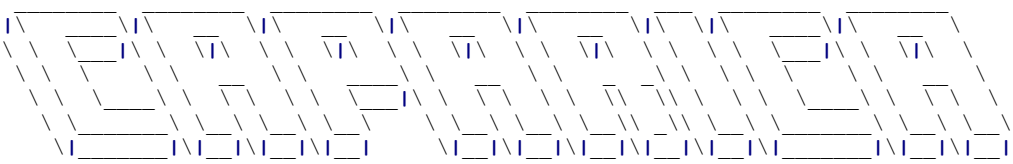

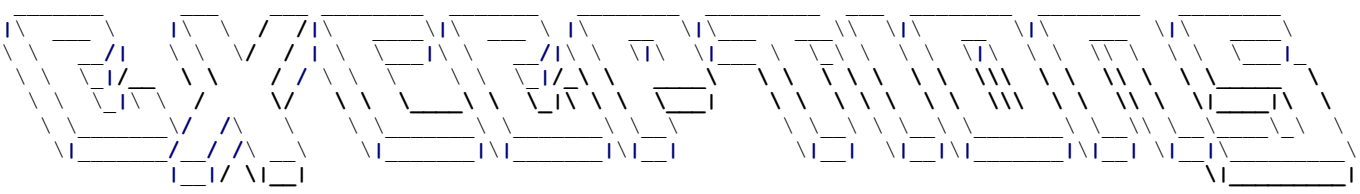

```
##########################################################################################################################


private static String INPUT = null;

private static Reader inputStream = INPUT == null ? new InputStreamReader(System.in) : new StringReader(INPUT);
private static BufferedReader br = new BufferedReader(inputStream);

private static String readLine() {
    try {
        return br.readLine();
    } catch (IOException e) {
        return null;
    }
}

private static BufferedOutputStream out = new BufferedOutputStream(System.out);
// usage: out.write(str.getBytes());

Caminho mais curto --- Com nó de partida --- Sem pesos --- Breadth-first search
                   |                     \-- Com pesos --- Não negativos --- Dijkstra
                   |                                   \-- Negativos --- Bellman-Ford
                   \-- Entre todos os nós -- |V| < 500 --- Floyd-Warshall

Caminho mais comprido --- pesos dos arcos para o simétrico --- Bellman-Ford

Encontrar ciclos --- grafo com direção --- ordenação topológica
                \-- grafo sem direção --- dijoint sets

Min Spanning Tree --- Kruscal
                  \-- Prim

Dependências --- ordenação topologica

Fluxo Máximo       --- Edmonds-Karp
n-partide matching --/



/**
### Linux Problems ###

Problem: The default keyboard shortcut for duplicating lines (ALT + CTRL + Up/Down) does not work
Configuration: Ubuntu, Eclipse 3.5

This is caused by the system using the same shortcuts to move to a virtual workspace above and below.
Go to System > Preferences > Keyboard Shortcuts and assign a different key combo for the following two actions:
Switch to workspace up
Switch to workspace down
Duplicate lines works again.
**/
```

```
############################
##### Topological Sort #####
############################

private static void topologicalSort(List<Integer>[] edges) {
    Queue<Integer> ready = new LinkedList<Integer>();

    int[] count = new int[edges.length];

    for (int i = 0; i < edges.length; i++) {
        for (int edge : edges[i]) {
            count[edge]++;
        }
    }

    for (int i = 0; i < edges.length; i++) {
        if (count[i] == 0)
            ready.add(i);
    }

    while (!ready.isEmpty()) {
        int next = ready.poll();
        TREAT(next);
        for (int edge : edges[next]) {
            count[edge]--;
            if (count[edge] == 0)
                ready.add(edge);
        }
    }
}

private static boolean isAcyclic(List<Integer>[] edges) {
    Queue<Integer> ready = new LinkedList<Integer>();

    int[] count = new int[edges.length];

    for (int i = 0; i < edges.length; i++) {
        for (int edge : edges[i]) {
            count[edge]++;
        }
    }

    for (int i = 0; i < edges.length; i++) {
        if (count[i] == 0)
            ready.add(i);
    }

    int numSortedVertices = 0;
    while (!ready.isEmpty()) {
        int next = ready.poll();
        TREAT(next);
        for (int edge : edges[next]) {
            count[edge]--;
            if (count[edge] == 0)
                ready.add(edge);
        }
    }
    return numSortedVertices == graph.numVertices();
}


############################
#####     DIJKSTRA     #####
############################

private static class Vertex {
    public int index, value;

    public Vertex(int index, int value) {
        this.index = index;
        this.value = value;
    }

    public boolean equals(Object obj); // create with <index> field
}

private static int[] dijkstra(List<Vertex>[] edges, int origin) {
    boolean[] selected = new boolean[edges.length];
    int[] length = new int[edges.length];
    int[] via = new int[edges.length];

    for (int i = 0; i < edges.length; i++) {
        length[i] = Integer.MAX_VALUE;
    }
    PriorityQueue<Vertex> connected = new PriorityQueue<Vertex>(edges.length, new Comparator<Vertex>() {
        @Override
```

```
        public int compare(Vertex o1, Vertex o2) {
            return o1.value - o2.value;
        }
    });
    length[origin] = 0;
    via[origin] = origin;
    connected.add(new Vertex(origin, length[origin]));
    while (!connected.isEmpty()) {
        int vertex = connected.poll().index;
        selected[vertex] = true;
        exploreVertex(edges, vertex, selected, length, via, connected);
    }

    return length; // or via or both (make them static fields)
}

private static void exploreVertex(List<Vertex>[] edges, int source, boolean[] selected, int[] length, int[] via,
        PriorityQueue<Vertex> connected) {
    for (Vertex edge : edges[source]) {
        final int vertex = edge.index;
        if (!selected[vertex]) {
            int newLength = length[source] + edge.value;
            if (newLength < length[vertex]) {
                boolean vertexIsInQueue = length[vertex] < Integer.MAX_VALUE;
                length[vertex] = newLength;
                via[vertex] = source;
                final Vertex newVertex = new Vertex(vertex, length[vertex]);
                if (vertexIsInQueue)
                    connected.remove(newVertex);
                connected.add(newVertex);
            }
        }
    }
}

############################
#####   Maximum Flow   #####
############################

private static class Edge {
    public final int target;
    public final int weight;

    public Edge(int target, int weight) {
        this.target = target;
        this.weight = weight;
    }
}

private static List<Edge>[] buildNetwork(List<Edge>[] edges) {
    List<Edge>[] newEdges = new List[edges.length];
    for (int i = 0; i < newEdges.length; i++) {
        newEdges[i] = new ArrayList<Edge>();
    }

    for (int i = 0; i < edges.length; i++) {
        List<Edge> vertex = edges[i];
        for (Edge edge : vertex) {
            newEdges[i].add(edge);
            if (!edges[edge.target].contains(i))
                newEdges[edge.target].add(new Edge(i, 0));
        }
    }
    return newEdges;
}

private static int edmondsKarp(List<Edge>[] edges, int source, int sink) {
    List<Edge>[] network = buildNetwork(edges);
    int numVert = network.length;
    int[][] flow = new int[numVert][numVert];

    int[] via = new int[numVert];
    int flowValue = 0;
    int increment = 0;
    while ((increment = findPath(network, flow, source, sink, via)) != 0) {
        flowValue += increment;

        int vertex = sink;
        while (vertex != source) {
            int origin = via[vertex];
            flow[origin][vertex] += increment;
            flow[vertex][origin] -= increment;
            vertex = origin;
        }

    }
```

```
    return flowValue; // or flow or both (make them static fields)
}

private static int findPath(List<Edge>[] network, int[][] flow, int source, int sink, int[] via) {
    int numVert = network.length;
    Queue<Integer> waiting = new LinkedList<Integer>();
    boolean[] found = new boolean[numVert];
    int[] pathIncr = new int[numVert];
    waiting.add(source);
    found[source] = true;
    via[source] = source;
    pathIncr[source] = Integer.MAX_VALUE;

    do {
        int origin = waiting.poll();
        for (final Edge edge : network[origin]) {
            int destin = edge.target;
            int residue = edge.weight - flow[origin][destin];
            if (!found[destin] && residue > 0) {
                via[destin] = origin;
                pathIncr[destin] = Math.min(pathIncr[origin], residue);
                if (destin == sink)
                    return pathIncr[destin];
                else {
                    waiting.add(destin);
                    found[destin] = true;
                }
            }
        }
    } while (!waiting.isEmpty());

    return 0;
}

##############################################
######### Algoritmo Floyd-Warshall ############
##############################################

int[][] functionD( int[][] W, int N) {
    int[][] tableD = new int[N+1][N+1];
    for (int i=0; i <= N; i++) //Linha0: Base da recursivade
        for (int j=0; j <= N; j++)
            tableD[i][j] = W[i][j];
    for (int k=1; k <= N; k++) //Caso geral
        for (int i=1; i <= N; i++)
            for (int j=1; j <= N; j++) {
                int value = tableD[i][k] + tableD[k][j];
                if (tableD[i][j] > value)
                    tableD[i][j] = value;
            }
    return tableD;
}

##############################################
###############   Prim   ####################
##############################################

private static class Edge2 implements Comparable<Edge2> {
    public final int origin;
    public final int target;
    public final int weight;

    public Edge2(int origin, int target, int weight) {
        this.origin = origin;
        this.target = target;
        this.weight = weight;
    }

    @Override
    public int compareTo(Edge2 o) {
        return Integer.compare(weight, o.weight);
    }

    public boolean equals(Object obj); // with target
}

static Iterator<Edge2> mstPrim(List<Edge2>[] edges, int origin) {
    boolean[] selected = new boolean[edges.length];
    int[] cost = new int[edges.length];
    Edge2[] via = new Edge2[edges.length];
    PriorityQueue<Edge2> connected = new PriorityQueue<Edge2>(edges.length);
    List<Edge2> mst = new LinkedList<Edge2>();
    for (int i = 0; i < edges.length; i++) {
        selected[i] = false;
        cost[i] = Integer.MAX_VALUE;
    }
```

```
    cost[origin] = 0;
    connected.add(new Edge2(-1, origin, cost[origin]));
    while (!connected.isEmpty()) {
        int vertex = connected.poll().target;
        selected[vertex] = true;
        if (vertex != origin)
            mst.add(via[vertex]);
        exploreVertex(edges, vertex, selected, cost, via, connected);
    }
    return mst.iterator();
}

private static void exploreVertex(List<Edge2>[] edges, int source, boolean[] selected, int[] cost, Edge2[] via,
        PriorityQueue<Edge2> connected) {
    for (Edge2 edge : edges[source]) {
        int vertex = edge.target;
        if (!selected[vertex] && edge.weight < cost[vertex]) {
            boolean vertexIsInQueue = cost[vertex] < Integer.MAX_VALUE;
            cost[vertex] = edge.weight;
            via[vertex] = edge;
            final Edge2 newEdge = new Edge2(-1, vertex, cost[vertex]);
            if (vertexIsInQueue)
                connected.remove(newEdge);
            connected.add(newEdge);
        }
    }
}

################################################
################  Kruskal  ####################
################################################

interface UnionFind {
    int find(int element);

    void union(int representative1, int representative2);
}

private static class Edge2 implements Comparable<Edge2> {
    public final int origin;
    public final int target;
    public final int weight;

    public Edge2(int origin, int target, int weight) {
        this.origin = origin;
        this.target = target;
        this.weight = weight;
    }

    @Override
    public int compareTo(Edge2 o) {
        return Integer.compare(weight, o.weight);
    }

    public boolean equals(Object obj); // with target
}

private static PriorityQueue<Edge2> buildEdgesPQ(List<Edge2>[] edges, int totalEdges) {
    PriorityQueue<Edge2> queue = new PriorityQueue<Edge2>(totalEdges);
    for (int i = 0; i < edges.length; i++) {
        for (Edge2 edge : edges[i]) {
            queue.add(edge);
        }
    }
    return queue;
}

private static Iterator<Edge2> mstKruskal(List<Edge2>[] edges, int totalEdges) {
    PriorityQueue<Edge2> allEdges = buildEdgesPQ(edges, totalEdges);
    UnionFind vertPartition = new UnionFindInArray(edges.length);
    List<Edge2> mst = new LinkedList<Edge2>();
    int mstFinalSize = edges.length - 1;
    while (mst.size() < mstFinalSize) {
        Edge2 edge = allEdges.poll();
        int rep1 = vertPartition.find(edge.origin);
        int rep2 = vertPartition.find(edge.target);
        if (rep1 != rep2) {
            mst.add(edge);
            vertPartition.union(rep1, rep2);
        }
    }
    return mst.iterator();
}

// <<Disjoint-Set Forests>>
```

```
private static class UnionFindInArray implements UnionFind {

    protected int[] partition;

    public UnionFindInArray(int domainSize) {
        partition = new int[domainSize];
        for (int i = 0; i < domainSize; i++)
            partition[i] = -1;
    }

    protected boolean isInTheDomain(int number) { // optional
        return number >= 0 && number < partition.length;
    }

    protected boolean isRepresentative(int element) { // optional
        return partition[element] < 0;
    }

    @Override
    public int find(int element) {
        int node = element;
        while (partition[node] >= 0)
            node = partition[node];
        return node;
    }

    @Override
    public void union(int rep1, int rep2) {
        if (partition[rep1] <= partition[rep2]) {
            //Height(S1) >= Height(S2)
            if (partition[rep1] == partition[rep2])
                partition[rep1]--;
            partition[rep2] = rep1;
        } else {
            partition[rep1] = rep2;
        }
    }
}

################################################
#############    Bellman-Ford    ################
################################################

public static class NegativeWeightCycleException extends Exception {
}

private static class Edge {
    public final int target;
    public final int weight;

    public Edge(int target, int weight) {
        this.target = target;
        this.weight = weight;
    }
}

private static int[] bellmanFord(List<Edge>[] edges, int origin) throws NegativeWeightCycleException {
    int[] length = new int[edges.length];
    int[] via = new int[edges.length];
    for (int i = 0; i < edges.length; i++) {
        length[i] = Integer.MAX_VALUE;
    }
    length[origin] = 0;
    via[origin] = origin;
    boolean changes = false;
    for (int i = 1; i < edges.length; i++) {
        changes = updateLengths(edges, length, via);
        if (!changes)
            break;
    }
    // Negative-weight cycles detection
    if (changes && updateLengths(edges, length, via))
        throw new NegativeWeightCycleException();
    else
        return length; // or via or both (make them static fields)
}

private static boolean updateLengths(List<Edge>[] edges, int[] length, int[] via) {
    boolean changes = false;
    for (int i = 0; i < edges.length; i++) {
        for (Edge edge : edges[i]) {
            int v1 = i;
            int v2 = edge.target;
            if (length[i] < Integer.MAX_VALUE) {
                int newLength = length[v1] + edge.weight;
                if (newLength < length[v2]) {
```

```
                    length[v2] = newLength;
                    via[v2] = v1;
                    changes = true;
                }
            }
        }
    }
    return changes;
}

################################
####### Kadane's algorithm #######
################################

int maxSubArray(int[] array) {
    int max_ending_here = 0;
    int max_so_far = 0;
    for (int x : array) {
        max_ending_here = Math.max(0, max_ending_here + x);
        if (max_ending_here > max_so_far)
            max_so_far = max_ending_here;
    }
    return max_so_far;
}

################################
########## Math Stuff ############
################################

########################################
########### Is Prime (simple) #############
########################################

public boolean isPrime(long n) {
    // fast even test.
    if(n > 2 && (n & 1) == 0)
        return false;
    // only odd factors need to be tested up to n^0.5
    for(int i = 3; i * i <= n; i += 2)
        if (n % i == 0)
            return false;
    return true;
}

################################################
### Is Prime (faster for numbers up to 2^32) ###
################################################

private static int[] primes;
private static void initializePrimes(int maxNumber) {
    int max = (int) Math.ceil(Math.sqrt(maxNumber));
    primes = new int[max / 2 + 1]; // some space is wasted
    primes[0] = 2;
    int k = 1;
    for (int i = 3; i <= max; i += 2) {
        if (checkPrimes(i)) {
            primes[k++] = i;
        }
    }
}

private static boolean checkPrimes(int n) {
    // fast even test.
    if (n < 2 || n > 2 && (n & 1) == 0)
        return false;
    // only odd factors need to be tested up to n^0.5
    for (int i = 0; primes[i] * primes[i] <= n; i++)
        if (n % primes[i] == 0)
            return false;
    return true;
}

//Máximo Divisor Comum:
private static long mdc(long a, long b){
    while(a % b != 0){
        long r = a % b;
        a = b;
        b = r;
    }
    return b;
}

//Algoritmo de Euclides:
// Calcula a e b tal que ax + by = mdc(a, b).

public static class Pair {
```

```
    long v1, v2;

    public Pair(long v1, long v2) {
        this.v1 = v1;
        this.v2 = v2;
    }
}

public static Pair extended_euclides(long a, long b) {
    long x = 0;
    long lastX = 1;
    long y = 1;
    long lastY = 0;

    while (b != 0) {
        long quotient = a / b;

        long before = a;
        a = b;
        b = mod(before, b);

        before = x;
        x = lastX - quotient * x;
        lastX = before;

        before = y;
        y = lastY - quotient * y;
        lastY = before;
    }

    return new Pair(lastX, lastY);
}

#################################
###### Congruência Linear ########
#################################

// Returns x such that a·x≡ b(mod m). Returns -1 if no solution exists.

// NOTA: obtem os numeros mais pequenos possiveis
public static long solve_lc_swap(long a, long b, long m) {
    if (m > a && m - a != b) {
        long aux = a;
        a = m;
        m = aux;
    }
    return solve_lc(a, b, m);
}

// Retorna sempre numero positivo
public static long solve_lc(long a, long b, long m) {
    if (m < 0) {
        return solve_lc(a, b, -m);
    } else if (a < 0 || b < 0 || a >= m || b >= m) {
        return solve_lc(mod(a, m), mod(b, m), m);
    } else {
        Pair pair = extended_euclides(a, m);
        long u = pair.v1;
        long v = pair.v2;
        long d = u * a + v * m;
        if (mod(b, d) != 0) {
            return -1;
        } else {
            return mod(u * b / d, m);
        }
    }
}

public static long mod(long a, long b) {
    return a % b + (a < 0 ? b : 0);
}

#################################
###### Factorização Rápida #######
#################################

private static List<Integer> factorize(int n){
    List<Integer> factors = new ArrayList<Integer>();
    int f = primes[0];
    int i = 1;

    while(n > 1 && f * f <= n) {
        while(n % f == 0){
            n = n / f;
            factors.add(f);
        }
```

```
        f = primes[i++];
    }
    if(n != 1)
        factors.add(n);
    return factors;
}

##################################
###### Todos os Tuplos ###########
##################################


/**
 * Same as len for's iterating from [0,k[
 *
 * @param vector
 *            the vector with the tuples result
 * @param depth
 *            0 on call
 * @param len
 *            number of "spaces"
 * @param k
 *            number of items to permutate
 */
private static void allTuples(int[] vector, int depth, int len, int k) throws IOException {
    if (depth == len) {
        analyseVector(vector, len);
    } else {
        for (int i = 0; i < k; i++) {
            vector[depth] = i;
            allTuples(vector, depth + 1, len, k);
        }
    }
}

##################################
###### Todas as Permutações ######
##################################

/**
 *
 * @param vector
 *            the vector with the permutation result
 * @param used
 *            boolean array (all false on call)
 * @param depth
 *            first call with 0
 * @param len
 *            number of "spaces"
 * @param k
 *            number of items to permutate
 */
private static void allPerm(int[] vector, boolean[] used, int depth, int len, int k) throws IOException {
    if (depth == len) {
        analyseVector(vector, len);
    } else {
        for (int i = 0; i < k; i++) {
            if (!used[i]) {
                used[i] = true;
                vector[depth] = i;
                allPerm(vector, used, depth + 1, len, k);
                used[i] = false;
            }
        }
    }
}

##################################
########## Combinatória ##########
##################################

(n)      n!
C = ----------
(k)  k!(n-k)!

(n)    (n)        (n-1)   (n-1)
C =     C      =    C    + C
(k)    (n-k)      (k-1)   (k)
```