**Contents**

[Math 1](#_Toc403503710)

[Công thức bị ném xiên  x = (v₀cosa)t  y = (v₀sina)t - (gt²)/2  y = ( -gx²) / ( 2v²₀cos²a) + (tana)x  Tầm bay cao  H = v²₀sin²a / 2g  Tầm bay xa  L = v²₀sin2a / g 2](#_Toc403503711)

[**Linear Diophantine Equations** 2](#_Toc403503712)

[**Factorization in O(sqrt(N))** 4](#_Toc403503713)

[**Geometry: Circle** 8](#_Toc403503714)

[Arrays 13](#_Toc403503715)

[**LCS** 13](#_Toc403503716)

[**Longest increasing subsequence in O(N \* logN)** 13](#_Toc403503717)

[**Dijstra** 14](#_Toc403503718)

[**Tarjan’s** 15](#_Toc403503719)

[**Biconnected components, bridges and cut points** 16](#_Toc403503720)

[**Greedy graph coloring in O(E \* logV)** 18](#_Toc403503721)

[**Shortest paths. Floyd–Warshall algorithm in O(V^3)** 19](#_Toc403503722)

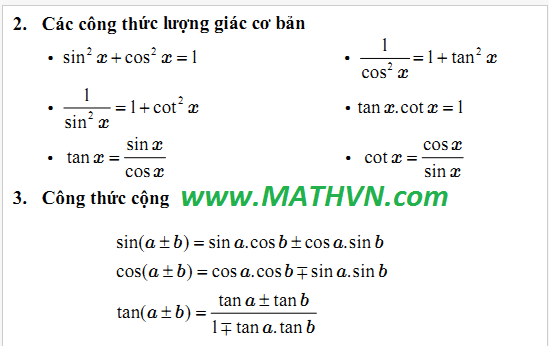
[**Shortest paths. Bellman–Ford algorithm in O(V\*E). Negative cycle.** 19](#_Toc403503723)

[**Maximum matching for bipartite graph. Kuhn's algorithm in O(E\*V)** 21](#_Toc403503724)

[**Prime numbers, sieve of Eratosthenes, Euler's totient function** 22](#_Toc403503725)

[**Enumeration of permutations** 23](#_Toc403503726)

# Math



# Công thức bị ném xiên  x = (v₀cosa)t  y = (v₀sina)t - (gt²)/2  y = ( -gx²) / ( 2v²₀cos²a) + (tana)x  Tầm bay cao  H = v²₀sin²a / 2g  Tầm bay xa  L = v²₀sin2a / g

## **Linear Diophantine Equations**

  // returns { gcd(a,b), x, y } such that gcd(a,b) = a\*x + b\*y  
  **public static long**[] euclid(**long**a, **long**b) {  
    **long**x = 1, y = 0, x1 = 0, y1 = 1, t;  
    **while**(b != 0) {  
      **long**q = a / b;  
      t = x;  
      x = x1;  
      x1 = t - q \* x1;  
      t = y;  
      y = y1;  
      y1 = t - q \* y1;  
      t = b;  
      b = a - q \* b;  
      a = t;  
    }  
    **return**a > 0 ? **new long**[]{a, x, y} : **new long**[]{-a, -x, -y};  
  }

**Number theory**

**private** **static** BigInteger fastFibonacciDoubling(**int** n) {

BigInteger a = BigInteger.ZERO;

BigInteger b = BigInteger.ONE;

**int** m = 0;

**for** (**int** i = 31 - Integer.*numberOfLeadingZeros*(n); i >= 0; i--) {

// Loop invariant: a = F(m), b = F(m+1)

**assert** a.equals(*slowFibonacci*(m));

**assert** b.equals(*slowFibonacci*(m+1));

// Double it

BigInteger d = *multiply*(a, b.shiftLeft(1).subtract(a));

BigInteger e = *multiply*(a, a).add(*multiply*(b, b));

a = d;

b = e;

m \*= 2;

**assert** a.equals(*slowFibonacci*(m));

**assert** b.equals(*slowFibonacci*(m+1));

// Advance by one conditionally

**if** (((n >>> i) & 1) != 0) {

BigInteger c = a.add(b);

a = b;

b = c;

m++;

**assert** a.equals(*slowFibonacci*(m));

**assert** b.equals(*slowFibonacci*(m+1));

}

}

**return** a;

}

**private** **static** BigInteger fastFibonacciMatrix(**int** n) {

BigInteger[] matrix = {BigInteger.ONE, BigInteger.ONE, BigInteger.ONE, BigInteger.ZERO};

**return** *pow*(matrix, n)[1];

}

// Computes the power of a matrix

**private** **static** BigInteger[] pow(BigInteger[] matrix, **int** n) {

**if** (n < 0)

**throw** **new** IllegalArgumentException();

BigInteger[] result = {BigInteger.ONE, BigInteger.ZERO, BigInteger.ZERO, BigInteger.ONE};

**while** (n != 0) {

**if** (n % 2 != 0)

result = *multiply*(result, matrix);

n /= 2;

matrix = *multiply*(matrix, matrix);

}

**return** result;

}

**private** **static** BigInteger[] multiply(BigInteger[] x, BigInteger[] y) {

**return** **new** BigInteger[] {

*multiply*(x[0], y[0]).add(*multiply*(x[1], y[2])),

*multiply*(x[0], y[1]).add(*multiply*(x[1], y[3])),

*multiply*(x[2], y[0]).add(*multiply*(x[3], y[2])),

*multiply*(x[2], y[1]).add(*multiply*(x[3], y[3]))

};

}

/\*

\* Simple slow method, using dynamic programming

\* F(n+2) = F(n+1) + F(n)

\*/

**private** **static** BigInteger slowFibonacci(**int** n) {

BigInteger a = BigInteger.ZERO;

BigInteger b = BigInteger.ONE;

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

BigInteger c = a.add(b);

a = b;

b = c;

}

**return** a;

}

// Multiplies two BigIntegers

**private** **static** BigInteger multiply(BigInteger x, BigInteger y) {

**return** x.multiply(y); // Replace this line with Karatsuba multiplication, etc. if available

}

### **Factorization in O(sqrt(N))**

**public static**Map<Long, Integer> factorize(**long**n) {  
    Map<Long, Integer> factors = **new**LinkedHashMap<>();  
    **for**(**long**d = 2; n > 1; ) {  
      **int**power = 0;  
      **while**(n % d == 0) {  
        ++power;  
        n /= d;  
      }  
      **if**(power > 0) {  
        factors.put(d, power);  
      }  
      ++d;  
      **if**(d \* d > n) {  
        d = n;  
      }  
    }  
    **return**factors;  
  }  
  
  **public static int**[] getAllDivisors(**int**n) {  
    List<Integer> divisors = **new**ArrayList<>();  
    **for**(**int**d = 1; d \* d <= n; d++)  
      **if**(n % d == 0) {  
        divisors.add(d);  
        **if**(d \* d != n)  
          divisors.add(n / d);  
      }  
    **int**[] res = **new int**[divisors.size()];  
    **for**(**int**i = 0; i < res.length; i++)  
      res[i] = divisors.get(i);  
    Arrays.sort(res);  
    **return**res;  
  }

**Geometry: Line**

**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()]);  
  }  
}

### **Geometry: Circle**

**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  
  **static boolean**eq(Point p1, Point p2) {  
    **return**!(fastHypot(p1.x - p2.x, p1.y - p2.y) > 1e-9);  
  }  
}

# Arrays

## **LCS**

**public class**Lcs {  
  
  **public static int**[] getLCS(**int**[] x, **int**[] y) {  
    **int**m = x.length;  
    **int**n = y.length;  
    **int**[][] lcs = **new int**[m + 1][n + 1];  
    **for**(**int**i = 0; i < m; i++) {  
      **for**(**int**j = 0; j < n; j++) {  
        **if**(x[i] == y[j]) {  
          lcs[i + 1][j + 1] = lcs[i][j] + 1;  
        } **else**{  
          lcs[i + 1][j + 1] = Math.max(lcs[i + 1][j], lcs[i][j + 1]);  
        }  
      }  
    }  
    **int**cnt = lcs[m][n];  
    **int**[] res = **new int**[cnt];  
    **for**(**int**i = m - 1, j = n - 1; i >= 0 && j >= 0; ) {  
      **if**(x[i] == y[j]) {  
        res[--cnt] = x[i];  
        --i;  
        --j;  
      } **else if**(lcs[i + 1][j] > lcs[i][j + 1]) {  
        --j;  
      } **else**{  
        --i;  
      }  
    }  
    **return**res;  
  }  
  
  // Usage example  
  **// ex**  
    **int**[] x = {1, 5, 4, 2, 3, 7, 6};  
    **int**[] y = {2, 7, 1, 3, 5, 4, 6};  
    **int**[] lcs = getLCS(x, y);

## **Longest increasing subsequence in O(N \* logN)**

**public class**Lis2 {  
  
  **public static int**[] lis(**int**[] a) {  
    **int**n = a.length;  
    **int**[] tail = **new int**[n];  
    **int**[] prev = **new int**[n];  
  
    **int**len = 0;  
    **for**(**int**i = 0; i < n; i++) {  
      **int**pos = lower\_bound(a, tail, len, a[i]);  
      **if**(pos == len) {  
        ++len;  
      }  
      prev[i] = pos > 0 ? tail[pos - 1] : -1;  
      tail[pos] = i;  
    }  
  
    **int**[] res = **new int**[len];  
    **for**(**int**i = tail[len - 1]; i >= 0; i = prev[i]) {  
      res[--len] = a[i];  
    }  
    **return**res;  
  }  
  
  **static int**lower\_bound(**int**[] a, **int**[] tail, **int**len, **int**key) {  
    **int**lo = -1;  
    **int**hi = len;  
    **while**(hi - lo > 1) {  
      **int**mid = (lo + hi) >>> 1;  
      **if**(a[tail[mid]] < key) {  
        lo = mid;  
      } **else**{  
        hi = mid;  
      }  
    }  
    **return**hi;  
  }

## **Dijstra**

**public class**DijkstraHeap {  
  
  **public static void**shortestPaths(List<Edge>[] edges, **int**s, **int**[] prio, **int**[] pred) {  
    Arrays.fill(pred, -1);  
    Arrays.fill(prio, Integer.MAX\_VALUE);  
    prio[s] = 0;  
    PriorityQueue<Long> q = **new**PriorityQueue<>();  
    q.add((**long**) s);  
    **while**(!q.isEmpty()) {  
      **long**cur = q.remove();  
      **int**curu = (**int**) cur;  
      **if**(cur >>> 32 != prio[curu])  
        **continue**;  
      **for**(Edge e : edges[curu]) {  
        **int**v = e.t;  
        **int**nprio = prio[curu] + e.cost;  
        **if**(prio[v] > nprio) {  
          prio[v] = nprio;  
          pred[v] = curu;  
          q.add(((**long**) nprio << 32) + v);  
        }  
      }  
    }  
  }  
  
  **static class**Edge {  
    **int**t, cost;  
  
    **public**Edge(**int**t, **int**cost) {  
      **this**.t = t;  
      **this**.cost = cost;  
    }  
  }  
  
  // Usage example  
  **public static void**main(String[] args) {  
    **int**[][] cost = { { 0, 3, 2 }, { 0, 0, -2 }, { 0, 0, 0 } };  
    **int**n = cost.length;  
    List<Edge>[] edges = **new**List[n];  
    **for**(**int**i = 0; i < n; i++) {  
      edges[i] = **new**ArrayList<>();  
      **for**(**int**j = 0; j < n; j++) {  
        **if**(cost[i][j] != 0) {  
          edges[i].add(**new**Edge(j, cost[i][j]));  
        }  
      }  
    }  
    **int**[] dist = **new int**[n];  
    **int**[] pred = **new int**[n];  
    shortestPaths(edges, 0, dist, pred);

## **Tarjan’s**

**public class**SCCTarjan {  
  
  List<Integer>[] graph;  
  **boolean**[] visited;  
  Stack<Integer> stack;  
  **int**time;  
  **int**[] lowlink;  
  List<List<Integer>> components;  
  
  **public**List<List<Integer>> scc(List<Integer>[] graph) {  
    **int**n = graph.length;  
    **this**.graph = graph;  
    visited = **new boolean**[n];  
    stack = **new**Stack<>();  
    time = 0;  
    lowlink = **new int**[n];  
    components = **new**ArrayList<>();  
  
    **for**(**int**u = 0; u < n; u++)  
      **if**(!visited[u])  
        dfs(u);  
  
    **return**components;  
  }  
  
  **void**dfs(**int**u) {  
    lowlink[u] = time++;  
    visited[u] = **true**;  
    stack.add(u);  
    **boolean**isComponentRoot = **true**;  
  
    **for**(**int**v : graph[u]) {  
      **if**(!visited[v])  
        dfs(v);  
      **if**(lowlink[u] > lowlink[v]) {  
        lowlink[u] = lowlink[v];  
        isComponentRoot = **false**;  
      }  
    }  
  
    **if**(isComponentRoot) {  
      List<Integer> component = **new**ArrayList<>();  
      **while**(**true**) {  
        **int**x = stack.pop();  
        component.add(x);  
        lowlink[x] = Integer.MAX\_VALUE;  
        **if**(x == u)  
          **break**;  
      }  
      components.add(component);  
    }  
  }

## **Biconnected components, bridges and cut points**

**public class**BiconnectedComponents {  
  
  List<Integer>[] graph;  
  **boolean**[] visited;  
  Stack<Integer> stack;  
  **int**time;  
  **int**[] tin;  
  **int**[] lowlink;  
  List<List<Integer>> components;  
  List<Integer> cutPoints;  
  List<String> bridges;  
  
  **public**List<List<Integer>> biconnectedComponents(List<Integer>[] graph) {  
    **int**n = graph.length;  
    **this**.graph = graph;  
    visited = **new boolean**[n];  
    stack = **new**Stack<>();  
    time = 0;  
    tin = **new int**[n];  
    lowlink = **new int**[n];  
    components = **new**ArrayList<>();  
    cutPoints = **new**ArrayList<>();  
    bridges = **new**ArrayList<>();  
  
    **for**(**int**u = 0; u < n; u++)  
      **if**(!visited[u])  
        dfs(u, -1);  
  
    **return**components;  
  }  
  
  **void**dfs(**int**u, **int**p) {  
    visited[u] = **true**;  
    lowlink[u] = tin[u] = time++;  
    stack.add(u);  
    **int**children = 0;  
    **boolean**cutPoint = **false**;  
    **for**(**int**v : graph[u]) {  
      **if**(v == p)  
        **continue**;  
      **if**(visited[v]) {  
        lowlink[u] = Math.min(lowlink[u], tin[v]);  
      } **else**{  
        dfs(v, u);  
        lowlink[u] = Math.min(lowlink[u], lowlink[v]);  
        cutPoint |= lowlink[v] >= tin[u];  
        **if**(lowlink[v] > tin[u])  
          bridges.add("(" + u + "," + v + ")");  
        ++children;  
      }  
    }  
    **if**(p == -1)  
      cutPoint = children >= 2;  
    **if**(cutPoint)  
      cutPoints.add(u);  
    **if**(lowlink[u] == tin[u]) {  
      List<Integer> component = **new**ArrayList<>();  
      **while**(**true**) {  
        **int**x = stack.pop();  
        component.add(x);  
        **if**(x == u)  
          **break**;  
      }  
      components.add(component);  
    }  
  }  
  // Usage example  
    BiconnectedComponents bc = **new**BiconnectedComponents();  
    List<List<Integer>> components = bc.biconnectedComponents(graph);  
  
    System.out.println("biconnected components:" + components);  
    System.out.println("cutPoints: " + bc.cutPoints);  
    System.out.println("bridges:" + bc.bridges);

## **Greedy graph coloring in O(E \* logV)**

**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. Floyd–Warshall algorithm in O(V^3)**

**public class**FloydWarshall {  
  
  **static final int**INF = Integer.MAX\_VALUE / 2;  
  
  // precondition: d[i][i] == 0  
  **public static int**[][] floydWarshall(**int**[][] d) {  
    **int**n = d.length;  
    **int**[][] pred = **new int**[n][n];  
    **for**(**int**i = 0; i < n; i++)  
      **for**(**int**j = 0; j < n; j++)  
        pred[i][j] = (i != j && d[i][j] != INF) ? i : -1;  
    **for**(**int**k = 0; k < n; k++) {  
      **for**(**int**i = 0; i < n; i++) {  
        **if**(d[i][k] == INF)  
          **continue**;  
        **for**(**int**j = 0; j < n; j++) {  
          **if**(d[k][j] == INF)  
            **continue**;  
          **if**(d[i][j] > d[i][k] + d[k][j]) {  
            d[i][j] = d[i][k] + d[k][j];  
            d[i][j] = Math.max(d[i][j], -INF);  
            pred[i][j] = pred[k][j];  
          }  
        }  
      }  
    }  
    **for**(**int**i = 0; i < n; i++)  
      **if**(d[i][i] < 0)  
        **return null**;  
    **return**pred;  
  }  
  
  // Usage example  
    **int**[][] dist = {{0, 3, 2}, {0, 0, 1}, {INF, 0, 0}};  
    **int**[][] pred = floydWarshall(dist);

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

**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));  
  }  
}

## **Maximum matching for bipartite graph. Kuhn's algorithm in O(E\*V)**

**public class**MaxMatching2 {  
  
  **public static int**maxMatching(List<Integer>[] graph, **int**n2) {  
    **int**n1 = graph.length;  
    **int**[] matching = **new int**[n2];  
    Arrays.fill(matching, -1);  
    **int**matches = 0;  
    **for**(**int**u = 0; u < n1; u++) {  
      **if**(findPath(graph, u, matching, **new boolean**[n1]))  
        ++matches;  
    }  
    **return**matches;  
  }  
  
  **static boolean**findPath(List<Integer>[] graph, **int**u1, **int**[] matching, **boolean**[] vis) {  
    vis[u1] = **true**;  
    **for**(**int**v : graph[u1]) {  
      **int**u2 = matching[v];  
      **if**(u2 == -1 || !vis[u2] && findPath(graph, u2, matching, vis)) {  
        matching[v] = u1;  
        **return true**;  
      }  
    }  
    **return false**;  
  }

## **Prime numbers, sieve of Eratosthenes, Euler's totient function**

**public class**PrimesAndDivisors {  
  
  **public static int**[] generatePrimes(**int**n) {  
    **boolean**[] prime = **new boolean**[n + 1];  
    Arrays.fill(prime, 2, n + 1, **true**);  
    **for**(**int**i = 2; i \* i <= n; i++)  
      **if**(prime[i])  
        **for**(**int**j = i \* i; j <= n; j += i)  
          prime[j] = **false**;  
    **int**[] primes = **new int**[n + 1];  
    **int**cnt = 0;  
    **for**(**int**i = 0; i < prime.length; i++)  
      **if**(prime[i])  
        primes[cnt++] = i;  
  
    **return**Arrays.copyOf(primes, cnt);  
  }  
  
  **public static int**[] generatePrimesLinear(**int**n) {  
    **int**[] lp = **new int**[n + 1];  
    **int**[] primes = **new int**[n + 1];  
    **int**cnt = 0;  
    **for**(**int**i = 2; i <= n; ++i) {  
      **if**(lp[i] == 0) {  
        lp[i] = i;  
        primes[cnt++] = i;  
      }  
      **for**(**int**j = 0; j < cnt && primes[j] <= lp[i] && i \* primes[j] <= n; ++j)  
        lp[i \* primes[j]] = primes[j];  
    }  
    **return**Arrays.copyOf(primes, cnt);  
  }  
  
  **public static boolean**isPrime(**long**n) {  
    **if**(n <= 1)  
      **return false**;  
    **for**(**long**i = 2; i \* i <= n; i++)  
      **if**(n % i == 0)  
        **return false**;  
    **return true**;  
  }  
  
  **public static int**[] numberOfPrimeDivisors(**int**n) {  
    **int**[] divisors = **new int**[n + 1];  
    Arrays.fill(divisors, 2, n + 1, 1);  
    **for**(**int**i = 2; i \* i <= n; ++i)  
      **if**(divisors[i] == 1)  
        **for**(**int**j = i; j \* i <= n; j++)  
          divisors[j \* i] = divisors[j] + 1;  
    **return**divisors;  
  }  
  
  **public static int**[] generateDivisorTable(**int**n) {  
    **int**[] divisor = **new int**[n + 1];  
    **for**(**int**i = 1; i <= n; i++)  
      divisor[i] = i;  
    **for**(**int**i = 2; i \* i <= n; i++)  
      **if**(divisor[i] == i)  
        **for**(**int**j = i \* i; j <= n; j += i)  
          divisor[j] = i;  
    **return**divisor;  
  }  
  
  **public static int**phi(**int**n) {  
    **int**res = n;  
    **for**(**int**i = 2; i \* i <= n; i++)  
      **if**(n % i == 0) {  
        **while**(n % i == 0)  
          n /= i;  
        res -= res / i;  
      }  
    **if**(n > 1)  
      res -= res / n;  
    **return**res;  
  }  
  
  **public static int**[] generatePhi(**int**n) {  
    **int**[] res = **new int**[n + 1];  
    **for**(**int**i = 1; i <= n; i++)  
      res[i] = i;  
    **for**(**int**i = 1; i <= n; i++)  
      **for**(**int**j = i + i; j <= n; j += i)  
        res[j] -= res[i];  
    **return**res;  
  }

## **Enumeration of permutations**

**public class**Permutations {  
  
  **public static boolean**nextPermutation(**int**[] p) {  
    **for**(**int**a = p.length - 2; a >= 0; --a)  
      **if**(p[a] < p[a + 1])  
        **for**(**int**b = p.length - 1; ; --b)  
          **if**(p[b] > p[a]) {  
            **int**t = p[a];  
            p[a] = p[b];  
            p[b] = t;  
            **for**(++a, b = p.length - 1; a < b; ++a, --b) {  
              t = p[a];  
              p[a] = p[b];  
              p[b] = t;  
            }  
            **return true**;  
          }  
    **return false**;  
  }  
  
  **public static int**[] permutationByNumber(**int**n, **long**number) {  
    **long**[] fact = **new long**[n];  
    fact[0] = 1;  
    **for**(**int**i = 1; i < n; i++) {  
      fact[i] = i \* fact[i - 1];  
    }  
    **int**[] p = **new int**[n];  
    **int**[] free = **new int**[n];  
    **for**(**int**i = 0; i < n; i++) {  
      free[i] = i;  
    }  
    **for**(**int**i = 0; i < n; i++) {  
      **int**pos = (**int**) (number / fact[n - 1 - i]);  
      p[i] = free[pos];  
      System.arraycopy(free, pos + 1, free, pos, n - 1 - pos);  
      number %= fact[n - 1 - i];  
    }  
    **return**p;  
  }  
  
  **public static long**numberByPermutation(**int**[] p) {  
    **int**n = p.length;  
    **long**[] fact = **new long**[n];  
    fact[0] = 1;  
    **for**(**int**i = 1; i < n; i++) {  
      fact[i] = i \* fact[i - 1];  
    }  
    **long**res = 0;  
    **for**(**int**i = 0; i < n; i++) {  
      **int**a = p[i];  
      **for**(**int**j = 0; j < i; j++) {  
        **if**(p[j] < p[i]) {  
          --a;  
        }  
      }  
      res += a \* fact[n - 1 - i];  
    }  
    **return**res;  
  }  
  
  **public static void**generatePermutations(**int**[] p, **int**depth) {  
    **int**n = p.length;  
    **if**(depth == n) {  
      System.out.println(Arrays.toString(p));  
      **return**;  
    }  
    **for**(**int**i = 0; i < n; i++) {  
      **if**(p[i] == 0) {  
        p[i] = depth;  
        generatePermutations(p, depth + 1);  
        p[i] = 0;  
      }  
    }  
  }  
  
  **public static long**nextPermutation(**long**x) {  
    **long**s = x & -x;  
    **long**r = x + s;  
    **long**ones = x ^ r;  
    ones = (ones >> 2) / s;  
    **return**r | ones;  
  }  
  
  **public static**List<List<Integer>> decomposeIntoCycles(**int**[] p) {  
    **int**n = p.length;  
    **boolean**[] vis = **new boolean**[n];  
    List<List<Integer>> res = **new**ArrayList<>();  
    **for**(**int**i = 0; i < n; i++) {  
      **if**(vis[i])  
        **continue**;  
      **int**j = i;  
      List<Integer> cur = **new**ArrayList<>();  
      **do**{  
        cur.add(j);  
        vis[j] = **true**;  
        j = p[j];  
      } **while**(j != i);  
      res.add(cur);  
    }  
    **return**res;  
  }