|  |  |
| --- | --- |
| **数据结构与算法 作业报告** | |
| 第1次 | |
| 图片包含 标牌  已生成极高可信度的说明 | |
|  | |
|  | |
|  | |
| **姓名** |  |
| **班级** |  |
| **学号** |  |
| **电话** |  |
| **Email** |  |
| **日期** |  |

# 目录

### 对若干个排序算法的比较 1

### 任务一 2

1. 题目 2
2. 数据设计 2
3. 算法设计 2
4. 主干代码说明 3
5. 运行结果展示 4
6. 总结和收获 4

### 任务二 4

1. 题目 4
2. 数据设计 5
3. 算法设计 5
4. 主干代码说明 5
5. 运行结果展示 5
6. 总结和收获 6

### 任务三 7

1. 题目 7
2. 数据设计 7
3. 算法设计 7
4. 主干代码说明 7
5. 运行结果展示 7
6. 总结和收获 10

### 任务四 10

1. 题目 10
2. 数据设计 10
3. 算法设计 10
4. 主干代码说明 10
5. 运行结果展示 11
6. 总结和收获 12

**附录：源代码 13**

# 对若干个排序算法的比较



# 任务一

##### 题目

##### 数据设计

1. 选择排序：需要一个变量minIndex来存储每次找到的最小值位置。此外，需要控制循环的变量。
2. Shell排序：利用控制循环的变量对不同间距的子序列进行插入排序即可。
3. 快速排序：利用pivotIndex来存储轴值，m，n分别控制从数组两端向中间的遍历。
4. 归并排序：利用midIndex存储数组的中间位置，利用新数组temp复制原数组objs，并利用index，n1，n2控制将两侧的数组按顺序复制进原数组。

##### 算法设计

1. 选择排序：在每一次循环中，利用第二个循环找到未排序部分数组中的最小值，将它放置到排序的数组部分。
2. Shell排序：分别对规模由小至大、间距由大至小的子序列进行insertSort插入排序。insertSort与插入排序同理，但需要指定子序列的起始和间距。
3. 快速排序：取数组中间点为轴值，将轴值放置在末尾，从两端移动数组下标，寻找到左侧大于轴值的位置和右侧小于轴值的位置，将二者调换。之后插入轴值，对轴值两侧的子序列递归调用插入排序。
4. 归并排序：将数组分为左右两子序列，对每一子序列递归调用归并排序直至子序列长度为一。对于两排好序的子序列，复制他们到temp，比较其第一位的值并取最小值记录至原数组即可。

##### 主干代码说明

1. 选择排序： 以下代码实现了算法设计中所述主要内容。

**for** (**int** i = 0; i < N - 1; i++) {

**int** minIndex = i;

**for** (**int** j = N - 1; j > i; j--) {

**if** (less(objs[j], objs[minIndex])) {

            minIndex = j;

        }

    }

    exchange(objs, i, minIndex);

}

1. Shell排序：以下代码利用for循环实现对不同子序列进行插入排序

**public** **void** sort(Comparable[] objs) {

**for** (**int** i = objs.length / 3; i > 3; i /= 3) {//元素间距

**for** (**int** j = 0; j < i; j++) {

                insertSort(objs, j, i);

            }

        }

        insertSort(objs, 0, 1);

    }

1. 快速排序：其中do-while循环为核心，循环的最后一次会把m,n增加至相反位置，需要通过 exchange(objs,m,n);换回来。

exchange(objs,pivotIndex,j);//将轴值放置在末尾，因为轴值所处的位置未确定

**int** m = i-1, n = j;

**do**{//从两端移动下标直至相遇

**while**(less(objs[++m], objs[j]));

**while**(n!=0&&less(objs[j], objs[--n]));//每次循环中，在两端分别寻找两侧小于和大于轴值的数

           exchange(objs,m,n);

       }**while**(m<n);

       exchange(objs,m,n);//最后一次循环时多换了一次

       exchange(objs,m,j);//将轴值放置在适当位置

1. 归并排序：以下代码实现了递归调用排序和对左右侧数组进行合并。

mergesort(objs,temp,i,midIndex);

       mergesort(objs,temp,midIndex+1,j);

       System.arraycopy(objs, i, temp, i, j + 1 - i);

**int** n1=i,n2=midIndex+1;//左侧数组和右侧数组的位置指示

**for**(**int** index=i;index<=j;index++) {//将左右侧数组按大小复制进原数组

**if** (n1 == midIndex + 1) {//左侧数组已复制完

               objs[index] = temp[n2++];

           } **else** **if** (n2 == j + 1) {//右侧数组已复制完

               objs[index] = temp[n1++];

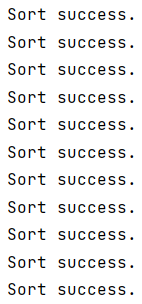
           } **else** **if** (less(temp[n1], temp[n2])) {//左侧数组首位较小

               objs[index] = temp[n1++];

           } **else** {//右侧数组首位较小

               objs[index] = temp[n2++];

           }

       }

##### 运行结果展示

所有五种排序方法都可以成功实现排序，如右图所示。

##### 总结和收获

本任务实现了对五种排序方法的实现，增进了对内排序几种方法的理解，同时这些实现是后续时间比较任务的基础。

# 任务二

##### 题目

##### 数据设计

本任务无需进行数据设计。

##### 算法设计

分别对五种算法运行SortTest.java，将他们在不同数据规模下的运行时间记录，将之输入LineXYDemo.java生成图表即可。

##### 主干代码说明

略

##### 运行结果展示

分别在2^8-2^16的数据量下：

插入排序：

1800940.0000 3647780.0000 3011340.0000 7755660.0000 27687280.0000 141797080.0000 477378660.0000 1879872760.0000 12448244660.0000

选择排序：

1966860.8000 2012300.0000 13650739.6000 8551279.8000 20753800.0000 98326539.6000 347124080.0000 1210900260.2000 5315471140.2000

Shell排序：

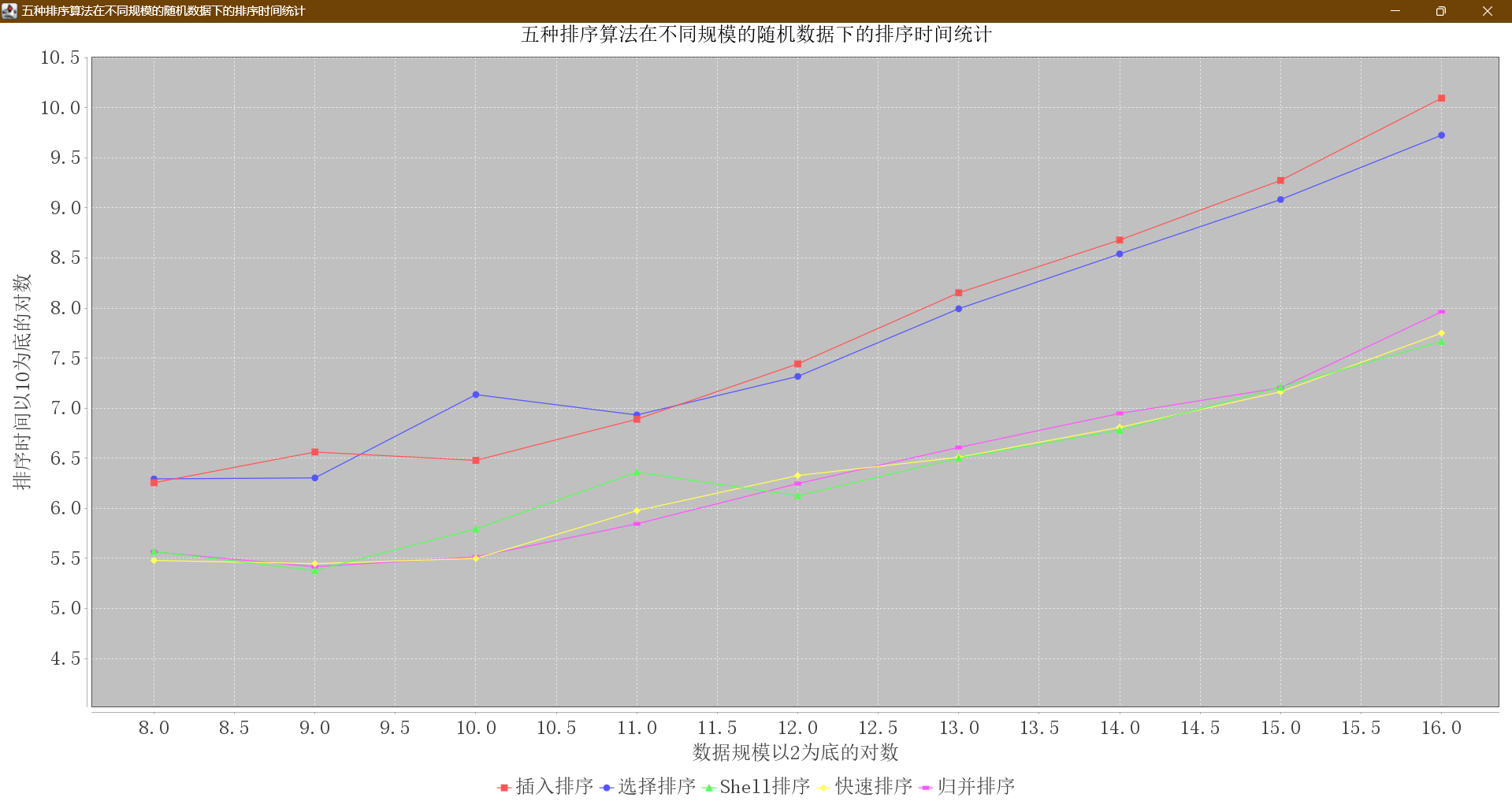
370280.4000 238760.2000 623160.4000 2289859.8000 1334020.4000 3169099.4000 6010799.8000 16191860.2000 46364159.6000

快速排序：

301540.0000 280020.0000 315980.0000 947520.0000 2125360.0000 3232740.0000 6437000.0000 14561780.0000 56196320.0000

归并排序：

366840.2000 260940.0000 326859.8000 699440.4000 1766240.0000 4053180.2000 8854479.8000 15997160.2000 91626700.0000

生成图表后如图：

##### 总结和收获

本任务利用图表清晰地展示了五种内排序方法在不同数据规模下的排序时间，显然，五种排序方法所需的运行时间都随数据规模的增大而增大，且运行时间的对数和数据规模的对数似乎成线性关系；而插入排序和选择排序所需的时间显著比另外三种排序方法高（差距可达3个数量级），且增长比另外三种快，另外三种排序方法所需的时间基本不相上下。

# 任务三

##### 题目

##### 数据设计

本任务无需进行数据设计。

##### 算法设计

分别对五种算法运行SortTest.java，将他们在不同数据规模下的运行时间记录；改变程序中的k值，再次运行几次，将数据输入LineXYDemo.java生成图表即可。

##### 主干代码说明

略

##### 运行结果展示

分别在2^8-2^16的数据量下：

插入排序：

k=1：1720.0000 7980.0000 17080.0000 12840.0000 31420.0000 152159.8000 191340.2000 297940.6000 684680.2000

k=5：19859.8000 46900.0000 41159.6000 124740.2000 215979.8000 446460.0000 910060.4000 1785379.6000 3207500.2000

k=10：30060.2000 44140.0000 86780.0000 170239.8000 352599.8000 698320.0000 1513820.0000 2758279.8000 4695440.0000

k=15：26899.8000 53339.6000 103600.2000 110379.6000 364540.0000 787560.0000 1829560.2000 2676080.4000 6545460.0000

选择排序：

k=1：59400.0000 326040.0000 1243740.0000 5005380.2000 16732920.4000 74174299.8000 288722239.8000 1166213859.6000 5426685580.0000

k=5：88099.8000 216660.4000 1395139.8000 5053839.8000 19151020.6000 75033039.6000 289245320.0000 1147405560.2000 5442412440.0000

k=10：67800.0000 221060.0000 978380.0000 4723460.0000 16682579.6000 72818480.0000 296105799.8000 1141472899.8000 6007012600.2000

k=15：82120.4000 310139.8000 2700120.0000 9639780.0000 34520180.0000 148597559.8000 579046920.2000 2394086580.0000 8329755599.6000

Shell排序：

k=1：6120.2000 8840.0000 34939.8000 85979.8000 251640.0000 384560.2000 1671160.2000 2464940.0000 5741320.2000

k=5：9019.8000 18380.2000 42740.2000 94919.4000 226840.2000 469100.2000 882999.8000 3719980.2000 8937300.0000

k=10：11599.8000 25720.0000 43539.8000 114360.2000 206620.2000 446059.6000 1590420.0000 4451699.8000 9795500.0000

k=15：13840.0000 30319.8000 60160.0000 137139.6000 264920.6000 565939.6000 1564939.8000 3384640.0000 9632840.0000

快速排序：

k=1：10480.2000 22419.6000 51660.0000 132379.8000 216900.2000 794100.2000 1323200.0000 2073139.6000 3879720.2000

k=5：15159.8000 29899.8000 59660.0000 142180.4000 313739.6000 761139.8000 1734759.6000 2818539.8000 6291980.6000

k=10：21699.6000 47340.0000 88439.6000 129400.0000 269239.8000 780080.0000 1737500.0000 3474700.2000 7442740.2000

k=15：18940.0000 33100.4000 67339.6000 131900.0000 280920.2000 639620.0000 1649700.2000 3471999.8000 7567660.0000

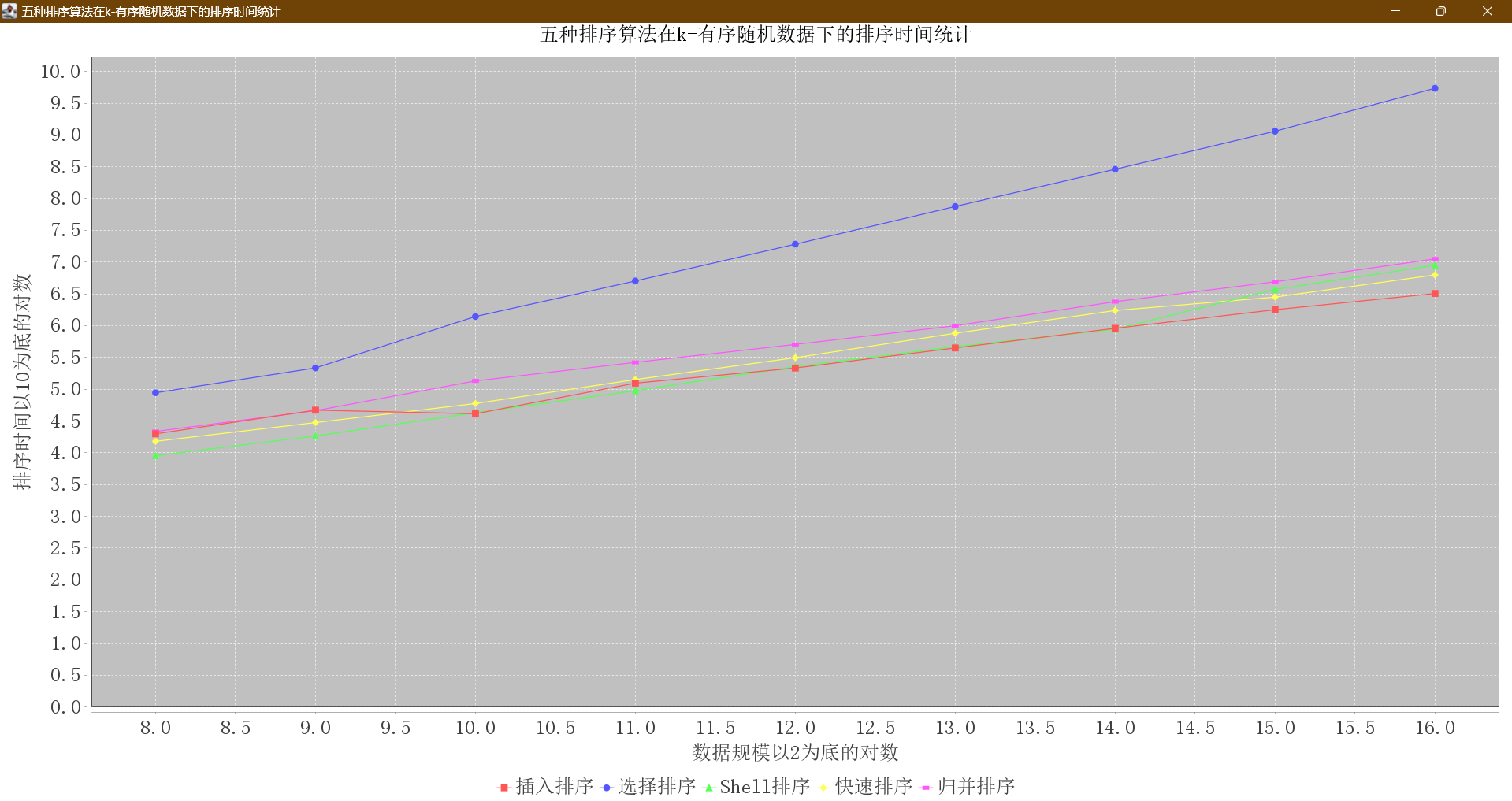
归并排序：

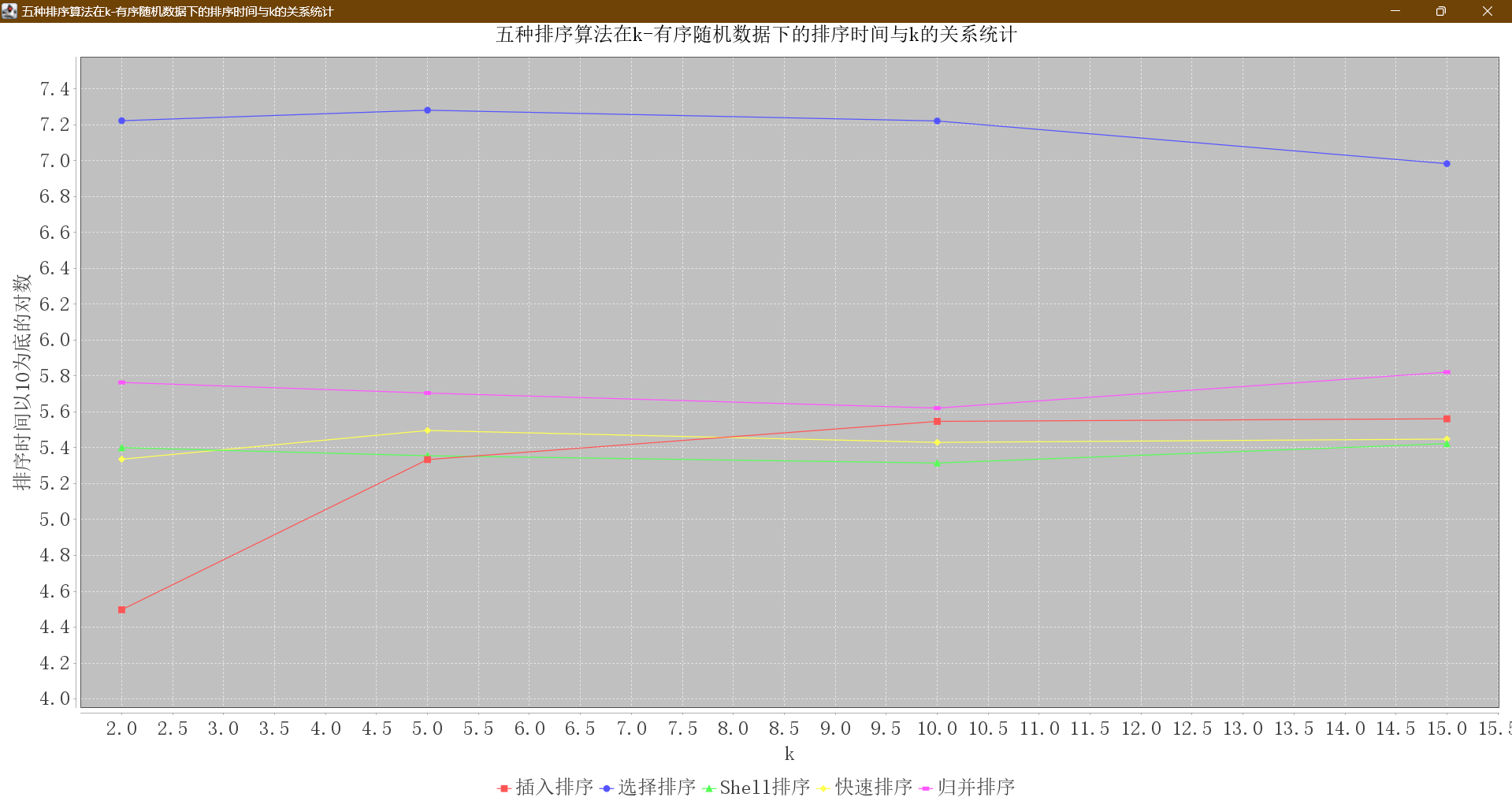
k=1：32540.0000 61320.0000 139400.0000 280280.0000 581340.0000 1185020.0000 2253600.0000 4695420.0000 7140120.0000

k=5：21680.0000 46040.0000 135220.0000 264920.0000 507200.0000 999760.0000 2387540.0000 4908160.0000 11249820.0000

k=10：26200.0000 50560.0000 101500.0000 248420.0000 418380.0000 933360.0000 2171780.0000 7795240.0000 12709300.0000

k=15：34940.0000 43300.0000 91120.0000 266040.0000 663260.0000 1181100.0000 2569160.0000 5798580.0000 10627520.0000

将各算法k=5时的数据绘制图表：

取2^12数据规模时k的不同取值的数据绘制图表：

##### 总结和收获

通过本题绘制的图表可以看出，对于k-有序的随机数据序列，插入排序的时间复杂度虽为O(n^2)，但其所用时间很少，甚至低于其他四种算法；而选择排序在这种数据序列所用的时间远远高于其他算法。通过对k的横向比较发现，插入排序在k极小时花费的时间也极小，而其他四种算法与k的相关度不高。由此可见，插入排序所耗的时间与数据有序性具有很强的相关度，数据基本有序时非常适合采用插入排序。

# 任务四

##### 题目

##### 数据设计

仿照之前的测试方法，将GenerateData.UNEVEN赋值为3，并利用其进行测试；数据生成过程中先赋值再打乱。

##### 算法设计

在不均匀序列的数据生成过程中，先分别给1/8，1/8，1/4和1/2的数据赋值3，2，1，之后再利用与k-有序序列生成方式相同的方法，利用shuffle函数打乱这些数据即可。

##### 主干代码说明

**switch**(dataProbabilityType){

**case** GenerateData.RANDOM -> numbers = GenerateData.getRandomData(dataLength);

**case** GenerateData.KSORTED -> numbers = GenerateData.getKSortedData(dataLength, k);

**case** GenerateData.UNEVEN -> numbers = GenerateData.getUnevenData(dataLength);

            }

为SortTest类的test方法添加UnevenData情形，以实现不均匀序列的测试。

**for** (**int** i = 0; i < N / 8; i++) {

    numbers[i] = 3.0;

}

**for** (**int** i = N / 8; i < N / 4; i++) {

    numbers[i] = 2.0;

}

**for** (**int** i = N / 4; i < N / 2; i++) {

    numbers[i] = 1.0;

}

**for** (**int** i = N / 2; i < N; i++) {

    numbers[i] = 0.0;

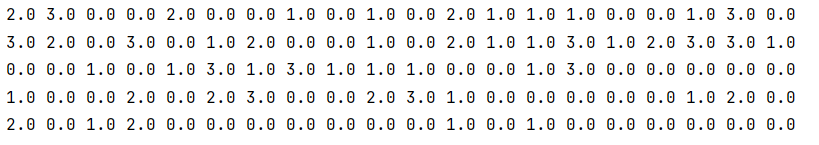
}

shuffle(numbers, 0, N);

实现了题目所述的不均匀序列。

##### 运行结果展示

数据生成（长度为100的不均匀序列）：



分别在2^8-2^16的数据量下：

插入排序：

1236640.2000 1374660.0000 3255679.8000 8858520.2000 30675539.8000 138843100.0000 647143319.6000 2355408840.0000 6822870719.6000

选择排序：

1147840.0000 2590599.6000 4305440.0000 5637679.8000 25926739.8000 144254600.4000 925226680.4000 3985957560.0000 23543593500.4000

Shell排序：

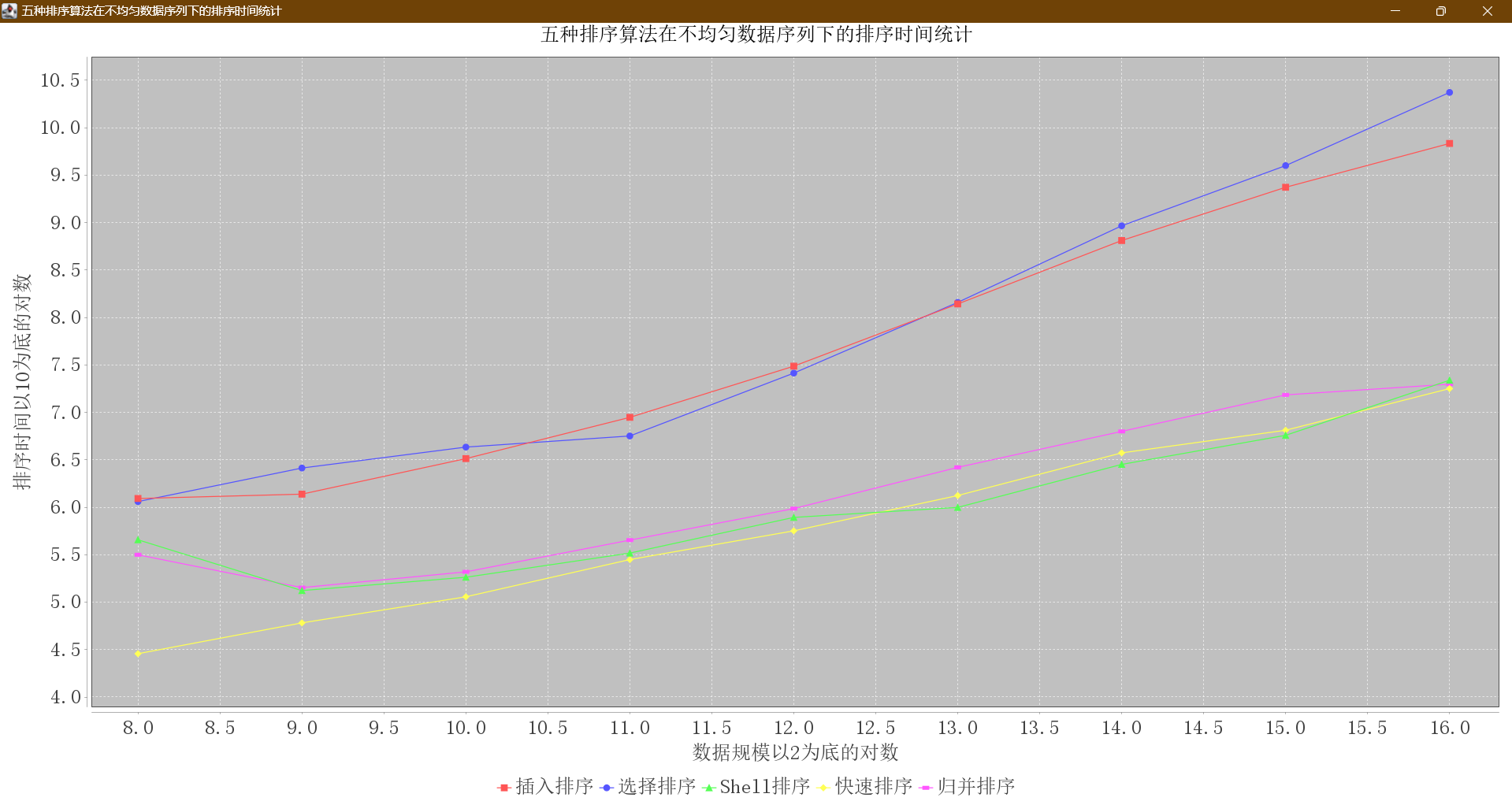
454079.8000 132359.6000 182960.2000 328720.2000 781760.2000 996400.0000 2828500.0000 5730940.2000 21818100.2000

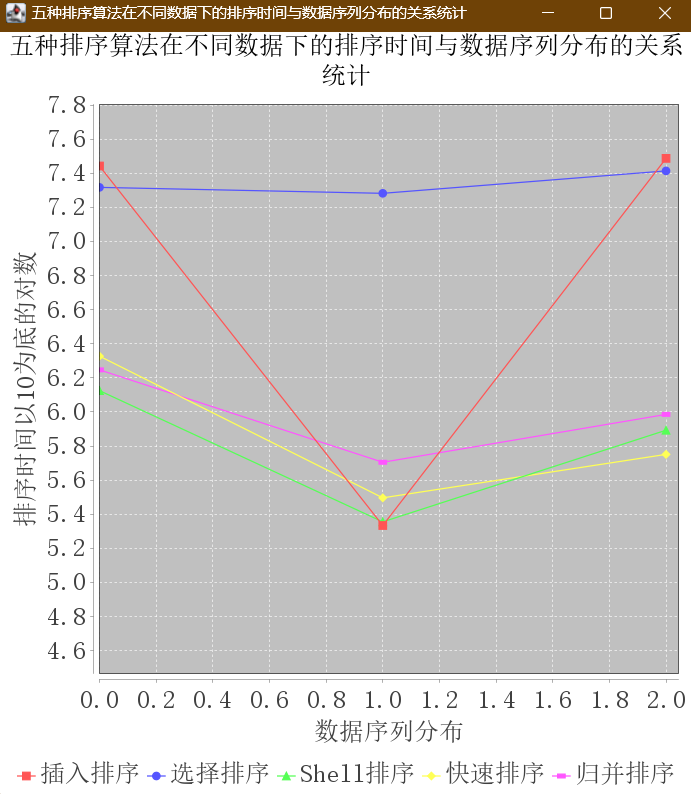
快速排序：

28619.8000 60460.2000 113880.2000 280979.6000 564120.2000 1329520.4000 3740479.8000 6480480.0000 17814080.0000

归并排序：

315540.2000 142339.8000 208640.0000 450419.4000 967700.0000 2632499.6000 6301220.4000 15282360.0000 19818280.2000

绘制图表：

结合任务二、任务三、任务四横向绘制图表（数据规模为2^12）：

其中最左侧是随机数据序列的数据（任务二）；中间是k-有序数据序列的数据（任务三，k=5）；右侧是不均匀数据序列的数据（任务四）。

##### 总结和收获

对于不均匀数据序列，其排序时间与数据规模的关系与随机数据序列基本一致。横向对比可以发现，Shell排序、快速排序、归并排序在不均匀数据序列的排序时间略低于随机数据序列，k-有序数据序列的排序时间更低；对于选择排序，其对三种数据序列进行排序的时间基本不变，这是因为不管何种数据序列，选择排序都会进行基本相同次数的比较和交换；对于插入排序，其所耗时间在k-有序数据序列显著降低，因为在基本有序的数据序列中，很多数据无需进行插入，节省了许多时间。

# 附录 源代码

SortTest.java

1. **package** homework;
3. **public** **class** SortTest {
4. // 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
5. **public** **static** **double** time(SortAlgorithm alg, Double[] numbers){
6. **double** start = System.nanoTime();
7. alg.sort(numbers);
8. **double** end = System.nanoTime();
9. **if**(alg.isSorted(numbers)) System.out.println("Sort success.");
10. **else** System.out.println("Sort failed.");
11. **return** end - start;
12. }
14. // 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
15. // 测试数据的生成交给test方法来完成，保证每次测试的数据都是新的数据
16. // 该方法重新修正了参数：一个参数是用来表示生成数据的分布模式，一个参数用来表示生成的数据的规模
17. **public** **static** **double** test(SortAlgorithm alg, **int** dataProbabilityType, **int** dataLength, **int** k, **int** T)
18. {
19. **double** totalTime = 0;
20. Double[] numbers = **null**;
21. **for**(**int** i = 0; i < T; i++) {
22. **switch**(dataProbabilityType){
23. **case** GenerateData.RANDOM -> numbers = GenerateData.getRandomData(dataLength);
24. **case** GenerateData.KSORTED -> numbers = GenerateData.getKSortedData(dataLength, k);
25. **case** GenerateData.UNEVEN -> numbers = GenerateData.getUnevenData(dataLength);
26. }
27. totalTime += time(alg, numbers);
28. }
29. **return** totalTime/T;
30. }
31. // 执行样例，仅供参考。
32. // 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。
33. **public** **static** **void** main(String[] args) {
34. **int**[] dataLength = **new** **int**[9];
35. **for**(**int** i=0;i<9;i++){
36. dataLength[i]=(**int**)Math.pow(2,i+8);
37. }
38. **double**[] elapsedTime = **new** **double**[dataLength.length];
39. SortAlgorithm alg = **new** Mergesort();
40. **for**(**int** i = 0; i < dataLength.length; i++)
41. elapsedTime[i] = test(alg, GenerateData.RANDOM, dataLength[i], 0, 5);
42. **for**(**double** time: elapsedTime)
43. System.out.printf("%6.4f ", time);
44. System.out.println();
45. **for**(**int** i = 0; i < dataLength.length; i++)
46. elapsedTime[i] = test(alg, GenerateData.KSORTED, dataLength[i], 5, 5);
47. **for**(**double** time: elapsedTime)
48. System.out.printf("%6.4f ", time);
49. System.out.println();
50. **for**(**int** i = 0; i < dataLength.length; i++)
51. elapsedTime[i] = test(alg, GenerateData.UNEVEN, dataLength[i], 0, 5);
52. **for**(**double** time: elapsedTime)
53. System.out.printf("%6.4f ", time);
54. System.out.println();
55. }
56. }

SortAlgorithm.java

1. **package** homework;
3. **public** **abstract** **class** SortAlgorithm {
4. **public** **abstract** **void** sort(Comparable[] objs);
5. **protected** **void** exchange(Comparable[] numbers, **int** i, **int** j){
6. Comparable temp;
7. temp = numbers[i];
8. numbers[i] = numbers[j];
9. numbers[j] = temp;
10. }
11. **protected** **boolean** less(Comparable one, Comparable other){
12. **return** one.compareTo(other) < 0;
13. }
14. **protected** **void** show(Comparable[] numbers){
15. **int** N = numbers.length;
16. **int** line = 0;
17. **for**(**int** i = 0; i < N; i++){
18. System.out.printf("%s ", numbers[i]);
19. line++;
20. **if**(line % 20 == 0) System.out.println();
21. }
22. System.out.println();
23. }
24. **protected** **boolean** isSorted(Comparable[] numbers){
25. **int** N = numbers.length;
26. **for**(**int** i = 0; i < N-1; i++)
27. **if**(numbers[i].compareTo(numbers[i+1]) > 0) **return** **false**;
28. **return** **true**;
29. }
31. }

GenerateData.java

1. **package** homework;
3. **import** java.util.Random;
5. **public** **class** GenerateData {
6. **public** **static** **final** **int** RANDOM = 1;
7. **public** **static** **final** **int** KSORTED = 2;
8. **public** **static** **final** **int** UNEVEN = 3;
9. // 生成一个长度为N的均匀分布的数据序列
10. **public** **static** Double[] getRandomData(**int** N){
11. Double[] numbers = **new** Double[N];
12. **for**(**int** i = 0; i < N; i++)
13. numbers[i] = Math.random();
14. **return** numbers;
15. }
16. // 生成一个K-sorted数据序列，其中该序列的长度为N，一般要求 K << N
17. **public** **static** Double[] getKSortedData(**int** N, **int** K){
18. Double[] numbers = **new** Double[N];
19. **double** step = 1.0/N;
20. numbers[0] = 0.0;
21. **for**(**int** i = 1; i < N; i++)
22. numbers[i] = numbers[i-1] + step;
23. **int** left = 0;
24. **int** right = left + K;
25. **while**(right < N){
26. shuffle(numbers, left, right);
27. left = right;
28. right += K;
29. }
30. shuffle(numbers, left, N);
31. **return** numbers;
32. }
33. //生成一个长度为N的不均匀数据序列，其中1/2的数据是0，1/4的数据是1，1/8的数据是2，1/8的数据是3
34. **public** **static** Double[] getUnevenData(**int** N) {
35. Double[] numbers = **new** Double[N];
36. **for** (**int** i = 0; i < N / 8; i++) {
37. numbers[i] = 3.0;
38. }
39. **for** (**int** i = N / 8; i < N / 4; i++) {
40. numbers[i] = 2.0;
41. }
42. **for** (**int** i = N / 4; i < N / 2; i++) {
43. numbers[i] = 1.0;
44. }
45. **for** (**int** i = N / 2; i < N; i++) {
46. numbers[i] = 0.0;
47. }
48. shuffle(numbers, 0, N);
49. **return** numbers;
50. }
51. // 将数组numbers中的[left,right)范围内的数据随机打乱
52. **private** **static** **void** shuffle(Double[] numbers, **int** left, **int** right){
53. **int** N = right - left;
54. Random rand = **new** Random();
55. **for**(**int** i = 0; i < N; i++){
56. **int** j = i + rand.nextInt(N-i);
57. exchange(numbers, i+left, j+left);
58. }
59. }
60. **private** **static** **void** exchange(Double[] numbers, **int** i, **int** j){
61. **double** temp = numbers[i];
62. numbers[i] = numbers[j];
63. numbers[j] = temp;
64. }
66. **public** **static** **void** main(String[] args) {
67. Double[] numbers = getUnevenData(100);
68. **for**(**int** i = 0; i < 100; i++)
69. System.out.printf("%5.3f ", numbers[i]);
70. }
71. }

Insertion.java

1. **package** homework;
3. **public** **class** Insertion **extends** SortAlgorithm{
4. **public** **void** sort(Comparable[] objs){
5. **int** N = objs.length;
6. **for**(**int** i = 1; i < N; i++){
7. **for**(**int** j = i; j > 0 && less(objs[j], objs[j-1]); j--)
8. exchange(objs, j, j-1);
9. }
10. }
11. }

Selection.java

1. **package** homework;
3. **public** **class** Selection **extends** SortAlgorithm {
4. **public** **void** sort(Comparable[] objs){
5. **int** N = objs.length;
6. **for** (**int** i = 0; i < N - 1; i++) {
7. **int** minIndex = i;
8. **for** (**int** j = N - 1; j > i; j--) {
9. **if** (less(objs[j], objs[minIndex])) {
10. minIndex = j;
11. }
12. }
13. exchange(objs, i, minIndex);
14. }
15. }
16. }

Shell.java

1. **package** homework;
3. **public** **class** Shell **extends** SortAlgorithm {
4. **public** **void** sort(Comparable[] objs) {
5. **for** (**int** i = objs.length / 3; i > 3; i /= 3) {//元素间距
6. **for** (**int** j = 0; j < i; j++) {
7. insertSort(objs, j, i);
8. }
9. }
10. insertSort(objs, 0, 1);
12. }
14. **private** **void** insertSort(Comparable[] objs, **int** start, **int** incr) {//对以start为始，incr为间距的子序列进行插入排序
15. **for** (**int** i = start + incr; i < objs.length; i += incr) {
16. **for** (**int** j = i; j > start && less(objs[j], objs[j - incr]); j -= incr) {
17. exchange(objs, j, j - incr);
18. }
19. }
20. }
21. }

QuickSort.java

1. **package** homework;
2. **public** **class** Quicksort **extends** SortAlgorithm {
3. **public** **void** sort(Comparable[] objs){
4. qsort(objs,0, objs.length-1);
5. }
6. **private** **void** qsort(Comparable[] objs,**int** i,**int** j){//对第i位到第j位的子序列进行快速排序
7. **int** pivotIndex = (i+j)/2;//取数组中间点为轴值
8. exchange(objs,pivotIndex,j);//将轴值放置在末尾，因为轴值所处的位置未确定
10. **int** m = i-1, n = j;
11. **do**{//从两端移动下标直至相遇
12. **while**(less(objs[++m], objs[j]));
13. **while**(n!=0&&less(objs[j], objs[--n]));//每次循环中，在两端分别寻找两侧小于和大于轴值的数
14. exchange(objs,m,n);
15. }**while**(m<n);
16. exchange(objs,m,n);//最后一次循环时多换了一次
17. exchange(objs,m,j);//将轴值放置在适当位置
19. **if**(j-(m+1)>0) qsort(objs,m+1,j);//m+1位到j位是新的右侧子序列
20. **if**(n-i>0) qsort(objs,i,n);//i位到n位是新的左侧子序列
21. }
22. }

MergeSort.java

1. **package** homework;
3. **public** **class** Mergesort **extends** SortAlgorithm {
4. **public** **void** sort(Comparable[] objs){
5. mergesort(objs,**new** Comparable[objs.length],0,objs.length-1);
6. }
7. **private** **void** mergesort(Comparable[] objs,Comparable[] temp,**int** i,**int** j){
8. **if**(i==j) **return**;
9. **int** midIndex = (i+j)/2;
10. mergesort(objs,temp,i,midIndex);
11. mergesort(objs,temp,midIndex+1,j);
12. System.arraycopy(objs, i, temp, i, j + 1 - i);
13. **int** n1=i,n2=midIndex+1;//左侧数组和右侧数组的位置指示
14. **for**(**int** index=i;index<=j;index++) {//将左右侧数组按大小复制进原数组
15. **if** (n1 == midIndex + 1) {//左侧数组已复制完
16. objs[index] = temp[n2++];
17. } **else** **if** (n2 == j + 1) {//右侧数组已复制完
18. objs[index] = temp[n1++];
19. } **else** **if** (less(temp[n1], temp[n2])) {//左侧数组首位较小
20. objs[index] = temp[n1++];
21. } **else** {//右侧数组首位较小
22. objs[index] = temp[n2++];
23. }
24. }
25. }
26. }

LineXYDemo.java

1. **package** homework;
3. **import** org.jfree.chart.ChartFactory;
4. **import** org.jfree.chart.ChartPanel;
5. **import** org.jfree.chart.JFreeChart;
6. **import** org.jfree.chart.axis.CategoryAxis;
7. **import** org.jfree.chart.axis.NumberAxis;
8. **import** org.jfree.chart.axis.ValueAxis;
9. **import** org.jfree.chart.plot.PlotOrientation;
10. **import** org.jfree.chart.plot.XYPlot;
11. **import** org.jfree.chart.renderer.xy.XYLineAndShapeRenderer;
12. **import** org.jfree.chart.title.TextTitle;
13. **import** org.jfree.chart.ui.ApplicationFrame;
14. **import** org.jfree.chart.ui.RectangleInsets;
15. **import** org.jfree.data.xy.XYDataset;
16. **import** org.jfree.data.xy.XYSeries;
17. **import** org.jfree.data.xy.XYSeriesCollection;
19. **import** java.awt.\*;
21. **public** **class** LineXYDemo **extends** ApplicationFrame {
22. // 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
23. **public** LineXYDemo(String title){
24. **super**(title);
25. XYDataset dataset = createDataset();             // 创建记录图中坐标点的数据集
26. 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
27. 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
28. // 设置GUI面板Panel的显示大小
29. chartPanel.setPreferredSize(**new** java.awt.Dimension(500, 270));
30. setContentPane(chartPanel);                      // 这是JavaGUI的步骤之一，不用过于关心
31. }
33. **private** JFreeChart createChart(XYDataset dataset) {
34. // 使用已经创建好的dataset生成图表对象
35. // 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
36. JFreeChart chart = ChartFactory.createXYLineChart(
37. "五种排序算法在不同数据下的排序时间与数据序列分布的关系统计",      // 图表的标题
38. "数据序列分布",                           // 横轴的标题名
39. "排序时间以10为底的对数",                           // 纵轴的标题名
40. dataset,                       // 图表对象中使用的数据集对象
41. PlotOrientation.VERTICAL,      // 图表显示的方向
42. **true**,                          // 是否显示图例
43. **false**,                         // 是否需要生成tooltips
44. **false**                          // 是否需要生成urls
45. );
46. // 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
47. // 同学们不必在意下面的代码
48. // 可以将下面的代码去掉对比一下显示的不同效果
50. chart.setBackgroundPaint(Color.WHITE);
51. XYPlot plot = (XYPlot)chart.getPlot();
53. //设置字体避免中文乱码
54. Font f = **new** Font("宋体",Font.PLAIN,20);
55. plot.getRangeAxis().setLabelFont(f);
56. plot.getRangeAxis().setTickLabelFont(f);
57. plot.getDomainAxis().setLabelFont(f);
58. plot.getDomainAxis().setTickLabelFont(f);
59. chart.getTitle().setFont(f);
60. chart.getLegend().setItemFont(f);
62. plot.setBackgroundPaint(Color.lightGray);
63. plot.setAxisOffset(**new** RectangleInsets(5.0, 5.0, 5.0, 6.0));
64. plot.setDomainGridlinePaint(Color.WHITE);
65. plot.setRangeGridlinePaint(Color.WHITE);
66. XYLineAndShapeRenderer renderer = (XYLineAndShapeRenderer) plot.getRenderer();
67. renderer.setDefaultShapesVisible(**true**);
68. renderer.setDefaultShapesFilled(**true**);
69. **return** chart;
70. }
72. **private** XYDataset createDataset() {
73. // 本样例中想要显示的是三组数据的变化图
74. // 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
75. **double**[] Y1 = {7.44228,5.334412,7.486792};
76. **double**[] Y2 = {7.317098,7.282192,7.413748};
77. **double**[] Y3 = {6.125162,5.35572,5.893074};
78. **double**[] Y4 = {6.327433,5.496569,5.751372};
79. **double**[] Y5 = {6.24705,5.705179,5.985741};
80. **double**[] X = {0.0,1.0,2.0};
81. **double**[][] Y = {Y1, Y2, Y3, Y4, Y5};
82. // 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
83. XYSeries[] series = {**new** XYSeries("插入排序"), **new** XYSeries("选择排序"), **new** XYSeries("Shell排序"),**new** XYSeries("快速排序"),**new** XYSeries("归并排序")};
84. **int** N = X.length;
85. **int** M = series.length;
86. **for**(**int** i = 0; i < M; i++)
87. **for**(**int** j = 0; j < N; j++)
88. series[i].add(X[j], Y[i][j]);
89. // 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
90. XYSeriesCollection dataset = **new** XYSeriesCollection();
91. **for**(**int** i = 0; i < M; i++)
92. dataset.addSeries(series[i]);
94. **return** dataset;
95. }
97. **public** **static** **void** main(String[] args) {
98. LineXYDemo demo = **new** LineXYDemo("五种排序算法在不同数据下的排序时间与数据序列分布的关系统计");
99. demo.pack();
100. demo.setVisible(**true**);
101. }
102. }