|  |  |
| --- | --- |
| **数据结构与算法 作业报告** | |
| 第 一 次 | |
| 图片包含 标牌  已生成极高可信度的说明 | |
|  | |
|  | |
|  | |
| **姓名** | 曹家豪 |
| **班级** | 软件2204班 |
| **学号** | 2226114017 |
| **电话** | 13572763245 |
| **Email** | caojiahao@stu.xjtu.edu.cn |
| **日期** | 2023-11-27 |

目录

任务一：证明

1.证明下列等式

2.使用数学归纳法证明等式

任务二：

1.题目：设计算法要多思考

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。

2.数据与算法设计

1）首先分析问题，需要达到O(mlogn)的时间复杂度，m，n分别为图片的列、行数，则可考虑一种算法，对每一列进行遍历，计算每一列的长度并比较得出最大值，遍历所有行的时间复杂度为O(m),那么当计算每列长度的算法满足时间复杂度为O(logn)时即可满足题意。

2）要使计算每列长度的算法满足时间复杂度为O(logn)，考虑采用二分法。

3）这里定义了*getLength(int i,Picture sample)*方法以获取sample图像第i列的长度，其实现逻辑如下：

因为两种颜色时严格分开的，那么只需找到分界点位置即可。定义两个指针left与right指向当前的边界，定义中间指针pointer指向left与right中间的位置，当pointer与pointer+1位置的颜色相同（需要在Picture类中定义isSameAsNext方法）时：

①如果pointer位置与首位置的颜色相同，说明分界点在右半部分，更新左指针，使left=pointer+1；

②反之说明分界点在左半部分，更新右指针，令right=pointer-1；

直到pointer到达边界或pointer与pointer+1位置颜色不同时终止，此时pointer即为两种颜色的分界点（或者纯色列的边界），此时只需要根据首位值的颜色判断最终长度是前半部分还是后半部分长度即可。

4）在主函数中定义*curlonggest*为当前的最长列长，初始化为0，然后开始遍历每一列，调用*getLength*方法计算当前列长，当结果大于*curlonggest*时，更新*curlonggest*的值。遍历结束后，*curlonggest*则为图片的最大列长度。

3.部分代码说明

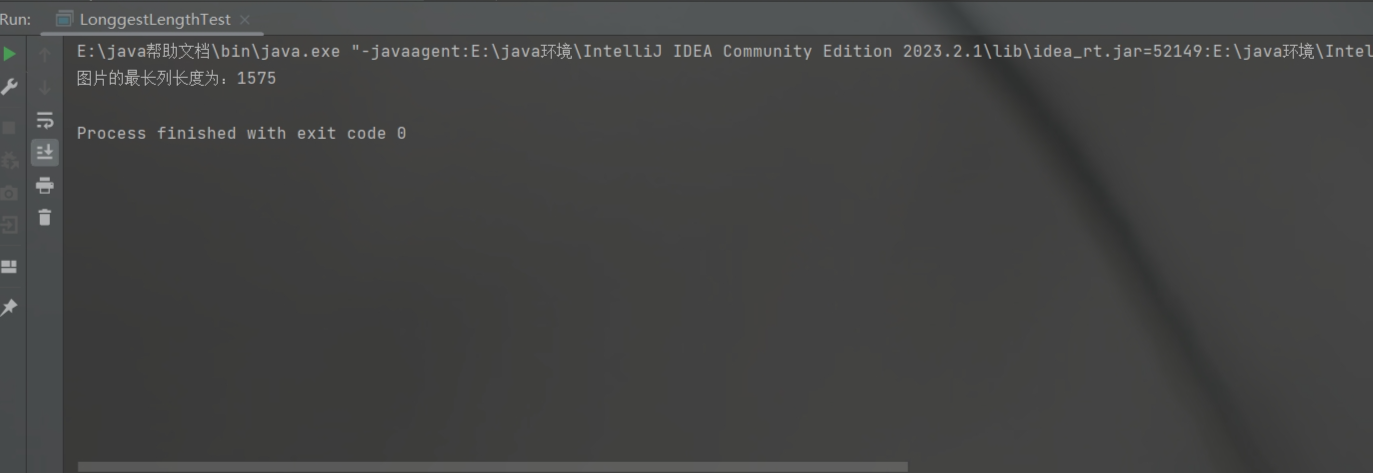
主函数：

1. int curlongest=0;
2. *//定义当前最大列长*
3. try {
4. Picture testimage = new Picture(testpicture);*//读入图片*
5. for(int i=0;i<testimage.getWidth();i++){
6. *//遍历每一列*
7. int curlength=getLength(i,testimage);
8. *//获取当前列长*
9. if(curlength>=curlongest){
10. curlongest=curlength;
11. *//若当前列长更大，则更新curlonggest*
12. }
13. }
14. System.out.println("图片的最长列长度为："+curlongest);
15. } catch (IOException e) {
16. System.out.println("图片读取失败");
17. }
18. }

getLength方法：

1. public static int getLength(int i, Picture sample) {
2. Color firstcolor = sample.getColor(i, 0);
3. *//获取首位值颜色*
4. int left = 0;
5. int right = sample.getHeight() - 1;
6. int pointer = (left + right) / 2;
7. *//定义头、中、尾指针*
8. while ((pointer < (sample.getHeight() - 1))&&sample.isSameAsNext(i, pointer))
9. *//开始二分，不断循环*
10. *//终止条件：pointer到达分界点或边界*
11. {*//pointer与下一个颜色相同时*
12. if (sample.isSame(i, 0, i, pointer))
13. *//如果pointer位置与首位置的颜色相同，说明分界点在右半部分，更新左指针*
14. {left = pointer + 1;}
15. else{
16. *//反之说明分界点在左半部分，更新右指针*
17. right = pointer - 1;
18. }
19. pointer = (left + right) / 2;
20. *//更新中间指针*
21. }
22. if (pointer == (sample.getHeight() - 1)) {
23. *//根据首位值颜色判断返回长度*
24. if (firstcolor.equals(WHITE)) {
25. return 0;
26. } else {
27. return sample.getHeight();
28. }}
29. if (firstcolor.equals(BLUE)) {
30. return pointer + 1;
31. }
32. else {
33. return sample.getHeight() - (pointer + 1);
34. }
35. }

4.运行结果：



5.总结收获：

1）对算法的时间复杂度有了更深的理解，能够根据给定的时间复杂度判读采用何种算法

2）更熟练地掌握与运用二分思想解题

3）解决问题过程中有许多对边界值的判断，一定程度上提高了处理边界的能力

任务三

1.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： ℎ1 = 1 ℎ𝑖 = ℎ𝑖−1 ∗ 3

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。

2.数据与算法设计：

1）主要工作原理：

①选择排序：开启一个循环，每轮从未排序区间选择最小的元素，将其放到已排序区间的末尾缩小未排序区间范围。

②希尔排序：选择一个间隔序列（本题中为：N/3, N/9, N/27...1）。按照间隔序列将原始数组分成若干个子数组。对每个子数组进行插入排序。逐步缩小间隔序列，重复前两个步骤，直到最后一次以间隔为1的插入排序完成排序。

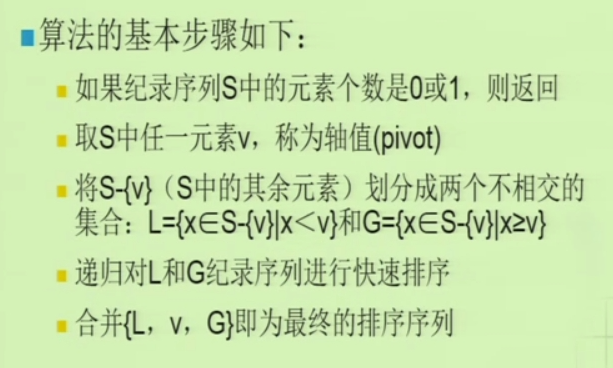
③快速排序：选择数组中的某个元素作为“轴值”，将所有小于轴值的元素移到其左侧，而大于轴值的元素移到其右侧。对左侧与右侧序列再次进行这一过程，直至序列无法分割。

④归并排序：归并排序包括“划分“和”归并“两个阶段，其中划分阶段通过递归不断地将数组从中点处分开，将长数组的排序问题转换为短数组的排序问题。合并阶段当子数组长度为 1 时终止划分，开始合并，持续地将左右两个较短的有序数组合并为一个较长的有序数组，直至结束。

2）在该任务中，Insertion、Selection、Shell、Quicksort、MergeSort都是虚拟类SortAlgorithm的子类，因而要实现SortAlgorithm中的sort方法，具体实现借助上面提到的工作原理。对于选择排序与希尔排序不再过多解释，主要阐述该任务中对快速排序与归并排序的实现：

3）快速排序的实现

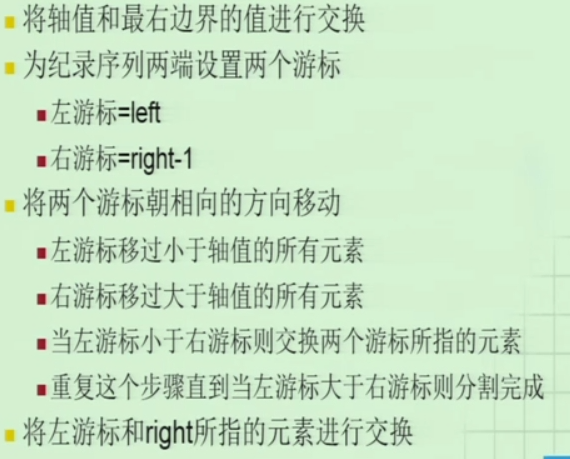
①主函数逻辑如下所示



②轴值选择：

该任务中选取首位置，中间位置与末位置元素的中间值为轴值，再完成轴值选取的过程中同时完成对这三个位置数据的排序

③划分阶段逻辑如下所示



4）归并排序的实现：

①当序列长度大于1时，持续递归划分左半序列与右半序列，划分结束后，实现归并操作

②归并的实现采用双指针，分别指向左半序列和右半序列的起始位置，开辟一个新序列，开始遍历，比较两个指针指向的值大小，将较小值放入新序列中并更新指针，其中一个指针到达末尾后停止，将另一个序列的剩余元素依次放入新序列，最后将新序列拷贝至原序列。

5）主函数：

在主函数中利用GenerateData类产生规模为20和10000的各五个随机序列，定义五个SortAlgorithm类对象分别指向五种算法，利用类中的sort方法对随机序列排序，利用isSorted方法对排序后的数据进行测试，展示对规模为20的排序结果和两种规模的测试结果。

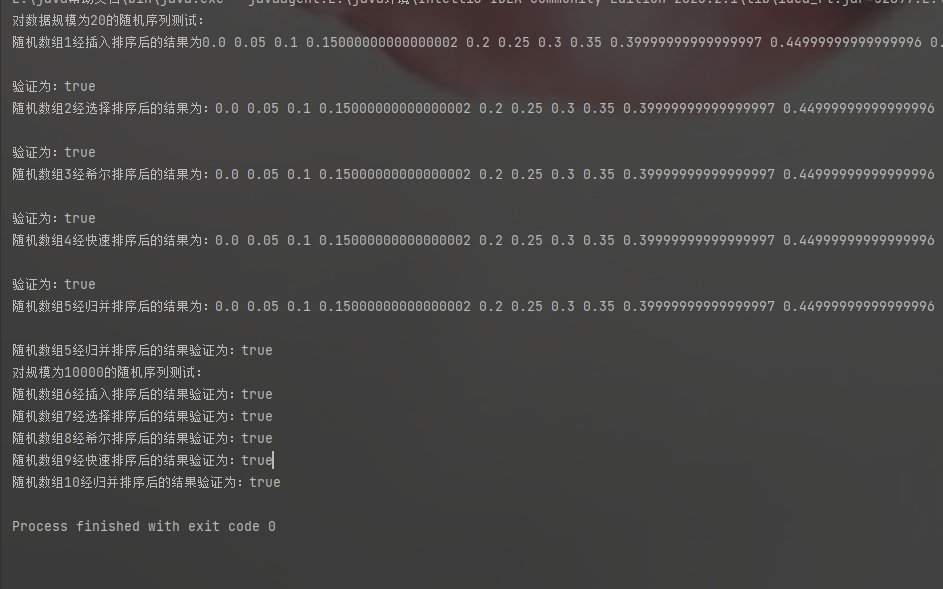
3.部分代码说明：

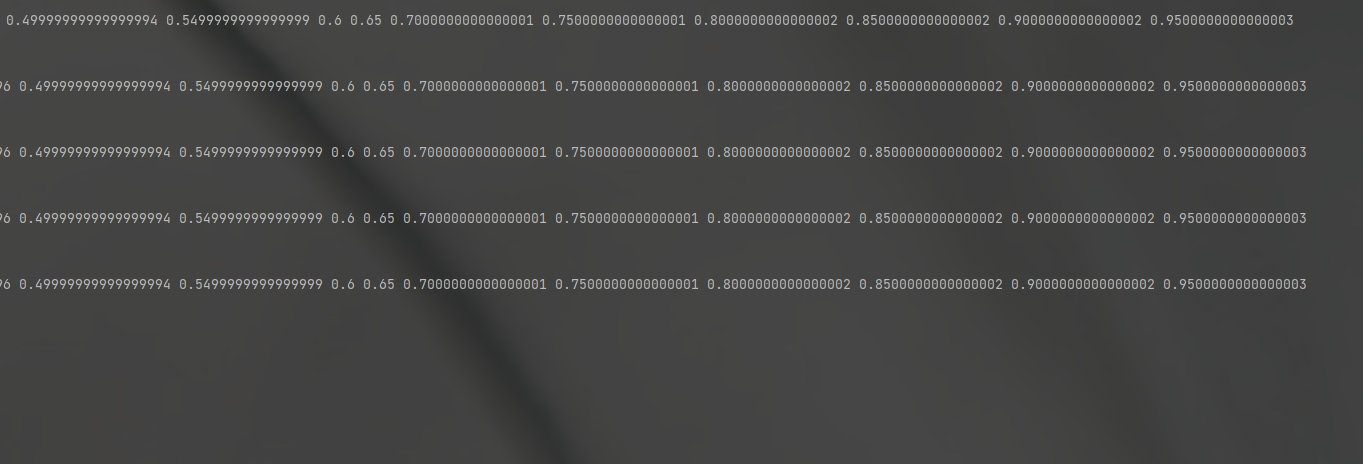
1）QuickSort：

1. public void sort(Comparable[] objs,int left,int right){
2. if(left<right)
3. *//当序列长度大于1时进行划分*
4. {
5. *//获取轴值位置*
6. int pivotIndex=partition(objs,left,right);
7. *//递归左子序列*
8. sort(objs,left,pivotIndex-1);
9. *//递归右子序列*
10. sort(objs,pivotIndex+1,right);
11. }
12. }
13. public int partition(Comparable[] objs,int left,int right)
14. *//按照轴值划分*
15. {
16. int index=findMiddle(objs,left,right);
17. *//取得轴值下标，同时完成对首、中、尾位置的排序*
18. int low=left,high=right-1;
19. exchange(objs,right,index);
20. *//将轴值先放在末尾*
21. Comparable pivot=objs[right];
22. while(low<high){
23. while(true){
24. *//左游标寻找大于轴值的元素*
25. if(less(pivot,objs[low])||(low>=high)){
26. break;
27. }
28. low++;
29. }
30. while(true){
31. *//右游标寻找小于轴值的元素*
32. if(less(objs[high],pivot)||(high<=low)){
33. break;
34. }
35. high--;
36. }
37. if(low<high){
38. *//交换两游标指向元素*
39. exchange(objs,low,high);
40. }
41. }
42. *//将轴值归位*
43. exchange(objs,low,right);
44. *//返回轴值位置*
45. return low;
46. }

2)MergeSort：

1. public void sort(Comparable[] objs,int left,int right){
2. if(left>=right)
3. return;
4. else{
5. int mid=(left+right)/2;
6. *//划分左子序列*
7. sort(objs,left,mid);
8. *//划分右子序列*
9. sort(objs,mid+1,right);
10. *//归并*
11. merge(objs,left,mid,right);
12. }
13. }
14. public void merge(Comparable[] objs,int left,int mid,int right){
15. *//采用双指针*
16. int frontPoint=left;
17. int behindPoint=mid+1;
18. *//开辟新数组用于中转*
19. Comparable[] tmp=new Comparable[right-left+1];
20. int k=0;
21. while((frontPoint<=mid)&&(behindPoint<=right)){
22. *//将两个指针指向的较小值存入新数组，同时更新指针*
23. if(less(objs[frontPoint],objs[behindPoint])){
24. tmp[k++]=objs[frontPoint++];
25. }
26. else{
27. tmp[k++]=objs[behindPoint++];
28. }
29. }
30. *//将剩余数据依次放入新数组*
31. while(frontPoint<=mid){
32. tmp[k++]=objs[frontPoint++];
33. }
34. while(behindPoint<=right){
35. tmp[k++]=objs[behindPoint++];
36. }
37. *//完成拷贝*
38. for(int i=left;i<=right;i++){
39. objs[i]=tmp[i-left];
40. }
41. }

4.运行结果展示：  




5.总结与收获：

1）学会自己独立写出选择、插入、希尔、快速、归并五种排序算法

2）在快速排序与归并排序的实现中运用了递归操作，更加了解递归思想的运用，对递归终止的条件也有所学习

3）学习了双指针法进行序列数据的移动

任务四

1.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：28 、29 、210、……、216 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。

2.数据与算法设计：

1）使用提供的GenerateData、SortTest和LineXYDemo类，在修改LineXYDemo中的数据集时注意：

①X轴为数据规模

②Y轴为对应算法的排序时长。需要用到SortTest类中的test方法，注意指定次数T

③修改“标题“”横轴标题“”纵轴“标题等，修改不同曲线的图例

④**原本提供的test方法有误**，进行了改进（每次测试时，测试原序列的拷贝，保证再次测试时原序列仍然无序）

2）依照LineXYDemo完成对五种算法对应图表类的创建，在主函数中创建五个对应对象，完成展示操作

3）做T次测试取平均值以排除干扰

4）此外，对运行时间的结果进行了取对数操作使图像更好展示

3.部分代码说明：

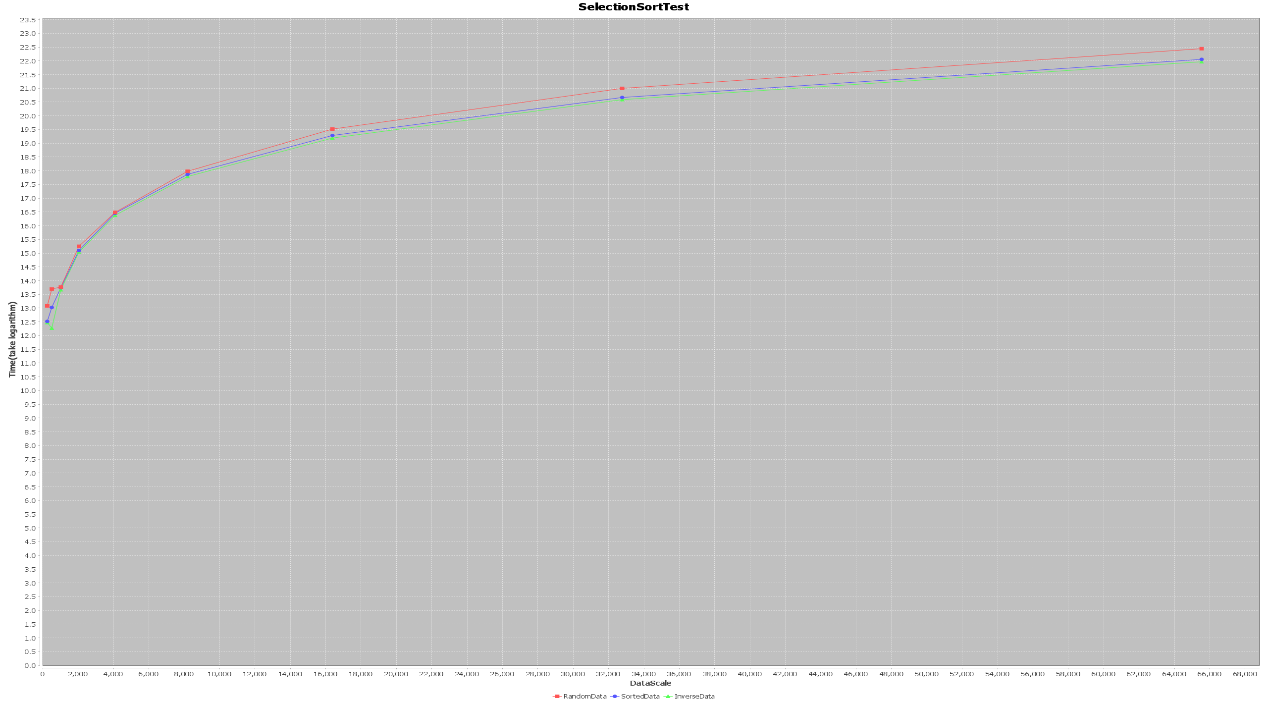
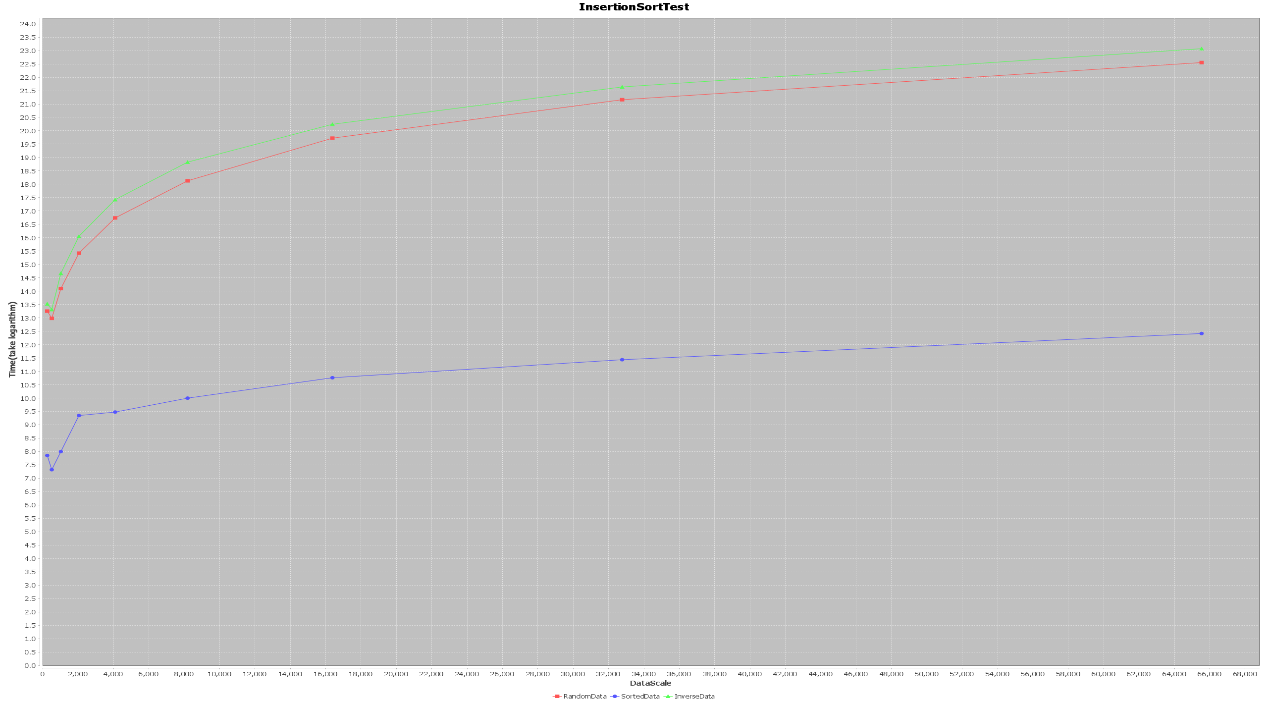
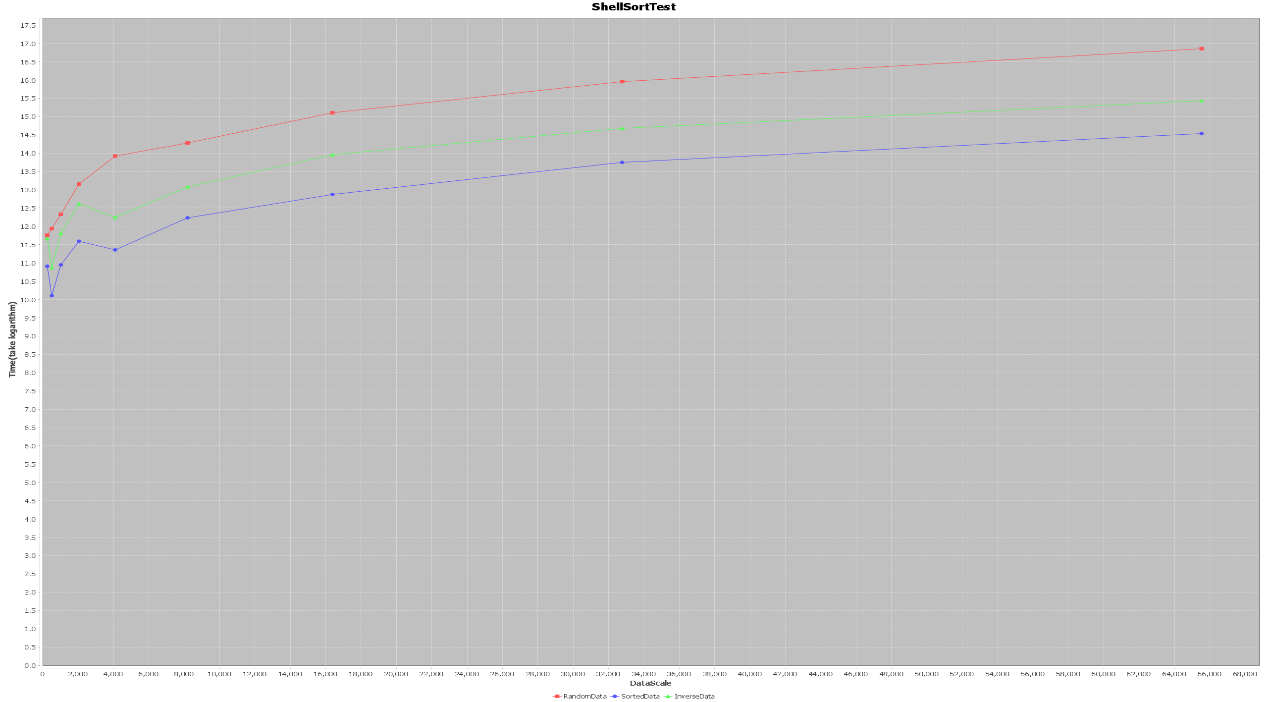
以InsertionTestDemo为例：在创建数据集时：

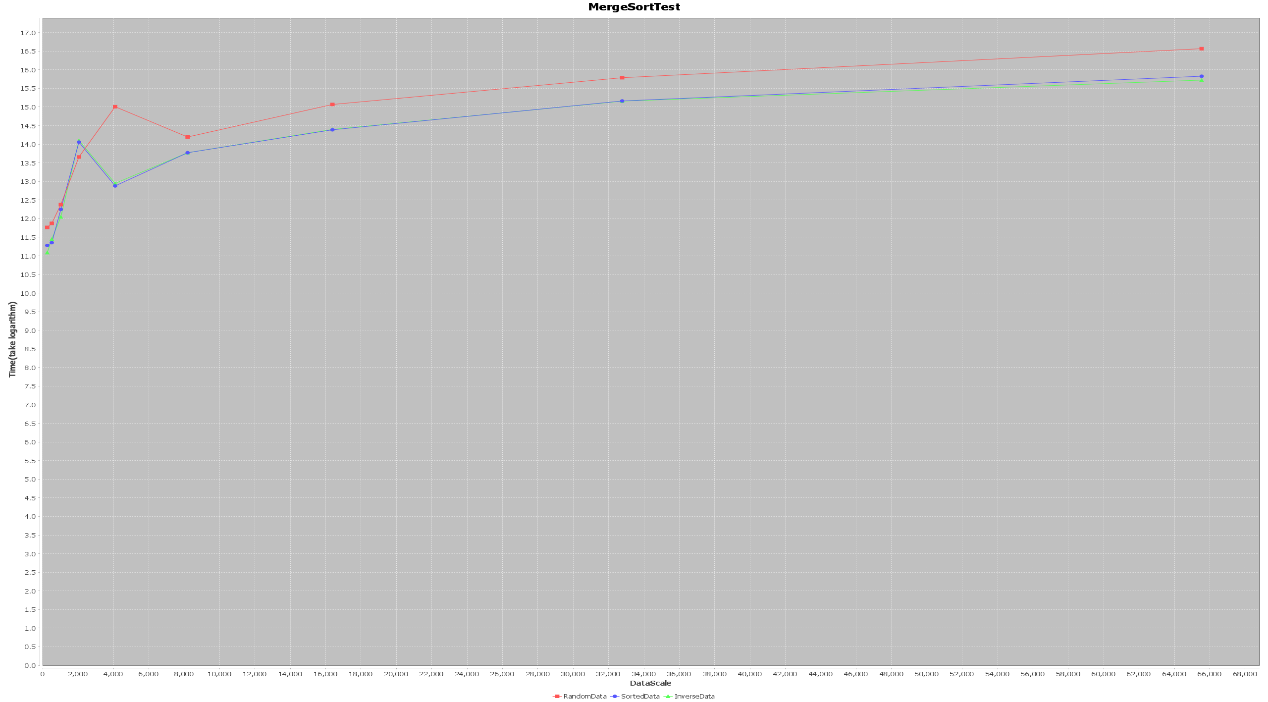
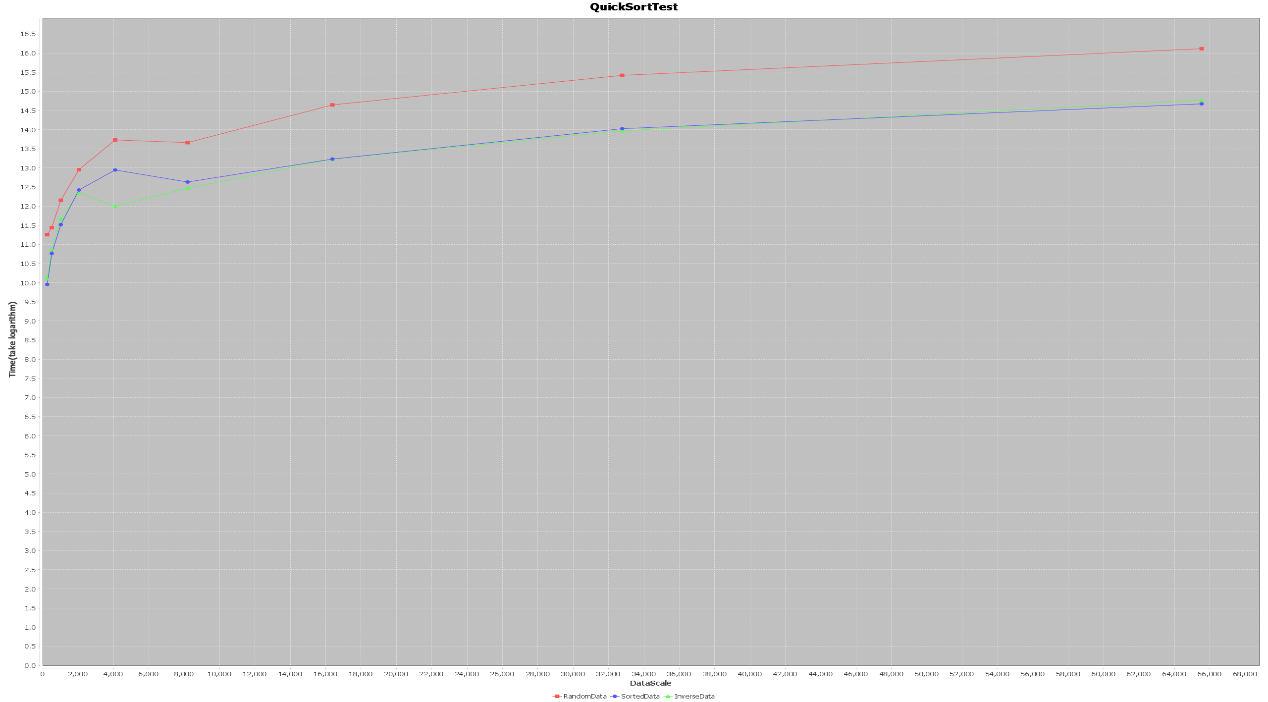
1. private XYDataset createDataset() {
2. *// 本样例中想要显示的是三组数据的变化图*
3. *// X数组是三组数据共同拥有的x坐标值；Y1、Y2和Y3数组分别存储了三组数据对应的y坐标值*
4. *// 共有9个节点值*
5. int[] X = new int[9];
6. *//建立X坐标*
7. double[] Y1 = new double[9];
8. double[] Y2 = new double[9];
9. double[] Y3 = new double[9];
10. *//建立三种Y坐标分别对应三种数据序列*
11. for (int j = 8; j <= 16; j++) {
12. int N = (int) pow(2, j);
13. X[j - 8] = N;
14. *//X为数据规模*
15. Double[] randomData = GenerateData.getAnotherRandomData(N);
16. Y1[j - 8] = Math.log(SortTest.test(new Insertion(), randomData, 10));
17. Double[] sortedData = GenerateData.getAnotherSortedData(N);
18. Y2[j - 8] = Math.log(SortTest.test(new Insertion(), sortedData, 10));
19. Double[] inverseData = GenerateData.getAnotherInversedData(N);
20. Y3[j - 8] = Math.log(SortTest.test(new Insertion(), inverseData, 10));
21. *//Y为该算法对不同种类数据序列测试10次用时的平均值*
22. }
23. double[][] Y = {Y1, Y2, Y3};
24. *// 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*
25. XYSeries[] series = {new XYSeries("RandomData"), new XYSeries("SortedData"), new XYSeries("InverseData")};
26. int N = X.length;
27. int M = series.length;
28. for (int i = 0; i < M; i++)
29. for (int j = 0; j < N; j++)
30. series[i].add(X[j], Y[i][j]);
31. *// 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*
32. XYSeriesCollection dataset = new XYSeriesCollection();
33. for (int i = 0; i < M; i++)
34. dataset.addSeries(series[i]);
35. return dataset;
36. }

4.运行结果展示：

（红色曲线表示随机序列，蓝色曲线表示顺序序列，绿色曲线表示逆序序列）

（运行时间进行了取对数操作）

****



5.总结与收获：

总结：基本上随着数据规模扩大，五种排序的用时都在增加

1）对于相同算法：

总体上插入排序和选择排序的测试时长随数据规模变化要快于其他三种排序算法（除了对顺序序列进行插入排序）

①插入排序：时长随数据规模增长一直较快，且对于顺序序列的测试时长最优，远小于另外两种测试时长，对于逆序序列的测试时长最劣

②选择排序：时长随数据规模增长一直较快，对于随机序列的测试时长最劣，另外两者没有极为显著的差异

③希尔排序：时长随数据规模增长相对较快，对于随机序列的测试时长最劣，对顺序序列的测试时长最优；

④快速排序：时长随数据规模增长较为稳定，对于随机序列的测试时长最劣，另外两种测试结果没有显著差异，且在不同数据规模下各有优劣

⑤归并排序：时长随数据规模增长较为稳定，对于随机序列的测试时长最劣，另外两种测试结果没有显著差异，且在不同数据规模下各有优劣

2）对于相同数据规模和分布

①随机数据：插入排序与选择排序差异不大且劣于其他三种，其他三种排序算法差异不大，快速排序最佳

②顺序数据：插入排序始终最优，选择排序最劣，其他三种算法测试时间大致为：希尔排序<快速排序<归并排序

③逆序数据：插入排序于选择排序差异不大且插入排序更劣，其他三种排序算法差异不大，快速排序最佳

任务五

1.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。

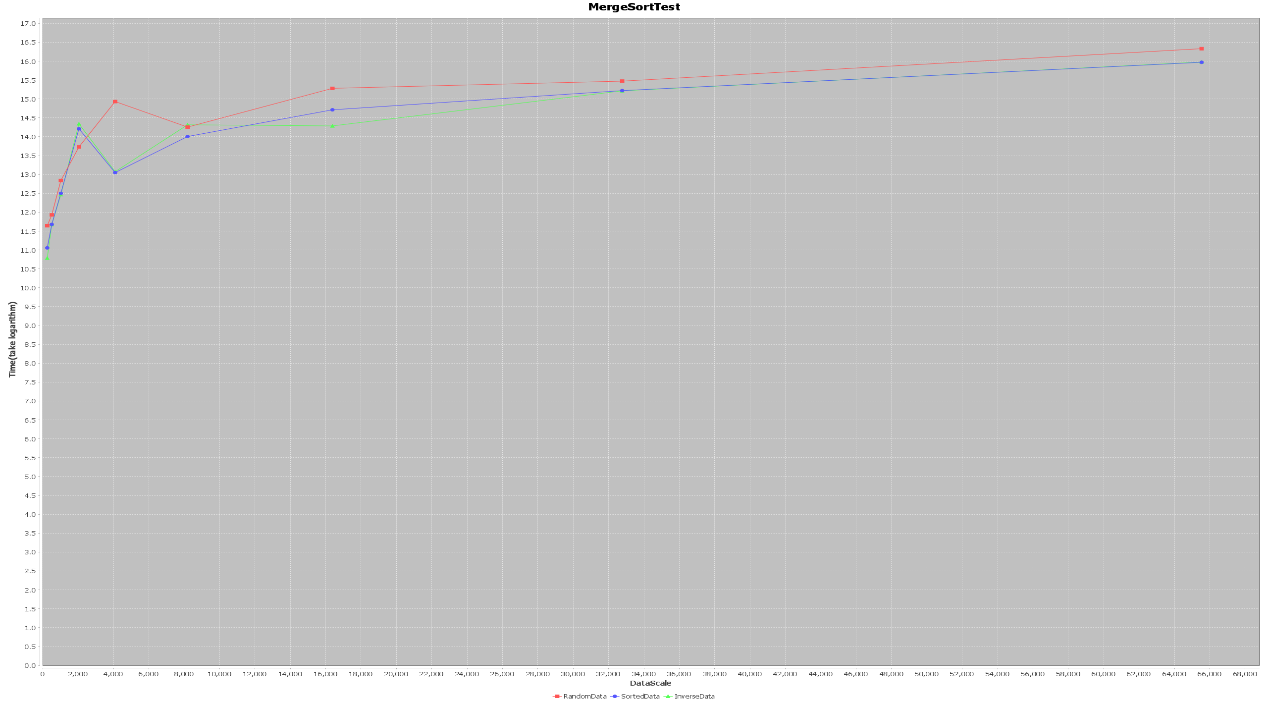
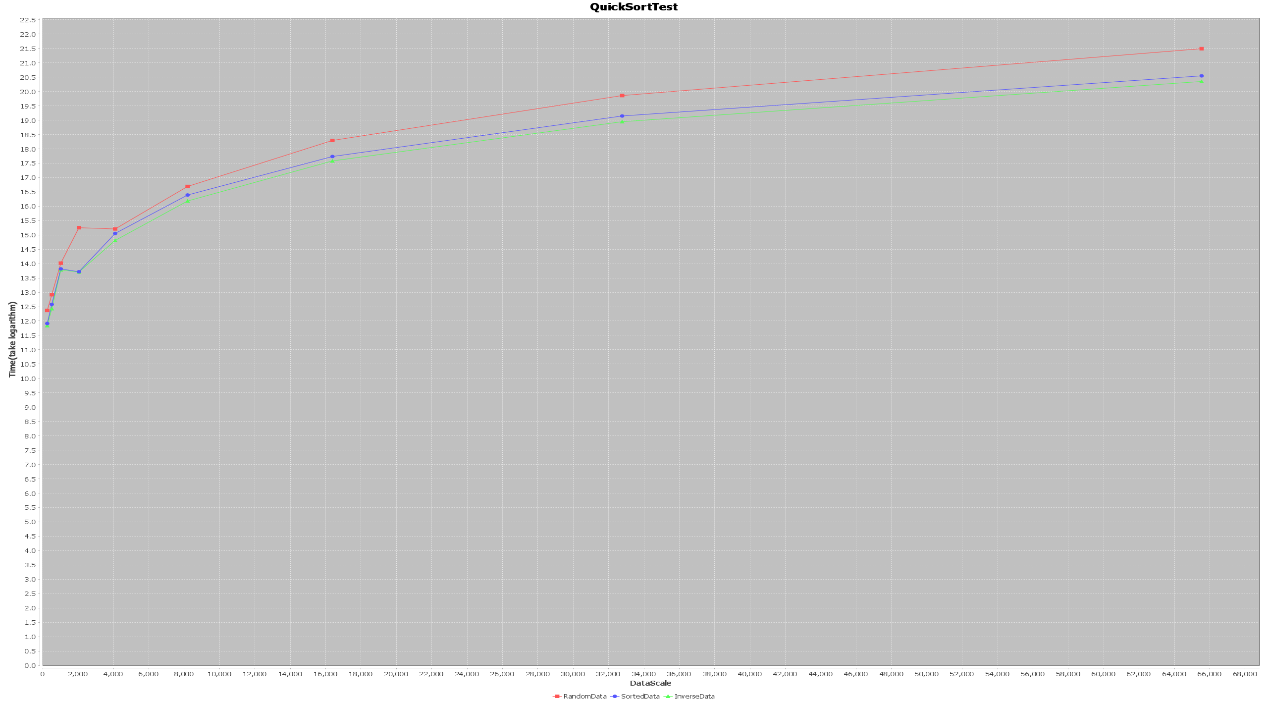
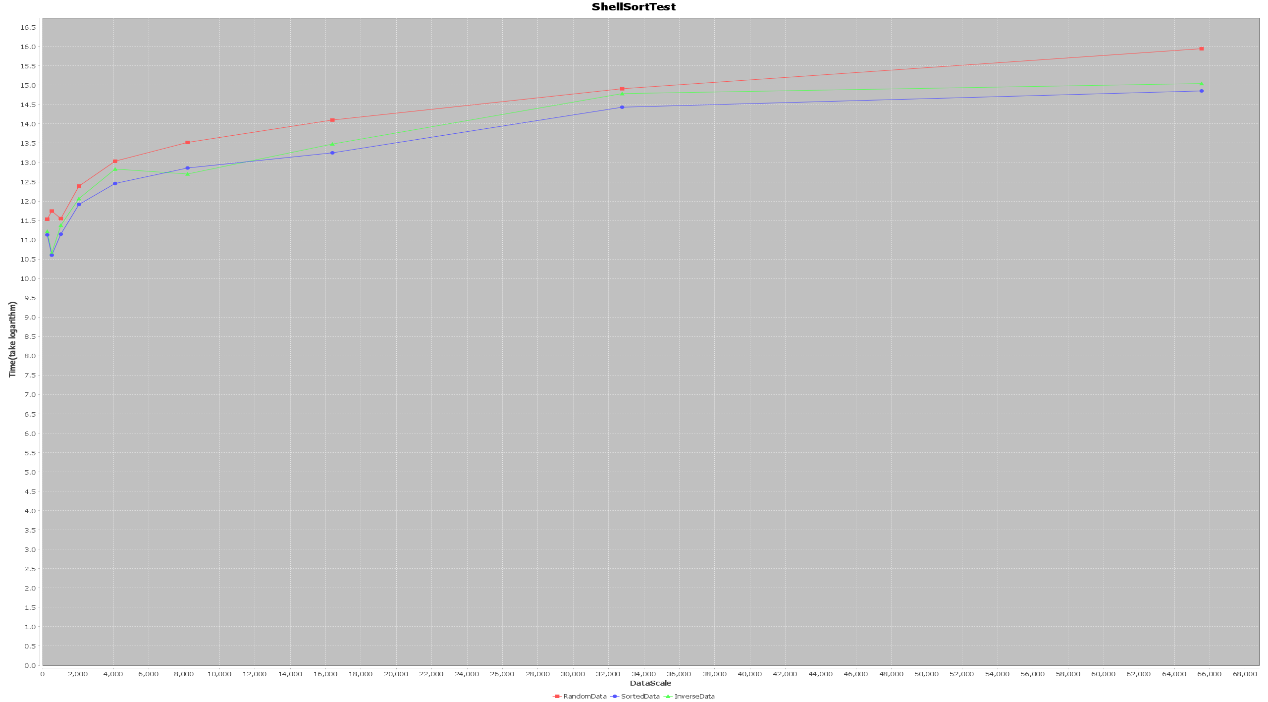
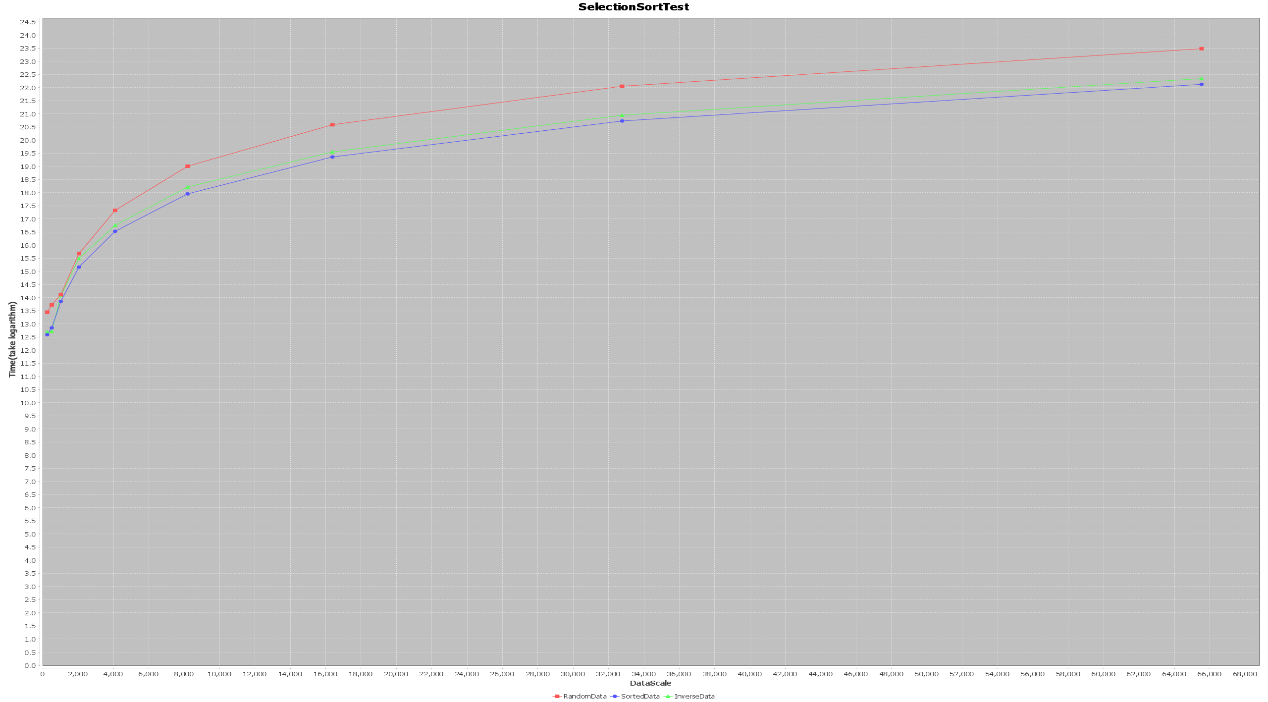
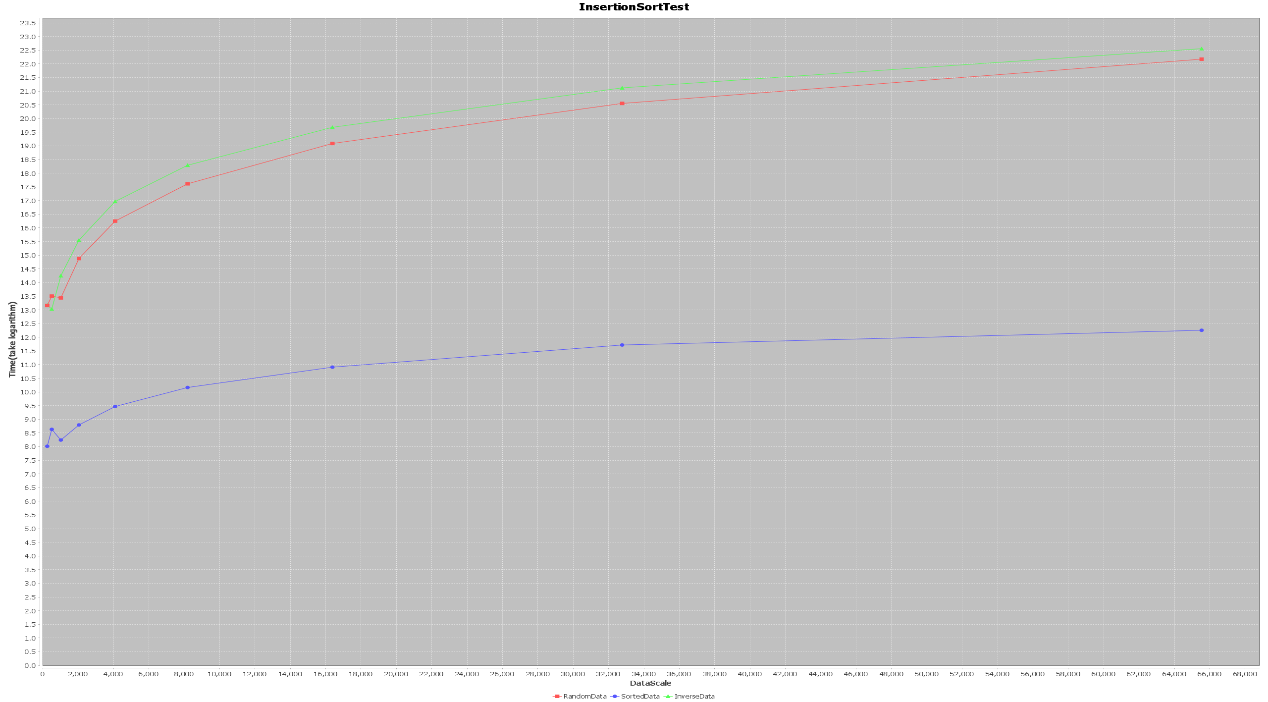
2.数据与算法设计：

同任务四一致，只是更换了数据序列，为GenerateData类增添了三种新方法，即生成随机、顺序、逆序的满足题意的序列，再更换生成图表类中的X轴数据即可。

3.部分代码说明：

1. public static Double[] getAnotherRandomData(int N){
2. *//打乱有序数据即可*
3. Double[] numbers = getAnotherSortedData(N);
4. shuffle(numbers, 0, numbers.length);
5. return numbers;
6. }
7. public static Double[] getAnotherSortedData(int N){
8. *//顺序序列*
9. Double[] numbers = new Double[N];
10. int i=0;
11. *//按顺序将相应数据确定下来*
12. while(i<(N/2)){
13. numbers[i]=0.0;
14. i++;
15. }
16. while(i<(3\*N/4)){
17. numbers[i]=1.0;
18. i++;
19. }
20. while(i<(7\*N/8)){
21. numbers[i]=2.0;
22. i++;
23. }
24. while (i<N){
25. numbers[i]=3.0;
26. i++;
27. }
28. return numbers;
29. }
30. public static Double[] getAnotherInversedData(int N){
31. *//逆序序列*
32. Double[] numbers = new Double[N];
33. int i=0;
34. *//按逆序将指定数据确定下来*
35. while (i<(N/8)){
36. numbers[i]=3.0;
37. i++;
38. }
39. while(i<(N/4)){
40. numbers[i]=2.0;
41. i++;
42. }
43. while(i<(N/2)){
44. numbers[i]=1.0;
45. i++;
46. }
47. while(i<N){
48. numbers[i]=0.0;
49. i++;
50. }
51. return numbers;
52. }

4.运行结果展示：



5.总结与收获：

总结：基本上随着数据规模扩大，五种排序的用时都在增加

1）对于相同算法：

总体上插入排序和选择排序的测试时长随数据规模变化要快于其他三种排序算法（除了对顺序序列进行插入排序）

①插入排序：时长随数据规模增长一直较快，且对于顺序序列的测试时长最优，远小于另外两种测试时长，对于逆序序列的测试时长最劣

②选择排序：时长随数据规模增长一直较快，对于随机序列的测试时长最劣，顺序序列欲逆序序列差异没有很显著，顺序序列相对更优

③希尔排序：时长随数据规模增长逐渐变快，对于随机序列的测试时长最劣，对顺序序列的测试时长最优；

④快速排序：时长随数据规模增长较为稳定，对于随机序列的测试时长最劣，另外两种测试结果差异不是很显著，逆序序列相对更优

⑤归并排序：时长随数据规模增长较为稳定，对于随机序列的测试时长最劣，另外两种测试结果没有显著差异，且在不同数据规模下各有优劣

2）对于相同数据规模和分布

①随机数据：插入排序与选择排序差异不大，插入排序略优与选择排序，且显著劣于其他三种，其他三种排序算法测试时间大致为：希尔排序<归并排序<快速排序（按照时长从小到大）

②顺序数据：插入排序始终最优，选择排序始终最劣，其他三种算法测试时间大致为：希尔排序<归并排序<快速排序（按照时长从小到大）

③逆序数据：插入排序与选择排序差异不大，其他三种排序算法测试时间大致为：希尔排序<归并排序<快速排序（按照时长从小到大）

3）与任务四纵向比对

①插入排序：更换后的分布不均匀数据序列使得相同数据规模下测试时长略微增大，因为重复数据的插入会更加快速

②选择排序：更换后的分布不均匀数据序列使得相同数据规模下测试时长略微增大，变化不大

③希尔排序：更换后的分布不均匀数据序列使得相同数据规模下测试时长略微减小，变化不大

④快速排序：更换后的分布不均匀数据序列使得相同数据规模下测试时长显著增大，因为过多的重复元素使得快速排序在轴值划分时容易令一边子序列的长度很小，使得划分次数增多。

⑤归并排序：数据序列变化影响不大，但仍使得相同规模数据下测试时长减小，因为在合并阶段，重复数据的合并操作会更加快速。

任务六

1.题目：快速排序的再探讨和应用

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：

2.问题一

2.1题目：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参照任务4的要求进行测试，并与任务 4 中没有优化的快速排序算法的执行时间进行对比；

2.2数据与算法设计：

只需更改原本QuickSort类中主函数的部分逻辑，当左边界离右边界的距离小过5时，切换为插入排序，直接调用Insertion即可，其他情况下仍保持原本快速排序结构。

修改生成图表的类，使其中test调用的算法为改进后的快速排序

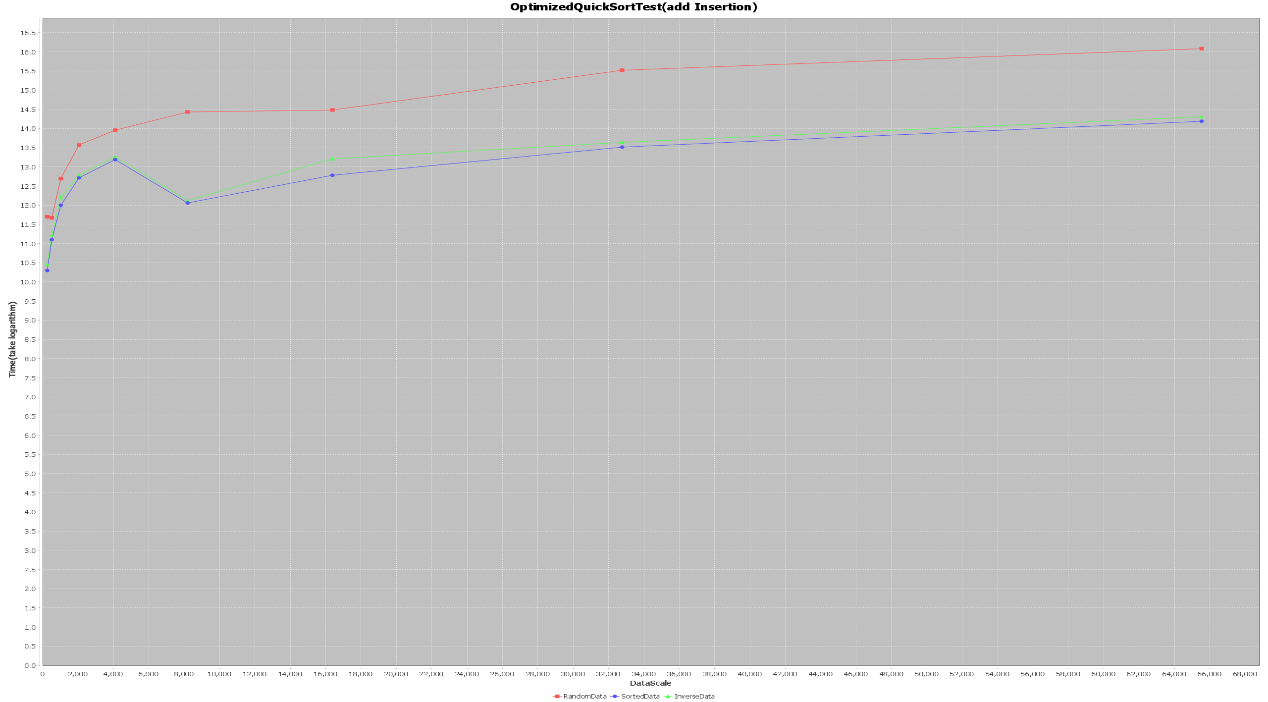
2.3.部分代码说明：

将原本的主函数

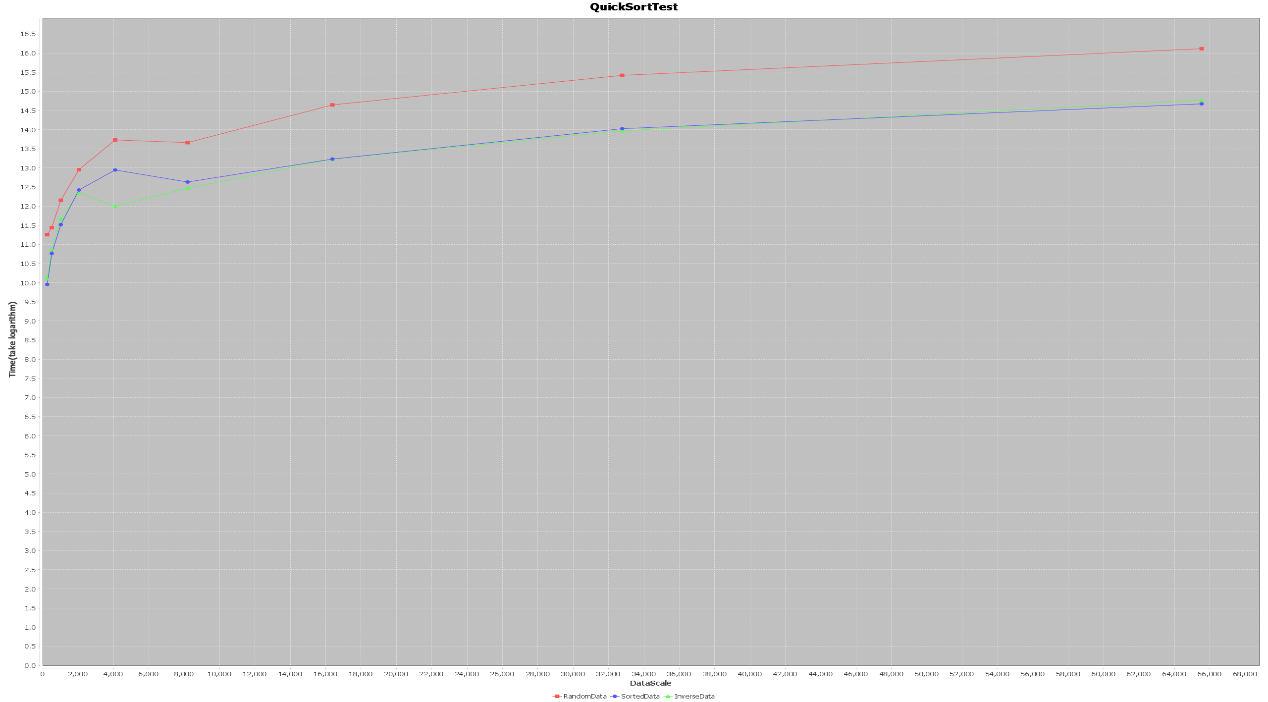
1. public void sort(Comparable[] objs,int left,int right){
2. if(left<right)
3. *//当序列长度大于1时进行划分*
4. {
5. *//获取轴值位置*
6. int pivotIndex=partition(objs,left,right);
7. *//递归左子序列*
8. sort(objs,left,pivotIndex-1);
9. *//递归右子序列*
10. sort(objs,pivotIndex+1,right);
11. }
12. }

改为

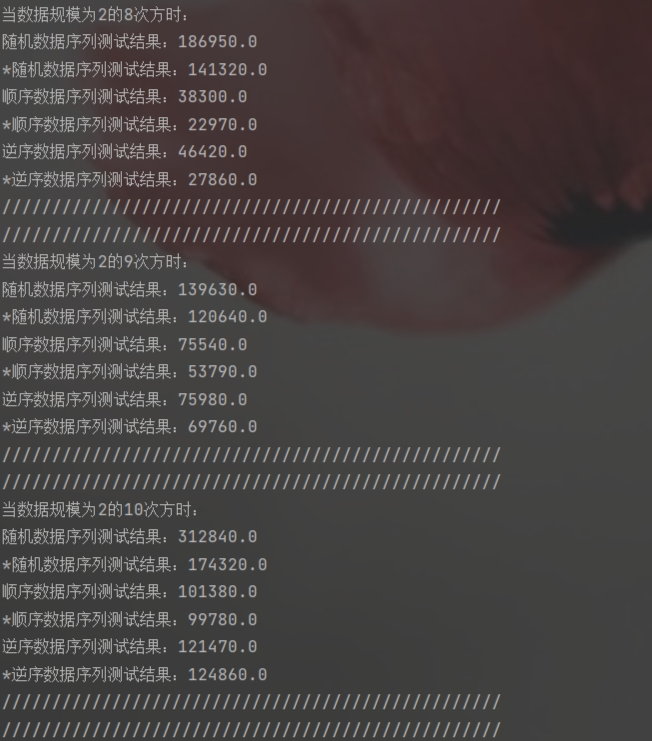
1. SortAlgorithm tmp=new Insertion();
2. public void sort(Comparable[] objs,int left,int right){
3. if(right<=left+5){*//确定阈值为5，当数据量小于5时，调用插入排序*
4. tmp.sort(objs,left,right);
5. return;}
6. int pivotIndex=partition(objs,left,right);
7. sort(objs,left,pivotIndex-1);
8. sort(objs,pivotIndex+1,right);
9. }

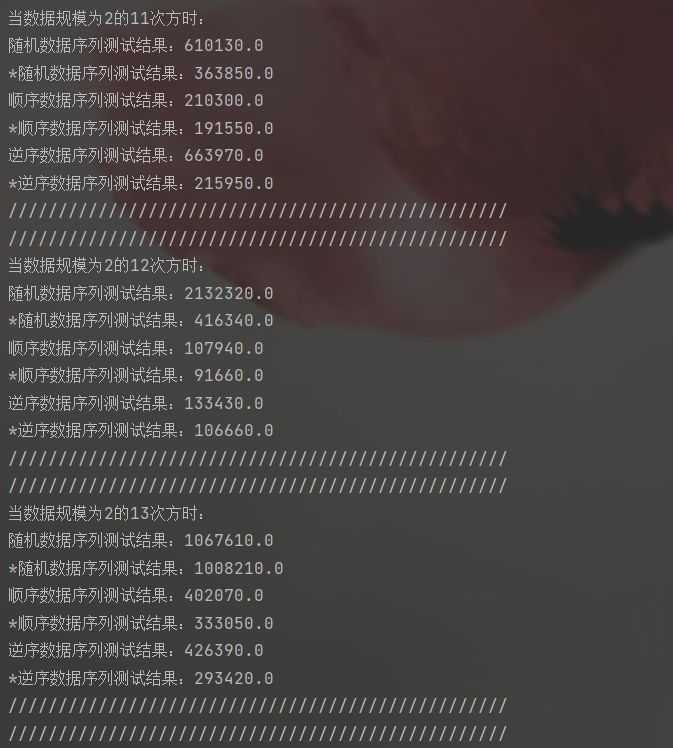
2.4.运行结果展示：

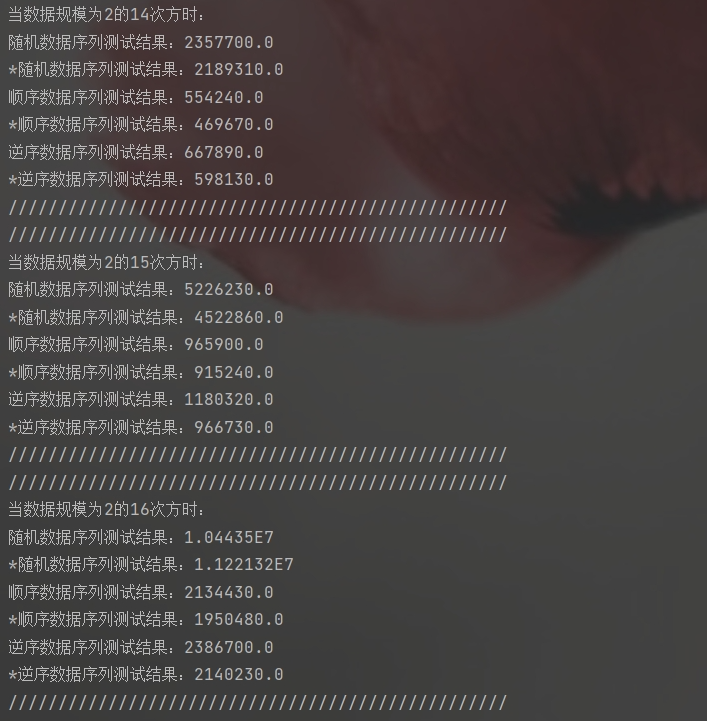
与未优化的快速排序比对



同时测得具体数据：（带\*为优化后快排得到数据）







2.5.总结与收获：

根据数据可以直观得出：优化后的快速排序始终比原排序用时更少，而在数据规模变得很大之后，优化的效果相对不够明显

3.问题二

3.1题目：

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

3.2.算法设计

需要完成的任务有：

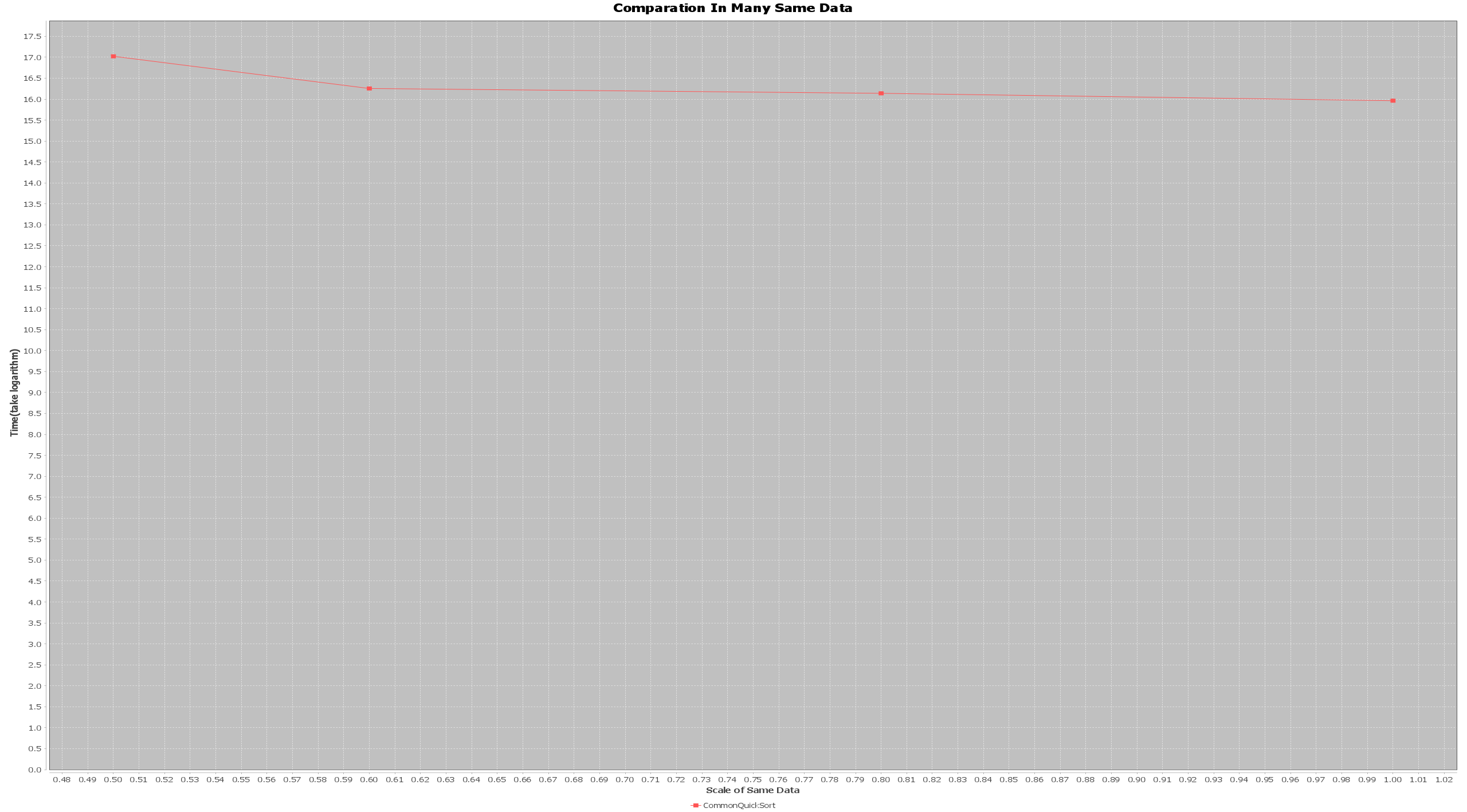
1）在GenerateData类中编写新方法，使得其能生成满足题意要求的数据序列，在此采用random类，先生成一个随机数，令序列中满足题意规模的数据等于该随机值，其余值随机设定，之后打乱该序列

2）在生成图表的类中需要修改X轴值，使其代表数据的重复率

3.3.部分代码说明

1. public static Double[] getfiftypercentSameData(int N){
2. Double[] numbers=new Double[N];
3. Double samedata=random.nextDouble()\*N;
4. *//随机确定重复数据的值*
5. for(int i=0;i<N;i++){
6. if(i<(N/2)){
7. *//使要求规模下的数据为重复值*
8. numbers[i]=samedata;}
9. else {
10. *//为剩余数据赋随机值*
11. numbers[i]=random.nextDouble()\*N;}
12. }
13. *//打乱序列*
14. shuffle(numbers,0,N);
15. return numbers;
16. }

3.4运行结果展示



3.5总结

发现在数据重复率为0.5，0.6，0.8，1的情况下，优化后快速排序算法的测试时间始终小于对随机序列的测试时间，且测试时间随着数据重复率的增多而减小

4.问题三