```
/**
 * SortingExperiment_mod
 *
 * @author Hawk Weisman
 * Based on SortingExperiment.java by Professor Gregory Kapfhammer
 *
 * An experiment to test various sorting algorithms on arrays with differing
     contents.
 */

import com.clarkware.Profiler;

import java.util.Random;
import java.io.*;

public class SortingExperiment_mod {

    private final static int NUM_EXPERIMENTS = 5;
    private static int EXPERIMENT_SIZE = 10;
    private static boolean verbose = false;
    public static enum TestType { REVERSED, RANDOM, FEW_UNIQUE, NEAR_SORTED };
    static TestType testType;

    /** This method calls all of the different experiments and uses the
     * Profiler tool in order to collect profiling information */
    public static void main(String[] args) {

        for (String arg : args) {

            if (arg.equals("verbose")) {
                verbose = true;
            } else if (arg.equals("reversed")) {
                testType = TestType.REVERSED;
            } else if (arg.equals("few-unique")) {
                testType = TestType.FEW_UNIQUE;
            } else if (arg.equals("random")) {
                testType = TestType.RANDOM;
            } else if (arg.equals("near-sorted")) {
                testType = TestType.NEAR_SORTED;
            } else {
                EXPERIMENT_SIZE = Integer.parseInt(arg);
        }
      }

    experimentBubbleSort(EXPERIMENT_SIZE);
    experimentSelectionSort(EXPERIMENT_SIZE);
    experimentInsertionSort(EXPERIMENT_SIZE);
    experimentMergeSort(EXPERIMENT_SIZE);
    experimentQuickSort(EXPERIMENT_SIZE);

    Profiler.print(new PrintWriter(System.out));

  }

    /**
     * createInputReversed
```

```java
   * @author Hawk Weisman
   * @param size The number of elements in the array
   * @return An array of values in reversed order
   */
  public static int[] createInputReversed(int size) {

      int[] values = new int[size];

      for(int i = size-1; i > 0; i--){
          values[i] = size - i;
      }
      return values;
}

  /**
   * createInputNearSorted
   * @author Hawk Weisman
   * @param size The number of elements in the array
   * @return An array of values in nearly sorted order
   */
  public static int[] createInputNearSorted(int size) {
      int[] values = new int[size];

      //Random generator = new Random((long)1.0);
      Random generator = new Random();

      for(int i = 0; i < size; i++){
          values[i] = i;
          }

      for(int i = 0; i < size/6; i++){
          int randomPosition = generator.nextInt(size);
          int temp;
          if (randomPosition > 1) {
              temp = values[randomPosition-1];
              values[randomPosition-1] = values[randomPosition];
              values[randomPosition] = temp;
          } else {
              temp = values[randomPosition+1];
              values[randomPosition+1] = values[randomPosition];
              values[randomPosition] = temp;
          }
      }
      return values;
  }

  /**
   * createInputFewUnique
   * @author Hawk Weisman
   * @param size The number of elements in the array
   * @return An array containing multiple copies of 6 unique values
   */
  public static int[] createInputFewUnique(int size) {

      int[] values = new int[size];
      int uniqueMax = (size/6);
```

```
          Random generator = new Random();

          int next_value = generator.nextInt();
          int count = 0;

          for(int i = 0; i < size; i++) {

              values[i] = next_value;
              count++;

              // if we've filled 1/6th of the array with the same value, grab a new
                  value
              if (count == uniqueMax) {
                  count = 0;
                  next_value = generator.nextInt();
              }
          }

          // shuffle the array before we return it

          for (int i=0; i < values.length; i++) {
              int randomPosition = generator.nextInt(values.length);
              int temp = values[i];
              values[i] = values[randomPosition];
              values[randomPosition] = temp;
          }
          return values;
      }

      /**
       * createInputRandom
       * @author Gregory Kapfhammer
       * @param size The number of elements in the array
       * @return An array random values
       */
      public static int[] createInputRandom(int size) {
          int[] values = new int[size];

          //Random generator = new Random((long)1.0);
          Random generator = new Random();

          for(int i = 0; i < size; i++) {

              int next_value = generator.nextInt();
              values[i] = next_value;
          }
              return values;
      }

    /**
     * arrayString
     * @author Gregory Kapfhammer
     * This method is responsible for producing a String representation of our
     * array so that we can easily print out the values.  This is useful when we
     * want to demonstrate that our sorting algorithm worked properly */
```

```java
  public static String arrayString(int[] values, int size) {

    StringBuffer value_buffer = new StringBuffer();
    for(int i = 0; i < size; i++) {

      value_buffer.append(values[i]);

      if(i < size) {
        value_buffer.append("\n");
      }
    }
      return value_buffer.toString();
  }

  /** This method conducts an experiment with the BubbleSort sorting algorithm */
  public static void experimentBubbleSort(int size)  {
      int[] values = new int[size];

      for(int i = 0; i < NUM_EXPERIMENTS; i++) {

          // ask our createInput method for some ints based upon the provided
              size
          switch (testType) {
              case RANDOM:
                  values = createInputRandom(size);
                  break;
              case REVERSED:
                  values = createInputReversed(size);
                  break;
              case NEAR_SORTED:
                  values = createInputNearSorted(size);
                  break;
              case FEW_UNIQUE:
                  values = createInputFewUnique(size);
                  break;
              default:
                  values = createInputRandom(size);
                  break;
          }

      if( verbose )
        System.out.println("Initial values: " + arrayString(values, size));

      Profiler.begin("Bubble Sort");
      BubbleSort.bubbleSort(values, size);
      Profiler.end("Bubble Sort");

      if( verbose )
        System.out.println("Final values: " + arrayString(values, size));
    }
  }

  /** This method conducts an experiment with the SelectionSort sorting algorithm
      */
  public static void experimentSelectionSort(int size) {
    int[] values = new int[size];
    for(int i = 0; i < NUM_EXPERIMENTS; i++) {

      // ask our createInput method for some ints based upon the provided size
        switch (testType) {
            case RANDOM:
                values = createInputRandom(size);
                break;
            case REVERSED:
                values = createInputReversed(size);
                break;
            case NEAR_SORTED:
                values = createInputNearSorted(size);
                break;
            case FEW_UNIQUE:
                values = createInputFewUnique(size);
                break;
            default:
                values = createInputRandom(size);
                break;
        }


      if( verbose )
        System.out.println("Initial values: " + arrayString(values, size));

      Profiler.begin("Selection Sort");
      SelectionSort.selectionSort(values, size);
      Profiler.end("Selection Sort");

      if( verbose )
        System.out.println("Final values: " + arrayString(values, size));
    }
  }

  /** This method conducts an experiment with the Insertion Sort algorithm */
  public static void experimentInsertionSort(int size) {

      int[] values = new int[size];

      for(int i = 0; i < NUM_EXPERIMENTS; i++) {

          // ask our createInput method for some ints based upon the provided size
          switch (testType) {
              case RANDOM:
                  values = createInputRandom(size);
                  break;
              case REVERSED:
                  values = createInputReversed(size);
                  break;
              case NEAR_SORTED:
                  values = createInputNearSorted(size);
                  break;
              case FEW_UNIQUE:
                  values = createInputFewUnique(size);
```

```
                    break;
                default:
                    values = createInputRandom(size);
                    break;
            }

        if( verbose )
          System.out.println("Initial values: " + arrayString(values, size));

        Profiler.begin("Insertion Sort");
        InsertionSort.insertionSort(values, size);
        Profiler.end("Insertion Sort");

        if( verbose )
          System.out.println("Final values: " + arrayString(values, size));

      }

    }

    /** This method conducts an experiment with the Merge Sort algorithm */
    public static void experimentMergeSort(int size) {

        int[] values = new int[size];

        for(int i = 0; i < NUM_EXPERIMENTS; i++) {

            // ask our createInput method for some ints based upon the provided
                size
            switch (testType) {
                case RANDOM:
                    values = createInputRandom(size);
                    break;
                case REVERSED:
                    values = createInputReversed(size);
                    break;
                case NEAR_SORTED:
                    values = createInputNearSorted(size);
                    break;
                case FEW_UNIQUE:
                    values = createInputFewUnique(size);
                    break;
                default:
                    values = createInputRandom(size);
                    break;
            }

        if( verbose )
          System.out.println("Initial values: " + arrayString(values, size));

        Profiler.begin("Merge Sort");
        MergeSort.mergeSort(values, size);
        Profiler.end("Merge Sort");

        if( verbose )
          System.out.println("Final values: " + arrayString(values, size));
      }
    }

    public static void experimentQuickSort(int size) {

        int[] values = new int[size];

        for(int i = 0; i < NUM_EXPERIMENTS; i++) {

            // ask our createInput method for some ints based upon the provided
                size
            switch (testType) {
                case RANDOM:
                    values = createInputRandom(size);
                    break;
                case REVERSED:
                    values = createInputReversed(size);
                    break;
                case NEAR_SORTED:
                    values = createInputNearSorted(size);
                    break;
                case FEW_UNIQUE:
                    values = createInputFewUnique(size);
                    break;
                default:
                    values = createInputRandom(size);
                    break;
            }

        if( verbose )
          System.out.println("Initial values: " + arrayString(values, size));

        Profiler.begin("Quick Sort");
        QuickSort.quickSort(values, size);
        Profiler.end("Quick Sort");

        if( verbose )
          System.out.println("Final values: " + arrayString(values, size));
      }
    }
  }
```