Assignment 4(Total 100)

**Problem 1(Total points 50)** When evaluating sorting algorithms, there many factors that might affect how fast an algorithm will run when it is programmed for a real machine: CPU speed, language used, type of data being sorted, type of operating system, and the number of concurrent processes running at the same time. For this reason, comparing actual running times of an algorithm is ineffectual for making judgments about how good the algorithm might be. To effectively make a judgment that is free of real world constraints, computer scientists, when comparing sorting algorithms, often simply count the number of comparisons an algorithm makes.

In this exercise, we will add a counter to the MergeSorter program to help us measure the number of comparisons of array elements being made while the routine is completing the sort.

The MergeSorter code is listed below.

private static void merge(int[] first, int[] second, int[] a)

{

int iFirst = 0; // Next element to consider in the first array

int iSecond = 0; // Next element to consider in the second array

int j = 0; // Next open position in a

// As long as neither iFirst nor iSecond is past the end, move

// the smaller element into a

while (iFirst < first.length && iSecond < second.length)

{

if (first[iFirst] < second[iSecond])

{

a[j] = first[iFirst];

iFirst++;

}

else

{

a[j] = second[iSecond];

iSecond++;

}

j++;

}

// Note that only one of the two loops below copies entries

// Copy any remaining entries of the first array

while (iFirst < first.length)

{

a[j] = first[iFirst];

iFirst++; j++;

}

// Copy any remaining entries of the second half

while (iSecond < second.length)

{

a[j] = second[iSecond];

iSecond++; j++;

}

}

The behavior we want to monitor occurs in this section of code,

if (first[iFirst] < second[iSecond])

{

a[j] = first[iFirst];

iFirst++;

}

else

{

a[j] = second[iSecond];

iSecond++;

}

It is in the above section that we compare array elements and decide which element becomes part of our merged and sorted array. By counting the number of comparisons here, we obtain a measurement that helps us gauge the speed of the algorithm. Add code that will count each comparison, maintaining the result in an integer counter. Supply a method getCounter to retrieve the counter. Supply a second method called resetCounter to set the counter to 0.

The code below supports creating and swapping elements in arrays of varying sizes.

/\*\*

This class contains utility methods for array manipulation.

\*/

public class ArrayUtil

{

/\*\*

Creates an array filled with random values.

@param length the length of the array

@param n the number of possible random values

@return an array filled with length numbers between

0 and n - 1

\*/

public static int[] randomIntArray(int length, int n)

{

int[] a = new int[length];

for (int i = 0; i < a.length; i++)

{

a[i] = (int) (Math.random() \* n);

}

return a;

}

/\*\*

Swaps two entries of an array.

@param a the array

@param i the first position to swap

@param j the second position to swap

\*/

public static void swap(int[] a, int i, int j)

{

int temp = a[i];

a[i] = a[j];

a[j] = temp;

}

}

Modify the main program below so you can test sorting arrays with sizes 10000, 20000, …,90000.

/\*\*

This program demonstrates the merge sort algorithm by

sorting an array that is filled with random numbers.

\*/

public class MergeSortDemo

{

public static void main(String[] args)

{

int[] a = ArrayUtil.randomIntArray(10000, 10000);

MergeSorter.resetCounter();

MergeSorter.sort(a);

System.out.println("Array size: 10000; comparisons: " + MergeSorter.getCounter());

}

}

What are your results?

**Problem 2(Total points 25)** In this lab, you will implement the bubble sort algorithm. The bubble sort is so called because it compares adjacent items, "bubbling" the smaller one up toward the beginning of the array. By comparing all pairs of adjacent items starting at the end of the array, the smallest item is guaranteed to reach the beginning of the array at the end of the first pass.

The second pass begins again at the end of the array, ultimately placing the second smallest item in the second position. During the second pass, there is no need to compare the first and second items, because the smallest element is guaranteed to be in the first position.

Bubble sort takes at most n - 1 passes for an array of n items. During the first pass, n - 1 pairs need to be compared. During the second pass, n - 2 pairs need to be compared. During the ith pass, n - i pairs need to be compared. During the last pass, n - (n - 1) or one pair needs to be compared. If, during any pass, no two adjacent items need to be interchanged, the array is in order and the sort can terminate. If it continues, no further interchanges will occur.

What is the code of your BubbleSorter class?

**Problem 3(Total points 25)** Write a CitySortDemo class that populates an ArrayList<City> with cities and then sorts it. What is the code of your class? Remember that you can use Collections.sort to sort an array list.