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.

```
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.

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

  hs.add("Hola");
  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. El siguiente ejemplo está con DFS pero funciona igual con BFS.

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.

```
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>();
  }

  int s = 0;
```

```
15
16    //Visita solo los nodos que son alcanzables desde el
          ↪ nodo 's'
17    dfs(s);
18
19    //Con el vector 'seen' vemos cuales son estos nodos
20    for(int i=0; i<n; i++){
21      if(seen[i]){
22        //'i' es alcanzable desde 's'
23      }
24    }
25
26    //Si queremos visitar todos los nodos
27    for(int u=0; u<n; u++){
28      if(!seen[u]){
29        //Si no hemos visitado 'u', hacer DFS en 'u'
30        dfs(u);
31      }
32    }
33  }
34
35  private static void dfs(int u){
36    seen[u] = true;
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        dfs(v);
42      }
43    }
44  }
45
46  private static void bfs(int u){
47    seen[u] = true;
48    Queue<Integer> q = new LinkedList<Integer>();
49    q.add(u);
50    while(!q.isEmpty()){
51      u = q.poll();
52      int len = g[u].size();
53      for(int i=0; i<len; i++){
54        int v = g[u].get(i);
55        if(!seen[v]){
56          seen[v] = true;
57          q.add(v);
58        }
59      }
60    }
61  }
```

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
11    g = new ArrayList[n];
12    for(int i=0; i<n; i++){
13      g[i] = new ArrayList<Integer>();
14    }
15
16    int s = 0;
17    shortestHop(s);
18    //Despues de llamar este metodo, en dist[i] esta la
          ↪ 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
```

```
    //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.

```
  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 = topoDfs(u);
      }
    }

    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){
        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 como 'negro'
    seen[u] = 2;
    topoSort.addFirst(u);
    return sinCiclo;
  }
```

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
6    int values[] = {15, 10, 9, 5};
7    int weights[] = {1, 5, 3, 4};
8
9    //Tener cuidado: En la matriz los items se numeran
       ↪ 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
       ↪ los values y weights
13   for (int i=1; i<=n; i++) {
14     for (int x=0; x<=W; x++) {
15       if (weights[i-1] > x) {
16         A[i][x] = A[i-1][x];
17       } else {
18         int p = A[i-1][x];
19         int q = A[i-1][x-weights[i-1]] + values[i-1];
20         A[i][x] = (p > q) ? p : q;
21       }
22     }
23   }
24
25   //El valor maximo que se puede obtener es A[n][W]
26   int solution = A[n][W];
27
28   //Si se quiere determinar cuales items se incluyeron
29   boolean chosen[] = new boolean[n];
30   int i = n;
31   int j = W;
32
33   while(i>0){
34     if(A[i][j] == A[i-1][j]){
35       i--;
36     }else{
37       chosen[i-1] = true;
38       i--;
39       j = j-weights[i];
40     }
41   }
42
43   //Si chosen[i]==true es porque i se incluyo
44 }
```

5. 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;
```

```
  }

  //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;
  }
}
```

6. 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. Otros

7.1. 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>();

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

  //Esto llama la funcion .size() y .get() cada vez
  //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);
  }
}
```

7.2. 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  }
```