Análisis de algoritmos

La complejidad de un algoritmo = los recursos computacionales que consume (tiempo de ejecución y espacio en memoria)

Al analizar una algoritmo investigamos las propiedades de la complejidad del algoritmo. Para analizar un algoritmo se usan estas 3 **notaciones asintóticas** (permite clasificar las funciones "a la larga", para grandes valores de n), dónde n es el tamaño de la entrada de la función.

- $\Theta(n) \rightarrow$ Límite asintótico exacto $= f$ crece al mismo ritmo que $\Theta(n)$
- $O(n) \rightarrow$ Límite asintótico superior $= f$ no crece más rápido que $O(n)$
- $\Omega(n) \rightarrow$ Límite asintótico inferior $= f$ crece al menos tan rápido como $\Omega(n)$

Costes frecuentes

$\log n$	$<$	$\sqrt{n}$	$<$	n	$<$	$n \log n$	$<$	n^2	$<$	n^3	$<$	k^n	|	$\Theta(1)$
Logarítmico		Lineal		Casi-Lineal		Cuadrático		Cúbico		Exponencial			|	Constante

Propiedades: (f i g son funciones)

- El coste de una **operación elemental** es $\Theta(1)$, esto incluye:
  - Asignación de tipos básicos (int, bool, double)
  - Operaciones aritméticas (suma, resta, multiplicación y división) e incrementos y decrementos de una variable.
  - Lectura o escritura de tipo básico
  - Una comparación
  - Acceso a un componente de un vector
- Suma de costes: $\Theta(f) + \Theta(g) = \Theta(\max\{f, g\})$
- Producto: $\Theta(f) \cdot \Theta(g) = \Theta(f \cdot g)$
- Si una constante $c > 0 \rightarrow O(f) = O(c \cdot f)$
- Paso de parámetro por valor (o return de un vector) $\rightarrow \Theta(n)$ (Porque el parámetro es copiado)
- Paso de parámetro por referencia $\rightarrow \Theta(1)$ (Referenciamos dónde se encuentra la variable)
- Cambio de base un logaritmo: $\log_c x = \frac{\log_b x}{\log_b c}$

$$\lim_{n\to\infty} \frac{g(n)}{f(n)} = 0 \Rightarrow g \in O(f) \wedge g \notin \Omega(f)$$
$$\lim_{n\to\infty} \frac{g(n)}{f(n)} = c \Rightarrow g \in \Theta(f) \Rightarrow g \in O(f) \wedge g \in \Omega(f)$$
$$\lim_{n\to\infty} \frac{g(n)}{f(n)} = \infty \Rightarrow g \notin O(f) \wedge g \in \Omega(f)$$

Análisis de algoritmos iterativos

```
if B then S1
else S2
```
$O(\max\{f + h, g + h\})$

El coste de `B` es h
El coste de `S1` es f
El coste de `S2` es g

```
while B {
   S
}
```
Si $p = \max\{f + h\} \rightarrow T = O(p \cdot g)$

El coste de `B` es f
El coste de `S` es h
g = número de iteraciones

Análisis de algoritmos recursivos

$$T(n) = a \cdot T(n - c) + f(n) \rightarrow T(n) = \begin{cases} \Theta(n^k) & If\ a < 1 \\ \Theta(n^{k+1}) & If\ a = 1 \\ \Theta\left(a^{\frac{n}{c}}\right) & If\ a > 1 \end{cases}$$

Dónde a es el número de llamadas recursivas, $c \geq 1$ es el número que restamos al parámetro que pasamos en la llamada recursiva y $f(n) = \Theta(n^k)$ es el coste de la parte no recursiva del algoritmo.

Ejemplos: Si $f(n) = \Theta(1) \rightarrow n^k = 1 \rightarrow k = 0$, Si $f(n) = \Theta(n) \rightarrow n^k = n \rightarrow k = 1$

$$T(n) = a \cdot T\left(\frac{n}{b}\right) + f(n) \rightarrow T(n) = \begin{cases} \Theta\left(n^{\log_b a}\right) & If\ \ k < \log_b a \\ \Theta\left(n^k \cdot \log n\right) & If\ k = \log_b a \\ \Theta\left(n^k\right) & If\ k > \log_b a \end{cases}$$

Dónde $a \geq 1$ es el número de llamadas recursivas, $b > 1$ es el número que dividimos al parámetro que pasamos en la llamada recursiva y $f(n) = \Theta(n^k)$ es el coste de la parte no recursiva del algoritmo.

También se pueden expresar en una recurrencia:

$$T(n) = \begin{cases} caso\ base & si\ 0 \leq n < n_0 \\ recurrencia\ substractiva\ o\ divisora & si\ n \geq n_0 \end{cases}$$

Dividir y Vencer

Dividir y vencer es una estrategia que resuelve un problema en 3 pasos:

1- Dividiendo el problema principal en subproblemas, casos más pequeños del mismo problema.
2- Resolviendo los subproblemas recursivamente.
3- Combinando las respuestas adecuadamente para resolver el problema principal.

Algoritmos de dividir y vencer

Mergesort (Ordenación por fusión) → Siempre: $\Theta(n \log n)$.

1- Divide la secuencia de entrada en dos mitades.
2- Ordena cada mitad recursivamente
3- Fusiona las dos secuencias ordenadas

```
void merge(vector<elem>& T, int l, int m, int u) {
   vector<elem> B(u - l + 1);
   int i = l, j = m + 1, k = 0;
   while (i <= m and j <= u) {
      // If the element of the lower part of the sequence is smaller than the
      // upper one then add the element of the lower part to the sequence
      if (T[i] <= T[j]) B[k++] = T[i++];
      // Otherwise add the element of the upper part to the sequence
      else B[k++] = T[j++];
   }
   // Add to the new sequence B the elements that hasn't been added yet
   while (i <= m) B[k++] = T[i++];
   while (j <= u) B[k++] = T[j++];
   // Overwrites the part of the sequence that has been processed
   for (k = 0; k <= u - l; ++k) T[l + k] = B[k];
}

// Initial call: mergesort(T, 0, T.size() - 1)
void mergesort(vector<elem>& T, int l, int u) {
   if (l < u) {
      int m = (l + u) / 2;
      mergesort(T, l, m);       // Sort the half downwards positions
      mergesort(T, m + 1, u);   // Sort the half upwards positions
      merge(T, l, m, u);        // Merge the two sorted sequences
   }
}
```

Quicksort (Ordenación rápida) → $\Omega(\mathrm{n}\log n)$, $\mathrm{O}(n^2)$ y de media $\Theta(n\log n)$.

1- Escoge un elemento de la lista que hará de pivote
   a. Si el pivote es el primer elemento de la lista → Si la entrada esta ordenada el algoritmo pierde tiempo $\Theta(\mathrm{n}^2)$
   b. Si pivote es un elemento aleatorio → En media divide el problema en subproblemas parecidos, pero no siempre hace el algoritmo más rápido.
   c. Si el pivote es la mediana de tres elementos (porque hacer la mediana de todo sería muy costoso) → Hace una buena estimación, normalmente se escoge el primer, el último y el elemento del medio.
2- Hacer dos grupos
   a. Los elementos mayores que el pivote van a un lado.
   b. Los elementos menores que el pivote al otro lado, los elementos iguales al pivote pueden ir a cualquiera de los dos lados.
3- Hace dos llamadas recursivas dividiendo la lista en dos (mayores y menores)
4- Repite el proceso por cada sublista.
5- Devuelve las dos sublistas ordenadas seguidas.

```
int partition(vector<elem>& T, int l, int u) {
   int x = T[l]; // In this case we take the first element as pivot
   int i = l - 1;
   int j = u + 1;
   while (1) {
         while (x < T[--j]);
         while (T[++i] < x);
         if (i >= j) return j;
         // If there is an element in the lower and upper part that is not
         // in the correct part, exchange them.
         swap(T[i], T[j]);
   }
}

// Initial call: quicksort(T, 0, T.size() - 1);
void quicksort(vector<elem>& T, int l, int u) {
   if (l < u) {
         int q = partition(T, l, u);
         // Elements smaller than the pivot
         quicksort(T, l, q);
         // Elements bigger than the pivot
         quicksort(T, q + 1, u);
   }
}
```

BinarySearch (Búsqueda Binaria) → Devuelve la posición de un elemento en un conjunto ordenado crecientemente → $\Omega(1)$, $\mathrm{O}(\log n)$ y de media $\Theta(\log n)$.

1- Selecciona el elemento que se encuentra en la posición del medio del conjunto.
2- Mirar si el elemento seleccionado es mayor o menor que el elemento que buscamos:
   a. Si es mayor seleccionamos la parte superior del conjunto.
   b. Si es menor seleccionamos la parte inferior del conjunto.
3- Repite el proceso hasta encontrar el número que buscamos, si el elemento buscado no se encuentra en el conjunto devuelve el elemento más cercano por la parte superior.

```
// Initial call: binarySearch(T, x, 0, T.size() - 1)
int binarySearch(const vector<elem>& T, const int& x, int l, int u) {
   if (l > u) return u;    // The element is not in the array

   int m = (l + u) / 2;
   if (x == T[m]) return m;  // The position of the number has been found
   else if (x < T[m]) return binarySearch(T, x, l, m - 1);  // [----]------
   else return binarySearch(T, x, m + 1, u);                // -----[-----]
}
```

QuickSelect (Selección rápida) → Busca el $j-th$ elemento más pequeño de un conjunto no ordenado dado → $\Omega(n)$, $O(n^2)$ y de media $\Theta(n)$.

1. Escoge un elemento del conjunto que hará de pivote.
2. Todos los elementos más grandes que el pivote se situan a la derecha de este.
3. Escoge otro elemento de la parte izquierda del conjunto como pivote.
4. Repetimos el proceso mientras el pivote $k > j$.

```
// Initial call: quickSelect(T, 0, j, T.size() - 1)
int quickSelect(vector<int> T, int l, int j, int u) {
   if (l == u) return T[l];

   int k = partition(T, l, u);  // Same function as quickSort algorithm
   if (k == j) return T[k];     // The j-th smallest element has been found
   else if (j < k) return quickSelect(T, l, j, k - 1);    // [------]-----
   else return quickSelect(T, k + 1, j, u);               // -------[----]
}
```

InsertionSort (Ordenación por inserción) y **BubbleSort** (Ordenación de burbuja) → $\Omega(n), O(n^2)$ y de media $\Theta(n^2)$.

Algoritmo de Karatsuba → Calcula el producto de dos números naturales de n bits → $\Theta\left(n^{\log_2 3}\right)$.

1- Supongamos que x e y son dos números naturales, los pasamos a binario y dividimos x e y en 2 mitades de n bits cada uno, en el caso de que no haya el mismo número de bits (n) o n no sea una potencia de 2 → añadir 0's a la izquierda:

   a. $x = [x_I][x_D] = 2^{n/2}x_I + x_D$
   b. $y = [y_I][y_D] = 2^{n/2}y_I + y_D$

$x =$ | 01... | 11... | (x_I | x_D) $y =$ | 10... | 10... | (y_I | x_D)

2- Calcular $x \cdot y = 2^n x_I y_I + 2^{n/2}\,(x_I y_D + x_D y_I) + x_D y_D$

Quick exponentiation → $\Theta(\log n)$.

```
double fast_exponentiation(double x, int n) {    // Computes x^n
   if (n == 0) return 1;

   double y = exponential (x, n / 2);
   if (n % 2 == 0) return y * y;
   else return y * y * x;
}
```

Algoritmo de Strassen → Calcula el producto de dos matrices de tamaño $n \times n$ → $\Theta\left(n^{\log_2 7}\right)$.

De esta manera se reducen el número de multiplicaciones necesarias de 8 a 7

$$\begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \cdot \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} = \begin{pmatrix} C_{11} = M_2 + M_3 & C_{12} = M_1 + M_2 + M_5 + M_6 \\ C_{21} = M_1 + M_2 + M_4 - M_7 & C_{22} = M_1 + M_2 + M_4 + M_5 \end{pmatrix}$$

$M_1 = (A_{21} + A_{22} - A_{11}) \cdot (B_{11} + B_{22} - B_{12})$
$M_2 = A_{11} \cdot B_{11}$
$M_3 = A_{12} \cdot B_{21}$
$M_4 = (A_{11} - A_{21}) \cdot (B_{22} - B_{12})$

$M_5 = (A_{21} + A_{22}) \cdot (B_{12} - B_{11})$
$M_6 = (A_{12} - A_{21} + A_{11} - A_{22}) \cdot B_{22}$
$M_7 = A_{22} \cdot (B_{11} + B_{22} - B_{12} - B_{21})$

Diccionarios

Un diccionario es una estructura de datos formada por diferentes llaves, que se usan como identificador para una información o valor. Estas llaves son únicas, no se pueden repetir.

Elemento = (llave, información)

Las operaciones permitidas son:

- Asignar: añadir un elemento (llave, información) al diccionario. Si existía un elemento con la misma llave, se sobrescribe la información.
- Eliminar: dada una llave, suprime el elemento que tiene dicha llave. Si no hay ningún elemento con dicha llave, no hace nada.
- Presente: dada una llave, devuelve un booleano que indica si e diccionario contiene un elemento con la llave dada.
- Búsqueda: dada una llave, devuelve una referencia al elemento con esa llave.
- Consulta: dada una llave, devuelve una referencia a la información de esa llave.
- Tamaño: devuelve el tamaño del diccionario.

A continuación veremos las diferentes estructuras de datos que implementan los diccionarios:

Tabla de acceso directo (un vector): son estructuras que cada posición del vector corresponde a una llave, por lo que gana en eficiencia, ya que las operaciones en el peor caso serán $\Theta(1)$, pero sacrificamos espacio.

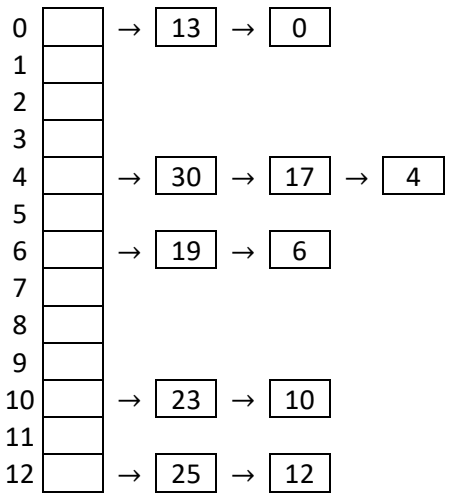
Hash tables (tabla de dispersión): son estructuras eficientes para implementar los diccionarios. → $O(n)$ y de media $\Theta(1)$ (Con hipótesis razonables). Supongamos que tenemos un máximo de n llaves y declaramos una tabla T de m posiciones, dónde $m \leq n$.

- Si $m = n$, usamos acceso directo: el elemento de la llave k va al espacio $T[k]$.
- Si $m < n$, usamos tablas de dispersión: el elemento de la llave k va al espacio $T[h(k)]$, dónde h es una función de dispersión y nos indica entre las m posiciones dónde estará la llave k.

Un método de función de dispersión es $h(k) = k \bmod m$

Llamamos **colisiones** a dos claves que coinciden en un mismo espacio, esto es debido a que hay menos posiciones que claves, por lo tanto es inevitable, aunque lo ideal es que haya cuantas menos colisiones. Hay diferentes métodos para tratar las colisiones:

- **Separate Chaining** (Encadenamiento separado): la solución de las colisiones por encadenamiento se resuelven haciendo que cada entrada de la tabla contenga una lista encadenada con las diferentes llaves de dispersión. De esta manera la función de dispersión h indica dónde debe ir cada llave.



El coste por consulta:

- En el peor de los casos $\Theta(n)$: suponiendo que todos los elementos se encuentren en el mismo valor de dispersión, formando una sola lista.
- De media $\Theta(1)$: porque la función de dispersión tarda $\Theta(1)$ y luego devuelve una llave de la lista.

Assignar y suprimir (con búsqueda previa), en el peor de los casos tiene coste $\Theta(n)$.

El **factor de carga** (load factor): indica la media de elementos por posición $\alpha = n/m$. De esta manera sabemos que el coste medio por búsqueda, asignación o eliminación (ambas con búsqueda previa) es de $\Theta(1+\alpha)$.

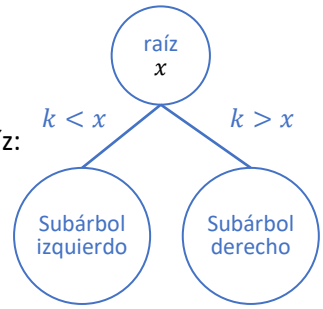
- **Open Adressing** (Direccionamiento abierto): Todos los elementos se almacenan en la tabla de dispersión, por lo que la tabla siempre será mayor o igual que el número de llaves existentes (la tabla puede ir aumentando de tamaño). Hay diferentes maneras de aplicar este método:
  - **Linear probing** (Exploración lineal): inicialmente comprueba la posición correspondiente a la función de dispersión de la llave dada, si no prueba con la siguiente posición, así consecutivamente.

0	0		0	0	occupied		0	0	occupied
1			1	13	occupied		1	13	occupied
2			2		free		2	25	occupied
3			3		free		3		free
4	4		4	4	occupied		4	4	occupied
5			5	17	occupied		5	17	occupied
6	6	→	6	6	occupied	→	6	6	occupied
7			7	19	occupied		7	19	occupied
8			8		free		8	30	occupied
9			9		free		9		free
10	10		10	10	occupied		10	10	occupied
11			11	23	occupied		11	23	occupied
12	12		12	12	occupied		12	12	occupied
$+\{0,4,6,10,12\}$			$+\{13,17,19,23\}$				$+\{25,30\}$		

Si el **factor de carga** es demasiado grande → lanza una excepción o redimensiona (normalmente doblando el tamaño) la tabla de dispersión y hace un **rehash**, que consiste en copiar el contenido de la la tabla inicial pero readaptando las llaves a las nuevas posiciones, es una operación costosa.

Binary Search Tree (BST, Árboles binarios de búsqueda): Son estructuras arborescentes las cuales la llave que contienen los nodos cumplen las siguientes propiedades:

- Los nodos mayores que la raíz van al subárbol derecho.
- Los nodos menores que la raíz van al subárbol izquierdo.



Consulta: supongamos que k es la llave que buscamos y x el la llave de la raíz:

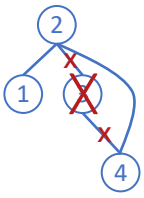
- Si $k = KEY(x)$ → Búsqueda completada.
- Si $k < KEY(x)$ → Hacemos una llamada recursiva al subárbol izquierdo.
- Si $k > KEY(x)$ → Hacemos una llamada recursiva al subárbol derecho.

Inserción:

- Si $k = KEY(x)$ → Sobrescribimos la información.
- Si $k < KEY(x)$ → Hacemos una llamada recursiva al subárbol izquierdo para insertarlo ahí.
- Si $k > KEY(x)$ → Hacemos una llamada recursiva al subárbol derecho para insertarlo ahí.

Eliminación

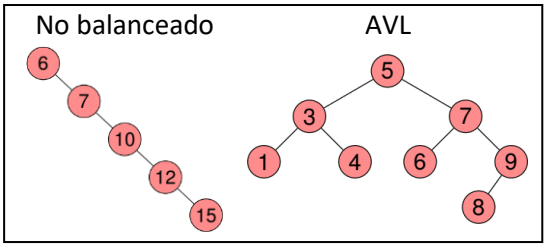
- Si el nodo que queremos eliminar es una hoja (ambos subárboles están vacíos) → Eliminar el nodo.
- Si el nodo que queremos eliminar solo tiene un subárbol → sustituir el árbol existente por el padre.
- Si el nodo tiene los dos subárboles → Buscar el nodo más grande en el subárbol izquierdo y remplazarlo por el nodo que queremos eliminar.



La altitud esperada de un árbol con n nodos es de $\Theta(\log n)$, por lo que los costes de las operaciones anteriores es de media $\Theta(\log n)$, en el peor de los casos $\Theta(n)$, si coincide la altura con el nombre de nodos.

AVLs es un BST balanceado, es decir, que la máxima diferencia que puede haber entre los subárboles izquierdo I y derecho D es 1:

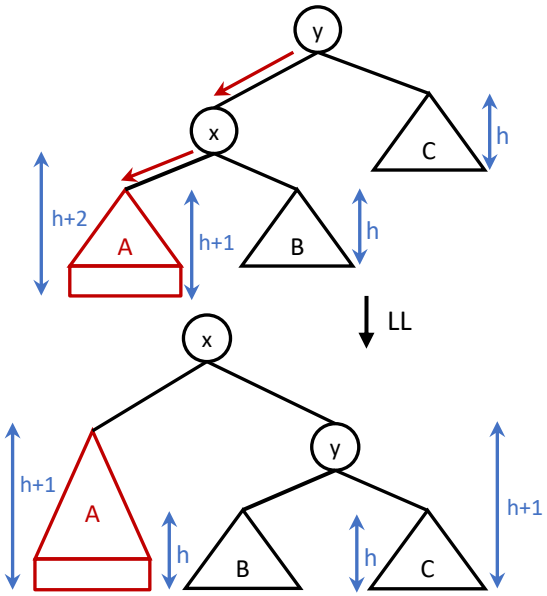
$$|altura(I) - altura(D)| \leq 1$$



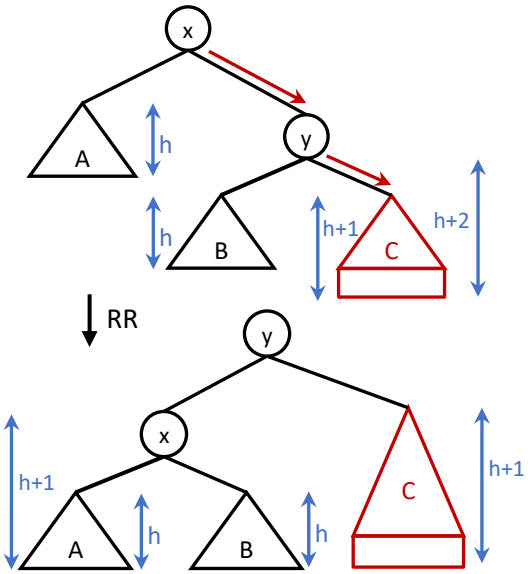
La **consulta** funciona de la misma manera que un BST no balanceado, pero el coste en el peor de los casos es $\Theta(\log n)$, debido a que la altura de un AVL de tamaño n es $\Theta(\log n)$.

Actualizar (insertar y eliminar) un AVL funcionan igual que un BST no balanceado, pero después comprobamos que el árbol se mantenga balanceado (respetando la norma de la altura), para ello utilizamos las **rotaciones**, estas tienen coste $\Theta(1)$, de esta manera el peor de los casos al actualizar un AVL tiene coste $\Theta(\log n)$.

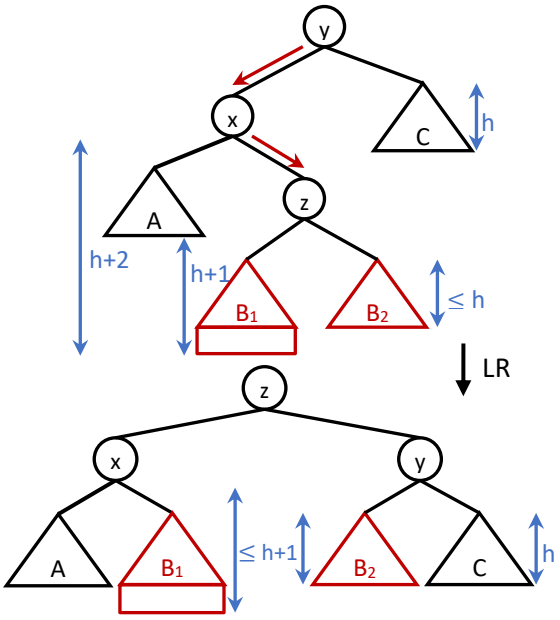
LL (left-left) simple rotation



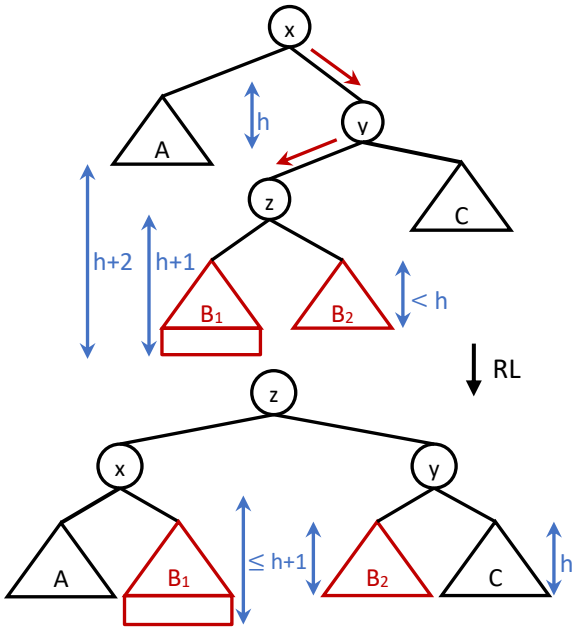
RR (right-right) simple rotation



LR (left-right) double rotation



RL (right-left) double rotation



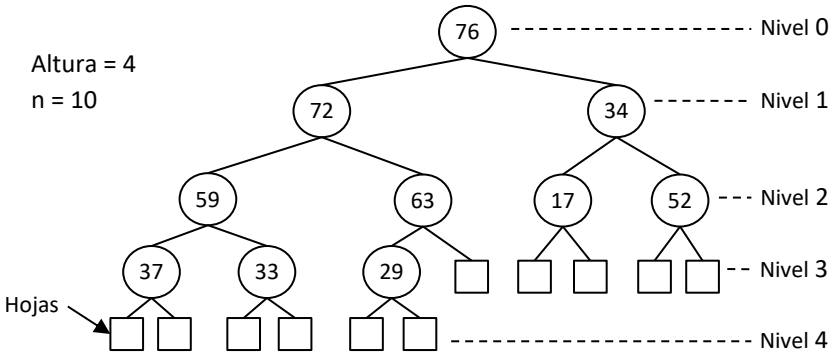
Priority queues (colas con prioridad)

Almacena un conjunto de elementos, cada uno tiene una prioridad asociada a él por el cual se ordena. Las colas de prioridad insertan y suprimen elementos de la mínima (o máxima) prioridad.

Las estructuras de datos de los diccionarios, excepto las tablas de dispersión, pueden usarse para implementar colas de prioridad con coste $\Theta(\log n)$.

Heap → es un árbol binario que:
- Todos los subárboles vacíos se encuentran en los últimos dos niveles del árbol.
- Si un nodo tiene un subárbol izquierdo vacío, el derecho también deberá estar vacío.

Hay dos tipos de heaps:
- Max-heaps → La prioridad de un elemento es mayor o igual que sus descendientes.
- Min-heaps → La prioridad de un elemento es menor o igual que sus descendientes.



Un heap de tamaño n tiene altura $h = \log(n+1)$

Proceso de **eliminar** un nodo $\Theta(\log n)$:

1- Sustituir la raíz o el elemento que se desea eliminar por su último elemento.
2- Eliminar el último elemento del heap
3- Comparar los subárboles izquierdo y derecho para ver si cumplen la condición de ser **menores (max-heap)** o **mayores (min-heap)** e ir intercambiando los nodos hasta que la prioridad del heap se cumpla.

Ejemplo: eliminar la raíz (10)

```
   10          4          5
  /  \        /  \       /  \
 5    3  →   5    3  →  4    3
/ \         /           /
2  4        2           2
```

Proceso de **insertar** un nodo $\Theta(\log n)$:

1- Insertar el **nuevo nodo** como última hoja **de izquierda a derecha**.
2- Comparar el **nuevo nodo** con su **predecesor** (el de arriba).
3- Si el **nuevo nodo** es **mayor (max-heap)** o **menor (min-heap)** que su predecesor los intercambiamos. Repetimos los pasos 2 y 3 hasta que la prioridad se cumpla o llegue a la raíz.

Ejemplo: ejemplo insertar el nodo 15

```
   10          10          10          15
  /  \        /  \        /  \        /  \
 5    3  →   5    3  →   5   15  →   5   10
/ \         / \  /      / \  /      / \  /
2  4        2  4 15     2  4 3     2  4 3
```

Heapsort

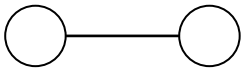
Ordena un conjunto de n elementos construyendo un heap de n elementos, para posteriormente extraerlos uno por uno del heap. El coste es $\Theta(n \log n)$.

Grafos

Conceptos básicos sobre grafos

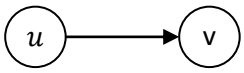
Un **grafo** es un par $G = \langle V, E \rangle$ dónde V es un conjunto finito de vértices (nodos) y E es un conjunto de aristas; cada arista $e \in E$ es un par no ordenado de vértices (u, v) siendo $u \neq v$ y $u, v \in V$.

El número de aristas $|E| = m; 0 \leq m \leq \frac{n(n-1)}{2}$
$\sum_{v \in V} in - \deg(v) = \sum_{v \in V} out - \deg(v) = |E|$



Un **dígrafo** o grafo dirigido es un grafo, pero las aristas tienen una dirección asociada.

El número de aristas $m; 0 \leq m \leq n(n-1)$
$\sum_{v \in V} \deg(v) = 2 \cdot |E|$



Para un arco $e = (u, v)$, al vértice u se le llama **origen** y al vértice v **destino**. Se dice que v es el **sucesor** de u, contrariamente, u es el **predecesor** de v. Los vértices de e se le llaman **extremos** y decimos que u y v son **adyacentes**, también decimos que la arista e es **incidente** a v.

Grado: número de aristas incidentes a un vértice $u \rightarrow \deg(u)$.

- **In-degree**: número de sucesores de un vértice $u \rightarrow in - \deg(u)$.
- **Out-degree**: número de predecesores de un vértice $u \rightarrow out - \deg(u)$.

Un grafo es:

- **Denso**: si el número de aristas se acerca al máximo número de vértices, $m = \Theta(n^2)$, cómo por ejemplo los grafos completos.
- **Disperso**: si es contrario al caso anterior, cómo por ejemplo: los grafos ciclo y los d-regulares.

- Un **camino** si existe mínimo una aresta entre cada vértice y el último, excepto entre el primero y el último, que no es necesario.
- Un **camino simple** es un camino en que ningún vértice se repite, excepto el primero y el último.
- Un **ciclo** si existe un camino en el cual el primer y último vértice son el mismo.
- **Acíclico** si no contiene ningún ciclo.
  - **Hamiltoniano** si contiene al menos un camino que visite todos los vértices del grafo una sola vez.
  - **Euleriano** si contiene un camino que visite todos los vértices del grafo sin repetir arista.
    - **Conexo** si existe un camino para cada par de vértices.

El **Diámetro** de un grado es la distancia máxima entre un par de vértices.

El **centro** de un grafo es un nodo el cual su distancia máxima a cualquier nodo es la distancia mínima.

Un **subgrafo** de G es el mismo grafo quitando vértices o aristas, hay dos tipos:

- **Subgrafo inducido**: contiene exactamente las mismas aristas, pero no todos los vértices.
- **Subgrafo generador**: contiene exactamente los mismos vértices, pero no todas las aristas.

Un **componente conexo** C de grafo G es un grafo inducido máximamente conectado, es decir que es un subgrafo que si añadiéramos otro vértice resultaría en un subgrafo inducido no conexo.

Un **árbol** es un grafo conexo acíclico, dónde $|E| = |V| - 1$.

- Un **árbol generador** es un subgrafo generador y un árbol a la vez.

Un **bosque** es un subgrafo generador acíclico que consiste en un conjunto de árboles, cada uno un árbol generador conexo.

- Un **bosque generador** es un subgrafo generador acíclico que consta de un conjunto de árboles, cada uno un árbol generador conexo.

Se acostumbran a ver dígrafos o grafos en los que los vértices contienen una etiqueta numérica la cual representa el peso de ese vértice. Este tipo de grafos se les llama **grafo etiquetado / ponderado / con peso**, y cada etiqueta del vértice se le llama **peso**.

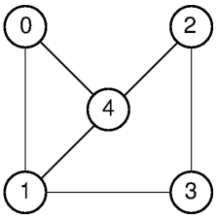
Implementación de los grafos

Matriz de adyacencia: son muy costosas en espacio. Si $|V(G)| = n \rightarrow$ su coste en espacio para representar un grafo es de $\Theta(n^2)$. Su uso es adecuado para **grafos densos**.

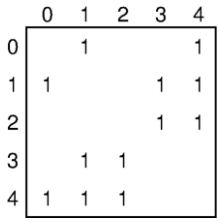
- Cada entrada $M[i][j]$ es un booleano que indica si los nodos i y j están unidos por una arista.
- Para un (di)grafo etiquetado $M[i][j]$ indica el peso asignado a la arista entre los nodos i y j.

Listas de adyacencia: tiene un coste $\Theta(n+m)$ dónde $n = |V|$ y $m = |E|$. Como norma general usar esta implementación.

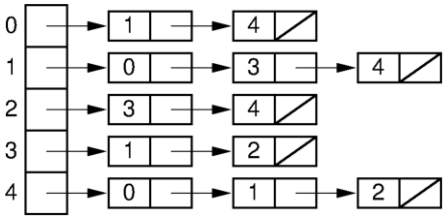
- Se usa un vector T, tal que al buscar un nodo u, $T[u]$ apunte a una lista de aristas incidentes a u o a una lista de vértices $v \in V$ adyacentes a u.
- Para dígrafos, hay una lista de sucesores para cualquier vértice u, y además si se desea, una lista de predecesores de cualquier vértice u.



Grafo



Matriz de adyacencia



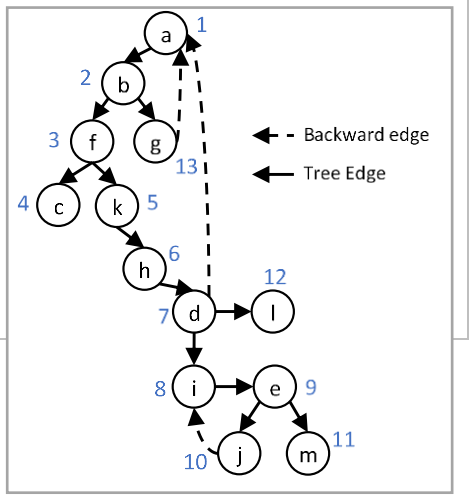
Listas de adyacencias

DFS (Depth-First-Search)

Empezamos visitando un nodo v, a partir de este, vamos recorriendo de manera recursiva todos los nodos no visitados adyacentes/sucesores a v. Un vértice se mantiene **abierto** hasta que se hayan recorrido recursivamente todos los nodos adyacentes/sucesores a v, después el vértice queda cerrado. $\Theta(n+m)$

```
void DFS_REC(const Graph& graph, Visited& vis, int u) {
   vis[u] = true;
   for (int v : graph[u])
         if (not vis[v])
            DFS_REC(graph, vis, v);
}

void DFS(const Graph& graph) {
   Visited vis(graph.size(), false);
   for (int u = 0; u < graph.size(); u++)
      if (not vis[u])
         DFS_REC(graph, vis, u);
}
```



Usos del DFS:

- Detectar ciclos o encontrar Componentes Conexas (CC).
- Saber si un grafo es bipartido.
- Saber si un grafo es N-Colorable.

Un **DAG** (Grafo dirigido acíclico) es un grafo dirigido que no contiene ciclos.

Una **ordenación topológica** de un DAG crea una secuencia de todos los vértices de tal manera que ningún vértice es visitado hasta que todos sus predecesores hayan sido visitados. $\Theta(n^2)$

```
// The graph is an adjacency list
vector<int> topological_sort(const Graph& graph) {
   // Array of predecesors
   vector<int> pred(graph.size(), 0);

   // For all adjacent nodes of u to v add a predecesor to v
   for (int u = 0; u < graph.size(); u++)
      for (int v : graph[u])
         pred[v]++;

   /** Priority queue because we need it to be in lexicographical order **/
   priority_queue<int, vector<int>, greater<int>> q;

   stack<int> s;
   for (int u = 0; u < graph.size(); u++) // Add all nodes without predecesors
      if (pred[u] == 0)
         s.push(u);

   vector<int> nodes;
   while (not s.empty()) {
      int u = s.top();
      s.pop();
      nodes.push_back(u);     // Adds the node to list

   for (int v : graph[u])     // For all Adjacent nodes
      if (--pred[v] == 0)     // If num of predecesors not visited = 0
         s.push(v);           // Adds to stack
   }
   return nodes;
}
```

BFS (Breadth-First Search)

Dado un vértice s visitamos todos los vértices en la componente conexa de s en orden creciente de distancia desde s. Cuando un vértice ha sido visitado, todos los vértices no visitados se añaden a la cola de vértices que posteriormente serán visitados. $\Theta(|V| + |E|)$.

```
// Finds the distance between a given source and destination
int BFS(const graph& G, int source, int destination) {
   // Vector that helps us to know the visited vertices
   vector<int> distance(G.size(), -1); // -1 indicates that hasn't been visited
   queue<int> q;
   q.push(source);         // Initialize the queue
   distance[source] = 0;   // Set the source as visited

   while (not q.empty()) {
      int aux = q.front();
      q.pop();

      // The node destination has been found!
      if (aux == destination) return distance[aux];
```

```
      for (int s : adjacent(aux)) {
         // Check if has been visited
         if (distance[s] == -1) {
            // Mark as visited and set the distance from source
            distance[s] == distance[aux] + 1;
            q.push(s);
         }
      }
   }
   return -1;   // Cannot go to destination from source
}
```

Un algoritmo BFS a partir de un vértice s, calcula:

- El camino más corto entre vértices.
- Diámetro de un grafo.
- El centro de un grafo.

Funciona con grafos y con dígrafos sin pesos.

Se usa como modelo para los siguientes algoritmos:

Algoritmo Dijkstra

Es un algoritmo que encuentra el camino más corto de un vértice a todos los otros vértices de un (di)grafo. Su coste en el peor de los casos es $\Theta\big((m+n)\log n\big)$.

```
int dijkstra_algorithm(const graph& G, int source, int destination) {
   vector<bool> dist(G.size(), false);    // Sets all nodes as not visited
   vector<int> dist(G.size(), INT_MAX);   // Sets distance as ∞ for all nodes
   dist[source] = 0; // Sets the distance from the origin node to 0

   // Queue that stores the elements increasingly by distance <dist, Node>
   priority_queue<pair<int,int>,vector<pair<int,int>>,greater<pair<int,int>>> q;
   q.push({0, source}); // Adds the source node to the queue

   while (not q.empty()) {
      pair<int, int> p = q.top();
      q.pop();

      int u = p.second;
      if (not vis[u]) {
         vis[u] = true;

         // For all adjacent nodes
         for (int s : adjacent(aux)) {
            // If the path is weightless
            if (dist[s] > dist[u] + weight(u_to_v)) {
               dist[s] = dist[u] + weight(u_to_v);
               q.push({dist[s], s});
            }

            /*** If not only exists a unique path ***/
            else if (dist[v] == dist[u] + weight(u_to_v))
               // do something
         }
      }
   }
   return dist[destination]; // Returns the distance of shortest path
}
```

Árboles de Expansión Mínimos: Algoritmo de Prim

Un árbol de expansión mínimo de G es un subgrafo $T = (V_A, A)$ de G que sigue siendo un árbol (conexo y acíclico), contiene todos los vértices de G (es decir, $V_A = V$) y su peso total es el mínimo entre todos los árbol de expansión de G.

Procedimiento: Empieza con un vértice dado, lo visita y busca la arista menos pesada incidente a los vértices que no han sido visitado, sigue este procedimiento hasta que todos los vértices del grafo original estén en el subgrafo. $\Theta(m \log n)$.

```
// pair<weight, node>
typedef pair<int, int> P;

int mst(const vector<vector<P>>& G) {
   vector<bool> vis(G.size(), false);
   // queue that order the inputs by wieght (first element weightless)
   priority_queue<P, vector<P>, greater<P>> q;

   // Push to queue all adjacent vertices of first node
   for (auto p : G[0]) q.push(p);
   vis[0] = true;    // Sets first node as visited

   int sum = 0;      // Sum of weights
   int count = 1;    // Count to visit all nodes
   while (count < G.size()) {
      P p = q.top();
      q.pop();

      int w = p.first;     // Get weight
      int v = p.second;    // Get node
      if (not vis[v]) {
         vis[v] = true;
         for (auto p : G[v]) q.push(p);
         sum += w;
         count++;
      }
   }
   return sum;
}
```

Búsqueda exhaustiva

Los **algoritmos de fuerza bruta** (**o búsqueda exhaustiva**) prueban todas las combinaciones posibles.

Backtracking (Búsqueda con retroceso o vuelta atrás)

Este tipo de algoritmos van construyendo soluciones parciales y su ventaja principal es que si la solución parcial deja de ser una solución viable no continua con esa solución parcial.

1- Explora la solución parcial haciendo una decisión adicional, avanzando un paso para la solución final, si la nueva solución parcial no incumple ninguna restricción, seguimos con dicha solución.
2- Si la solución parcial no satisface alguna restricción, entonces deshacemos la última decisión hecha (backtracking).

Branch & Bound (Ramificación y poda)

Es una versión mejorada de la variante del Backtracking, se usa principalmente para resolver problemas de optimización. Esta variante se suele interpretar como un árbol de soluciones donde cada rama nos lleva a una posible solución posterior a la actual y su principal diferencia a la variante anterior es que detecta cuando una ramificación deja de ser optima para "podar" esa rama y no seguir malgastando recursos con esa solución.

Esta versión es más costosa en memoria que el Backtracking.

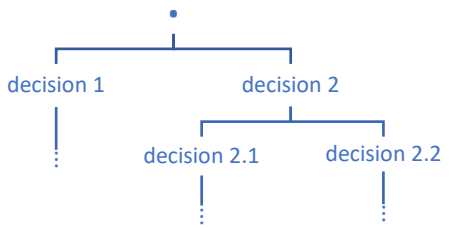
```
procedure BRANCH-AND-BOUND(z)
   Alive: a priority queue of nodes
   y := ROOT_NODE(z)
   Alive.INSERT(y, ESTIMATED-COST(y))
   Initialize best_solution and best_cost, e.g., best_cost := +1
   while ¬Alive:IS_EMPTY() do
      y := Alive.MIN_ELEM(); Alive.REMOVE_MIN()
      for y' ∈ SUCCESSORS(y, z) do
         if IS_SOLUTION(y', z) then
            if COST(y') < best_cost then

               best_cost := COST(y')

               best_solution := y'

               Purge nodes with larger priority = cost

            end if
         else
            feasible := IS_FEASIBLE(y', z)^
               ESTIMATED-COST(y') < best_cost
            if feasible then
               Alive.INSERT(y', ESTIMATED-COST(y'))
            end if
         end if
      end for
   end while
end procedure
```



Complejidad

La teoría de complejidad considera los algoritmos posibles que resuelven un mismo problema y clasifica estos problemas según su complejidad. Mientras el análisis de algoritmos se centra en los algoritmos, la teoría de complejidad se centra en los problemas.

Un **problema decisional** es un problema en el que la salida es **sí** o **no** (equivalentemente, decidir si una entrada satisface una cierta propiedad). Ejemplos: decidir si un grafo es conexo, 3-colorable.

Si *T* es una propiedad que los elementos de un conjunto de entradas *E* pueden tener o no, el problema decisional que se presenta es el siguiente:

Problema A

Dado $x \in E$, determinar si se cumple la $T(x)$ — Manera informal de representarlo

$A = \{x \in E \mid T(x)\}$ — Manera formal de representarlo

- Las entradas positivas, pertenecen a A.
- Las entradas negativas, pertenecen a $E - A$.

Decimos que un algoritmo A tiene coste t si su coste en el peor de los casos pertenece a $\mathrm{O}(t)$.

Si el algoritmo A recibe entradas de un conjunto E y tienen una salida binaria, escribiremos:

$$A : E \to \{0,1\}$$

Decimos que un problema decisional A es decidible en tiempo t si existe un algoritmo de coste t que lo decida (lo resuelva), es decir, si existe $A : E \to \{0,1\}$ de coste t tal que, por todo $x \in E$:

$$x \in A \Rightarrow A(x) = 1$$

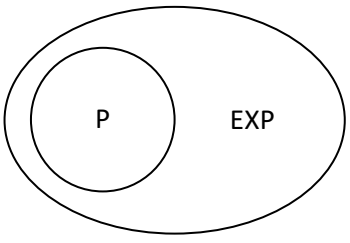
$$x \notin A \Rightarrow A(x) = 0$$

Agrupamos los problemas decidibles en tiempo t: $TIME(t) = \{A \mid A\ es\ decidible\ en\ tiempo\ t\}$.

Dividiremos los grupos en 2 genéricos, esto es debido a la gran diferencia en coste de estos:

- $\boldsymbol{P}$ (Polinómicos), si el problema es decidible en tiempo n^k por alguna k.
  - Estos problemas los consideramos **tratables**.
- $\boldsymbol{EXP}$ (Exponenciales), si el problema es decidible en tiempo 2^{n^k} por alguna k.
  - Estos problemas los consideraremos **intratables**.

$P \subset EXP$ significa:



- Conectividad ∈ P
- Accesibilidad ∈ P
- Primalidad ∈ P
- Camino corto ∈ P
- 2-Colorabilidad ∈ P
- 3-Colorabilidad ∈ EXP (no se sabe si P)
- Viajante ∈ EXP (no se sabe si P)
- Problema de la detención ∈ EXP (no se sabe si P)

Indeterminismo

Los algoritmos vistos por ahora son **deterministas**, es decir, que siguen un único camino de cálculo des de la entrada hasta el resultado.

Un algoritmo indeterminista para una misma entrada no da siempre la misma solución. Ya que el algoritmo utiliza una función ESCOGER(X) que devuelve un valor y entre 0 y x, entonces:

- A empieza el cálculo de manera determinista hasta la primera instrucción.
- Para cada valor devuelto de ESCOGER, el cálculo se divide en diferentes ramas con el valor correspondiente.
- Decimos que A devuelve 1 (que hay solución), si lo hace alguna de las ramas del árbol de cálculo.
  - La solución que nos devuelve cierto, se llama testimonio.
    - Este no puede ser mayor que la entrada $|y| \leq |x|$
    - El algoritmo que puede verificar que es correcta la solución tiene tiempo polinómico.

Decidibilidad en tiempo polinómico indeterminista

- Indicamos los **testimonios** (todas las soluciones)
- Damos un polinomio $p(n)$, respecto a la entrada.
- Encontramos un **verificador** (algoritmo que verifique el testimonio en tiempo polinómico)

Ejemplos:

Compuestos

$$COMPUESTOS = \{x \mid \exists y \ \ 1 < y < x \ i \ y \ divide \ x\}$$

- Los testimonios son todos los valores $y \neq 1$ e $y \neq x$ dónde y divide x.
- El polinomio es $p(n)$
- El verificador es:

```
V(x, y)
   si 1 < y < x e y divide x entones
      retornar 1
   sino
      retornar 0
```

En el caso de la 3-Colorabilidad

$$3 - COLOR = \{G \mid G \ es \ 3\text{-}colorable\}$$

- Los testimonios por $G = (V, E)$ son todas las 3-coloraciones C de G de la forma $C = (c_1, \ c_2, \dots, \ c_n)$, dónde $n = |V|$ y $c_i \in \{0, 1, 2\}$ por todo $i \leq n$.
- El **polinomio** es $p(n) = n$.
- El **verificador** es:

```
V(G, C)
   n <- |V|
   si C es una 3-coloración de G entones
      retornar 1
   sino
      retornar 0
```

Definimos la clase NP (nondeterministic polynomial time) com:

$$NP = \{A \mid A \ es \ decidible \ en \ tiempo \ polinómico \ indeterminista\}$$

$P \subseteq NP \subseteq EXP$

Diferencias entre P y NP:

- Los testimonios de P se pueden **encontrar** en tiempo polinómico.
- Los testimonios de NP se pueden **verificar** en tiempo polinómico.

Demostrar que $A \in P \Rightarrow A \in NP$

- Los testimonios $E' = \{algo\}$ que no nos interesa
- El verificador $V(x, y) \coloneqq A(x)$, es decir, que el verificador usará el algoritmo que ya tenemos en $A \in P$, ignorando la función y = ESCOGER(x) de NP, por eso E' no nos importa ya que $V: E \times \cancel{E'} \rightarrow \{0, 1\}$ no tendrá en cuenta los testimonios.
  - Si $x \in A \rightarrow V(x, y) = A(x) = 1$
  - Si $x \notin A \rightarrow V(x, y) = A(x) = 0$

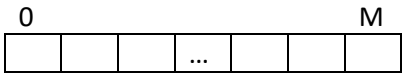
Diferencias entre NP y EXP

- Los problemas de NP tienen testimonios verificables en tiempo polinómico.
- Los problemas EXP pueden tener testimonios exponencialmente largos.

Demostrar que $A \in NP \Rightarrow A \in EXP$

Se que tengo un conjunto p' de candidatos a testimonios

Si sabemos que un testimonio tiene m bits habrá 2^m testimonios, por lo que sabemos que hay un número finito de testimonios y que es exponencialmente mayor que la entrada.



Por lo que el algoritmo quedaría de la siguiente manera:

```
entrada x
por todo candidato de testimonio y
   si V(x, y) = 1 entones
      retornar 1
retornar 0
```

Podemos ver que es un algoritmo se ejecuta en tiempo exponencial, debido a que el bucle se puede ejecutar tantas veces como testimonios haya que es exponencial. Y sabemos que cada $V(x, y)$ tarda tiempo polinómico. Por lo que todo junto tarda tiempo exponencial.

Reducciones

Queremos demostrar que:

$\left.\begin{matrix} A \leq^p B \\ B \in P \end{matrix}\right\} \Rightarrow A \in P$ Sabemos que A y B son problemas pero no sabemos que resuelven.

Construimos un algoritmo para A

Dada una entrada x

1) Calculamos $F(x)$
2) Aplicamos el algoritmo polinómico para B con la entrada $F(x)$

Comprobamos que el algoritmo es correcto: (Propiedad 1 de la reducción)

- $x \in A \Rightarrow F(x) \in B \Rightarrow Si$ Si x es una entrada correcta en el problema A, lo será en B.
- $x \notin A \Rightarrow F(x) \notin B \Rightarrow No$ Si x es una entrada incorrecta en el problema A, lo será en B.

Calculamos su coste: Asumimos que $|x| = n$ (el tamaño de x es n)

1) $p(n)$ Tarda tiempo polinómico respecto x (Propiedad 2 de la reducción)
2) $q(r(n))$ Lo que calcula tiene tiempo polinómico (Propiedad 3 de la reducción)

Por lo que el coste del algoritmo es polinómico porque es la suma de los polinomios.

$A \rightarrow B$ o $A \leq^p B$ Para pasar del problema A al problema B, hay que transformar la entrada del problema A en una entrada compatible para el problema B, demostrar que si la entrada de A es cierta, la entrada de B también y viceversa.

NP-Completo

Un problema A es **NP-difícil** si por todo problema $B \in NP$ tenemos que $B \leq^p A$, es decir, si todo problema B que pertenece a NP, puede reducirse al problema A (es igual o más difícil que NP).

Para demostrar que un problema es NP-difícil: NP-difícil {SAT, CNF-SAT, ciclo, SI}

Demostramos que $NP - dificil \leq^p B$, es decir que NP-difícil puede reducirse a B.

Un problema A es **NP-completo** si es NP-difícil y $A \in NP$ (A es NP-difícil y es NP a la vez).

Se sabe que los problemas NP-completos hay dos:

- $SAT = \{F \mid F\ es\ una\ fórmula\ booleana\ satisfactible\}$
- $CNF\text{-}SAT = \{F \mid F\ es\ una\ fórmula\ booleana\ en\ CNF\ satisfactible\}$ (conjunto de cláusulas)
  - Un literal es una variable afirmada o negada $(x, \neg x)$
  - Una cláusula es una disyunción de literales $(x \vee \neg y \vee z)$ (esto o esto o esto)
  - Es una fórmula conjuntiva porque $(x \vee \neg y \vee z) \wedge (p \vee r) \wedge \ldots$ (esto y esto y…)

Proposición: Sea A un problema NP-completo y B un problema tal que $B \in NP$ y $A \leq^p B$ entonces, B también es NP-completo.