**Contents**

[Input/ output 1](#_Toc399515156)

[Math 3](#_Toc399515157)

[**Linear Diophantine Equations** 3](#_Toc399515158)

[Arrays 3](#_Toc399515159)

[**LCS** 3](#_Toc399515160)

[**Longest increasing subsequence in O(N \* logN)** 4](#_Toc399515161)

[Graph 5](#_Toc399515162)

[**Minimum spanning tree. Prim's algorithm in O(E \* logV)** 6](#_Toc399515163)

[**Dijstra** 7](#_Toc399515164)

[**Tarjan’s** 8](#_Toc399515165)

[**Biconnected components, bridges and cut points** 9](#_Toc399515166)

[**Greedy graph coloring in O(E \* logV)** 11](#_Toc399515167)

[**Shortest paths. Floyd–Warshall algorithm in O(V^3)** 12](#_Toc399515168)

[**Shortest paths. Bellman–Ford algorithm in O(V\*E). Negative cycle.** 13](#_Toc399515169)

[**Maximum matching for bipartite graph. Kuhn's algorithm in O(E\*V)** 15](#_Toc399515170)

[Data structure 16](#_Toc399515171)

[**Segment tree** 16](#_Toc399515172)

[**Union set** 18](#_Toc399515173)

[**Prime numbers, sieve of Eratosthenes, Euler's totient function** 18](#_Toc399515174)

[**Chặt nhị phân** 20](#_Toc399515175)

[**Universal sqrt-decomposition for queries** 20](#_Toc399515176)

# Input/ output

import java.io.BufferedReader;

import java.io.IOException;

import java.io.InputStream;

import java.io.InputStreamReader;

import java.io.OutputStream;

import java.io.PrintWriter;

import java.util.\*;

public class Main {

public static void main(String[] args) {

InputStream inputStream = System.in;

OutputStream outputStream = System.out;

InputReader in = new InputReader(inputStream);

PrintWriter out = new PrintWriter(outputStream);

Task solver = new Task();

solver.solve(in, out);

out.close();

}

}

class Task {

public void solve(InputReader in, PrintWriter out) {

// your code here

}

}

class InputReader {

public BufferedReader reader;

public StringTokenizer tokenizer;

public InputReader(InputStream stream) {

reader = new BufferedReader(new InputStreamReader(stream), 32768);

tokenizer = null;

}

public String next() {

while (tokenizer == null || !tokenizer.hasMoreTokens()) {

try {

tokenizer = new StringTokenizer(reader.readLine());

} catch (IOException e) {

throw new RuntimeException(e);

}

}

return tokenizer.nextToken();

}

public int nextInt() {

return Integer.parseInt(next());

}

public long nextLong() {

return Long.parseLong(next());

}

public double nextDouble() {

return Double.parseDouble(next());

}

}

# Math

## **Linear Diophantine Equations**

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

# Arrays

## **LCS**

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

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

**public class**Lis2 {  
  
  **public static int**[] lis(**int**[] a) {  
    **int**n = a.length;  
    **int**[] tail = **new int**[n];  
    **int**[] prev = **new int**[n];  
  
    **int**len = 0;  
    **for**(**int**i = 0; i < n; i++) {  
      **int**pos = lower\_bound(a, tail, len, a[i]);  
      **if**(pos == len) {  
        ++len;  
      }  
      prev[i] = pos > 0 ? tail[pos - 1] : -1;  
      tail[pos] = i;  
    }  
  
    **int**[] res = **new int**[len];  
    **for**(**int**i = tail[len - 1]; i >= 0; i = prev[i]) {  
      res[--len] = a[i];  
    }  
    **return**res;  
  }  
  
  **static int**lower\_bound(**int**[] a, **int**[] tail, **int**len, **int**key) {  
    **int**lo = -1;  
    **int**hi = len;  
    **while**(hi - lo > 1) {  
      **int**mid = (lo + hi) >>> 1;  
      **if**(a[tail[mid]] < key) {  
        lo = mid;  
      } **else**{  
        hi = mid;  
      }  
    }  
    **return**hi;  
  }  
  
  // random test  
  **public static void**main(String[] args) {  
    Random rnd = **new**Random(1);  
    **for**(**int**step = 0; step < 10\_000; step++) {  
      **int**n = rnd.nextInt(10) + 1;  
      **int**[] a = **new int**[n];  
      **for**(**int**i = 0; i < n; i++)  
        a[i] = rnd.nextInt(10);  
      **int**[] lis = lis(a);  
      checkLis(a, lis);  
    }  
  }  
  
  **static void**checkLis(**int**[] a, **int**[] lis) {  
    **int**n = a.length;  
    **boolean**found = **false**;  
    m1:  
    **for**(**int**mask = 0; mask < 1 << n; mask++) {  
      **int**len = Integer.bitCount(mask);  
      **if**(len < lis.length)  
        **continue**;  
      **for**(**int**i = 0, prev = Integer.MIN\_VALUE; i < n; i++)  
        **if**((mask & (1 << i)) != 0) {  
          **if**(prev >= a[i])  
            **continue**m1;  
          prev = a[i];  
        }  
      **if**(len > lis.length)  
        **throw new**RuntimeException();  
      **boolean**ok = **true**;  
      **for**(**int**i = 0, j = 0; i < n; i++)  
        **if**((mask & (1 << i)) != 0)  
          ok &= a[i] == lis[j++];  
      found |= ok;  
    }  
    **if**(!found)  
      **throw new**RuntimeException();  
  }  
}

# Graph

## **Minimum spanning tree. Prim's algorithm in O(E \* logV)**

**public class**PrimHeap {  
  
  **public static long**mst(List<Edge>[] edges, **int**[] pred) {  
    **int**n = edges.length;  
    Arrays.fill(pred, -1);  
    **boolean**[] used = **new boolean**[n];  
    **int**[] prio = **new int**[n];  
    Arrays.fill(prio, Integer.MAX\_VALUE);  
    prio[0] = 0;  
    PriorityQueue<Long> q = **new**PriorityQueue<>();  
    q.add(0L);  
    **long**res = 0;  
  
    **while**(!q.isEmpty()) {  
      **long**cur = q.poll();  
      **int**u = (**int**) cur;  
      **if**(used[u])  
        **continue**;  
      used[u] = **true**;  
      res += cur >>> 32;  
      **for**(Edge e : edges[u]) {  
        **int**v = e.t;  
        **if**(!used[v] && prio[v] > e.cost) {  
          prio[v] = e.cost;  
          pred[v] = u;  
          q.add(((**long**) prio[v] << 32) + v);  
        }  
      }  
    }  
    **return**res;  
  }  
  
  **static class**Edge {  
    **int**t, cost;  
  
    **public**Edge(**int**t, **int**cost) {  
      **this**.t = t;  
      **this**.cost = cost;  
    }  
  }  
  
  // Usage example  
  **public static void**main(String[] args) {  
    **int**[][] cost = { { 0, 1, 2 }, { 1, 0, 3 }, { 2, 3, 0 } };  
    **int**n = cost.length;  
    List<Edge>[] edges = **new**List[n];  
    **for**(**int**i = 0; i < n; i++) {  
      edges[i] = **new**ArrayList<>();  
      **for**(**int**j = 0; j < n; j++) {  
        **if**(cost[i][j] != 0) {  
          edges[i].add(**new**Edge(j, cost[i][j]));  
        }  
      }  
    }  
    **int**[] pred = **new int**[n];  
    System.out.println(mst(edges, pred));  
  }  
}

## **Dijstra**

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

## **Tarjan’s**

**public class**SCCTarjan {  
  
  List<Integer>[] graph;  
  **boolean**[] visited;  
  Stack<Integer> stack;  
  **int**time;  
  **int**[] lowlink;  
  List<List<Integer>> components;  
  
  **public**List<List<Integer>> scc(List<Integer>[] graph) {  
    **int**n = graph.length;  
    **this**.graph = graph;  
    visited = **new boolean**[n];  
    stack = **new**Stack<>();  
    time = 0;  
    lowlink = **new int**[n];  
    components = **new**ArrayList<>();  
  
    **for**(**int**u = 0; u < n; u++)  
      **if**(!visited[u])  
        dfs(u);  
  
    **return**components;  
  }  
  
  **void**dfs(**int**u) {  
    lowlink[u] = time++;  
    visited[u] = **true**;  
    stack.add(u);  
    **boolean**isComponentRoot = **true**;  
  
    **for**(**int**v : graph[u]) {  
      **if**(!visited[v])  
        dfs(v);  
      **if**(lowlink[u] > lowlink[v]) {  
        lowlink[u] = lowlink[v];  
        isComponentRoot = **false**;  
      }  
    }  
  
    **if**(isComponentRoot) {  
      List<Integer> component = **new**ArrayList<>();  
      **while**(**true**) {  
        **int**x = stack.pop();  
        component.add(x);  
        lowlink[x] = Integer.MAX\_VALUE;  
        **if**(x == u)  
          **break**;  
      }  
      components.add(component);  
    }  
  }  
  
  // Usage example  
  **public static void**main(String[] args) {  
    List<Integer>[] g = **new**List[3];  
    **for**(**int**i = 0; i < g.length; i++)  
      g[i] = **new**ArrayList<>();  
  
    g[2].add(0);  
    g[2].add(1);  
    g[0].add(1);  
    g[1].add(0);  
  
    List<List<Integer>> components = **new**SCCTarjan().scc(g);  
    System.out.println(components);  
  }  
}

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

**public class**BiconnectedComponents {  
  
  List<Integer>[] graph;  
  **boolean**[] visited;  
  Stack<Integer> stack;  
  **int**time;  
  **int**[] tin;  
  **int**[] lowlink;  
  List<List<Integer>> components;  
  List<Integer> cutPoints;  
  List<String> bridges;  
  
  **public**List<List<Integer>> biconnectedComponents(List<Integer>[] graph) {  
    **int**n = graph.length;  
    **this**.graph = graph;  
    visited = **new boolean**[n];  
    stack = **new**Stack<>();  
    time = 0;  
    tin = **new int**[n];  
    lowlink = **new int**[n];  
    components = **new**ArrayList<>();  
    cutPoints = **new**ArrayList<>();  
    bridges = **new**ArrayList<>();  
  
    **for**(**int**u = 0; u < n; u++)  
      **if**(!visited[u])  
        dfs(u, -1);  
  
    **return**components;  
  }  
  
  **void**dfs(**int**u, **int**p) {  
    visited[u] = **true**;  
    lowlink[u] = tin[u] = time++;  
    stack.add(u);  
    **int**children = 0;  
    **boolean**cutPoint = **false**;  
    **for**(**int**v : graph[u]) {  
      **if**(v == p)  
        **continue**;  
      **if**(visited[v]) {  
        // lowlink[u] = Math.min(lowlink[u], lowlink[v]);  
        lowlink[u] = Math.min(lowlink[u], tin[v]);  
      } **else**{  
        dfs(v, u);  
        lowlink[u] = Math.min(lowlink[u], lowlink[v]);  
        cutPoint |= lowlink[v] >= tin[u];  
        //if (lowlink[v] == tin[v])  
        **if**(lowlink[v] > tin[u])  
          bridges.add("(" + u + "," + v + ")");  
        ++children;  
      }  
    }  
    **if**(p == -1)  
      cutPoint = children >= 2;  
    **if**(cutPoint)  
      cutPoints.add(u);  
    **if**(lowlink[u] == tin[u]) {  
      List<Integer> component = **new**ArrayList<>();  
      **while**(**true**) {  
        **int**x = stack.pop();  
        component.add(x);  
        **if**(x == u)  
          **break**;  
      }  
      components.add(component);  
    }  
  }  
  
  // Usage example  
  **public static void**main(String[] args) {  
    **int**n = 6;  
    List<Integer>[] graph = **new**List[n];  
    **for**(**int**i = 0; i < n; i++) {  
      graph[i] = **new**ArrayList<>();  
    }  
    **int**[][] edges = {{0, 1}, {1, 2}, {0, 2}, {2, 3}, {3, 4}, {4, 5}, {3, 5}};  
    **for**(**int**[] edge : edges) {  
      graph[edge[0]].add(edge[1]);  
      graph[edge[1]].add(edge[0]);  
    }  
  
    BiconnectedComponents bc = **new**BiconnectedComponents();  
    List<List<Integer>> components = bc.biconnectedComponents(graph);  
  
    System.out.println("biconnected components:" + components);  
    System.out.println("cutPoints: " + bc.cutPoints);  
    System.out.println("bridges:" + bc.bridges);  
  }  
}

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

**public class**GraphColoringGreedy {  
  
  // similar to DSatur coloring  
  **public static int**[] color(List<Integer>[] graph) {  
    **int**n = graph.length;  
    BitSet[] used = **new**BitSet[n];  
    **int**[] colors = **new int**[n];  
    PriorityQueue<Long> q = **new**PriorityQueue<>(n);  
    **for**(**int**i = 0; i < n; i++) {  
      used[i] = **new**BitSet();  
      colors[i] = -1;  
      q.add((**long**) i);  
    }  
    **for**(**int**i = 0; i < n; i++) {  
      **int**bestu;  
      **while**(**true**) {  
        bestu = q.remove().intValue();  
        **if**(colors[bestu] == -1)  
          **break**;  
      }  
      **int**c = used[bestu].nextClearBit(0);  
      colors[bestu] = c;  
      **for**(**int**v : graph[bestu]) {  
        **if**(!used[v].get(c)) {  
          used[v].set(c);  
          **if**(colors[v] == -1)  
            q.add(v - ((**long**) used[v].cardinality() << 32));  
        }  
      }  
    }  
    **return**colors;  
  }  
  
  // Usage example  
  **public static void**main(String[] args) {  
    **int**n = 5;  
    List<Integer>[] g = **new**List[n];  
    **for**(**int**i = 0; i < n; i++) {  
      g[i] = **new**ArrayList<>();  
    }  
    **for**(**int**i = 0; i < n; i++) {  
      **for**(**int**j = 0; j < n; j++) {  
        g[i].add((i + 1) % n);  
        g[(i + 1) % n].add(i);  
      }  
    }  
    System.out.println(Arrays.toString(color(g)));  
  }  
}

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

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

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

**public class**BellmanFord {  
  
  **static final int**INF = Integer.MAX\_VALUE / 2;  
  
  **public static class**Edge {  
    **int**v, cost;  
  
    **public**Edge(**int**v, **int**cost) {  
      **this**.v = v;  
      **this**.cost = cost;  
    }  
  }  
  
  **public static boolean**bellmanFord(List<Edge>[] graph, **int**s, **int**[] dist, **int**[] pred) {  
    Arrays.fill(pred, -1);  
    Arrays.fill(dist, INF);  
    dist[s] = 0;  
    **int**n = graph.length;  
    **boolean**updated = **false**;  
    **for**(**int**step = 0; step < n; step++) {  
      updated = **false**;  
      **for**(**int**u = 0; u < n; u++) {  
        **if**(dist[u] == INF) **continue**;  
        **for**(Edge e : graph[u]) {  
          **if**(dist[e.v] > dist[u] + e.cost) {  
            dist[e.v] = dist[u] + e.cost;  
            dist[e.v] = Math.max(dist[e.v], -INF);  
            pred[e.v] = u;  
            updated = **true**;  
          }  
        }  
      }  
      **if**(!updated)  
        **break**;  
    }  
    // if updated is true then a negative cycle exists  
    **return**updated == **false**;  
  }  
  
  **public static int**[] findNegativeCycle(List<Edge>[] graph) {  
    **int**n = graph.length;  
    **int**[] pred = **new int**[n];  
    Arrays.fill(pred, -1);  
    **int**[] dist = **new int**[n];  
    **int**last = -1;  
    **for**(**int**step = 0; step < n; step++) {  
      last = -1;  
      **for**(**int**u = 0; u < n; u++) {  
        **if**(dist[u] == INF) **continue**;  
        **for**(Edge e : graph[u]) {  
          **if**(dist[e.v] > dist[u] + e.cost) {  
            dist[e.v] = Math.max(dist[u] + e.cost, -INF);  
            dist[e.v] = Math.max(dist[e.v], -INF);  
            pred[e.v] = u;  
            last = e.v;  
          }  
        }  
      }  
      **if**(last == -1)  
        **return null**;  
    }  
    **for**(**int**i = 0; i < n; i++) {  
      last = pred[last];  
    }  
    **int**[] p = **new int**[n];  
    **int**cnt = 0;  
    **for**(**int**u = last; u != last || cnt == 0; u = pred[u]) {  
      p[cnt++] = u;  
    }  
    **int**[] cycle = **new int**[cnt];  
    **for**(**int**i = 0; i < cycle.length; i++) {  
      cycle[i] = p[--cnt];  
    }  
    **return**cycle;  
  }  
  
  // Usage example  
  **public static void**main(String[] args) {  
    List<Edge>[] graph = **new**List[4];  
    **for**(**int**i = 0; i < graph.length; i++) {  
      graph[i] = **new**ArrayList<>();  
    }  
    graph[0].add(**new**Edge(1, 1));  
    graph[1].add(**new**Edge(0, 1));  
    graph[1].add(**new**Edge(2, 1));  
    graph[2].add(**new**Edge(3, -10));  
    graph[3].add(**new**Edge(1, 1));  
    **int**[] cycle = findNegativeCycle(graph);  
    System.out.println(Arrays.toString(cycle));  
  }  
}

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

**public class**MaxMatching2 {  
  
  **public static int**maxMatching(List<Integer>[] graph, **int**n2) {  
    **int**n1 = graph.length;  
    **int**[] matching = **new int**[n2];  
    Arrays.fill(matching, -1);  
    **int**matches = 0;  
    **for**(**int**u = 0; u < n1; u++) {  
      **if**(findPath(graph, u, matching, **new boolean**[n1]))  
        ++matches;  
    }  
    **return**matches;  
  }  
  
  **static boolean**findPath(List<Integer>[] graph, **int**u1, **int**[] matching, **boolean**[] vis) {  
    vis[u1] = **true**;  
    **for**(**int**v : graph[u1]) {  
      **int**u2 = matching[v];  
      **if**(u2 == -1 || !vis[u2] && findPath(graph, u2, matching, vis)) {  
        matching[v] = u1;  
        **return true**;  
      }  
    }  
    **return false**;  
  }  
  
  // random tests  
  **public static void**main(String[] args) {  
    Random rnd = **new**Random(1);  
    **for**(**int**step = 0; step < 1000; step++) {  
      **int**n1 = rnd.nextInt(20) + 1;  
      **int**n2 = rnd.nextInt(20) + 1;  
      List<Integer>[] g = **new**List[n1];  
      **for**(**int**i = 0; i < n1; i++)  
        g[i] = **new**ArrayList<>();  
      **for**(**int**i = 0; i < n1; i++)  
        **for**(**int**j = 0; j < n2; j++)  
          g[i].add(j);  
      **int**res1 = maxMatching(g, n2);  
      **int**res2 = slowMinVertexCover(g, n2);  
      **if**(res1 != res2)  
        **throw new**RuntimeException();  
    }  
  }  
  
  **static int**slowMinVertexCover(List<Integer>[] g, **int**n2) {  
    **int**n1 = g.length;  
    **int**[] mask = **new int**[n1];  
    **for**(**int**i = 0; i < n1; i++)  
      **for**(**int**j : g[i])  
        mask[i] |= 1 << j;  
    **int**res = n2;  
    **for**(**int**m = 0; m < 1 << n2; m++) {  
      **int**cur = Integer.bitCount(m);  
      **for**(**int**i = 0; i < n1; i++)  
        **if**((mask[i] & m) != mask[i])  
          ++cur;  
      res = Math.min(res, cur);  
    }  
    **return**res;  
  }  
}

# Data structure

## **Segment tree**

/\*\*  
\* Build and init tree  
\*/  
void build\_tree(int node, int a, int b) {  
    if(a > b) return; // Out of range  
    if(a == b) { // Leaf node  
        tree[node] = arr[a]; // Init value  
        return;  
    }  
    build\_tree(node\*2, a, (a+b)/2); // Init left child  
    build\_tree(node\*2+1, 1+(a+b)/2, b); // Init right child  
    tree[node] = max(tree[node\*2], tree[node\*2+1]); // Init root value  
}  
  
/\*\*  
\* Increment elements within range [i, j] with value value  
\*/  
void update\_tree(int node, int a, int b, int i, int j, int value) {  
    if(lazy[node] != 0) { // This node needs to be updated  
        tree[node] += lazy[node]; // Update it  
  
        if(a != b) {  
            lazy[node\*2] += lazy[node]; // Mark child as lazy  
            lazy[node\*2+1] += lazy[node]; // Mark child as lazy  
        }  
  
        lazy[node] = 0; // Reset it  
    }  
    if(a > b || a > j || b < i) // Current segment is not within range [i, j]  
        return;  
    if(a >= i && b <= j) { // Segment is fully within range  
        tree[node] += value;  
  
        if(a != b) { // Not leaf node  
            lazy[node\*2] += value;  
            lazy[node\*2+1] += value;  
        }  
  
        return;  
    }  
  
    update\_tree(node\*2, a, (a+b)/2, i, j, value); // Updating left child  
    update\_tree(1+node\*2, 1+(a+b)/2, b, i, j, value); // Updating right child  
  
    tree[node] = max(tree[node\*2], tree[node\*2+1]); // Updating root with max value  
}  
  
/\*\*  
\* Query tree to get max element value within range [i, j]  
\*/  
int query\_tree(int node, int a, int b, int i, int j) {  
    if(a > b || a > j || b < i) return -inf; // Out of range  
  
    if(lazy[node] != 0) { // This node needs to be updated  
        tree[node] += lazy[node]; // Update it  
  
        if(a != b) {  
            lazy[node\*2] += lazy[node]; // Mark child as lazy  
            lazy[node\*2+1] += lazy[node]; // Mark child as lazy  
        }  
  
        lazy[node] = 0; // Reset it  
    }  
  
    if(a >= i && b <= j) // Current segment is totally within range [i, j]  
        return tree[node];  
  
    int q1 = query\_tree(node\*2, a, (a+b)/2, i, j); // Query left child  
    int q2 = query\_tree(1+node\*2, 1+(a+b)/2, b, i, j); // Query right child  
  
    int res = max(q1, q2); // Return final result  
    return res;  
}  
  
int main() {  
    for(int i = 0; i < N; i++) arr[i] = 1;  
  
    build\_tree(1, 0, N-1);  
  
    memset(lazy, 0, sizeof lazy);  
  
    update\_tree(1, 0, N-1, 0, 6, 5); // Increment range [0, 6] by 5  
    update\_tree(1, 0, N-1, 7, 10, 12); // Incremenet range [7, 10] by 12  
    update\_tree(1, 0, N-1, 10, N-1, 100); // Increment range [10, N-1] by 100  
  
    cout << query\_tree(1, 0, N-1, 0, N-1) << endl; // Get max element in range [0, N-1]  
}

## **Union set**

**public class**DisjointSets {  
  
  **public static int**[] createSets(**int**size) {  
    **int**[] p = **new int**[size];  
    **for**(**int**i = 0; i < size; i++)  
      p[i] = i;  
    **return**p;  
  }  
  
  **public static int**root(**int**[] p, **int**x) {  
    **return**x == p[x] ? x : (p[x] = root(p, p[x]));  
  }  
  
  **public static void**unite(**int**[] p, **int**a, **int**b) {  
    a = root(p, a);  
    b = root(p, b);  
    **if**(a != b)  
      p[a] = b;  
  }  
  
  // Usage example  
  **public static void**main(String[] args) {  
    **int**[] p = createSets(10);  
    System.out.println(**false**== (root(p, 0) == root(p, 9)));  
    unite(p, 0, 9);  
    System.out.println(**true**== (root(p, 0) == root(p, 9)));  
  }  
}

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

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

## **Universal sqrt-decomposition for queries**

**public class**DistinctQuerySqrt {  
  
  **static int**solveSlow(**int**[] cnt, **int**[] a, **int**haveLeft, **int**haveRight, **int**needLeft, **int**needRight) {  
    **int**res = 0;  
    **for**(**int**i = haveRight + 1; i <= needRight; ++i) {  
      res += cnt[a[i]] == 0 ? 1 : 0;  
      ++cnt[a[i]];  
    }  
    **for**(**int**i = haveLeft; i < needLeft; ++i) {  
      res += cnt[a[i]] == 1 ? -1 : 0;  
      --cnt[a[i]];  
    }  
    **for**(**int**i = haveRight + 1; i <= needRight; ++i) {  
      --cnt[a[i]];  
    }  
    **for**(**int**i = haveLeft; i < needLeft; ++i) {  
      ++cnt[a[i]];  
    }  
    **return**res;  
  }  
  
  **static int**[] solve(**int**[] a, **int**[][] queries) {  
    **int**n = a.length;  
    Map<Integer, Integer> m = **new**HashMap<>();  
    **int**id = 0;  
    **for**(**int**i = 0; i < n; i++) {  
      **if**(!m.containsKey(a[i]))  
        m.put(a[i], id++);  
      a[i] = m.get(a[i]);  
    }  
    **int**blockSize = (**int**) Math.sqrt(n);  
    **int**blocks = (n + blockSize - 1) / blockSize;  
    **int**[][] head = **new int**[blocks][blocks];  
    **for**(**int**i = 0; i < blocks; i++)  
      Arrays.fill(head[i], -1);  
    **int**q = queries.length;  
    **int**[] left = **new int**[q];  
    **int**[] right = **new int**[q];  
    **int**[] next = **new int**[q];  
    **for**(**int**i = 0; i < q; i++) {  
      left[i] = queries[i][0];  
      right[i] = queries[i][1];  
      **int**lb = left[i] / blockSize;  
      **int**rb = right[i] / blockSize;  
      next[i] = head[lb][rb];  
      head[lb][rb] = i;  
    }  
    **int**[] res = **new int**[q];  
    **for**(**int**leftBlock = 0; leftBlock < blocks; leftBlock++) {  
      **int**[] cnt = **new int**[id];  
      **int**curAnswer = 0;  
      **for**(**int**rightBlock = leftBlock; rightBlock < blocks; rightBlock++) {  
        **for**(**int**i = head[leftBlock][rightBlock]; i >= 0; i = next[i]) {  
          res[i] = curAnswer  
              + solveSlow(cnt, a, leftBlock \* blockSize, rightBlock \* blockSize - 1, left[i], right[i]);  
        }  
        **if**(rightBlock + 1 < blocks) {  
          **for**(**int**i = 0; i < blockSize; i++) {  
            **int**cur = a[rightBlock \* blockSize + i];  
            curAnswer += cnt[cur] == 0 ? 1 : 0;  
            ++cnt[cur];  
          }  
        }  
      }  
    }  
    **return**res;  
  }  
  
  **public static void**main(String[] args) {  
    **int**[] a = {1, 2, 3, 4};  
    **int**[][] queries = {{3, 3}};  
    **int**[] res = solve(a, queries);  
    System.out.println(Arrays.toString(res));  
  }  
}

## **Enumeration of permutations**

**public class**Permutations {  
  
  **public static boolean**nextPermutation(**int**[] p) {  
    **for**(**int**a = p.length - 2; a >= 0; --a)  
      **if**(p[a] < p[a + 1])  
        **for**(**int**b = p.length - 1; ; --b)  
          **if**(p[b] > p[a]) {  
            **int**t = p[a];  
            p[a] = p[b];  
            p[b] = t;  
            **for**(++a, b = p.length - 1; a < b; ++a, --b) {  
              t = p[a];  
              p[a] = p[b];  
              p[b] = t;  
            }  
            **return true**;  
          }  
    **return false**;  
  }  
  
  **public static int**[] permutationByNumber(**int**n, **long**number) {  
    **long**[] fact = **new long**[n];  
    fact[0] = 1;  
    **for**(**int**i = 1; i < n; i++) {  
      fact[i] = i \* fact[i - 1];  
    }  
    **int**[] p = **new int**[n];  
    **int**[] free = **new int**[n];  
    **for**(**int**i = 0; i < n; i++) {  
      free[i] = i;  
    }  
    **for**(**int**i = 0; i < n; i++) {  
      **int**pos = (**int**) (number / fact[n - 1 - i]);  
      p[i] = free[pos];  
      System.arraycopy(free, pos + 1, free, pos, n - 1 - pos);  
      number %= fact[n - 1 - i];  
    }  
    **return**p;  
  }  
  
  **public static long**numberByPermutation(**int**[] p) {  
    **int**n = p.length;  
    **long**[] fact = **new long**[n];  
    fact[0] = 1;  
    **for**(**int**i = 1; i < n; i++) {  
      fact[i] = i \* fact[i - 1];  
    }  
    **long**res = 0;  
    **for**(**int**i = 0; i < n; i++) {  
      **int**a = p[i];  
      **for**(**int**j = 0; j < i; j++) {  
        **if**(p[j] < p[i]) {  
          --a;  
        }  
      }  
      res += a \* fact[n - 1 - i];  
    }  
    **return**res;  
  }  
  
  **public static void**generatePermutations(**int**[] p, **int**depth) {  
    **int**n = p.length;  
    **if**(depth == n) {  
      System.out.println(Arrays.toString(p));  
      **return**;  
    }  
    **for**(**int**i = 0; i < n; i++) {  
      **if**(p[i] == 0) {  
        p[i] = depth;  
        generatePermutations(p, depth + 1);  
        p[i] = 0;  
      }  
    }  
  }  
  
  **public static long**nextPermutation(**long**x) {  
    **long**s = x & -x;  
    **long**r = x + s;  
    **long**ones = x ^ r;  
    ones = (ones >> 2) / s;  
    **return**r | ones;  
  }  
  
  **public static**List<List<Integer>> decomposeIntoCycles(**int**[] p) {  
    **int**n = p.length;  
    **boolean**[] vis = **new boolean**[n];  
    List<List<Integer>> res = **new**ArrayList<>();  
    **for**(**int**i = 0; i < n; i++) {  
      **if**(vis[i])  
        **continue**;  
      **int**j = i;  
      List<Integer> cur = **new**ArrayList<>();  
      **do**{  
        cur.add(j);  
        vis[j] = **true**;  
        j = p[j];  
      } **while**(j != i);  
      res.add(cur);  
    }  
    **return**res;  
  }  
  
  // Usage example  
  **public static void**main(String[] args) {  
    // print all permutations method 1  
    generatePermutations(**new int**[2], 1);  
  
    // print all permutations method 2  
    **int**[] p = {0, 1, 2};  
    **int**cnt = 0;  
    **do**{  
      System.out.println(Arrays.toString(p));  
      **if**(!Arrays.equals(p, permutationByNumber(p.length, numberByPermutation(p))) ||  
          cnt != numberByPermutation(permutationByNumber(p.length, cnt)))  
        **throw new**RuntimeException();  
      ++cnt;  
    } **while**(nextPermutation(p));  
  
    System.out.println(5 == numberByPermutation(p));  
    System.out.println(Arrays.equals(**new int**[]{1, 0, 2}, permutationByNumber(3, 2)));  
  
    System.out.println(0b1101 == nextPermutation(0b1011));  
    System.out.println(decomposeIntoCycles(**new int**[]{0, 2, 1, 3}));  
  }  
}