Dante Bencivenga’s Java Code Book

All programs after Node come from <https://github.com/indy256/codelibrary/tree/master/java/src> with some select edits and comments by Dante

Table of Contents

[Generic Template 2](#_Toc508787697)

[Graph Methods 3](#_Toc508787698)

[Node (Specialized BST) 4](#_Toc508787699)

[Aho-Corasick Algorithm (String Searching) 5](#_Toc508787700)

[All Nearest Smaller Values 6](#_Toc508787701)

[Array Rotation 7](#_Toc508787702)

[Bellman-Ford (Shortest Path with Negative Weights) 7](#_Toc508787703)

[Biconnectivity 8](#_Toc508787704)

[Binary Search 10](#_Toc508787705)

[Binomial Coefficients 10](#_Toc508787706)

[Bron-Kerbosch Algorithm (Maximal Clique Finding) 11](#_Toc508787707)

[Circle Operations 12](#_Toc508787708)

[Closest Pair of Points 16](#_Toc508787709)

[Complex Arithmetic 17](#_Toc508787710)

[Convex Hull (Rubber Band Analogy) 18](#_Toc508787711)

[Cover Tree (Nearest Neighbour Search Optimization) 20](#_Toc508787712)

[Determinant 21](#_Toc508787713)

[Euclid’s Algorithm (GCD) 22](#_Toc508787714)

[Euler Circuit (Path which visits each edge once) 24](#_Toc508787715)

[Prime Factorization 25](#_Toc508787716)

[Fenwick Tree (Fast Prefix Sums) 26](#_Toc508787717)

[Fast Fourier Transform 27](#_Toc508787718)

[Floyd-Warshall Algorithm (All Shortest Paths) 28](#_Toc508787719)

[Gaussian Elimination 29](#_Toc508787720)

[Graph Colouring 30](#_Toc508787721)

[Hill Climbing (Optimize Function of Two Arguments) 31](#_Toc508787722)

[Inversion Counting 31](#_Toc508787723)

[Karatsuba Algorithm (Multiplying Large Polynomials) 32](#_Toc508787724)

[KMP String Searching 33](#_Toc508787725)

[Lowest Common Ancestor 34](#_Toc508787726)

[Linear Equality 34](#_Toc508787727)

[Line Geometry 35](#_Toc508787728)

[Longest Increasing Subsequence 37](#_Toc508787729)

[Matrix Methods 37](#_Toc508787730)

[Max Flow 38](#_Toc508787731)

[Max Palindrome 39](#_Toc508787732)

[Min Cost Flow 40](#_Toc508787733)

[Maximal Independent Set 41](#_Toc508787734)

[Point to Line Segment Distance 42](#_Toc508787735)

[Prim’s Algorithm (Minimum Spanning Tree for Weighted Graph) 43](#_Toc508787736)

[Primes and Divisors 43](#_Toc508787737)

[Rational Numbers 45](#_Toc508787738)

[Shortest Hamiltonian Cycle (Visits each Vertex Once) 46](#_Toc508787739)

[Simpson Function Integration 46](#_Toc508787740)

[Suffix Array 47](#_Toc508787741)

[Longest Common Subsequence 48](#_Toc508787742)

[Ternary Search 49](#_Toc508787743)

# Generic Template

**import** java.util.\*;

**import** java.io.\*;

**public** **class** Generic {

**static** InputReader *in* = **new** InputReader(System.***in***);

**static** PrintWriter *out* = **new** PrintWriter(System.***out***);

**public** **static** **void** main(String[] args) {

*finish*();

}

**public** **static** **void** finish() {

*out*.close();

*in*.close();

System.*exit*(0);

}

**static** **class** InputReader **implements** Iterator<String>, Closeable {

// Fast input reader. Based on Kattio.java from open.kattis.com

// but has method names to match Scanner

**private** BufferedReader r;

**private** String line;

**private** StringTokenizer st;

**private** String token;

**public** InputReader(InputStream i) {

r = **new** BufferedReader(**new** InputStreamReader(i));

}

**public** **boolean** hasNext() {

**return** peekToken() != **null**;

}

**public** **int** nextInt() {

**return** Integer.*parseInt*(next());

}

**public** **double** nextDouble() {

**return** Double.*parseDouble*(next());

}

**public** **long** nextLong() {

**return** Long.*parseLong*(next());

}

**public** String next() {

String ans = peekToken();

token = **null**;

**return** ans;

}

**public** String nextLine() {

peekToken();

String ans = line;

token = **null**;

st = **null**;

**return** ans;

}

**public** **void** close() {

**try** {

r.close();

} **catch** (IOException e) {

}

}

**private** String peekToken() {

**if** (token == **null**)

**try** {

**while** (st == **null** || !st.hasMoreTokens()) {

line = r.readLine();

**if** (line == **null**)

**return** **null**;

st = **new** StringTokenizer(line);

}

token = st.nextToken();

} **catch** (IOException e) {

}

**return** token;

}

}

}

# Graph Methods

import java.util.\*;

import java.io.\*;

public class GraphMethods {

  static Scanner in = new Scanner(System.in);

  static PrintWriter out = new PrintWriter(System.out);

  public static void createGraph() {

    // commented lines are for weighted graphs

    int v = in.nextInt();

    int e = in.nextInt();

    @SuppressWarnings("unchecked")

    ArrayList<Integer>[] edges = new ArrayList[v];

    for (int i = 0; i < v; i++)

      edges[i] = new ArrayList<Integer>();

    for (int i = 0; i < e; i++) {

      int x = in.nextInt()-1;

      int y = in.nextInt()-1;

      edges[x].add(y);

      edges[y].add(x);

    }

  }

  public static void dfs(int vertex, ArrayList<Integer>[] edges, boolean[] checked) {

    checked[vertex] = true;

    int children = edges[vertex].size();

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

      int next = edges[vertex].get(i);

      if (checked[next]) {

        continue;

      } else {

        dfs(next, edges, checked);

      }

    }

  }

  public static void createWeightedGraph() {

    int n\_nodes = in.nextInt();

    int n\_edges = in.nextInt();

    @SuppressWarnings("unchecked")

    HashMap<Integer, Integer>[] edges = new HashMap[n\_nodes];

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

      edges[i] = new HashMap<Integer, Integer>();

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

      int x = in.nextInt() - 1;

      int y = in.nextInt() - 1;

      int w = in.nextInt();

      // following if duplicate edges are expected

      if (edges[x].containsKey(y) && edges[x].get(y) < w)

        continue;

      edges[x].put(y, w);

      edges[y].put(x, w);

    }

  }

  public static void dijkstra(int src, HashMap<Integer, Integer>[] edges, int[] dist, int[] prev) {

    // edges, dist, and prev must all be of size V

    TreeSet<Integer> vertices = new TreeSet<Integer> (new Comparator<Integer>() {

      public int compare(Integer i1, Integer i2) {

        if (dist[i1] < dist[i2]) return -1;

        if (dist[i1] > dist[i2]) return 1;

        return i1.compareTo(i2);

      }

    });

    dist[src] = 0;

    vertices.add(src);

    boolean[] checked = new boolean[dist.length];

    for (int v = 0; v < dist.length; v++) {

      if (v != src) {

        dist[v] = Integer.MAX\_VALUE;

      }

      prev[v] = -1;

      checked[v] = false;

    }

    while (!vertices.isEmpty()) {

      int u = vertices.pollFirst();

      for (Map.Entry<Integer, Integer> neighbour : edges[u].entrySet()) {

        int v = neighbour.getKey();

        if (checked[v])

          continue;

        int alt = dist[u] + neighbour.getValue();

        if (alt < dist[v]) {

          vertices.remove(v);

          dist[v] = alt;

          prev[v] = u;

          vertices.add(v);

        }

      }

      checked[u] = true;

    }

  }

}

# Node (Specialized BST)

**class** Node {

// replace specific conditions depending on the question

**public** Node left;

**public** Node right;

**public** **int** size = 0;

**public** **int** depth;

**public** Node() {

depth = 1 << 29;

}

**public** Node(**int** num, **int** depth) {

**this**.depth = depth;

add(num);

}

**public** **void** add(**int** num) {

size++;

**if** (depth == 0)

**return**;

**if** ((num & depth) == 0) {

**if** (left == **null**) {

left = **new** Node(num, depth >> 1);

} **else** {

left.add(num);

}

} **else** {

**if** (right == **null**) {

right = **new** Node(num, depth >> 1);

} **else** {

right.add(num);

}

}

}

**public** Node getLeft() {

Node res = left;

size--;

**if** (left.size == 1)

left = **null**;

**return** res;

}

**public** Node getRight() {

Node res = right;

size--;

**if** (right.size == 1)

right = **null**;

**return** res;

}

**public** **void** remove(**int** num) {

size--;

**if** (depth == 0)

**return**;

**if** ((num ^ depth) == 0) {

left.remove(num);

} **else** {

right.remove(num);

}

}

}

# Aho-Corasick Algorithm (String Searching)

import java.util.\*;

// https://en.wikipedia.org/wiki/Aho–Corasick\_algorithm

public class AhoCorasick {

    static final int ALPHABET\_SIZE = 26;

    Node[] nodes;

    int nodeCount;

    public static class Node {

        int parent;

        char charFromParent;

        int suffLink = -1;

        int[] children = new int[ALPHABET\_SIZE];

        int[] transitions = new int[ALPHABET\_SIZE];

        boolean leaf;

        {

            Arrays.fill(children, -1);

            Arrays.fill(transitions, -1);

        }

    }

    public AhoCorasick(int maxNodes) {

        nodes = new Node[maxNodes];

        // create root

        nodes[0] = new Node();

        nodes[0].suffLink = 0;

        nodes[0].parent = -1;

        nodeCount = 1;

    }

    public void addString(String s) {

        int cur = 0;

        for (char ch : s.toCharArray()) {

            int c = ch - 'a';

            if (nodes[cur].children[c] == -1) {

                nodes[nodeCount] = new Node();

                nodes[nodeCount].parent = cur;

                nodes[nodeCount].charFromParent = ch;

                nodes[cur].children[c] = nodeCount++;

            }

            cur = nodes[cur].children[c];

        }

        nodes[cur].leaf = true;

    }

    public int suffLink(int nodeIndex) {

        Node node = nodes[nodeIndex];

        if (node.suffLink == -1)

            node.suffLink = node.parent == 0 ? 0 : transition(suffLink(node.parent), node.charFromParent);

        return node.suffLink;

    }

    public int transition(int nodeIndex, char ch) {

        int c = ch - 'a';

        Node node = nodes[nodeIndex];

        if (node.transitions[c] == -1)

            node.transitions[c] = node.children[c] != -1 ? node.children[c] : (nodeIndex == 0 ? 0 : transition(suffLink(nodeIndex), ch));

        return node.transitions[c];

    }

    // Usage example

    public static void main(String[] args) {

        AhoCorasick ahoCorasick = new AhoCorasick(1000);

        ahoCorasick.addString("bc");

        ahoCorasick.addString("abc");

        String s = "tabcbc";

        int node = 0;

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

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

            node = ahoCorasick.transition(node, s.charAt(i));

            if (ahoCorasick.nodes[node].leaf)

                positions.add(i);

        }

        System.out.println(positions);

    }

}

# All Nearest Smaller Values

import java.util.\*;

public class AllNearestSmallerValues {

    // https://en.wikipedia.org/wiki/All\_nearest\_smaller\_values

    public static int[] nsv(int[] a) {

        int n = a.length;

        int[] p = new int[n];

        for (int i = 0; i < n; i++) {

            int j = i - 1;

            while (j != -1 && a[j] >= a[i]) {

                j = p[j];

            }

            p[i] = j;

        }

        return p;

    }

    public static long maxInscribedRectangle(int[] heights) {

        int n = heights.length;

        int[] rheights = new int[n];

        for (int i = 0; i < n; i++) {

            rheights[i] = heights[n - 1 - i];

        }

        int[] lnsv = nsv(heights);

        int[] rnsv = nsv(rheights);

        long res = 0;

        for (int i = 0; i < n; i++) {

            int a = lnsv[i] + 1;

            int b = n - 1 - rnsv[n - 1 - i] - 1;

            long cur = (long) (b - a + 1) \* heights[i];

            res = Math.max(res, cur);

        }

        return res;

    }

    public static long maxInscribedRectangle2(int[] heights) {

        long res = 0;

        Stack<Integer> spos = new Stack<>();

        Stack<Integer> sh = new Stack<>();

        sh.push(0);

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

            int h = i < heights.length ? heights[i] : 0;

            int pos = i;

            while (sh.peek() > h) {

                pos = spos.pop();

                res = Math.max(res, (long) sh.pop() \* (i - pos));

            }

            spos.push(pos);

            sh.push(h);

        }

        return res;

    }

    // Usage example

    public static void main(String[] args) {

        System.out.println(maxInscribedRectangle(new int[]{1, 2, 3}));

        System.out.println(maxInscribedRectangle2(new int[]{1, 2, 3}));

        System.out.println(Arrays.toString(nsv(new int[]{1, 1, 3, 2})));

        Random rnd = new Random(1);

        for (int step = 0; step < 1000; step++) {

            int n = rnd.nextInt(10) + 1;

            int[] h = rnd.ints(n, 0, 10).toArray();

            long res1 = maxInscribedRectangle(h);

            long res2 = maxInscribedRectangle2(h);

            if (res1 != res2)

                throw new RuntimeException();

        }

    }

}

# Array Rotation

import java.util.\*;

public class ArrayRotate {

    public static void rotate1(int[] a, int first, int middle, int last) {

        int next = middle;

        while (first != next) {

            swap(a, first++, next++);

            if (next == last)

                next = middle;

            else if (first == middle)

                middle = next;

        }

    }

    public static void rotate2(int[] a, int first, int middle, int last) {

        reverse(a, first, middle);

        reverse(a, middle, last);

        reverse(a, first, last);

    }

    static void reverse(int[] a, int from, int to) {

        while (from < --to)

            swap(a, from++, to);

    }

    public static void rotate3(int[] a, int first, int middle, int last) {

        int n = last - first;

        int jump = middle - first;

        int gcd = gcd(jump, n);

        int cycle = n / gcd;

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

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

                int next = cur + jump;

                if (next >= n)

                    next -= n;

                swap(a, cur, next);

                cur = next;

            }

        }

    }

    static int gcd(int a, int b) {

        return a == 0 ? b : gcd(b % a, a);

    }

    static void swap(int[] a, int i, int j) {

        int t = a[j];

        a[j] = a[i];

        a[i] = t;

    }

    // random test

    public static void main(String[] args) {

        Random rnd = new Random(1);

        for (int step = 0; step < 1000; step++) {

            int n = rnd.nextInt(10) + 1;

            int middle = rnd.nextInt(n);

            int[] a = rnd.ints(n, 0, 10).toArray();

            int[] b1 = a.clone();

            rotate1(b1, 0, middle, n);

            int[] b2 = a.clone();

            rotate2(b2, 0, middle, n);

            int[] b3 = a.clone();

            rotate3(b3, 0, middle, n);

            if (!Arrays.equals(b1, b2) || !Arrays.equals(b1, b3))

                throw new RuntimeException();

        }

    }

}

# Bellman-Ford (Shortest Path with Negative Weights)

import java.util.\*;

import java.util.stream.Stream;

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) {

        @SuppressWarnings("unchecked")

        List<Edge>[] graph = Stream.generate(ArrayList::new).limit(4).toArray(List[]::new);

        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));

    }

}

# Biconnectivity

import java.util.\*;

import java.util.stream.Stream;

public class Biconnectivity {

    List<Integer>[] graph;

    boolean[] visited;

    Stack<Integer> stack;

    int time;

    int[] tin;

    int[] lowlink;

    List<List<Integer>> edgeBiconnectedComponents;

    List<Integer> cutPoints;

    List<String> bridges;

    public List<List<Integer>> biconnectivity(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];

        edgeBiconnectedComponents = new ArrayList<>();

        cutPoints = new ArrayList<>();

        bridges = new ArrayList<>();

        for (int u = 0; u < n; u++)

            if (!visited[u])

                dfs(u, -1);

        return edgeBiconnectedComponents;

    }

    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]); // or lowlink[u] = Math.min(lowlink[u], lowlink[v]);

            } else {

                dfs(v, u);

                lowlink[u] = Math.min(lowlink[u], lowlink[v]);

                cutPoint |=  tin[u] <= lowlink[v];

                if (tin[u] < lowlink[v]) // or if (lowlink[v] == tin[v])

                    bridges.add("(" + u + "," + v + ")");

                ++children;

            }

        }

        if (p == -1)

            cutPoint = children >= 2;

        if (cutPoint)

            cutPoints.add(u);

        if (tin[u] == lowlink[u]) {

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

            while (true) {

                int x = stack.pop();

                component.add(x);

                if (x == u)

                    break;

            }

            edgeBiconnectedComponents.add(component);

        }

    }

    // tree of edge-biconnected components

    public static List<Integer>[] ebcTree(List<Integer>[] graph, List<List<Integer>> components) {

        int[] comp = new int[graph.length];

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

            for (int u : components.get(i))

                comp[u] = i;

        @SuppressWarnings("unchecked")

        List<Integer>[] g = Stream.generate(ArrayList::new).limit(components.size()).toArray(List[]::new);

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

            for (int v : graph[u])

                if (comp[u] != comp[v])

                    g[comp[u]].add(comp[v]);

        return g;

    }

    // Usage example

    public static void main(String[] args) {

        @SuppressWarnings("unchecked")

        List<Integer>[] graph = Stream.generate(ArrayList::new).limit(6).toArray(List[]::new);

        int[][] esges = {{0, 1}, {1, 2}, {0, 2}, {2, 3}, {1, 4}, {4, 5}, {5, 1}};

        for (int[] edge : esges) {

            graph[edge[0]].add(edge[1]);

            graph[edge[1]].add(edge[0]);

        }

        Biconnectivity bc = new Biconnectivity();

        List<List<Integer>> components = bc.biconnectivity(graph);

        System.out.println("edge-biconnected components:" + components);

        System.out.println("cut points: " + bc.cutPoints);

        System.out.println("bridges:" + bc.bridges);

        System.out.println("condensation tree:" + Arrays.toString(ebcTree(graph, components)));

    }

}

# Binary Search

import java.util.\*;

import java.util.function.\*;

public class BinarySearch {

// 000[1]11

// warning: overflows in lines 1-4

public static int binarySearchFirstTrueSimple(IntPredicate predicate, int fromInclusive, int toInclusive) {

int lo = fromInclusive - 1;

int hi = toInclusive + 1;

while (hi - lo > 1) {

int mid = (lo + hi) / 2;

if (!predicate.test(mid)) {

lo = mid;

} else {

hi = mid;

}

}

return hi;

}

// 000[1]11

// correct binary search

public static int binarySearchFirstTrue(IntPredicate predicate, int fromInclusive, int toExclusive) {

int lo = fromInclusive;

int hi = toExclusive;

while (lo < hi) {

// int mid = lo + ((hi - lo) >>> 1);

int mid = (lo & hi) + ((lo ^ hi) >> 1);

if (!predicate.test(mid)) {

lo = mid + 1;

} else {

hi = mid;

}

}

return hi;

}

public static double binarySearch(DoublePredicate predicate, double lo, double hi) {

for (int step = 0; step < 1000; step++) {

double mid = (lo + hi) / 2;

if (!predicate.test(mid)) {

lo = mid;

} else {

hi = mid;

}

}

return hi;

}

// random test

public static void main(String[] args) {

Random rnd = new Random(1);

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

int n = rnd.nextInt(20);

boolean[] b = new boolean[n];

int firstTrue = rnd.nextInt(n + 1);

Arrays.fill(b, firstTrue, n, true);

int res1 = binarySearchFirstTrueSimple(i -> b[i], 0, n - 1);

int res2 = binarySearchFirstTrue(i -> b[i], 0, n);

if (res1 != firstTrue || res1 != res2)

throw new RuntimeException();

}

System.out.println(Math.sqrt(2) == binarySearch(x -> x \* x >= 2, 0, 2));

}

}

# Binomial Coefficients

import java.math.BigInteger;

public class BinomialCoefficients {

public static long[][] binomialTable(int n) {

long[][] c = new long[n + 1][n + 1];

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

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

c[i][j] = (j == 0) ? 1 : c[i - 1][j - 1] + c[i - 1][j];

return c;

}

public static long binomial(long n, long m) {

m = Math.min(m, n - m);

long res = 1;

for (long i = 0; i < m; i++) {

res = res \* (n - i) / (i + 1);

}

return res;

}

// for (int i = 1; i < f.length; i++) f[i] = f[i - 1] + Math.log(i);

public static double binomial(int n, int m, double[] f) {

if (m < 0 || m > n) return 0;

return Math.exp(f[n] - f[m] - f[n - m]);

}

// n! % mod

public static int factorial(int n, int mod) {

long res = 1;

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

res = res \* i % mod;

return (int) (res % mod);

}

// n! mod p, p - prime, O(p\*log(n)) complexity

public static int factorial2(int n, int p) {

int res = 1;

while (n > 1) {

res = (res \* ((n / p) % 2 == 1 ? p - 1 : 1)) % p;

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

res = (res \* i) % p;

n /= p;

}

return res % p;

}

public static int binomial(int n, int m, int mod) {

m = Math.min(m, n - m);

long res = 1;

for (int i = n - m + 1; i <= n; i++)

res = res \* i % mod;

return (int) (res \* BigInteger.valueOf(factorial(m, mod)).modInverse(BigInteger.valueOf(mod)).intValue() % mod);

}

// Usage example

public static void main(String[] args) {

}

}

# Bron-Kerbosch Algorithm (Maximal Clique Finding)

import java.util.\*;

// https://en.wikipedia.org/wiki/Bron–Kerbosch\_algorithm

public class BronKerbosh {

public static int BronKerbosch(long[] g, long cur, long allowed, long forbidden, int[] weights) {

if (allowed == 0 && forbidden == 0) {

int res = 0;

for (int u = Long.numberOfTrailingZeros(cur); u < g.length; u += Long.numberOfTrailingZeros(cur >> (u + 1)) + 1)

res += weights[u];

return res;

}

if (allowed == 0)

return -1;

int res = -1;

int pivot = Long.numberOfTrailingZeros(allowed | forbidden);

long z = allowed & ~g[pivot];

for (int u = Long.numberOfTrailingZeros(z); u < g.length; u += Long.numberOfTrailingZeros(z >> (u + 1)) + 1) {

res = Math.max(res, BronKerbosch(g, cur | (1L << u), allowed & g[u], forbidden & g[u], weights));

allowed ^= 1L << u;

forbidden |= 1L << u;

}

return res;

}

// random test

public static void main(String[] args) {

Random rnd = new Random(1);

for (int step = 0; step < 1000; step++) {

int n = rnd.nextInt(16) + 1;

long[] g = new long[n];

int[] weights = new int[n];

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

weights[i] = rnd.nextInt(1000);

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

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

if (rnd.nextBoolean()) {

g[i] |= 1L << j;

g[j] |= 1L << i;

}

int res1 = BronKerbosch(g, 0, (1L << n) - 1, 0, weights);

int res2 = maxCliqueSlow(g, weights);

if (res1 != res2)

throw new RuntimeException();

}

}

static int maxCliqueSlow(long[] g, int[] weights) {

int res = 0;

int n = g.length;

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

boolean ok = true;

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

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

ok &= (set & (1 << i)) == 0 || (set & (1 << j)) == 0 || (g[i] & (1L << j)) != 0;

if (ok) {

int cur = 0;

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

if ((set & (1 << i)) != 0)

cur += weights[i];

res = Math.max(res, cur);

}

}

return res;

}

}

# Circle Operations

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 {

double x, y;

public Point(double x, double y) {

this.x = x;

this.y = y;

}

}

public static class Circle {

double x, y, r;

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[] points) {

if (points.length == 0)

return new Circle(0, 0, 0);

if (points.length == 1)

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

Collections.shuffle(Arrays.asList(points));

Circle circle = getCircumCircle(points[0], points[1]);

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

if (!circle.contains(points[i])) {

circle = getCircumCircle(points[0], points[i]);

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

if (!circle.contains(points[j])) {

circle = getCircumCircle(points[j], points[i]);

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

if (!circle.contains(points[k])) {

circle = getCircumCircle(points[i], points[j], points[k]);

}

}

}

}

}

}

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);

}

}

# Closest Pair of Points

import java.util.\*;

public class Closest2Points {

public static class Point {

long x, y;

public Point(long x, long y) {

this.x = x;

this.y = y;

}

}

public static final Comparator<Point> CMP\_X = (a, b) -> Long.compare(a.x, b.x) != 0 ? Long.compare(a.x, b.x) : Long.compare(a.y, b.y);

public static final Comparator<Point> CMP\_Y = (a, b) -> Long.compare(a.y, b.y);

public static Point[] findClosestPair(Point[] points) {

Point[] result = new Point[2];

Arrays.sort(points, CMP\_X);

rec(points, 0, points.length - 1, result, Long.MAX\_VALUE);

return result;

}

static long rec(Point[] points, int l, int r, Point[] result, long mindist) {

if (l == r)

return Long.MAX\_VALUE;

int mid = (l + r) >> 1;

long midx = points[mid].x;

long d1 = rec(points, l, mid, result, mindist);

mindist = Math.min(mindist, d1);

long d2 = rec(points, mid + 1, r, result, mindist);

mindist = Math.min(mindist, d2);

Arrays.sort(points, l, r + 1, CMP\_Y);

int[] t = new int[r - l + 1];

int size = 0;

for (int i = l; i <= r; i++)

if (Math.abs(points[i].x - midx) < mindist)

t[size++] = i;

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

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

Point a = points[t[i]];

Point b = points[t[j]];

if (b.y - a.y >= mindist)

break;

long dist = dist(a, b);

if (mindist > dist) {

mindist = dist;

result[0] = a;

result[1] = b;

}

}

}

return mindist;

}

static long dist(Point a, Point b) {

long dx = a.x - b.x;

long dy = a.y - b.y;

return dx \* dx + dy \* dy;

}

// random test

public static void main(String[] args) {

Random rnd = new Random();

for (int step = 0; step < 10\_000; step++) {

int n = 100;

Point[] points = new Point[n];

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

points[i] = new Point(rnd.nextInt(1000) - 500, rnd.nextInt(1000) - 500);

}

Point[] p = findClosestPair(points);

long res1 = dist(p[0], p[1]);

long res2 = slowClosestPair(points);

if (res1 != res2)

throw new RuntimeException(res1 + " " + res2);

}

}

static long slowClosestPair(Point[] points) {

long res = Long.MAX\_VALUE;

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

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

res = Math.min(res, dist(points[i], points[j]));

return res;

}

}

# Complex Arithmetic

public class Complex {

final double x;

final double y;

public Complex(double x) {

this.x = x;

this.y = 0;

}

public Complex(double x, double y) {

this.x = x;

this.y = y;

}

public Complex conj() {

return new Complex(x, -y);

}

public Complex sub(Complex b) {

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

}

public Complex add(Complex b) {

return new Complex(x + b.x, y + b.y);

}

public Complex mul(Complex b) {

return new Complex(x \* b.x - y \* b.y, x \* b.y + y \* b.x);

}

public Complex div(Complex b) {

return this.mul(b.conj()).mul(1 / b.len2());

}

public Complex mul(double b) {

return new Complex(x \* b, y \* b);

}

public double len2() {

return x \* x + y \* y;

}

public double abs() {

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

}

public Complex norm() {

return abs() == 0 ? new Complex(0, 0) : mul(1 / abs());

}

public double cross(Complex b) {

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

}

double cross2(Complex b) {

return this.conj().mul(b).y;

}

public double dot(Complex b) {

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

}

double dot2(Complex b) {

return this.conj().mul(b).x;

}

public static Complex polar(double r, double theta) {

return new Complex(r \* Math.cos(theta), r \* Math.sin(theta));

}

public static Complex exp(Complex a) {

return polar(Math.exp(a.x), a.y);

}

public double arg() {

return Math.atan2(y, x);

}

public Complex rot90() {

return new Complex(-y, x);

}

public Complex rotate(Complex p, double angle) {

return p.sub(this).mul(exp(new Complex(0, angle))).add(this);

}

public Complex rotate2(Complex p, double angle) {

p = p.sub(this);

double cs = Math.cos(angle);

double sn = Math.sin(angle);

return new Complex(p.x \* cs - p.y \* sn, p.x \* sn + p.y \* cs).add(this);

}

public Complex reflect(Complex p, Complex q) {

Complex s = q.sub(p);

return this.sub(p).div(s).conj().mul(s).add(p);

}

public double proj(Complex p) {

return dot(p) / this.abs();

}

public static double angle(Complex a, Complex p, Complex b) {

a = a.sub(p);

b = b.sub(p);

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

}

@Override

public String toString() {

return "Complex [x=" + x + ", y=" + y + "]";

}

// Usage example

public static void main(String[] args) {

Complex z = new Complex(3, 2);

z = z.div(z);

System.out.println(z);

System.out.println();

Complex u = new Complex(0, 0);

Complex v = new Complex(1, 0);

Complex a = u.rotate(v, Math.PI \* 1.0);

Complex b = v.rot90().rot90();

System.out.println(a);

System.out.println(b);

}

}

# Convex Hull (Rubber Band Analogy)

import java.util.\*;

public class ConvexHull {

public static Point[] convexHull(Point[] points) {

Arrays.sort(points, (a, b) -> Integer.compare(a.x, b.x) != 0 ? Integer.compare(a.x, b.x) : Integer.compare(a.y, b.y));

int n = points.length;

Point[] hull = new Point[n + 1];

int cnt = 0;

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

int j = i < n ? i : 2 \* n - 1 - i;

while (cnt >= 2 && isNotRightTurn(hull[cnt - 2], hull[cnt - 1], points[j]))

--cnt;

hull[cnt++] = points[j];

}

return Arrays.copyOf(hull, cnt - 1);

}

static boolean isNotRightTurn(Point a, Point b, Point c) {

long cross = (long) (a.x - b.x) \* (c.y - b.y) - (long) (a.y - b.y) \* (c.x - b.x);

long dot = (long) (a.x - b.x) \* (c.x - b.x) + (long) (a.y - b.y) \* (c.y - b.y);

return cross < 0 || cross == 0 && dot <= 0;

}

public static class Point {

public final int x, y;

public Point(int x, int y) {

this.x = x;

this.y = y;

}

}

public static Point[] convexHull2(Point[] p) {

int n = p.length;

if (n <= 1)

return p;

Arrays.sort(p, (a, b) -> Integer.compare(a.x, b.x) != 0 ? Integer.compare(a.x, b.x) : Integer.compare(a.y, b.y));

Point[] h = new Point[n \* 2];

int cnt = 0;

for (int i = 0; i < n; h[cnt++] = p[i++])

while (cnt > 1 && cross(h[cnt - 2], h[cnt - 1], p[i]) >= 0)

--cnt;

for (int i = n - 2, t = cnt; i >= 0; h[cnt++] = p[i--])

while (cnt > t && cross(h[cnt - 2], h[cnt - 1], p[i]) >= 0)

--cnt;

return Arrays.copyOf(h, cnt - 1 - (h[0].x == h[1].x && h[0].y == h[1].y ? 1 : 0));

}

static long cross(Point a, Point b, Point c) {

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

}

// random test

public static void main(String[] args) {

Random rnd = new Random(1);

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

int n = rnd.nextInt(10) + 1;

Point[] points = new Point[n];

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

int range = 10;

points[i] = new Point(rnd.nextInt(range) - range / 2, rnd.nextInt(range) - range / 2);

}

Point[] convexHull = convexHull(points);

Point[] convexHull2 = convexHull2(points);

for (int i = 0; i < Math.max(convexHull.length, convexHull2.length); i++)

if (convexHull[i].x != convexHull2[i].x || convexHull[i].y != convexHull2[i].y)

throw new RuntimeException();

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

final Point[] hull;

if (i == 0) {

hull = convexHull;

} else {

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

Collections.addAll(list, convexHull);

list.remove(i - 1);

hull = list.toArray(new Point[list.size()]);

}

boolean exterior = false;

for (Point point : points)

exterior |= pointInPolygon(point.x, point.y, hull) == -1;

if (exterior != (i > 0))

throw new RuntimeException();

}

}

}

static int pointInPolygon(int qx, int qy, Point[] points) {

int n = points.length;

int cnt = 0;

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

if (points[i].y == qy && (points[i].x == qx || points[j].y == qy && (points[i].x <= qx || points[j].x <= qx) && (points[i].x >= qx || points[j].x >= qx)))

return 0; // boundary

if ((points[i].y > qy) != (points[j].y > qy)) {

long det = (long) (points[i].x - qx) \* (points[j].y - qy) - (long) (points[j].x - qx) \* (points[i].y - qy);

if (det == 0)

return 0; // boundary

if ((det > 0) != (points[j].y - points[i].y > 0))

++cnt;

}

}

return cnt % 2 == 0 ? -1 /\* exterior \*/ : 1 /\* interior \*/;

}

}

# Cover Tree (Nearest Neighbour Search Optimization)

import java.util.\*;

// Based on http://hunch.net/~jl/projects/cover\_tree/paper/paper.pdf

public class CoverTree {

static double dist(Node p1, Node p2) {

double dx = p1.x - p2.x;

double dy = p1.y - p2.y;

return Math.sqrt(dx \* dx + dy \* dy);

}

static final int levels = 64;

double[] layerRadius = new double[levels];

public CoverTree() {

layerRadius[0] = 1 << 30;

for (int i = 1; i < layerRadius.length; i++)

layerRadius[i] = layerRadius[i - 1] / 2;

}

Node root;

static class Node {

double x, y;

@SuppressWarnings("unchecked")

        List<Node>[] children = new List[levels];

int maxChildLevel = 0;

public void addChild(Node node, int level) {

if (children[level] == null)

children[level] = new ArrayList<>(1);

children[level].add(node);

maxChildLevel = Math.max(maxChildLevel, level);

}

public List<Node> getChildren(int level) {

return children[level] != null ? children[level] : Collections.<Node>emptyList();

}

public Node(double x, double y) {

this.x = x;

this.y = y;

}

}

public void insert(Node p) {

if (root == null)

root = p;

else

insert(p, Arrays.asList(root), 0);

}

boolean insert(Node p, List<Node> Qi, int level) {

Node parent = null;

List<Node> nQi = new ArrayList<>();

for (Node q : Qi) {

if (dist(p, q) <= layerRadius[level]) {

nQi.add(q);

parent = q;

}

}

if (parent == null) // separation holds

return true;

for (Node q : Qi)

for (Node ch : q.getChildren(level))

if (dist(p, ch) <= layerRadius[level])

nQi.add(ch);

if (insert(p, nQi, level + 1))

parent.addChild(p, level);

return false;

}

double bestDist;

Node bestNode;

public Node findNearest(Node p) {

bestDist = p != root ? dist(p, root) : Double.POSITIVE\_INFINITY;

bestNode = p != root ? root : null;

findNearest(p, Arrays.asList(root), 0);

return bestNode;

}

void findNearest(Node p, List<Node> Qi, int level) {

for (; !Qi.isEmpty(); level++) {

List<Node> Q = new ArrayList<>();

for (Node q : Qi) {

Q.add(q);

for (Node ch : q.getChildren(level)) {

if (ch != p) {

double dist = dist(p, ch);

if (bestDist > dist) {

bestDist = dist;

bestNode = ch;

}

}

Q.add(ch);

}

}

Qi = new ArrayList<>();

for (Node q : Q)

if (q.maxChildLevel > level && dist(p, q) <= bestDist + layerRadius[level])

Qi.add(q);

}

}

// Usage example

public static void main(String[] args) {

CoverTree tree = new CoverTree();

tree.insert(new Node(1, 1));

tree.insert(new Node(2, 2));

Node p = tree.findNearest(new Node(1.6, 1.6));

System.out.println(2 == p.x && 2 == p.y);

}

}

# Determinant

import java.util.stream.IntStream;

public class Determinant {

public static double det(double[][] matrix) {

final double EPS = 1e-10;

int n = matrix.length;

double[][] a = IntStream.range(0, n).mapToObj(i -> matrix[i].clone()).toArray(double[][]::new); // make a copy

double res = 1;

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

int p = i;

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

if (Math.abs(a[j][i]) > Math.abs(a[p][i]))

p = j;

if (Math.abs(a[p][i]) < EPS)

return 0;

if (i != p) {

res = -res;

double[] t = a[i];

a[i] = a[p];

a[p] = t;

}

res \*= a[i][i];

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

a[i][j] /= a[i][i];

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

if (j != i && Math.abs(a[j][i]) > EPS /\*optimizes complexity to O(n^2) for sparse matrices\*/)

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

a[j][k] -= a[i][k] \* a[j][i];

}

return res;

}

// Usage example

public static void main(String[] args) {

double d = det(new double[][]{{0, 1}, {-1, 0}});

System.out.println(Math.abs(d - 1) < 1e-10);

}

}

# Euclid’s Algorithm (GCD)

import java.math.BigInteger;

import java.util.\*;

public class Euclid {

public static long gcd(long a, long b) {

return b == 0 ? Math.abs(a) : gcd(b, a % b);

}

public static long gcd2(long a, long b) {

while (b != 0) {

long t = b;

b = a % b;

a = t;

}

return Math.abs(a);

}

public static long lcm(long a, long b) {

return Math.abs(a / gcd(a, b) \* b);

}

// 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;

// invariant: a=a\*x+b\*y, b=a\*x1+b\*y1

while (b != 0) {

long q = a / b;

long \_x1 = x1;

long \_y1 = y1;

long \_b = b;

x1 = x - q \* x1;

y1 = y - q \* y1;

b = a - q \* b;

x = \_x1;

y = \_y1;

a = \_b;

}

return a > 0 ? new long[]{a, x, y} : new long[]{-a, -x, -y};

}

public static long[] euclid2(long a, long b) {

if (b == 0)

return a > 0 ? new long[]{a, 1, 0} : new long[]{-a, -1, 0};

long[] r = euclid2(b, a % b);

return new long[]{r[0], r[2], r[1] - a / b \* r[2]};

}

public static int mod(long a, int m) {

int A = (int) (a % m);

return A >= 0 ? A : A + m;

}

// precondition: m > 0 && gcd(a, m) = 1

public static int modInverse(int a, int m) {

a = mod(a, m);

return a == 0 ? 0 : mod((1 - (long) modInverse(m % a, a) \* m) / a, m);

}

// precondition: m > 0 && gcd(a, m) = 1

public static int modInverse2(int a, int m) {

return mod(euclid(a, m)[1], m);

}

// precondition: p is prime

public static int[] generateInverse(int p) {

int[] res = new int[p];

res[1] = 1;

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

res[i] = (p - (p / i) \* res[p % i] % p) % p;

return res;

}

// returns x ≡ a[i] (mod p[i]), where gcd(p[i], p[j]) == 1

public static BigInteger garnerRestore(int[] a, int[] p) {

int[] x = a.clone();

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

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

x[i] = mod(BigInteger.valueOf(p[j]).modInverse(BigInteger.valueOf(p[i])).longValue() \* (x[i] - x[j]), p[i]);

BigInteger res = BigInteger.valueOf(x[0]);

BigInteger m = BigInteger.ONE;

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

m = m.multiply(BigInteger.valueOf(p[i - 1]));

res = res.add(m.multiply(BigInteger.valueOf(x[i])));

}

return res;

}

// returns x ≡ a[i] (mod p[i]), where gcd(p[i], p[j]) == 1

public static int simpleRestore(int[] a, int[] p) {

int res = 0;

for (int i = 0, m = 1; i < a.length; i++, m \*= p[i])

while (res % p[i] != a[i])

res += m;

return res;

}

// Usage example

public static void main(String[] args) {

Random rnd = new Random(1);

for (int steps = 0; steps < 10000; steps++) {

int a = rnd.nextInt(20) - 10;

int b = rnd.nextInt(20) - 10;

BigInteger xa = BigInteger.valueOf(a);

BigInteger xb = BigInteger.valueOf(b);

long gcd1 = gcd(a, b);

long gcd2 = gcd2(a, b);

long gcd = xa.gcd(xb).longValue();

long[] euclid1 = euclid(a, b);

long[] euclid2 = euclid2(a, b);

int inv1 = 0;

int inv2 = 0;

int inv = 0;

if (gcd == 1 && b > 0) {

inv1 = modInverse(a, b);

inv2 = modInverse2(a, b);

inv = xa.modInverse(xb).intValue();

}

if (gcd1 != gcd || gcd2 != gcd || !Arrays.equals(euclid1, euclid2) || euclid1[0] != gcd || inv1 != inv

|| inv2 != inv) {

System.err.println(a + " " + b);

}

}

long a = 6;

long b = 9;

long[] res = euclid(a, b);

System.out.println(res[1] + " \* (" + a + ") " + " + " + res[2] + " \* (" + b + ") = gcd(" + a + "," + b + ") = "

+ res[0]);

System.out.println(Arrays.toString(generateInverse(7)));

}

}

# Euler Circuit (Path which visits each edge once)

import java.util.\*;

import java.util.stream.Stream;

public class EulerCycle {

public static List<Integer> eulerCycleUndirected(List<Integer>[] graph, int u) {

Set<Long> usedEdges = new HashSet<>();

int n = graph.length;

int[] curEdge = new int[n];

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

dfs(graph, curEdge, usedEdges, res, u);

Collections.reverse(res);

return res;

}

static void dfs(List<Integer>[] graph, int[] curEdge, Set<Long> usedEdges, List<Integer> res, int u) {

while (curEdge[u] < graph[u].size()) {

int v = graph[u].get(curEdge[u]++);

if (usedEdges.add(((long) Math.min(u, v) << 32) + Math.max(u, v)))

dfs(graph, curEdge, usedEdges, res, v);

}

res.add(u);

}

public static List<Integer> eulerCycleUndirected2(List<Integer>[] graph, int u) {

int[] curEdge = new int[graph.length];

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

Stack<Integer> stack = new Stack<>();

Set<Long> usedEdges = new HashSet<>();

stack.add(u);

while (!stack.isEmpty()) {

u = stack.pop();

while (curEdge[u] < graph[u].size()) {

int v = graph[u].get(curEdge[u]++);

if (usedEdges.add((((long) Math.min(u, v) << 32) + Math.max(u, v)))) {

stack.push(u);

u = v;

}

}

res.add(u);

}

Collections.reverse(res);

return res;

}

public static List<Integer> eulerCycleDirected(List<Integer>[] graph, int u) {

int n = graph.length;

int[] curEdge = new int[n];

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

dfs(graph, curEdge, res, u);

Collections.reverse(res);

return res;

}

static void dfs(List<Integer>[] graph, int[] curEdge, List<Integer> res, int u) {

while (curEdge[u] < graph[u].size()) {

dfs(graph, curEdge, res, graph[u].get(curEdge[u]++));

}

res.add(u);

}

public static List<Integer> eulerCycleDirected2(List<Integer>[] graph, int v) {

int[] curEdge = new int[graph.length];

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

Stack<Integer> stack = new Stack<>();

stack.add(v);

while (!stack.isEmpty()) {

v = stack.pop();

while (curEdge[v] < graph[v].size()) {

stack.push(v);

v = graph[v].get(curEdge[v]++);

}

res.add(v);

}

Collections.reverse(res);

return res;

}

// Usage example

public static void main(String[] args) {

int n = 5;

List<Integer>[] g = Stream.generate(ArrayList::new).limit(n).toArray(List[]::new);

g[0].add(1);

g[1].add(2);

g[2].add(0);

g[1].add(3);

g[3].add(4);

g[4].add(1);

System.out.println(eulerCycleDirected(g, 0));

System.out.println(eulerCycleDirected2(g, 0));

n = 5;

g = Stream.generate(ArrayList::new).limit(n).toArray(List[]::new);

g[0].add(1);

g[1].add(0);

g[1].add(2);

g[2].add(1);

g[2].add(3);

g[3].add(2);

g[0].add(3);

g[3].add(0);

g[0].add(4);

g[4].add(0);

g[1].add(4);

g[4].add(1);

g[0].add(2);

g[2].add(0);

g[1].add(3);

g[3].add(1);

System.out.println(eulerCycleUndirected(g, 2));

System.out.println(eulerCycleUndirected2(g, 2));

}

}

# Prime Factorization

Note: use sieve from Primes and Divisors to accelerate method “factorize”

import java.util.\*;

public class Factorization {

// prime\_divisor -> power

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;

}

public static long ferma(long n) {

long x = (long) Math.sqrt(n);

long y = 0;

long r = x \* x - y \* y - n;

while (true) {

if (r == 0)

return x != y ? x - y : x + y;

else if (r > 0) {

r -= y + y + 1;

++y;

} else {

r += x + x + 1;

++x;

}

}

}

public static long pollard(long n) {

Random rnd = new Random(1);

long x = Math.abs(rnd.nextLong()) % n;

long y = x;

while (true) {

x = f(x, n);

y = f(f(y, n), n);

if (x == y)

return -1;

long d = gcd(Math.abs(x - y), n);

if (d != 1)

return d;

}

}

static long gcd(long a, long b) {

return a == 0 ? b : gcd(b % a, a);

}

static long f(long x, long n) {

return (41 \* x + 1) % n;

}

// Usage example

public static void main(String[] args) {

Map<Long, Integer> f = factorize(24);

System.out.println(f);

System.out.println(Arrays.toString(getAllDivisors(16)));

long n = 1000\_003L \* 100\_000\_037;

System.out.println(ferma(n));

System.out.println(pollard(n));

}

}

# Fenwick Tree (Fast Prefix Sums)

public class FenwickTree {

// T[i] += value

public static void add(int[] t, int i, int value) {

for (; i < t.length; i |= i + 1)

t[i] += value;

}

// sum[0..i]

public static int sum(int[] t, int i) {

int res = 0;

for (; i >= 0; i = (i & (i + 1)) - 1)

res += t[i];

return res;

}

///////////////////////////////////////////////////

// T[i] = max(T[i], value)

public static void set(int[] t, int i, int value) {

for (; i < t.length; i |= i + 1)

t[i] = Math.max(t[i], value);

}

// max[0..i]

public static int max(int[] t, int i) {

int res = Integer.MIN\_VALUE;

for (; i >= 0; i = (i & (i + 1)) - 1)

res = Math.max(res, t[i]);

return res;

}

// Usage example

public static void main(String[] args) {

int[] t = new int[10];

add(t, 0, 1);

add(t, 9, -2);

System.out.println(-1 == sum(t, 9));

}

}

# Fast Fourier Transform

import java.math.BigInteger;

import java.util.Random;

public class FFT {

// a.length == b.length == 2^x

public static void fft(double[] a, double[] b, boolean invert) {

int n = a.length;

int shift = 32 - Integer.numberOfTrailingZeros(n);

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

int j = Integer.reverse(i << shift);

if (i < j) {

double temp = a[i];

a[i] = a[j];

a[j] = temp;

temp = b[i];

b[i] = b[j];

b[j] = temp;

}

}

for (int len = 2; len <= n; len <<= 1) {

int halfLen = len >> 1;

double angle = 2 \* Math.PI / len \* (invert ? -1 : 1);

double wLenA = Math.cos(angle);

double wLenB = Math.sin(angle);

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

double wA = 1;

double wB = 0;

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

double uA = a[i + j];

double uB = b[i + j];

double vA = a[i + j + halfLen] \* wA - b[i + j + halfLen] \* wB;

double vB = a[i + j + halfLen] \* wB + b[i + j + halfLen] \* wA;

a[i + j] = uA + vA;

b[i + j] = uB + vB;

a[i + j + halfLen] = uA - vA;

b[i + j + halfLen] = uB - vB;

double nextWA = wA \* wLenA - wB \* wLenB;

wB = wA \* wLenB + wB \* wLenA;

wA = nextWA;

}

}

}

if (invert) {

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

a[i] /= n;

b[i] /= n;

}

}

}

public static long[] multiply(long[] a, long[] b) {

int resultSize = Integer.highestOneBit(Math.max(a.length, b.length) - 1) << 2;

resultSize = Math.max(resultSize, 2);

double[] aReal = new double[resultSize];

double[] aImaginary = new double[resultSize];

double[] bReal = new double[resultSize];

double[] bImaginary = new double[resultSize];

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

aReal[i] = a[i];

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

bReal[i] = b[i];

fft(aReal, aImaginary, false);

fft(bReal, bImaginary, false);

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

double real = aReal[i] \* bReal[i] - aImaginary[i] \* bImaginary[i];

aImaginary[i] = aImaginary[i] \* bReal[i] + bImaginary[i] \* aReal[i];

aReal[i] = real;

}

fft(aReal, aImaginary, true);

long[] result = new long[resultSize];

long carry = 0;

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

result[i] = Math.round(aReal[i]) + carry;

carry = result[i] / 10;

result[i] %= 10;

}

return result;

}

// random test

public static void main(String[] args) {

Random rnd = new Random(1);

for (int step = 0; step < 1000; step++) {

int n1 = rnd.nextInt(10) + 1;

String s1 = "";

long[] a = new long[n1];

for (int i = 0; i < n1; i++) {

int x = rnd.nextInt(10);

s1 = x + s1;

a[i] = x;

}

int n2 = rnd.nextInt(10) + 1;

String s2 = "";

long[] b = new long[n2];

for (int i = 0; i < n2; i++) {

int x = rnd.nextInt(10);

s2 = x + s2;

b[i] = x;

}

long[] res = multiply(a, b);

String s = "";

for (long v : res) {

s = v + s;

}

BigInteger mul = new BigInteger(s1).multiply(new BigInteger(s2));

if (!mul.equals(new BigInteger(s)))

throw new RuntimeException();

}

}

}

# Floyd-Warshall Algorithm (All Shortest Paths)

import java.util.Arrays;

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) ? -1 : i;

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;

}

public static int[] restorePath(int[][] pred, int i, int j) {

int n = pred.length;

int[] path = new int[n];

int pos = n;

while (true) {

path[--pos] = j;

if (i == j) break;

j = pred[i][j];

}

return Arrays.copyOfRange(path, pos, n);

}

// Usage example

public static void main(String[] args) {

int[][] dist = {{0, 3, 2}, {0, 0, 1}, {INF, 0, 0}};

int[][] pred = floydWarshall(dist);

int[] path = restorePath(pred, 0, 1);

System.out.println(0 == dist[0][0]);

System.out.println(2 == dist[0][1]);

System.out.println(2 == dist[0][2]);

System.out.println(-1 == pred[0][0]);

System.out.println(2 == pred[0][1]);

System.out.println(0 == pred[0][2]);

System.out.println(Arrays.equals(new int[]{0, 2, 1}, path));

}

}

# Gaussian Elimination

public class Gauss {

public static double[] gauss(double[][] a, double[] b) {

int n = a.length;

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

int best = row;

for (int i = row + 1; i < n; i++)

if (Math.abs(a[best][row]) < Math.abs(a[i][row]))

best = i;

double[] tt = a[row];

a[row] = a[best];

a[best] = tt;

double t = b[row];

b[row] = b[best];

b[best] = t;

for (int i = row + 1; i < n; i++)

a[row][i] /= a[row][row];

b[row] /= a[row][row];

// a[row][row] = 1;

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

double x = a[i][row];

if (i != row && x != 0) {

// row + 1 instead of row is an optimization

for (int j = row + 1; j < n; j++)

a[i][j] -= a[row][j] \* x;

b[i] -= b[row] \* x;

}

}

}

return b;

}

// Usage example

public static void main(String[] args) {

double[][] a = { { 4, 2, -1 }, { 2, 4, 3 }, { -1, 3, 5 } };

double[] b = { 1, 0, 0 };

double[][] a1 = a.clone();

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

a1[i] = a[i].clone();

double[] b1 = b.clone();

double[] x = gauss(a, b);

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

double y = 0;

for (int j = 0; j < a[i].length; j++)

y += a1[i][j] \* x[j];

if (Math.abs(b1[i] - y) > 1e-9) {

System.err.println("error");

return;

}

}

}

}

# Graph Colouring

import java.util.\*;

import java.util.stream.Stream;

public class GraphColoringGreedy2 {

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

int n = graph.length;

int[] used = new int[n];

int[] colors = new int[n];

Arrays.fill(colors, -1);

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

int best\_cnt = -1;

int bestu = -1;

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

if (colors[u] == -1) {

int cnt = Integer.bitCount(used[u]);

if (best\_cnt < cnt) {

best\_cnt = cnt;

bestu = u;

}

}

}

int c = Integer.numberOfTrailingZeros(~used[bestu]);

colors[bestu] = c;

for (int v : graph[bestu]) {

used[v] |= 1 << c;

}

}

return colors;

}

// Usage example

public static void main(String[] args) {

int n = 5;

@SuppressWarnings("unchecked")

    List<Integer>[] g = Stream.generate(ArrayList::new).limit(n).toArray(List[]::new);

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)));

}

}

# Hill Climbing (Optimize Function of Two Arguments)

import java.util.function.BiFunction;

public class HillClimbing {

public static double findMinimum(BiFunction<Double, Double, Double> f) {

double curX = 0;

double curY = 0;

double curF = f.apply(curX, curY);

for (double step = 1e6; step > 1e-7; ) {

double bestF = curF;

double bestX = curX;

double bestY = curY;

boolean find = false;

for (int i = 0; i < 6; i++) {

double a = 2 \* Math.PI \* i / 6;

double nextX = curX + step \* Math.cos(a);

double nextY = curY + step \* Math.sin(a);

double nextF = f.apply(nextX, nextY);

if (bestF > nextF) {

bestF = nextF;

bestX = nextX;

bestY = nextY;

find = true;

}

}

if (!find) {

step /= 2;

} else {

curX = bestX;

curY = bestY;

curF = bestF;

}

}

System.out.println(curX + " " + curY);

return curF;

}

// Usage example

public static void main(String[] args) {

System.out.println(findMinimum((x, y) -> (x - 2) \* (x - 2) + (y - 3) \* (y - 3)));

}

}

# Inversion Counting

import java.util.\*;

public class Inversions {

// warning: a is modified during processing

public static long inversions(int[] a, int low, int high) {

if (high - low < 2)

return 0;

int mid = (low + high) >>> 1;

long res = inversions(a, low, mid) + inversions(a, mid, high);

int[] b = Arrays.copyOfRange(a, low, mid);

for (int i = low, j = mid, k = 0; k < b.length; i++)

if (j == high || b[k] <= a[j]) {

a[i] = b[k++];

} else {

a[i] = a[j++];

res += b.length - k;

}

return res;

}

// random test

public static void main(String[] args) {

Random rnd = new Random(1);

for (int step = 0; step < 1000; step++) {

int n = rnd.nextInt(100) + 1;

int[] p = rnd.ints(n, 0, n).toArray();

long res1 = inversions(p.clone(), 0, p.length);

long res2 = slowInversions(p);

if (res1 != res2)

throw new RuntimeException();

}

}

static long slowInversions(int[] p) {

long res = 0;

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

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

if (p[j] > p[i])

++res;

return res;

}

}

# Karatsuba Algorithm (Multiplying Large Polynomials)

import java.util.Arrays;

// https://en.wikipedia.org/wiki/Karatsuba\_algorithm

public class KaratsubaMultiply {

public static int[] karatsubaMultiply(int[] a, int[] b) {

if (a.length < b.length) a = Arrays.copyOf(a, b.length);

if (a.length > b.length) b = Arrays.copyOf(b, a.length);

int n = a.length;

int[] res = new int[n + n];

if (n <= 10) {

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

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

res[i + j] = res[i + j] + a[i] \* b[j];

} else {

int k = n >> 1;

int[] a1 = Arrays.copyOfRange(a, 0, k);

int[] a2 = Arrays.copyOfRange(a, k, n);

int[] b1 = Arrays.copyOfRange(b, 0, k);

int[] b2 = Arrays.copyOfRange(b, k, n);

int[] a1b1 = karatsubaMultiply(a1, b1);

int[] a2b2 = karatsubaMultiply(a2, b2);

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

a2[i] = a2[i] + a1[i];

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

b2[i] = b2[i] + b1[i];

int[] r = karatsubaMultiply(a2, b2);

for (int i = 0; i < a1b1.length; i++)

r[i] = r[i] - a1b1[i];

for (int i = 0; i < a2b2.length; i++)

r[i] = r[i] - a2b2[i];

System.arraycopy(r, 0, res, k, r.length);

for (int i = 0; i < a1b1.length; i++)

res[i] = res[i] + a1b1[i];

for (int i = 0; i < a2b2.length; i++)

res[i + n] = res[i + n] + a2b2[i];

}

return res;

}

// Usage example

public static void main(String[] args) {

// (3\*x^2+2\*x+1) \* (4\*x+3) = 12\*x^3 + 17\*x^2 + 10\*x + 3

System.out.println(Arrays.equals(new int[]{3, 10, 17, 12, 0, 0}, karatsubaMultiply(new int[]{1, 2, 3}, new int[]{3, 4})));

}

}

# KMP String Searching

**import** java.util.ArrayList;

**import** java.util.Random;

// https://en.wikipedia.org/wiki/Knuth–Morris–Pratt\_algorithm

**public** **class** Kmp {

**public** **static** **int**[] prefixFunction(String s) {

**int**[] p = **new** **int**[s.length()+1];

p[0] = -1;

**int** k = 0;

**for** (**int** i = 1; i < s.length(); i++) {

**while** (k > 0 && s.charAt(k) != s.charAt(i))

k = p[k];

**if** (s.charAt(k) == s.charAt(i))

++k;

p[i+1] = k;

}

**return** p;

}

**public** **static** ArrayList<Integer> kmpMatcher(String text, String pattern) {

ArrayList<Integer> p = **new** ArrayList<Integer>();

**if** (pattern.length() == 0) {

p.add(0);

**return** p;

} **else** **if** (pattern.length() == 1) {

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

**if** (text.charAt(i) == pattern.charAt(0)) {

p.add(i);

}

}

**return** p;

}

**int** m = 0;

**int** i = 0;

**int**[] t = *prefixFunction*(pattern);

**while** (m + i < text.length()) {

**if** (pattern.charAt(i) == text.charAt(m + i)) {

**if** (i + 1 == pattern.length()) {

p.add(m);

m = m + i - t[i];

i = t[i];

} **else** {

i++;

}

} **else** {

**if** (t[i] > -1) {

m = m + i - t[i];

i = t[i];

} **else** {

m = m + i + 1;

i = 0;

}

}

}

**return** p;

}

// random tests

**public** **static** **void** main(String[] args) {

Random rnd = **new** Random(1);

**for** (**int** step = 0; step < 10\_000; step++) {

String s = *getRandomString*(rnd, 10);

String pattern = *getRandomString*(rnd, 5);

ArrayList<Integer> matches = *kmpMatcher*(s, pattern);

**int** pos1;

**if** (matches.size() > 0) {

pos1 = matches.get(0);

} **else** {

pos1 = -1;

}

**int** pos2 = s.indexOf(pattern);

**if** (pos1 != pos2)

**throw** **new** RuntimeException();

**else**

System.***out***.println(step);

}

}

**static** String getRandomString(Random rnd, **int** maxlen) {

**int** n = rnd.nextInt(maxlen);

**char**[] s = **new** **char**[n];

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

s[i] = (**char**) ('a' + rnd.nextInt(3));

**return** **new** String(s);

}

}

# Lowest Common Ancestor

import java.util.\*;

import java.util.stream.Stream;

// Answering LCA queries in O(1) with O(n\*log(n)) preprocessing

public class LcaSparseTable {

int len;

int[][] up;

int[] tin;

int[] tout;

int time;

void dfs(List<Integer>[] tree, int u, int p) {

tin[u] = time++;

up[0][u] = p;

for (int i = 1; i < len; i++)

up[i][u] = up[i - 1][up[i - 1][u]];

for (int v : tree[u])

if (v != p)

dfs(tree, v, u);

tout[u] = time++;

}

public LcaSparseTable(List<Integer>[] tree, int root) {

int n = tree.length;

len = 1;

while ((1 << len) <= n) ++len;

up = new int[len][n];

tin = new int[n];

tout = new int[n];

dfs(tree, root, root);

}

boolean isParent(int parent, int child) {

return tin[parent] <= tin[child] && tout[child] <= tout[parent];

}

public int lca(int a, int b) {

if (isParent(a, b))

return a;

if (isParent(b, a))

return b;

for (int i = len - 1; i >= 0; i--)

if (!isParent(up[i][a], b))

a = up[i][a];

return up[0][a];

}

// Usage example

public static void main(String[] args) {

List<Integer>[] tree = Stream.generate(ArrayList::new).limit(5).toArray(List[]::new);

tree[0].add(1);

tree[1].add(0);

tree[1].add(2);

tree[2].add(1);

tree[3].add(1);

tree[1].add(3);

tree[0].add(4);

tree[4].add(0);

LcaSparseTable t = new LcaSparseTable(tree, 0);

System.out.println(1 == t.lca(3, 2));

System.out.println(0 == t.lca(2, 4));

}

}

# Linear Equality

public class LinearEquality {

// number of integer solutions of equality a[0]\*x1 + ... + a[n-1]\*xn = b, xi >= 0

public static long countSolutions(int[] a, int b) {

long[] dp = new long[b + 1];

dp[0] = 1;

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

for (int j = a[i]; j <= b; j++) {

dp[j] += dp[j - a[i]];

}

}

return dp[b];

}

public static void main(String[] args) {

System.out.println(5 == countSolutions(new int[] { 1, 2, 3 }, 5));

}

}

# Line Geometry

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) {

}

}

# Longest Increasing Subsequence

import java.util.\*;

// https://en.wikipedia.org/wiki/Longest\_increasing\_subsequence

public class Lis {

public static int[] getLis(int[] x) {

int n = x.length;

int[] len = new int[n];

Arrays.fill(len, 1);

int[] pred = new int[n];

Arrays.fill(pred, -1);

int bi = 0;

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

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

if (x[j] < x[i] && len[i] < len[j] + 1) {

len[i] = len[j] + 1;

pred[i] = j;

}

}

if (len[bi] < len[i]) {

bi = i;

}

}

int cnt = len[bi];

int[] res = new int[cnt];

for (int i = bi; i != -1; i = pred[i]) {

res[--cnt] = x[i];

}

return res;

}

// Usage example

public static void main(String[] args) {

int[] a = { 1, 5, 4, 2, 3, 7, 6 };

int[] lis = getLis(a);

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

}

}

# Matrix Methods

public class Matrix {

public static int[][] matrixAdd(int[][] a, int[][] b) {

int n = a.length;

int m = a[0].length;

int[][] res = new int[n][m];

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

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

res[i][j] = a[i][j] + b[i][j];

}

}

return res;

}

public static int[][] matrixMul(int[][] a, int[][] b) {

int n = a.length;

int m = a[0].length;

int k = b[0].length;

int[][] res = new int[n][k];

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

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

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

res[i][j] = res[i][j] + a[i][p] \* b[p][j];

}

}

}

return res;

}

public static int[][] matrixPow(int[][] a, int p) {

if (p == 0) {

return matrixUnit(a.length);

} else if (p % 2 == 0) {

return matrixPow(matrixMul(a, a), p / 2);

} else {

return matrixMul(a, matrixPow(a, p - 1));

}

}

public static int[][] matrixPowSum(int[][] a, int p) {

int n = a.length;

if (p == 0) {

return new int[n][n];

}

if (p % 2 == 0) {

return matrixMul(matrixPowSum(a, p / 2), matrixAdd(matrixUnit(n), matrixPow(a, p / 2)));

} else {

return matrixAdd(a, matrixMul(matrixPowSum(a, p - 1), a));

}

}

public static int[][] matrixUnit(int n) {

int[][] res = new int[n][n];

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

res[i][i] = 1;

}

return res;

}

// Usage example

public static void main(String[] args) {

int[][] a = { { 1, 2 }, { 3, 4 } };

int[][] b = matrixUnit(2);

int[][] c = matrixMul(a, b);

}

}

# Max Flow

import java.util.\*;

import java.util.stream.Stream;

public class MaxFlowEdmondsKarp {

static class Edge {

int s, t, rev, cap, f;

public Edge(int s, int t, int rev, int cap) {

this.s = s;

this.t = t;

this.rev = rev;

this.cap = cap;

}

}

public static void addEdge(List<Edge>[] graph, int s, int t, int cap) {

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

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

}

public static int maxFlow(List<Edge>[] graph, int s, int t) {

int flow = 0;

int[] q = new int[graph.length];

while (true) {

int qt = 0;

q[qt++] = s;

Edge[] pred = new Edge[graph.length];

for (int qh = 0; qh < qt && pred[t] == null; qh++) {

int cur = q[qh];

for (Edge e : graph[cur]) {

if (pred[e.t] == null && e.cap > e.f) {

pred[e.t] = e;

q[qt++] = e.t;

}

}

}

if (pred[t] == null)

break;

int df = Integer.MAX\_VALUE;

for (int u = t; u != s; u = pred[u].s)

df = Math.min(df, pred[u].cap - pred[u].f);

for (int u = t; u != s; u = pred[u].s) {

pred[u].f += df;

graph[pred[u].t].get(pred[u].rev).f -= df;

}

flow += df;

}

return flow;

}

// Usage example

public static void main(String[] args) {

List<Edge>[] graph = Stream.generate(ArrayList::new).limit(3).toArray(List[]::new);

addEdge(graph, 0, 1, 3);

addEdge(graph, 0, 2, 2);

addEdge(graph, 1, 2, 2);

System.out.println(4 == maxFlow(graph, 0, 2));

}

}

# Max Palindrome

// Find maximum palindromic subsequence of the given string

public class MaxPalindrome {

public static String maxPalindrome(String p) {

int n = p.length();

char[] s = p.toCharArray();

int[][] dp = new int[n + 1][n + 1];

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

dp[i][0] = dp[0][i] = i;

}

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

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

dp[i + 1][j + 1] = (s[i] == s[n - 1 - j]) ? dp[i][j] : Math.min(dp[i][j + 1] + 1, dp[i + 1][j] + 1);

}

}

int min = n;

int x = 0;

int y = n;

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

if (min > dp[i][n - i]) {

min = dp[i][n - i];

x = i;

y = n - i;

}

}

String middle = "";

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

if (min > dp[i][n - i - 1]) {

min = dp[i][n - i - 1];

x = i;

y = n - i - 1;

middle = "" + s[i];

}

}

String res = "";

while (x > 0 && y > 0) {

int a = dp[x - 1][y - 1];

int b = dp[x - 1][y];

int c = dp[x][y - 1];

int m = Math.min(a, Math.min(b, c));

if (a == m) {

res += s[x - 1];

--x;

--y;

} else if (b == m) {

--x;

} else {

--y;

}

}

return new StringBuilder(res).reverse() + middle + res;

}

// Usage example

public static void main(String[] args) {

String res = maxPalindrome("3213");

System.out.println("323".equals(res));

}

}

# Min Cost Flow

import java.util.\*;

import java.util.stream.Stream;

public class MinCostFlowBF {

static class Edge {

int to, f, cap, cost, rev;

Edge(int to, int cap, int cost, int rev) {

this.to = to;

this.cap = cap;

this.cost = cost;

this.rev = rev;

}

}

public static void addEdge(List<Edge>[] graph, int s, int t, int cap, int cost) {

graph[s].add(new Edge(t, cap, cost, graph[t].size()));

graph[t].add(new Edge(s, 0, -cost, graph[s].size() - 1));

}

static void bellmanFord(List<Edge>[] graph, int s, int[] dist, int[] prevnode, int[] prevedge, int[] curflow) {

int n = graph.length;

Arrays.fill(dist, 0, n, Integer.MAX\_VALUE);

dist[s] = 0;

curflow[s] = Integer.MAX\_VALUE;

boolean[] inqueue = new boolean[n];

int[] q = new int[n];

int qt = 0;

q[qt++] = s;

for (int qh = 0; (qh - qt) % n != 0; qh++) {

int u = q[qh % n];

inqueue[u] = false;

for (int i = 0; i < graph[u].size(); i++) {

Edge e = graph[u].get(i);

if (e.f >= e.cap)

continue;

int v = e.to;

int ndist = dist[u] + e.cost;

if (dist[v] > ndist) {

dist[v] = ndist;

prevnode[v] = u;

prevedge[v] = i;

curflow[v] = Math.min(curflow[u], e.cap - e.f);

if (!inqueue[v]) {

inqueue[v] = true;

q[qt++ % n] = v;

}

}

}

}

}

public static int[] minCostFlow(List<Edge>[] graph, int s, int t, int maxf) {

int n = graph.length;

int[] dist = new int[n];

int[] curflow = new int[n];

int[] prevedge = new int[n];

int[] prevnode = new int[n];

int flow = 0;

int flowCost = 0;

while (flow < maxf) {

bellmanFord(graph, s, dist, prevnode, prevedge, curflow);

if (dist[t] == Integer.MAX\_VALUE)

break;

int df = Math.min(curflow[t], maxf - flow);

flow += df;

for (int v = t; v != s; v = prevnode[v]) {

Edge e = graph[prevnode[v]].get(prevedge[v]);

e.f += df;

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

flowCost += df \* e.cost;

}

}

return new int[]{flow, flowCost};

}

// Usage example

public static void main(String[] args) {

List<Edge>[] graph = Stream.generate(ArrayList::new).limit(3).toArray(List[]::new);

addEdge(graph, 0, 1, 3, 1);

addEdge(graph, 0, 2, 2, 1);

addEdge(graph, 1, 2, 2, 1);

int[] res = minCostFlow(graph, 0, 2, Integer.MAX\_VALUE);

int flow = res[0];

int flowCost = res[1];

System.out.println(4 == flow);

System.out.println(6 == flowCost);

}

}

# Maximal Independent Set

import java.util.\*;

public class Mis {

public static int mis(BitSet[] g, BitSet unused) {

int v = -1;

for (int u = unused.nextSetBit(0); u >= 0; u = unused.nextSetBit(u + 1))

if (v == -1 || g[v].cardinality() > g[u].cardinality())

v = u;

if (v == -1)

return 0;

int res = -1;

BitSet nv = (BitSet) g[v].clone();

nv.and(unused);

for (int y = nv.nextSetBit(0); y >= 0; y = nv.nextSetBit(y + 1)) {

BitSet newUnused = (BitSet) unused.clone();

newUnused.andNot(g[y]);

res = Math.max(res, 1 + mis(g, newUnused));

}

return res;

}

// random test

public static void main(String[] args) {

Random rnd = new Random(1);

for (int step = 0; step < 1000; step++) {

int n = rnd.nextInt(16) + 1;

BitSet[] g = new BitSet[n];

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

g[i] = new BitSet(n);

// for convenience of mis()

g[i].set(i);

}

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

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

if (rnd.nextBoolean()) {

g[i].set(j);

g[j].set(i);

}

BitSet unused = new BitSet();

unused.set(0, n);

int res1 = mis(g, unused);

int res2 = misSlow(g);

if (res1 != res2)

throw new RuntimeException();

}

}

static int misSlow(BitSet[] g) {

int res = 0;

int n = g.length;

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

boolean ok = true;

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

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

ok &= (set & (1 << i)) == 0 || (set & (1 << j)) == 0 || !g[i].get(j);

if (ok)

res = Math.max(res, Integer.bitCount(set));

}

return res;

}

}

# Point to Line Segment Distance

import java.awt.geom.Line2D;

import java.util.Random;

public class PointToSegmentDistance {

public static double pointToSegmentDistance(int x, int y, int x1, int y1, int x2, int y2) {

long dx = x2 - x1;

long dy = y2 - y1;

long px = x - x1;

long py = y - y1;

long squaredLength = dx \* dx + dy \* dy;

long dotProduct = dx \* px + dy \* py;

if (dotProduct <= 0 || squaredLength == 0)

return fastHypot(px, py);

if (dotProduct >= squaredLength)

return fastHypot(px - dx, py - dy);

double q = (double) dotProduct / squaredLength;

return fastHypot(px - q \* dx, py - q \* dy);

}

static double fastHypot(double x, double y) {

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

}

// Line2D.ptLineDist

public static double pointToLineDistance(long x, long y, long a, long b, long c) {

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

}

// random test

public static void main(String[] args) {

Random rnd = new Random(1);

for (int step = 0; step < 1000\_000; step++) {

int r = 10;

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

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

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

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

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

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

double res1 = pointToSegmentDistance(x, y, x1, y1, x2, y2);

double res2 = Line2D.ptSegDist(x1, y1, x2, y2, x, y);

if (!(Math.abs(res1 - res2) < 1e-9))

throw new RuntimeException();

}

}

}

# Prim’s Algorithm (Minimum Spanning Tree for Weighted Graph)

import java.util.Arrays;

// https://en.wikipedia.org/wiki/Prim%27s\_algorithm

public class Prim {

public static long mstPrim(int[][] d) {

int n = d.length;

int[] prev = new int[n];

int[] dist = new int[n];

Arrays.fill(dist, Integer.MAX\_VALUE);

dist[0] = 0;

boolean[] visited = new boolean[n];

long res = 0;

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

int u = -1;

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

if (!visited[j] && (u == -1 || dist[u] > dist[j]))

u = j;

}

res += dist[u];

visited[u] = true;

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

if (!visited[j] && dist[j] > d[u][j]) {

dist[j] = d[u][j];

prev[j] = u;

}

}

}

return res;

}

// Usage example

public static void main(String[] args) {

}

}

# Primes and Divisors

import java.util.\*;

public class PrimesAndDivisors {

public static int[] generatePrimes(int n) {

boolean[] sieve = new boolean[n + 1];

Arrays.fill(sieve, 2, n + 1, true);

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

if (sieve[i])

for (int j = i \* i; j <= n; j += i)

sieve[j] = false;

int[] primes = new int[n + 1];

int cnt = 0;

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

if (sieve[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;

}

// Usage example

public static void main(String[] args) {

int n = 31;

int[] primes1 = generatePrimes(n);

int[] primes2 = generatePrimesLinear(n);

System.out.println(Arrays.toString(primes1));

System.out.println(Arrays.toString(primes2));

System.out.println(Arrays.equals(primes1, primes2));

System.out.println(Arrays.toString(numberOfPrimeDivisors(n)));

System.out.println(Arrays.toString(generateDivisorTable(n)));

n = 1000;

int[] phi = generatePhi(n);

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

if (phi[i] != phi(i)) {

System.err.println(i);

}

}

}

}

# Rational Numbers

import java.math.BigInteger;

public class Rational implements Comparable<Rational> {

public static final Rational ZERO = new Rational(0);

public static final Rational ONE = new Rational(1);

public static final Rational POSITIVE\_INFINITY = new Rational(1, 0);

public static final Rational NEGATIVE\_INFINITY = new Rational(-1, 0);

final BigInteger num;

final BigInteger den;

public Rational(BigInteger num, BigInteger den) {

BigInteger gcd = num.gcd(den);

BigInteger g = den.signum() > 0 ? gcd : den.signum() < 0 ? gcd.negate() : BigInteger.ONE;

this.num = num.divide(g);

this.den = den.divide(g);

}

public Rational(long num, long den) {

this(BigInteger.valueOf(num), BigInteger.valueOf(den));

}

public Rational(long num) {

this.num = BigInteger.valueOf(num);

this.den = BigInteger.ONE;

}

public Rational add(Rational r) {

return new Rational(num.multiply(r.den).add(r.num.multiply(den)), den.multiply(r.den));

}

public Rational sub(Rational r) {

return new Rational(num.multiply(r.den).subtract(r.num.multiply(den)), den.multiply(r.den));

}

public Rational mul(Rational r) {

return new Rational(num.multiply(r.num), den.multiply(r.den));

}

public Rational div(Rational r) {

return new Rational(num.multiply(r.den), den.multiply(r.num));

}

public Rational negate() {

return new Rational(num.negate(), den);

}

public Rational inverse() {

return new Rational(den, num);

}

public Rational abs() {

return new Rational(num.abs(), den);

}

public int signum() {

return num.signum();

}

public double doubleValue() {

return num.doubleValue() / den.doubleValue();

}

public long longValue() {

return num.longValue() / den.longValue();

}

public int compareTo(Rational other) {

return (num.multiply(other.den).compareTo(other.num.multiply(den)));

}

public boolean equals(Object obj) {

return num.equals(((Rational) obj).num) && den.equals(((Rational) obj).den);

}

public int hashCode() {

return num.hashCode() \* 31 + den.hashCode();

}

public String toString() {

return num + "/" + den;

}

// Usage example

public static void main(String[] args) {

Rational a = new Rational(1, 3);

Rational b = new Rational(1, 6);

Rational c = new Rational(1, 2);

System.out.println(true == c.equals(a.add(b)));

Rational d = new Rational(1, -2);

System.out.println(d);

}

}

# Shortest Hamiltonian Cycle (Visits each Vertex Once)

import java.util.\*;

public class ShortestHamiltonianCycle2 {

public static int getShortestHamiltonianCycle(int[][] dist) {

int n = dist.length;

int[][] dp = new int[1 << n][n];

for (int[] d : dp)

Arrays.fill(d, Integer.MAX\_VALUE / 2);

dp[0][0] = 0;

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

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

if ((mask & 1 << next) == 0) {

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

dp[mask | 1 << next][next] = Math.min(dp[mask | 1 << next][next], dp[mask][cur] + dist[cur][next]);

}

}

}

}

return dp[(1 << n) - 1][0];

}

// Usage example

public static void main(String[] args) {

int[][] dist = {{0, 1, 1}, {1, 0, 10}, {1, 10, 0}};

int tourLength = getShortestHamiltonianCycle(dist);

System.out.println(12 == tourLength);

}

}

# Simpson Function Integration

import java.util.function.DoubleFunction;

public class SimpsonIntegration {

public static double integrate(DoubleFunction<Double> f, double a, double b) {

double eps = 1e-10;

double m = (a + b) / 2;

double am = simpsonIntegration(f, a, m);

double mb = simpsonIntegration(f, m, b);

double ab = simpsonIntegration(f, a, b);

if (Math.abs(am + mb - ab) < eps)

return ab;

return integrate(f, a, m) + integrate(f, m, b);

}

static double simpsonIntegration(DoubleFunction<Double> f, double a, double b) {

return (f.apply(a) + 4 \* f.apply((a + b) / 2) + f.apply(b)) \* (b - a) / 6;

}

// Usage example

public static void main(String[] args) {

System.out.println(integrate(x -> Math.sin(x), 0, Math.PI / 2));

}

}

# Suffix Array

**import** java.util.\*;

**import** java.util.stream.IntStream;

// https://en.wikipedia.org/wiki/Suffix\_array

**public** **class** SuffixArray {

// sort suffixes of S in O(n\*log(n))

**public** **static** **int**[] suffixArray(CharSequence S) {

**int** n = S.length();

// stable sort of characters

**int**[] sa = IntStream.*range*(0, n).mapToObj(i -> n - 1 - i).

sorted((a, b) -> Character.*compare*(S.charAt(a), S.charAt(b))).mapToInt(Integer::intValue).toArray();

**int**[] classes = S.chars().toArray();

// sa[i] - suffix on i'th position after sorting by first len characters

// classes[i] - equivalence class of the i'th suffix after sorting by first len characters

**for** (**int** len = 1; len < n; len \*= 2) {

**int**[] c = classes.clone();

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

// condition sa[i - 1] + len < n simulates 0-symbol at the end of the string

// a separate class is created for each suffix followed by simulated 0-symbol

classes[sa[i]] = i > 0 && c[sa[i - 1]] == c[sa[i]] && sa[i - 1] + len < n && c[sa[i - 1] + len / 2] == c[sa[i] + len / 2] ? classes[sa[i - 1]] : i;

}

// Suffixes are already sorted by first len characters

// Now sort suffixes by first len \* 2 characters

**int**[] cnt = IntStream.*range*(0, n).toArray();

**int**[] s = sa.clone();

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

// s[i] - order of suffixes sorted by first len characters

// (s[i] - len) - order of suffixes sorted only by second len characters

**int** s1 = s[i] - len;

// sort only suffixes of length > len, others are already sorted

**if** (s1 >= 0)

sa[cnt[classes[s1]]++] = s1;

}

}

**return** sa;

}

// sort rotations of S in O(n\*log(n))

**public** **static** **int**[] rotationArray(CharSequence S) {

**int** n = S.length();

**int**[] sa = IntStream.*range*(0, n).mapToObj(Integer::*valueOf*).

sorted((a, b) -> Character.*compare*(S.charAt(a), S.charAt(b))).mapToInt(Integer::intValue).toArray();

**int**[] classes = S.chars().toArray();

**for** (**int** len = 1; len < n; len \*= 2) {

**int**[] c = classes.clone();

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

classes[sa[i]] = i > 0 && c[sa[i - 1]] == c[sa[i]] && c[(sa[i - 1] + len / 2) % n] == c[(sa[i] + len / 2) % n] ? classes[sa[i - 1]] : i;

**int**[] cnt = IntStream.*range*(0, n).toArray();

**int**[] s = sa.clone();

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

**int** s1 = (s[i] - len + n) % n;

sa[cnt[classes[s1]]++] = s1;

}

}

**return** sa;

}

// longest common prefixes array in O(n)

**public** **static** **int**[] lcp(**int**[] sa, CharSequence s) {

**int** n = sa.length;

**int**[] rank = **new** **int**[n];

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

rank[sa[i]] = i;

**int**[] lcp = **new** **int**[n - 1];

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

**if** (rank[i] < n - 1) {

**for** (**int** j = sa[rank[i] + 1]; Math.*max*(i, j) + h < s.length() && s.charAt(i + h) == s.charAt(j + h); ++h)

;

lcp[rank[i]] = h;

**if** (h > 0)

--h;

}

}

**return** lcp;

}

// Usage example

**public** **static** **void** main(String[] args) {

String s1 = "abcab";

**int**[] sa1 = *suffixArray*(s1);

// print suffixes in lexicographic order

**for** (**int** p : sa1)

System.***out***.println(s1.substring(p));

System.***out***.println("lcp = " + Arrays.*toString*(*lcp*(sa1, s1)));

// random test

Random rnd = **new** Random(1);

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

**int** n = rnd.nextInt(100) + 1;

StringBuilder s = rnd.ints(n, 0, 10).collect(StringBuilder::**new**, (sb, i) -> sb.append((**char**) ('\1' + i)), StringBuilder::append);

**int**[] sa = *suffixArray*(s);

**int**[] ra = *rotationArray*(s.toString() + '\0');

**int**[] lcp = *lcp*(sa, s);

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

String a = s.substring(sa[i]);

String b = s.substring(sa[i + 1]);

**if** (a.compareTo(b) >= 0

|| !a.substring(0, lcp[i]).equals(b.substring(0, lcp[i]))

|| (a + " ").charAt(lcp[i]) == (b + " ").charAt(lcp[i])

|| sa[i] != ra[i + 1])

**throw** **new** RuntimeException();

}

}

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

}

}

# Longest Common Subsequence

import java.util.Arrays;

public class StringDistances {

// https://en.wikipedia.org/wiki/Longest\_common\_subsequence\_problem

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;

}

// https://en.wikipedia.org/wiki/Levenshtein\_distance

public static int getLevensteinDistance(String a, String b) {

int m = a.length();

int n = b.length();

int[][] len = new int[m + 1][n + 1];

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

len[i][0] = i;

}

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

len[0][j] = j;

}

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

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

if (a.charAt(i) == b.charAt(j)) {

len[i + 1][j + 1] = len[i][j];

} else {

len[i + 1][j + 1] = 1 + Math.min(len[i][j], Math.min(len[i + 1][j], len[i][j + 1]));

}

}

}

return len[m][n];

}

// Usage example

public static void main(String[] args) {

int[] x = {1, 5, 4, 2, 3, 7, 6};

int[] y = {2, 7, 1, 3, 5, 4, 6};

int[] lcs = getLCS(x, y);

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

String a = "abc";

String b = "ac";

System.out.println(getLevensteinDistance(a, b));

}

}

# Ternary Search

import java.util.Random;

import java.util.function.DoubleUnaryOperator;

import java.util.function.IntUnaryOperator;

// https://en.wikipedia.org/wiki/Ternary\_search

public class TernarySearch {

// finds maximum of strictly increasing and then strictly decreasing function

public static double ternarySearch(DoubleUnaryOperator f, double lo, double hi) {

for (int step = 0; step < 1000; step++) {

double m1 = lo + (hi - lo) / 3;

double m2 = hi - (hi - lo) / 3;

if (f.applyAsDouble(m1) < f.applyAsDouble(m2))

lo = m1;

else

hi = m2;

}

return (lo + hi) / 2;

}

// finds maximum of strictly increasing and then strictly decreasing function

public static int ternarySearch(IntUnaryOperator f, int fromInclusive, int toInclusive) {

int lo = fromInclusive;

int hi = toInclusive;

while (hi > lo + 2) {

int m1 = lo + (hi - lo) / 3;

int m2 = hi - (hi - lo) / 3;

if (f.applyAsInt(m1) < f.applyAsInt(m2))

lo = m1;

else

hi = m2;

}

int res = lo;

for (int i = lo + 1; i <= hi; i++)

if (f.applyAsInt(res) < f.applyAsInt(i))

res = i;

return res;

}

public static int ternarySearch2(IntUnaryOperator f, int fromInclusive, int toInclusive) {

int lo = fromInclusive - 1;

int hi = toInclusive;

while (hi - lo > 1) {

int mid = (lo + hi) >>> 1;

if (f.applyAsInt(mid) < f.applyAsInt(mid + 1)) {

lo = mid;

} else {

hi = mid;

}

}

return hi;

}

// random tests

public static void main(String[] args) {

System.out.println(ternarySearch((DoubleUnaryOperator) x -> -(x - 2) \* (x - 2), -10, 10));

Random rnd = new Random(1);

for (int step = 0; step < 1000; step++) {

int n = rnd.nextInt(20) + 1;

int p = rnd.nextInt(n);

int[] a = new int[n];

final int range = 10;

a[p] = rnd.nextInt(range);

for (int i = p - 1; i >= 0; i--)

a[i] = a[i + 1] - rnd.nextInt(range) - 1;

for (int i = p + 1; i < n; i++)

a[i] = a[i - 1] - rnd.nextInt(range) - 1;

int res1 = ternarySearch((IntUnaryOperator) i -> a[i], 0, a.length - 1);

int res2 = ternarySearch2(i -> a[i], 0, a.length - 1);

if (p != res1 || p != res2)

throw new RuntimeException();

}

}

}