Lab 7 Algorithm Analysis and Sorting

Goal

In this lab you will experiment with timing various sorting methods, writing your own bubble sort method, writing a compareTo method for a BankAccount class, and sorting a file of BankAccount records by both account number and balance.

Resources

- Chapters 4, 8 and (optional) 9

Introduction

Sorting data is essential to processing data in that it usually speeds up further processing such as searching for items in our data. Often sorting IS the processing we wish to perform, for example, ranking a number of items by their value;

Directed Lab Work

Part A Bench Testing Algorithm Efficiency (Speed)

To begin, we will spend some time comparing execution speeds of the 5 different sorting algorithms covered in chapters 8 and 9. The lab7.java file has code that manages generating and scrambling arrays of int or Integer for testing the algorithms in SortArray.java.

Step 1) Run this program and observe the output. The program currently just tests selectionSort using an array of 30 items. Read through the file and notice how the StopWatch object sw is being used to time the sort. Also note how we are declaring both int and Integer arrays for testing the algorithms (some require Generic objects, while others require primitive int). There are several helper methods after main for printing and scrambling arrays of int or Integer.

Step 2) Add statements to reset the copyArray back to sourceArray and time a second sort using insertionSort. Print the array before and after to verify the sort is working, and show the execution time. You'll notice that the execution speed for insertionSort is 0! We will need to increase SIZE later but leave it at 30 for now while we continue to verify the sorts are working properly on a small collection of data.

Step 3) Both selection and insertion sorts are designed to sort int arrays. The remaining sorts are defined to accept objects of any generic class that provides a compareTo method (note that class Object does not define this method so arrays of Object cannot be sorted.) This is why sourceArray2 and copyArray2 were defined at the top of main. Add statements now to perform, verify, and time a third sort using Shell sort on copyArray2.

Step 4) Reset copyArray2 and perform, verify and time a forth sort, Merge sort on copyArray2.

Step 5) Reset copyArray2 and perform, verify and time a fifth sort, Quick sort on copyArray2.

Step 6) Now that you've proven the sorts work properly, we can increase the SIZE of the arrays and record execution speed for all algorithms as SIZE increases. Open the spreadsheet benchTest.xls in the lab7 folder and fill in the execution times for the 5 algorithms as SIZE changes from 1000 to 1,000,000. Then plot the 5 curves that result.

To plot the curves, select the table by clicking on SIZE and dragging across the entire table. Choose Insert>Scatter>Scatter with Smooth Lines and Markers and place the chart next to the table in the spreadsheet.

Which is the best algorithm? Is one of them the winner for all values of SIZE? Why do you think that is?

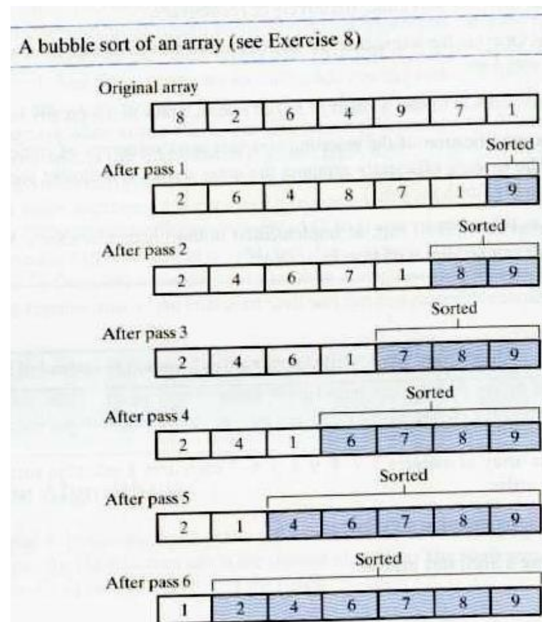
Part B) Writing your own Selection/Bubble sort and Sorting arrays of objects of Comparable<T>

Writing Selection Sort and Bubble Sort Methods

Step 7) Write pseudocode for a selection sort algorithm that selects the largest, instead of the smallest, entry in an array of integers and sorts the array into descending order. Then make a new version of the selectionSort method in SortArray, called selectionSortDescending, so that it implements your algorithm. Test your new sort method.

Pseudocode:

Step 8) A **bubble sort** can sort an array of *n* entries into ascending order by making $n - 1$ passes through the array. On each pass, it compares adjacent entries and swaps them if they are out of order. For example, on the first pass, it compares the first and second entries, then the second and third entries, and so on. At the end of the first pass, the largest entry is in its proper position at the end of the array. We say that it has bubbled to its correct spot. Each subsequent pass ignores the entries at the end of the array, since they are sorted and are larger than any of the remaining entries. Thus, each pass makes one fewer comparison than the previous pass. See the diagram below for a visual example of a bubble sort. Implement the bubble sort so it sorts an array of generic objects



A bubble sort of an array (see Exercise 8)

You can look at Shell sort to see how sorting generic arrays is handled. Perform time trials on your bubbleSort algorithm (until it takes about 10 seconds or longer to run) and add the statistics to your spreadsheet.

Sorting Objects of a User Defined Class

Step 9) Sort a file of BankAccount data using your bubble sort algorithm. To do this you will need to define a compareTo method for the BankAccount class and state that it implements the Comparable<T> interface (Read segment D.18 – D.23 starting on page D.12). Start by reading the code in class BankAccount, BankApp and BankReadFile, which reads a file of bank account data. Try running BankApp and BankReadFile, noticing the data is read from file bankData.txt. Then, add statements to the end of BankReadFile that sorts the array of accounts a) by account number and b) by balance.

Sorting by different fields of an Object (a non-standard way)

Sometimes we might want to sort by a BankAccount's account number, and other times we might want to sort by balance. The **standard** Java way to do this is to create a private class that implements **Comparator**, which isn't hard, but requires another class added to your existing class file. Another, perhaps clever, way to handle multiple ways to compare BankAccount data is to add a private **static** member `compareType` to the `BankAccount` class, and some **public** static constants representing the two options for `compare`. In other words (add the lines in italics):

```
public class BankAccount implements Comparable<BankAccount>
{
      public static final int COMPARE_ACCOUNT = 0;  // value to compare by account
      public static final int COMPARE_BALANCE = 1;  // value to compare by balance

      private static int compareType = COMPARE_ACCOUNT; // sets default way all
                                                        // BankAccount objects are compared
      private String account;  // the account number
      private double balance;  // the balance associated with account

      ...
}
```

You would also then set up a static method `compareBy` inside the `BankAccount` class that allows the comparison field to be changed:

```
public static void compareBy(int  compare)
{
      compareType = compare;
}
```

To change the sorting field for all BankAccount objects to compare by `balance` you would then say:

```
      BankAccount.compareBy(BankAccount.COMPARE_BALANCE);
```

And of course your `compareTo` method would then have to have an if statement added to it to check before comparing which field it is supposed to compare. If the `compareType` is `COMPARE_BALANCE` it will

```
      return  (int)(100*balance - 100*other.balance);     // turn double difference *100 into int
```

but if the `compareType` is `COMPARE_ACCOUNT` it will

```
      return  account.compareTo(other.account);     // pass the buck to the String compareTo method
```

This is a good example of the use of static variables because it shows how they are associated with the entire class of objects, not an individual instance of an object.

After implementing and testing your compareTo and compareBy methods, call your bubbleSort method to sort the accounts and display them in sorted order, first by account, then by balance.

Another possibility: you could also add another static variable and extra static mehods to also be able to switch between ascending and descending comparisons!

Post-Lab Visualization

1) Show the contents of the array of integers 5 7 4 9 8 5 6 3 for each pass of a selection sort while sorting the array into ascending order.

2) Show the contents of the array of integers 5 7 4 9 8 5 6 3 for each pass of an insertion sort while sorting the array into ascending order.