

# Índice

## 1. Mapas

## 2. Sets

## 3. Grafos

3.1. BFS y DFS . . . . .	2
3.2. Shortest Hop . . . . .	2
3.3. Ordenamiento Topológico . . . . .	3
3.4. Componentes Fuertemente Conexas (Algoritmo de Tarjan) . . . . .	4
3.5. Puntos de Articulación . . . . .	5
3.6. Puentes . . . . .	6
3.7. Algoritmo de Dijkstra . . . . .	6
3.8. Algoritmo de Floyd-Warshall . . . . .	7

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

## 5. KMP

## 6. Union-Find

## 7. Teoría de números

7.1. Algoritmo de Euclides . . . . .	10
7.2. Verificar si un número es primo . . . . .	10
7.3. Criba de Eratóstenes . . . . .	10
7.4. Factorización prima de un número . . . . .	10
7.5. Cantidad de divisores de un número . . . . .	11
7.6. Los $n$ primeros números primos . . . . .	11

## 8. Otros

8.1. Ordenamiento de Arrays y Listas . . . . .	11
8.2. Imprimir números decimales redondeados . . . . .	12
8.3. BufferedReader y BufferedWriter . . . . .	12

## 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.

```
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
22     System.out.println(map.get("tres"));
23
24     //Recorrer el mapa
25     for(Entry<String, Integer> e : map.entrySet()){
26         System.out.println(e.getKey() + " : " + e.getValue
27             ↪ ());
28     }
29 }
```

## 2. Sets

Estructura de datos que actúa como “bolsa” donde se almacenan elementos, pero no pueden 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. Grafos

#### 3.1. BFS y DFS

Recorren un grafo a partir de un nodo origen y visitan todos los nodos alcanzables desde éste. Ambos algoritmos tienen un tiempo de ejecución 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        int n = 10;
6
7        seen = new boolean[n];
8        g = new ArrayList[n];
9        for(int i = 0; i < n; i++){
10            g[i] = new ArrayList<Integer>();
11        }
12
13        //Visita solo los nodos que son alcanzables desde el
14        //    ↗ nodo 's'
15        int s = 0;
16        dfs(s);
17
18        //Con el vector 'seen' vemos cuales son estos nodos
19        for(int i=0; i<n; i++){
20            if(seen[i]){
21                // 'i' es alcanzable desde 's'
22            }
23        }
24    }

```

```

23
24    //Si queremos visitar todos los nodos
25    for(int u=0; u<n; u++){
26        if(!seen[u]){
27            //Si no hemos visitado 'u', hacer DFS en 'u'
28            dfs(u);
29        }
30    }
31 }
32
33 private static void dfs(int u){
34     seen[u] = true;
35     int len = g[u].size();
36     for(int i=0; i<len; i++){
37         int v = g[u].get(i);
38         if(!seen[v]){
39             dfs(v);
40         }
41     }
42 }
43
44 private static void bfs(int u){
45     seen[u] = true;
46     Queue<Integer> q = new LinkedList<Integer>();
47     q.add(u);
48     while(!q.isEmpty()){
49         u = q.poll();
50         int len = g[u].size();
51         for(int i=0; i<len; i++){
52             int v = g[u].get(i);
53             if(!seen[v]){
54                 seen[v] = true;
55                 q.add(v);
56             }
57         }
58     }
59 }

```

#### 3.2. 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

tiempo de ejecución es el mismo 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
24     //Distancia "infinita" hacia todos los nodos
25     for(int i=0; i<n; i++){
26         dist[i] = Integer.MAX_VALUE;
27     }
28     //Distancia 0 hacia el nodo de origen
29     dist[u] = 0;
30
31     //BFS "modificado"
32     seen[u] = true;
33     Queue<Integer> q = new LinkedList<Integer>();
34     q.add(u);
35     while(!q.isEmpty()){
36         u = q.poll();
37         int len = g[u].size();
38         for(int i=0; i<len; i++){
39             int v = g[u].get(i);
40             if(!seen[v]){
41                 seen[v] = true;

```

```

41         q.add(v);
42         //Lo unico que cambia es que se calcula el
43         //    ↪ dist[v]
44         dist[v] = dist[u] + 1;
45     }
46 }
47 }

```

### 3.3. 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. En realidad es una modificación de DFS y su tiempo de ejecución es el mismo:  $O(n + m)$ . 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
11     g = new ArrayList[n];
12     for(int i = 0; i < n; i++){
13         g[i] = new ArrayList<Integer>();
14     }
15
16     boolean sinCiclo = true;
17
18     //Es necesario hacer el ciclo para visitar todos los
19     //    ↪ nodos
20     for(int u=0; u<n; u++){
21         if(seen[u] == 0){
22             sinCiclo = sinCiclo && topoDfs(u);
23         }
24     }

```

```

24     if(sinCiclo){
25         //La lista 'topoSort' contiene los nodos en su
26             ↪ orden topologico
27     }else{
28         //Hay un ciclo
29     }
30 }
31
32 private static boolean topoDfs(int u){
33     //DFS "modificado" para hacer ordenamiento
34         ↪ topologico
35     //Se marca 'u' como 'gris'
36     seen[u] = 1;
37     int len = g[u].size();
38     boolean sinCiclo = true;
39     for(int i=0; i<len; i++){
40         int v = g[u].get(i);
41         if(seen[v] == 0){
42             sinCiclo = sinCiclo && topoDfs(v);
43         }else if(seen[v] == 1){
44             //Hay un ciclo, retorna falso
45             sinCiclo = false;
46         }
47     }
48     //Se agrega el nodo 'u' al inicio de la lista y se
49         ↪ marca 'negro'
50     seen[u] = 2;
51     topoSort.addFirst(u);
52     return sinCiclo;
53 }

```

### 3.4. Componentes Fuertemente Conexas (Algoritmo de Tarjan)

Calcula la componente fuertemente conexa 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 de  $v$  a  $u$ . Su tiempo de ejecución es  $O(n + m)$ .

```

1  static ArrayList<Integer> g[];
2  static boolean seen[];
3  static boolean stackMember[];
4  static int disc[];

```

```

5  static int low[];
6  static int scc[];
7  static Stack<Integer> st;
8  static int time;
9  static int component;
10
11 public static void main(String[] args) {
12     int n = 10;
13
14     seen = new boolean[n];
15     stackMember = new boolean[n];
16     disc = new int[n];
17     low = new int[n];
18     scc = new int[n];
19     st = new Stack<Integer>();
20     time = 0;
21     component = 0;
22     g = new ArrayList[n];
23     for(int i = 0; i < n; i++){
24         g[i] = new ArrayList<Integer>();
25     }
26
27     for(int u=0; u<n; u++){
28         if(!seen[u]){
29             tarjan(u);
30         }
31     }
32     //scc[i]==x significa que 'i' pertenece a la
33         ↪ componente 'x'
34 }
35
36 private static void tarjan(int u){
37     seen[u] = true;
38     st.add(u);
39     stackMember[u] = true;
40     disc[u] = time;
41     low[u] = time;
42     time++;
43
44     int len = g[u].size();
45     for(int i=0; i<len; i++){
46         int v = g[u].get(i);
47         if(!seen[v]){

```

```

47     tarjan(v);
48     low[u] = Math.min(low[u], low[v]);
49 }else if(stackMember[v]){
50     low[u] = Math.min(low[u], disc[v]);
51 }
52 }
53
54 if(low[u] == disc[u]){
55     int w;
56     do{
57         w = st.pop();
58         stackMember[w] = false;
59         scc[w] = component;
60     }while(w != u);
61     component++;
62 }
63 }

```

### 3.5. 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. El tiempo de ejecución del algoritmo es  $O(n + m)$ .

```

1  static ArrayList<Integer> g[];
2  static boolean seen[];
3  static int disc[];
4  static int low[];
5  static int time;
6  static int parent[];
7  static boolean ap[];
8
9  public static void main(String[] args) {
10     int n = 10;
11
12     seen = new boolean[n];
13     disc = new int[n];
14     low = new int[n];
15     time = 0;
16     ap = new boolean[n];
17     g = new ArrayList[n];

```

```

18     for(int i = 0; i < n; i++){
19         g[i] = new ArrayList<Integer>();
20     }
21     parent = new int[n];
22     for(int i=0; i<n; i++){
23         parent[i] = -1;
24     }
25
26     for(int u=0; u<n; u++){
27         if(!seen[u]){
28             articulationPoints(u);
29         }
30     }
31     //Si ap[i]==true, 'i' es un punto de articulacion
32 }
33
34 private static void articulationPoints(int u){
35     seen[u] = true;
36     disc[u] = time;
37     low[u] = time;
38     time++;
39     int children = 0;
40
41     int len = g[u].size();
42     for(int i=0; i<len; i++){
43         int v = g[u].get(i);
44         if(!seen[v]){
45             children++;
46             parent[v] = u;
47             articulationPoints(v);
48             low[u] = Math.min(low[u], low[v]);
49             if(parent[u] == -1 && children > 1){
50                 ap[u] = true;
51             }else if(parent[u] != -1 && low[v] >= disc[u]){
52                 ap[u] = true;
53             }
54         }else if(v != parent[u]){
55             low[u] = Math.min(low[u], disc[v]);
56         }
57     }
58 }

```

### 3.6. 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. El tiempo de ejecución del algoritmo es  $O(n + m)$ .

```
1 class Bridge {
2     public int u;
3     public int v;
4     public Bridge(int u, int v){
5         this.u = u;
6         this.v = v;
7     }
8 }
9
10 public class GraphBridges {
11
12     static ArrayList<Integer> g[];
13     static boolean seen[];
14     static int disc[];
15     static int low[];
16     static int time;
17     static int parent[];
18     static ArrayList<Bridge> bridgeEdges;
19
20     public static void main(String[] args) {
21         int n = 10;
22
23         seen = new boolean[n];
24         disc = new int[n];
25         low = new int[n];
26         time = 0;
27         parent = new int[n];
28         bridgeEdges = new ArrayList<Bridge>();
29
30         g = new ArrayList[n];
31         for(int i = 0; i < n; i++){
32             g[i] = new ArrayList<Integer>();
33         }
34
35         for(int i=0; i<n; i++){
36             parent[i]=-1;
37         }
```

```
38
39         for(int u=0; u<n; u++){
40             if(!seen[u]){
41                 bridges(u);
42             }
43         }
44         // 'bridgeEdges' contiene objetos tipo Bridge que
45         ↪ indican que la arista u,v es un puente
46     }
47
48     private static void bridges(int u){
49         seen[u] = true;
50         disc[u] = time;
51         low[u] = time;
52         time++;
53
54         int len = g[u].size();
55         for(int i=0; i<len; i++){
56             int v = g[u].get(i);
57             if(!seen[v]){
58                 parent[v] = u;
59                 bridges(v);
60                 low[u] = Math.min(low[u], low[v]);
61                 if(low[v] > disc[u]){
62                     Bridge b = new Bridge(u, v);
63                     bridgeEdges.add(b);
64                 }
65             } else if(v != parent[u]){
66                 low[u] = Math.min(low[u], disc[v]);
67             }
68         }
69     }
```

### 3.7. 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. Su tiempo de ejecución es  $O(m + n \log n)$

```
1 public class Dijkstra {
2
3     static ArrayList<Integer> g[];
```

```

4  static Nodo[] dist;
5  static PriorityQueue<Nodo> proximo;
6  static int[] parent;
7  static int[][] p;
8
9  public static void main(String[] args) throws
    ↪ IOException {
10     int nodos = 5;
11
12     g = new ArrayList[nodos];
13     dist = new Nodo[nodos];
14     parent = new int[nodos];
15     proximo = new PriorityQueue<Nodo>();
16     p = new int[nodos][nodos];
17
18     for (int i = 0; i < nodos; i++) {
19         g[i] = new ArrayList<Integer>();
20         dist[i] = new Nodo(i, Integer.MAX_VALUE);
21     }
22
23     int src = 0;
24     int dest = 4;
25
26     dist[src].peso = 0;
27     proximo.add(dist[src]);
28     parent[src] = src;
29
30     Nodo a;
31     while (!proximo.isEmpty()) {
32         a = proximo.poll();
33         calcularDistancia(a);
34     }
35
36     int mejorDistancia = dist[dest].peso;
37
38     int r = dest;
39     //bw.write(r+"\n");
40     while (r != src) {
41         r = parent[r];
42         //bw.write(r+"\n");
43     }
44     //bw.flush();
45 }

```

```

46
47 public static void calcularDistancia(Nodo u){
48     int t = g[u.n].size();
49     for (int i = 0; i < t; i++) {
50         int x = g[u.n].get(i);
51         if (u.peso + p[u.n][x] < dist[x].peso) {
52             dist[x].peso = u.peso + p[u.n][x];
53             proximo.add(dist[x]);
54             parent[x] = u.n;
55         }
56     }
57 }
58 }
59
60 class Nodo implements Comparable<Nodo>{
61     int n;
62     int peso;
63
64     public Nodo(int n, int peso){
65         this.n = n;
66         this.peso = peso;
67     }
68
69     public int compareTo(Nodo o) {
70         return this.peso - o.peso;
71     }
72 }

```

### 3.8. 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.

## 4. 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 que maximicen el valor total pero cuyos pesos no excedan  $W$ . Su tiempo de ejecución 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 // n = numero de items, W = capacidad de la mochila
3 int n = 4;
4 int W = 8;
5 int values[] = {15, 10, 9, 5};
6 int weights[] = {1, 5, 3, 4};
7
8 //Tener cuidado: En la matriz los items se numeran
9   ↪ 1...n y la capacidad de la mochila 1...W
10 int A[][] = new int[n+1][W+1];
11
12 //Aca se resuelve el problema. Asegurarse de tener
13   ↪ los values y weights
14 for (int i=1; i<=n; i++) {
15     for (int x=0; x<=W; x++) {
16         if (weights[i-1] > x) {
17             A[i][x] = A[i-1][x];
18         } else {
19             int p = A[i-1][x];
20             int q = A[i-1][x-weights[i-1]] + values[i-1];
21             A[i][x] = (p > q) ? p : q;
22         }
23     }
24 }
25
26 //El valor maximo que se puede obtener es A[n][W]
27 int solution = A[n][W];
28
29 //Si se quiere determinar cuales items se incluyeron
30 boolean chosen[] = new boolean[n];
31 int i = n;
32 int j = W;
33
34 while(i>0){
35     if(A[i][j] == A[i-1][j]){
36         i--;
37     }else{
38         chosen[i-1] = true;
39         i--;
40         j = j-weights[i];
41     }
42 }
43
44 //Si chosen[i]==true es porque i se incluye
45 }

```

## 5. KMP

Algoritmo para buscar una cadena *pattern* dentro de una cadena *text*. Su tiempo de ejecución 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
2   ↪ pattern){
3     int lps[] = new int[pattern.length()];
4     int index = 0;
5     int i = 1;
6     while(i < pattern.length()){
7         if(pattern.charAt(i) == pattern.charAt(index)){
8             lps[i] = index + 1;
9             index++;
10            i++;
11        }else{
12            if(index != 0){
13                index = lps[index-1];
14            }else{
15                lps[i] = 0;
16                i++;
17            }
18        }
19    }
20    return lps;
21 }
22
23 public static int KMP(String text, String pattern){
24     int lps[] = computeTemporaryArray(pattern);
25     int i=0;
26     int j=0;
27     while(i < text.length() && j < pattern.length()){
28         if(text.charAt(i) == pattern.charAt(j)){
29             i++;
30             j++;
31         }else{
32             if(j!=0){
33                 j = lps[j-1];
34             }else{
35                 i++;
36             }
37         }
38     }
39 }

```



```

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. Union-Find

Estructura de datos que soporta las siguientes operaciones eficientemente:

- Unir dos elementos  $p, q$ , es decir, indicar que pertenecen al mismo conjunto
- Determinar si dos elementos  $p, q$  pertenecen al mismo conjunto o no

```

1 class UnionFind{
2     private int parent[];
3     private int size[];
4     private int components;
5
6     // n = Numero de nodos
7     public UnionFind(int n){
8         components = n;
9         parent = new int[n];
10        size = new int[n];
11        for(int i=0; i<n; i++){
12            parent[i] = i;
13            size[i] = 1;
14        }
15    }
16
17    private int root(int p){
18        while(p != parent[p]){
19            parent[p] = parent[parent[p]];
20            p = parent[p];

```

```

21    }
22    return p;
23 }
24
25 //Une los nodos p,q
26 public void union(int p, int q){
27     int rootP = root(p);
28     int rootQ = root(q);
29     if(rootP != rootQ){
30         if(size[rootP] < size[rootQ]){
31             parent[rootP] = rootQ;
32             size[rootQ] = size[rootQ] + size[rootP];
33         }else{
34             parent[rootQ] = rootP;
35             size[rootP] = size[rootP] + size[rootQ];
36         }
37         components--;
38     }
39 }
40
41 //Retorna true si p,q estan conectados
42 public boolean connected(int p, int q){
43     return root(p) == root(q);
44 }
45
46 //Retorna el numero de componentes conexas
47 public int getComponents(){
48     return components;
49 }
50 }
51
52 class Main {
53     public static void main(String[] args){
54         UnionFind uf = new UnionFind(5);
55         uf.union(0, 2);
56         uf.union(1, 0);
57         uf.union(3, 4);
58
59         //El numero de componentes es
60         int comp = uf.getComponents();
61
62         //Dos nodos estan conectados?
63         boolean connected = uf.connected(0, 3);

```

```
64 }
65 }
```

## 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 void main(String[] args) {
2     isPrime(63); //true
3     isPrime(99); //false
4
5     //De hecho estos si caben en un int pero los pongo
6     // como ejemplo
7     BigInteger a = new BigInteger("104723");
8     BigInteger b = new BigInteger("104727");
```

```
8     a.isProbablePrime(10); //true
9     b.isProbablePrime(10); //false
10 }
11
12 public static boolean isPrime(int x){
13     if(x == 1) return false;
14     if(x == 2) return true;
15
16     int s = (int) Math.ceil(Math.sqrt(x));
17
18     for(int i=2; i<=s; i++){
19         if(x % i == 0) return false;
20     }
21
22     return true;
23 }
```

### 7.3. Criba de Eratóstenes

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

```
1 public static ArrayList<Integer> sieve(int n){
2     boolean marked[] = new boolean[n+1];
3     ArrayList<Integer> primes = new ArrayList<Integer>()
4     // ↪ ;
5
6     for(int i=2; i<=n; i++){
7         if(!marked[i]){
8             primes.add(i);
9             int j = 2*i;
10            while(j <= n){
11                marked[j] = true;
12                j = j+i;
13            }
14        }
15    }
16
17    return primes;
18 }
```

## 7.4. Factorización prima de un número

Se hace una Criba de Eratóstenes hasta  $n$ . Luego, se verifica si  $n$  es divisible por los números primos hallados. El método retorna un HashMap donde la *clave* es el factor primo y el *valor* su multiplicidad.

```
1 //Usa el metodo sieve()
2 public static HashMap<Integer, Integer> factor(int n){
3     ArrayList<Integer> primes = sieve(n);
4     HashMap<Integer, Integer> primeFactors = new HashMap
        ↪ <Integer, Integer>();
5
6     for(int p: primes){
7         while(n % p == 0){
8             n = n/p;
9             if(primeFactors.containsKey(p)){
10                 primeFactors.put(p, primeFactors.get(p)+1);
11             }else{
12                 primeFactors.put(p, 1);
13             }
14         }
15     }
16
17     return primeFactors;
18 }
```

## 7.5. Cantidad de divisores de un número

Este método utiliza la factorización prima descrita en la sección anterior para hallar la cantidad de divisores que tiene un número. No lista los divisores, sólo dice cuántos son.

```
1 //Usa el metodo factor(), que a su vez usa a sieve()
2 public static int divisors(int n){
3     int div = 1;
4     HashMap<Integer, Integer> primeFactors = factor(n);
5
6     for(Entry<Integer, Integer> e: primeFactors.entrySet
        ↪ ()){
7         div = div * (e.getValue() + 1);
8     }
9
10    return div;
11 }
```

## 7.6. Los $n$ primeros números primos

Si queremos hallar los primeros  $n$  números primos, la criba de Eratóstenes no nos puede ayudar, porque no sabemos hasta dónde tenemos que buscar. Con este método se optimiza un poco el acercamiento “fuerza bruta” a este problema.

```
1 static ArrayList<Integer> primes;
2
3 public static void generatePrimes(int n){
4     primes = new ArrayList<Integer>();
5     primes.add(2);
6     int x = 3;
7
8     while(primes.size() < n){
9         if(isPrime(x)){
10             primes.add(x);
11         }
12         x = x+2;
13     }
14 }
15
16 private static boolean isPrime(int x){
17     int s = (int) Math.ceil(Math.sqrt(x));
18     boolean prime = true;
19
20     for(int p: primes){
21         if(p > s) break;
22         if(x % p == 0){
23             prime = false;
24             break;
25         }
26     }
27
28     return prime;
29 }
```

## 8. Otros

### 8.1. Ordenamiento de Arrays y Listas

Cuando necesite ordenar un vector o una lista, utilice los métodos `.sort()` que tiene Java. El algoritmo que utilizan es QuickSort y su tiempo de ejecución

es de  $O(n \log n)$ .

```
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() tambien ordena LinkedList
8     Collections.sort(l);
9
10    //En ArrayList .get() es O(1) pero en LinkedList es
        ↳ O(n), por lo cual hacer esto es fatal
11    for(int i=0; i<l.size(); i++){
12        System.out.println(l.get(i));
13    }
14
15    //La forma eficiente
16    for(int k : l){
17        System.out.println(k);
18    }
19 }
```

## 8.2. 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     //Para redondear a 4 decimales
4     //Hay que poner el \n si se quiere imprimir tambien
        ↳ un salto de linea
5     System.out.format("%.4f\n", d);
```

```
6 }
```

## 8.3. 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
        ↳ IOException {
2     BufferedReader br = new BufferedReader(new
        ↳ InputStreamReader(System.in));
3     //Solo lee por lineas
4     String s;
5     while((s = br.readLine()) != null){
6         int n = Integer.parseInt(br.readLine());
7         String l[] = br.readLine().split("_");
8     }
9
10    BufferedWriter bw = new BufferedWriter(new
        ↳ OutputStreamWriter(System.out));
11    //No pone un salto de linea al final como si lo hace
        ↳ System.out.println(). Por tanto, se debe
        ↳ poner \n cuando sea necesario
12    bw.write("Hola_mundo\n");
13    //El flush es el que realmente imprime en consola.
        ↳ En lo posible, hacer flush solo una vez, al
        ↳ final de todo
14    bw.flush();
15 }
```