```java
package reflectionSort;

import java.io.BufferedReader;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;
import java.util.ArrayList;
import java.util.Collections;
import java.util.LinkedList;
import java.util.List;

public class Main {

  public static void main(String[] args) {
    long startTime = System.currentTimeMillis();
    ArrayList<Reflection> reflections = new ArrayList<Reflection>();
    ArrayList<SymmetryOperator> symmetryOperators = new ArrayList<SymmetryOperator>();

    // Read symmetry operators and reflections
    try {
      BufferedReader in = new BufferedReader(new FileReader("in"));
      DataFileReader reader = new DataFileReader(in);
      reader.readData(symmetryOperators, reflections);
    } catch (FileNotFoundException e) {
      e.printStackTrace();
      return;
    } catch (IOException e) {
      e.printStackTrace();
      return;
    }
    long readTime = System.currentTimeMillis();

    // Standardize reflections
    for (Reflection reflection : reflections) {
      reflection.standardizeHkl(symmetryOperators);
    }
    long standardizeTime = System.currentTimeMillis();

    // Sort reflections
    Collections.sort(reflections);
    long sortTime = System.currentTimeMillis();

    // Merge reflections, remove systematic absences, and count centric reflections
    LinkedList<Reflection> uniqueReflections = new LinkedList<Reflection>();
    List<Reflection> systematicallyAbsentReflections;
    double rint = ReflectionAnalyzer.mergeUniqueReflections(reflections, uniqueReflections, new SimpleMergeFunction());
```

```
    systematicallyAbsentReflections = ReflectionAnalyzer.removeSystematicAbsences(uniqueReflections, symmetryOperators);
    int centricReflections = ReflectionAnalyzer.countCentricReflections(uniqueReflections, symmetryOperators);

    long mergeTime = System.currentTimeMillis();

    // Print list of systematically absent reflections
    for (Reflection reflection : systematicallyAbsentReflections) {
      System.out.println(String.format("Systematically absent: %3d %3d %3d  I/sigma = %5.2f", reflection.getHkl().x,
          reflection.getHkl().y, reflection.getHkl().z, reflection.getI() / reflection.getSigI()));
    }

    // Print statistics
    System.out.println(reflections.size() + " total reflections");
    System.out.println(systematicallyAbsentReflections.size() + " systematically absent reflections");
    System.out.println(uniqueReflections.size() + " unique reflections");
    System.out.println(centricReflections + " centric reflections");
    System.out.println("R(int) = " + String.format("%5.4f", rint));

    long outputTime = System.currentTimeMillis();

    boolean printTiming = true;
    if (printTiming) {
      System.out.println(String.format("%4.2f seconds reading", (readTime - startTime) / 1000.0));
      System.out.println(String.format("%4.2f seconds standardizing", (standardizeTime - readTime) / 1000.0));
      System.out.println(String.format("%4.2f seconds sorting", (sortTime - standardizeTime) / 1000.0));
      System.out.println(String.format("%4.2f seconds merging", (mergeTime - sortTime) / 1000.0));
      System.out.println(String.format("%4.2f seconds printing", (outputTime - mergeTime) / 1000.0));
      System.out.println(String.format("%4.2f seconds total", (outputTime - startTime) / 1000.0));
    }
  }
}
```

```java
package reflectionSort;

import java.io.BufferedReader;
import java.io.IOException;
import java.util.List;

public class DataFileReader {

  private final BufferedReader input;

  public DataFileReader(BufferedReader in) {
    this.input = in;
  }

  public void readData(List<SymmetryOperator> symmetryOperators, List<Reflection> reflections) throws IOException {
    // Ignore first two lines of input
    String line = input.readLine();
    line = input.readLine();
    // Read line which starts with an integer indicating the number of
    // symmetry operators
    line = input.readLine();
    String[] tokens = line.split("\\s");
    int numberOfSymmetryOperators = Integer.parseInt(tokens[0]);
    // Read symmetry opertors in the format:
    // r11 r12 r13 r21 r22 r23 r31 r32 r33 t1 t2 t3
    int[] matrix = new int[9];
    float[] vector = new float[3];
    for (int i = 0; i < numberOfSymmetryOperators; ++i) {
      line = input.readLine();
      tokens = line.split("\\s");
      for (int j = 0; j < matrix.length; ++j) {
        matrix[j] = Integer.parseInt(tokens[j]);
      }
      for (int j = 0; j < vector.length; ++j) {
        vector[j] = Float.parseFloat(tokens[9 + j]);
      }
      symmetryOperators.add(new SymmetryOperator(matrix, vector));
    }
    // Read read reflections in the format:
    // h k l I sigI
    line = input.readLine();
    while (line != null) {
      tokens = line.split("\\s");
      Reflection r = new Reflection(Integer.parseInt(tokens[0]), Integer.parseInt(tokens[1]), Integer
          .parseInt(tokens[2]), Double.parseDouble(tokens[3]), Double.parseDouble(tokens[4]));
      reflections.add(r);
```

```
      line = input.readLine();
    }
  }
}
```

```
package reflectionSort;


public class SymmetryOperator {

  int[] reciprocalMatrix = new int[9];
  float[] vector = new float[3];

  public SymmetryOperator(int[] matrix, float[] vector) {
    reciprocalMatrix[0] = matrix[0];
    reciprocalMatrix[1] = matrix[3];
    reciprocalMatrix[2] = matrix[6];
    reciprocalMatrix[3] = matrix[1];
    reciprocalMatrix[4] = matrix[4];
    reciprocalMatrix[5] = matrix[7];
    reciprocalMatrix[6] = matrix[2];
    reciprocalMatrix[7] = matrix[5];
    reciprocalMatrix[8] = matrix[8];
    this.vector[0] = vector[0];
    this.vector[1] = vector[1];
    this.vector[2] = vector[2];
  }

  public void applyReciprocalRotation(Tuple3i hkl, Tuple3i transformedHkl) {
    transformedHkl.x = reciprocalMatrix[0]*hkl.x + reciprocalMatrix[1]*hkl.y + reciprocalMatrix[2]*hkl.z;
    transformedHkl.y = reciprocalMatrix[3]*hkl.x + reciprocalMatrix[4]*hkl.y + reciprocalMatrix[5]*hkl.z;
    transformedHkl.z = reciprocalMatrix[6]*hkl.x + reciprocalMatrix[7]*hkl.y + reciprocalMatrix[8]*hkl.z;
  }

  public float phaseShift(Tuple3i hkl) {
    return hkl.z*vector[0] + hkl.y*vector[1] + hkl.z*vector[2];
  }

}
```

```java
package reflectionSort;

import java.util.List;


class Reflection implements Comparable<Reflection>, Cloneable {
  MillerIndex hkl;

  double i;

  double sigI;

  protected Reflection() {
    hkl = new MillerIndex();
  }

  Reflection(int h, int k, int l, double i, double sigI) {
    this.hkl = new MillerIndex(h, k, l);
    this.i = i;
    this.sigI = sigI;
  }

  public Object clone() {
    Reflection r = new Reflection();
    r.hkl = new MillerIndex(hkl);
    return r;
  }

  public String toString() {
    return String.format("%4d %4d %4d %6.2f %6.2f", hkl.x, hkl.y, hkl.z, i, sigI);
  }

  void standardizeHkl(List<SymmetryOperator> symmetryOperators) {
    MillerIndex transformedHkl = new MillerIndex();
    applySymmetryOperators(symmetryOperators, transformedHkl);
  }

  private void applySymmetryOperators(List<SymmetryOperator> symmetryOperators, MillerIndex transformedHkl) {
    for (SymmetryOperator symmetryOperator : symmetryOperators) {
      symmetryOperator.applyReciprocalRotation(hkl, transformedHkl);
      setIfStandardHkl(transformedHkl);
      transformedHkl.negate();
      setIfStandardHkl(transformedHkl);
    }
  }
```

```
  private void setIfStandardHkl(MillerIndex transformedHkl) {
    if (transformedHkl.z > hkl.z || (transformedHkl.z == hkl.z && transformedHkl.y > hkl.y)
        || (transformedHkl.z == hkl.z && transformedHkl.y == hkl.y && transformedHkl.x > hkl.x)) {
      hkl.set(transformedHkl);
    }
  }

  public MillerIndex getHkl() {
    return hkl;
  }

  public double getI() {
    return i;
  }

  public double getSigI() {
    return sigI;
  }

  @Override
  public boolean equals(Object obj) {
    return hkl.equals(((Reflection) obj).hkl);
  }

  @Override
  public int hashCode() {
    return hkl.hashCode();
  }

  public int compareTo(Reflection other) {
    return hkl.compareTo(other.hkl);
  }

  public void setI(double i) {
    this.i = i;
  }

  public void setSigI(double sigI) {
    this.sigI = sigI;
  }


}
```

```java
package reflectionSort;

import java.util.ArrayList;
import java.util.Collection;
import java.util.Iterator;
import java.util.List;


public class ReflectionAnalyzer {

  public static double mergeUniqueReflections(List<Reflection> reflectionsList,
      List<Reflection> uniqueReflectionsList, SimpleMergeFunction mergeFunction) {

    List<Reflection> duplicateReflections = new ArrayList<Reflection>();
    double rIntSumIDeviation = 0.0;
    double rIntSumI = 0.0;

    // Merge duplicate reflections and determine systematic absences
    PeekableIterator iter = new PeekableIterator(reflectionsList.iterator());
    do {
      Reflection reflection = (Reflection) iter.next();
      duplicateReflections.clear();
      duplicateReflections.add(reflection);
      while (iter.peek() != null && iter.peek().equals(reflection)) {
        Reflection duplicateReflection = (Reflection) iter.next();
        duplicateReflections.add(duplicateReflection);
      }
      if (duplicateReflections.size() > 1) {
        Reflection mergedReflection = mergeFunction.merge(duplicateReflections);
        for (Reflection duplicateReflection : duplicateReflections) {
          rIntSumIDeviation += Math.abs(duplicateReflection.getI() - mergedReflection.getI());
          rIntSumI += mergedReflection.getI();
        }
        uniqueReflectionsList.add(mergedReflection);
      } else {
        uniqueReflectionsList.add(reflection);
      }
    } while (iter.hasNext());
    return rIntSumIDeviation / rIntSumI;
  }

  public static List<Reflection> removeSystematicAbsences(List<Reflection> uniqueReflectionsList, List<SymmetryOperator>
 symmetryOperators) {
    ArrayList<Reflection> systemmaticallyAbsentReflections = new ArrayList<Reflection>();
    for (Iterator iter = uniqueReflectionsList.iterator(); iter.hasNext();) {
      Reflection reflection = (Reflection) iter.next();
```

```
      if (MillerIndex.isSystematicallyAbsent(reflection.getHkl(), symmetryOperators)) {
        iter.remove();
        systemmaticallyAbsentReflections.add(reflection);
      }
    }
    return systemmaticallyAbsentReflections;
  }

  public static int countCentricReflections(Collection<Reflection> uniqueReflectionsList, List<SymmetryOperator>
 symmetryOperators) {
    int i = 0;
    for (Reflection reflection : uniqueReflectionsList) {
      if (MillerIndex.isCentrosymmetric(reflection.getHkl(), symmetryOperators)) {
        ++i;
      }
    }
    return i;
  }

}
```

```java
package reflectionSort;

import java.util.List;

class SimpleMergeFunction {

  public Reflection merge(List<Reflection> reflections) {
    double sumI = 0.0;
    double sumSigI = 0.0;
    for (Reflection reflection : reflections) {
      sumI += reflection.getI();
      sumSigI += 1.0 / (reflection.getSigI() * reflection.getSigI());
    }
    double i = sumI / reflections.size();
    double sigI = 1.0 / Math.sqrt(sumSigI);
    Reflection mergedReflection = (Reflection) reflections.get(0).clone();
    mergedReflection.setI(i);
    mergedReflection.setSigI(sigI);
    return mergedReflection;
  }

}
```

```java
package reflectionSort;

import java.util.List;


public class MillerIndex extends Tuple3i {

  public MillerIndex() {
    super();
  }

  public MillerIndex(MillerIndex hkl) {
    super(hkl);
  }

  public MillerIndex(int h, int k, int l) {
    super(h, k, l);
  }

  void standardize(List<SymmetryOperator> symmetryOperators) {
    MillerIndex transformedHkl = new MillerIndex();
    for (SymmetryOperator symmetryOperator : symmetryOperators) {
      symmetryOperator.applyReciprocalRotation(this, transformedHkl);
      setIfStandard(transformedHkl);
      transformedHkl.negate();
      setIfStandard(transformedHkl);
    }
  }

  private void setIfStandard(MillerIndex transformedHkl) {
    if (transformedHkl.z > z || (transformedHkl.z == z && transformedHkl.y > y)
        || (transformedHkl.z == z && transformedHkl.y == y && transformedHkl.z > z)) {
      set(transformedHkl);
    }
  }

  public static boolean isSystematicallyAbsent(MillerIndex hkl, List<SymmetryOperator> symmetryOperators) {
    for (SymmetryOperator symmetryOperator : symmetryOperators) {
      if (isSymmetric(hkl, symmetryOperator)) {
        double n = symmetryOperator.phaseShift(hkl);
        if (Math.abs(Math.rint(n) - n) > 0.001) {
          return true;
        }
      }
    }
    return false;
```

```
  }

  public static boolean isCentrosymmetric(MillerIndex hkl, List<SymmetryOperator> symmetryOperators) {
    for (SymmetryOperator symmetryOperator : symmetryOperators) {
      if (isCentrosymmetric(hkl, symmetryOperator)) {
        return true;
      }
    }
    return false;
  }

  static boolean isSymmetric(MillerIndex hkl, SymmetryOperator symmetryOperator) {
    MillerIndex transformedHkl = new MillerIndex();
    symmetryOperator.applyReciprocalRotation(hkl, transformedHkl);
    return hkl.equals(transformedHkl);
  }

  static boolean isCentrosymmetric(MillerIndex hkl, SymmetryOperator symmetryOperator) {
    MillerIndex transformedHkl = new MillerIndex();
    symmetryOperator.applyReciprocalRotation(hkl, transformedHkl);
    transformedHkl.negate();
    return transformedHkl.equals(hkl);
  }
}
```

```java
package reflectionSort;

public class Tuple3i implements Comparable<Tuple3i> {

  public int x;
  public int y;
  public int z;

  public Tuple3i() {
  }

  public Tuple3i(Tuple3i t) {
    this(t.x, t.y, t.z);
  }

  public Tuple3i(int x, int y, int z) {
    this.x = x;
    this.y = y;
    this.z = z;
  }

  public Tuple3i(int[] xyz) {
    this(xyz[0], xyz[1], xyz[2]);
  }

  public String toString() {
    return String.format("%d %d %d", x, y, z);
  }

  public int hashCode() {
    int result = x;
    result += 29 * y;
    result += 29 * z;
    return result;
  }

  public boolean equals(Object o) {
    return compareTo((MillerIndex) o) == 0;
  }

  public int compareTo(Tuple3i o) {
    int result = z - o.z;
    if (result == 0) {
      result = y - o.y;
      if (result == 0) {
        result = x - o.x;
```

```
      }
    }
    return result;
  }

  public void negate() {
    x *= -1;
    y *= -1;
    z *= -1;
  }

  public void set(Tuple3i t) {
    set(t.x, t.y, t.z);
  }

  public void set(int x, int y, int z) {
    this.x = x;
    this.y = y;
    this.z = z;
  }

}
```

```java
package reflectionSort;

import java.util.Iterator;

public class PeekableIterator implements Iterator {

  Object next;
  Object peek;
  Iterator iter;

  PeekableIterator(Iterator iter) {
    this.iter = iter;
    if (iter.hasNext()) {
      peek = iter.next();
    }
  }

  public boolean hasNext() {
    return peek != null;
  }

  public Object next() {
    next = peek;
    if (iter.hasNext()) {
      peek = iter.next();
    } else {
      peek = null;
    }
    return next;
  }

  public void remove() {
    throw new UnsupportedOperationException();
  }

  public Object peek() {
    return peek;
  }

}
```