Contenido

[Circle 2](#_Toc498000799)

[Line 6](#_Toc498000800)

[Delaunay triangulation 9](#_Toc498000801)

[Voronoi Diagram 12](#_Toc498000802)

[Treap 1 23](#_Toc498000803)

[Treap 2 26](#_Toc498000804)

[Treap 3 30](#_Toc498000805)

[Spase Table 33](#_Toc498000806)

[Coloreado de Grafos 34](#_Toc498000807)

[Shortest paths. Bellman–Ford algorithm in O(V\*E). Negative cycle detection. 35](#_Toc498000808)

[Dinitz in bipartite matching (attacking rooks) 37](#_Toc498000809)

## Circle

Geometry: Circle

import java.util.\*;

public class CircleOperations {

static final double EPS = 1e-10;

public static double fastHypot(double x, double y) {

return Math.sqrt(x \* x + y \* y);

}

public static class Point {

public double x, y;

public Point(double x, double y) {

this.x = x;

this.y = y;

}

}

public static class Circle {

public double r, x, y;

public Circle(double x, double y, double r) {

this.x = x;

this.y = y;

this.r = r;

}

public boolean contains(Point p) {

return fastHypot(p.x - x, p.y - y) < r + EPS;

}

}

public static class Line {

double a, b, c;

public Line(double a, double b, double c) {

this.a = a;

this.b = b;

this.c = c;

}

public Line(Point p1, Point p2) {

a = +(p1.y - p2.y);

b = -(p1.x - p2.x);

c = p1.x \* p2.y - p2.x \* p1.y;

}

}

// geometric solution

public static Point[] circleLineIntersection(Circle circle, Line line) {

double a = line.a;

double b = line.b;

double c = line.c + circle.x \* a + circle.y \* b;

double r = circle.r;

double aabb = a \* a + b \* b;

double d = c \* c / aabb - r \* r;

if (d > EPS)

return new Point[0];

double x0 = -a \* c / aabb;

double y0 = -b \* c / aabb;

if (d > -EPS)

return new Point[]{new Point(x0 + circle.x, y0 + circle.y)};

d /= -aabb;

double k = Math.sqrt(d < 0 ? 0 : d);

return new Point[]{

new Point(x0 + k \* b + circle.x, y0 - k \* a + circle.y),

new Point(x0 - k \* b + circle.x, y0 + k \* a + circle.y)};

}

// algebraic solution

public static Point[] circleLineIntersection2(Circle circle, Line line) {

return Math.abs(line.a) >= Math.abs(line.b)

? intersection(line.a, line.b, line.c, circle.x, circle.y, circle.r, false)

: intersection(line.b, line.a, line.c, circle.y, circle.x, circle.r, true);

}

static Point[] intersection(double a, double b, double c, double CX, double CY, double R, boolean swap) {

// ax+by+c=0

// (by+c+aCX)^2+(ay-aCY)^2=(aR)^2

double A = a \* a + b \* b;

double B = 2.0 \* b \* (c + a \* CX) - 2.0 \* a \* a \* CY;

double C = (c + a \* CX) \* (c + a \* CX) + a \* a \* (CY \* CY - R \* R);

double d = B \* B - 4 \* A \* C;

if (d < -EPS)

return new Point[0];

d = Math.sqrt(d < 0 ? 0 : d);

double y1 = (-B + d) / (2 \* A);

double x1 = (-c - b \* y1) / a;

double y2 = (-B - d) / (2 \* A);

double x2 = (-c - b \* y2) / a;

return swap ? d > EPS ? new Point[]{new Point(y1, x1), new Point(y2, x2)} : new Point[]{new Point(y1, x1)}

: d > EPS ? new Point[]{new Point(x1, y1), new Point(x2, y2)} : new Point[]{new Point(x1, y1)};

}

public static Point[] circleCircleIntersection(Circle c1, Circle c2) {

if (fastHypot(c1.x - c2.x, c1.y - c2.y) < EPS) {

if (Math.abs(c1.r - c2.r) < EPS)

return null; // infinity intersection points

return new Point[0];

}

double dx = c2.x - c1.x;

double dy = c2.y - c1.y;

double A = -2 \* dx;

double B = -2 \* dy;

double C = dx \* dx + dy \* dy + c1.r \* c1.r - c2.r \* c2.r;

Point[] res = circleLineIntersection(new Circle(0, 0, c1.r), new Line(A, B, C));

for (Point point : res) {

point.x += c1.x;

point.y += c1.y;

}

return res;

}

public static double circleCircleIntersectionArea(Circle c1, Circle c2) {

double r = Math.min(c1.r, c2.r);

double R = Math.max(c1.r, c2.r);

double d = fastHypot(c1.x - c2.x, c1.y - c2.y);

if (d < R - r + EPS)

return Math.PI \* r \* r;

if (d > R + r - EPS)

return 0;

double area = r \* r \* Math.acos((d \* d + r \* r - R \* R) / 2 / d / r) + R \* R

\* Math.acos((d \* d + R \* R - r \* r) / 2 / d / R) - 0.5

\* Math.sqrt((-d + r + R) \* (d + r - R) \* (d - r + R) \* (d + r + R));

return area;

}

public static Line[] tangents(Circle a, Circle b) {

List<Line> lines = new ArrayList<>();

for (int i = -1; i <= 1; i += 2)

for (int j = -1; j <= 1; j += 2)

tangents(new Point(b.x - a.x, b.y - a.y), a.r \* i, b.r \* j, lines);

for (Line line : lines)

line.c -= line.a \* a.x + line.b \* a.y;

return lines.toArray(new Line[lines.size()]);

}

static void tangents(Point center2, double r1, double r2, List<Line> lines) {

double r = r2 - r1;

double z = center2.x \* center2.x + center2.y \* center2.y;

double d = z - r \* r;

if (d < -EPS) return;

d = Math.sqrt(d < 0 ? 0 : d);

lines.add(new Line((center2.x \* r + center2.y \* d) / z, (center2.y \* r - center2.x \* d) / z, r1));

}

// min enclosing circle in O(n) on average

public static Circle minEnclosingCircle(Point[] pointsArray) {

if (pointsArray.length == 0)

return new Circle(0, 0, 0);

if (pointsArray.length == 1)

return new Circle(pointsArray[0].x, pointsArray[0].y, 0);

List<Point> points = Arrays.asList(pointsArray);

Collections.shuffle(points);

Circle circle = getCircumCircle(points.get(0), points.get(1));

for (int i = 2; i < points.size(); i++)

if (!circle.contains(points.get(i)))

circle = minEnclosingCircleWith1Point(points.subList(0, i), points.get(i));

return circle;

}

static Circle minEnclosingCircleWith1Point(List<Point> points, Point q) {

Circle circle = getCircumCircle(points.get(0), q);

for (int i = 1; i < points.size(); i++)

if (!circle.contains(points.get(i)))

circle = minEnclosingCircleWith2Points(points.subList(0, i), points.get(i), q);

return circle;

}

static Circle minEnclosingCircleWith2Points(List<Point> points, Point q1, Point q2) {

Circle circle = getCircumCircle(q1, q2);

for (Point point : points)

if (!circle.contains(point))

circle = getCircumCircle(q1, q2, point);

return circle;

}

public static Circle getCircumCircle(Point a, Point b) {

double x = (a.x + b.x) / 2.;

double y = (a.y + b.y) / 2.;

double r = fastHypot(a.x - x, a.y - y);

return new Circle(x, y, r);

}

public static Circle getCircumCircle(Point a, Point b, Point c) {

double Bx = b.x - a.x;

double By = b.y - a.y;

double Cx = c.x - a.x;

double Cy = c.y - a.y;

double d = 2 \* (Bx \* Cy - By \* Cx);

if (Math.abs(d) < EPS)

return getCircumCircle(new Point(Math.min(a.x, Math.min(b.x, c.x)), Math.min(a.y, Math.min(b.y, c.y))),

new Point(Math.max(a.x, Math.max(b.x, c.x)), Math.max(a.y, Math.max(b.y, c.y))));

double z1 = Bx \* Bx + By \* By;

double z2 = Cx \* Cx + Cy \* Cy;

double cx = Cy \* z1 - By \* z2;

double cy = Bx \* z2 - Cx \* z1;

double x = cx / d;

double y = cy / d;

double r = fastHypot(x, y);

return new Circle(x + a.x, y + a.y, r);

}

// Usage example

public static void main(String[] args) {

Random rnd = new Random(1);

for (int step = 0; step < 100\_000; step++) {

int range = 10;

int x = rnd.nextInt(range) - range / 2;

int y = rnd.nextInt(range) - range / 2;

int r = rnd.nextInt(range);

int x1 = rnd.nextInt(range) - range / 2;

int y1 = rnd.nextInt(range) - range / 2;

int x2 = rnd.nextInt(range) - range / 2;

int y2 = rnd.nextInt(range) - range / 2;

if (x1 == x2 && y1 == y2)

continue;

Point[] p1 = circleLineIntersection(new Circle(x, y, r), new Line(new Point(x1, y1), new Point(x2, y2)));

Point[] p2 = circleLineIntersection2(new Circle(x, y, r), new Line(new Point(x1, y1), new Point(x2, y2)));

if (p1.length != p2.length || p1.length == 1 && !eq(p1[0], p2[0])

|| p1.length == 2 && !(eq(p1[0], p2[0]) && eq(p1[1], p2[1]) || eq(p1[0], p2[1]) && eq(p1[1], p2[0])))

throw new RuntimeException();

}

}

static boolean eq(Point p1, Point p2) {

return !(fastHypot(p1.x - p2.x, p1.y - p2.y) > 1e-9);

}

}

# Line

import java.util.\*;

public class LineGeometry {

static final double EPS = 1e-10;

public static int sign(double a) {

return a < -EPS ? -1 : a > EPS ? 1 : 0;

}

public static class Point implements Comparable<Point> {

public double x, y;

public Point(double x, double y) {

this.x = x;

this.y = y;

}

public Point minus(Point b) {

return new Point(x - b.x, y - b.y);

}

public double cross(Point b) {

return x \* b.y - y \* b.x;

}

public double dot(Point b) {

return x \* b.x + y \* b.y;

}

public Point rotateCCW(double angle) {

return new Point(x \* Math.cos(angle) - y \* Math.sin(angle), x \* Math.sin(angle) + y \* Math.cos(angle));

}

@Override

public int compareTo(Point o) {

// return Double.compare(Math.atan2(y, x), Math.atan2(o.y, o.x));

return Double.compare(x, o.x) != 0 ? Double.compare(x, o.x) : Double.compare(y, o.y);

}

}

public static class Line {

public double a, b, c;

public Line(double a, double b, double c) {

this.a = a;

this.b = b;

this.c = c;

}

public Line(Point p1, Point p2) {

a = +(p1.y - p2.y);

b = -(p1.x - p2.x);

c = p1.x \* p2.y - p2.x \* p1.y;

}

public Point intersect(Line line) {

double d = a \* line.b - line.a \* b;

if (sign(d) == 0) {

return null;

}

double x = -(c \* line.b - line.c \* b) / d;

double y = -(a \* line.c - line.a \* c) / d;

return new Point(x, y);

}

}

// Returns -1 for clockwise, 0 for straight line, 1 for counterclockwise order

public static int orientation(Point a, Point b, Point c) {

Point AB = b.minus(a);

Point AC = c.minus(a);

return sign(AB.cross(AC));

}

public static boolean cw(Point a, Point b, Point c) {

return orientation(a, b, c) < 0;

}

public static boolean ccw(Point a, Point b, Point c) {

return orientation(a, b, c) > 0;

}

public static boolean isCrossIntersect(Point a, Point b, Point c, Point d) {

return orientation(a, b, c) \* orientation(a, b, d) < 0 && orientation(c, d, a) \* orientation(c, d, b) < 0;

}

public static boolean isCrossOrTouchIntersect(Point a, Point b, Point c, Point d) {

if (Math.max(a.x, b.x) < Math.min(c.x, d.x) - EPS || Math.max(c.x, d.x) < Math.min(a.x, b.x) - EPS

|| Math.max(a.y, b.y) < Math.min(c.y, d.y) - EPS || Math.max(c.y, d.y) < Math.min(a.y, b.y) - EPS) {

return false;

}

return orientation(a, b, c) \* orientation(a, b, d) <= 0 && orientation(c, d, a) \* orientation(c, d, b) <= 0;

}

public static double pointToLineDistance(Point p, Line line) {

return Math.abs(line.a \* p.x + line.b \* p.y + line.c) / fastHypot(line.a, line.b);

}

public static double fastHypot(double x, double y) {

return Math.sqrt(x \* x + y \* y);

}

public static double sqr(double x) {

return x \* x;

}

public static double angleBetween(Point a, Point b) {

return Math.atan2(a.cross(b), a.dot(b));

}

public static double angle(Line line) {

return Math.atan2(-line.a, line.b);

}

public static double signedArea(Point[] points) {

int n = points.length;

double area = 0;

for (int i = 0, j = n - 1; i < n; j = i++) {

area += (points[i].x - points[j].x) \* (points[i].y + points[j].y);

// area += points[i].x \* points[j].y - points[j].x \* points[i].y;

}

return area / 2;

}

public static enum Position {

LEFT, RIGHT, BEHIND, BEYOND, ORIGIN, DESTINATION, BETWEEN

}

// Classifies position of point p against vector a

public static Position classify(Point p, Point a) {

int s = sign(a.cross(p));

if (s > 0) {

return Position.LEFT;

}

if (s < 0) {

return Position.RIGHT;

}

if (sign(p.x) == 0 && sign(p.y) == 0) {

return Position.ORIGIN;

}

if (sign(p.x - a.x) == 0 && sign(p.y - a.y) == 0) {

return Position.DESTINATION;

}

if (a.x \* p.x < 0 || a.y \* p.y < 0) {

return Position.BEYOND;

}

if (a.x \* a.x + a.y \* a.y < p.x \* p.x + p.y \* p.y) {

return Position.BEHIND;

}

return Position.BETWEEN;

}

// cuts right part of poly (returns left part)

public static Point[] convexCut(Point[] poly, Point p1, Point p2) {

int n = poly.length;

List<Point> res = new ArrayList<>();

for (int i = 0, j = n - 1; i < n; j = i++) {

int d1 = orientation(p1, p2, poly[j]);

int d2 = orientation(p1, p2, poly[i]);

if (d1 >= 0)

res.add(poly[j]);

if (d1 \* d2 < 0)

res.add(new Line(p1, p2).intersect(new Line(poly[j], poly[i])));

}

return res.toArray(new Point[res.size()]);

}

// Usage example

public static void main(String[] args) {

}

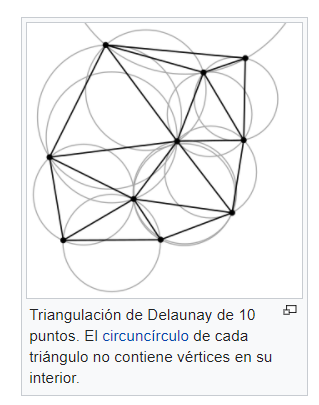
}

# Delaunay triangulation

Una triangulación de Delaunay (pronunciado /dəlo'ne/, a veces escrito fonéticamente «Deloné»), es una red de triángulos conexa y convexa que cumple la condición de Delaunay. Esta condición dice que la circunferencia circunscrita de cada triángulo de la red no debe contener ningún vértice de otro triángulo. Las triangulaciones de Delaunay tienen importante relevancia en el campo de la geometría computacional, especialmente en gráficos 3D por computadora.

Se le denomina así por el matemático ruso Borís Nikolaevich Delone quien lo ideó en 1934;1​ el mismo Delone usó la forma francesa de su apellido, «Delaunay», como apreciación a sus antecesores franceses.

https://es.wikipedia.org/wiki/Triangulaci%C3%B3n\_de\_Delaunay



Delaunay triangulation in O(N^4) (with demo)

http://sites.google.com/site/indy256/delaunay.jnlp

import java.awt.\*;

import java.util.\*;

import java.util.List;

import javax.swing.\*;

public class Delaunay extends JFrame {

public static Triangle[] triangulateDelaunay(long[] x, long[] y) {

int n = x.length;

long[] z = new long[n];

for (int i = 0; i < n; i++)

z[i] = x[i] \* x[i] + y[i] \* y[i];

List<Triangle> res = new ArrayList<Triangle>();

for (int i = 0; i < n - 2; i++) {

for (int j = i + 1; j < n; j++) {

m1: for (int k = i + 1; k < n; k++) {

if (j == k)

continue;

long nx = (y[j] - y[i]) \* (z[k] - z[i]) - (y[k] - y[i]) \* (z[j] - z[i]);

long ny = (x[k] - x[i]) \* (z[j] - z[i]) - (x[j] - x[i]) \* (z[k] - z[i]);

long nz = (x[j] - x[i]) \* (y[k] - y[i]) - (x[k] - x[i]) \* (y[j] - y[i]);

if (nz >= 0)

continue;

for (int m = 0; m < n; m++) {

long dot = (x[m] - x[i]) \* nx + (y[m] - y[i]) \* ny + (z[m] - z[i]) \* nz;

if (dot > 0)

continue m1;

}

// to deal with 4 points on a circle

int[] s1 = { i, j, k, i };

for (Triangle t : res) {

int[] s2 = { t.a, t.b, t.c, t.a };

for (int u = 0; u < 3; u++)

for (int v = 0; v < 3; v++)

if (isCrossIntersect(x[s1[u]], y[s1[u]], x[s1[u + 1]], y[s1[u + 1]], x[s2[v]],

y[s2[v]], x[s2[v + 1]], y[s2[v + 1]]))

continue m1;

}

res.add(new Triangle(i, j, k));

}

}

}

return res.toArray(new Triangle[0]);

}

static class Triangle {

int a, b, c;

public Triangle(int a, int b, int c) {

this.a = a;

this.b = b;

this.c = c;

}

}

static boolean isCrossIntersect(long x1, long y1, long x2, long y2, long x3, long y3, long x4, long y4) {

long z1 = (x3 - x1) \* (y2 - y1) - (y3 - y1) \* (x2 - x1);

long z2 = (x4 - x1) \* (y2 - y1) - (y4 - y1) \* (x2 - x1);

if (z1 < 0 && z2 < 0 || z1 > 0 && z2 > 0 || z1 == 0 || z2 == 0)

return false;

long z3 = (x1 - x3) \* (y4 - y3) - (y1 - y3) \* (x4 - x3);

long z4 = (x2 - x3) \* (y4 - y3) - (y2 - y3) \* (x4 - x3);

if (z3 < 0 && z4 < 0 || z3 > 0 && z4 > 0 || z3 == 0 || z4 == 0)

return false;

return true;

}

public Delaunay() {

// 4 points on a circle

// final long[] x = { 0, 0, 100, 100 };

// final long[] y = { 0, 100, 100, 0 };

int n = 100;

final long[] x = new long[n];

final long[] y = new long[n];

Random rnd = new Random();

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

x[i] = rnd.nextInt(500) + 1;

y[i] = rnd.nextInt(500) + 1;

}

final Triangle[] ts = triangulateDelaunay(x, y);

JPanel panel = new JPanel() {

protected void paintComponent(Graphics g) {

super.paintComponent(g);

((Graphics2D) g).setStroke(new BasicStroke(3));

g.setColor(Color.BLUE);

for (Triangle t : ts) {

g.drawLine((int) x[t.a], (int) y[t.a], (int) x[t.b], (int) y[t.b]);

g.drawLine((int) x[t.a], (int) y[t.a], (int) x[t.c], (int) y[t.c]);

g.drawLine((int) x[t.c], (int) y[t.c], (int) x[t.b], (int) y[t.b]);

}

g.setColor(Color.RED);

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

g.drawOval((int) x[i] - 1, (int) y[i] - 1, 3, 3);

}

}

};

setContentPane(panel);

}

public static void main(String[] args) {

JFrame frame = new Delaunay();

frame.setSize(new Dimension(800, 600));

frame.setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);

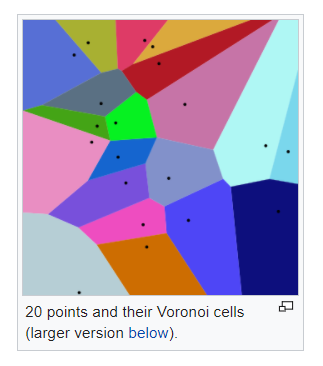
frame.setVisible(true);

}

}

# Voronoi Diagram

In mathematics, a Voronoi diagram is a partitioning of a plane into regions based on distance to points in a specific subset of the plane. That set of points (called seeds, sites, or generators) is specified beforehand, and for each seed there is a corresponding region consisting of all points closer to that seed than to any other. These regions are called Voronoi cells. The Voronoi diagram of a set of points is dual to its Delaunay triangulation.



Delaunay triangulation and Voronoi diagram in O(N\*sqrt(N)) (with demo)

https://sites.google.com/site/indy256/delaunayvoronoi.jnlp

import javax.swing.\*;

import java.awt.\*;

import java.awt.geom.Point2D;

import java.util.\*;

public class DelaunayVoronoi extends JFrame {

static class SubDivision {

List<QuadEdge> quadEdges = new ArrayList<QuadEdge>();

QuadEdge startingEdge;

static final double tolerance = 0.001;

Point.Double[] frameVertex = new Point.Double[3];

private QuadEdge lastEdge = null;

SubDivision(Collection<Point.Double> siteCoords) {

double minX = Double.POSITIVE\_INFINITY;

double maxX = Double.NEGATIVE\_INFINITY;

double minY = Double.POSITIVE\_INFINITY;

double maxY = Double.NEGATIVE\_INFINITY;

for (Point.Double p : siteCoords) {

minX = Math.min(minX, p.x);

maxX = Math.max(maxX, p.x);

minY = Math.min(minX, p.y);

maxY = Math.max(maxX, p.y);

}

double offset = Math.max(maxX - minX, maxX - minX) \* 10;

frameVertex[0] = new Point.Double((maxX + minX) / 2, maxY + offset);

frameVertex[1] = new Point.Double(minX - offset, minY - offset);

frameVertex[2] = new Point.Double(maxX + offset, minY - offset);

QuadEdge a = makeEdge(frameVertex[0], frameVertex[1]);

QuadEdge b = makeEdge(frameVertex[1], frameVertex[2]);

QuadEdge.splice(a.sym(), b);

QuadEdge c = makeEdge(frameVertex[2], frameVertex[0]);

QuadEdge.splice(b.sym(), c);

QuadEdge.splice(c.sym(), a);

startingEdge = a;

}

QuadEdge makeEdge(Point.Double o, Point.Double d) {

QuadEdge q = QuadEdge.makeEdge(o, d);

quadEdges.add(q);

return q;

}

QuadEdge connect(QuadEdge a, QuadEdge b) {

QuadEdge q = QuadEdge.connect(a, b);

quadEdges.add(q);

return q;

}

void delete(QuadEdge e) {

QuadEdge.splice(e, e.oPrev());

QuadEdge.splice(e.sym(), e.sym().oPrev());

QuadEdge eSym = e.sym();

QuadEdge eRot = e.rot();

QuadEdge eRotSym = e.rot().sym();

// this is inefficient on an ArrayList, but this method should be called infrequently

quadEdges.remove(e);

quadEdges.remove(eSym);

quadEdges.remove(eRot);

quadEdges.remove(eRotSym);

e.delete();

eSym.delete();

eRot.delete();

eRotSym.delete();

}

// The edge returned has the property that either v is on e, or e is an edge of a triangle containing v.

QuadEdge locate(Point.Double v) {

if (lastEdge == null || !lastEdge.isNotDeleted()) {

lastEdge = quadEdges.iterator().next();

}

QuadEdge e = lastEdge;

for (int iter = 0; ; iter++) {

if (iter > quadEdges.size()) {

throw new RuntimeException("Possible topology error");

}

if (v.equals(e.orig()) || v.equals(e.dest())) {

break;

} else if (rightOf(v, e)) {

e = e.sym();

} else if (!rightOf(v, e.oNext())) {

e = e.oNext();

} else if (!rightOf(v, e.dPrev())) {

e = e.dPrev();

} else {

// on edge or in triangle containing edge

// System.out.println("Locate count: " + iter);

break;

}

}

return e;

}

// Tests whether a QuadEdge is an edge incident on a frame triangle vertex

boolean isFrameEdge(QuadEdge e) {

return isFrameVertex(e.orig()) || isFrameVertex(e.dest());

}

boolean isFrameVertex(Point.Double v) {

return v.equals(frameVertex[0]) || v.equals(frameVertex[1]) || v.equals(frameVertex[2]);

}

static double distancePointLine(Point.Double p, Point.Double A, Point.Double B) {

if (A.x == B.x && A.y == B.y) return p.distance(A);

double r = ((p.x - A.x) \* (B.x - A.x) + (p.y - A.y) \* (B.y - A.y)) / ((B.x - A.x) \* (B.x - A.x) + (B.y - A.y) \* (B.y - A.y));

if (r <= 0.0) return p.distance(A);

if (r >= 1.0) return p.distance(B);

double s = ((A.y - p.y) \* (B.x - A.x) - (A.x - p.x) \* (B.y - A.y)) / ((B.x - A.x) \* (B.x - A.x) + (B.y - A.y) \* (B.y - A.y));

return Math.abs(s) \* Math.sqrt(((B.x - A.x) \* (B.x - A.x) + (B.y - A.y) \* (B.y - A.y)));

}

static boolean isOnEdge(QuadEdge e, Point.Double p) {

double dist = distancePointLine(p, e.orig(), e.dest());

double EDGE\_COINCIDENCE\_TOL\_FACTOR = 1000;

return dist < tolerance / EDGE\_COINCIDENCE\_TOL\_FACTOR;

}

boolean isEndPointOfEdge(QuadEdge e, Point.Double v) {

return equals(v, e.orig(), tolerance) || equals(v, e.dest(), tolerance);

}

// A TriangleVisitor which computes and sets the circumCenter as the origin of the dual edges originating in each triangle

static class TriangleCircumcentreVisitor implements TriangleVisitor {

public void visit(QuadEdge[] triEdges) {

Point.Double cc = circumCenter(triEdges[0].orig(), triEdges[1].orig(), triEdges[2].orig());

// save the circumCenter as the origin for the dual edges originating in this triangle

triEdges[0].rot().setOrig(cc);

triEdges[1].rot().setOrig(cc);

triEdges[2].rot().setOrig(cc);

}

static Point.Double circumCenter(Point.Double a, Point.Double b, Point.Double c) {

double Bx = b.x - a.x;

double By = b.y - a.y;

double Cx = c.x - a.x;

double Cy = c.y - a.y;

double d = 2 \* (Bx \* Cy - By \* Cx);

double z1 = Bx \* Bx + By \* By;

double z2 = Cx \* Cx + Cy \* Cy;

double cx = Cy \* z1 - By \* z2;

double cy = Bx \* z2 - Cx \* z1;

return new Point.Double(cx / d + a.x, cy / d + a.y);

}

}

// Stores the edges for a visited triangle. Also pushes sym (neighbour) edges on stack to visit later.

QuadEdge[] fetchTriangleToVisit(QuadEdge edge, Queue<QuadEdge> edgeStack, boolean includeFrame, Set<QuadEdge> visitedEdges) {

QuadEdge[] triEdges = new QuadEdge[3];

QuadEdge curr = edge;

int edgeCount = 0;

boolean isFrame = false;

do {

triEdges[edgeCount] = curr;

if (isFrameEdge(curr))

isFrame = true;

// push sym edges to visit next

QuadEdge sym = curr.sym();

if (!visitedEdges.contains(sym))

edgeStack.add(sym);

// mark this edge as visited

visitedEdges.add(curr);

edgeCount++;

curr = curr.lNext();

} while (curr != edge);

return isFrame && !includeFrame ? null : triEdges;

}

void visitTriangles(TriangleVisitor triVisitor, boolean includeFrame) {

Queue<QuadEdge> q = new ArrayDeque<QuadEdge>();

q.add(startingEdge);

Set<QuadEdge> visitedEdges = Collections.newSetFromMap(new IdentityHashMap<QuadEdge, Boolean>());

while (!q.isEmpty()) {

QuadEdge edge = q.remove();

if (!visitedEdges.contains(edge)) {

QuadEdge[] triEdges = fetchTriangleToVisit(edge, q, includeFrame, visitedEdges);

if (triEdges != null)

triVisitor.visit(triEdges);

}

}

}

/\*\*

\* Gets a collection of {@link QuadEdge}s whose origin

\* vertices are a unique set which includes

\* all vertices in the subdivision.

\* The frame vertices can be included if required.

\* <p/>

\* This is useful for algorithms which require traversing the

\* subdivision starting at all vertices.

\* Returning a quadedge for each vertex

\* is more efficient than

\* the alternative of finding the actual vertices

\* quadedges attached to them.

\*

\* @param includeFrame true if the frame vertices should be included

\* @return a collection of QuadEdge with the vertices of the subdivision as their origins

\*/

List<QuadEdge> getVertexUniqueEdges(boolean includeFrame) {

List<QuadEdge> edges = new ArrayList<QuadEdge>();

Set<Point.Double> visitedVertices = new HashSet<Point2D.Double>();

for (QuadEdge qe : quadEdges) {

Point.Double v = qe.orig();

if (!visitedVertices.contains(v)) {

visitedVertices.add(v);

if (includeFrame || !isFrameVertex(v)) {

edges.add(qe);

}

}

/\*\*

\* Inspect the sym edge as well, since it is

\* possible that a vertex is only at the

\* dest of all tracked quadedges.

\*/

QuadEdge qd = qe.sym();

Point.Double vd = qd.orig();

if (!visitedVertices.contains(vd)) {

visitedVertices.add(vd);

if (includeFrame || !isFrameVertex(vd)) {

edges.add(qd);

}

}

}

return edges;

}

// Gets the coordinates for each triangle in the subdivision as an array

List<Point.Double[]> getTriangleCoordinates() {

TriangleCoordinatesVisitor visitor = new TriangleCoordinatesVisitor();

visitTriangles(visitor, false);

return visitor.getTriangles();

}

static class TriangleCoordinatesVisitor implements TriangleVisitor {

private List<Point.Double[]> triCoords = new ArrayList<Point.Double[]>();

public void visit(QuadEdge[] triEdges) {

Point.Double[] coords = new Point.Double[4];

for (int i = 0; i < 3; i++) {

coords[i] = triEdges[i].orig();

}

coords[3] = coords[0];

triCoords.add(coords);

}

List<Point.Double[]> getTriangles() {

return triCoords;

}

}

/\*\*

\* Gets a List of {@link Polygon}s for the Voronoi cells

\* of this triangulation.

\* <p/>

\* The userData of each polygon is set to be the {@link Point.Double)

\* of the cell site. This allows easily associating external

\* data associated with the sites to the cells.

\*

\* @param geomFact a geometry factory

\* @return a List of Polygons

\*/

List<Point.Double[]> getVoronoiCellPolygons() {

/\*

\* Compute circumcentres of triangles as vertices for dual edges.

\* Precomputing the circumcentres is more efficient,

\* and more importantly ensures that the computed centres

\* are consistent across the Voronoi cells.

\*/

visitTriangles(new TriangleCircumcentreVisitor(), true);

List<Point.Double[]> cells = new ArrayList<Point.Double[]>();

for (QuadEdge qe : getVertexUniqueEdges(false)) {

cells.add(getVoronoiCellPolygon(qe));

}

return cells;

}

/\*\*

\* Gets the Voronoi cell around a site specified

\* by the origin of a QuadEdge.

\* <p/>

\* The userData of the polygon is set to be the {@link Point.Double)

\* of the site. This allows attaching external

\* data associated with the site to this cell polygon.

\*

\* @param qe a quadedge originating at the cell site

\* @param geomFact a factory for building the polygon

\* @return a polygon indicating the cell extent

\*/

Point.Double[] getVoronoiCellPolygon(QuadEdge qe) {

List<Point.Double> coordList = new ArrayList<Point.Double>();

QuadEdge startQE = qe;

do {

Point.Double cc = qe.rot().orig();

coordList.add(cc);

// move to next triangle CW around vertex

qe = qe.oPrev();

} while (qe != startQE);

coordList.add(coordList.get(0));

if (coordList.size() < 4) {

System.out.println(coordList);

coordList.add(coordList.get(coordList.size() - 1));

}

Point.Double v = startQE.orig();

Point.Double[] pts = coordList.toArray(new Point.Double[0]);

return pts;

}

interface TriangleVisitor {

void visit(QuadEdge[] triEdges);

}

/\*\*

\* Inserts a new point into a subdivision representing a Delaunay triangulation,

\* and fixes the affected edges so that the result is still a Delaunay triangulation

\*

\* @return a quadedge containing the inserted vertex

\*/

QuadEdge insertSite(Point.Double v) {

/\*\*

\* This code is based on Guibas and Stolfi (1985), with minor modifications

\* and a bug fix from Dani Lischinski (Graphic Gems 1993). (The modification

\* I believe is the test for the inserted site falling exactly on an

\* existing edge. Without this test zero-width triangles have been observed

\* to be created)

\*/

QuadEdge e = locate(v);

if (isEndPointOfEdge(e, v)) {

// point is already in subdivision.

return e;

} else if (isOnEdge(e, v)) {

// the point lies exactly on an edge, so delete the edge

// (it will be replaced by a pair of edges which have the point as a vertex)

e = e.oPrev();

delete(e.oNext());

}

// Connect the new point to the vertices of the containing triangle (or quadrilateral, if the new point fell on an existing edge.)

QuadEdge base = makeEdge(e.orig(), v);

QuadEdge.splice(base, e);

QuadEdge startEdge = base;

do {

base = connect(e, base.sym());

e = base.oPrev();

} while (e.lNext() != startEdge);

// Examine suspect edges to ensure that the Delaunay condition is satisfied.

do {

QuadEdge t = e.oPrev();

if (rightOf(t.dest(), e) && isInCircle(e.orig(), t.dest(), e.dest(), v)) {

QuadEdge.swap(e);

e = e.oPrev();

} else if (e.oNext() == startEdge) {

return base; // no more suspect edges.

} else {

e = e.oNext().lPrev();

}

} while (true);

}

static boolean equals(Point.Double p1, Point.Double p2, double tolerance) {

return p1.distance(p2) < tolerance;

}

static boolean isInCircle(Point.Double a, Point.Double b, Point.Double c, Point.Double p) {

double adx = a.x - p.x;

double ady = a.y - p.y;

double bdx = b.x - p.x;

double bdy = b.y - p.y;

double cdx = c.x - p.x;

double cdy = c.y - p.y;

double abdet = adx \* bdy - bdx \* ady;

double bcdet = bdx \* cdy - cdx \* bdy;

double cadet = cdx \* ady - adx \* cdy;

double alift = adx \* adx + ady \* ady;

double blift = bdx \* bdx + bdy \* bdy;

double clift = cdx \* cdx + cdy \* cdy;

double disc = alift \* bcdet + blift \* cadet + clift \* abdet;

return disc > 0;

}

static boolean isCCW(Point.Double a, Point.Double b, Point.Double c) {

return (b.x - a.x) \* (c.y - a.y) - (b.y - a.y) \* (c.x - a.x) > 0;

}

static boolean rightOf(Point.Double p, QuadEdge e) {

return isCCW(p, e.dest(), e.orig());

}

}

// Guibas and Stolfi,"Primitives for the manipulation of general subdivisions and the computation of Voronoi diagrams"

static class QuadEdge {

QuadEdge next; // next CCW edge

QuadEdge rot; // the dual of this edge, directed from right to left

Point.Double vertex; // The vertex that this edge represents

static QuadEdge makeEdge(Point.Double o, Point.Double d) {

QuadEdge q0 = new QuadEdge();

QuadEdge q1 = new QuadEdge();

QuadEdge q2 = new QuadEdge();

QuadEdge q3 = new QuadEdge();

q0.rot = q1;

q1.rot = q2;

q2.rot = q3;

q3.rot = q0;

q0.setOnext(q0);

q1.setOnext(q3);

q2.setOnext(q2);

q3.setOnext(q1);

q0.setOrig(o);

q0.setDest(d);

return q0;

}

/\*\*

\* Creates a new QuadEdge connecting the destination of a to the origin of

\* b, in such a way that all three have the same left face after the connection is complete.

\*/

static QuadEdge connect(QuadEdge a, QuadEdge b) {

QuadEdge e = makeEdge(a.dest(), b.orig());

splice(e, a.lNext());

splice(e.sym(), b);

return e;

}

/\*\*

\* Splices two edges together or apart.

\* Splice affects the two edge rings around the origins of a and b, and, independently, the two

\* edge rings around the left faces of a and b.

\* In each case, (i) if the two rings are distinct,

\* Splice will combine them into one, or (ii) if the two are the same ring, Splice will break it

\* into two separate pieces. Thus, Splice can be used both to attach the two edges together, and

\* to break them apart.

\*/

static void splice(QuadEdge a, QuadEdge b) {

QuadEdge alpha = a.oNext().rot();

QuadEdge beta = b.oNext().rot();

QuadEdge t1 = b.oNext();

QuadEdge t2 = a.oNext();

QuadEdge t3 = beta.oNext();

QuadEdge t4 = alpha.oNext();

a.setOnext(t1);

b.setOnext(t2);

alpha.setOnext(t3);

beta.setOnext(t4);

}

// Turns an edge counterclockwise inside its enclosing quadrilateral

static void swap(QuadEdge e) {

QuadEdge a = e.oPrev();

QuadEdge b = e.sym().oPrev();

splice(e, a);

splice(e.sym(), b);

splice(e, a.lNext());

splice(e.sym(), b.lNext());

e.setOrig(a.dest());

e.setDest(b.dest());

}

void delete() {

rot = null;

}

boolean isNotDeleted() {

return rot != null;

}

void setOnext(QuadEdge next) {

this.next = next;

}

QuadEdge rot() {

return rot;

}

QuadEdge sym() {

return rot.rot;

}

QuadEdge invRot() {

return rot.rot.rot;

}

QuadEdge oNext() {

return next;

}

QuadEdge oPrev() {

return rot.next.rot;

}

QuadEdge dNext() {

return sym().next.sym();

}

QuadEdge dPrev() {

return invRot().next.invRot();

}

QuadEdge lNext() {

return invRot().next.rot;

}

QuadEdge lPrev() {

return next.sym();

}

QuadEdge rNext() {

return rot.next.invRot();

}

QuadEdge rPrev() {

return sym().next;

}

void setOrig(Point.Double o) {

vertex = o;

}

void setDest(Point.Double d) {

sym().setOrig(d);

}

Point.Double orig() {

return vertex;

}

Point.Double dest() {

return sym().orig();

}

}

// visualization

Random rnd = new Random(1);

JPanel panel;

List<Point.Double> points = new ArrayList<Point.Double>();

List<Point2D.Double[]> tr = Collections.emptyList();

List<Point2D.Double[]> voronoiCellPolygons = Collections.emptyList();

public DelaunayVoronoi() throws HeadlessException {

int n = 100;

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

int x = rnd.nextInt(950) + 1;

int y = rnd.nextInt(700) + 1;

Point.Double c = new Point.Double(x, y);

points.add(c);

}

panel = new JPanel() {

protected void paintComponent(Graphics g) {

super.paintComponent(g);

Graphics2D g2 = ((Graphics2D) g);

g2.setRenderingHint(RenderingHints.KEY\_ANTIALIASING, RenderingHints.VALUE\_ANTIALIAS\_ON);

g2.setStroke(new BasicStroke(1));

g.setColor(Color.BLUE);

for (Object o : tr) {

Point.Double[] a = (Point.Double[]) o;

for (int i = 0; i + 1 < a.length; i++) {

g.drawLine((int) a[i].x, (int) a[i].y, (int) a[i + 1].x, (int) a[i + 1].y);

}

}

g2.setStroke(new BasicStroke(3));

g.setColor(Color.BLACK);

for (Object o : voronoiCellPolygons) {

Point.Double[] a = (Point.Double[]) o;

for (int i = 0; i + 1 < a.length; i++) {

g.drawLine((int) a[i].x, (int) a[i].y, (int) a[i + 1].x, (int) a[i + 1].y);

}

}

g.setColor(Color.RED);

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

g.drawOval((int) points.get(i).x - 2, (int) points.get(i).y - 2, 5, 5);

}

}

};

setContentPane(panel);

}

public static void main(String[] args) {

final DelaunayVoronoi dv = new DelaunayVoronoi();

dv.setSize(new Dimension(1024, 768));

dv.setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);

dv.setVisible(true);

new Thread() {

public void run() {

while (true) {

final Point2D.Double pivot = dv.points.get(0);

pivot.x += dv.rnd.nextInt(4);

pivot.y += dv.rnd.nextInt(2);

pivot.x %= 950;

pivot.y %= 700;

Set<Point.Double> uniquePoints = new HashSet<Point2D.Double>(dv.points);

SubDivision subdivision = new SubDivision(uniquePoints);

for (Point.Double p : uniquePoints) subdivision.insertSite(p);

dv.tr = subdivision.getTriangleCoordinates();

dv.voronoiCellPolygons = subdivision.getVoronoiCellPolygons();

dv.panel.repaint();

try {

Thread.sleep(15);

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

}.start();

}

}

# Treap 1

Treap as a set with kth-element operation

import java.util.\*;

public class TreapSimple {

static Random random = new Random();

static class Treap {

int key;

long prio;

Treap left;

Treap right;

int count;

Treap(int key) {

this.key = key;

prio = random.nextLong();

count = 1;

}

void update() {

count = 1 + getCount(left) + getCount(right);

}

}

static int getCount(Treap root) {

return root == null ? 0 : root.count;

}

static class TreapPair {

Treap left;

Treap right;

TreapPair(Treap left, Treap right) {

this.left = left;

this.right = right;

}

}

static TreapPair split(Treap root, int minRight) {

if (root == null)

return new TreapPair(null, null);

if (root.key >= minRight) {

TreapPair leftSplit = split(root.left, minRight);

root.left = leftSplit.right;

root.update();

leftSplit.right = root;

return leftSplit;

} else {

TreapPair rightSplit = split(root.right, minRight);

root.right = rightSplit.left;

root.update();

rightSplit.left = root;

return rightSplit;

}

}

static Treap merge(Treap left, Treap right) {

if (left == null)

return right;

if (right == null)

return left;

if (left.prio > right.prio) {

left.right = merge(left.right, right);

left.update();

return left;

} else {

right.left = merge(left, right.left);

right.update();

return right;

}

}

static Treap insert(Treap root, int x) {

TreapPair t = split(root, x);

return merge(merge(t.left, new Treap(x)), t.right);

}

static Treap remove(Treap root, int x) {

if (root == null) {

return null;

}

if (x < root.key) {

root.left = remove(root.left, x);

root.update();

return root;

} else if (x > root.key) {

root.right = remove(root.right, x);

root.update();

return root;

} else {

return merge(root.left, root.right);

}

}

static int kth(Treap root, int k) {

if (k < getCount(root.left))

return kth(root.left, k);

else if (k > getCount(root.left))

return kth(root.right, k - getCount(root.left) - 1);

return root.key;

}

static void print(Treap root) {

if (root == null)

return;

print(root.left);

System.out.println(root.key);

print(root.right);

}

// random test

public static void main(String[] args) {

Treap treap = null;

Set<Integer> set = new TreeSet<>();

for (int i = 0; i < 100000; i++) {

int x = random.nextInt(100000);

if (random.nextBoolean()) {

treap = remove(treap, x);

set.remove(x);

} else if (!set.contains(x)) {

treap = insert(treap, x);

set.add(x);

}

if (set.size() != getCount(treap))

throw new RuntimeException();

}

// print(treap);

}

}

# Treap 2

Treap with implicit key with interval modification

import java.util.\*;

public class TreapImplicitKey {

// Modify the following 5 methods to implement your custom operations on the tree.

// This example implements Add/Max operations. Operations like Add/Sum, Set/Max can also be implemented.

static int modifyOperation(int x, int y) {

return x + y;

}

// query (or combine) operation

static int queryOperation(int leftValue, int rightValue) {

return Math.max(leftValue, rightValue);

}

static int deltaEffectOnSegment(int delta, int segmentLength) {

// Here you must write a fast equivalent of following slow code:

// int result = delta;

// for (int i = 1; i < segmentLength; i++) result = queryOperation(result, delta);

// return result;

return delta;

}

static int getNeutralDelta() {

return 0;

}

static int getNeutralValue() {

return Integer.MIN\_VALUE;

}

// generic code

static Random random = new Random();

static int joinValueWithDelta(int value, int delta) {

if (delta == getNeutralDelta()) return value;

return modifyOperation(value, delta);

}

static int joinDeltas(int delta1, int delta2) {

if (delta1 == getNeutralDelta()) return delta2;

if (delta2 == getNeutralDelta()) return delta1;

return modifyOperation(delta1, delta2);

}

static void applyDelta(Treap root, int delta) {

if (root == null)

return;

root.delta = joinDeltas(root.delta, delta);

root.nodeValue = joinValueWithDelta(root.nodeValue, delta);

root.subTreeValue = joinValueWithDelta(root.subTreeValue, deltaEffectOnSegment(delta, root.count));

}

static void pushDelta(Treap root) {

if (root == null)

return;

applyDelta(root.left, root.delta);

applyDelta(root.right, root.delta);

root.delta = getNeutralDelta();

}

public static class Treap {

int nodeValue;

int subTreeValue;

int delta;

int count;

long prio;

Treap left;

Treap right;

Treap(int value) {

nodeValue = value;

subTreeValue = value;

delta = getNeutralDelta();

count = 1;

prio = random.nextLong();

}

void update() {

subTreeValue = queryOperation(queryOperation(getSubTreeValue(left), nodeValue), getSubTreeValue(right));

count = 1 + getCount(left) + getCount(right);

}

}

static int getCount(Treap root) {

return root == null ? 0 : root.count;

}

static int getSubTreeValue(Treap root) {

return root == null ? getNeutralValue() : root.subTreeValue;

}

public static class TreapPair {

Treap left;

Treap right;

TreapPair(Treap left, Treap right) {

this.left = left;

this.right = right;

}

}

public static TreapPair split(Treap root, int minRight) {

if (root == null)

return new TreapPair(null, null);

pushDelta(root);

if (getCount(root.left) >= minRight) {

TreapPair sub = split(root.left, minRight);

root.left = sub.right;

root.update();

sub.right = root;

return sub;

} else {

TreapPair sub = split(root.right, minRight - getCount(root.left) - 1);

root.right = sub.left;

root.update();

sub.left = root;

return sub;

}

}

public static Treap merge(Treap left, Treap right) {

pushDelta(left);

pushDelta(right);

if (left == null)

return right;

if (right == null)

return left;

if (left.prio > right.prio) {

left.right = merge(left.right, right);

left.update();

return left;

} else {

right.left = merge(left, right.left);

right.update();

return right;

}

}

public static Treap insert(Treap root, int index, int value) {

TreapPair t = split(root, index);

return merge(merge(t.left, new Treap(value)), t.right);

}

public static Treap remove(Treap root, int index) {

TreapPair t = split(root, index);

return merge(t.left, split(t.right, index + 1 - getCount(t.left)).right);

}

public static Treap modify(Treap root, int a, int b, int delta) {

TreapPair t1 = split(root, b + 1);

TreapPair t2 = split(t1.left, a);

applyDelta(t2.right, delta);

return merge(merge(t2.left, t2.right), t1.right);

}

public static class TreapAndResult {

Treap treap;

int value;

TreapAndResult(Treap t, int value) {

this.treap = t;

this.value = value;

}

}

public static TreapAndResult query(Treap root, int a, int b) {

TreapPair t1 = split(root, b + 1);

TreapPair t2 = split(t1.left, a);

int value = getSubTreeValue(t2.right);

return new TreapAndResult(merge(merge(t2.left, t2.right), t1.right), value);

}

public static void print(Treap root) {

if (root == null)

return;

pushDelta(root);

print(root.left);

System.out.print(root.nodeValue + " ");

print(root.right);

}

// Random test

public static void main(String[] args) {

Treap treap = null;

List<Integer> list = new ArrayList<>();

Random rnd = new Random();

for (int step = 0; step < 100000; step++) {

int cmd = rnd.nextInt(6);

if (cmd < 2 && list.size() < 100) {

int pos = rnd.nextInt(list.size() + 1);

int delta = rnd.nextInt(100);

list.add(pos, delta);

treap = insert(treap, pos, delta);

} else if (cmd < 3 && list.size() > 0) {

int pos = rnd.nextInt(list.size());

list.remove(pos);

treap = remove(treap, pos);

} else if (cmd < 4 && list.size() > 0) {

int b = rnd.nextInt(list.size());

int a = rnd.nextInt(b + 1);

int res = list.get(a);

for (int i = a + 1; i <= b; i++)

res = queryOperation(res, list.get(i));

TreapAndResult tr = query(treap, a, b);

treap = tr.treap;

if (res != tr.value) {

System.out.println(list);

print(treap);

return;

}

} else if (cmd < 5 && list.size() > 0) {

int b = rnd.nextInt(list.size());

int a = rnd.nextInt(b + 1);

int delta = rnd.nextInt(100) - 50;

for (int i = a; i <= b; i++)

list.set(i, joinValueWithDelta(list.get(i), delta));

treap = modify(treap, a, b, delta);

} else {

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

TreapAndResult tr = query(treap, i, i);

treap = tr.treap;

int v = tr.value;

if (list.get(i) != v) {

System.out.println(list);

print(treap);

return;

}

}

}

}

System.out.println("Test passed");

}

}

# Treap 3

struct item {

int key, prior;

item \* l, \* r;

item() { }

item (int key, int prior) : key(key), prior(prior), l(NULL), r(NULL) { }

};

typedef item \* pitem;

void split (pitem t, int key, pitem & l, pitem & r) {

if (!t)

l = r = NULL;

else if (key < t->key)

split (t->l, key, l, t->l), r = t;

else

split (t->r, key, t->r, r), l = t;

}

void insert (pitem & t, pitem it) {

if (!t)

t = it;

else if (it->prior > t->prior)

split (t, it->key, it->l, it->r), t = it;

else

insert (it->key < t->key ? t->l : t->r, it);

}

void merge (pitem & t, pitem l, pitem r) {

if (!l || !r)

t = l ? l : r;

else if (l->prior > r->prior)

merge (l->r, l->r, r), t = l;

else

merge (r->l, l, r->l), t = r;

}

void erase (pitem & t, int key) {

if (t->key == key)

merge (t, t->l, t->r);

else

erase (key < t->key ? t->l : t->r, key);

}

pitem unite (pitem l, pitem r) {

if (!l || !r) return l ? l : r;

if (l->prior < r->prior) swap (l, r);

pitem lt, rt;

split (r, l->key, lt, rt);

l->l = unite (l->l, lt);

l->r = unite (l->r, rt);

return l;

}

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

int cnt (pitem t) {

return t ? t->cnt : 0;

}

void upd\_cnt (pitem t) {

if (t)

t->cnt = 1 + cnt(t->l) + cnt (t->r);

}

void merge (pitem & t, pitem l, pitem r) {

if (!l || !r)

t = l ? l : r;

else if (l->prior > r->prior)

merge (l->r, l->r, r), t = l;

else

merge (r->l, l, r->l), t = r;

upd\_cnt (t);

}

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

void split (pitem t, pitem & l, pitem & r, int key, int add = 0) {

if (!t)

return void( l = r = 0 );

int cur\_key = add + cnt(t->l); // вычисляем неявный ключ

if (key <= cur\_key)

split (t->l, l, t->l, key, add), r = t;

else

split (t->r, t->r, r, key, add + 1 + cnt(t->l)), l = t;

upd\_cnt (t);

}

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

typedef struct item \* pitem;

struct item {

int prior, value, cnt;

bool rev;

pitem l, r;

};

int cnt (pitem it) {

return it ? it->cnt : 0;

}

void upd\_cnt (pitem it) {

if (it)

it->cnt = cnt(it->l) + cnt(it->r) + 1;

}

void push (pitem it) {

if (it && it->rev) {

it->rev = false;

swap (it->l, it->r);

if (it->l) it->l->rev ^= true;

if (it->r) it->r->rev ^= true;

}

}

void merge (pitem & t, pitem l, pitem r) {

push (l);

push (r);

if (!l || !r)

t = l ? l : r;

else if (l->prior > r->prior)

merge (l->r, l->r, r), t = l;

else

merge (r->l, l, r->l), t = r;

upd\_cnt (t);

}

void split (pitem t, pitem & l, pitem & r, int key, int add = 0) {

if (!t)

return void( l = r = 0 );

push (t);

int cur\_key = add + cnt(t->l);

if (key <= cur\_key)

split (t->l, l, t->l, key, add), r = t;

else

split (t->r, t->r, r, key, add + 1 + cnt(t->l)), l = t;

upd\_cnt (t);

}

void reverse (pitem t, int l, int r) {

pitem t1, t2, t3;

split (t, t1, t2, l);

split (t2, t2, t3, r-l+1);

t2->rev ^= true;

merge (t, t1, t2);

merge (t, t, t3);

}

void output (pitem t) {

if (!t) return;

push (t);

output (t->l);

printf ("%d ", t->value);

output (t->r);

}

# Spase Table

public class RmqSparseTable {

int[] logTable;

int[][] rmq;

int[] a;

public RmqSparseTable(int[] a) {

this.a = a;

int n = a.length;

logTable = new int[n + 1];

for (int i = 2; i <= n; i++)

logTable[i] = logTable[i >> 1] + 1;

rmq = new int[logTable[n] + 1][n];

for (int i = 0; i < n; ++i)

rmq[0][i] = i;

for (int k = 1; (1 << k) < n; ++k) {

for (int i = 0; i + (1 << k) <= n; i++) {

int x = rmq[k - 1][i];

int y = rmq[k - 1][i + (1 << k - 1)];

rmq[k][i] = a[x] <= a[y] ? x : y;

}

}

}

public int minPos(int i, int j) {

int k = logTable[j - i];

int x = rmq[k][i];

int y = rmq[k][j - (1 << k) + 1];

return a[x] <= a[y] ? x : y;

}

public static void main(String[] args) {

int[] a = { 1, 5, -2, 3 };

RmqSparseTable st = new RmqSparseTable(a);

System.out.println(2 == st.minPos(0, 3));

}

}

# Coloreado de Grafos

import java.util.\*;

public class GraphColoringGreedy {

// similar to DSatur coloring

public static int[] color(List<Integer>[] graph) {

int n = graph.length;

BitSet[] used = new BitSet[n];

int[] colors = new int[n];

PriorityQueue<Long> q = new PriorityQueue<>(n);

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

used[i] = new BitSet();

colors[i] = -1;

q.add((long) i);

}

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

int bestu;

while (true) {

bestu = q.remove().intValue();

if (colors[bestu] == -1)

break;

}

int c = used[bestu].nextClearBit(0);

colors[bestu] = c;

for (int v : graph[bestu]) {

if (!used[v].get(c)) {

used[v].set(c);

if (colors[v] == -1)

q.add(v - ((long) used[v].cardinality() << 32));

}

}

}

return colors;

}

// Usage example

public static void main(String[] args) {

int n = 5;

List<Integer>[] g = new List[n];

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

g[i] = new ArrayList<>();

}

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

for (int j = 0; j < n; j++) {

g[i].add((i + 1) % n);

g[(i + 1) % n].add(i);

}

}

System.out.println(Arrays.toString(color(g)));

}

}

# Shortest paths. Bellman–Ford algorithm in O(V\*E). Negative cycle detection.

import java.util.\*;

public class BellmanFord {

static final int INF = Integer.MAX\_VALUE / 2;

public static class Edge {

int v, cost;

public Edge(int v, int cost) {

this.v = v;

this.cost = cost;

}

}

public static boolean bellmanFord(List<Edge>[] graph, int s, int[] dist, int[] pred) {

Arrays.fill(pred, -1);

Arrays.fill(dist, INF);

dist[s] = 0;

int n = graph.length;

boolean updated = false;

for (int step = 0; step < n; step++) {

updated = false;

for (int u = 0; u < n; u++) {

if (dist[u] == INF) continue;

for (Edge e : graph[u]) {

if (dist[e.v] > dist[u] + e.cost) {

dist[e.v] = dist[u] + e.cost;

dist[e.v] = Math.max(dist[e.v], -INF);

pred[e.v] = u;

updated = true;

}

}

}

if (!updated)

break;

}

// if updated is true then a negative cycle exists

return updated == false;

}

public static int[] findNegativeCycle(List<Edge>[] graph) {

int n = graph.length;

int[] pred = new int[n];

Arrays.fill(pred, -1);

int[] dist = new int[n];

int last = -1;

for (int step = 0; step < n; step++) {

last = -1;

for (int u = 0; u < n; u++) {

if (dist[u] == INF) continue;

for (Edge e : graph[u]) {

if (dist[e.v] > dist[u] + e.cost) {

dist[e.v] = Math.max(dist[u] + e.cost, -INF);

dist[e.v] = Math.max(dist[e.v], -INF);

pred[e.v] = u;

last = e.v;

}

}

}

if (last == -1)

return null;

}

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

last = pred[last];

}

int[] p = new int[n];

int cnt = 0;

for (int u = last; u != last || cnt == 0; u = pred[u]) {

p[cnt++] = u;

}

int[] cycle = new int[cnt];

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

cycle[i] = p[--cnt];

}

return cycle;

}

// Usage example

public static void main(String[] args) {

List<Edge>[] graph = new List[4];

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

graph[i] = new ArrayList<>();

}

graph[0].add(new Edge(1, 1));

graph[1].add(new Edge(0, 1));

graph[1].add(new Edge(2, 1));

graph[2].add(new Edge(3, -10));

graph[3].add(new Edge(1, 1));

int[] cycle = findNegativeCycle(graph);

System.out.println(Arrays.toString(cycle));

}

}

# Dinitz in bipartite matching (attacking rooks)

#include <bits/stdc++.h>

using namespace std;

#define FOR(i, a, b) for(int i = int(a); i < int(b); i++)

#define RFOR(i, a, b) for(int i = int(a); i >= int(b); i--)

#define pb push\_back

typedef pair<int, int> ii;

typedef vector<int> vi ;

typedef vector<ii> vii ;

typedef long long ll ;

#define PI acos(0)

#define INF 100000000

#define MAXN 20005

#define MAXS 10000

#define src 0

#define dest 20001

int n;

int dist[MAXN], q[MAXN], work[MAXN];

struct Edge {

int to, rev;

int f = 0, cap;

Edge() {}

Edge(int t, int r, int c) :

to(t), rev(r), cap(c) {}

};

vector<Edge> graph[MAXN];

void addEdge(int s, int t, int cap) {

graph[s].pb(Edge(t, graph[t].size(), cap));

graph[t].pb(Edge(s, graph[s].size() - 1, 0));

}

bool dinitz\_bfs() {

memset(dist, -1, sizeof(dist));

dist[src] = 0;

int qt = 0;

q[qt++] = src;

FOR(qh, 0, qt) {

int u = q[qh];

FOR(j, 0, graph[u].size()) {

Edge &e = graph[u][j];

int v = e.to;

if (dist[v] < 0 && e.f < e.cap) {

dist[v] = dist[u] + 1;

q[qt++] = v;

}

}

}

return dist[dest] >= 0;

}

int dinitz\_dfs(int u, int f) {

if (u == dest) return f;

for (int &i = work[u]; i < (int) graph[u].size(); i++) {

Edge &e = graph[u][i];

if (e.cap <= e.f) continue;

int v = e.to;

if (dist[v] == dist[u] + 1) {

int df = dinitz\_dfs(v, min(f, e.cap - e.f));

if (df > 0) {

e.f += df;

graph[v][e.rev].f -= df;

return df;

}

}

}

return 0;

}

int maxFlow() {

int result = 0;

while (dinitz\_bfs()) {

memset(work, 0, sizeof(work));

while (int delta = dinitz\_dfs(src, INT\_MAX))

result += delta;

}

return result;

}

ii g[105][105];

string mat[105];

void create() {

memset(g, 0, sizeof(g));

int iniC = 1, iniF = MAXS+1;

FOR(i, 0, n) {

FOR(j, 0, n) {

if (mat[i][j] == '.') {

g[i][j].first = iniC;

}

else {

g[i][j].first = -1;

iniC ++;

}

if (mat[j][i] == '.') {

g[j][i].second = iniF;

}

else {

g[j][i].second = -1;

iniF ++;

}

}

iniC ++, iniF ++;

}

FOR(i, 0, n) {

FOR(j, 0, n) {

// cout << g[i][j].first << " ";

if(mat[i][j] == '.') {

addEdge(g[i][j].first, g[i][j].second, 1);

}

}

// cout << endl;

}

}

int main() {

while (cin >> n) {

FOR(i, 0, MAXN) graph[i].clear();

FOR(i, 0, n) cin >> mat[i];

FOR(i, 0, MAXS) {

addEdge(0, i+1, 1);

addEdge(MAXS+i+1, dest, 1);

}

create();

cout << maxFlow() << endl;

}

return 0;

}