Índice

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.

```java
public static void main(String args[]){
  HashMap<String, Integer> map = new HashMap<String, Integer>();
  //TreeMap<String, Integer> map = new TreeMap<String, Integer>();
  //LinkedHashMap<String, Integer> map = new LinkedHashMap<String, Integer>();

  String s = "tres tristes tigres tragaban trigo en un trigal en tres tristes trastos";
  String palabras[] = s.split(" ");

  for(int i=0; i<palabras.length; i++){
    if(!map.containsKey(palabras[i])){
      map.put(palabras[i], 1);
    }else{
      map.put(palabras[i], map.get(palabras[i])+1);
    }
  }

  //Obtener un elemento
  System.out.println(map.get("tres"));

  //Recorrer el mapa
  for(Entry<String, Integer> e : map.entrySet()){
    System.out.println(e.getKey() + " : " + e.getValue());
  }
}
```

2. Sets

Estructura de datos que actúan 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.

```java
public static void main(String[] args) {
  HashSet<String> hs = new HashSet<String>();
  //TreeSet<String> ts = new TreeSet<String>();

  hs.add("Hola");
```

```java
    hs.add("Hola");
    hs.add("Mundo");
    //Imprime 2, porque no se aceptan repetidos
    System.out.println(hs.size());

    //Recorrido
    for(String s: hs){
      System.out.println(s);
    }
}
```

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.

```java
static ArrayList<Integer> g[];
static boolean seen[];

public static void main(String[] args) {
  int n = 10;

  seen = new boolean[n];
  g = new ArrayList[n];
  for(int i = 0; i < n; i++){
    g[i] = new ArrayList<Integer>();
  }

  //Visita solo los nodos que son alcanzables desde el nodo 's'
  int s = 0;
  dfs(s);

  //Con el vector 'seen' vemos cuales son estos nodos
  for(int i=0; i<n; i++){
    if(seen[i]){
      //'i' es alcanzable desde 's'
    }
  }

  //Si queremos visitar todos los nodos
  for(int u=0; u<n; u++){
    if(!seen[u]){
      //Si no hemos visitado 'u', hacer DFS en 'u'
      dfs(u);
    }
  }
}

private static void dfs(int u){
  seen[u] = true;
  int len = g[u].size();
  for(int i=0; i<len; i++){
    int v = g[u].get(i);
    if(!seen[v]){
      dfs(v);
    }
  }
}

private static void bfs(int u){
  seen[u] = true;
  Queue<Integer> q = new LinkedList<Integer>();
  q.add(u);
  while(!q.isEmpty()){
    u = q.poll();
    int len = g[u].size();
    for(int i=0; i<len; i++){
      int v = g[u].get(i);
      if(!seen[v]){
        seen[v] = true;
        q.add(v);
      }
    }
  }
}
```

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: $O(n+m)$.

```java
static ArrayList<Integer> g[];
static boolean seen[];
static int dist[];

public static void main(String[] args) {
  int n = 10;

  seen = new boolean[n];
  dist = new int[n];
  g = new ArrayList[n];
  for(int i=0; i<n; i++){
    g[i] = new ArrayList<Integer>();
  }

  int s = 0;
  shortestHop(s);
  //Despues de llamar este metodo, en dist[i] esta la distancia mas corta (s,i)
}

public static void shortestHop(int u){
  int n = g.length;

  //Distancia "infinita" hacia todos los nodos
  for(int i=0; i<n; i++){
    dist[i] = Integer.MAX_VALUE;
  }
  //Distancia 0 hacia el nodo de origen
  dist[u] = 0;

  //BFS "modificado"
  seen[u] = true;
  Queue<Integer> q = new LinkedList<Integer>();
  q.add(u);
  while(!q.isEmpty()){
    u = q.poll();
    int len = g[u].size();
    for(int i=0; i<len; i++){
      int v = g[u].get(i);
      if(!seen[v]){
        seen[v] = true;
        q.add(v);
        //Lo unico que cambia es que se calcula el dist[v]
        dist[v] = dist[u] + 1;
      }
    }
  }
}
```

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.

```java
static ArrayList<Integer> g[];
static int seen[];
static LinkedList<Integer> topoSort;

public static void main(String[] args) {
  int n = 10;

  seen = new int[n];
  topoSort = new LinkedList<Integer>();

  g = new ArrayList[n];
  for(int i = 0; i < n; i++){
    g[i] = new ArrayList<Integer>();
  }

  boolean sinCiclo = true;

  //Es necesario hacer el ciclo para visitar todos los nodos
  for(int u=0; u<n; u++){
    if(seen[u] == 0){
      sinCiclo = sinCiclo && topoDfs(u);
    }
  }
```

```java

    if(sinCiclo){
      //La lista 'topoSort' contiene los nodos en su orden topologico
    }else{
      //Hay un ciclo
    }
  }

  private static boolean topoDfs(int u){
    //DFS "modificado" para hacer ordenamiento topologico
    //Se marca 'u' como 'gris'
    seen[u] = 1;
    int len = g[u].size();
    boolean sinCiclo = true;
    for(int i=0; i<len; i++){
      int v = g[u].get(i);
      if(seen[v] == 0){
        sinCiclo = sinCiclo && topoDfs(v);
      }else if(seen[v] == 1){
        //Hay un ciclo, retorna falso
        sinCiclo = false;
      }
    }
    //Se agrega el nodo 'u' al inicio de la lista y se marca 'negro'
    seen[u] = 2;
    topoSort.addFirst(u);
    return sinCiclo;
  }
```

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 $O(n+m)$.

```java
  static ArrayList<Integer> g[];
  static boolean seen[];
  static boolean stackMember[];
  static int disc[];
  static int low[];
  static int scc[];
  static Stack<Integer> st;
  static int time;
  static int component;

  public static void main(String[] args) {
    int n = 10;

    seen = new boolean[n];
    stackMember = new boolean[n];
    disc = new int[n];
    low = new int[n];
    scc = new int[n];
    st = new Stack<Integer>();
    time = 0;
    component = 0;
    g = new ArrayList[n];
    for(int i = 0; i < n; i++){
      g[i] = new ArrayList<Integer>();
    }

    for(int u=0; u<n; u++){
      if(!seen[u]){
        tarjan(u);
      }
    }
    //scc[i]==x significa que 'i' pertenece a la componente 'x'
  }

  private static void tarjan(int u){
    seen[u] = true;
    st.add(u);
    stackMember[u] = true;
    disc[u] = time;
    low[u] = time;
    time++;

    int len = g[u].size();
    for(int i=0; i<len; i++){
      int v = g[u].get(i);
      if(!seen[v]){
```

```java
          tarjan(v);
          low[u] = Math.min(low[u], low[v]);
        }else if(stackMember[v]){
          low[u] = Math.min(low[u], disc[v]);
        }
      }

      if(low[u] == disc[u]){
        int w;
        do{
          w = st.pop();
          stackMember[w] = false;
          scc[w] = component;
        }while(w != u);
        component++;
      }
    }
```

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 $O(n+m)$.

```java
static ArrayList<Integer> g[];
static boolean seen[];
static int disc[];
static int low[];
static int time;
static int parent[];
static boolean ap[];

public static void main(String[] args) {
  int n = 10;

  seen = new boolean[n];
  disc = new int[n];
  low = new int[n];
  time = 0;
  ap = new boolean[n];
  g = new ArrayList[n];
  for(int i = 0; i < n; i++){
    g[i] = new ArrayList<Integer>();
  }
  parent = new int[n];
  for(int i=0; i<n; i++){
    parent[i] = -1;
  }5

  for(int u=0; u<n; u++){
    if(!seen[u]){
      articulationPoints(u);
    }
  }
  //Si ap[i]==true, 'i' es un punto de articulacion
}

private static void articulationPoints(int u){
  seen[u] = true;
  disc[u] = time;
  low[u] = time;
  time++;
  int children = 0;

  int len = g[u].size();
  for(int i=0; i<len; i++){
    int v = g[u].get(i);
    if(!seen[v]){
      children++;
      parent[v] = u;
      articulationPoints(v);
      low[u] = Math.min(low[u], low[v]);
      if(parent[u] == -1 && children > 1){
        ap[u] = true;
      }else if(parent[u] != -1 && low[v] >= disc[u]){
        ap[u] = true;
      }
    }else if(v != parent[u]){
      low[u] = Math.min(low[u], disc[v]);
    }
  }
}
```

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 $O(n+m)$.

```java
class Bridge {
  public int u;
  public int v;
  public Bridge(int u, int v){
    this.u = u;
    this.v = v;
  }
}

public class GraphBridges {

  static ArrayList<Integer> g[];
  static boolean seen[];
  static int disc[];
  static int low[];
  static int time;
  static int parent[];
  static ArrayList<Bridge> bridgeEdges;

  public static void main(String[] args) {
    int n = 10;

    seen = new boolean[n];
    disc = new int[n];
    low = new int[n];
    time = 0;
    parent = new int[n];
    bridgeEdges = new ArrayList<Bridge>();

    g = new ArrayList[n];
    for(int i = 0; i < n; i++){
      g[i] = new ArrayList<Integer>();
    }

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

    for(int u=0; u<n; u++){
      if(!seen[u]){
        bridges(u);
      }
    }
    //'bridgeEdges' contiene objetos tipo Bridge que indican que la arista u,v es un puente
  }

  private static void bridges(int u){
    seen[u] = true;
    disc[u] = time;
    low[u] = time;
    time++;

    int len = g[u].size();
    for(int i=0; i<len; i++){
      int v = g[u].get(i);
      if(!seen[v]){
        parent[v] = u;
        bridges(v);
        low[u] = Math.min(low[u], low[v]);
        if(low[v] > disc[u]){
          Bridge b = new Bridge(u, v);
          bridgeEdges.add(b);
        }
      }else if(v != parent[u]){
        low[u] = Math.min(low[u], disc[v]);
      }
    }
  }
}
```

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 $O(m + n\log n)$

```java
public class Dijkstra {

  static ArrayList<Integer> g[];
```

```
  static Nodo[] dist;
  static PriorityQueue<Nodo> proximo;
  static int[] parent;
  static int[][] p;

  public static void main(String[] args) throws IOException {
    int nodos = 5;

    g = new ArrayList[nodos];
    dist = new Nodo[nodos];
    parent = new int[nodos];
    proximo = new PriorityQueue<Nodo>();
    p = new int[nodos][nodos];

    for (int i = 0; i < nodos; i++) {
      g[i] = new ArrayList<Integer>();
      dist[i] = new Nodo(i, Integer.MAX_VALUE);
    }

    int src = 0;
    int dest = 4;

    dist[src].peso = 0;
    proximo.add(dist[src]);
    parent[src] = src;

    Nodo a;
    while(!proximo.isEmpty()) {
      a = proximo.poll();
      calcularDistancia(a);
    }

    int mejorDistancia = dist[dest].peso;

    int r = dest;
    //bw.write(r+"\n");
    while(r != src){
      r = parent[r];
      //bw.write(r+"\n");
    }
    //bw.flush();
  }

  public static void calcularDistancia(Nodo u){
    int t = g[u.n].size();
    for (int i = 0; i < t; i++) {
      int x = g[u.n].get(i);
      if(u.peso + p[u.n][x] < dist[x].peso){
        dist[x].peso = u.peso + p[u.n][x];
        proximo.add(dist[x]);
        parent[x] = u.n;
      }
    }
  }
}

class Nodo implements Comparable<Nodo>{
  int n;
  int peso;

  public Nodo(int n, int peso){
    this.n = n;
    this.peso = peso;
  }

  public int compareTo(Nodo o) {
    return this.peso-o.peso;
  }
}
```

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.

```
  public static void main(String[] args) {
```

```java
    // n = numero de items, W = capacidad de la mochila
    int n = 4;
    int W = 8;
    int values[] = {15, 10, 9, 5};
    int weights[] = {1, 5, 3, 4};

    //Tener cuidado: En la matriz los items se numeran 1...n y la capacidad de la mochila 1...W
    int A[][] = new int[n+1][W+1];

    //Aca se resuelve el problema. Asegurarse de tener los values y weights
    for (int i=1; i<=n; i++) {
      for (int x=0; x<=W; x++) {
        if (weights[i-1] > x) {
          A[i][x] = A[i-1][x];
        } else {
          int p = A[i-1][x];
          int q = A[i-1][x-weights[i-1]] + values[i-1];
          A[i][x] = (p > q) ? p : q;
        }
      }
    }

    //El valor maximo que se puede obtener es A[n][W]
    int solution = A[n][W];

    //Si se quiere determinar cuales items se incluyeron
    boolean chosen[] = new boolean[n];
    int i = n;
    int j = W;

    while(i>0){
      if(A[i][j] == A[i-1][j]){
        i--;
      }else{
        chosen[i-1] = true;
        i--;
        j = j-weights[i];
      }
    }
    //Si chosen[i]==true es porque i se incluyo
}
```

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.

```java
private static int[] computeTemporaryArray(String pattern){
  int lps[] = new int[pattern.length()];
  int index = 0;
  int i = 1;
  while(i < pattern.length()){
    if(pattern.charAt(i) == pattern.charAt(index)){
      lps[i] = index + 1;
      index++;
      i++;
    }else{
      if(index != 0){
        index = lps[index-1];
      }else{
        lps[i] = 0;
        i++;
      }
    }
  }
  return lps;
}

public static int KMP(String text, String pattern){
  int lps[] = computeTemporaryArray(pattern);
  int i=0;
  int j=0;
  while(i < text.length() && j < pattern.length()){
    if(text.charAt(i) == pattern.charAt(j)){
      i++;
      j++;
    }else{
      if(j!=0){
        j = lps[j-1];
      }else{
        i++;
      }
```

```
      }
    }
    if(j == pattern.length()){
      return i-j;
    }
    return -1;
  }

  public static void main(String[] args) {
    String text = "ABABABABC";
    String pattern = "BABABC";
    int index = KMP(text, pattern);
  }
```

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

```
class UnionFind{
  private int parent[];
  private int size[];
  private int components;

  // n = Numero de nodos
  public UnionFind(int n){
    components = n;
    parent = new int[n];
    size = new int[n];
    for(int i=0; i<n; i++){
      parent[i] = i;
      size[i] = 1;
    }
  }

  private int root(int p){
    while(p != parent[p]){
      parent[p] = parent[parent[p]];
      p = parent[p];
    }
    return p;
  }

  //Une los nodos p,q
  public void union(int p, int q){
    int rootP = root(p);
    int rootQ = root(q);
    if(rootP != rootQ){
      if(size[rootP] < size[rootQ]){
        parent[rootP] = rootQ;
        size[rootQ] = size[rootQ] + size[rootP];
      }else{
        parent[rootQ] = rootP;
        size[rootP] = size[rootP] + size[rootQ];
      }
      components--;
    }
  }

  //Retorna true si p,q estan conectados
  public boolean connected(int p, int q){
    return root(p) == root(q);
  }

  //Retorna el numero de componentes conexas
  public int getComponents(){
    return components;
  }
}

class Main {
  public static void main(String[] args){
    UnionFind uf = new UnionFind(5);
    uf.union(0, 2);
    uf.union(1, 0);
    uf.union(3, 4);

    //El numero de componentes es
    int comp = uf.getComponents();

    //Dos nodos estan conectados?
    boolean connected = uf.connected(0, 3);
```

```
  }
}
```

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).

Para hallar el MCM o MCD entre más de dos números se puede hacer de manera "iterativa": $MCD(a, b, c) = MCD(MCD(a, b), c)$.

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

//Dividir primero para evitar overflow en a*b
public static int mcm(int a, int b){
  return a * (b / mcd(a, b));
}
```

7.2. Criba de Eratóstenes

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

7.3. Factorización prima de un número

Se hace una Criba de Eratóstenes hasta $\lceil \sqrt{n} \rceil$. Luego, se verifica si n es divisible por los números primos hallados.

7.4. 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 retona *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.

7.5. 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.

Podemos verificar si un número x es primo o no probando si es divisible por los primos $\leq \sqrt{x}$. Partimos del hecho de que el primer número primo es el 2.

$$p_1 = 2$$

Suponga que se está buscando el p_{k+1} y se tiene como candidato a un número x. En el momento se conocen los primeros k primos, y $p_k < x$. Por ende, se conocen todos los primos menores a $\leq \sqrt{x}$. Usamos entonces los primos hallados anteriormente para verificar si x es primo o no. Si lo es, lo agregamos a la lista. Si no lo es, seguimos buscando. Paramos cuando hallemos n primos.

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 $O(n \log n)$.

```
public static void main(String[] args) {
  int n = 10;
  String v[] = new String[n];
  ArrayList<Integer> l = new ArrayList<Integer>();

  Arrays.sort(v);
  //Collections.sort() tambien ordena LinkedList
  Collections.sort(l);

  //En ArrayList .get() es O(1) pero en LinkedList es O(n), por lo cual hacer esto es fatal
  for(int i=0; i<l.size(); i++){
    System.out.println(l.get(i));
  }

  //La forma eficiente
  for(int k : l){
```

```
        System.out.println(k);
    }
}
```

8.2. Imprimir números decimales redondeados

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

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

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 imprimr. *BufferedWriter* es más rápido, nunca está de más usarlo.

```
public static void main(String[] args) throws
    IOException {
  BufferedReader br = new BufferedReader(new
      InputStreamReader(System.in));
  //Solo lee por lineas
  String s;
  while((s = br.readLine()) != null){
    int n = Integer.parseInt(br.readLine());
    String l[] = br.readLine().split(" ");
  }

  BufferedWriter bw = new BufferedWriter(new
      OutputStreamWriter(System.out));
  //No pone un salto de linea al final como si lo hace
      System.out.println(). Por tanto, se debe
      poner \n cuando sea necesario
  bw.write("Hola mundo\n");
  //El flush es el que realmente imprime en consola.
      En lo posible, hacer flush solo una vez, al
      final de todo
  bw.flush();
}
```