CS 100: Programming Assignment P5

Due: Thursday, April 8, 5pm, Carpenter Lab (or in lecture)

You may work in pairs. Do not submit your assignment for grading unless you have read and understand the CS100 webpage on Academic Integrity. Follow the course rules for the submission of assignments or lose points.

Background

Suppose we have an array of integers

30	50	20	10	40

and that we wish to permute its values so that they range from smallest to largest:

10	20	30	40	50

This is the problem of *sorting*. BubbleSort is one of many possible sorting methods and it works by repeatedly comparing adjacent entries in the array and swapping their contents if they are out of order. The search for out-of-order pairs is broken down into *passes*. Here is what happens to our example during the first pass:

30	50	20	10	40	>> No Swap >>	30	50	20	10	40
30	50	20	10	40	>> Swap >>	30	20	50	10	40
30	20	50	10	40	>> Swap >>	30	20	10	50	40
30	20	10	50	40	>> Swap >>	30	20	10	40	50

Let `a` refer to the array. For *j* = 0, 1, 2, and 3 in the example we compare `a[j]` and `a[j+1]` and swap their values if necessary. Clearly a simple `for`-loop can oversee the process. Note that regardless of the initial distribution of values in the array, we are guaranteed that after the first pass the largest array value is in its "final resting place". This means that we can live with one fewer comparison in the second pass:

30	20	10	40	50	>> Swap >>	20	30	10	40	50
20	30	10	40	50	>> Swap >>	20	10	30	40	50
20	10	30	40	50	>> No Swap >>	20	10	30	40	50

During the second pass we compare (and possibly swap) the values in `a[j]` and `a[j+1]` for *j* = 0, 1, and 2. Again we make a general observation: after the second pass, the two largest values in the array are in their final position. Proceeding on to the third pass:

20	10	30	40	50	>> Swap >>	10	20	30	40	50
10	20	30	40	50	>> No Swap >>	10	20	30	40	50

During the third pass we compare (and possibly swap) the values in `a[j]` and `a[j+1]` for j = 0 and 1. After the second pass, the three largest values in the array are in their final position. Even though in this example we know that the array is now sorted, we proceed on to the fourth and final pass:

10	20	30	40	50	>> No Swap >>	10	20	30	40	50

We draw some "general n" conclusions from our n = 5 example:

1. After k passes array entries `a[n-k]`, `a[n-k+1]`,…,`a[n-1]` are sorted and do not have to be "looked at" again because they already house the k largest values in the array.
2. The array is guaranteed to be sorted after n-1 passes.
3. If no swaps are performed during a pass, then the array is sorted and there is no need to continue.
4.

In case you were wondering, none of these conclusions depend on the array having distinct values as was the case in our example.

In Part A you implement a BubbleSort method for integers and for strings. This part of the assignment is designed to give you practice with array subscripting and nested loops. In Part B you compare Bubblesort with MergeSort, a much faster sorting method. You will run some experiments, gather data on performance, and plot the results. This will give you further practice with arrays and will deepen your appreciation about "running time" issues. Parts C and D are motivated by Part A. Your implementations of integer BubbleSort and string BubbleSort are so similar that you should wonder why we cannot just write one general BubbleSort method that can then sort *any* sortable array. Indeed, Java does provide a framework for doing this and by stepping you through the issues you will come to appreciate more the "object oriented" approach to programming.

Part A. Bubble Sort on Integer and String Arrays (5 Correctness + 2 Style)

Get a copy of `P5A.java` from the website and in the class `P5A` implement the following method:

```
// Permutes the values in data so that they are arranged from smallest to largest.
// Returns the number of required comparisons.
public static int bubbleSort(int[] data)
```

The overall process should be under the control of a `while`-loop that terminates upon completion of the n-1st pass or upon completion of a pass that involves no swaps. (See points 2 and 3 above.) Note that the body of the `while` loop will contain a `for`-loop that scans the array swapping adjacent entries that are out-of-order. Thus, BubbleSort involves a nested loop construction.

Count only comparisons between array elements. Ignore the comparisons that are associated with the loops as they check for termination, i.e., `k<a.length`. You will see from `P5A.java` where to place your code. Set your Java target to `P5A` and test your implementation on various random arrays to make sure it works. You can use the method `randomInt` that we provide to produce a random integer array of a given length. If you comment out (put `//` in front of) the print lines then you can experiment with very large arrays. Depending on the speed of your machine you should be able to sort an array of length 10,000 using about 50,000,000 comparisons in a few seconds.

Once you have `bubbleSort` working, make a copy a version of it that it is able to sort arrays of strings:

```
// Permutes the values in data so that they are arranged in lexicographic order.
// Returns the number of required comparisons.
public static int bubbleSort(String[] data)
```

Lexicographic order means "dictionary order". Thus, "`def`" is lexicographically greater than "`abc`" because "`def`" would comes after "`abc`" in a dictionary. Use the `String` method `compareTo` for lexicographic testing. If `s` and `t` are strings, then `s.compareTo(t)` returns an integer less than, equal to, or greater than 0 according as `s` is lexicographically less than, equal to, or greater than `t`, respectively. Try out your string version of `bubbleSort` on the string array provided in `P5A.java.`

The production of the string version of BubbleSort is largely a cut-and-paste exercise as only a few changes need to be made, e.g., the type annotations and the comparison.

Submit a listing of your final version of `P5A.java` (showing your two BubbleSort implementations) and a copy of the output produced when it is run. The `main` method that we provide must be used. It will confirm that both implementations are doing the required sort and that they are correctly counting the number of comparisons.

Part B. Comparing BubbleSort with MergeSort (5 Correctness + 2 Style)

When analyzing the performance of a sorting method, it is customary to talk about the number of comparisons that are required. In the $n = 5$ example of BubbleSort that we presented in Part A, 4+3+2+1 = 10 comparisons are required. For general n, BubbleSort requires

$$(n-1) + (n-2) + \ldots + 3 + 2 + 1 = n(n-1)/2$$

comparisons. We say that BubbleSort is an $O(n^2)$ method (pronounced order n-squared method) because the number of comparisons grows with the square of the input array length n. That is to say, if we multiply n by 10 then we can expect running time to increase by 100.

Other sorting methods involve many fewer comparisons than BubbleSort. MergeSort is one example. It works by repeatedly merging pairs of sorted subarrays into larger sorted subarrays. Here is how it proceeds on an n =16 example:

J	H	E	B	P	D	I	K	A	N	F	O	C	L	G	M

(An array of strings with the quote delimiters deleted for clarity.) It starts by looking at 8 subproblems:

J	H	E	B	P	D	I	K	A	N	F	O	C	L	G	M

Each subproblem involves the merging of two length-1 sorted arrays into a single length-2 sorted array:

H	J	B	E	D	P	I	K	A	N	F	O	C	L	G	M

These are then paired giving 4 subproblems at the "next level":

H	J	B	E	D	P	I	K	A	N	F	O	C	L	G	M

The subproblems here involve merging a pair of sorted length-2 arrays into a single sorted length 4 array:

B	E	H	J	D	K	I	P	A	F	N	O	C	G	L	M

These are paired to give 2 subproblems at the next level:

B	E	H	J	D	K	I	P	A	F	N	O	C	G	L	M

These subproblems involve the merging of a pair of sorted length-4 arrays into a single sorted length-8 array:

B	D	E	H	I	J	K	P	A	C	F	G	L	M	N	O

At the final level we pair these two subarrays

B	D	E	H	I	J	K	P	A	C	F	G	L	M	N	O

and merge them to produce a single, length-16 sorted array:

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P

At most m-1 comparisons are required to merge two length-m sorted arrays. Thus, in our example 8*1 + 4*3 + 2*7 + 1*15 = 49 comparisons are required. Note that BubbleSort would require 15*16/2 = 120 comparisons.

For larger n the difference is more dramatic. MergeSort is an O(n log n) method. For $n = 10^6$, this means that BubbleSort would be slower by a factor of around 100,000, the approximate ratio of one million to its logarithm. In this part of the assignment you produce tables and plots that confirm the advantage of MergeSort over BubbleSort.

Start by getting copies of the files `P5B.java` and `Plot.java` from the website. Include `P5B.`java (listing attached) and `Plot.java` in your project. (Keep `P5A.java` in the project as we'll need access to `P5A.random`.) Cut and paste your integer `bubbleSort` method from `P5A.java` into `P5B.java` alongside the `mergeSort` method that is provided. (We have implemented a recursive version of MergeSort and it will be discussed in lecture. In this assignment you simply use `MergeSort`. Understanding how the underlying recursive process works is not necessary at this time.) Set your Java target to `P5B` and run it to make sure your settings are correct. A test plot should appear.

Rewrite `P5B.main` to do the following. For each length 200, 500, 1000, 1500, 2000, 2500, 3333, 5000, 6400, 8000, 9999, create a random integer array of that length and sort it using both `bubbleSort` and `mergeSort.` These lengths are predefined for you in the integer array `sizes`. Use the `randomInt` method in the class `P5A` to create the random integer arrays. Make a copy of each array before you sort it, because you must run `bubbleSort` and `mergeSort` on the same array. Do this by using the procedure `copy` that is implemented for you in the class `P5B`.

As you sort these arrays, you must keep track of the number of comparisons used by `bubbleSort` and `mergeSort` for each length. Do this by creating a pair of integer arrays `bubbleSortComps` and `mergeSortComps`. After you run `bubbleSort` on an array of length `sizes[i]`, record in `bubbleSortComps[i]` the number of required comparisons. Likewise, after you run `mergeSort` on an array of length `sizes[i],` record in `mergeSortComps[i]` the number of required comparisons.

Print a 3-column table that displays the value of `sizes[i]`, `bubbleSortComps[i],` and `mergeSortComps[i]` for i = 0 through `sizes.length-1`. The table should be "nice looking" with appropriate headings.

Finally, use the `Plot` class that we provide to display the comparison counts. The use of this class is amply illustrated by this fragment that is in the given `P5B` template:

```
int[] sizes = { 200, 500,1000,1500,2000,2500,3333,5000,6400,8000, 9999};
int[] y     = {1200,1500,2000,2500,3000,3500,4333,6000,7400,9000,10999};
int[] z     = {5000,   0,6000,1000,7000,2000,8000,3000,9000,4000, 9999};
new Plot("Test",sizes,y,z);
```

Your `P5B.main` should produce two plots via the statements

```
new Plot("BubbleSort Results",sizes, bubbleSortComps, q1);
new Plot("MergeSortResults",sizes, mergeSortComps,  q2);
```

where the `q1` and `q2` arrays are defined by

```
q1[i] = sizes[i]*size[i]

q2[i] = sizes[i]*(int)Math.log(sizes[i]).
```

for `i=0` through `sizes.length-1.` With these plots you'll see how well n^2 and n log n "track" the number of comparisons required when `bubbleSort` and `mergeSort` are applied to a length n sorting problem.

Submit a listing of `P5B.main`, copies of the two plots, and the table..

Part C. Object Oriented Sorting (2 Correctness + 1 Style)

You have probably noted that it was a waste to have written essentially the same code for `bubbleSort` twice, once for strings and once for integers. If we had wanted to sort strings using `mergeSort`, we would have had to copy our integer `mergeSort` routine and modify it in the same way we modified `bubbleSort.` It would be nice to have one version of `mergeSort` and one version of `bubbleSort` that would work for both integers and strings (or for that matter any other comparable objects that we might wish to sort, such as records in a database).

Java provides a convenient way of doing this. The class `Sortable` in the file `Sortable.java` encapsulates an array of objects to be sorted (the array `data`) and provides a general version of `mergeSort` that will work for both arrays and strings. Add `P5C.java`, `Sortable.java`, and `Comparable.java` to your project. Set your Java target to `P5C` and run it to make sure your settings are correct. Here is a excerpt from the class `Sortable.java`:

```
public class Sortable
{
  Comparable[] data;     // the array of objects to be sorted
  int comps = 0;         //  number of comparisons needed to sort this array

  //The constructor creates a new instance of Sortable with the given data array
  Sortable(Comparable[] array)
  {
     data = array;
  }

  Comparable temp[];    // Temporary storage for mergeSort

  // Permutes the values in the array so that they are arranged from smallest to largest.
  // Returns the number of required comparisons.
  public int mergeSort()
  {
  }

  // Permutes the values in elements p through q-1 of the array so that they are arranged
  // from smallest to largest. Returns the number of required comparisons.
  public int mergeSort(int p, int q) {
         :
      while (i < mid && j < q)
      {
         if (data[i].leq(data[j]))   temp[k++] = data[i++];
         else                        temp[k++] = data[j++];
         comps++; //tally this comparison
      }
         :
   }
```

The method `Sortable.mergeSort` implements exactly the same algorithm as `P5B.mergeSort`, except that it is now an instance method and it operates on arrays of type `Comparable.` Also note that the required comparisons are carried out in terms of what looks like a "comparison method" `leq.` (Think of `leq` as "less than or equal to").

To summarize the changes, in `P5B` we have

```
public static int[] temp; // Temporary storage for mergesort
```

while in `Sortable` it is

```
Comparable temp[];    // Temporary storage for mergeSort
```

In `P5B` the sorting methods are static:

```
// Permutes the values in the array so that they are arranged from smallest to largest.
// Returns the number of required comparisons.
public static int mergeSort(int[] data)
{…}

// Permutes the values in elements p through q-1 of the array so that they are arranged
// from smallest to largest. Returns the number of required comparisons.
public static int mergeSort(int[] data, int p, int q)
{…}
```

while in `Sortable` they are instance methods that operate on the instance array `data:`

```
// Permutes the values in the array so that they are arranged from smallest to largest.
// Returns the number of required comparisons.
public int mergeSort()
{…}

// Permutes the values in elements p through q-1 of the array so that they are arranged
// from smallest to largest. Returns the number of required comparisons.
public int mergeSort(int p, int q)
{…}
```

Finally, in `P5B` we compare array entries with the usual <= operator

```
if (data[i] <= data[j])     temp[k++] = data[i++];
else                        temp[k++] = data[j++];
```

whereas in `Sortable` it is

```
if (data[i].leq(data[j]))   temp[k++] = data[i++];
else                        temp[k++] = data[j++];
```

In this part of the assignment we'd like you to "generalize" your integer `P5B.bubbleSort` in exactly the same way. Cut and paste it into the `Sortable` class in `Sortable.java` alongside the `mergeSort method` provided. Modify it appropriately so that it can be used to sort any array of `Comparable` objects. Model your changes on the `given mergeSort` implementation.

To be sorted by these routines, objects must be "wrapped" in a class that implements the `Comparable` "interface". This interface forces the class to provide a procedure `leq` that defines what it means for one object to be "less than or equal to" another. The interface is defined in the file `Comparable.java` (attached) along with two implementations, one for integers and one for strings.

This approach allows a sorting routine to sort very different types of objects in a uniform way. Instead of writing different versions for integers, strings, etc., which are compared using very different comparison operators, we can write the sorting routine once and for all in terms of the abstract comparison operator `leq.` The sorting routine knows nothing about how `leq` is implemented. This is determined by the objects that are being sorted.

Read through the code carefully and think about the relationship between the `Sortable` class and the classes `Int` and `LexStr` that implement the `Comparable` interface. Note that the interface name `Comparable` can be used as a type. Nothing in `Sortable` ever refers to `Int` or `LexStr`, but only to `Comparable`.

Before we can sort a `String` or `int` array, the array must be converted. For example, a `String` array must be converted to a `LexStr` array, and a new instance of `Sortable` must be created with that array as `data`. The conversion from `String[]` to `LexStr[]` is handled by the static procedure `LexStr.convert`. The `LexStr[]` object can then be passed to the constructor of `Sortable`, e.g.,

```
 // Create an array of strings to sort.
String[] s1 = {"the","quick","brown","fox","jumped","over","the","lazy","dog"};
// Convert it a sortable object referenced by ls1.
Sortable ls1 = new Sortable(LexStr.convert(s1));
ls1.print();
// Sort the object referenced by ls1 and count comparisons.
comps = ls1.mergeSort();
// Print results.
System.out.println("Merge Sort Result:");
ls1.print();
System.out.println("Number of comparisons = " + comps);
```

This is an excerpt from `P5C.java`. Run this program and submit the output along with a listing of your `bubbleSort` implementation in the class `Sortable`.

Part D. A New Comparable Class (2 Correctness + 1 Style)

Now suppose we want to sort strings according to a different order, which we will call *modified lexicographic order*. In this order, a string *x* is less than or equal to another string *y* if either

- *x* is shorter than *y*, or
- *x* and *y* are the same length, and *x* is lexicographically less than or equal to *y*.

Thus we compare the strings by length first; if the lengths are different, then the shorter string is always the smaller in modified lex order. If the lengths are the same, then we compare them lexicographically. For example, if we were to sort

`the quick brown fox jumped over the lazy dog`

according to ordinary lex order, the result would be

`brown dog fox jumped lazy over quick the the`,

whereas if we were to sort them according to modified lex order, the result would be

`dog fox the the lazy over brown quick jumped.`

Define a new class `ModLexStr` for comparing strings according to modified lexicographic order. Your class should implement the `Comparable` interface and should be included in the file `Comparable.java`. Use the class `LexStr` in the same file for guidance as to what to do. Test your implementation by running `P5D.java.` If you did this right, you should not have to change any code in `Sortable.`

Submit a listing of the `class ModLexStr` and the output associated with `P5D.java.`

The File P5B.java:

```
import java.io.*;
import java.awt.*;

public class P5B {
   public static void main(String args[])
   {
      int[] sizes = { 200, 500,1000,1500,2000,2500,3333,5000,6400,8000, 9999};

      // After you run the template, delete the next three lines and develop the
      // required main method.
      int[] y     = {1200,1500,2000,2500,3000,3500,4333,6000,7400,9000,10999};
      int[] z     = {5000,   0,6000,1000,7000,2000,8000,3000,9000,4000, 9999};
      new Plot("Test",sizes,y,z);
   }

   // Place your bubbleSort method from P5A.java here:

   public static int[] temp; //temporary storage for mergesort
   // Permutes the values in the array so that they are arranged from smallest to largest.
   // Returns the number of required comparisons.
   public static int mergeSort(int[] data)
   {
      //allocate temporary storage array
      temp = new int[data.length];
      //call recursive mergeSort procedure on whole array
      return mergeSort(data,0,data.length);
   }

   // Permutes the values in elements p through q-1 of the array so that they are arranged
   // from smallest to largest. Returns the number of required comparisons.
   public static int mergeSort(int[] data, int p, int q)
   {
      int comps = 0;
      //if length is 0 or 1, nothing to do
      if (q - p < 2) return 0;

      //otherwise split into two subarrays of roughly equal length
      //and sort them recursively
      int mid = (p + q)/2;
      comps = comps + mergeSort(data,p,mid);
      comps = comps + mergeSort(data,mid,q);
      //merge the two sorted subarrays
      int i = p;                //index into first subarray
      int j = mid;   //index into last subarray
      int k = p;                //index into temp array
      while ((i < mid) && (j < q)) {
         if (data[i] <= data[j])          temp[k++] = data[i++];
         else                              temp[k++] = data[j++];
         comps++; //tally this comparison
      }
      while (i < mid)          temp[k++] = data[i++];
      while (j < q)  temp[k++] = data[j++];

      //copy sorted subarray back to original array
      for (i = p; i < q; i++)
      {
         data[i] = temp[i];
      }
      return comps;
   }

   // Yields a reference to a second copy of the array.
   public static int[] copy(int[] a)
   {
      int[] c = new int[a.length];
      int i;
      for (i = 0; i < a.length; i++)
      {
         c[i] = a[i];
      }
      return c;
   }
}
```

The File Comparable.java

```
import java.util.*;

interface Comparable {
          boolean leq(Comparable q);
          String toString();
}


// For comparing strings lexicographically
public class LexStr implements Comparable
{
   String value; // The String value of this LexStr object

   // The constructor.
   LexStr(String v) {
      value = v;
   }

   //  Yields true if this string "equals or comes before" q.
   public boolean leq(Comparable q)
   {
      String qvalue = ((LexStr)q).value;
      //compare the strings value and qvalue lexicographically
      //compareTo is defined in the String class
      return value.compareTo(qvalue) <= 0;
   }

    // Yields the value of this object as a string.
    public String toString() {
       return value;
    }

    // Convert a String array to a LexStr array.
    public static LexStr[] convert(String[] s)
    {
       int i;
       LexStr[] ls = new LexStr[s.length];
       for (i = 0; i < s.length; i++) ls[i] = new LexStr(s[i]);
       return ls;
    }
}

// For comparing integers
public class Int implements Comparable {
   int value; //the integer value of this Int object

   // The Constructor.
   Int(int v) {
      value = v;
   }

   //  Yields true if this integer is less than or equal to q.
   public boolean leq(Comparable q)
   {
      return value <= ((Int)q).value;
   }

   // Yields the value of this object as a string.
   public String toString()
   {
      return String.valueOf(value);
   }

   // Convert an integer array to an Int array
   public static Int[] convert(int[] s)
   {
      int i;
      Int[] ls = new Int[s.length];
      for (i = 0; i < s.length; i++) ls[i] = new Int(s[i]);
      return ls;
   }
}
```