# Algoritmos y Estructuras de Datos I

Bibliografía:

- Programa de teoría (diapositivas), A.E.D. 1, Facultad de Informática, Universidad de Murcia.
- Texto guía, anónimo.
- Wikipedia, la Enciclopedia Libre (https://es.wikipedia.org/).
- Competitive Programming 3: The New Lower Bound of Programming Contests (2013), Steven & Felix Halim.

# Capítulo 1

# Abstracción y especificación

Una **abstracción** consiste en dejar a un lado lo irrelevante y centrarse en lo importante. Distinguimos **niveles de abstracción** según el nivel de detalle. El **diseño mediante abstracciones** consiste en identificar los subproblemas de cierto problema, especificar cada uno de forma abstracta, resolverlos de forma separada, unir las soluciones y verificar la solución para el problema original. Esto se hace de forma recursiva.

Junto con el concepto de abstracción viene el de **especificación** de dicha abstracción.

**Mecanismos de abstracción**:

- **Por especificación**: Nos quedamos con la descripción del efecto o resultado, independientemente de cómo calcularlo (**principio de ocultación de la implementación**). Da lugar a la **encapsulación**, agrupar un conjunto de operaciones o tipos relacionados en un mismo módulo o paquete.
- **Por parametrización**: El significado de la abstracción no es fijo, sino que depende de parámetros. En la **parametrización de tipo**, el parámetro es un tipo de datos.

## 1.1. Especificaciones informales

Una **especificación** es la descripción de una abstracción, bien textualmente mediante el lenguaje natural (especificación **informal**) o bien con una notación precisa de las propiedades de la abstracción (especificación **formal**), mediante una **notación**. A continuación vemos los tipos de abstracciones y una notación propuesta para su especificación informal.

- **Funcionales**: Abstraemos un trozo de código en una **rutina**, **procedimiento** o **función**, al que asignamos un nombre. Su especificación indica los parámetros de entrada, los de salida y el efecto que tendrá su ejecución.

  Especificación informal

  **Operación** *nombre* [[*T*,...:*TipoDeTipo*; ...]] ([**ent** *v1*, ...:*Tipo*1; *vn*, ...:*Tipo2*; ...[;]] [**sal** *tipoResultado*|)
  **Require:** *Opcional, requisitos y restricciones de uso de la operación.*
  **Modifica:** *Opcional, lista de parámetros de entrada que pueden ser modificados.*
  **Calcula:** *Descripción del resultado de la operación.*

Donde el *TipoDeTipo* suele ser tipo. Tanto en la especificación informal como en el pseudocódigo, los tipos tanto de los parámetros de tipo como los propios de la función pueden depender de parámetros anteriores.

- **De datos**: Se abstrae un dominio de valores, junto con operaciones sobre ese dominio con **ocultación de la implementación**, en un **tipo abstracto de datos** (**TAD**). Un **tipo contenedor** es un TAD compuesto por un número arbitrario de valores de otro tipo. Generalmente están **parametrizados**: dependen de un parámetro que indica el tipo de objetos almacenados.

  Especificación informal

  **TAD** *nombre* **es** *listaDeOperaciones*
  **Descripción**
  *Descripción del significado, comportamiento y modo de uso del TAD.*
  **Operaciones**
  *Especificación informal de cada operación de la lista, según el punto anterior.*
  **Fin** *nombre*.

  Así, un **tipo de datos** es un concepto de implementación que debe ajustarse al comportamiento de un TAD, y una **estructura de datos** es la disposición en memoria de estos datos.

- **De iteradores**: Permiten realizar un recorrido de forma abstracta sobre los elementos de una colección. La especificación informal puede ser:

  - Similar a la de una abstracción funcional, cambiando **Operación** por **Iterador** y añadiendo un parámetro de entrada de tipo Operacion. La función puede devolver un valor si, por ejemplo, el iterador filtra una colección de valores de un cierto tipo.
  - Similar a la de un TAD, cambiando **TAD** por **TipoIterador** y añadiendo operaciones como Iniciar, Actual, Avanzar y EsUltimo.

La abstracción en lenguajes de programación generalmente consiste en ofrecer características más próximas al concepto teórico de TAD. Los lenguajes primitivos como Fortran o BASIC ofrecían un conjunto predefinido de tipos elementales, pero no permitían definir nuevos tipos, con lo que el programador debía definir y manejar las variables adecuadas de forma manual. Posteriormente, lenguajes como C y Pascal permiten agrupar un conjunto de variables bajo un mismo nombre, aunque no hay ocultación. Los módulos o paquetes permiten agrupar bajo un mismo nombre una serie de tipos y operaciones relacionados, y en algunos casos como en Modula-2, estos ofrecen ocultación de la implementación: los tipos definidos en un módulo sólo se pueden usar a través de las operaciones de este.

Finalmente, en la programación orientada a objetos surge el concepto de clase, que es al mismo tiempo un tipo definido por el usuario y un módulo donde se encapsulan sus datos y operaciones, y soportan ocultación. Un objeto es una instancia de una clase. El **principio de acceso uniforme** consiste en que los atributos y operaciones de una clase están conceptualmente al mismo nivel, y se accede con *objeto*`.`*atributo* u *objeto*`.`*operación*`(...)`. Algunos lenguajes orientados a objetos permiten parametrización de tipo, dando lugar a TAD genéricos o parametrizados.

Una implementación es **robusta** si se autoprotege de valores inconsistentes de entrada, bien lanzando una excepción o mediante programación por contrato, en la que el usuario se

compromete, de forma comprobable, a no pasar un valor inconsistente según la especificación de la función.

## 1.2. Especificaciones formales

Una **especificación formal** define un TAD u operación de forma precisa, permitiendo la verificación automática y formal de la corrección del programa y el uso de la especificación en distintos ámbitos. Estas especificaciones pueden llegar a ser ejecutables. Distinguimos entre el **método axiomático** o **algebraico**, que establece el significado de las operaciones de forma implícita mediante **axiomas** que especifican sus relaciones, y el **método constructivo** u **operacional**, en el que cada operación se define por sí misma con significado explícito.

Veremos un método constructivo al que llamaremos método axiomático, y que escribimos en lenguaje Maude, describiendo un TAD del siguiente modo:

| Especificación formal | Maude |
|---|---|
| **NOMBRE**<br>*NombreTAD* [[*T1*,...]] | `fmod NOMBRETAD is` |
| **CONJUNTOS**<br>*CONJ* *Descripción*<br>... | `protecting TADIMPORTADO .`<br>`sort CONJ .`<br>`subsort A <B .`<br>... |
| **SINTAXIS**<br>*operación*: [*conjPar1*×...] → *conjResultado*<br>... | `op operación : [conjPar11 × ...] ->conjResultado1 .`<br>`ops op1 ... : [conjPar21 × ...] ->conjResultado2 .`<br>... |
| **SEMÁNTICA**<br>∀ *var1*, ... ∈ *CONJ*<br>*ExprA* = *ExprB*<br>... | `var v : CONJ .`<br>`vars v1 ... : CONJ .`<br>...<br>`eq ExprA = ExprB .`<br>... |
| | `endfm` |

El nombre de los conjuntos suele ser de una letra, y debe haber un conjunto asociado al TAD, y si el TAD es genérico, uno para cada parámetro con el mismo nombre dado en la sección anterior (*T1*, etc.). También, en la especificación se declaran conjuntos asociados a otros TADs pero que se usan en este, mientras que en Maude se usa `protecting` para importar todos los conjuntos, junto con sus operaciones, de otro TAD. Para tipos finitos en la especificación, si en la *Descripción* se añade al final {*valor1, ...*}, estos valores se pueden usar en la semántica.

Entre las operaciones encontramos **constructores**, operaciones cuya semántica no se define y que sirven para definir elementos del TAD, y tales que, si es posible, un elemento del TAD debería tener representación única en términos de estos. El resto son de **modificación** o de **consulta** según si devuelven un elemento del conjunto asociado al TAD o de otro tipo, respectivamente.

Si una operación que normalmente devuelve valores de tipo T no esta definida para cierta entrada, se crea un conjunto M de mensajes cuya descripción contiene al final {"*Texto mensaje*

*1''*, ...}, y el conjunto resultado de la operación se declara T∪M, mientras que en Maude se crea un tipo de errores `NoT` con un constructor sin parámetros para cada tipo de error y se usa `subsort No`*`T`* `<`*`T`* para indicar que toda operación que toda operación que devuelva un valor de *T* también puede devolver uno de `No`*`T`*.

La semántica consiste en una serie de reglas que, dada una expresión, Maude va comprobando de arriba a abajo sucesivamente para ver si parte de la expresión coincide con *ExprA* y sustituirla en tan caso por *ExprB* hasta que no queden sustituciones por hacer, momento en que la expresión debería estar en términos de constructores. Una expresión tiene sintaxis *op*`[(`*par1*`, ...)]`, donde los parámetros de la operación son a su vez otras expresiones. *ExprB* o alguna de sus subexpresiones puede ser **SI** *condición* $\implies$ *valorSiCierto* | *valorSiFalso*, o en Maude, `if` *condición* `then` *valorSiCierto* `else` *valorSiFalso* `fi`.

En Maude, fuera de la definición de un TAD podemos usar `in` *nombre* `.` para leer todo lo que hay en un fichero llamado *nombre* o *nombre*`.maude` o `red` *expr* para reducir una expresión según la semántica de las operaciones. Un fallo en Maude puede tiene consecuencias impredecibles porque Maude es una mierda.

Si el orden de las reglas no es relevante las llamamos axiomas[1], y estos deben cumplir las propiedades de **completitud** (deben ser suficientes para deducir el significado de cualquier expresión) y **corrección**; dicho de otra forma, se deben considerar todos los casos al definir cada operación no constructora.

## 1.3. Órdenes de ejecución

Medimos el tiempo que tarda en ejecutarse un algoritmo con una función $t : \mathbb{N} \to \mathbb{R}^{\geq 0}$ para cada uno de los casos mejor, peor y promedio, que toma el tamaño del problema medido de algún modo y devuelve el tiempo que tarda en realizarse en cierta máquina, el número de instrucciones total o el de un tipo de instrucción que sabemos que es la más relevante. En la práctica se usan **notaciones asintóticas**:

- $f(x) = O(g(x)) : \iff \exists x_0, k > 0 : \forall x \geq x_0, |f(x)| \leq k|g(x)|$.
- $f(x) = \Omega(g(x)) : \iff \exists x_0, k > 0 : \forall x \geq x_0, kg(x) \leq f(x)$.

Estas definiciones sólo son válidas cuando $x$ tiende a $+\infty$.

[1] Cada vez que llamas axiomas a algo que depende del orden muere un matemático y un gatito.

# Capítulo 2

# Conjuntos y diccionarios

## 2.1. Conjuntos

Un conjunto es una colección no ordenada de elementos distintos. El TAD Conjunto[$T$] define conjuntos finitos con elementos de un cierto tipo. Operaciones comunes:

$$\begin{array}{lll}
\text{Vacío} :\to C & \text{Unión} : C \times C \to C & \text{Intersección} : C \times C \to C \\
\mapsto \emptyset & (a,b) \mapsto a \cup b & (a,b) \mapsto a \cap b \\
\\
\text{Diferencia} : C \times C \to C & \text{Combina} : C \times C \to C & \text{Miembro} : T \times C \to B \\
(a,b) \mapsto a \backslash b & (a,b) \to a \dot{\cup} b & (x,a) \mapsto x \in a \\
\\
\text{Inserta} : T \times C \to C & \text{Suprime} : T \times C \to C & \text{Igual} : C \times C \to B \\
(x,a) \mapsto a \cup \{x\} & (x,a) \mapsto a \backslash \{x\} & (a,b) \mapsto a = b
\end{array}$$

Si además tenemos un orden total sobre $T$, podemos definir $\text{Min} : C \to T$ y $\text{Max} : C \to T$ para conjuntos no vacíos. Implementaciones básicas:

- **Tabla de booleanos**, donde cada elemento de T se representa con un bit 1 si pertenece al conjunto o 0 en caso contrario. La unión, intersección, diferencia y comprobación de igualdad son $O(n)$ con $n \coloneqq |T|$, mientras que la inserción y eliminación y comprobación de pertenencia son $O(1)$. Las operaciones son muy sencillas de implementar y no hace falta memoria dinámica, pero el tamaño de un conjunto es proporcional a $|T|$ y por tanto solo es adecuado cuando $|T|$ es pequeño.
- **Lista de elementos**. La unión, intersección y diferencia son $O(mn)$ siendo $m$ y $n$ el tamaño de los conjuntos de entrada, mientras que la comprobación de pertenencia, inserción y eliminación son $O(n)$. La comprobación de igualdad es $O(1)$ en el caso mejor ($m \neq n$) y $O(n^2)$ en el peor. Se usa un espacio proporcional al del conjunto representado, pero es más complejo de implementar, y las operaciones son menos eficientes si el conjunto es grande o $|T|$ es pequeño.
- **Lista de elementos ordenada**. La unión, intersección y diferencia pasan a ser $O(\text{máx}\{m,n\})$ en el caso mejor y $O(m+n)$ en el peor, y la comprobación de igualdad en el caso peor pasa a ser $O(n)$.

## 2.2. Diccionarios

Una asociación es un par clave-valor, y un diccionario es un conjunto de asociaciones sin más de un valor para una misma clave. Nos limitamos a Diccionario[$T_k$, $T_v$], diccionarios finitos con claves de un cierto tipo $T_k$ y valores de tipo $T_v$. Operaciones comunes:

$$\begin{array}{rl} \mathsf{Inserta} : T_k \times T_v \times D \to D & \mathsf{Consulta} : T_k \times D \to T_v \\ (k, v, d) \overset{k \notin \mathrm{Dom}(d)}{\mapsto} D \cup \{(k, v)\} & (k, d) \overset{k \in \mathrm{Dom}(d)}{\mapsto} d(k) \end{array}$$

$$\begin{array}{rl} \mathsf{Suprime} : T_k \times D \to D & \mathsf{Vacío} :\to D \\ (k, d) \mapsto \{(a, b) \in d \mid a \neq k\} & \mapsto \emptyset \end{array}$$

La representación más sencilla es mediante una lista de asociaciones.

## 2.3. Tablas de dispersión

Permiten una representación más eficiente de conjuntos y diccionarios mediante una lista contigua de $M$ elementos y una **función *hash*** o **de dispersión** $h : T \to \{0, \ldots, M-1\}$. La idea es que, para diccionarios, $h$ solo dependa de la clave de la asociación, no del valor. Se dice que $x \neq y \in T$ son sinónimos si $h(x) = h(y)$. Existen dos formas de dispersión:

- **Abierta**: Los elementos de la tabla son apuntadores a listas (enlazadas) de elementos, llamadas **cubetas**, que contienen los elementos $\{e \in c \mid h(e) = k\}$, siendo $c$ el conjunto y $k$ el índice de la cubeta.
  El tamaño de las cubetas en un reparto uniforme (lo ideal) es $\frac{n}{M}$, siendo $n$ el número de elementos del conjunto, luego en este caso la búsqueda es $O(\frac{n}{M})$ y la inserción es $O(1 + \frac{n}{M})$. El uso de memoria es el de $M + n$ apuntadores y $n$ elementos. Así, a más cubetas las operaciones son más rápidas, pero se usa más memoria.

- **Cerrada**: Los elementos de la tabla son del tipo $T$, y si al ir a insertar un elemento $e$ en el índice $h(e)$ este ya está ocupado se dice que ocurre una **colisión**, y se hace una **redispersión**: se aplican sucesivamente los elementos de una sucesión de funciones $(h_n : T \to \{0, \ldots, M-1\})_n$ sobre $e$ hasta que devuelva un índice de la tabla no ocupado, donde se guarda $e$.
  Llamamos **cadena** o **secuencia de búsqueda** a la secuencia $h(e), h_1(e), \ldots$ de posiciones recorridas en este proceso. Al consultar un elemento se recorre esta secuencia hasta encontrar el elemento o una celda vacía, que indica que este no está. En la eliminación, para no romper la secuencia, se sustituye el elemento por una marca de eliminado, que se trata como celda libre en la inserción pero no en la búsqueda, o bien se mueven elementos cuya secuencia de búsqueda pase por la posición eliminada.
  Como la probabilidad de colisión es $\frac{n}{M}$ y se ha de encontrar un elemento libre, el coste de una inserción es $O(\frac{1}{1-\frac{n}{M}})$, que tiende a infinito cuando $n \to M$, mientras que el de búsqueda es de $O(\frac{1}{1-\frac{n-1}{M}})$. El uso de memoria es el de $M$ elementos, o el de $M$ apuntadores más $n$ elementos si se decide que la tabla almacene apuntadores.

Para evitar la pérdida de eficiencia cuando $n$ aumenta, la tabla se puede **reestructurar**, creando una nueva con distinto $M$ y copiando los elementos, por ejemplo cuando $n > 2M$

en dispersión abierta o cuando $n > \frac{3}{4}M$ en cerrada. Para que las tablas de dispersión sean eficientes se debe usar una buena función, que reparta los elementos de la forma más aleatoria pero a la vez sea fácil de calcular.

Métodos para enteros:

- **División**: $h(k) \coloneqq k \mod M$, siendo $M$ normalmente primo.
- **Multiplicación**: $h(k) \coloneqq \lfloor Ck \rfloor \mod M, C \in \mathbb{R}$.
  Variante: $h(k) \coloneqq \lfloor \mathrm{d}(\frac{Ak}{W})M \rfloor, \mathrm{mcd}(A, K) = 1$, donde $\mathrm{d}(x)$ es la parte decimal de $x$.
- **Centro del cuadrado**: $h(k) \coloneqq \lfloor \frac{k^2}{C} \rfloor \mod M$.

Para secuencias:

- **Suma**: $h(k) \coloneqq \sum_i x_i \mod M$. Muchas colisiones.
- **Suma posicional**: $h(k) \coloneqq \sum_i k^i x_i \mod M$, siendo $k$ normalmente primo.
- **Plegado** (*folding*): $h(k) \coloneqq \sum_{i=0}^{\lfloor n/p \rfloor} \prod_{j=1}^{p} x_{ip+j} \mod M$, tomando $x_k = 1 \forall k > n$.
- **Extracción**: $h(k) \coloneqq x_{n_1} \cdots x_{n_k} \mod M$.
- **Iterativos**: Tomar un valor inicial y, para cada $x_i$, en orden, multiplicar por un valor base y combinar el resultado con $x_i$ mediante alguna operación. Devolver esto módulo $M$. Así, la suma posicional toma como valor inicial, base y combinación, respectivamente, 0, $k$ y +; djb2 toma 5381, 33 y +; djb2a toma 5381, 33 y XOR, y FNV-1 toma 2166136261, 16777619 y XOR.

Funciones de redispersión:

- **Lineal**: $h_i(k) \coloneqq h(k) + i \mod M$. Sufre **agrupamiento**: Si se llenan varias cubetas consecutivas y hay colisión, se debe consultar todo el grupo.
- **Con saltos**: $h_i(k) \coloneqq h(k) + Ci \mod M$. Sufre agrupamiento.
- **Cuadrática**: $h_i(k) \coloneqq h(k) + D(i) \mod M$ con

$$D(i) = \begin{cases} \left(\frac{i+1}{2}\right)^2 & \text{si } x \text{ es impar} \\ -\left(\frac{i}{2}\right)^2 & \text{si } x \text{ es par} \end{cases}$$

  Funciona cuando $M = 4R + 3$ para algún $R \in \mathbb{N}$.
- **Doble**: $h_i(k) = h(k) + C(k)i \mod M$ para $C : T \to \{1, \ldots, M-1\}$.

# Capítulo 3

# Árboles

## 3.1. Árboles Trie

Sea $A$ un conjunto llamado **alfabeto**, un árbol Trie o **de prefijos** es aquél en que la raíz del árbol representa una cadena vacía (()) y un nodo puede tener un hijo etiquetado para cada elemento de $A \cup \{\$\}$, con la condición de que un nodo etiquetado con \$, la **marca de fin**, es hoja. Llamamos **palabra** a la tupla de etiquetas desde la raíz hasta un nodo con \$, sin contarlo. Un ArbolTrie[$A$] ($T$) es un apuntador a un NodoTrie[$A$] ($N$), un Diccionario[$A$,ArbolTrie[$A$]]. Operaciones:

$$\begin{array}{lll} \mathsf{Consulta} : N \times A \to T & \mathsf{Inserta} : N \times A \to N & \mathsf{PonMarca} : N \to N \\ (n,x) \overset{x \in Dom(n)}{\mapsto} n(x) & (n,x) \overset{x \notin Dom(n)}{\mapsto} n \cup \{(x,\emptyset)\} & n \mapsto n \cup \{(\$,\emptyset)\} \\ \\ \mathsf{QuitaMarca} : N \to N & \mathsf{HayMarca} : N \to B & \\ n \mapsto n \backslash \{(\$,\emptyset)\} & n \mapsto (\$,\emptyset) \in n & \end{array}$$

Los objetos devueltos por Consulta son apuntadores, luego se puede insertar sobre ellos. Otra operación es Inserta : $T \times \bigcup_{n=1}^{\infty} A^n \to T$, que inserta una palabra en un árbol. Podemos implementar un nodo trie:

- Como una tupla de apuntadores en la que cada índice corresponde a un elemento de $A \cup \{\$\}$ de forma biyectiva. Permite un acceso a los valores en $O(n)$, siendo $n$ la longitud de la palabra, pero si $p$ es el total de prefijos y $d = |A \cup \{\$\}|$, esto requerirá la memoria usada por $(p+1)d$ apuntadores (mucha).
- Como una lista enlazada de asociaciones. Presenta un uso de memoria razonable, el usado por $4p+2$ apuntadores y $2p+1$ caracteres de $A$, pero el acceso es $O(ns)$ en promedio, siendo $s$ la longitud media de las listas (normalmente despreciable).

Si el total de la longitud de las palabras entre el número de prefijos es grande, digamos mayor que 6, las palabras comparten muchos prefijos y suele ser conveniente una representación como tupla, mientras que de lo contrario es mejor usar una lista enlazada.

## 3.2. Relaciones de equivalencia

Definimos el TAD RelEquiv[$T$] como el de las relaciones de equivalencia en un Conjunto[$T$]. Operaciones:

- Crear : $C \to R$: Crea una relación de equivalencia con $x \approx y : \iff x = y$.
- Encuentra : $R \times T \to R$: Devuelve un representante canónico de la clase del elemento dado.
- Unión : $R \times T \times T \to R$: Hace equivalentes a los dos elementos dados de $T$, combinando sus clases de equivalencia. En el análisis posterior supondremos que los elementos dados son representantes canónicos.

Representaciones básicas:

- Mediante una tabla en la que cada índice corresponde a un elemento del conjunto y contiene el índice de su representante canónico. La búsqueda de clase de equivalencia es $O(1)$ pero la unión es $O(n)$ con $n$ el cardinal del conjunto.
- Mediante listas (enlazadas) de clases, con una lista para cada clase cuyo nombre es el del representante canónico. La unión es $O(1)$ con una representación adecuada de las listas, pero la búsqueda es $O(n)$ con $n$ el cardinal del conjunto.

Otra forma es una estructura de árboles disjuntos, una representación por tabla en la que cada índice contiene el índice del padre en su árbol (o un valor especial si es la raíz), cada árbol es una clase de equivalencia y el representante canónico es la raíz del árbol. De esta forma la unión es $O(1)$ y la búsqueda es $O(\log n)$ en caso promedio, siendo $n$ el cardinal de la clase de equivalencia, aunque sigue siendo $O(n)$ en el caso peor. Para evitar esto:

- **Balanceo**: Al unir los árboles $a$ y $b$, podemos poner como hijo al menos alto, para lo cual guardamos en la raíz (indicando apropiadamente que es la raíz, por ejemplo, con números negativos) la altura del árbol.
- **Compresión de caminos**: Al hacer una búsqueda, poner la raíz del árbol como el padre del elemento buscado, y del resto de elementos recorridos.

Así, la búsqueda es $O(1)$ en el caso mejor y $O(\log n)$ en el peor.

## 3.3. Árboles AVL

Un **árbol binario de búsqueda** (ABB) es un árbol binario en que, para cada nodo, los elementos del subárbol izquierdo son menores que el valor del nodo y los del derecho son mayores. El recorrido ordenado es $O(n)$ y la búsqueda e inserción son $O(\log n)$ en el caso promedio, pero $O(n)$ en el peor.

Un **ABB perfectamente balanceado** es aquel en que, para cada nodo, la cantidad de elementos en sus dos subárboles difiere como mucho en 1. La búsqueda es $O(\log n)$ en el peor caso, pero en este caso la inserción es $O(n)$.

Un **árbol balanceado** o **Adelson-Velskii-Landis** (**AVL**) es un ABB en que, para cada nodo, la altura de sus subárboles difiere como mucho en 1. La búsqueda es $O(\log n)$ en el caso

peor y se hace igual que en un ABB. La inserción y eliminación requieren **rebalancear** el árbol a la vuelta de la llamada recursiva usada para insertar o eliminar el elemento, lo que se hace almacenando la altura de cada nodo en el propio nodo y mediante **rotaciones**:

- **RSI** (rotación simple a la izquierda): Sobre un nodo `A`, el proceso es:
  `B` $\leftarrow$ `LEFT[A]`, `LEFT[A]` $\leftarrow$ `RIGHT[B]`, `RIGHT[B]` $\leftarrow$ `A`,
  `HEIGHT[A]` $\leftarrow 1 +$ máx{`HEIGHT[LEFT[A]]`, `HEIGHT[RIGHT[A]]`},
  `HEIGHT[B]` $\leftarrow 1 +$ máx{`HEIGHT[LEFT[B]]`, `HEIGHT[A]`} y `A` $\leftarrow$ `B`.
- **RSD** (rotación simple a la derecha): Simétrico de RSI.
- **RDI** (rotación doble a la izquierda): Equivale a `RSD(LEFT[A])` seguido de `RSI(A)`.
- **RDD** (rotación doble a la derecha): Simétrico de RDI.

Tras la inserción, si $|$`HEIGHT[LEFT[A]]` $-$ `HEIGHT[RIGHT[A]]`$| > 1$ se rebalancea, con 4 situaciones según cuál de las ramas que hay 2 niveles bajo `A` tiene mayor altura (dónde se ha insertado). Si es la rama II (izquierda-izquierda) se hace RSI, para DD se hace RSD, para ID (izquierda y después derecha) se hace RDI y para DI se hace RDD. Ninguno de los casos cambia la altura final del árbol.

Para la eliminación, si el nodo a eliminar es hoja, se elimina directamente. Si solo tiene un hijo, se conecta el padre del nodo eliminado con este hijo. Si tiene dos hijos, se escoge el más a la derecha del subárbol izquierdo, o el más a la izquierda del derecho, para sustituirlo.

Tras esto puede ser necesario rebalancear, para lo cual hay 3 casos de eliminación en el subárbol izquierdo según las alturas $h_l$ y $h_r$ de los subárboles respectivos izquierdo y derecho del hijo derecho, más sus 3 casos simétricos de eliminación en el derecho. Así, si $h_l = h_r$ se aplica $RSD$ y la altura final del árbol no cambia; si $h_l < h_r$ se aplica $RSD$ y la altura disminuye en 1, y si $h_l > h_r$ se hace RDD y la altura disminuye en 1. Así, la inserción y eliminación son $O(\log n)$ en todos los casos.

## 3.4. Árboles B

Un árbol B de orden $p$ o árbol $p$-ario de búsqueda es aquel en que cada nodo está formado por hasta $p-1$ claves y $p$ hijos (siempre una clave menos que hijos), la raíz no tiene hijos o tiene entre 2 y $p$, los nodos internos (ni raíz ni hoja) tienen entre $\lceil \frac{p}{2} \rceil$ y $p$ hijos y los nodos hoja aparecen todos al mismo nivel (condición de balanceo) y, para cada nodo con $M$ hijos, las claves del primer subárbol son todas menores que la primera de este, los del $M$-ésimo son mayores que las de la última clave, las del $k$-ésimo, $1 < k < M$, están entre la $(k-1)$-ésima clave y la $k$-ésima, y las claves de un mismo nodo están ordenadas de menor a mayor. Se usan mucho en bases de datos, ajustando $p$ para que cada nodo esté en un bloque de disco minimizando las operaciones de E/S, y existen variantes como $B+$, $B^*$, etc.

La búsqueda es igual que en los árboles binarios y es $O(\frac{\log n}{\log p})$, siendo $n$ el número de elementos. Para la inserción se empieza buscando el nodo hoja donde se debería colocar la entrada. Si quedan huecos libres, se inserta. Si no, la hoja (que ya tiene $p-1$ claves) se divide en 2 hojas con $\lceil \frac{p-1}{2} \rceil$ y $\lfloor \frac{p-1}{2} \rfloor$ claves, añadiendo, de los $p$ valores, el que queda en medio en el nodo padre, repitiendo el proceso en este recursivamente si es necesario.

# Capítulo 4

# Grafos

Un **grafo dirigido** o **digrafo** es un par $(V, E)$ formado por un conjunto de **vértices** o **nodos** $V \neq \emptyset$ y un conjunto de **aristas** $E \subseteq \{(a, b) \in V \times V \mid a \neq b\}$, mientras que uno **no dirigido** es un par $(V, E)$ con $V \neq \emptyset$ y $E \subseteq \{x \in \mathcal{P}(V) \mid |x| = 2\}$. Algunas definiciones permiten **bucles**, aristas de un nodo a sí mismo, y entonces basta que sea $E \subseteq V \times V$ para que el grafo sea dirigido o que $E \subseteq \{x \in \mathcal{P}(V) \mid |x| \in \{1, 2\}\}$ para que sea no dirigido. En adelante identificamos los grafos no dirigidos $(V, E)$ con los correspondientes grafos dirigidos $(V, \bigcup_{\{u,v\} \in E} \{(u, v), (v, u)\})$.

Un **grafo etiquetado** es una terna $(V, E, \sigma)$ donde $(V, E)$ es un grafo (dirigido o no) y $\sigma : E \to X$ es una función que relaciona cada arista con una **etiqueta**. Un **grafo con pesos** es un grafo etiquetado $(V, E, \sigma)$ con $\sigma : E \to \mathbb{R}$. Un grafo es **finito** si $V$ lo es.

Un **subgrafo** de un grafo $G \coloneqq (V, E)$ es un grafo $(V', E')$ con $V' \subseteq V$ y $E' \subseteq E$. Un subgrafo de un grafo etiquetado $G \coloneqq (V, E, \sigma)$ es un grafo etiquetado $(V', E', \sigma|_{E'})$ donde $(V', E')$ es subgrafo de $(V, E)$.

Un nodo $u \in V$ es **adyacente a** $v \in V$ si $(v, u) \in E$, y es **adyacente de** $v$ si $(u, v) \in E$. Un **camino** de $u$ a $v$ es una secuencia $(w_1, \ldots, w_q)$ de elementos de $V$ con $(w_{i-1}, w_i) \in E \forall i \in \{2, \ldots, q\}$, y es **simple** si para $i, j \in \{1, \ldots, q\}$ con $i \neq j$ e $\{i, j\} \neq \{1, q\}$ se tiene $w_i \neq w_j$. Si $w_1 = w_q$, el camino es un **ciclo**. Un grafo es **acíclico** si no contiene ciclos.

Es **conexo** o **conectado** si existe un camino entre cualquier par de vértices, y es **fuertemente conexo** si además es dirigido. Un **componente conexo** de un grafo (también **fuertemente conexo** si el grafo es dirigido) es un subgrafo conexo que no es subgrafo de ningún otro subgrafo conexo del grafo.

Un grafo es **completo** si existe una arista entre cualquier par de vértices. En un grafo no dirigido $(V, E)$, el **grado** de un vértice $v \in V$ es el número de arcos adyacentes a él ($|\{X \in E \mid v \in X\}|$), mientras que en uno dirigido $(V, A)$, definimos el **grado de entrada** de un vértice $v \in V$ como $|\{(a, b) \in A \mid b = v\}|$ y el **grado de salida** como $|\{(a, b) \in A \mid a = v\}|$.

Operaciones elementales:

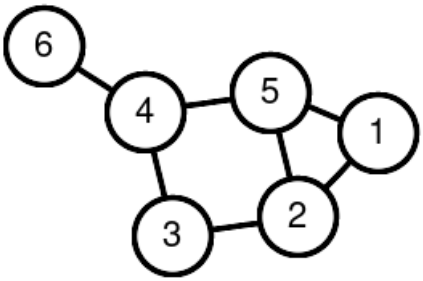


Figura 4.1: Grafo no dirigido.

$$\begin{aligned}
\mathsf{Vacío} &:\to G & \mathsf{Crear} &: \mathbb{N} \to G \\
&\mapsto (\emptyset, \emptyset) & n &\mapsto (\{1, \dots, n\}, \emptyset) \\
\mathsf{IntertarNodo} &: G \times \mathcal{U} \to G & \mathsf{InsertarArista} &: G \times (\mathcal{U} \times \mathcal{U}) \to G \\
((V, E), v) &\mapsto (V \cup \{v\}, E) & ((V, E), (a, b)) &\overset{a,b \in V}{\mapsto} (V, E \cup \{e\}) \\
\mathsf{EliminarNodo} &: G \times \mathcal{U} \to G & \mathsf{EliminarArista} &: G \times (\mathcal{U} \times \mathcal{U}) \to G \\
((V, E), v) &\mapsto (V \backslash \{e\}, \{(a, b) \in E \mid a, b \neq v\}) & ((V, E), e) &\mapsto (V, E \backslash \{e\}) \\
\mathsf{ConsultarArista} &: G \times (\mathcal{U} \times \mathcal{U}) \to B \\
((V, E), (a, b)) &\mapsto (a, b) \in A
\end{aligned}$$

Donde $\mathcal{U}$ es el conjunto de los vértices. Definimos también iteradores sobre los nodos adyacentes a un cierto $v \in V$ o adyacentes de este.

## 4.1. Representación

Podemos dibujar un grafo indicando los nodos mediante círculos con etiqueta (o cualquier otra forma) y uniendo con una línea los vértices de cada arista. Si el grafo es dirigido, cada línea es una flecha que apunta al segundo elemento del par. Si es etiquetado, cada línea tendrá su etiqueta indicada.

En un ordenador podemos representar un grafo finito $(V \coloneqq \{1, \dots, n\}, E)$ o $(V \coloneqq \{1, \dots, n\}, E, \sigma : E \to X)$ mediante:

- **Matrices de adyacencia**: Matriz $n \times n$ de booleanos $((i, j) \in E)_{ij}$. Si el grafo es etiquetado, cuando $(i, j) \in E$ se añade en la celda la etiqueta, lo que en general se hace tomando un valor $c \notin X$ que representa que $(i, j) \notin E$ y entendiendo que cualquier otro valor implica $(i, j) \in E$. Si el grafo es no dirigido, la matriz es simétrica. Las operaciones son sencillas y el acceso a una arista dada es $O(1)$, pero si $|E| \ll n^2$ (**matriz escasa**) se usa mucha memoria ($O(n^2)$).
- **Listas de adyacencia**: $n$ conjuntos $(C_1, \dots, C_n)$ (representados como listas enlazadas en una lista contigua) de los que $C_i = \{j \mid (i, j) \in E\}$. Si el grafo es etiquetado, $C_i =$

$\{(j, \sigma(i,j)) \mid (i,j) \in E\}$. Esto es más adecuado si $|E| \ll n^2$, pues la memoria usada es $O(n + |E|)$, pero la representación es más compleja y es ineficiente para encontrar las aristas adyacentes de un cierto nodo.

En adelante, salvo que se indique lo contrario, suponemos un grafo $(V := \{1, \ldots, n\}, E, \sigma : E \to X)$, y que las variables en pseudocódigo se inicializan con su valor por defecto.[1]

## 4.2. Recorridos

Ambos son $O(|V|^2)$ con matrices de adyacencia y $O(|V| + |E|)$ con listas de adyacencia.

**Búsqueda primero en profundidad**

```
var visitado: array [1..n] de booleano()
operación dfs(u: [1..n])
    visitado[u] := verdadero; visitar(u)
    para cada nodo v adyacente a u hacer si no visitado[v] entonces dfs(u)
para i := 1 hasta n hacer si no visitado[i] entonces dfs(i)
```

Podemos ver el órden de visita de unos nodos a otros como un **árbol de expansión en profundidad** asociado al grafo o, si aparecen varios árboles, como un **bosque de expansión en profundidad**. Para construirlo, hacemos que cada elemento de visitado pueda almacenar un valor [1..n] indicando el padre del nodo, lo que nos permite encontrar los componentes conexos de un grafo no dirigido. Decimos que $(a, b) \in E$ es:

- Un **arco de avance** si $a$ es padre a $b$ en uno de los árboles del bosque.
- Un **arco de retroceso** si $a$ es hijo de $b$.
- Un **arco de cruce** si no es de avance ni de retroceso.

Un grafo no dirigido contiene un ciclo si y sólo si algún arco no está en el árbol de expansión. Un grafo dirigido contiene un ciclo si y sólo si algún arco es de retroceso.

**Búsqueda primero en anchura o amplitud**

```
var visitado: array [1..n] de booleano
    C: Cola[[1..n]]
operación bfs(u: [1..n])
    visitado[u] := verdadero; C.meter(u); visitar(u)
    mientras no C.vacía hacer
        v := C.cabeza; C.sacar
        para cada nodo w adyacente a v hacer
            si no visitado[w] entonces
                visitado[w] := verdadero; C.meter(w); visitar(w)
para i := 1 hasta n hacer si no visitado[i] entonces bfs(i)
```

[1] Para booleano es falso; para entero, real y $[0, +\infty]$ es 0; para **array**s y **registro**s se inicializa con el valor por defecto de cada campo; para tipos contenedor se crea una instancia vacía, y para RelEquiv las clases de equivalencia son de un solo elemento. Las variables del resto de tipos no se inicializan por defecto.

## 4.3. Árboles de expansión mínimos

Un **árbol de expansión** de un grafo no dirigido y conexo es un subgrafo acíclico conexo. El **coste** de un grafo con pesos[2] es la suma de los costes de las aristas. Los siguientes algoritmos obtienen coste del árbol de expansión de coste mínimo de un grafo, donde la variable coste contiene dicho coste.

**Algoritmo de Prim**

```
var visitado: array [1..n] de booleano
    C: ColaPrioridad[registro coste: real; nodo: [1..n]] // A menor coste, mayor prioridad
    coste: real
operación procesar(u: [1..n])
    visitado[u] := verdadero
    para cada nodo v adyancente a u con peso w hacer
        si no visitado[v] entonces C.meter((w, v))
procesar(0)
mientras no C.vacía hacer
    p := C.cabeza; C.sacar
    si no visitado(p.nodo) entonces
        coste := coste + p.coste
        procesar(p.nodo)
```

Como introducir y extraer elementos de la cola de prioridad es $O(\log n)$, siendo $n$ el total de elementos de la cola que será proporcional a $|E|$, el algoritmo se ejecuta en $O(|E| \log |E|)$.

**Algoritmo de Kruskal**

```
var RE: RelEquiv[[1..n]]
    coste: real
para cada arista (u, v) con peso w ordenadas por peso ascendente hacer
    si RE.encuentra(u) ≠ RE.encuentra(v) entonces
        coste := coste + w
        RE.unión(u, v)
```

El algoritmo se ejecuta en $O(|E| \log |E|)$, pues es lo que se tarda en ordenar las aristas.

## 4.4. Caminos mínimos

El coste de un camino es la suma de los costes de las aristas por las que pasa, y el **camino mínimo** entre dos nodos es el que tiene menor coste de entre los que empiezan en el primero y terminan en el segundo.

El **algoritmo de Dijkstra** obtiene el mínimo coste del camino de cierto nodo (que supondremos que es el 1) y todos los demás. Para ello trabaja con un conjunto de nodos **escogidos**, para los cuales se conoce ya el camino mínimo desde el origen, y otro de **candidatos**, de los que no sabemos el camino mínimo salvo si nos restringimos a **caminos especiales**, los que

[2] En general, de un grafo etiquetado por elementos de algún grupo.

pasan solo por nodos escogidos (más él mismo al final). En cada paso el algoritmo toma el nodo candidato con menor distancia al origen, lo añade a los candidatos y recalcula los caminos mínimos del resto de candidatos. Tras repetir esto $n-1$ veces, el camino mínimo de cada nodo coincide con su camino especial. La siguiente versión del algoritmo funciona con una matriz de adyacencia C que usa $+\infty$ para indicar la ausencia de arista:

```
var coste: array [2..n] de real ∪ {+∞}
    paso: array [2..n] de [1..n]
    escogido: array [2..n] de booleano
para i := 2 hasta n hacer
    coste[i] := C[1, i]; paso[i] := 1; escogido[v] := falso
para i := 1 hasta n−1 hacer
    u := i si no escogido[i] minimizando coste[i]
    escogido[u] := verdadero
    para cada nodo v adyacente a u hacer
        si no escogido[v] y coste[u] + C[u, v] < coste[v] entonces
            coste[v] := coste[u] + C[u, v]
            paso[v] := u
```

El algoritmo almacena en coste[i] el coste del camino mínimo del nodo 1 al i, y en paso[i] el vértice que precede a i en este camino, con lo que accediendo sucesivamente podemos encontrar el camino mínimo[3]. Si solo necesitamos el coste, podemos omitir del algoritmo la obtención del camino.

Esta implementación ejecuta $n-1$ veces un código que busca un mínimo de entre $n$ valores y actualiza hasta $n$ candidatos, luego en total el algoritmo se ejecuta en $O(|V|^2)$. Otra versión con listas de adyacencia consigue llegar a $O(|E| \log |V|)$.

Para los caminos mínimos entre todos los pares, aplicamos el **algoritmo de Floyd**:

```
var coste: array [1..n, 1..n] de real ∪ {+∞}
    paso: array [1..n, 1..n] de [1..n]
coste := C
para k := 1 hasta n hacer
    para i := 1 hasta n hacer
        para j := 1 hasta n hacer
            si coste[i, k] + coste[k, j] < coste[i, j] entonces
                coste[i, j] := coste[i, k] + coste[k, j]
                paso[i, j] := k
```

Este algoritmo $O(n^3)$ funciona hallando, en cada iteración del bucle externo, los caminos mínimos pasando por un máximo de $k$ vértices. El camino de un vértice a a otro b distintos se obtiene como el camino mínimo de a a paso[a, b] y aquí a b.

El **algoritmo de Warshall** permite obtener el **cierre transitivo** de un grafo, una matriz de booleanos ($a_{ij}$ := existe un camino de $i$ a $j$), y es similar al de Floyd pero cambiando la condición dentro de los bucles por A[i, j] := A[i, j] **o** (A[i, k] **y** A[k, j]).

[3]Los caminos mínimos desde un único nodo al resto forman una estructura de árbol.

## 4.5. Componentes fuertemente conexos

Para hallarlos, usamos el **algoritmo de Tarjan**:

1. Hacer búsqueda en profundidad del grafo numerando los vértices.
2. Construir el grafo invertido: $(V, E) \mapsto (V, \{(b, a)\}_{(a,b)\in E})$.
3. Hacer búsqueda en profundidad del grafo invertido, empezando en el nodo con mayor número en el paso 1. Mientras queden nodos por visitar, seguir con el nodo no visitado de mayor número.
4. Cada árbol del bosque resultante del paso 3 es un componente fuertemente conexo.

```
var componentes: Conjunto[Conjunto[entero]]
    tiempo: entero
    número: array [1..n] de entero
    enlace: array [1..n] de entero
    pila: Pila[[1..n]]
    apilado: array [1..n] de booleano
operación tarjan(u: [1..n])
    número[u] := tiempo; enlace[u] := tiempo
    tiempo := tiempo + 1
    pila.insertar(u); apilado[u] := verdadero
    para cada nodo v adyacente a u hacer
        si número[v] = 0 entonces tarjan(v)
        si apilado[v] entonces enlace[u] = mín(enlace[u], enlace[v])
    si enlace[u] = visitado[u] entonces
        s: Conjunto[entero]
        repetir
            v := pila.tope; pila.sacar; apilado[v] := falso
            s.inserta(v)
        mientras u ≠ v
        componentes.inserta(s)
para i := 1 hasta n hacer si número[i] = 0 entonces conectar(i)
```

Tras ejecutar el algoritmo, componentes es el conjunto de componentes conexos del grafo. Llamamos **grafo reducido** $G_R := (V_R, E_R)$ de $G := (V, E)$ al grafo dirigido acíclico dado por $V_R :=$ {componentes fuertemente conexos de $G$} y $E_R := \{(A, B) \in V_R \mid \exists a \in A, b \in B : (a, b) \in E\}$.

Los grafos dirigidos acíclicos sirven, por ejemplo, para representar órdenes parciales. El **recorrido en orden topológico** de un GDA es aquel en que un vértice solo se visita tras visitar todos los nodos adyacentes a él. La **numeración en orden topológico** consiste en numerar los nodos de un GDA de forma que, si $a$ es adyacente a $b$, el número asignado a $a$ es menor que el asignado a $b$.

```
var orden: Pila[[1..n]]
    visitado: array [1..n] de booleano
operación numerar(u: [1..n])
    visitado[u] := verdadero
    para cada nodo v adyacente a u hacer si no visitado[v] entonces numerar(v)
    orden.insertar(u)
para i := 1 hasta n hacer si no visitado[i] entonces numerar(i)
```

Esto guarda un orden topológico en la pila orden de forma que el elemento en la cima es el primero, el siguiente el segundo, etc.

## 4.6. Grafos eulerianos

Un **punto de articulación** de un grafo no dirigido es un vértice que, al eliminarlo del grafo (junto a las aristas incidentes) se divide un componente conexo en dos o más. Un grafo no dirigido es **biconexo** si no tiene puntos de articulación. Un **componente biconexo** de un grafo es un subgrafo biconexo de este que no es subgrafo de otro subgrafo biconexo.

Un grafo tiene **conectividad** $k \in \mathbb{N}$ si es necesario eliminar $k$ nodos para que el grafo no sea conexo, por lo que un grafo es biconexo si tiene conectividad 2 o más. El siguiente algoritmo busca puntos de articulación:

```
var tiempo: entero
    raíz: entero
    hijosRaíz: entero
    número: array [1..n] de entero
    enlace: array [1..n] de entero
    padre: array [1..n] de entero
    articulación: array [1..n] de booleano
operación puntosArticulación(u: [1..n])
    número[u] := tiempo; enlace[u] := tiempo
    tiempo := tiempo + 1
    para cada nodo v adyacente a u hacer
        si número[v] = 0 entonces
            padre[v] := u
            si u = raíz entonces hijosRaíz := hijosRaíz + 1
            puntosArticulación(v)
            si enlace[v] ≥ número[u] entonces articulación[u] := verdadero
            enlace[u] := mín(enlace[u], enlace[v])
        sino si v ≠ padre(u) entonces
            enlace[u] := mín(enlace[u], número[v])
para i := 1 hasta n hacer
    si número[i] = 0 entonces
        raíz := i; hijosRaíz := 0; puntosArticulación(i)
        articulación[raíz] := hijosRaíz > 1
```

Un **camino de Euler** es aquel que visita todas las aristas exactamente una vez. Si además es un ciclo, se llama **circuito de Euler**. Un grafo no dirigido tiene un circuito de Euler si y sólo si todos los nodos tienen grado par y, tras eliminar los de grado 0, es conexo. El siguiente

algoritmo busca un ciclo de Euler, supuesto que exista, a partir del vértice 1, supuesto que cada arista (u, v) tiene una marca disponible(u, v) que inicialmente está establecida a verdadero. Insertar un elemento en un iterador lo inserta antes del elemento al que apunta y devuelve un iterador al elemento insertado.

```
var ciclo: Lista[entero]
    visitado: array [1..n] de booleano
    v: entero
    j: Iterador[Lista[entero]]
operación circuito(i: Iterador[Lista[entero]], u: entero)
    para cada nodo v adyacente a u hacer
        si disponible(u, v) entonces
            disponible(u, v) := falso; disponible(v, u) := falso
            tour(i.insertar(u), v)
circuito(iterar(ciclo), 1)
```

La idea es partir de un vértice, tomar una arista no visitada, añadirla al ciclo y repetir el proceso partiendo de este nuevo vértice.

## 4.7. Otros problemas

- **Flujo máximo**: Los nodos representan puntos de una red, las aristas son canales de comunicación entre ellos y los pesos de estas son el caudal máximo que puede circular. El problema es hallar el flujo (caudal) máximo que puede haber desde un cierto nodo (origen) a otro (destino), con las restricciones de que la suma del flujo que entra a cada nodo interior (no origen ni destino) debe ser igual a la que sale y el flujo en cada arista no puede superar el máximo.
  Un posible algoritmo es encontrar un camino del origen al destino, sumar al flujo el mínimo caudal de las aristas que lo forman, restar dicho flujo de estas aristas y repetir hasta que no haya caminos con peso no nulo, si bien esto no garantiza solución óptima.

- **Ciclo hamiltoniano** o **de Hamilton**: Es un ciclo simple que visita todos los nodos. Determinar si existe en un grafo es un problema NP-completo, y en la práctica se usan **heurísticas**, soluciones aproximadas que pueden o no funcionar.

- **Problema del viajero** o **del viajante**: Dado un grafo no dirigido, completo y con pesos, encontrar el ciclo de menor coste que pase por todos los nodos. También es NP-completo, y podemos usar heurísticas, técnicas probabilísticas, algoritmos genéticos, etc. para obtener aproximaciones.

- **Coloración de grafos**: Asignar un color o etiqueta a cada nodo de forma que dos nodos unidos por una arista no tengan el mismo color, usando un mínimo número de colores. Es NP-completo.

- **Isomorfismo**: Dos grafos $G := (V, E)$ y $G' := (V', E')$ son **isomorfos** si existe una biyección $\sigma : V \to V'$ tal que $(v, w) \in E \iff (\sigma(v), \sigma(w)) \in E'$. Determinar si existe es NP-completo.