```
import java.awt.geom.Line2D;
import java.io.BufferedReader;
import java.io.FileInputStream;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

public class Monster {

  ////////////// Geometry Library //////////////////////
  private static final double EPS = 1e-10;

  private static int cmp(double x, double y) {
    return (x <= y + EPS) ? (x + EPS < y) ? -1 : 0 : 1;
  }
  // Immutable Point Class.
  private static class Point implements Comparable<Point> {
    public double x;
    public double y;
    public Point(double x, double y) {
      this.x = x;
      this.y = y;
    }
    public Point() {
      this.x = 0.0;
      this.y = 0.0;
    }
    public double dotProduct(Point o) {
      return this.x * o.x + this.y * o.y;
    }
    public double crossProduct(Point o) {
      return this.x * o.y - this.y * o.x;
    }
    public Point add(Point o) {
      return new Point(this.x + o.x, this.y + o.y);
    }
    public Point substract(Point o) {
      return new Point(this.x - o.x, this.y - o.y);
    }
    public Point multiply(double m) {
      return new Point(this.x * m, this.y * m);
    }
    public Point divide(double m) {
      return new Point(this.x / m, this.y / m);
    }
    @Override
    public int compareTo(Point o) {
      if (this.x < o.x) return -1;
      if (this.x > o.x) return 1;
      if (this.y < o.y) return -1;
      if (this.y > o.y) return 1;
      return 0;
    }
    // Euclidean distance between two points;
    double distance(Point o) {
        double d1 = x - o.x, d2 = y - o.y;
        return Math.sqrt(d1 * d1 + d2 * d2);
    }
```

```
}

// Calculates the angle between the two vectors defined by p - r and q - r.
// The formula comes from the definition of the dot and the cross product:
//
// A . B = |A||B|cos(c)
// A x B = |A||B|sin(c)
//
// sin(c)   A x B
// ------ = ------ = tan(c)
// cos(c)   A . B
private static double angle(Point p, Point q, Point r) {
  Point u = p.substract(r), v = q.substract(r);
  return Math.atan2(u.crossProduct(v), u.dotProduct(v));
}

// Calculates sign of the turn between the two vectors defined by <p-r> and
// <q-r>.
//
// Just to remember, the cross product is defined by (x1 * y2) - (x2 * y1) and
// is negative if it is a right turn and positive if it is a left turn. e.g.
//
//             .p3
//             ^
//            /
//       .p2 /
//       ^  /
//       | /
//       .p1
// The cross product between the vectors <p2-p1> and <p3-p1> is negative, that
// means it is a right turn.
private static int turn(Point p, Point q, Point r) {
  return cmp((p.substract(r)).crossProduct(q.substract(r)), 0.0);
}

// Decides if the point r is inside the segment defined by the points p and q.
// To do this, we have to check two conditions:
// 1. That the turn between the two vectors formed by p - q and r - q is zero
// (that means they are parallel).
// 2. That the dot product between the vector formed by p - r and q - r (that
// means the testing point as the initial point for both vectors) is less than
// or equal to zero (that means that the two vectors have opposite direction).
private static boolean between(Point p, Point q, Point r) {
  return turn(p, r, q) == 0 && cmp((p.substract(r)).dotProduct(q.substract(r)), 0.0) <= 0;
}

// Returns 0, -1 or 1 depending if p is in the exterior, the frontier or the
// interior of the given polygon respectively, the polygon must be in clockwise
// or counterclockwise order [MANDATORY!!].
// The idea is to iterate over each of the points in the polygon and consider
// the segment formed by two adjacent points, if the test points is inside that
// segment, the point is in the frontier, if not, we add the angles inside the
// vectors formed by the two points of the polygon and the test point. For a
// point outside the polygon this sum is zero because the angles cancel
// themselves.
private static int inPolygon(Point p, Point[] polygon, int polygonSize) {
  double a = 0; int N = polygonSize;
  for (int i = 0; i < N; ++i) {
    if (between(polygon[i], polygon[(i + 1) % N], p)) return -1;
    a += angle(polygon[i], polygon[(i + 1) % N], p);
```

```
    }
    return (cmp(a, 0.0) == 0) ? 0 : 1;
  }

  private static Point GetIntersection(Line2D.Double l1, Line2D.Double l2) {
    double A1 = l1.y2 - l1.y1;
    double B1 = l1.x1 - l1.x2;
    double C1 = A1 * l1.x1 + B1 * l1.y1;
    double A2 = l2.y2 - l2.y1;
    double B2 = l2.x1 - l2.x2;
    double C2 = A2 * l2.x1 + B2 * l2.y1;
    double det = A1*B2 - A2*B1;
      if(det == 0){
          // Lines are parallel, check if they are on the same line.
        double m1 = A1 / B1;
        double m2 = A2 / B2;
        // Check whether their slopes are the same or not, or if they are vertical.
        if (cmp(m1, m2) == 0 || (B1 == 0 && B2 == 0)) {
          if ((l1.x1 == l2.x1 && l1.y1 == l2.y1) ||
              (l1.x1 == l2.x2 && l1.y1 == l2.y2)) return new Point(l1.x1, l1.y1);
          if ((l1.x2 == l2.x1 && l1.y2 == l2.y1) ||
              (l1.x2 == l2.x2 && l1.y2 == l2.y2)) return new Point(l1.x2, l1.y2);
        }
        return null;
      }
        double x = (B2*C1 - B1*C2) / det;
        double y = (A1*C2 - A2*C1) / det;
        return new Point(x, y);
  }
  ////////////////////////////////////////////////////////

  private static Line2D.Double[] lines;
  private static List<Integer>[] graph;
  private static int[] marked;
  private static int[] stack;
  private static int[] cycle;
  private static int stackLen;
  private static int cycleLen;
  private static boolean res;

  private static void FindCycle(int node) {
    cycleLen = 0;
    cycle[cycleLen++] = node;
    int k = stackLen - 1;
    while (stack[k] != node) {
      cycle[cycleLen++] = stack[k];
      --k;
    }
    cycle[cycleLen++] = stack[k];
    Point[] points = new Point[cycleLen];
    for (int i = 0; i < cycleLen - 1; ++i) {
      points[i] = GetIntersection(lines[cycle[i]], lines[cycle[i + 1]]);
    }
    if (inPolygon(new Point(), points, cycleLen - 1) != 0) res = true;
  }

  private static void DoIt(int act, int last) {
    marked[act] = 1;
    stack[stackLen++] = act;
    for (Integer i : graph[act]) {
```

```
      if (marked[i] == 1 && i != last) {
        FindCycle(i);
      } else if (marked[i] == 0) {
        DoIt(i, act);
      }
    }
    --stackLen;
    marked[act] = 2;
  }

  @SuppressWarnings("unchecked")
  public static void main(String[] args) throws IOException {
    System.setIn(new FileInputStream("monster.in"));
    BufferedReader reader = new BufferedReader(new InputStreamReader(System.in));
    String[] parts;
    while (true) {
      int num = Integer.parseInt(reader.readLine());
      if (num == 0) break;
      lines = new Line2D.Double[num];
      for (int i = 0; i < num; ++i) {
        parts = reader.readLine().split("[ ]+");
        lines[i] = new Line2D.Double(Integer.parseInt(parts[0]),
            Integer.parseInt(parts[1]), Integer.parseInt(parts[2]),
            Integer.parseInt(parts[3]));
      }
      graph = (List<Integer>[]) new List[num];
      for (int i = 0; i < num; ++i) {
        graph[i] = new ArrayList<Integer>();
      }
      for (int i = 0; i < num; ++i) {
        for (int j = i + 1; j < num; ++j) {
          if (lines[i].intersectsLine(lines[j])) {
            graph[i].add(j);
            graph[j].add(i);
          }
        }
      }

      res = false;
      marked = new int[num];
      stack = new int[num];
      cycle = new int[num + 1];
      stackLen = 0;
      Arrays.fill(marked, 0);
      for (int i = 0; i < num; ++i) {
        if (marked[i] != 0) continue;
        DoIt(i, -1);
      }

      if (res) System.out.println("yes");
      else System.out.println("no");
    }
  }
}
```