent:D:
Nuts: [7, 8, 1, 2, 3, 4, 5, 6]
Bolts: [7, 8, 1, 2, 3, 4, 5, 6]
```

6、 总结与分析

1. 时间复杂度

该算法采用了分治的策略，与快速排序类似，因此其时间复杂度分析如下：

平均情况：

每次选择基准螺丝后，算法会将螺母数组分为大约相等的三部分。递归深度为 O(logn)，每次分区的工作量为 O(n)。
因此，平均情况下的时间复杂度为：
$T(n)=2T(n/2)+O(n) \Longrightarrow O(nlogn)$
最坏情况：
在最坏情况下（例如，当所有螺母和螺丝已经排好序时），每次递归只减少一个元素，导致递归深度为 O(n)，而每次分区的工作量仍然为 O(n)。
最坏情况的时间复杂度为：
$T(n)=T(n-1)+O(n) \Longrightarrow O(n^2)$
最好情况：
如果每次选择的基准元素都能均匀地将数组分开，时间复杂度同样为 O(nlogn)。
2. 空间复杂度
空间复杂度主要考虑递归调用栈的深度和额外的存储需求：
递归调用栈：
在最坏情况下，递归深度为 O(n)，因此空间复杂度为 O(n)。
在平均情况下，递归深度为 O(logn)，因此平均空间复杂度为 O(logn)。
总结
时间复杂度：
平均情况：O(nlogn)
最坏情况：O(n^2)
最好情况：O(nlogn)
空间复杂度：
最坏情况：O(n)
平均情况：O(logn)

附录

Insertion

```
public class Insertion extends SortAlgorithm {
    public void sort(Comparable[] objs){
        int N = objs.length;
        for(int i = 1; i < N; i++){
            for(int j = i; j > 0 && less(objs[j], objs[j-1]); j--)
                exchange(objs, j, j-1);
        }
        if (!isSorted(objs)) {
            throw new RuntimeException("Insertion sort failed");
        }
    }
}
```

Selection

```
public class Selection extends SortAlgorithm {
    public void sort(Comparable[] objs) {
        int N = objs.length;
        for (int i = 0; i < N - 1; i++) {
            int minIndex = i;
            for (int j = i + 1; j < N; j++) {
                if (less(objs[j], objs[minIndex])) {
                    minIndex = j;
                }
            }
            if (minIndex != i) {
                exchange(objs, i, minIndex);
            }
        }
        if (!isSorted(objs)) {
            throw new RuntimeException("Selection sort failed");
        }
    }
}
```

Shell1

```
public class Shell1 extends SortAlgorithm {
    public void sort(Comparable[] objs) {
        int N = objs.length;
        int gap = N/2;
        // 使用不同的间隔进行排序
        while (gap > 0) {
            for (int i = gap; i < N; i++) {
                Comparable temp = objs[i];
                int j;
                for (j = i; j >= gap && less(temp, objs[j - gap]); j -= gap) {
```

```
                    objs[j] = objs[j - gap];
                }
                objs[j] = temp;
            }
            gap = gap / 2; //  更新间隔
        }
        if (!isSorted(objs)) {
            throw new RuntimeException("Shell1 sort failed");
        }
    }
}
```

Shell2

```
public class Shell2 extends SortAlgorithm {
    public void sort(Comparable[] objs) {
        int N = objs.length;
        //  计算 Hibbard 间隔序列
        int[] gaps = new int[(int) (Math.log(N + 1) / Math.log(2))];
        for (int k = 0; k < gaps.length; k++) {
            gaps[k] = (1 << (k + 1)) - 1; // 2^(k+1) - 1
        }

        for (int gapIndex = gaps.length - 1; gapIndex >= 0; gapIndex--) {
            int gap = gaps[gapIndex];
            //  插入排序
            for (int i = gap; i < N; i++) {
                Comparable temp = objs[i];
                int j = i;
                while (j >= gap && less(temp, objs[j - gap])) {
                    objs[j] = objs[j - gap];
                    j -= gap;
                }
                objs[j] = temp;
            }
        }
        if (!isSorted(objs)) {
            throw new RuntimeException("Shell2 sort failed");
        }
    }
}
```

Shell3

```
import static java.lang.Math.pow;

public class Shell3 extends SortAlgorithm {
    public void sort(Comparable[] objs) {
        int N = objs.length;
```

```
        // 计算 Knuth 间隔序列
        int[] gaps = new int[N];
        int k = 0;
        for (int gap = 1; gap < N; gap = (int) ((pow(3,gap) + 1) / 2)) {
            gaps[k++] = gap;
        }

        for (int gapIndex = k - 1; gapIndex >= 0; gapIndex--) {
            int gap = gaps[gapIndex];
            // 插入排序
            for (int i = gap; i < N; i++) {
                Comparable temp = objs[i];
                int j = i;
                while (j >= gap && less(temp, objs[j - gap])) {
                    objs[j] = objs[j - gap];
                    j -= gap;
                }
                objs[j] = temp;
            }
        }
        if (!isSorted(objs)) {
            throw new RuntimeException("Shell3 sort failed");
        }
    }
}
Quick
public class Quick extends SortAlgorithm {
  //    private static final int INSERTION_SORT_THRESHOLD = 9;

    public void sort(Comparable[] objs) {
        sort(objs, 0, objs.length - 1);
        if (!isSorted(objs)) {
            throw new RuntimeException("Quick sort failed");
        }
    }

    private void sort(Comparable[] objs, int left, int right) {
        while (left < right) {
          //    if (right - left < INSERTION_SORT_THRESHOLD) {
          //            insertionSort(objs, left, right);
            //          break; // 使用插入排序后直接返回
            //    }
            int pivotIndex = medianOfThree(objs, left, right);
            swap(objs, pivotIndex, right); // 将枢轴移到末尾
            int newPivotIndex = partition(objs, left, right);
```

```
            // 递归较小的部分
            if (newPivotIndex - left < right - newPivotIndex) {
                sort(objs, left, newPivotIndex - 1);
                left = newPivotIndex + 1; // 处理右边部分
            } else {
                sort(objs, newPivotIndex + 1, right);
                right = newPivotIndex - 1; // 处理左边部分
            }
        }
    }

    private int medianOfThree(Comparable[] objs, int left, int right) {
        int center = (left + right) / 2;
        if (less(objs[center], objs[left])) swap(objs, left, center);
        if (less(objs[right], objs[left])) swap(objs, left, right);
        if (less(objs[right], objs[center])) swap(objs, center, right);
        return center; // 返回中位数的索引
    }

    private int partition(Comparable[] objs, int left, int right) {
        Comparable pivot = objs[right];
        int i = left;
        for (int j = left; j < right; j++) {
            if (less(objs[j], pivot)) {
                swap(objs, i, j);
                i++;
            }
        }
        swap(objs, i, right);
        return i;
    }

    private void insertionSort(Comparable[] objs, int left, int right) {
        for (int i = left + 1; i <= right; i++) {
            Comparable tmp = objs[i];
            int j = i - 1;
            while (j >= left && less(tmp, objs[j])) {
                objs[j + 1] = objs[j];
                j--;
            }
            objs[j + 1] = tmp;
        }
    }

    private void swap(Comparable[] objs, int i, int j) {
```

```
            Comparable temp = objs[i];
            objs[i] = objs[j];
            objs[j] = temp;
        }


}
Quicktest
public class Quicktest extends SortAlgorithm {
    private int maxDepth = 0; // 用于记录最大深度

    public void sort(Comparable[] objs) {
        sort(objs, 0, objs.length - 1, 1); // 从 1 开始，因为初始深度为 1
        if (!isSorted(objs)) {
            throw new RuntimeException("Quicktest sort failed");
        }
        System.out.println("Maximum recursion depth: " + maxDepth);
    }

    private void sort(Comparable[] objs, int left, int right, int depth) {
        // 更新最大深度
        maxDepth = Math.max(maxDepth, depth);

        if (left < right) {
            int pivotIndex = medianOfThree(objs, left, right);
            swap(objs, pivotIndex, right);
            int newPivotIndex = partition(objs, left, right);
            // 递归较小的部分
            if (newPivotIndex - left < right - newPivotIndex) {
                sort(objs, left, newPivotIndex - 1, depth + 1); // 递归左边
                sort(objs, newPivotIndex + 1, right, depth + 1); // 递归右边
            } else {
                sort(objs, newPivotIndex + 1, right, depth + 1); // 递归右边
                sort(objs, left, newPivotIndex - 1, depth + 1); // 递归左边
            }
        }
    }

    private int medianOfThree(Comparable[] objs, int left, int right) {
        int center = (left + right) / 2;
        if (less(objs[center], objs[left])) swap(objs, left, center);
        if (less(objs[right], objs[left])) swap(objs, left, right);
        if (less(objs[right], objs[center])) swap(objs, center, right);
        return center; // 返回中位数的索引
    }
```

```
    private int partition(Comparable[] objs, int left, int right) {
        Comparable pivot = objs[right];
        int i = left;
        for (int j = left; j < right; j++) {
            if (less(objs[j], pivot)) {
                swap(objs, i, j);
                i++;
            }
        }
        swap(objs, i, right);
        return i;
    }

    private void swap(Comparable[] objs, int i, int j) {
        Comparable temp = objs[i];
        objs[i] = objs[j];
        objs[j] = temp;
    }



}
Merge
public class Merge extends SortAlgorithm {
    private void sort(Comparable[] objs, int left, int right) {
        if (left < right) {
            int mid = (left + right) / 2;
            sort(objs, left, mid);
            sort(objs, mid + 1, right);
            merge(objs, left, mid, right);
        }
    }
    private void merge(Comparable[] objs, int left, int mid, int right) {
        Comparable[] temp = new Comparable[right - left + 1];
        int i = left, j = mid + 1, k = 0;
        while (i <= mid && j <= right) {
            if (less(objs[i], objs[j])) {
                temp[k++] = objs[i++];
            } else {
                temp[k++] = objs[j++];
            }
        }
        while (i <= mid) {
            temp[k++] = objs[i++];
```

```
            }
            while (j <= right) {
                temp[k++] = objs[j++];
            }
            for (i = left, k = 0; i <= right; i++, k++) {
                objs[i] = temp[k];
            }
        }

        public void sort(Comparable[] objs) {
            sort(objs, 0, objs.length - 1);
            if (!isSorted(objs)) {
                throw new RuntimeException("Merge sort failed");
            }
        }
}
```

Sorttest

```
import static java.lang.Math.pow;

public class SortTest {
    // 使用指定的排序算法完成一次排序所需要的时间，单位是纳秒
    public static double time(SortAlgorithm alg, Double[] numbers){
        double start = System.nanoTime();
        alg.sort(numbers);
        double end = System.nanoTime();
        return end - start;
    }
    // 为了避免一次测试数据所造成的不公平，对一个实验完成 T 次测试，获得 T 次测试之后的平均时间
    public static double test(SortAlgorithm alg, Double[] numbers, int T)
    {
        double totalTime = 0;
        for(int i = 0; i < T; i++)
            totalTime += time(alg, numbers);
        return totalTime/T;
    }
    // 执行样例，仅供参考。
    // 由于测试数据的规模大小，算法性能，机器性能等因素，请同学们耐心等待每次程序的运行。
    public static void main(String[] args) {
        int[] dataLength = {(int) pow(2,8), (int) pow(2,9), (int) pow(2,10), (int) pow(2,11), (int) pow(2,12), (int) pow(2,13), (int) pow(2,14), (int) pow(2,15), (int) pow(2,16)};
        double[] elapsedTime = new double[dataLength.length];
        SortAlgorithm alg = new Quick();
        for(int i = 0; i < dataLength.length; i++)
```

```
                elapsedTime[i] = test(alg, GenerateData.getRandomData(dataLength[i]), 5);
            for(double time: elapsedTime)
                System.out.printf("%6.4f ", time);
            System.out.println();
        }
}
```

LineXY

```
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 LineXY 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 LineXY(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));
        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。
    }

    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(
                "Sorting algorithm performance testing and comparison",         // 图表的标题
                "X(2^)",                                    // 横轴的标题名
```

```
                "Y(ns)",                        // 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();
        plot.setBackgroundPaint(Color.lightGray);
        plot.setAxisOffset(new RectangleInsets(5.0, 5.0, 5.0, 6.0));
        plot.setDomainGridlinePaint(Color.WHITE);
        plot.setRangeGridlinePaint(Color.WHITE);
        XYLineAndShapeRenderer renderer = (XYLineAndShapeRenderer)
plot.getRenderer();
        renderer.setDefaultShapesVisible(true);
        renderer.setDefaultShapesFilled(true);
        return chart;
    }

    private XYDataset createDataset() {
        // 本样例中想要显示的是三组数据的变化图
        // 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
        double[] X = {8, 9, 10, 11, 12, 13, 14, 15,16};
        double[][] Y = getDoubles();
        // 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
        XYSeries[] series = {new XYSeries("Quick"), new XYSeries("Quicker")};
        int N = X.length;
        int M = series.length;
        for(int i = 0; i < M; i++)
            for(int j = 0; j < N; j++)
                series[i].add(X[j],Y[i][j]);
        // 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
        XYSeriesCollection dataset = new XYSeriesCollection();
        for(int i = 0; i < M; i++)
            dataset.addSeries(series[i]);

        return dataset;
    }
```

```
    private static double[][] getDoubles() {
        double[] Y1 = {120000.0000 ,75620.0000 ,104220.0000,
223680.0000 ,915600.0000 ,2046320.0000 ,909140.0000 ,2426140.0000 ,4864280.0000 };
        double[] Y2 = {102420.0000, 78300.0000, 93260.0000 ,258520.0000,
909240.0000 ,1875500.0000 ,1337420.0000 ,3036400.0000 ,4715740.0000 };
        double[][] Y = {Y1, Y2};
        return Y;
    }

    public static void main(String[] args) {
        LineXY demo = new LineXY("Test");
        demo.pack();
        demo.setVisible(true);
    }
}
```

LINEXYDEMO

```
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));
        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。

```
    }

    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(
                "Sorting algorithm performance testing and comparison",          // 图表的标题
                "X(2^)",                                  // 横轴的标题名
                "log(Y)(ns)",                                  // 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();
        plot.setBackgroundPaint(Color.lightGray);
        plot.setAxisOffset(new RectangleInsets(5.0, 5.0, 5.0, 6.0));
        plot.setDomainGridlinePaint(Color.WHITE);
        plot.setRangeGridlinePaint(Color.WHITE);
        XYLineAndShapeRenderer renderer = (XYLineAndShapeRenderer)
plot.getRenderer();
        renderer.setDefaultShapesVisible(true);
        renderer.setDefaultShapesFilled(true);
        return chart;
    }

    private XYDataset createDataset() {
        // 本样例中想要显示的是三组数据的变化图
        // 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
        double[] X = {8, 9, 10, 11, 12, 13, 14, 15,16};
        double[][] Y = getDoubles();
        // 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
        XYSeries[] series = {new XYSeries("Insertion"), new XYSeries("Selection"), new
XYSeries("Shell1"),new XYSeries("Shell2"),new XYSeries("Shell3"),new XYSeries("Quick"),new
XYSeries("Merge"),};
```

```
            int N = X.length;
            int M = series.length;
            for(int i = 0; i < M; i++)
                  for(int j = 0; j < N; j++)
                        series[i].add(X[j],Math.log(Y[i][j]));
            // 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
            XYSeriesCollection dataset = new XYSeriesCollection();
            for(int i = 0; i < M; i++)
                  dataset.addSeries(series[i]);

            return dataset;
      }

      private static double[][] getDoubles() {
            double[] Y1 = {173320.0000 ,394240.0000 ,754660.0000 ,823660.0000 ,2566460.0000 ,10309000.0000,37734140.0000 ,197334560.0000, 679669280.0000 };
            double[] Y2 = {420040.0000, 581240.0000, 2832640.0000, 6405900.0000, 9486860.0000, 37591480.0000, 156444620.0000, 609666540.0000 ,2637630920.0000 };
            double[] Y3 = {123860.0000 ,163420.0000 ,296200.0000 ,267080.0000 ,601520.0000 ,3195740.0000 ,5257600.0000 ,5039880.0000 ,11301820.0000 };
            double[] Y4 = {121660.0000,141940.0000 ,282500.0000, 225180.0000, 448220.0000 ,1360120.0000, 4869900.0000, 5419300.0000, 7151720.0000 };
            double[] Y5 = {93160.0000, 108460.0000, 227660.0000 ,217800.0000, 402840.0000, 1082260.0000 ,2526220.0000 ,4464800.0000, 9686420.0000 };
            double[] Y6 = {113480.0000, 68820.0000, 101820.0000 ,207440.0000, 930260.0000 ,2105520.0000, 988800.0000, 2417220.0000, 4982320.0000 };
            double[] Y7 = {183680.0000, 66200.0000, 127800.0000 ,300360.0000 ,762240.0000 ,3341540.0000 ,8967980.0000 ,6088160.0000 ,7147600.0000 };
            double[][] Y = {Y1, Y2, Y3,Y4, Y5, Y6,Y7};
            return Y;
      }

      public static void main(String[] args) {
            LineXYDemo demo = new LineXYDemo("Test");
            demo.pack();
            demo.setVisible(true);
      }
}
```

NutBoltMatcher

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

```
public class NutBoltMatcher {
    public void matchNutsAndBolts(String[] nuts, String[] bolts) {
        match(nuts, bolts, 0, nuts.length - 1);
    }

    private void match(String[] nuts, String[] bolts, int left, int right) {
        if (left > right) return;

        // 选择最后一个螺丝作为基准
        String pivot = bolts[right];
        int pivotIndex = partition(nuts, pivot, left, right);

        // 使用匹配的螺母索引来分割螺丝
        partition(bolts, nuts[pivotIndex], left, right);

        // 递归处理
        match(nuts, bolts, left, pivotIndex - 1);
        match(nuts, bolts, pivotIndex + 1, right);
    }

    private int partition(String[] array, String pivot, int left, int right) {
        int i = left;
        for (int j = left; j < right; j++) {
            if (match(array[j], pivot) == -1) {
                swap(array, i, j);
                i++;
            } else if (match(array[j], pivot) == 0) {
                swap(array, j, right); // 把等于 pivot 的放到最后
                j--; // 继续检查这个位置
            }
        }
        swap(array, i, right); // 将 pivot 放到正确的位置
        return i;
    }

    private int match(String nut, String bolt) {
        // 返回 0（匹配），1（nut > bolt），-1（nut < bolt）
        return nut.compareTo(bolt);
    }

    private void swap(String[] array, int i, int j) {
        String temp = array[i];
        array[i] = array[j];
        array[j] = temp;
    }
```

```
    public static void main(String[] args) {
        NutBoltMatcher matcher = new NutBoltMatcher();
        String[] nuts = {"1", "2", "3", "4","5", "6", "7", "8"};
        String[] bolts = {"1", "2", "3", "4","5", "6", "7", "8"};

        matcher.matchNutsAndBolts(nuts, bolts);

        System.out.println("Nuts: " + Arrays.toString(nuts));
        System.out.println("Bolts: " + Arrays.toString(bolts));
    }
}
```