

Algoritmos de referencia para competencias ICPC

Grupo de Estudio en Programación Competitiva
Semillero de Investigación en Linux y Software libre SILUX
Gerson Lázaro - Melissa Delgado

21 de septiembre de 2017

Índice

1. Bonus: Input Output	2	5. Graphs	8
1.1. Scanner	2	5.1. AdjacencyList	8
1.2. printWriter	2	5.2. AdjacencyMatrix	8
2. Data Structures	3	5.3. BFS	9
2.1. Disjoint Set	3	5.4. Bipartite Check	9
2.2. RMQ	3	5.5. DFS	9
3. Dynamic Programming	4	5.6. Dijkstra's Algorithm	10
3.1. Knapsack	4	5.7. Edge	10
3.2. Longest Common Subsequence	4	5.8. EdgeList	11
3.3. Longest increasing subsequence	5	5.9. FloodFill	11
3.4. Max Range Sum	5	5.10. Floyd Warshall	12
4. Geometry	5	5.11. Kruskal	12
4.1. Angle	5	5.12. LoopCheck	12
4.2. Area	5	5.13. Maxflow	13
4.3. Collinear Points	6	5.14. Node	14
4.4. Convex Hull	6	5.15. Prim	14
4.5. Euclidean Distance	6	5.16. Puentes itmos	15
4.6. Gometric Vector	7	5.17. Tarjan	15
4.7. Perimeter	7	5.18. Topological Sort	16
4.8. Point in Polygon	7	5.19. init	16
4.9. Point	7	6. Math	17
4.10. Sexagesimal degrees and radians	7	6.1. Binary Exponentiation	17
		6.2. Binomial Coefficient	17
		6.3. Catalan Number	17
		6.4. Euler Totient	17
		6.5. Gaussian Elimination	18
		6.6. Greatest common divisor	18
		6.7. Lowest Common multiple	19

6.8. Miller-Rabin	19
6.9. Modular Multiplication	19
6.10. Pollard Rho	19
6.11. Prime Factorization	19
6.12. Sieve of Eratosthenes	20
7. String	20
7.1. KMP's Algorithm	20
7.2. String Hashing	21
7.3. Suffix Array Init	21
7.4. Suffix Array Longest Common Prefix	21
7.5. Suffix Array Longest Common Substring	22
7.6. Suffix Array Longest Repeated Substring	22
7.7. Suffix Array String Matching Boolean	22
7.8. Suffix Array String Matching	22
7.9. Suffix Array strncmp	23
8. Tips and formulas	23
8.1. ASCII Table	23
8.2. Formulas	24
8.3. Sequences	26
8.4. Time Complexities	27

1. Bonus: Input Output

1.1. Scanner

Libreria para recibir las entradas; reemplaza el Scanner original, mejorando su eficiencia.

Contiene los metodos next, nextLine y hasNext. Para recibir datos numericos, hacer casting de next.

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.StringTokenizer;

public class Main {

    static class Scanner{
        BufferedReader br = new BufferedReader(new
            InputStreamReader(System.in));
```

```
StringTokenizer st = new StringTokenizer("");
int spaces = 0;

public String nextLine() throws IOException{
    if (spaces-- > 0) return "";
    else if (st.hasMoreTokens()) return new
        StringBuilder(st.nextToken("\n")).toString();
    return br.readLine();
}

public String next() throws IOException{
    spaces = 0;
    while (!st.hasMoreTokens()) st = new StringTokenizer(br.readLine());
    return st.nextToken();
}

public boolean hasNext() throws IOException{
    while (!st.hasMoreTokens()) {
        String line = br.readLine();
        if (line == null) return false;
        if (line.equals("")) spaces = Math.max(spaces, 0) + 1;
        st = new StringTokenizer(line);
    }
    return true;
}
}
```

1.2. printWriter

Utilizar en lugar del System.out.println para mejorar la eficiencia.

```
import java.io.PrintWriter;

PrintWriter so = new PrintWriter(new BufferedWriter(new
    OutputStreamWriter(System.out)));
so.print("Imprime sin salto de linea");
so.println("Imprime con salto de linea");

//Al finalizar
so.close();
```

2. Data Structures

2.1. Disjoint Set

Estructura de datos para modelar una coleccion de conjuntos disyuntos. Permite determinar de manera eficiente a que conjunto pertenece un elemento, si dos elementos se encuentran en un mismo conjunto y unir dos conjuntos disyuntos en un conjunto mayor.

```
class DisjointSet{
    private int [] parent, size;
    private int cantSets;

    //Inicializa todas las estructuras :)
    public DisjointSet(int n){
        parent=new int[n];
        size=new int[n];
        cantSets=n;

        int i;
        for(i=0; i<n; i++){
            parent[i]=i;
            size[i]=1;
        }

        //Operaciones
        public int find(int i){
            parent[i] = ( parent[i] == i ) ? i : find(parent[i]);
            return parent[i];
        }

        public void union(int i, int j){
            int x=find(i);
            int y=find(j);

            if(x!=y){
                cantSets--;
                parent[x] = y;
                size[y] += size[x];
            }

            public int numDisjointSets(){
                return cantSets;
            }
        }
    }
}
```

```
    }

    public int sizeOfSet(int i){
        return size[find(i)];
    }
}
```

2.2. RMQ

Range minimum query. Recibe como parametro en el constructor un array de valores. Las consultas se realizan con el método rmq(indice_inicio, indice_final) y pueden actualizarse los valores con update_point(indice, nuevo_valor)

```
import java.util.*;

class SegmentTree {
    private int[] st, A;
    private int n;
    private int left (int p) { return p << 1; }
    private int right(int p) { return (p << 1) + 1; }

    private void build(int p, int L, int R) {
        if (L == R)
            st[p] = L;
        else {
            build(left(p) , L, (L + R) / 2);
            build(right(p), (L + R) / 2 + 1, R );
            int p1 = st[left(p)], p2 = st[right(p)];
            st[p] = (A[p1] <= A[p2]) ? p1 : p2;
        }
    }

    private int rmq(int p, int L, int R, int i, int j) {
        if (i > R || j < L) return -1;
        if (L >= i && R <= j) return st[p];
        int p1 = rmq(left(p) , L, (L+R) / 2, i, j);
        int p2 = rmq(right(p), (L+R) / 2 + 1, R, i, j);

        if (p1 == -1) return p2;
        if (p2 == -1) return p1;
        return (A[p1] <= A[p2]) ? p1 : p2; }
}
```

```

private int update_point(int p, int L, int R, int idx, int new_value) {
    int i = idx, j = idx;
    if (i > R || j < L)
        return st[p];
    if (L == i && R == j) {
        A[i] = new_value;
        return st[p] = L;
    }
    int p1, p2;
    p1 = update_point(left(p), L, (L + R) / 2, idx, new_value);
    p2 = update_point(right(p), (L + R) / 2 + 1, R, idx, new_value);

    return st[p] = (A[p1] <= A[p2]) ? p1 : p2;
}

public SegmentTree(int[] _A) {
    A = _A; n = A.length;
    st = new int[4 * n];
    for (int i = 0; i < 4 * n; i++) st[i] = 0;
    build(1, 0, n - 1);
}

public int rmq(int i, int j) { return rmq(1, 0, n - 1, i, j); }
public int update_point(int idx, int new_value) {
    return update_point(1, 0, n - 1, idx, new_value); }
}

class Main {
    public static void main(String[] args) {
        int[] A = new int[] { 18, 17, 13, 19, 15, 11, 20 };
        SegmentTree st = new SegmentTree(A);
    }
}

```

3. Dynamic Programming

3.1. Knapsack

Dados N articulos, cada uno con su propio valor y peso y un tamaño maximo de una mochila, se debe calcular el valor maximo de los elementos que es posible llevar.

Debe seleccionarse un subconjunto de objetos, de tal manera que quepan en la mochila y representen el mayor valor posible.

```

static int MAX_WEIGHT = 40; //Peso maximo de la mochila
static int MAX_N = 1000; //Numero maximo de objetos
static int N; //Numero de objetos
static int prices[] = new int[MAX_N]; //precios de cada producto
static int weights[] = new int[MAX_N]; //pesos de cada producto
static int memo[] [] = new int[MAX_N][MAX_WEIGHT]; //tabla dp

//El metodo debe llamarse con 0 en el id, y la capacidad de la mochila en
w
static int knapsack (int id, int w) {
    if (id == N || w == 0) return 0;
    if (memo[id][w] != -1) return memo[id][w];
    if (weights[id] > w) memo[id][w] = knapsack(id + 1, w);
    else memo[id][w] = Math.max(knapsack(id + 1, w), prices[id] +
        knapsack(id + 1, w - weights[id]));
    return memo[id][w];
}

//Antes de llamar al metodo, todos los campos de la tabla memo deben
iniciarse a -1

```

3.2. Longest Common Subsequence

Dados dos Strings, encuentra el largo de la subsecuencia en común mas larga entre ellas.

```

static int M_MAX = 20; // Máximo size del String 1
static int N_MAX = 20; // Máximo size del String 2
static int m, n; // Size de Strings 1 y 2
static char X[]; // toCharArray del String 1
static char Y[]; // toCharArray del String 2
static int memo[] [] = new int[M_MAX + 1][N_MAX + 1];

static int lcs (int m, int n) {
    for (int i = 0; i <= m; i++) {
        for (int j = 0; j <= n; j++) {
            if (i == 0 || j == 0) memo[i][j] = 0;
            else if (X[i - 1] == Y[j - 1]) memo[i][j] = memo[i - 1][j - 1] + 1;
            else memo[i][j] = Math.max(memo[i - 1][j], memo[i][j - 1]);
        }
    }
}

```

```

    }
}
return memo[m][n];
}

```

3.3. Longest increasing subsequence

Halla la longitud de la subsecuencia creciente mas larga. MAX debe definirse en el tamaño limite del array, n es el tamaño del array. Si debe admitir valores repetidos, cambiar el < de `I[mid] < values[i]` por <=

```

static int inf = 2000000000;
static int MAX = 100000;
static int n;
static int values[] = new int[MAX + 5];
static int L[] = new int[MAX + 5];
static int I[] = new int[MAX + 5];

static int lis() {
    int i, low, high, mid;
    I[0] = -inf;
    for (i = 1; i <= n; i++) I[i] = inf;
    int ans = 0;
    for(i = 0; i < n; i++) {
        low = mid = 0;
        high = ans;
        while(low <= high) {
            mid = (low + high) / 2;
            if(I[mid] < values[i]) low = mid + 1;
            else high = mid - 1;
        }
        I[low] = values[i];
        if(ans < low) ans = low;
    }
    return ans;
}

```

3.4. Max Range Sum

Dado un arreglo de enteros, retorna la máxima suma de un rango de la lista.

```

static int maxRangeSum (int[] a) {
    int sum = 0, ans = 0;
    for (int i = 0; i < a.length; i++) {
        if (sum + a[i] >= 0) {
            sum += a[i];
            ans = Math.max(ans, sum);
        } else sum = 0;
    }
    return ans;
}

```

4. Geometry

4.1. Angle

Dados 3 puntos A, B, y C, determina el valor del angulo ABC (origen en B) en radianes. IMPORTANTE: Definir la clase Point y Vec (Geometric Vector). Si se desea convertir a grados sexagesimales, revisar Sexagesimal degrees [and](#) radians.

```

static double angle(Point a, Point b, Point c) {
    Vec ba = toVector(b, a);
    Vec bc = toVector(b, c);
    return Math.acos((ba.x * bc.x + ba.y * bc.y) / Math.sqrt((ba.x *
        ba.x + ba.y * ba.y) * (bc.x * bc.x + bc.y * bc.y)));
}

```

4.2. Area

Calcula el area de un polígono representado como un ArrayList de puntos. IMPORTANTE: Definir `P[0] = P[n-1]` para cerrar el polígono. El algoritmo utiliza el metodo de determinante de la matriz de puntos de la figura. IMPORTANTE: Debe definirse previamente la clase Point.

```

public static double area(ArrayList<Point> P) {
    double result = 0.0;
    for (int i = 0; i < P.size()-1; i++) {

```

```

        result += ((P.get(i).x * P.get(i + 1).y) - (P.get(i + 1).x *
            P.get(i).y));
    }
    return Math.abs(result) / 2.0;
}

```

4.3. Collinear Points

Determina si el punto r está en la misma línea que los puntos p y q.
 IMPORTANTE: Deben incluirse las estructuras point y vec.

```

static double cross(Vec a, Vec b) {
    return a.x * b.y - a.y * b.x;
}
static boolean collinear(Point p, Point q, Point r) {
    return Math.abs(cross(toVector(p, q), toVector(p, r))) < 1e-9;
}

```

4.4. Convex Hull

Retorna el polígono convexo mas pequeño que cubre (ya sea en el borde o en el interior) un set de puntos. Recibe un vector de puntos, y retorna un vector de puntos indicando el polígono resultante. Es necesario que esten definidos previamente:

Estructuras: point y vec

Métodos: collinear, euclideanDistance, ccw (de inPolygon) y angle.

```

import java.util.ArrayList;
import java.util.Comparator;
import java.util.Collections;

static ArrayList<Point> ConvexHull (ArrayList<Point> P) {
    int i, j, n = (int)P.size();
    if (n <= 3) {
        if (P.get(0).x != P.get(n-1).x || P.get(0).y != P.get(n-1).y)
            P.add(P.get(0));
        return P;
    }
    int P0 = 0;

```

```

for (i = 1; i < n; i++)
    if (P.get(i).y < P.get(P0).y || (P.get(i).y == P.get(P0).y &&
        P.get(i).x > P.get(P0).x)) P0 = i;
Point temp = P.get(0); P.set(0, P.get(P0)); P.set(P0, temp);
Point pivot = P.get(0);
Collections.sort(P, new Comparator<Point>(){
    public int compare(Point a, Point b) {
        if (collinear(pivot, a, b)) return euclideanDistance(pivot, a) <
            euclideanDistance(pivot, b) ? -1 : 1;
        double d1x = a.x - pivot.x, d1y = a.y - pivot.y;
        double d2x = b.x - pivot.x, d2y = b.y - pivot.y;
        return (Math.atan2(d1y, d1x) - Math.atan2(d2y, d2x)) < 0 ? -1 : 1;
    }
});
ArrayList<Point> S = new ArrayList<Point>();
S.add(P.get(n-1)); S.add(P.get(0)); S.add(P.get(1));
i = 2;
while (i < n) {
    j = S.size() - 1;
    if (ccw(S.get(j-1), S.get(j), P.get(i))) S.add(P.get(i++));
    else S.remove(S.size() - 1);
}
return S;
}

```

4.5. Euclidean Distance

Halla la distancia euclidean de 2 puntos en dos dimensiones (x,y). Para usar el primer método, debe definirse previamente la clase Point

```

/*Trabajando con la clase Point*/
static double euclideanDistance(Point p1, Point p2) {
    return Math.hypot(p1.x - p2.x, p1.y - p2.y);
}
/*Trabajando con los valores x y y de cada punto*/
static double euclideanDistance(double x1, double y1, double x2, double
    y2){
    return Math.hypot(x2 - x1, y2 - y1);
}

```

4.6. Gometric Vector

Dados dos puntos A y B, crea el vector A->B. IMPORTANTE: Debe definirse la clase Point. Es llamado Vec para no confundirlo con vector como colección de elementos.

```
static class Vec {
    public double x, y;
    public Vec(double _x, double _y) {
        this.x = _x;
        this.y = _y;
    }
}
static Vec toVector(Point a, Point b) {
    return new Vec(b.x - a.x, b.y - a.y);
}
```

4.7. Perimeter

Calcula el perímetro de un polígono representado como un vector de puntos. IMPORTANTE: Definir P[0] = P[n-1] para cerrar el polígono. La estructura point debe estar definida, al igual que el método euclideanDistance.

```
public static double perimeter (ArrayList<Point> P) {
    double result = 0.0;
    for (int i = 0; i < P.size()-1; i++){
        result += euclideanDistance(P.get(i), P.get(i+1));
    }
    return result;
}
```

4.8. Point in Polygon

Determina si un punto pt se encuentra en el polígono P. Este polígono se define como un vector de puntos, donde el punto 0 y n-1 son el mismo. IMPORTANTE: Deben incluirse las estructuras point y vec, además del método angle y el método cross que se encuentra en Collinear Points.

```
static boolean ccw (Point p, Point q, Point r) {
    return cross(toVector(p, q), toVector(p, r)) > 0;
}

static boolean inPolygon (Point pt, ArrayList<Point> P) {
    if (P.size() == 0) return false;
    double sum = 0;
    for (int i = 0; i < P.size()-1; i++) {
        if (ccw(pt, P.get(i), P.get(i+1))) sum += angle(P.get(i), pt,
            P.get(i+1));
        else sum -= angle(P.get(i), pt, P.get(i+1));
    }
    if(Math.abs(Math.abs(sum) - 2*Math.acos(-1.0)) < 1e-9) return true;
    return false;
}
```

4.9. Point

La clase punto será la base sobre la cual se ejecuten otros algoritmos.

```
static class Point {
    public double x, y;
    public Point() { this.x = this.y = 0.0; }
    public Point(double _x, double _y){
        this.x = _x;
        this.y = _y;
    }
    public boolean equals(Point other){
        if(Math.abs(this.x - other.x) < 1e-9 && (Math.abs(this.y -
            other.y) < 1e-9)) return true;
        return false;
    }
}
```

4.10. Sexagesimal degrees and radians

Conversiones de grados sexagesimales a radianes y viceversa.

```
static double DegToRad(double d) {
    return d * Math.PI / 180.0;
}
```

```
static double RadToDeg(double r) {
    return r * 180.0 / Math.PI;
}
```

5. Graphs

5.1. AdjacencyList

```
import java.util.*;

public class Main{

    static int v, e; //v = cantidad de nodos, e = cantidad de aristas
    static int MAX=1000; //Cantidad Máxima de Nodos
    static ArrayList<Integer> ady[] = new ArrayList[MAX]; //Lista de
        Adyacencia del grafo

    public static void main( String [] args ){
        int origen, destino;
        Scanner sc = new Scanner( System.in );

        //Al iniciar cada caso de prueba
        v = sc.nextInt();
        e = sc.nextInt();
        init();

        while( e > 0 ){
            origen = sc.nextInt();
            destino = sc.nextInt();

            ady[ origen ].add( destino );
            ady[ destino ].add( origen );
            e--;
        }
    }

    static void init(){
        int i;
        for( i = 0; i < v; i++ ){
            ady[i] = new ArrayList<Integer>();
        }
    }
}
```

```
}
}
```

5.2. AdjacencyMatrix

```
import java.util.*;

public class Main{

    static int v, e; //v = cantidad de nodos, e = cantidad de aristas
    static int MAX=1000; //Cantidad Máxima de Nodos
    static int ady[] [] = new int [MAX] [MAX];

    public static void main( String [] args ){
        int origen, destino;
        Scanner sc = new Scanner( System.in );

        //Al iniciar cada caso de prueba
        v = sc.nextInt();
        e = sc.nextInt();
        init();

        while( e > 0 ){
            origen = sc.nextInt();
            destino = sc.nextInt();

            ady[ origen ][ destino ] = 1;
            ady[ destino ][ origen ] = 1;
            e--;
        }
    }

    static void init(){
        int i, j;
        for( i = 0; i < v; i++ ){
            for( j = 0; j < v; j++ ){
                ady[i][j] = 0;
            }
        }
    }
}
```

5.3. BFS

Algoritmo de búsqueda en anchura en grafos, recibe un nodo inicial *s* y visita todos los nodos alcanzables desde *s*. BFS también halla la distancia más corta entre el nodo inicial *s* y los demás nodos si todas las aristas tienen peso 1.

SE DEBEN LIMPIAR LAS ESTRUCTURAS DE DATOS ANTES DE UTILIZARSE

```
static int v, e; //vertices, arcos
static int MAX=100005;
static ArrayList<Integer> ady[] = new ArrayList[MAX]; //lista de
    Adyacencia
static long distance[] = new long[MAX];

//Recibe el nodo inicial s
static void bfs(int s){
    Queue<Integer> q = new LinkedList<Integer>();
    q.add(s);
    distance[s] = 0;
    int actual, i, next;

    while( !q.isEmpty() ){
        actual = q.poll();
        for( i = 0; i < ady[actual].size(); i++){
            next = ady[actual].get(i);
            if( distance[next] == -1 ){
                distance[next] = distance[actual] + 1;
                q.add(next);
            }
        }
    }
}
```

5.4. Bipartite Check

Algoritmo para la detección de grafos bipartitos. Modificación de BFS. SE DEBEN LIMPIAR LAS ESTRUCTURAS DE DATOS ANTES DE UTILIZARSE

```
static int v, e; //vertices, arcos
static int MAX=100005;
static ArrayList<Integer> ady[] = new ArrayList[MAX]; //lista de
    Adyacencia
static int color[] = new int[MAX];
```

```
static boolean bipartite;

//Recibe el nodo inicial s
static void bfs(int s){
    Queue<Integer> q = new LinkedList<Integer>();
    q.add(s);
    color[s] = 0;
    int actual, i, next;

    while( !q.isEmpty() && bipartite ){
        actual = q.poll();
        for( i = 0; i < ady[actual].size(); i++){
            next = ady[actual].get(i);
            if( color[next] == -1 ){
                color[next] = 1 - color[actual];
                q.add(next);
            }else if( color[next] == color[actual] ){
                bipartite = false;
                return;
            }
        }
    }
}
```

5.5. DFS

Algoritmo de búsqueda en profundidad para grafos. Parte de un nodo inicial *s* visita a todos sus vecinos. DFS puede ser usado para contar la cantidad de componentes conexas en un grafo y puede ser modificado para que retorne información de los nodos dependiendo del problema. Permite hallar ciclos en un grafo.

SE DEBEN LIMPIAR LAS ESTRUCTURAS DE DATOS ANTES DE UTILIZARSE

```
static int v, e; //vertices, arcos
static int MAX=100005; //Cantidad máxima de nodos del grafo
static ArrayList<Integer> ady[] = new ArrayList[MAX]; //Lista de
    adyacencia
static boolean marked[] = new boolean[MAX]; //Estructura auxiliar para
    marcar los grafos visitados

//Recibe el nodo inicial s
static void dfs( int s ){
    marked[s] = true;
```

```

    int i, next;

    for( i = 0; i < ady[s].size(); i++){
        next = ady[s].get(i);
        if( !marked[next] ){
            dfs(next);
        }
    }
}

```

5.6. Dijkstra's Algorithm

Algoritmo que dado un grafo con pesos no negativos halla la ruta mínima entre un nodo inicial s y todos los demás nodos.

SE DEBEN LIMPIAR LAS ESTRUCTURAS DE DATOS ANTES DE UTILIZARSE

```

static int v, e; //v = cantidad de nodos, e = cantidad de aristas
static int MAX=100005; //Cantidad Máxima de Nodos
static ArrayList<Node> ady[] = new ArrayList[MAX]; //Lista de Adyacencia
del grafo
static boolean marked[] = new int[MAX]; //Estructura auxiliar para marcar
los nodos visitados
static long distance[] = new long[MAX]; //Estructura auxiliar para llevar
las distancias a cada nodo
static int prev[] = new int[MAX]; //Estructura auxiliar para almacenar
las rutas

//Recibe el nodo inicial s
static void dijkstra( int s ) {
    PriorityQueue<Node> pq = new PriorityQueue<Node>();
    pq.add(new Node(s, 0)); //se inserta a la cola el nodo Inicial.
    distance[s] = 0;
    int actual, j, adjacent;
    long weight;
    Node x;

    while( pq.size() > 0 ) {
        actual = pq.peek().adjacent;
        pq.poll();
        if ( !marked[actual] ) {
            marked[actual] = true;
            for ( j = 0; j < ady[actual].size(); j++ ) {
                adjacent = ady[actual].get(j).adjacent;

```

```

                weight = ady[actual].get(j).cost;
                if ( !marked[adjacent] ) {
                    if (distance[adjacent] > distance[actual] + weight) {
                        distance[adjacent] = distance[actual] + weight;
                        prev[adjacent] = actual;
                        pq.add(new Node(adjacent, distance[adjacent]));
                    }
                }
            }
        }
    }

    //Retorna en un String la ruta desde s hasta t
    //Recibe el nodo destino t
    static String path(int t) {
        String r="";
        while(prev[t]!=-1){
            r="-"+t+r;
            t=prev[t];
        }
        if(t!=-1){
            r=t+r;
        }
        return r;
    }
}

```

5.7. Edge

Estructura Edge con su comparador. Usada en algoritmos como Kruskal y Puentes e Itmos.

```

/* Arco Simple */
static class Edge{

    public int src, dest;

    public Edge(int s, int d) {
        this.src = s;
        this.dest = d;
    }
}

```

```

/* Arco con pesos */
static class Edge implements Comparable<Edge> {

    public int src, dest, weight;

    public Edge(int s, int d, int w) {
        this.src = s;
        this.dest = d;
        this.weight=w;
    }

    @Override
    public int compareTo(Edge o) {
        return this.weight-o.weight;
    }
}

```

5.8. EdgeList

```

import java.util.*;

public class Main{

    static int v, e; //v = cantidad de nodos, e = cantidad de aristas
    static ArrayList<Edge> edges;

    public static void main( String [] args ){
        int origen, destino;
        Scanner sc = new Scanner( System.in );

        //Al iniciar cada caso de prueba
        v = sc.nextInt();
        e = sc.nextInt();
        init();

        while( e > 0 ){
            origen = sc.nextInt();
            destino = sc.nextInt();
            edges.add( new Edge(origen, destino) );
            e--;
        }
    }
}

```

```

static void init(){
    edges = new ArrayList<Edge>(); //Kruskal
}

/* Arco Simple */
static class Edge{

    public int src, dest;

    public Edge(int s, int d) {
        this.src = s;
        this.dest = d;
    }
}

```

5.9. FloodFill

Dado un grafo implícito colorea y cuenta el tamaño de las componentes conexas. Normalmente usado en rejillas 2D.

//aka Coloring the connected components

```

static int tam = 1000; //Máximo tamaño de la rejilla
static int dy[] = {1,1,0,-1,-1,-1, 0, 1}; //Estructura auxiliar
    para los desplazamientos
static int dx[] = {0,1,1, 1, 0,-1,-1,-1};
static char grid[] [] = new char [tam][tam]; //Matriz de caracteres
static int X, Y; //Tamaño de la matriz

/*Este método debe ser llamado con las coordenadas x, y donde se
    inicia el
recorrido. c1 es el color que estoy buscando, c2 el color con el
    que se va
a pintar. Retorna el tamaño de la componente conexas*/
static int floodfill(int y, int x, char c1, char c2) {
    if (y < 0 || y >= Y || x < 0 || x >= X) return 0;

    if (grid[y][x] != c1) return 0; // base case

    int ans = 1;
    grid[y][x] = c2; // se cambia el color para prevenir ciclos

```

```

        for (int i = 0; i < 8; i++)
            ans += floodfill(y + dy[i], x + dx[i], c1, c2);

        return ans;
    }

```

5.10. Floyd Warshall

Algoritmo para grafos que halla la distancia mínima entre cualquier par de nodos. `ady[i][j]` guardará la distancia mínima entre el nodo `i` y el `j`.

Ajustar los tipos de datos segun el problema.

SE DEBEN LIMPIAR LAS ESTRUCTURAS DE DATOS ANTES DE UTILIZARSE

```

static int v, e; //vertices, arcos
static int MAX = 505;
static int ady[][] = new int [MAX][MAX];

static void floydWarshall(){
    int i,j,k, aux;

    for( k = 0; k < v; k++ ){
        for( i = 0; i < v; i++ ){
            for( j = 0; j < v; j++ ){
                ady[i][j] = min( ady[i][j], ( ady[i][k] + ady[k][j] ) );
            }
        }
    }
}

```

5.11. Kruskal

Algoritmo para hallar el arbol cobertor mínimo de un grafo no dirigido y conexo. Utiliza la técnica de Union-Find(Conjuntos disjuntos) para detectar que aristas generan ciclos.

Requiere la clase Edge(con pesos).

SE DEBEN LIMPIAR LAS ESTRUCTURAS DE DATOS ANTES DE UTILIZARSE

```

static int v, e; //vertices, arcos
static int MAX=100005;
static int parent[] = new int [MAX];

```

```

static ArrayList<Edge> edges;
static ArrayList<Edge> answer;

//UNION-FIND
static int find(int i){
    parent[i] = ( parent[i] == i ) ? i : find(parent[i]);
    return parent[i];
}

static void unionFind(int x, int y){
    parent[ find(x) ] = find(y);
}

static void kruskall(){
    Edge actual;
    int aux, i, x,y;
    aux = i = 0;
    Collections.sort(edges);

    while( aux < (v-1) && i < edges.size() ){
        actual = edges.get(i);
        x = find(actual.src);
        y = find(actual.dest);

        if( x != y ){
            answer.add(actual);
            aux++;
            unionFind(x, y);
        }
        i++;
    }
}

```

5.12. LoopCheck

Determina si un Grafo DIRIGIDO tiene o no ciclos.

SE DEBEN LIMPIAR LAS ESTRUCTURAS DE DATOS ANTES DE UTILIZARSE

```

static final int MAX = 10010; //Cantidad maxima de nodos
static int v; //Cantidad de Nodos del grafo
static ArrayList<Integer> ady[] = new ArrayList[MAX]; //Estructura para
    almacenar el grafo
static int dfs_num[] = new int[MAX];

```

```

static boolean loops; //Bandera de ciclos en el grafo

/* DFS_NUM STATES
   2 - Explored
   3 - Visited
  -1 - Unvisited
*/

/*
Este metodo debe ser llamado desde un nodo inicial u.
Cortara su ejecucion en el momento que encuentre algun ciclo en el grafo.
*/
static void graphCheck( int u ){
    int j, next;

    if( loops ) return;

    dfs_num[u] = 2;

    for( j = 0; j < ady[u].size(); j++ ){
        next = ady[u].get(j);

        if( dfs_num[next] == -1 )    graphCheck( next );
        else if( dfs_num[next] == 2 ){
            loops = true;
            break;
        }
    }

    dfs_num[u] = 3;
}

public static void main(String args[]){
    for( int s = 1; s <= v && !loops; s++ ){ //Por si el grafo es NO
        conexo
        if( dfs_num[s] == -1 ) graphCheck(s);
    }
}

```

5.13. Maxflow

Dado un grafo, halla el máximo flujo entre una fuente s y un sumidero t.

SE DEBEN LIMPIAR LAS ESTRUCTURAS DE DATOS ANTES DE UTILIZARSE

```

static int n; //Cantidad de nodos del grafo
static ArrayList<Integer> ady[] = new ArrayList[105]; //lista de
    Adyacencia
static int capacity[][] = new int[105][105]; //Capacidad de aristas de la
    red
static int flow[][] = new int[105][105]; //Flujo de cada arista
static int prev[] = new int[105];

static void connect(int i, int j, int cap){
    ady[i].add(j);
    ady[j].add(i);
    capacity[i][j] += cap;
    //Si el grafo es dirigido no hacer esta linea
    //capacity[j][i] += cap;
}

static int maxflow(int s, int t, int n){ //s=fuente, t=sumidero, n=numero
    de nodos
    int i, j, maxFlow, u, v, extra, start, end;
    for( i = 0; i <= n; i++ ){
        for( j = 0; j <= n; j++ ){
            flow[i][j] = 0;
        }
    }

    maxFlow = 0;

    while( true ){
        for( i = 0; i <= n; i++ ) prev[i] = -1;

        Queue<Integer> q = new LinkedList<Integer>();
        q.add(s);
        prev[s] = -2;

        while( !q.isEmpty() ){
            u = q.poll();
            if( u == t ) break;
            for( j = 0; j < ady[u].size(); j++){
                v = ady[u].get(j);
                if( prev[v] == -1 && capacity[u][v] - flow[u][v] > 0 ){
                    q.add(v);
                    prev[v] = u;
                }
            }
        }
    }
}

```

```

    }
}
if( prev[t] == -1 ) break;

extra = Integer.MAX_VALUE;
end = t;
while( end != s ){
    start = prev[end];
    extra = Math.min( extra, capacity[start][end]-flow[start][end]
    );
    end = start;
}

end = t;
while( end != s){
    start = prev[end];
    flow[start][end] += extra;
    flow[end][start] = -flow[start][end];
    end = start;
}

maxFlow += extra;
}

return maxFlow;
}

public static void main( String args[] ){
    //Para cada arista
    connect( s, d, f); //origen, destino, flujo
}

```

5.14. Node

Estructura Node con su comparador. Usada en algoritmos como Dijkstra.

```

static class Node implements Comparable<Node> {
    public int adjacent;
    public int cost;

    public Node(int ady, int c) {
        this.adjacent = ady;
        this.cost = c;
    }
}

```

```

    }

    @Override
    public int compareTo(Node o) {
        if (this.cost >= o.cost) return 1;
        else return -1;
    }
}

```

5.15. Prim

Algoritmo para hallar el árbol cobertor mínimo de un grafo no dirigido y conexo.

Requiere de la clase Node

SE DEBEN LIMPIAR LAS ESTRUCTURAS DE DATOS ANTES DE UTILIZARSE

```

static int v, e; //vertices, arcos
static int MAX=100005;
static ArrayList<Node> ady[] = new ArrayList[MAX];
static boolean marked[] = new boolean[MAX];
static int rta;
static PriorityQueue<Node> pq;

static void prim(){
    process(0); //Nodo inicial;
    int u, w;

    while( pq.size() > 0 ){
        u = pq.peek().adjacent;
        w = pq.peek().cost;

        pq.poll();

        if( !marked[u] ){
            rta += w;
            process(u);
        }
    }
}

static void process( int u ){
    marked[u] = true;
    int i, v;
}

```

```

for( i = 0; i < ady[u].size(); i++){
    v = ady[u].get(i).adjacent;
    if( !marked[v] ){
        pq.add( new Node( v, ady[u].get(i).cost ) );
    }
}
}

```

5.16. Puentes itmos

Algoritmo para hallar los puentes e itmos en un grafo no dirigido.
 Requiere de la clase Edge.
 SE DEBEN LIMPIAR LAS ESTRUCTURAS DE DATOS ANTES DE UTILIZARSE

```

static int n, e; //vertices, arcos
static int MAX=1010;
static ArrayList<Integer> ady[]=new ArrayList [MAX];
static boolean marked[]=new boolean [MAX];
static int prev[]=new int [MAX];
static int dfs_low[]=new int [MAX];
static int dfs_num[]=new int [MAX];
static boolean itsmos[]=new int [MAX];
static ArrayList<Edge> bridges;
static int dfsRoot, rootChildren, cont;

/* Recibe el nodo inicial */
static void dfs(int u){
    dfs_low[u] = dfs_num[u] = cont;
    cont++;
    marked[u] = true;
    int j, v;

    for(j = 0; j < ady[u].size(); j++){
        v = ady[u].get(j);
        if( !marked[v] ){
            prev[v] = u;
            //Caso especial
            if( u == dfsRoot ) rootChildren++;
            dfs(v);

            //Itmos
            if( dfs_low[v] >= dfs_num[u] ) itsmos[u] = true;
        }
    }
}

```

```

//Puentes
if( dfs_low[v] > dfs_num[u] ) bridges.add(new Edge(
    Math.min(u,v),Math.max(u,v)) );

dfs_low[u] = Math.min(dfs_low[u], dfs_low[v]);
}else if( v != prev[u] ) dfs_low[u] = Math.min(dfs_low[u],
    dfs_num[v]);
}
}

public static void main(String args[]){
    dfs( dfsRoot );
    /* Caso especial */
    itmos[dfsRoot] = ( itmos[ dfsRoot ] && rootChildren > 1 ) ? true :
        false;
}

```

5.17. Tarjan

Algoritmo para hallar componentes fuertemente conexas(SCC) en grafos
 dirigidos.
 SE DEBEN LIMPIAR LAS ESTRUCTURAS DE DATOS ANTES DE UTILIZARSE

```

int v, e;
static int n = 5000; // Máxima cantidad de nodos
static int dfs_low[] = new int[n];
static int dfs_num[] = new int[n];
static boolean marked[] = new boolean[n];
static Stack<Integer> s;
static int dfsCont, cantSCC;
static ArrayList<Integer> ady[] = new ArrayList[n];

public static void main (String[] args){
    for( int i = 0; i < v; i++ ){ //Por si el grafo no es conexo
        if( dfs_num[i] == -1 ){
            dfsCont = 0;
            s = new Stack<Integer>();
            tarjanSCC(i);
        }
    }
}

```

```

public static void tarjanSCC( int u ){
    dfs_low[u] = dfs_num[u] = dfsCont;
    dfsCont++;
    s.push(u);
    marked[u] = true;

    int j, v;

    for( j = 0; j < ady[u].size(); j++ ){
        v = ady[u].get( j );

        if( dfs_num[v] == -1 ){
            tarjanSCC( v );
        }

        if( marked[v] ){
            dfs_low[u] = Math.min( dfs_low[u], dfs_low[v] );
        }
    }

    if( dfs_low[u] == dfs_num[u] ){
        cantSCC++;
        /* ***** */
        /* Esta seccion se usa para imprimir las componentes conexas */
        System.out.println("COMPONENTE CONEXA #" + cantSCC );
        while( !s.empty() ){
            v = s.peek();
            s.pop();
            marked[v] = false;
            System.out.println(v);
            if( u == v ) break;
        }
        /* ***** */
    }
}

```

5.18. Topological Sort

Dado un grafo acíclico y dirigido, ordena los nodos linealmente de tal manera que si existe una arista entre los nodos u y v entonces u aparece antes que v.

Este ordenamiento es una manera de poner todos los nodos en una línea recta de tal manera que las aristas vayan de izquierda a derecha. SE DEBEN LIMPIAR LAS ESTRUCTURAS DE DATOS ANTES DE UTILIZARSE

```

static int v, e; //vertices, arcos
static int MAX=100005; //Cantidad máxima de nodos del grafo
static ArrayList<Integer> ady[] = new ArrayList[MAX]; //Lista de
    adyacencia
static ArrayList<Integer> topoSort; //Lista de adyacencia
static boolean marked[] = new boolean[MAX]; //Estructura auxiliar para
    marcar los grafos visitados

//Recibe un nodo inicial u
static void dfs( int u ){
    int i, v;
    marked[u] = 1;
    for( i = 0; i < ady[u].size(); i++){
        v = ady[u].get(i);
        if( !marked[v] ) dfs(v);
    }
    topoSort.add(u);
}

public static void main( String args[]){
    for(i=0; i<v; i++){
        if( !marked[i] )      dfs(i)
    }
    //imprimir topoSort en reversa :3
}

```

5.19. init

Método para la limpieza de TODAS las estructuras de datos utilizadas en TODOS los algoritmos de grafos.

Copiar solo las necesarias, de acuerdo al algoritmo que se este utilizando.

```

/*Debe llamarse al iniciar cada caso de prueba luego de haber leído la
    cantidad de nodos v
Limpia todas las estructuras de datos.*/
static void init() {
    long max = Long.MAX_VALUE;
    edges = new ArrayList<Edge>(); //Kruskal
}

```



```

answer = new ArrayList<Edge>(); //Kruskal
loops = false; //Loop Check
rta = 0; //Prim
pq = new PriorityQueue<Node>(); //Prim
cont = dfsRoot = rootChildren = 0; //Puentes
bridges = new ArrayList<Edge>(); //Puentes
cantSCC = 0; //Tarjan
topoSort = new ArrayList<Integer>(); //Topological Sort
bipartite = true;

for (int j = 0; j <= v; j++) {
    distance[j] = -1; //Distancia a cada nodo (BFS)
    distance[j] = max; //Distancia a cada nodo (Dijkstra)
    ady[j] = new ArrayList<Integer>(); //Lista de Adyacencia
    ady[j] = new ArrayList<Node>(); //Lista de Adyacencia (Dijkstra)
    marked[j] = false; //Estructura auxiliar para marcar los nodos ya
        visitados
    prev[j] = -1; //Estructura auxiliar para almacenar las rutas
    parent[j] = j; //Estructura auxiliar para DS
    dfs_num[j] = -1;
    dfs_low[j] = 0;
    itsmos[j] = false;
    color[j] = -1; //Bipartite Check

    for(j = 0; j < v; j++) ady[i][j] = Integer.MAX_VALUE; //Warshall
}
}

```

6. Math

6.1. Binary Exponentiation

Realiza a^b y retorna el resultado módulo c

```

static long binaryExponentiation (long a, long b, long c) {
    if (b == 0) return 1;
    if (b % 2 == 0){
        long temp = binaryExponentiation(a, b/2, c);
        return (temp * temp) % c;
    } else {
        long temp = binaryExponentiation(a, b-1, c);
        return (temp * a) % c;
    }
}

```

```

}
}

```

6.2. Binomial Coefficient

Calcula el coeficiente binomial nCr , entendido como el número de subconjuntos de k elementos escogidos de un conjunto con n elementos.

```

static long binomialCoefficient(long n, long r) {
    if (r < 0 || n < r) return 0;
    r = Math.min(r, n - r);
    long ans = 1;
    for (int i = 1; i <= r; i++) {
        ans = ans * (n - i + 1) / i;
    }
    return ans;
}

```

6.3. Catalan Number

Guarda en el array Catalan Numbers los numeros de Catalan hasta MAX.

```

static int MAX = 30;
static long catalanNumbers[] = new long[MAX+1];

static void catalan(){
    catalanNumbers[0] = 1;
    for(int i = 1; i <= MAX; i++){
        catalanNumbers[i] =
            (long)(catalanNumbers[i-1]*((double)(2*((2 * i)-
                1))/(i + 1)));
    }
}

```

6.4. Euler Totient

Función totient o indicatriz de Euler. Para cada posición n del array result retorna el número de enteros positivos menores o iguales a n que son coprimos con n (Coprimos: MCD=1)

```

static void totient (int n, int resultados[]) {
    boolean aux[] = new boolean[n];
    for (int i = 0; i < n; i++) {
        resultados[i] = i;
    }
    for (int i = 2; i < n; i++){
        if (!aux[i]) {
            for (int j = i; j < n ; j += i){
                aux[j] = true;
                resultados[j] = resultados[j] -
                    (resultados[j] / i) ;
            }
            aux[i] = false;
        }
    }
}

```

6.5. Gaussian Elimination

Resuelve sistemas de ecuaciones lineales por eliminación Gaussiana. matrix contiene los valores de la matriz cuadrada y result los resultados de las ecuaciones. Retorna un vector con el valor de las n incongnitas. Los resultados pueden necesitar redondeo.

```

import java.util.ArrayList;

static int MAX = 100;
static int n = 3;
static double matrix[][] = new double[MAX][MAX];
static double result[] = new double[MAX];

static ArrayList<Double> gauss() {
    ArrayList<Double> ans = new ArrayList<Double>();
    for(int i = 0; i < n; i++) ans.add(0.0);
    double temp;
    for (int i = 0; i < n; i++) {
        int pivot = i;
        for (int j = i + 1; j < n; j++) {
            temp = Math.abs(matrix[j][i]) - Math.abs(matrix[pivot][i]);
            if (temp > 0.000001) pivot = j;
        }
        double temp2[] = new double[n];

```

```

        System.arraycopy(matrix[i],0,temp2,0,n);
        System.arraycopy(matrix[pivot],0,matrix[i],0,n);
        System.arraycopy(temp2,0,matrix[pivot],0,n);
        temp = result[i];
        result[i] = result[pivot];
        result[pivot] = temp;
        if (!(Math.abs(matrix[i][i]) < 0.000001)) {
            for (int k = i + 1; k < n; k++) {
                temp = -matrix[k][i] / matrix[i][i];
                matrix[k][i] = 0;
                for (int l = i + 1; l < n; l++) {
                    matrix[k][l] += matrix[i][l] * temp;
                }
                result[k] += result[i] * temp;
            }
        }
    }
    for (int m = n - 1; m >= 0; m--) {
        temp = result[m];
        for (int i = n - 1; i > m; i--) {
            temp -= ans.get(i) * matrix[m][i];
        }
        ans.set(m,temp / matrix[m][m]);
    }
    return ans;
}

```

6.6. Greatest common divisor

Calcula el máximo común divisor entre a y b mediante el algoritmo de Euclides

```

static int mcd(int a, int b) {
    int aux;
    while (b != 0){
        a %= b;
        aux = b;
        b = a;
        a = aux;
    }
    return a;
}

```

6.7. Lowest Common multiple

Calculo del mínimo común múltiplo usando el máximo común divisor REQUIERE

```
mcd(a,b)
```

```
static int mcm (int a, int b) {  
    return a * b / mcd(a, b);  
}
```

6.8. Miller-Rabin

La función de Miller-Rabin determina si un número dado es o no un número primo. IMPORTANTE: Debe utilizarse el método binaryExponentiation y Modular Multiplication.

```
public static boolean miller (long p) {  
    if (p < 2 || (p != 2 && p % 2==0)) return false;  
    long s = p - 1;  
    while (s % 2 == 0) s /= 2;  
    for (int i = 0; i < 5; i++){  
        long a = (long)(Math.random() * 1000000000) % (p - 1) + 1;  
        long temp = s;  
        long mod = binaryExponentiation(a, temp, p);  
        while (temp != p - 1 && mod != 1 && mod != p - 1){  
            mod = mulmod(mod, mod, p);  
            temp *= 2;  
        }  
        if (mod != p - 1 && temp % 2 == 0) return false;  
    }  
    return true;  
}
```

6.9. Modular Multiplication

Realiza la operación $(a * b) \% \text{mod}$ minimizando posibles desbordamientos.

```
public static long mulmod (long a, long b, long mod) {  
    long x = 0;  
    long y = a % mod;  
    while (b > 0){  
        if (b % 2 == 1) x = (x + y) % mod;
```

```
        y = (y * 2) % mod;  
        b /= 2;  
    }  
    return x % mod;  
}
```

6.10. Pollard Rho

La función Rho de Pollard calcula un divisor no trivial de n . IMPORTANTE: Deben implementarse Modular Multiplication y Greatest Common Divisor (para `long long`).

```
public static long pollardRho (long n) {  
    int i = 0, k = 2;  
    long d, x = 3, y = 3;  
    while (true) {  
        i++;  
        x = (mulmod(x, x, n) + n - 1) % n; // generating function  
        d = mcd(Math.abs(y - x), n); // the key insight  
        if (d != 1 && d != n) return d;  
        if (i == k) {  
            y = x;  
            k *= 2;  
        }  
    }  
}
```

6.11. Prime Factorization

Guarda en `primeFactors` la lista de factores primos del `value` de menor a mayor. IMPORTANTE: Debe ejecutarse primero la criba de Eratostenes. La criba debe existir al menos hasta la raíz cuadrada de `value` (se recomienda dejar un poco de excedente).

```
import java.util.ArrayList;  
  
static ArrayList<Long> primeFactors = new ArrayList<Long>();  
  
static void calculatePrimeFactors(long value){  
    primeFactors.clear();  
    long temp = value;
```

```

    int factor;
    for (int i = 0; (long)primes.get(i) * primes.get(i) <= value; ++i){
        factor = primes.get(i);
        while (temp % factor == 0){
            primeFactors.add((long)factor);
            temp /= factor;
        }
    }
    if (temp != 1) primeFactors.add(temp);
}

```

6.12. Sieve of Eratosthenes

Guarda en primes los números primos menores a MAX

```

import java.util.ArrayList;

static int MAX = 10000000;
static ArrayList<Integer> primes = new ArrayList<Integer>();
static boolean sieve[] = new boolean[MAX+5];

static void calculatePrimes() {
    sieve[0] = sieve[1] = true;
    int i;
    for (i = 2; i * i <= MAX; ++i) {
        if (!sieve[i]) {
            primes.add(i);
            for (int j = i * i; j <= MAX; j += i) sieve[j] = true;
        }
    }
    for(; i <= MAX; i++){
        if (!sieve[i]) primes.add(i);
    }
}

```

7. String

7.1. KMP's Algorithm

Encuentra si el string pattern se encuentra en el string cadena.

```

import java.util.ArrayList;

static ArrayList<Integer> table(String pattern){
    int m=pattern.length();
    ArrayList<Integer> border = new ArrayList<Integer>();
    border.add(0);
    int temp;
    for(int i=1; i<m; ++i){
        border.add(border.get(i-1));
        temp = border.get(i);
        while(temp>0 && pattern.charAt(i)!=pattern.charAt(temp)){
            if(temp <= i+1){
                border.set(i,border.get(temp-1));
                temp = border.get(i);
            }
            if(pattern.charAt(i) == pattern.charAt(temp)){
                border.set(i,temp+1);
            }
        }
        return border;
    }
}

static boolean kmp(String cadena, String pattern){
    int n=cadena.length();
    int m=pattern.length();
    ArrayList<Integer> tab=table(pattern);
    int seen=0;

    for(int i=0; i<n; i++){
        while(seen>0 && cadena.charAt(i)!=pattern.charAt(seen)){
            seen=tab.get(seen-1);
        }
        if(cadena.charAt(i)==pattern.charAt(seen))
            seen++;
        if(seen==m){
            return true;
        }
    }
    return false;
}

```

7.2. String Hashing

Estructura para realizar operaciones de hashing.

```
static class Hash {
    char[] s;
    int[] h;
    int[] pot;
    int p = 265; //Número pseudo-aleatorio base del polinomio (mayor
        al tamaño del lenguaje)
    long MOD = 1000000009; //Número primo grande

    public Hash(String _s) {
        h = new int[_s.length() + 1];
        pot = new int[_s.length() + 1];
        s = _s.toCharArray(); pot[0] = 1;
        for(int i = 1; i <= s.length; i++) {
            h[i] = (int)(((long)h[i - 1] * p + s[i - 1]) % MOD);
            pot[i] = (int)(((long)pot[i - 1] * p) % MOD);
        }
    }

    int hashValue(int i, int j) {
        int ans = (int)(h[j] - (long) h[i] * pot[j - i] % MOD);
        return (ans >= 0) ? ans : (int)(ans + MOD);
    }
}
```

7.3. Suffix Array Init

Crea el suffix array. Deben inicializarse las variables s (String original), N_MAX (Máximo size que puede tener s), y n (Size del string actual).

```
static String s;
static int N_MAX = 30;
static int n;
static char _s[];
static int sa[] = new int[N_MAX];
static int rk[] = new int[N_MAX];
static long rk2[] = new long[N_MAX];

static List<Integer> wrapper = new AbstractList<Integer>() {
    @Override
```

```
public Integer get(int i) { return sa[i]; }

    @Override
    public int size() { return n; }

    @Override
    public Integer set(int i, Integer e) {
        int v = sa[i];
        sa[i] = e;
        return v;
    }
};

static void suffixArray() {
    _s = s.toCharArray();
    for (int i = 0; i < n; i++) {
        sa[i] = i; rk[i] = _s[i]; rk2[i] = 0;
    }
    for (int l = 1; l < n; l <= 1) {
        for (int i = 0; i < n; i++) {
            rk2[i] = ((long) rk[i] << 32) + (i + l < n ? rk[i + l] : -1);
        }
        Collections.sort(wrapper, new Comparator<Integer>() {
            @Override
            public int compare(Integer o1, Integer o2) {
                if(rk2[o1.intValue()] > rk2[o2.intValue()]) return 1;
                else if(rk2[o1.intValue()] == rk2[o2.intValue()]) return 0;
                else return -1;
            }
        });
    }
    for (int i = 0; i < n; i++) {
        if (i > 0 && rk2[sa[i]] == rk2[sa[i - 1]])
            rk[sa[i]] = rk[sa[i - 1]];
        else rk[sa[i]] = i;
    }
}
```

7.4. Suffix Array Longest Common Prefix

Calcula el array Longest Common Prefix para todo el suffix array.

IMPORTANTE: Debe haberse ejecutado primero suffixArray(), incluido en Suffix Array Init.java

```
static int lcp[] = new int[N_MAX];

static void calculateLCP() {
    for (int i = 0; i < n; i++) rk[sa[i]] = i;
    for (int i = 0, l = 0; i < n; i++) {
        if (rk[i] > 0) {
            int j = sa[rk[i] - 1];
            while (_s[i + l] == _s[j + l]) l++;
            lcp[rk[i]] = l;
            if (l > 0) l--;
        }
    }
}
```

7.5. Suffix Array Longest Common Substring

Busca el substring común mas largo entre dos strings. Retorna un `int[2]`, con el size del substring y uno de los indices del suffix array. Debe ejecutarse previamente `suffixArray()` y `calculateLCP()`

```
// Los substrings deben estar concatenados de la forma
// "string1#string2$", antes de ejecutar suffixArray() y calculateLCS()
// m debe almacenar el size del string2.
```

```
static int[] longestCommonSubstring() {
    int i, ans[] = new int[2];
    ans[0] = -1; ans[1] = 0;
    for (i = 1; i < n; i++) {
        if (((sa[i] < n - m - 1) != (sa[i - 1] < n - m - 1)) && lcp[i] >
            ans[0]) {
            ans[0] = lcp[i]; ans[1] = i;
        }
    }
    return ans;
}
```

7.6. Suffix Array Longest Repeated Substring

Retorna un `int[]` con el size y el indice del suffix array en el cual se encuentra el substring repetido mas largo. Debe ejecutarse primero

`suffixArray()` y `calculateLCP()`.

```
static int[] longestRepeatedSubstring() {
    int ans[] = new int[2]; ans[0] = -1; ans[1] = -1;
    for (int i = 0; i < n; i++) {
        if (ans[0] < lcp[i]) {
            ans[0] = lcp[i]; ans[1] = i;
        }
    }
    return ans;
}
```

7.7. Suffix Array String Matching Boolean

Busca el string p en el string s (definido en `init`), y retorna `true` si se encuentra, o `false` en caso contrario. Debe inicializarse m con el tamaño de p, y debe ejecutarse previamente `suffixArray()` de `Suffix Array Init.java`.

```
static String p;
static int m;

static boolean stringMatching() {
    if (m - 1 > n) return false;
    char [] _p = p.toCharArray();
    int l = 0, h = n - 1, c = 1;
    while (l <= h) {
        c = (l + h) / 2;
        int r = strncmp(_s, sa[c], _p);
        if (r > 0) h = c - 1;
        else if (r < 0) l = c + 1;
        else return true;
    }
    return false;
}
```

7.8. Suffix Array String Matching

Busca el string p en el string s (definido en `init`), y retorna un `int[2]` con el primer y ultimo indice del suffix array que coinciden con la busqueda. Si no se encuentra, retorna `[-1, -1]`. Debe inicializarse m

con el tamaño de p, y debe ejecutarse previamente suffixArray() de Suffix Array Init.java.

```
static String p;
static int m;

static int[] stringMatching() {
    int[] ans = {-1, -1};
    if(m - 1 > n) return ans;
    char [] _p = p.toCharArray();
    int l = 0, h = n - 1, c = 1;
    while (l < h) {
        c = (l + h) / 2;
        if(strncmp(_s, sa[c], _p) >= 0) h = c;
        else l = c + 1;
    }
    if (strncmp(_s, sa[l], _p) != 0) return ans;
    ans[0] = l;
    l = 0; h = n - 1; c = 1;
    while (l < h) {
        c = (l + h) / 2;
        if (strncmp(_s, sa[c], _p) > 0) h = c;
        else l = c + 1;
    }
    if (strncmp(_s, sa[h], _p) != 0) h--;
    ans[1] = h;
    return ans;
}
```

7.9. Suffix Array strncmp

Método utilitario. Necesario para las dos versiones de Matching.

```
static int strncmp(char[] a, int i, char[] b) {
    for (int k = 0; i + k < a.length && k < m - 1; k++) {
        if (a[i + k] != b[k]) return a[i + k] - b[k];
    }
    return 0;
}
```

8. Tips and formulas

8.1. ASCII Table

Caracteres ASCII con sus respectivos valores numéricos.

No.	ASCII	No.	ASCII
0	NUL	16	DLE
1	SOH	17	DC1
2	STX	18	DC2
3	ETX	19	DC3
4	EOT	20	DC4
5	ENQ	21	NAK
6	ACK	22	SYN
7	BEL	23	ETB
8	BS	24	CAN
9	TAB	25	EM
10	LF	26	SUB
11	VT	27	ESC
12	FF	28	FS
13	CR	29	GS
14	SO	30	RS
15	SI	31	US

No.	ASCII	No.	ASCII
32	(space)	48	0
33	!	49	1
34	"	50	2
35	#	51	3
36	\$	52	4
37	%	53	5
38	&	54	6
39	'	55	7
40	(56	8
41)	57	9
42	*	58	:
43	+	59	;
44	,	60	i
45	-	61	=
46	.	62	¿
47	/	63	?

No.	ASCII	No.	ASCII
64	@	80	P
65	A	81	Q
66	B	82	R
67	C	83	S
68	D	84	T
69	E	85	U
70	F	86	V
71	G	87	W
72	H	88	X
73	I	89	Y
74	J	90	Z
75	K	91	[
76	L	92	\
77	M	93]
78	N	94	^
79	O	95	-

No.	ASCII	No.	ASCII
96	`	112	p
97	a	113	q
98	b	114	r
99	c	115	s
100	d	116	t
101	e	117	u
102	f	118	v
103	g	119	w
104	h	120	x
105	i	121	y
106	j	122	z
107	k	123	{
108	l	124	
109	m	125	}
110	n	126	~
111	o	127	

8.2. Formulas

PERMUTACIÓN Y COMBINACIÓN	
Combinación (Coeficiente Binomial)	Número de subconjuntos de k elementos escogidos de un conjunto con n elementos. $\binom{n}{k} = \binom{n}{n-k} = \frac{n!}{k!(n-k)!}$
Combinación con repetición	Número de grupos formados por n elementos, partiendo de m tipos de elementos. $CR_m^n = \binom{m+n-1}{n} = \frac{(m+n-1)!}{n!(m-1)!}$
Permutación	Número de formas de agrupar n elementos, donde importa el orden y sin repetir elementos $P_n = n!$
Permutación múltiple	Elegir r elementos de n posibles con repetición n^r
Permutación con repetición	Se tienen n elementos donde el primer elemento se repite a veces , el segundo b veces , el tercero c veces, ... $PR_n^{a,b,c,\dots} = \frac{P_n}{a!b!c! \dots}$
Permutaciones sin repetición	Número de formas de agrupar r elementos de n disponibles, sin repetir elementos $\frac{n!}{(n-r)!}$
DISTANCIAS	

Continúa en la siguiente columna

Distancia Euclidean	$d_E(P_1, P_2) = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$
Distancia Manhattan	$d_M(P_1, P_2) = x_2 - x_1 + y_2 - y_1 $
CIRCUNFERENCIA Y CÍRCULO	
Considerando r como el radio, α como el ángulo del arco o sector, y (R, r) como radio mayor y menor respectivamente.	
Área	$A = \pi * r^2$
Longitud	$L = 2 * \pi * r$
Longitud de un arco	$L = \frac{2 * \pi * r * \alpha}{360}$
Área sector circular	$A = \frac{\pi * r^2 * \alpha}{360}$
Área corona circular	$A = \pi(R^2 - r^2)$
TRIÁNGULO	
Considerando b como la longitud de la base, h como la altura, letras minúsculas como la longitud de los lados, letras mayúsculas como los ángulos, y r como el radio de circunferencias asociadas.	
Área conociendo base y altura	$A = \frac{1}{2} b * h$

Continúa en la siguiente columna

Área conociendo 2 lados y el ángulo que forman	$A = \frac{1}{2} b * a * \sin(C)$
Área conociendo los 3 lados	$A = \sqrt{p(p-a)(p-b)(p-c)}$ con $p = \frac{a+b+c}{2}$
Área de un triángulo circunscrito a una circunferencia	$A = \frac{abc}{4r}$
Área de un triángulo inscrito a una circunferencia	$A = r(\frac{a+b+c}{2})$
Área de un triángulo equilátero	$A = \frac{\sqrt{3}}{4} a^2$
RAZONES TRIGONOMÉTRICAS	
Considerando un triángulo rectángulo de lados a, b y c , con vértices A, B y C (cada vértice opuesto al lado cuya letra minúscula coincide con el) y un ángulo α con centro en el vértice A . a y b son catetos, c es la hipotenusa:	
$\sin(\alpha) = \frac{\text{cateto opuesto}}{\text{hipotenusa}} = \frac{a}{c}$	

Continúa en la siguiente columna

$\cos(\alpha) = \frac{\text{cateto adyacente}}{\text{hipotenusa}} = \frac{b}{c}$	
$\tan(\alpha) = \frac{\text{cateto opuesto}}{\text{cateto adyacente}} = \frac{a}{b}$	
$\sec(\alpha) = \frac{1}{\cos(\alpha)} = \frac{c}{b}$	
$\csc(\alpha) = \frac{1}{\sin(\alpha)} = \frac{c}{a}$	
$\cot(\alpha) = \frac{1}{\tan(\alpha)} = \frac{b}{a}$	
PROPIEDADES DEL MÓDULO (RESIDUO)	
Propiedad neutro	$(a \% b) \% b = a \% b$
Propiedad asociativa en multiplicación	$(ab) \% c = ((a \% c)(b \% c)) \% c$
Propiedad asociativa en suma	$(a + b) \% c = ((a \% c) + (b \% c)) \% c$
CONSTANTES	
Pi	$\pi = \text{acos}(-1) \approx 3,14159$

Continúa en la siguiente columna

e	$e \approx 2,71828$
Número áureo	$\phi = \frac{1 + \sqrt{5}}{2} \approx 1,61803$

8.3. Sequences

Listado de secuencias mas comunes y como hallarlas.

Estrellas octangulares	0, 1, 14, 51, 124, 245, 426, 679, 1016, 1449, 1990, 2651, ...
	$f(n) = n * (2 * n^2 - 1).$
Euler totient	1, 1, 2, 2, 4, 2, 6, 4, 6, 4, 10, 4, 12, 6,...
	$f(n)$ = Cantidad de números naturales $\leq n$ coprimos con n.
Números de Bell	1, 1, 2, 5, 15, 52, 203, 877, 4140, 21147, 115975, ...
	Se inicia una matriz triangular con $f[0][0] = f[1][0] = 1$. La suma de estos dos se guarda en $f[1][1]$ y se traslada a $f[2][0]$. Ahora se suman $f[1][0]$ con $f[2][0]$ y se guarda en $f[2][1]$. Luego se suman $f[1][1]$ con $f[2][1]$ y se guarda en $f[2][2]$ trasladandose a $f[3][0]$ y así sucesivamente. Los valores de la primera columna contienen la respuesta.
Números de Catalán	1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862, 16796, 58786, ...
	$f(n) = \frac{(2n)!}{(n + 1)!n!}$
Números de Fermat	3, 5, 17, 257, 65537, 4294967297, 18446744073709551617, ...
	$f(n) = 2^{(2^n)} + 1$
Números de Fibonacci	0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, ...
	$f(0) = 0; f(1) = 1; f(n) = f(n - 1) + f(n - 2)$ para $n > 1$

Continúa en la siguiente columna

Números de Lucas	2, 1, 3, 4, 7, 11, 18, 29, 47, 76, 123, 199, 322, ... $f(0) = 2; f(1) = 1; f(n) = f(n-1) + f(n-2)$ para $n > 1$
Números de Pell	0, 1, 2, 5, 12, 29, 70, 169, 408, 985, 2378, 5741, 13860, ... $f(0) = 0; f(1) = 1; f(n) = 2f(n-1) + f(n-2)$ para $n > 1$
Números de Tribonacci	0, 0, 1, 1, 2, 4, 7, 13, 24, 44, 81, 149, 274, 504, ... $f(0) = f(1) = 0; f(2) = 1; f(n) = f(n-1) + f(n-2) + f(n-3)$ para $n > 2$
Números factoriales	1, 1, 2, 6, 24, 120, 720, 5040, 40320, 362880, ... $f(0) = 1; f(n) = \prod_{k=1}^n k$ para $n > 0$.
Números piramidales cuadrados	0, 1, 5, 14, 30, 55, 91, 140, 204, 285, 385, 506, 650, ... $f(n) = \frac{n * (n+1) * (2 * n + 1)}{6}$
Números primos de Mersenne	3, 7, 31, 127, 8191, 131071, 524287, 2147483647, ... $f(n) = 2^{p(n)} - 1$ donde p representa valores primos iniciando en $p(0) = 2$.
Números tetraedrales	1, 4, 10, 20, 35, 56, 84, 120, 165, 220, 286, 364, 455, ... $f(n) = \frac{n * (n+1) * (n+2)}{6}$
Números triangulares	0, 1, 3, 6, 10, 15, 21, 28, 36, 45, 55, 66, 78, 91, 105, ... $f(n) = \frac{n(n+1)}{2}$

Continúa en la siguiente columna

OEIS A000127	1, 2, 4, 8, 16, 31, 57, 99, 163, 256, 386, 562, ... $f(n) = \frac{(n^4 - 6n^3 + 23n^2 - 18n + 24)}{24}$.
Secuencia de Narayana	1, 1, 1, 2, 3, 4, 6, 9, 13, 19, 28, 41, 60, 88, 129, ... $f(0) = f(1) = f(2) = 1; f(n) = f(n-1) + f(n-3)$ para todo $n > 2$.
Secuencia de Silvestre	2, 3, 7, 43, 1807, 3263443, 10650056950807, ... $f(0) = 2; f(n+1) = f(n)^2 - f(n) + 1$
Secuencia de vendedor perezoso	1, 2, 4, 7, 11, 16, 22, 29, 37, 46, 56, 67, 79, 92, 106, ... Equivale al triangular(n) + 1. Máxima número de piezas que se pueden formar al hacer n cortes a un disco. $f(n) = \frac{n(n+1)}{2} + 1$
Suma de los divisores de un número	1, 3, 4, 7, 6, 12, 8, 15, 13, 18, 12, 28, 14, 24, ... Para todo $n > 1$ cuya descomposición en factores primos es $n = p_1^{a_1} p_2^{a_2} \dots p_k^{a_k}$ se tiene que: $f(n) = \frac{p_1^{a_1+1} - 1}{p_1 - 1} * \frac{p_2^{a_2+1} - 1}{p_2 - 1} * \dots * \frac{p_k^{a_k+1} - 1}{p_k - 1}$

8.4. Time Complexities

Aproximación del mayor número n de datos que pueden procesarse para cada una de las complejidades algorítmicas. Tomar esta tabla solo como referencia.

Complexity	n
$O(n!)$	11
$O(n^5)$	50
$O(2^n * n^2)$	18
$O(2^n * n)$	22
$O(n^4)$	100
$O(n^3)$	500
$O(n^2 \log_2 n)$	1.000
$O(n^2)$	10.000
$O(n \log_2 n)$	10^6
$O(n)$	10^8
$O(\sqrt{n})$	10^{16}