

# Índice

<b>1. Mapas</b>	<b>1</b>
<b>2. Sets</b>	<b>2</b>
<b>3. Union-Find</b>	<b>2</b>
<b>4. Grafos</b>	<b>3</b>
4.1. BFS y DFS	3
4.2. Shortest Hop	4
4.3. Ordenamiento topológico	4
4.4. Componentes fuertemente conexas (Algoritmo de Tarjan)	5
4.5. Puntos de articulación	6
4.6. Puentes	7
4.7. Minimum Spanning Tree (Algoritmo de Kruskal)	7
4.8. Algoritmo de Dijkstra	8
4.9. Algoritmo de Floyd-Warshall	9
4.10. Máximo flujo y mínimo corte	10
<b>5. KMP</b>	<b>11</b>
<b>6. Programación dinámica</b>	<b>12</b>
6.1. Longest Increasing Subsequence	12
6.2. Longest Common Subsequence	12
6.3. Coin Change Problem	13
6.4. Edit Distance	13
6.5. El problema de la mochila (Knapsack)	13
<b>7. Teoría de números</b>	<b>13</b>
7.1. Algoritmo de Euclides	13
7.2. Verificar si un número es primo	14
7.3. Criba de Eratóstenes	14
7.4. Factorización prima de un número	14
7.5. Fórmulas	15
7.5.1. Cantidad de divisores	15
7.5.2. Suma de divisores	15
7.5.3. Función $\varphi$ de Euler	16
<b>8. Combinatoria</b>	<b>16</b>
8.1. Permutaciones	16
8.2. Subconjuntos	16
8.3. Coeficientes binomiales	16
8.4. Coeficientes multinomiales	17

8.5. Multiconjuntos	17
8.6. Particiones	17
8.7. Desarreglos	17
8.8. Números de Catalan	17

## 9. Otros 18

9.1. Ordenamiento de Arrays y Listas (ascendente y descendente)	18
9.2. Cola de Prioridad	18
9.3. Interfaz Comparable	18
9.4. Imprimir números decimales redondeados	19
9.5. BufferedReader y BufferedWriter	19

## 1. Mapas

Estructura de datos que guarda pares (*clave*, *valor*). El HashMap no pone las claves en ningún orden en particular. TreeMap ordena las claves de acuerdo a su orden natural. LinkedHashMap pone las claves en el orden en que se ingresen.

Las operaciones `.put()`, `.get()` y `.containsKey()` son  $O(1)$  en HashMap y LinkedHashMap, y  $O(\log n)$  en TreeMap.

Ejemplo: Contar cuántas veces aparece cada palabra en un String.

```
1 public static void main(String args[]) {
2     HashMap<String, Integer> map = new HashMap<String,
3         Integer>();
4     //TreeMap<String, Integer> map = new TreeMap<String,
5         Integer>();
6     //LinkedHashMap<String, Integer> map = new
7         LinkedHashMap<String, Integer>();
8
9     String s = "tres_tristes_tigres_tragaban_trigo_en_un
10         _trigal_en_tres_tristes_trastos";
11     String palabras[] = s.split("_");
12
13     for(int i=0; i<palabras.length; i++){
14         if(!map.containsKey(palabras[i])){
15             map.put(palabras[i], 1);
16         }else{
17             map.put(palabras[i], map.get(palabras[i])+1);
18         }
19     }
20
21     //Obtener un elemento
```

```

18 System.out.println(map.get("tres"));
19
20 //Recorrer el mapa
21 for(Entry<String, Integer> e : map.entrySet()){
22     System.out.println(e.getKey() + "↵↵" + e.getValue()
23         ↵↵());
24 }

```

## 2. Sets

Estructura de datos que actúa como “bolsa” donde se almacenan elementos, pero no puede almacenar elementos duplicados.

En `HashSet` `.add()` y `.contains()` son  $O(1)$ , mientras que en `TreeSet` son  $O(\log n)$ . Sin embargo, en el `TreeSet` los elementos quedan ordenados.

```

1 public static void main(String[] args) {
2     HashSet<String> hs = new HashSet<String>();
3     //TreeSet<String> ts = new TreeSet<String>();
4
5     hs.add("Hola");
6     hs.add("Hola");
7     hs.add("Mundo");
8     //Imprime 2, porque no se aceptan repetidos
9     System.out.println(hs.size());
10
11     //Recorrido
12     for(String s: hs){
13         System.out.println(s);
14     }
15 }

```

## 3. Union-Find

Estructura de datos que soporta las siguientes operaciones eficientemente:

- Unir los conjuntos de los elementos  $p, q$
- Determinar si los elementos  $p, q$  pertenecen al mismo conjunto o no

```

1 class UnionFind{
2     private int[] parent, size;

```

```

3     private int components;
4
5     // n = Numero de nodos
6     public UnionFind(int n){
7         components = n;
8         parent = new int[n];
9         size = new int[n];
10        for(int i=0; i<n; i++){
11            parent[i] = i;
12            size[i] = 1;
13        }
14    }
15
16    private int root(int p){
17        while(p != parent[p]){
18            parent[p] = parent[parent[p]];
19            p = parent[p];
20        }
21        return p;
22    }
23
24    //Une los nodos p,q
25    public void union(int p, int q){
26        int rootP = root(p);
27        int rootQ = root(q);
28        if(rootP != rootQ){
29            if(size[rootP] < size[rootQ]){
30                parent[rootP] = rootQ;
31                size[rootQ] = size[rootQ] + size[rootP];
32            }else{
33                parent[rootQ] = rootP;
34                size[rootP] = size[rootP] + size[rootQ];
35            }
36            components--;
37        }
38    }
39
40    //Retorna true si p,q estan conectados
41    public boolean connected(int p, int q){
42        return root(p) == root(q);
43    }
44
45    //Retorna el numero de componentes conexas

```

```

46 public int getComponents() {
47     return components;
48 }
49 }
50
51 class Main {
52     public static void main(String[] args) {
53         UnionFind uf = new UnionFind(5);
54         uf.union(0, 2);
55         uf.union(1, 0);
56         uf.union(3, 4);
57
58         //El numero de componentes es
59         int comp = uf.getComponents();
60
61         //Dos nodos estan conectados?
62         boolean connected = uf.connected(0, 3);
63     }
64 }

```

## 4. Grafos

### 4.1. BFS y DFS

Recorren un grafo a partir de un nodo origen y visitan todos los nodos alcanzables desde éste. Ambos algoritmos tienen una complejidad de  $O(n + m)$  donde  $n$  es el número de nodos y  $m$  es el número de aristas del grafo. El siguiente ejemplo está con DFS pero funciona igual con BFS.

```

1 static ArrayList<Integer> g[];
2 static boolean seen[];
3
4 public static void main(String[] args) {
5     Scanner sc = new Scanner(System.in);
6     int n = sc.nextInt();
7
8     seen = new boolean[n];
9     g = new ArrayList[n];
10    for(int i = 0; i < n; i++){
11        g[i] = new ArrayList<Integer>();
12    }
13
14    while(sc.hasNextInt()){

```

```

15        int u = sc.nextInt();
16        int v = sc.nextInt();
17        g[u].add(v);
18        // Si el grafo es no-dirigido, tambien se agrega
19        //      ↪ arista de v a u
20        g[v].add(u);
21    }
22
23    //Visita solo los nodos que son alcanzables desde el
24    //      ↪ nodo 's'
25    int s = 0;
26    dfs(s);
27
28    //Con el vector 'seen' vemos cuales son estos nodos
29    for(int i=0; i<n; i++){
30        if(seen[i]){
31            // 'i' es alcanzable desde 's'
32        }
33    }
34
35    //Si queremos visitar todos los nodos
36    for(int u=0; u<n; u++){
37        if(!seen[u]){
38            //Si no hemos visitado 'u', hacer DFS en 'u'
39            dfs(u);
40        }
41    }
42
43    private static void dfs(int u){
44        seen[u] = true;
45        int len = g[u].size();
46        for(int i=0; i<len; i++){
47            int v = g[u].get(i);
48            if(!seen[v]){
49                dfs(v);
50            }
51        }
52    }
53
54    private static void bfs(int u){
55        seen[u] = true;
56        Queue<Integer> q = new LinkedList<Integer>();

```

```

56 q.add(u);
57 while(!q.isEmpty()){
58     u = q.poll();
59     int len = g[u].size();
60     for(int i=0; i<len; i++){
61         int v = g[u].get(i);
62         if(!seen[v]){
63             seen[v] = true;
64             q.add(v);
65         }
66     }
67 }
68 }

```

## 4.2. Shortest Hop

Modificación de BFS que calcula el camino más corto desde un nodo origen  $s$  a todos los demás. Sólo funciona cuando el peso de todas las aristas es 1. Su complejidad es la misma de BFS:  $O(n + m)$ .

```

1  static ArrayList<Integer> g[];
2  static boolean seen[];
3  static int dist[];
4
5  public static void main(String[] args) {
6      int n = 10;
7
8      seen = new boolean[n];
9      dist = new int[n];
10     g = new ArrayList[n];
11     for(int i=0; i<n; i++){
12         g[i] = new ArrayList<Integer>();
13     }
14
15     int s = 0;
16     shortestHop(s);
17     //Despues de llamar este metodo, en dist[i] esta la
18     //    ↪ distancia mas corta (s,i)
19 }
20
21 public static void shortestHop(int u){
22     int n = g.length;

```

```

23 //Distancia "infinita" hacia todos los nodos
24 Arrays.fill(dist, Integer.MAX_VALUE);
25 //Distancia 0 hacia el nodo de origen
26 dist[u] = 0;
27
28 //BFS "modificado"
29 seen[u] = true;
30 Queue<Integer> q = new LinkedList<Integer>();
31 q.add(u);
32 while(!q.isEmpty()){
33     u = q.poll();
34     int len = g[u].size();
35     for(int i=0; i<len; i++){
36         int v = g[u].get(i);
37         if(!seen[v]){
38             seen[v] = true;
39             q.add(v);
40             //Lo unico que cambia es que se calcula el
41             //    ↪ dist[v]
42             dist[v] = dist[u] + 1;
43         }
44     }
45 }

```

## 4.3. Ordenamiento topológico

Todo grafo dirigido acíclico (DAG) tiene un ordenamiento topológico. Esto significa que para todas las aristas  $(u, v)$ ,  $u$  aparece en el ordenamiento antes que  $v$ . Visualmente es como si se pusieran todos los nodos en línea recta y todas las aristas fueran de izquierda a derecha, ninguna de derecha a izquierda.

El algoritmo para hallar dicho ordenamiento es una modificación de DFS y complejidad es la misma:  $O(n + m)$ . El método retorna falso si detecta un ciclo en el grafo, ya que en este caso no existe ordenamiento topológico posible.

```

1  static ArrayList<Integer> g[];
2  static int seen[];
3  static LinkedList<Integer> topoSort;
4
5  public static void main(String[] args) {
6      int n = 10;
7
8      seen = new int[n];

```

```

9      topoSort = new LinkedList<Integer>();
10     g = new ArrayList[n];
11     for(int i = 0; i < n; i++){
12         g[i] = new ArrayList<Integer>();
13     }
14
15     boolean sinCiclo = true;
16
17     //Es necesario hacer el ciclo para visitar todos los
18     ↪ nodos
19     for(int u=0; u<n; u++){
20         if(seen[u] == 0){
21             sinCiclo = sinCiclo && topoDfs(u);
22         }
23
24         if(sinCiclo){
25             //La lista 'topoSort' contiene los nodos en su
26             ↪ orden topologico
27         }else{
28             //Hay un ciclo
29         }
30
31     private static boolean topoDfs(int u){
32         //DFS "modificado" para hacer ordenamiento
33         ↪ topologico
34         //Se marca 'u' como 'gris'
35         seen[u] = 1;
36         int len = g[u].size();
37         boolean sinCiclo = true;
38         for(int i=0; i<len; i++){
39             int v = g[u].get(i);
40             if(seen[v] == 0){
41                 sinCiclo = sinCiclo && topoDfs(v);
42             }else if(seen[v] == 1){
43                 //Hay un ciclo, retorna falso
44                 sinCiclo = false;
45             }
46         }
47         //Se agrega el nodo 'u' al inicio de la lista y se
48         ↪ marca 'negro'
49         seen[u] = 2;

```

```

48     topoSort.addFirst(u);
49     return sinCiclo;
50 }

```

#### 4.4. Componentes fuertemente conexas (Algoritmo de Tarjan)

Calcula la componente fuertemente conexas a la que pertenece cada nodo de un grafo dirigido. Si dos nodos  $u, v$  están en la misma componente, significa que existe un camino de  $u$  a  $v$  y uno de  $v$  a  $u$ . Su complejidad es  $O(n + m)$ .

```

1  static ArrayList<Integer> g[];
2  static boolean[] seen, stackMember;
3  static int[] disc, low, scc;
4  static Stack<Integer> st;
5  static int time, component;
6
7  public static void main(String[] args) {
8      int n = 10;
9
10     seen = new boolean[n];
11     stackMember = new boolean[n];
12     disc = new int[n];
13     low = new int[n];
14     scc = new int[n];
15     st = new Stack<Integer>();
16     time = 0;
17     component = 0;
18     g = new ArrayList[n];
19     for(int i = 0; i < n; i++){
20         g[i] = new ArrayList<Integer>();
21     }
22
23     for(int u=0; u<n; u++){
24         if(!seen[u]){
25             tarjan(u);
26         }
27     }
28     //scc[i]==x significa que 'i' pertenece a la
29     ↪ componente 'x'
30
31     private static void tarjan(int u){

```

```

32  seen[u] = true;
33  st.add(u);
34  stackMember[u] = true;
35  disc[u] = time;
36  low[u] = time;
37  time++;
38
39  int len = g[u].size();
40  for(int i=0; i<len; i++){
41      int v = g[u].get(i);
42      if(!seen[v]){
43          tarjan(v);
44          low[u] = Math.min(low[u], low[v]);
45      }else if(stackMember[v]){
46          low[u] = Math.min(low[u], disc[v]);
47      }
48  }
49
50  if(low[u] == disc[u]){
51      int w;
52      do{
53          w = st.pop();
54          stackMember[w] = false;
55          scc[w] = component;
56      }while(w != u);
57      component++;
58  }
59  }

```

#### 4.5. Puntos de articulación

Halla los puntos de articulación de un grafo. Un punto de articulación es un nodo del grafo que si se quitara causaría que el grafo se “desconectara”. Si el grafo no era conexo en un principio, un punto de articulación es un nodo que, si se quitara, incrementaría el número de componentes conexas. La complejidad del algoritmo es  $O(n + m)$ .

```

1  static ArrayList<Integer> g[];
2  static boolean[] seen, ap;
3  static int[] disc, low, parent;
4  static int time;
5
6  public static void main(String[] args) {

```

```

7  int n = 10;
8
9  g = new ArrayList[n];
10 seen = new boolean[n];
11 ap = new boolean[n];
12 disc = new int[n];
13 low = new int[n];
14 parent = new int[n];
15 time = 0;
16 for(int i = 0; i < n; i++){
17     g[i] = new ArrayList<Integer>();
18     parent[i] = -1;
19 }
20
21 for(int u=0; u<n; u++){
22     if(!seen[u]){
23         articulationPoints(u);
24     }
25 }
26 //Si ap[i]==true, 'i' es un punto de articulacion
27 }
28
29 private static void articulationPoints(int u){
30     seen[u] = true;
31     disc[u] = time;
32     low[u] = time;
33     time++;
34     int children = 0;
35
36     int len = g[u].size();
37     for(int i=0; i<len; i++){
38         int v = g[u].get(i);
39         if(!seen[v]){
40             children++;
41             parent[v] = u;
42             articulationPoints(v);
43             low[u] = Math.min(low[u], low[v]);
44             if(parent[u] == -1 && children > 1){
45                 ap[u] = true;
46             }else if(parent[u] != -1 && low[v] >= disc[u]){
47                 ap[u] = true;
48             }
49         }else if(v != parent[u]){

```

```

50         low[u] = Math.min(low[u], disc[v]);
51     }
52 }
53 }

```

## 4.6. Puentes

Halla los puentes de un grafo. Un puente es una arista del grafo que si se quitara causaría que el grafo se “desconectara”. Si el grafo no era conexo en un principio, un puente es una arista que, si se quitara, incrementaría el número de componentes conexas. La complejidad del algoritmo es  $O(n + m)$ .

```

1 class Bridge {
2     public int u, v;
3     public Bridge(int u, int v){
4         this.u = u;
5         this.v = v;
6     }
7 }
8
9 public class GraphBridges {
10
11     static ArrayList<Integer> g[];
12     static boolean[] seen;
13     static int[] disc, low, parent;
14     static int time;
15     static ArrayList<Bridge> bridgeEdges;
16
17     public static void main(String[] args) {
18         int n = 10;
19
20         g = new ArrayList[n];
21         seen = new boolean[n];
22         disc = new int[n];
23         low = new int[n];
24         parent = new int[n];
25         time = 0;
26         bridgeEdges = new ArrayList<Bridge>();
27         for(int i = 0; i < n; i++){
28             g[i] = new ArrayList<Integer>();
29             parent[i] = -1;
30         }
31

```

```

32         for(int u=0; u<n; u++){
33             if(!seen[u]){
34                 bridges(u);
35             }
36         }
37         // 'bridgeEdges' contiene objetos tipo Bridge que
38         //     ↪ indican que la arista u,v es un puente
39     }
40
41     private static void bridges(int u){
42         seen[u] = true;
43         disc[u] = time;
44         low[u] = time;
45         time++;
46
47         int len = g[u].size();
48         for(int i=0; i<len; i++){
49             int v = g[u].get(i);
50             if(!seen[v]){
51                 parent[v] = u;
52                 bridges(v);
53                 low[u] = Math.min(low[u], low[v]);
54                 if(low[v] > disc[u]){
55                     Bridge b = new Bridge(u, v);
56                     bridgeEdges.add(b);
57                 }
58             } else if(v != parent[u]){
59                 low[u] = Math.min(low[u], disc[v]);
60             }
61         }
62     }

```

## 4.7. Minimum Spanning Tree (Algoritmo de Kruskal)

Halla el árbol de cubrimiento mínimo de un grafo no-dirigido y conexo. Si no está garantizado de antemano que el grafo sea conexo, hay que verificarlo antes de correr este algoritmo.

Utiliza la estructura de datos Union-Find discutida anteriormente. El grafo debe ser representado como una lista de objetos tipo Arista. Tiene una complejidad de  $O(m \log n)$ .

```

1 //Se necesita implementar tambien la clase UnionFind

```

```

2
3 class Arista implements Comparable<Arista>{
4     public int u, v, costo;
5     public Arista(int u, int v, int costo){
6         this.u = u;
7         this.v = v;
8         this.costo = costo;
9     }
10    public int compareTo(Arista o) {
11        return this.costo - o.costo;
12    }
13}
14
15 public class Kruskal {
16
17     public static void main(String[] args) {
18         int n = 10; //Cantidad de nodos del grafo
19         ArrayList<Arista> aristas = new ArrayList<Arista>();
20         int mst = kruskal(aristas, n);
21     }
22
23     public static int kruskal(ArrayList<Arista> aristas,
24         ↪ int n){
25         Collections.sort(aristas);
26
27         UnionFind uf = new UnionFind(n);
28         int costoMST = 0;
29         int i = 0;
30         while(uf.getComponents() != 1){
31             Arista a = aristas.get(i);
32             if(!uf.connected(a.u, a.v)){
33                 uf.union(a.u, a.v);
34                 costoMST += a.costo;
35             }
36             i++;
37         }
38         return costoMST;
39     }
40 }

```

## 4.8. Algoritmo de Dijkstra

Halla la distancia más corta desde un nodo origen *src* hacia todos los demás nodos. Funciona con grafos dirigidos y no dirigidos, siempre y cuando los pesos de las aristas sean no-negativos. Se debe representar el grafo tanto en lista como en matriz de adyacencia. Su complejidad es  $O(m + n \log n)$ .

```

1 class Nodo implements Comparable<Nodo>{
2     int id, distancia;
3     public Nodo(int id, int distancia){
4         this.id = id;
5         this.distancia = distancia;
6     }
7     public int compareTo(Nodo o) {
8         return this.distancia-o.distancia;
9     }
10}
11
12 public class Dijkstra {
13
14     static ArrayList<Integer> g[];
15     static int[][] p;
16     static int[] distancias, padre;
17     static boolean[] visitado;
18     static PriorityQueue<Nodo> proximo;
19
20     public static void main(String[] args) {
21         int n = 8;
22
23         g = new ArrayList[n];
24         p = new int[n][n];
25         distancias = new int[n];
26         padre = new int[n];
27         visitado = new boolean[n];
28         proximo = new PriorityQueue<Nodo>();
29         for (int i=0; i<n; i++) {
30             g[i] = new ArrayList<Integer>();
31             distancias[i] = Integer.MAX_VALUE;
32         }
33
34         int src = 0;
35         dijkstra(src);
36

```



```

37 //El vector 'nodos' contiene la menor distancia de '
    ↪ src' a todos los nodos
38 //Por ejemplo, la menor distancia de 'src' a 4 es:
39 int menorDist = distancias[4];
40
41 //Para hallar el camino como tal entre 'src' y un
    ↪ nodo
42 LinkedList<Integer> camino = new LinkedList<Integer>
    ↪ >();
43 int r = 4;
44 camino.add(r);
45 while(r != src){
46     r = padre[r];
47     camino.addFirst(r);
48 }
49 }
50
51 public static void dijkstra(int src){
52     distancias[src] = 0;
53     proximo.add(new Nodo(src, 0));
54     padre[src] = src;
55
56     while(!proximo.isEmpty()) {
57         Nodo u = proximo.poll();
58         if(!visitado[u.id]){
59             visitado[u.id] = true;
60             int len = g[u.id].size();
61             for (int i=0; i<len; i++) {
62                 int v = g[u.id].get(i);
63                 if(u.distancia + p[u.id][v] < distancias[v]){
64                     distancias[v] = u.distancia + p[u.id][v];
65                     proximo.add(new Nodo(v, distancias[v]));
66                     padre[v] = u.id;
67                 }
68             }
69         }
70     }
71 }
72 }

```

## 4.9. Algoritmo de Floyd-Warshall

Halla la distancia más corta desde todos los nodos hacia todos los demás. El grafo debe estar representado en matriz de adyacencia, y puede tener aristas con peso negativo. Sin embargo, no puede tener ciclos de peso negativo. En caso de que exista un ciclo negativo, el algoritmo lo detectará. Si todas las aristas son no-negativas, se puede omitir la comprobación del ciclo negativo.

La matriz de adyacencia debe armarse así:

$$\text{grafo}(i,j) = \begin{cases} 0 & \text{si } i = j \\ c_{i,j} & \text{si existe una arista de } i \text{ a } j \text{ con costo } c_{i,j} \\ \infty & \text{si } i \neq j \text{ y no existe arista de } i \text{ a } j \end{cases}$$

La complejidad del algoritmo es  $O(n^3)$ .

```

1 public static void main(String[] args) {
2     int n = 5;
3     int grafo[][] = new int[n][n];
4     int A[][] = new int[n][n];
5
6     for(int i=0; i<n; i++){
7         for(int j=0; j<n; j++){
8             if(i == j){
9                 grafo[i][j] = 0;
10            }else{
11                grafo[i][j] = Integer.MAX.VALUE;
12            }
13        }
14    }
15
16    //Tener la matriz del grafo llena antes de hacer
    ↪ esto
17    for(int i=0; i<n; i++){
18        for(int j=0; j<n; j++){
19            A[i][j] = grafo[i][j];
20        }
21    }
22
23    for(int k=0; k<n; k++){
24        for(int i=0; i<n; i++){
25            for(int j=0; j<n; j++){
26                int option1 = A[i][j];
27                int option2;
28                if(A[i][k] == Integer.MAX.VALUE || A[k][j] ==
    ↪ Integer.MAX.VALUE){

```

```

29         option2 = Integer.MAX_VALUE;
30     }else{
31         option2 = A[i][k] + A[k][j];
32     }
33     A[i][j] = Math.min(option1, option2);
34 }
35 }
36 }
37
38 //Verificar si el grafo tiene un ciclo negativo
39 boolean negativeCycle = false;
40 for(int i=0; i<n && !negativeCycle; i++){
41     if(A[i][i] < 0){
42         negativeCycle = true;
43     }
44 }
45 }

```

#### 4.10. Máximo flujo y mínimo corte

Halla el máximo flujo que se puede emitir desde un origen  $s$  hacia un destino  $t$ , dado que los enlaces (aristas) tienen una capacidad dada.

Es el algoritmo de Edmonds-Karp, una implementación del método de Ford-Fulkerson que usa BFS para hallar los caminos en el grafo residual. Su complejidad es de  $O(nm^2)$ .

Si hay varios orígenes  $\{s_1, s_2, \dots\}$ , se pone un “super origen”  $s$ , y se agregan aristas  $(s, s_i)$  con capacidad infinita. Análogamente, si hay varios destinos  $\{t_1, t_2, \dots\}$ , se agrega un “super destino”  $t$  y se agregan aristas  $(t_i, t)$  con capacidad infinita.

El máximo flujo es igual al mínimo corte (esto es un teorema). Por ende, este algoritmo halla también el mínimo costo de cortar aristas de manera que  $s$  y  $t$  queden desconectados.

```

1  static ArrayList<Integer> g[];
2  static int[] parent;
3  static int[][] cap;
4  static int[][] flow;
5
6  public static void main(String[] args) {
7      Scanner sc = new Scanner(System.in);
8      int n = sc.nextInt();
9      int m = sc.nextInt();
10     g = new ArrayList[n];

```

```

11     for(int i = 0; i < n; i++){
12         g[i] = new ArrayList<Integer>();
13     }
14     parent = new int[n];
15     cap = new int[n][n];
16     flow = new int[n][n];
17
18     for(int i=0; i<m; i++){
19         int u = sc.nextInt();
20         int v = sc.nextInt();
21         int c = sc.nextInt();
22         g[u].add(v);
23         // Siempre se agrega esta arista, aunque el grafo
24         // sea dirigido
25         g[v].add(u);
26         cap[u][v] = c;
27         // La siguiente linea solo se agrega si el grafo
28         // es no-dirigido
29         // cap[v][u] = c;
30     }
31
32     int s = sc.nextInt();
33     int t = sc.nextInt();
34     int max = maxFlow(s, t);
35 }
36
37 public static int maxFlow(int s, int t) {
38     int ans = 0;
39     while(true) {
40         Arrays.fill(parent, -1);
41         Queue<Integer> q = new LinkedList<Integer>();
42         q.add(s);
43
44         while(!q.isEmpty()) {
45             int u = q.poll();
46             if(u == t) break;
47
48             int len = g[u].size();
49             for(int i=0; i<len; i++){
50                 int v = g[u].get(i);
51                 if(parent[v] == -1 && cap[u][v] - flow[u][v] >
52                     // 0){
53                     q.add(v);
54             }
55         }
56     }
57 }

```

```

51         parent[v] = u;
52     }
53 }
54 }
55 if(parent[t] == -1) break;
56
57 int bottleneck = Integer.MAX_VALUE;
58 int end = t;
59 while(end != s) {
60     int start = parent[end];
61     bottleneck = Integer.min(bottleneck, cap[start][
        ↪ end] - flow[start][end]);
62     end = start;
63 }
64
65 end = t;
66 while(end != s) {
67     int start = parent[end];
68     flow[start][end] += bottleneck;
69     flow[end][start] = -flow[start][end];
70     end = start;
71 }
72
73 ans += bottleneck;
74 }
75 return ans;
76 }

```

## 5. KMP

Algoritmo para buscar una cadena *pattern* dentro de una cadena *text*. Su complejidad es de  $O(m+n)$  donde  $m$  es la longitud de *text* y  $n$  es la longitud de *pattern*. Retorna la posición donde inicia la primera ocurrencia de *text* dentro de *pattern*, o -1 si no existe.

```

1  private static int[] computeTemporaryArray(String
    ↪ pattern){
2      int lps[] = new int[pattern.length()];
3      int index = 0;
4      int i = 1;
5      while(i < pattern.length()){
6          if(pattern.charAt(i) == pattern.charAt(index)){

```

```

7          lps[i] = index + 1;
8          index++;
9          i++;
10         }else{
11             if(index != 0){
12                 index = lps[index-1];
13             }else{
14                 lps[i] = 0;
15                 i++;
16             }
17         }
18     }
19     return lps;
20 }
21
22 public static int KMP(String text, String pattern){
23     int lps[] = computeTemporaryArray(pattern);
24     int i=0;
25     int j=0;
26     while(i < text.length() && j < pattern.length()){
27         if(text.charAt(i) == pattern.charAt(j)){
28             i++;
29             j++;
30         }else{
31             if(j!=0){
32                 j = lps[j-1];
33             }else{
34                 i++;
35             }
36         }
37     }
38     if(j == pattern.length()){
39         return i-j;
40     }
41     return -1;
42 }
43
44 public static void main(String[] args) {
45     String text = "ABABABABC";
46     String pattern = "BABABC";
47     int index = KMP(text, pattern);
48 }

```

## 6. Programación dinámica

### 6.1. Longest Increasing Subsequence

Halla la longitud de la subsecuencia creciente más larga que hay en un vector (o String). También halla los elementos que pertenecen a dicha subsecuencia, por si llega a ser necesario. Su complejidad es  $O(n^2)$ .

```
1 public static void main(String[] args) {
2     int array[] = {3, 4, -1, 0, 6, 2, 3};
3     int n = array.length;
4
5     int T[] = new int[n];
6     int previous[] = new int[n];
7     for(int i=0; i<n; i++){
8         T[i] = 1;
9         previous[i] = i;
10    }
11
12    for(int i=1; i<n; i++){
13        for(int j=0; j<i; j++){
14            if(array[i] > array[j]){
15                if(T[j] + 1 > T[i]){
16                    T[i] = T[j] + 1;
17                    previous[i] = j;
18                }
19            }
20        }
21    }
22
23    int maxIndex = 0;
24    for(int i=0; i<n; i++){
25        if(T[i] > T[maxIndex]){
26            maxIndex = i;
27        }
28    }
29
30    //Longitud de la LIS
31    int lisLength = T[maxIndex];
32
33    //La subsecuencia como tal
34    int t = maxIndex;
35    int newT = maxIndex;
```

```
36    LinkedList<Integer> subsequence = new LinkedList<
37        ↪ Integer>();
38    do{
39        t = newT;
40        subsequence.addFirst(array[t]);
41        newT = previous[t];
42    }while(t != newT);
43 }
```

### 6.2. Longest Common Subsequence

Halla la longitud de la subsecuencia común más larga entre dos Strings (o vectores). También halla los elementos que pertenecen a dicha subsecuencia, por si llega a ser necesario. Su complejidad es  $O(mn)$ , donde  $m$  y  $n$  son las longitudes de los Strings.

```
1 public static void main(String[] args) {
2     String str1 = "ABCDGHLQR";
3     String str2 = "AEDPHR";
4
5     char x[] = str1.toCharArray();
6     char y[] = str2.toCharArray();
7     int T[][] = new int[x.length + 1][y.length + 1];
8
9     for(int i=1; i<=x.length; i++){
10        for(int j=1; j<=y.length; j++){
11            if(x[i-1] == y[j-1]){
12                T[i][j] = T[i-1][j-1] + 1;
13            }else{
14                T[i][j] = Math.max(T[i][j-1], T[i-1][j]);
15            }
16        }
17    }
18
19    //Longitud de la LCS
20    int lcsLength = T[x.length][y.length];
21
22    //La LCS como tal
23    int i = x.length;
24    int j = y.length;
25    StringBuilder sb = new StringBuilder();
26    while(i > 0 && j > 0){
27        if(T[i][j] == T[i-1][j]){
```

```

28     i--;
29 }else if(T[i][j] == T[i][j-1]){
30     j--;
31 }else{
32     sb.append(x[i-1]);
33     i--;
34     j--;
35 }
36 }
37 String lcs = sb.reverse().toString();
38 }

```

### 6.3. Coin Change Problem

### 6.4. Edit Distance

### 6.5. El problema de la mochila (Knapsack)

Se tiene una mochila con capacidad  $W$ , y  $n$  items con un peso  $w_i$  y un valor  $v_i$  cada uno. Se quiere hallar el conjunto de items tal que la suma de sus pesos no exceda  $W$ , y que la suma de sus valores sea lo más grande posible. Su complejidad es  $O(nW)$ . Es posible indicar cuál es el mayor valor posible, y con un ciclo adicional, indicar exactamente cuáles items se seleccionaron.

```

1  public static void main(String[] args) {
2      int n = 4;
3      int W = 8;
4      int values[] = {15, 10, 9, 5};
5      int weights[] = {1, 5, 3, 4};
6
7      //Tener cuidado: En la matriz los items se numeran
      //  ↪ 1...n y la capacidad de la mochila 1...W
8      int A[][] = new int[n+1][W+1];
9
10     //Aca se resuelve el problema. Asegurarse de tener
      //  ↪ los values y weights
11     for (int i=1; i<=n; i++) {
12         for (int x=0; x<=W; x++) {
13             if (weights[i-1] > x) {
14                 A[i][x] = A[i-1][x];
15             } else {
16                 int p = A[i-1][x];
17                 int q = A[i-1][x-weights[i-1]] + values[i-1];
18                 A[i][x] = (p > q) ? p : q;

```

```

19     }
20 }
21 }
22
23 //El valor maximo que se puede obtener es A[n][W]
24 int solution = A[n][W];
25
26 //Si se quiere determinar cuales items se incluyeron
27 boolean chosen[] = new boolean[n];
28 int i = n;
29 int j = W;
30
31 while(i>0){
32     if(A[i][j] == A[i-1][j]){
33         i--;
34     }else{
35         chosen[i-1] = true;
36         i--;
37         j = j-weights[i];
38     }
39 }
40 //Si chosen[i]==true es porque i se incluye
41 }

```

## 7. Teoría de números

### 7.1. Algoritmo de Euclides

Se utiliza para hallar el máximo común divisor (MCD) entre dos números. También se puede usar para hallar el mínimo común múltiplo (MCM).

```

1  public static int mcd(int a, int b){
2      while(b != 0){
3          int t = b;
4          b = a % b;
5          a = t;
6      }
7      return a;
8  }
9
10 //Dividir primero para evitar overflow en a*b
11 public static int mcm(int a, int b){
12     return a * (b / mcd(a, b));

```

```
13 }
```

## 7.2. Verificar si un número es primo

Dependiendo del problema, puede que nos sirva la forma “fuerza bruta”. Esta forma tiene una complejidad de  $O(\sqrt{n})$ . Sin embargo, si tenemos números de más de 64 bits (que no caben en un *long*) ya esta forma no es viable.

La clase `BigInteger` provee un método probabilístico para determinar si un número es primo. Si el número es compuesto, el método retorna *false* siempre. Si el método retorna *true*, hay una probabilidad de  $1 - \frac{1}{2^x}$  de que el número sea primo, donde  $x$  es un parámetro que se le pasa a la función. Generalmente un valor de  $x = 10$  está bien.

```
1 public static boolean isPrime(int x){
2     if(x == 1) return false;
3     if(x == 2) return true;
4     if(x % 2 == 0) return false;
5
6     int s = (int) Math.ceil(Math.sqrt(x));
7
8     for(int i=3; i<=s; i+=2){
9         if(x % i == 0) return false;
10    }
11
12    return true;
13 }
14
15 public static void main(String[] args) {
16     isPrime(63); //true
17     isPrime(99); //false
18
19     //De hecho estos si caben en un int pero los pongo
20     //    como ejemplo
21     BigInteger a = new BigInteger("104723");
22     BigInteger b = new BigInteger("104727");
23     a.isProbablePrime(10); //true
24     b.isProbablePrime(10); //false
25 }
```

## 7.3. Criba de Eratóstenes

Algoritmo para hallar los números primos menores o iguales a  $n$ . Su complejidad es  $O(n \log \log n)$ .

```
1 public static ArrayList<Integer> criba(int n){
2     boolean marked[] = new boolean[n+1];
3     marked[0] = true;
4     marked[1] = true;
5     ArrayList<Integer> primos = new ArrayList<Integer>()
6     //    ;
7
8     for(int i=2; i<=n; i++){
9         if(!marked[i]){
10            primos.add(i);
11            //OJO: Si esta teniendo problemas de overflow,
12            //    cambie la siguiente linea por j = 2*i
13            int j = i*i;
14            while(j <= n){
15                marked[j] = true;
16                j = j+i;
17            }
18        }
19    }
20    return primos;
21 }
```

## 7.4. Factorización prima de un número

Se busca expresar un número  $n$  como una multiplicación de factores primos, de la forma:

$$n = \prod p_i^{a_i} = p_1^{a_1} \cdot p_2^{a_2} \cdot p_3^{a_3} \dots p_k^{a_k}$$

Previamente se debe hacer una Criba de Eratóstenes modificada. Verifique en la especificación de la entrada del problema cuál es el máximo número  $x$  que tendrá que factorizar, y haga la Criba hasta  $\sqrt{x}$ .

El método retorna un `HashMap` donde la *clave* es el factor primo  $p_i$  y el *valor* su multiplicidad  $a_i$ . Se puede modificar fácilmente para retornar una lista de todos los factores, o retornar la cantidad de factores.

Con algunas modificaciones, puede funcionar más o menos hasta  $n = 10^{12}$ .

```
1 static int menorFactor[];
2 static ArrayList<Integer> primos;
3
4 public static void main(String[] args) {
5     //Por ejemplo, si el mayor valor posible es 10000,
6     //    se hace la criba hasta 100
7 }
```

```

6   cribaFactores(100);
7   HashMap<Integer, Integer> fac = factorizar(895);
8   System.out.println(fac.toString());
9   }
10
11  public static void cribaFactores(int n){
12      menorFactor = new int[n+1];
13      Arrays.fill(menorFactor, -1);
14      primos = new ArrayList<Integer>();
15
16      for(int i=2; i<=n; i++){
17          if(menorFactor[i] == -1){
18              primos.add(i);
19              //OJO: Si esta teniendo problemas de overflow,
20                  ↪ cambie la siguiente linea por j = 2*i
21              int j = i*i;
22              while(j <= n){
23                  if(menorFactor[j] == -1){
24                      menorFactor[j] = i;
25                  }
26                  j = j+i;
27              }
28          }
29      }
30
31      public static HashMap<Integer, Integer> factorizar(int
32          ↪ n){
33          HashMap<Integer, Integer> factores = new HashMap<
34          ↪ Integer, Integer>();
35
36          if(n >= menorFactor.length){
37              for(int p : primos){
38                  if(n % p == 0){
39                      int count = 0;
40                      while(n % p == 0){
41                          n = n/p;
42                          count++;
43                      }
44                      factores.put(p, count);
45                  }
46              }
47          }
48          if(n < menorFactor.length) break;
49      }

```

```

46   if(n >= menorFactor.length){
47       factores.put(n, 1);
48       return factores;
49   }
50   }
51
52   while(n > 1){
53       int f = menorFactor[n];
54       if(f == -1){
55           f = n;
56       }
57       if(factores.containsKey(f)){
58           factores.put(f, factores.get(f)+1);
59       }else{
60           factores.put(f, 1);
61       }
62       n = n / f;
63   }
64
65   return factores;
66   }

```

## 7.5. Fórmulas

Para  $n \geq 2$  es posible calcular algunas cosas partiendo de la factorización prima de  $n$ :

$$n = \prod p_i^{a_i} = p_1^{a_1} \cdot p_2^{a_2} \cdot p_3^{a_3} \dots p_k^{a_k}$$

$n = 1$  es un caso especial:

$$d(1) = \sigma(1) = \varphi(1) = 1$$

### 7.5.1. Cantidad de divisores

$$d(n) = \prod (a_i + 1)$$

### 7.5.2. Suma de divisores

$$\sigma(n) = \prod \frac{p_i^{a_i+1} - 1}{p_i - 1}$$

Esta función toma todos los divisores. Por ejemplo, los divisores de 12 son  $\{1, 2, 3, 4, 6, 12\}$ . Por ende,  $\sigma(12) = 1 + 2 + 3 + 4 + 6 + 12 = 28$ . Si se quiere la

suma de los divisores propios (es decir, los divisores excluyendo a  $n$ ), basta con hallar:

$$s(n) = \sigma(n) - n$$

En el ejemplo anterior,  $s(12) = 28 - 12 = 16$ .

### 7.5.3. Función $\varphi$ de Euler

Dos números son relativamente primos (o coprimos) si no tienen divisores en común (es decir, si su MCD es 1).  $\varphi(n)$  se define como la cantidad de enteros positivos menores a  $n$  y coprimos a  $n$ .

$$\varphi(n) = \prod (p_i - 1)p_i^{a_i - 1}$$

## 8. Combinatoria

### 8.1. Permutaciones

Un conjunto de  $n$  elementos diferentes tiene  $n!$  permutaciones. El número máximo cuyo factorial cabe en un *long* de 64 bits es  $n = 20$ . Más allá, se requiere usar BigInteger.

El siguiente código es para generar las permutaciones explícitamente. Se puede hacer hasta  $n = 10$ .

```

1  static String elements[];
2  static boolean p[];
3  static LinkedList<String> permutation;
4
5  public static void main(String[] args) {
6      int n = 3;
7      elements = new String[n];
8      p = new boolean[n];
9      permutation = new LinkedList<String>();
10
11     elements[0] = "A";
12     elements[1] = "B";
13     elements[2] = "C";
14
15     generatePermutations();
16 }
17
18 public static void generatePermutations() {
19     int n = elements.length;
20     if(permutation.size() == n){

```

```

21         System.out.println(permutation.toString());
22     }else{
23         for(int i=0; i<n; i++){
24             if(!p[i]){
25                 p[i] = true;
26                 permutation.addLast(elements[i]);
27                 generatePermutations();
28                 p[i] = false;
29                 permutation.pollLast();
30             }
31         }
32     }
33 }

```

### 8.2. Subconjuntos

Un conjunto de  $n$  elementos tiene  $2^n$  posibles subconjuntos.

Si se busca generarlos, cada subconjunto puede representarse como un número  $b$  de  $n$  bits. El elemento  $k$  pertenece al subconjunto si el bit  $k$  de  $b$  está en 1. No es viable hacerlo para  $n > 30$ .

```

1  public static void main(String[] args) {
2      String elements[] = {"A", "B", "C", "D"};
3      int n = elements.length;
4
5      for(int b=0; b < (1 << n); b++){
6          ArrayList<String> subset = new ArrayList<String>()
7              ↪ ;
8          for(int i=0; i<n; i++){
9              if((b & (1 << i)) != 0){
10                 subset.add(elements[i]);
11             }
12         }
13         System.out.println(subset.toString());
14     }
15 }

```

### 8.3. Coeficientes binomiales

El número de subconjuntos de tamaño  $k$  de un conjunto de tamaño  $n$ , está dado por:

$$\binom{n}{k} = \frac{n!}{k! \cdot (n-k)!}$$



Sin embargo no es muy práctico usar esta fórmula ya que involucra factoriales. Se puede utilizar la recurrencia  $\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$  para generar todos los coeficientes binomiales  $\binom{i}{j}$  para  $0 \leq i, j \leq n$ .

```

1 public static void main(String[] args) {
2     int n = 10;
3     long[][] C = new long[n][n];
4
5     for(int i=0; i<n; i++){
6         C[i][0] = 1;
7         for(int j=1; j<=i; j++){
8             C[i][j] = C[i-1][j-1] + C[i-1][j];
9         }
10    }
11 }
```

Este código funciona para  $n \leq 67$ . Más allá, se requiere usar BigInteger.

## 8.4. Coeficientes multinomiales

Es el número de formas de particionar un conjunto de tamaño  $n$  en subconjuntos disjuntos de tamaño  $a_1, a_2, \dots, a_m$ .

$$\binom{n}{a_1, a_2, \dots, a_m} = \frac{n!}{a_1! \cdot a_2! \cdot \dots \cdot a_m!}$$

## 8.5. Multiconjuntos

Un multiconjunto es un conjunto que permite elementos repetidos. El número de multiconjuntos de cardinalidad  $k$ , con elementos tomados de un conjunto de cardinalidad  $n$ , se puede contar como:

$$\binom{n}{k} = \binom{k+n-1}{k}$$

También se puede usar una recurrencia:

$$\binom{n}{0} = 1, \quad \binom{0}{k} = 0 \text{ para } k > 0$$

$$\binom{n}{k} = \binom{n}{k-1} + \binom{n-1}{k}$$

## 8.6. Particiones

Una partición de un conjunto es una colección de subconjuntos disjuntos, tales que la unión de todos ellos es el conjunto original. La cantidad de maneras diferentes de particionar un conjunto de  $n$  elementos en  $k$  partes se denota como  $S(n, k)$  (números de Stirling de tipo 2).

$$S(n, k) = \begin{cases} 1 & n = 0 \\ 1 & k = 0 \\ S(n-1, k-1) + k \cdot S(n-1, k) & n \geq 1, k \geq 1 \end{cases}$$

## 8.7. Desarrreglos

Un desarrreglo es una permutación que no “mapea” ningún elemento a sí mismo. Por ejemplo, 231 es un desarrreglo de 123, pero 132 no lo es (ya que el 1 queda en la misma posición).

El número de desarrreglos de un conjunto de  $n$  elementos es:

$$D_n = n! \cdot \sum_{k=0}^n \frac{(-1)^k}{k!} \approx \frac{n!}{e}$$

Puede ser más práctico usar la siguiente recurrencia:

$$D_n = \begin{cases} 1 & n = 0 \\ 0 & n = 1 \\ (n-1)(D_{n-1} + D_{n-2}) & n \geq 2 \end{cases}$$

El máximo  $D_n$  que cabe en un *long* es con  $n = 20$ .

## 8.8. Números de Catalan

$$C_n = \frac{1}{n+1} \binom{2n}{n}$$

Sin embargo, puede ser más práctico usar la siguiente recurrencia:

$$C_0 = 1, \quad C_n = \frac{2(2n-1)C_{n-1}}{(n+1)}$$

El número  $C_n$  tiene muchas interpretaciones. Entre ellas:

- El número de árboles binarios diferentes con  $n$  vértices
- El número formas de hacer una expresión con  $n$  pares de paréntesis balanceados

- El número de formas de triangular un polígono con  $n + 2$  lados
- El número de caminos monótonos que no pasan sobre la diagonal en una cuadrícula de  $n \cdot n$ .

El  $C_n$  más grande que cabe en un *long* es con  $n = 33$ .

## 9. Otros

### 9.1. Ordenamiento de Arrays y Listas (ascendente y descendente)

Cuando necesite ordenar un vector o una lista, utilice los métodos `.sort()` que tiene Java. El algoritmo que utilizan es QuickSort y su complejidad es  $O(n \log n)$ .

```
1 public static void main(String[] args) {
2     int n = 10;
3     String v[] = new String[n];
4     ArrayList<Integer> l = new ArrayList<Integer>();
5
6     Arrays.sort(v);
7     Collections.sort(l);
8     //Collections.sort() tambien ordena LinkedList
9 }
```

### 9.2. Cola de Prioridad

Cola en la que la cabeza siempre es el menor elemento presente. Las operaciones `add()` y `poll()` son  $O(\log n)$ . La operación `peek()` es  $O(1)$ .

También se puede invertir, haciendo que la cabeza siempre sea el mayor elemento, pasando el parámetro `Collections.reverseOrder()`.

```
1 public static void main(String[] args) {
2     PriorityQueue<Integer> pq = new PriorityQueue<
3         Integer>();
4     pq.add(3);
5     pq.add(5);
6     pq.add(1);
7     pq.peek(); //No saca el 1 de la cola
8     pq.poll(); //Saca el 1 de la cola y el 3 queda en la
9         cabeza
10
11     //Orden inverso
```

```
10 PriorityQueue<Integer> pqInversa = new PriorityQueue
11     <Integer>(Collections.reverseOrder());
12 pqInversa.add(3);
13 pqInversa.add(1);
14 pq.poll(); //Saca el 3
```

### 9.3. Interfaz Comparable

En ocasiones se puede necesitar ordenar un vector o lista de un tipo de datos definido por el usuario (clase), o utilizar una cola de prioridad. Para hacer esto, la clase debe implementar la interfaz `Comparable` de Java.

El método `compareTo()` debe retornar:

- Negativo si *this* < *obj*.
- 0 si *this* = *obj*.
- Positivo si *this* > *obj*.

Por ejemplo, se tiene una clase `Persona` con dos campos: nombre y edad. Se quiere ordenar una lista de objetos tipo `Persona` de menor a mayor edad.

```
1 class Persona implements Comparable<Persona>{
2     public String nombre;
3     public int edad;
4     public Persona(String nombre, int edad){
5         this.nombre = nombre;
6         this.edad = edad;
7     }
8     public int compareTo(Persona obj) {
9         return this.edad - obj.edad;
10    }
11 }
12
13 class Main {
14     public static void main(String[] args) {
15         ArrayList<Persona> p = new ArrayList<Persona>();
16         p.add(new Persona("Carlos", 20));
17         p.add(new Persona("Juan", 20));
18         Collections.sort(p);
19     }
20 }
```

## 9.4. Imprimir números decimales redondeados

Generalmente basta con esta función de Java para redondear correctamente números decimales.

```
1 public static void main(String[] args) {
2     double d = 9.2651659;
3     //Por ejemplo, para redondear a 4 decimales
4     System.out.format("%.4f\n", d);
5     //Hay que poner el \n si se quiere imprimir tambien
6     //    ↪ un salto de linea
7 }
```

## 9.5. BufferedReader y BufferedWriter

*Scanner* es sencillo de utilizar pero es lento. Se recomienda utilizar siempre *BufferedReader* para leer entradas.

En algunas ocasiones también se necesitará un modo más rápido que *System.out.println()* para imprimir. *BufferedWriter* es más rápido, nunca está de más usarlo.

```
1 public static void main(String[] args) throws
2     //    ↪ IOException {
3     BufferedReader br = new BufferedReader(new
4     //    ↪ InputStreamReader(System.in));
```

```
3 //Solo lee por lineas
4 //Ciclo hasta end of input
5 String s;
6 while((s = br.readLine()) != null){
7     String l[] = s.split(" ");
8 }
9
10 //Ciclo con numero de casos
11 int t = Integer.parseInt(br.readLine());
12 for(int i=0; i<t; i++){
13     String l[] = br.readLine().split(" ");
14 }
15
16 BufferedWriter bw = new BufferedWriter(new
17     //    ↪ OutputStreamWriter(System.out));
18 //No pone un salto de linea al final. Por tanto, se
19     //    ↪ debe poner \n cuando sea necesario
20 bw.write("Hola_mundo\n");
21 //El flush es el que realmente imprime en consola.
22     //    ↪ En lo posible, hacer flush solo una vez, al
23     //    ↪ final de procesar todos los casos
24 bw.flush();
25 }
```