**Stopwatch.java**  
  
*/\*\*  
 \* A utility class to measure the running time (wall clock) of a program.  
 \*  
 \* Improved over textbook implementation by switching to nanoTime.  
 \*  
 \* @author Acuna  
 \* @author Sedgewick  
 \* @author Wayne  
 \* @version 1.0  
 \*/*public class Stopwatch {  
  
 private final long start;  
  
 */\*\*  
 \* Initializes a new stopwatch.  
 \*/* public Stopwatch() {  
 start = System.*nanoTime*();  
 }  
  
 */\*\*  
 \* Returns the elapsed CPU time (in seconds) since the stopwatch was created.  
 \*  
 \* @return elapsed CPU time (in seconds) since the stopwatch was created  
 \*/* public double elapsedTime() {  
 long now = System.*nanoTime*();  
 return (now - start) / 1000000000.0;  
 }  
}

**BenchmarkTool.java**

*/\*\*  
 \* An interface that defines a student's solution to the Section 02.01  
 \* programming homework. Contains methods that make up the structural core of a  
 \* sorting algorithm benchmarking tool.  
 \*  
 \* @author Ruben Acuna, Robert Sedgewick  
 \* @version 1.1  
 \*/*public interface BenchmarkTool {  
  
 */\*\*  
 \* Generates an array of integers where half the data is 0s, half 1s.  
 \* @param size number of elements in the array.  
 \* @return generated test set.  
 \*/* public Integer[] generateTestDataBinary(int size);  
  
 */\*\*  
 \* Generates an array of integers where half the data is 0s, half the  
 \* remainder is 1s, half the reminder is 2s, half the reminder is 3s, and so  
 \* forth.   
 \*  
 \* @param size number of elements in the array.  
 \* @return generated test set.  
 \*/* public Integer[] generateTestDataHalves(int size);  
  
 */\*\*  
 \* Generates an array of integers where half the data is 0s, and half random  
 \* int values. All values will be positive.  
 \* @param size  
 \* @return  
 \*/* public Integer[] generateTestDataHalfRandom(int size);  
  
 */\*\*  
 \* Computes the double formula value for two run times.  
 \*  
 \* @param t1 first time  
 \* @param t2 second time  
 \* @return b value  
 \*/* public double computeDoublingFormula(double t1, double t2);  
  
 */\*\*  
 \* Computes an empirical b value for insertion sort by running it on a pair  
 \* of inputs and using the doubling formula.  
 \*  
 \* @param small small test data array  
 \* @param large large test data array. twice the same of small array.  
 \* @return b value  
 \*/* public double benchmarkInsertionSort(Integer[] small, Integer[] large);  
  
 */\*\*  
 \* Computes an empirical b value for shellsort sort by running it on a pair  
 \* of inputs and using the doubling formula.  
 \* @param small small test data array  
 \* @param large large test data array. twice the same of small array.  
 \*  
 \* @return b value  
 \*/* public double benchmarkShellsort(Integer[] small, Integer[] large);  
  
 */\*\*  
 \* Runs the two sorting algorithms on the three types of test data to  
 \* produce six different b values. B values are displayed to the user.  
 \*  
 \* @param size size of benchmark array. to be doubled later.  
 \*/* public void runBenchmarks(int size);  
}

**CompletedBenchmarkTool.java**

import java.util.Arrays;  
import java.util.Random;  
  
*/\*\*  
 \* (basic description of the program or class)  
 \*  
 \* Completion time: (estimation of hours spent on this program)  
 \*  
 \* @author Eyad Mohamed AbdelMohsen Ghanem, Acuna, Sedgewick  
 \* @version 1.0  
 \*/*public class CompletedBenchmarkTool implements BenchmarkTool {  
  
 */\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
 \* START - SORTING UTILITIES, DO NOT MODIFY (FROM SEDGEWICK) \*  
 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/* public static void insertionSort(Comparable[] a) {  
 int N = a.length;  
  
 for (int i = 1; i < N; i++) {  
 // Insert a[i] among a[i-1], a[i-2], a[i-3]... ..  
 for (int j = i; j > 0 && *less*(a[j], a[j - 1]); j--)  
 *exch*(a, j, j - 1);  
 }  
 }  
  
  
 public static void shellsort(Comparable[] a) {  
 int N = a.length;  
 int h = 1;  
  
 while (h < N / 3) h = 3 \* h + 1; // 1, 4, 13, 40, 121, 364, 1093, ...  
  
 while (h >= 1) {  
 // h-sort the array.  
 for (int i = h; i < N; i++) {  
 // Insert a[i] among a[i-h], a[i-2\*h], a[i-3\*h]... .  
 for (int j = i; j >= h && *less*(a[j], a[j - h]); j -= h)  
 *exch*(a, j, j - h);  
 }  
 h = h / 3;  
 }  
 }  
  
  
 private static boolean less(Comparable v, Comparable w) {  
 return v.compareTo(w) < 0;  
 }  
  
  
 private static void exch(Comparable[] a, int i, int j) {  
 Comparable t = a[i];  
 a[i] = a[j];  
 a[j] = t;  
 }  
  
 */\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
 \* END - SORTING UTILITIES, DO NOT MODIFY \*  
 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/* @Override  
 public Integer[] generateTestDataBinary(int size) {  
 Integer[] testData = new Integer[size];  
 Random random = new Random();  
  
 for (int i = 0; i < size; i++) {  
 testData[i] = random.nextInt(2); // Generate 0 or 1  
 }  
  
 return testData;  
 }  
  
 @Override  
 public Integer[] generateTestDataHalves(int size) {  
 Integer[] testData = new Integer[size];  
 int segment = size;  
 int value = 0;  
  
 for (int i = 0; i < size; i++) {  
 if (i >= segment / 2) {  
 value++;  
 segment /= 2;  
 }  
 testData[i] = value;  
 }  
  
 return testData;  
 }  
  
 @Override  
 public Integer[] generateTestDataHalfRandom(int size) {  
 Integer[] testData = new Integer[size];  
 Random random = new Random();  
  
 for (int i = 0; i < size; i++) {  
 testData[i] = random.nextInt(100); // Generate numbers between 0 and 99  
 }  
  
 Arrays.*sort*(testData); // Sort the array in ascending order  
  
 int halfSize = size / 2;  
 Integer[] firstHalf = Arrays.*copyOfRange*(testData, 0, halfSize);  
 Integer[] secondHalf = Arrays.*copyOfRange*(testData, halfSize, size);  
  
 // Shuffle the second half  
 for (int i = 0; i < halfSize; i++) {  
 int randomIndex = random.nextInt(halfSize);  
 int temp = secondHalf[i];  
 secondHalf[i] = secondHalf[randomIndex];  
 secondHalf[randomIndex] = temp;  
 }  
  
 // Concatenate the first half and the shuffled second half  
 System.*arraycopy*(firstHalf, 0, testData, 0, halfSize);  
 System.*arraycopy*(secondHalf, 0, testData, halfSize, halfSize);  
  
 return testData;  
 }  
  
 @Override  
 public double computeDoublingFormula(double t1, double t2) {  
 if (t1 == 0) {  
 return Double.*NaN*; // Avoid division by zero  
 }  
  
 return t2 / t1;  
 }  
  
 @Override  
 public double benchmarkInsertionSort(Integer[] small, Integer[] large) {  
 Stopwatch stopwatch = new Stopwatch();  
 *insertionSort*(small);  
 double smallTime = stopwatch.elapsedTime();  
  
 stopwatch = new Stopwatch();  
 *insertionSort*(large);  
 double largeTime = stopwatch.elapsedTime();  
  
 return computeDoublingFormula(smallTime, largeTime);  
 }  
  
 @Override  
 public double benchmarkShellsort(Integer[] small, Integer[] large) {  
 Stopwatch stopwatch = new Stopwatch();  
 *shellsort*(small);  
 double smallTime = stopwatch.elapsedTime();  
  
 stopwatch = new Stopwatch();  
 *shellsort*(large);  
 double largeTime = stopwatch.elapsedTime();  
  
 return computeDoublingFormula(smallTime, largeTime);  
 }  
  
 @Override  
 public void runBenchmarks(int size) {  
 // Generate small and large data sets  
 Integer[] smallBinary = generateTestDataBinary(size);  
 Integer[] largeBinary = generateTestDataBinary(size \* 2);  
  
 Integer[] smallHalves = generateTestDataHalves(size);  
 Integer[] largeHalves = generateTestDataHalves(size \* 2);  
  
 Integer[] smallRandom = generateTestDataHalfRandom(size);  
 Integer[] largeRandom = generateTestDataHalfRandom(size \* 2);  
  
 System.*out*.println("\t\tInsertion\t\t\tShellsort");  
 System.*out*.println("Bin\t\t" + benchmarkInsertionSort(smallBinary, largeBinary) + "\t"  
 + benchmarkShellsort(smallBinary, largeBinary));  
 System.*out*.println("Half\t" + benchmarkInsertionSort(smallHalves, largeHalves) + "\t"  
 + benchmarkShellsort(smallHalves, largeHalves));  
 System.*out*.println("RanInt\t" + benchmarkInsertionSort(smallRandom, largeRandom) + "\t"  
 + benchmarkShellsort(smallRandom, largeRandom));  
 }  
  
  
 public static void main(String args[]) {  
 BenchmarkTool me = new CompletedBenchmarkTool();  
 int size = 99999;  
  
 //NOTE: feel free to change size here. all other code must go in the  
 // methods.  
  
 me.runBenchmarks(size);  
 }  
}