

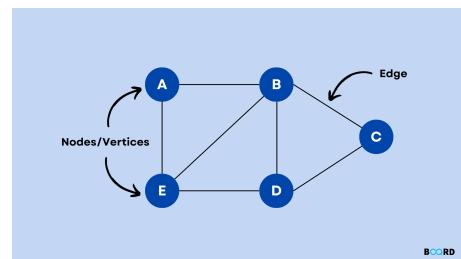
UNIDAD 1 - GRAFOS

GRAFOS

Un grafo es una **estructura de datos compuesta por 2 objetos**, **V un conjunto de vértices (nodos)** y **E un conjunto de aristas** que **conectan pares de vértices**. Se usa para modelar relaciones entre objetos.

💡 Imagina un mapa de ciudades (vértices) conectadas por carreteras (aristas).

Cada ciudad es un nodo, y cada carretera es una conexión entre ellas.



* Definimos un grafo como $G = (V, E)$

vertices ↴

aristas con $E \subseteq V \times V$

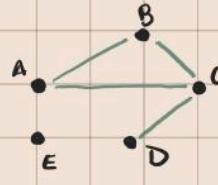
(v, v)

tupla de vértices

que se conectan

- $|V| = n$ cantidad de vértices.
- $|E| = m$ cantidad de aristas.

Ejemplo :



$$V = \{A, B, C, D, E\} \quad |V| = 5$$

$$E = \{(A, B), (A, C), (B, C), (B, D), (C, D)\} \quad |E| = 4$$

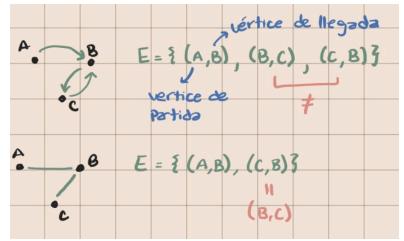
- Un nodo es **adyacente** a otro si están conectados.
- Una **arista** es **incidente** a un nodo si conecta dicho nodo con algún otro.

TIPOS DE GRAFOS

Dirigidos (Dagrafo) y No Dirigidos:

Dirigido o Dagrafo → Las aristas tienen dirección, por ende la arista $(A,B) \neq (B,A)$.

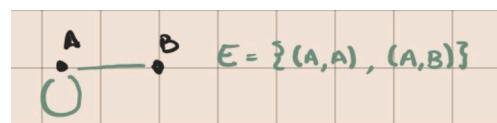
No dirigido → Las aristas no tienen dirección, por ende las aristas $(A,B) = (B,A)$.



💡 El grafo subyacente de un dígrafo G es el grafo G^s que resulta de remover las direcciones de sus vértices (si hay aristas en ambas direcciones, solo se coloca una arista entre ellos)

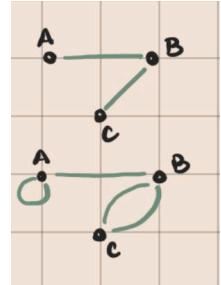
Pseudografo (Self-Loop)

Grafo que permite self-loops es una arista que conecta un vértice consigo mismo. Es decir, una arista de la forma (A,A) .



Simple y No Simples:

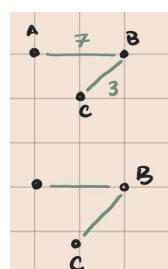
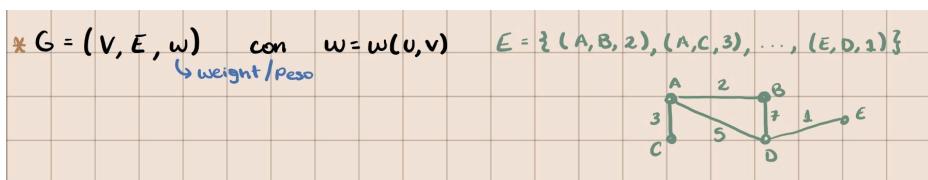
Grafo Simple → Un grafo simple es aquel que no tiene self-loops y no tiene aristas múltiples entre el mismo par de vértices (no tiene aristas repetidas).



Grafo no Simple → Un grafo no simple es aquel que, puede tener self-loops y puede tener múltiples aristas entre el mismo par de vértices (aristas paralelas).

Grafo Pesado (Ponderados) y No Ponderados:

Grafo Pesado o Ponderado → Cada arista tiene un peso (ejemplo: distancia entre ciudades).



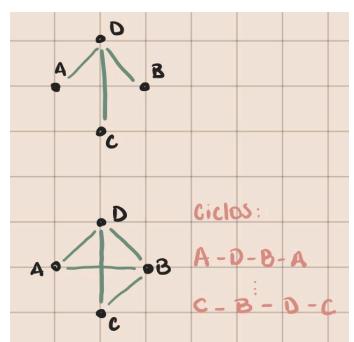
💡 $w(e)$ es una “función de pesos” que asigna a cada arista un peso.

No ponderado → Las aristas no tienen pesos.

Cíclicos y Acíclicos:

Cíclico → Contiene al menos un ciclo, es decir, un ciclo es un camino en un grafo que comienza y termina en el mismo vértice.

Acíclico → No contiene ciclos.



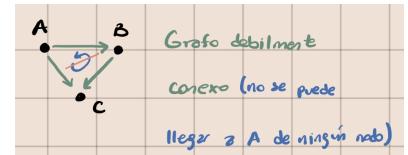
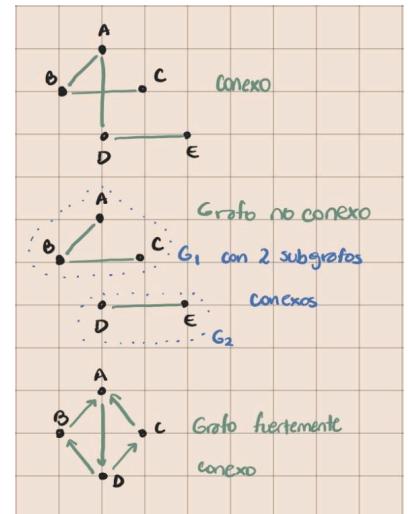
Conexo, No Conexo y Fuertemente Conexo:

Conexo → Un grafo no dirigido es conexo si existe un camino entre cualquier par de vértices.

No Conexo → Un grafo no dirigido es no conexo si existe al menos un par de vértices que no están conectados por ningún camino.

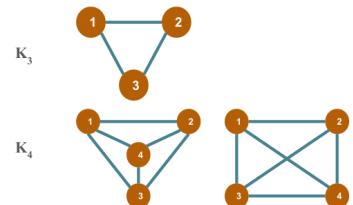
Grafo fuertemente conexo → Un grafo dirigido es fuertemente conexo si existe un camino dirigido desde cualquier vértice a cualquier otro vértice.

Grafo débilmente conexo → Un grafo dirigido es débilmente conexo si, al ignorar la dirección de las aristas, el grafo subyacente (**no dirigido**) es conexo, pero existe un nodo que no se puede alcanzar.



Grafo completo:

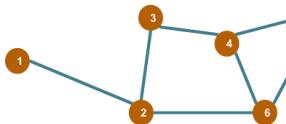
Todos sus vértices son adyacentes entre sí, o **tiene todas las aristas posibles y se nota K_n** (todos los vértices son adyacentes entre sí).



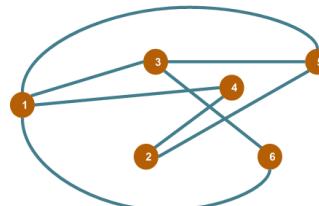
Grafo complemento (G^c):

Tiene el **mismo conjunto de vértices** que el grafo original G , y además **contiene todas las aristas que no están en G** .

$$G = (V, E)$$



$$\bar{G} = (V, \bar{E}) = G^c$$

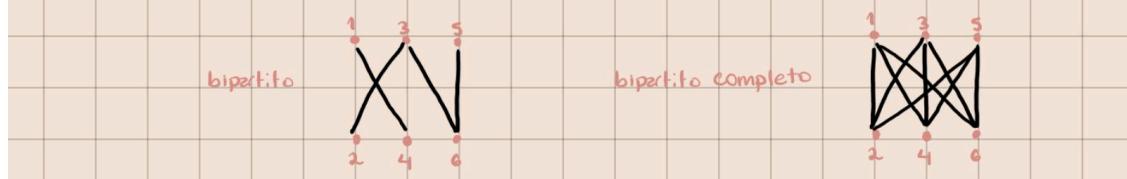


$$m_{\bar{G}} = \frac{n(n-1)}{2} - m$$

Grafo bipartitos:

Un grafo $G = (V, E)$ se dice **bipartito** si existen dos subconjuntos V_1, V_2 del conjunto de vértices V tal que: $V = V_1 \cup V_2$, $V_1 \cap V_2 = \emptyset$ → Partimos los vértices en 2 subconjuntos y tal que todas las aristas de G tienen un extremo en V_1 y otro en V_2 . (w, u) con $w \in V_1$ y $u \in V_2$.

Un grafo bipartito con subconjuntos V_1 y V_2 es **bipartito completo** si todo vértice en V_1 es adyacente a todo vértice en V_2 .



- Un grafo G es bipartito \Leftrightarrow no tiene ciclos de longitud impar

Grafo Isomorfo:

Isomorfismo de Grafos una generalización para decir que 2 grafos son iguales

Dados 2 grafos $G = (V, E)$ y $G' = (V', E')$ se dicen **isomorfos** si $\exists f: V \rightarrow V'$ biyectiva tal que $\forall v, w \in V: (v, w) \in E \Leftrightarrow (f(v), f(w)) \in E'$ (se respetan las adyacencias)

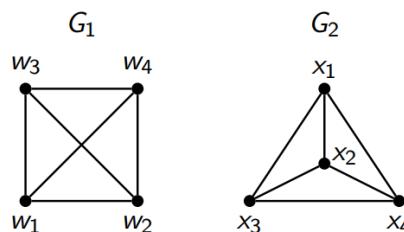
↑ se respetan las aristas en la imagen
→ renombra los vértices

Llamamos a f la **función de isomorfismo**

Cuando G y G' son isomorfos lo notaremos como $G \cong G'$ o $G = G'$

💡 Mismo grafo con renombramiento de vértices.

Dados $G_1 = (V_1, X_1)$ y $G_2 = (V_2, X_2)$:



Si definimos $f: V_1 \rightarrow V_2$ como

$$f(w_1) = x_1 \quad f(w_2) = x_2 \quad f(w_3) = x_3 \quad f(w_4) = x_4$$

podemos ver que f respeta las adyacencias y, por lo tanto, G_1 y G_2 son isomorfos.

Proposición:

Si dos grafos $G = (V, X)$ y $G' = (V', X')$ son isomorfos, entonces

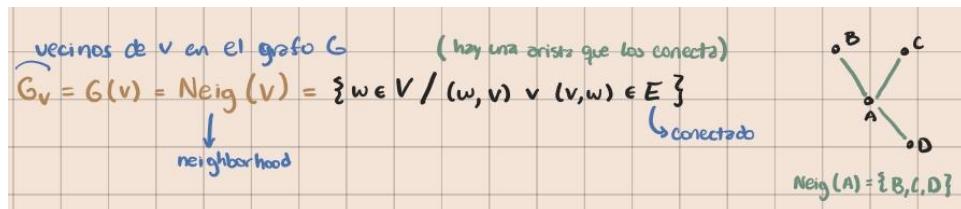
- ▶ tienen el mismo número de vértices,
- ▶ tienen el mismo número de aristas,
- ▶ para todo k , $0 \leq k \leq n - 1$, tienen el mismo número de vértices de grado k ,
- ▶ tienen el mismo número de componentes conexas,
- ▶ para todo k , $1 \leq k \leq n - 1$, tienen el mismo número de caminos de longitud k .

VECINOS DE UN GRAFO

En un grafo, los vecinos de un vértice v son todos los vértices que están directamente conectados a v por una arista.

Nodos Adyacentes → Los nodos adyacentes son aquellos que están conectados entre sí por una arista. También se les conoce como nodos vecinos.

- En un **grafo no dirigido**, si hay una arista entre v y u , entonces u es vecino de v , y viceversa.



- En un **grafo dirigido**, los vecinos se dividen en dos tipos:
 - **Vecinos de salida (out-neighbors):** Vértices a los que v apunta (desde v sale una arista).
 - **Vecinos de entrada (in-neighbors):** Vértices que apuntan a v (hacia v llega una arista).

4. Propiedades de los vecinos

1. En grafos no dirigidos:

- El número de vecinos de un vértice es igual a su **grado**.
- Ejemplo: Si un vértice tiene 3 vecinos, su grado es 3.

2. En grafos dirigidos:

- El número de vecinos de salida es igual al **outdegree**.
- El número de vecinos de entrada es igual al **indegree**.

3. Vértices aislados:

- Un vértice sin vecinos se llama **vértice aislado**.

GRADOS DE UN GRAFO

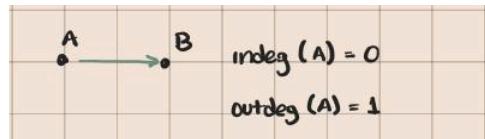
En un grafo no dirigido, el **grado de un vértice** ($\deg(v)$) es el número de aristas incidentes a él (cantidad de conexiones que tiene un nodo).

$$\deg(v) = \# \text{ de aristas conectados a } v.$$

$\# \{e \in E / v \in e\}$ tuplas donde está el v

aristas
 $(v, w), (u, v)$
 e_0, e_1

- **Indeg(v)** → En un **grafo dirigido**, el indegree de un vértice es el **número de aristas que apuntan hacia ese nodo**.
- **Outdeg(v)** → En un **grafo dirigido**, el outdegree de un vértice es el **número de aristas que salen de él**.



💡 $\delta(G)$ es el grado mínimo en G y $\Delta(G)$ es el grado máximo en G .

OBS: • $\deg(v) = \text{Indeg}(v) + \text{Outdeg}(v)$

• Si G es simple no dirigido $0 \leq |E| \leq \frac{n(n-1)}{2} = \binom{n}{2}$

Grafo completo
 (existen todas las aristas)

• $\sum_{v \in V} \deg(v) = \sum_{i=1}^n \deg(v_i) = 2|E|$

Ej:

 $\deg(A) + \deg(B) + \deg(C) + \deg(D) + \deg(E) = 10 = 2.5$

* Por inducción (sobre aristas)

- CASO BASE:

El caso base de nuestra demo. es cuando $m = 1$ (también podría haber sido $m=0$). En este caso el grafo G solo tiene 1 arista $E = \{(u, v)\} \rightarrow d(u) = d(v) = 1$ y $d(w) = 0 \quad \forall w \in V, w \neq u, v$

Por lo tanto $\sum_{v \in V} d(v) = 2$ y $2m = 2$ cumpliendo la prop.

$$G = (V, E)$$

$d(u) = 1$
 $d(v) = 1$
 $d(w_i) = 0$

$\Rightarrow 2|E| = \sum_{v \in V} d(v) ?$
 $2 \cdot 1 = 1 + 1 + 0 + 0 + 0 + 0$
 $2 = 2 \checkmark$

Se cumple el Caso base

- PASO INDUCTIVO:

Para demostrar el paso inductivo, consideremos un grafo G con m aristas, $m \geq 1$.

- Hipótesis inducida: En todo grafo $G' = (V', E')$ con $m' < m$ aristas, se cumple que

$$\sum_{v \in V'} d_{G'}(v) = 2m'$$

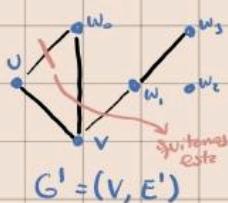
Elijamos una arista cualquiera de nuestro grafo G , y llamemos G' al grafo que resulta si se la quitamos, esto es $G' = (V, E')$ con $E' = E \setminus \{e\}$

$$(u, w_i)$$

$< m$

Como la cant. de aristas de G' es $m-1 \Rightarrow G'$ cumple con la HI, por lo que podemos aplicar la HI sobre G' :

$$G = (V, E)$$



$$G' = (V, E')$$

Vale
HI
 $\sum_{v \in V} d_G(v) = 2(m-1)$

$$4+3+1+2+0+1 = 2 \cdot (5-1)$$

$B = B \checkmark$

Como $d_G(u) = d_{G'}(u) + 1$, $d_G(w_i) = d_{G'}(w_i) + 1$

y $d_G(v) = d_{G'}(v) \quad \forall v \in V \quad v \neq u, w_i$



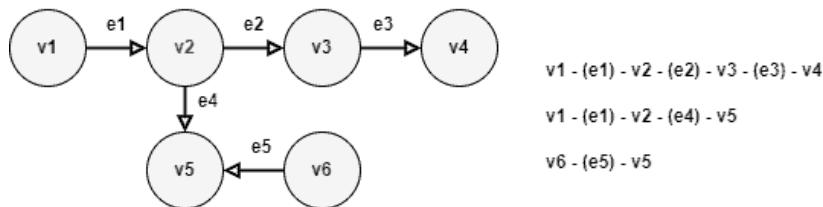
Usando esto obtenemos que:

$$\sum_{v \in V} d_G(v) = \sum_{v \in V} d_{G'}(v) + 2 = 2(m-1) + 2 = 2m$$

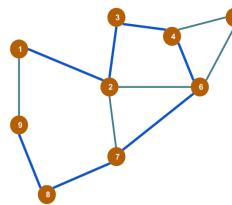
que es lo que queríamos probar

RECORRIDOS Y DISTANCIAS DE UN GRAFO

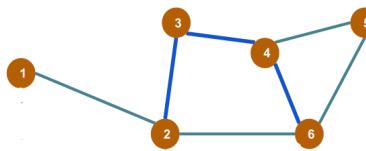
Recorrido → Un recorrido, lo **notamos P**, es una **sucesión de vértices y aristas de un grafo**, tal que e_i sea incidente a v_{i-1} y v_i para todo $i = 1, \dots, k$. ($P = v_0, e_1, v_1, e_2, \dots, e_k, v_k$)



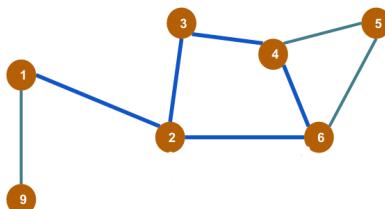
Camino → Un camino es una secuencia de vértices v_1, v_2, \dots, v_k (**o sucesión de aristas**) tal que cada par de vértices consecutivos (v_i, v_{i+1}) está conectado por una arista. **Un camino es un recorrido que no pasa dos veces por el mismo vértice.**



Sección → Una sección es un **tramo del recorrido P**, se nota $P_{vi, vj}$ ($P_{2,6} = 2 \rightarrow 3 \rightarrow 4 \rightarrow 6$)

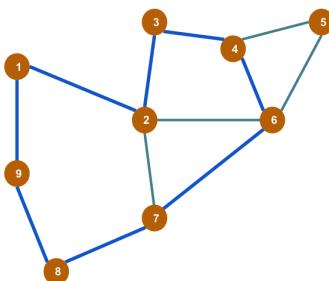


Círculo → Un círculo es un **recorrido que empieza y termina en el mismo vértice**. ($P_{1,1} = 1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 6 \rightarrow 2 \rightarrow 1$)



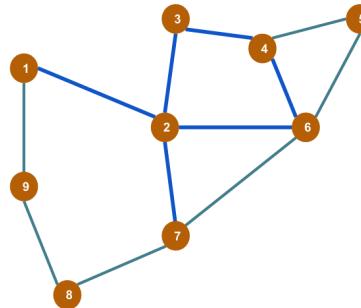
Ciclo o Circuito Simple → Un ciclo o circuito simple es un **círculo** (**de tres o más vértices**) **que no repite vértices** (excepto el vértice inicial = final).

$$P_{1,1} = 1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 6 \rightarrow 7 \rightarrow 8 \rightarrow 9 \rightarrow 1$$



Para grafos no consideramos como válido el ciclo de longitud 2 ($1 \rightarrow 2 \rightarrow 1$)

Longitud → Dado un recorrido P , su longitud, $l(P)$ es la cantidad de aristas que tiene un recorrido, ($P = 1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 6 \rightarrow 2 \rightarrow 7$ y $l(P) = 6$)



Distancia → La distancia entre dos vértices u y v se define como la **longitud del recorrido (camino) más corto entre u y v** , y lo notamos $d(u,v)$. ($P = 1 \rightarrow 2 \rightarrow 7$ y $d(1,7) = 2$)

💡 Si no existe recorrido entre u y v se define la distancia como infinito, $d(u,v) = \infty$.

💡 La distancia de vértice consigo mismo es 0, $d(u,u) = 0$

La función de distancia cumple las siguientes propiedades para todo u, v, w pertenecientes a V :

- ▶ $d(u, v) \geq 0$ y $d(u, v) = 0$ si y sólo si $u = v$.
- ▶ $d(u, v) = d(v, u)$.
- ▶ $d(u, w) \leq d(u, v) + d(v, w)$.

Proposiciones:

Proposición 1: Si un recorrido P entre u y v tiene longitud $d(u,v)$ $\Rightarrow P \Rightarrow Q$
entonces P es un camino.

Demostración (absurdo):

Supongamos que P no es un camino (hipótesis absurdo),

es decir que existe un vértice z que se repite en $P \Rightarrow$

ciclo o loop

$P = u \rightarrow \dots \rightarrow z \rightarrow \dots \rightarrow z \rightarrow \dots \rightarrow v \Rightarrow$

si armo un nuevo recorrido Q uniendo las secciones P_{uz} y P_{zv}

tengo $l(Q) = l(P_{uz}) + l(P_{zv})$ y $l(P) = l(P_{uz}) + l(P_{zz}) + l(P_{zv}) \Rightarrow$

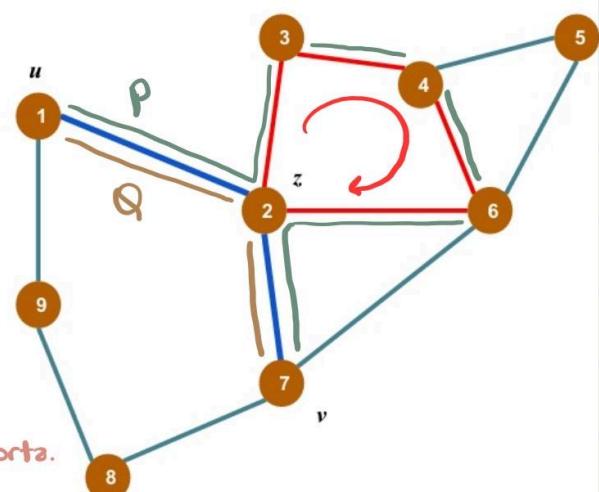
$l(Q) < l(P) = d(u,v)$

¡Absurdo! Porque por definición de distancia $d(u,v)$ es la longitud

del camino más corto.

y encontramos un recorrido con long. más corta.

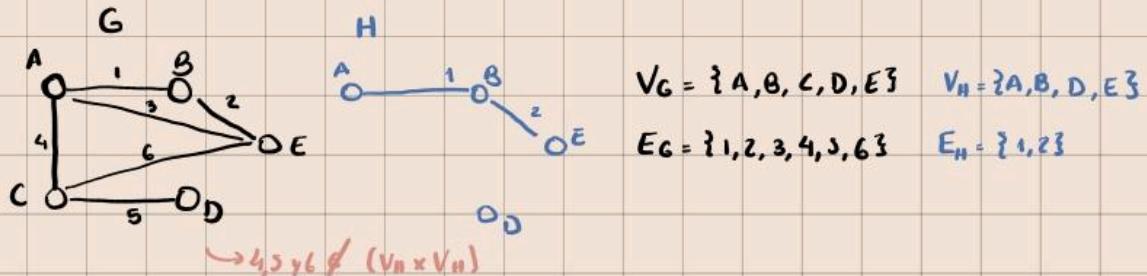
que es un camino



SUBGRAFOS

* Dado un grafo $G = (V_G, E_G)$ un **SUBGRAFO** es un grafo $H = (V_H, E_H)$ tal que $V_H \subseteq V_G$, $E_H \subseteq E_G \cap (V_H \times V_H) \Rightarrow H \subseteq G$

todas las aristas de G cuyas tuplas (V_H, V_H) contengan solo nodos que estan en V_H

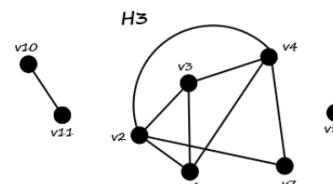
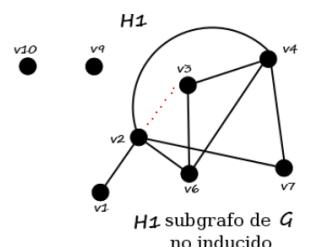
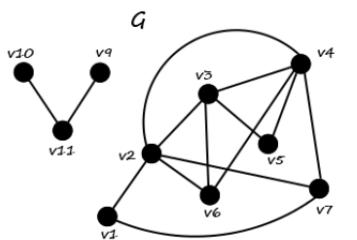
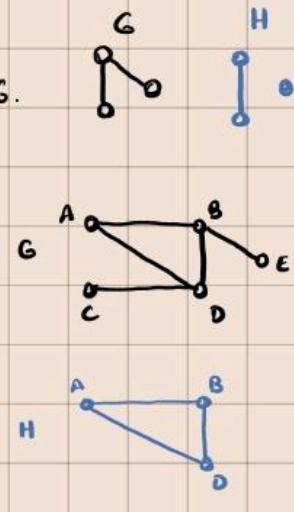


* Si $H \subseteq G$ y $H \neq G \Rightarrow H$ es un **subgrafo propio** de G .

* Si $H \subseteq G$ y $V_H = V_G \Rightarrow H$ es un **subgrafo generador** de G .

* Si $H \subseteq G$ y $\forall u, v \in V_H$ con $e = (u, v) \in E_G$ entonces $e \in E_H$ (tienen todas las aristas que conectan los vértices de H y están en G) y H es un **subgrafo inducido** de G .

Notamos a H $V_{[G]}$



H_3 subgrafo inducido de G por $\{v2, v3, v4, v6, v7, v8, v10, v11\}$

GRAFOS BIPARTITOS

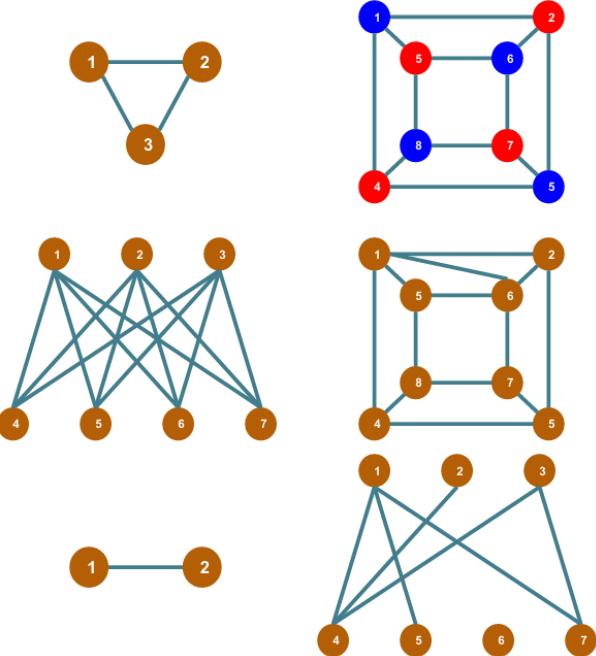
Un grafo $G = (V, E)$ es bipartito si su conjunto de vértices V puede dividirse en dos subconjuntos disjuntos U y W (es decir, $U \cap W = \emptyset$ y $U \cup W = V$), de modo que:

- Toda arista en E conecta un vértice de U con uno de W .
- No hay aristas entre vértices del mismo subconjunto.

Grafos bipartitos

Definición 12:

- Un grafo $G = (V, E)$ es **bipartito** si existen dos subconjuntos V_1 y V_2 de V tal que
 - $V = V_1 \cup V_2$
 - $V_1 \cap V_2 = \emptyset$.
 - Para todo $e = (u, v) \in E$, $u \in V_1$ y $v \in V_2$.
- Un grafo $G = (V, E)$ es **bipartito completo** con particiones V_1 y V_2 , si además todo vértice en V_1 es adyacente a todo vértice en V_2 .



GRAFOS ISOMORFOS

Dos grafos $G_1 = (V_1, E_1)$ y $G_2 = (V_2, E_2)$ son isomorfos si existe una biyección (correspondencia uno a uno) $f: V_1 \rightarrow V_2$ tal que:

- Dos vértices u y v son adyacentes en G_1 , si y solo si $f(u)$ y $f(v)$ son adyacentes en G_2 .

Misma estructura de conexiones, pero los vértices pueden estar "renombrados" o "reubicados".

Definición 13: Isomorfos

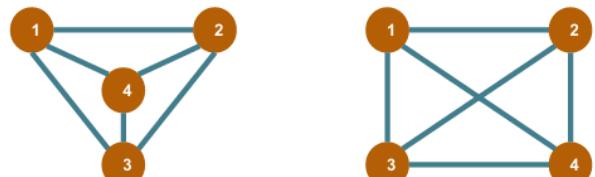
Dados $G = (V, E)$ y $G' = (V', E')$ son **isomorfos** si existe una función biyectiva $f: V \rightarrow V'$ tq para todo $u, v \in V$:

$$(u, v) \in E \Leftrightarrow (f(u), f(v)) \in E'$$

f : función de isomorfismo, $G = G'$ (abuso de notación)

Si dos grafos $G = (V, E)$ y $G' = (V', E')$ son isomorfos, entonces:

- Tienen el mismo número de vértices
- Tienen el mismo número de aristas
- Para todo k , $1 \leq k \leq n-1$, tienen el mismo número de vértices de grado k (la misma distribución de grado)
- Tienen el mismo número de c.c (componente conexa).
- Para todo k , $1 \leq k \leq n-1$, tienen el mismo número de caminos simples de longitud k



REPRESENTACIÓN DE GRAFOS

El tipo de representación que elijamos va a depender de:

- Las **características del grafo** (por ejemplo si es ralo, es decir, pocas aristas -lista- o denso, es decir, muchas aristas -matriz-)
- Para que lo vamos a querer **usar** y que complejidades queremos para sus operaciones.

Dpto. de Educación

Formas de representación

Conjuntos de nodos y aristas

 $V = \{1, 2, 3, 4, 5, 6\}$
 $E = \{(1,2), (1,3), (2,4), (2,5), (3,5), (3,6), (5,6)\}$
 $E = \{(1,2), (2,1), (1,3), (3,1), (2,4), (4,2), \dots\}$

(enlace simple) (enlace doble)

Lista de adyacencia

 $V = \{1, 2, 3, 4, 5, 6\}$

Nodos	Adyacentes
1	[2,3]
2	[1,4,5]
3	[1,5,6]
4	[2]
5	[2,3,6]
6	[3,5]

Matriz de adyacencia

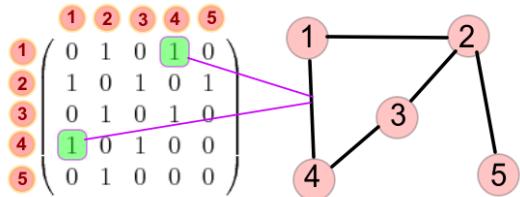
$$A: VxV = (a_{ij}) \quad a_{ij} = \begin{cases} 1 & \text{si } (i,j) \in E \\ 0 & \text{si } (i,j) \notin E \end{cases}$$

		Nodos					
		1	2	3	4	5	6
Nodos	1	0	1	1	0	0	0
	2	1	0	0	1	1	0
3	1	0	0	0	1	1	
4	0	1	0	0	0	0	
5	0	1	1	0	0	1	
6	0	0	1	0	1	0	

UNIVERSIDAD CAECE
Cátedra Argentina de Comercio y Servicios

Matriz de adyacencia:

Una matriz de adyacencia es una matriz cuadrada que representa un grafo, es decir, un conjunto de vértices y aristas. Cada fila y columna de la matriz representa un vértice del grafo, donde se rellena la matriz con unos y ceros, según si hay o no una arista entre cada par de vértices



- Es una buena implementación para un grafo cuando el número de aristas es grande
- Sin embargo, la mayoría de las celdas de la matriz están vacías, lo que hace que esta matriz sea "rala" (espacio $O(n^2)$)

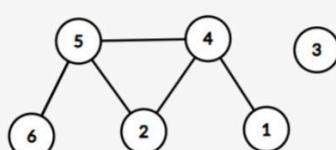
Lista de adyacencia:

Lista donde cada vértice tiene una lista de sus vecinos e indicamos a cada nodo con el índice del array, también se puede usar un diccionario para esto.

Lista de adyacencia

El diccionario es un vector y los vecindarios son listas de tamaño $d(v)$ conteniendo a los nodos vecinos.

Tomemos el siguiente grafo como ejemplo:



Lista de adyacencia:

Nodo : lista de vecinos

- 1 : 4
- 2 : 4 → 5
- 3 :
- 4 : 5 → 1 → 2
- 5 : 2 → 6 → 4
- 6 : 5

Nota: Se pueden hacer cosas como ordenar los vecindarios, pero es caro mantenerlo si el grafo cambia.

También se puede representar con un vector de vectores donde cada índice representa los $N(v)$ de un vértice.

Lista de aristas:

Una lista de aristas es una representación de un grafo que consiste en una lista de pares de vértices.

Ejemplo: [(A,B), (D,A), (B,C)]

Complejidades

Las complejidades para las representaciones vistas quedarían:

	lista de aristas	matriz de ady.	listas de ady.
construcción	$O(m)$	$O(n^2)$	$O(n + m)$
adyacentes	$O(m)$	$O(1)$	$O(d(v))$
vecinos	$O(m)$	$O(n)$	$O(d(v))$
agregarArista	$O(m)$	$O(1)$	$O(d(u) + d(v))$
removerArista	$O(m)$	$O(1)$	$O(d(u) + d(v))$
agregarVértice	$O(1)$	$O(n^2)$	$O(n) \rightarrow$ redimensionar el array
removerVértice	$O(m)$	$O(n^2)$	$O(n + m)$

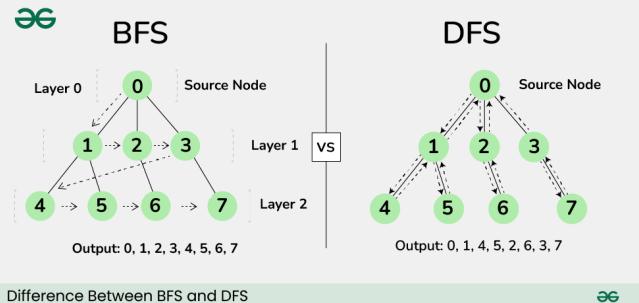
Nota: como el tamaño del grafo está dado por su cantidad de nodos y aristas, una complejidad $O(n+m)$ es lineal respecto al tamaño del grafo.

💡 la d indica degree (cantidad de aristas incidentes al nodo v)

BFS & DFS

BFS (búsqueda en amplitud) y **DFS** (búsqueda en profundidad) son algoritmos recursivos que permiten recorrer y buscar nodos en un grafo, lo cual es útil para determinar ciertas características de un grafo, como por ejemplo saber si es conexo, si hay ciclos, si existen caminos, etc.

Estos algoritmos se usan en grafos dirigidos y no dirigidos, y almacenan la información en pilas (en caso del DFS)



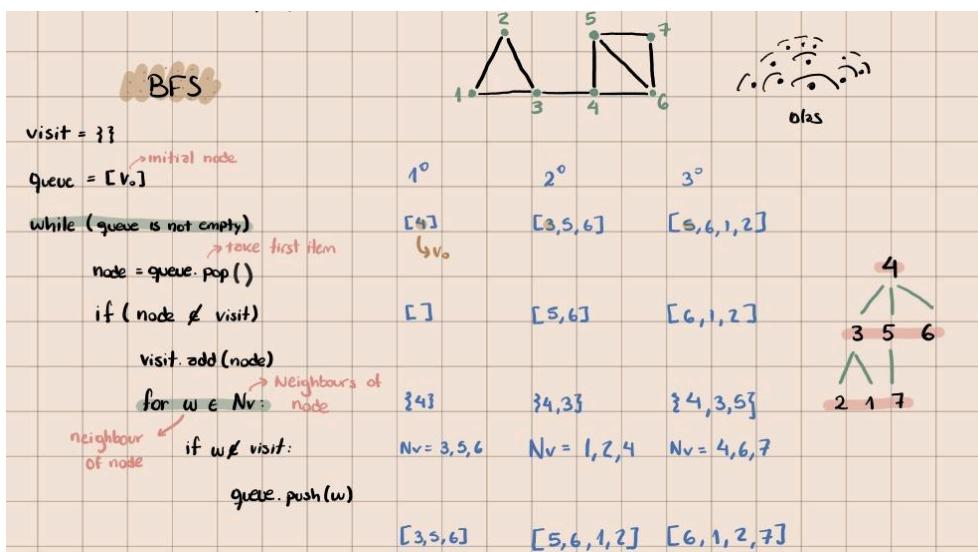
	BFS	DFS
Cómo explora	BFS es un enfoque en el que primero recorremos todos los nodos del mismo nivel antes de pasar al siguiente nivel.	DFS es un enfoque en el que el recorrido comienza en el nodo raíz y continúa a través de los nodos lo más lejos posible en cada rama hasta llegar al nodo sin nodos cercanos sin visitar.
Estructura de datos	Utiliza una cola (queue)	Utiliza una pila (stack)
Ventajas	Encuentra la ruta más corta (camino más corto) en grafos no ponderados.	Puede consumir menos memoria
Desventajas	Puede consumir más memoria	Puede no encontrar la ruta más corta.
Aplicaciones	Rutas más cortas, gráficos bipartitos	Búsqueda de componentes fuertemente conectados, gráficos acíclicos

➡️ DFS vs BFS, When to Use Which?

BFS

➡️ Breadth-first search in 4 minutes

➡️ Breadth First Search Algorithm | Shortest Path | Graph Theory



 Si busco hallar el camino más corto de un nodo a otro, contando la cantidad de aristas puedo usar un BFS.

DFS

La idea es **recorrer en profundidad** (Depth): **siempre se va hasta el final de la rama y de ahí sube** (como en backtracking).

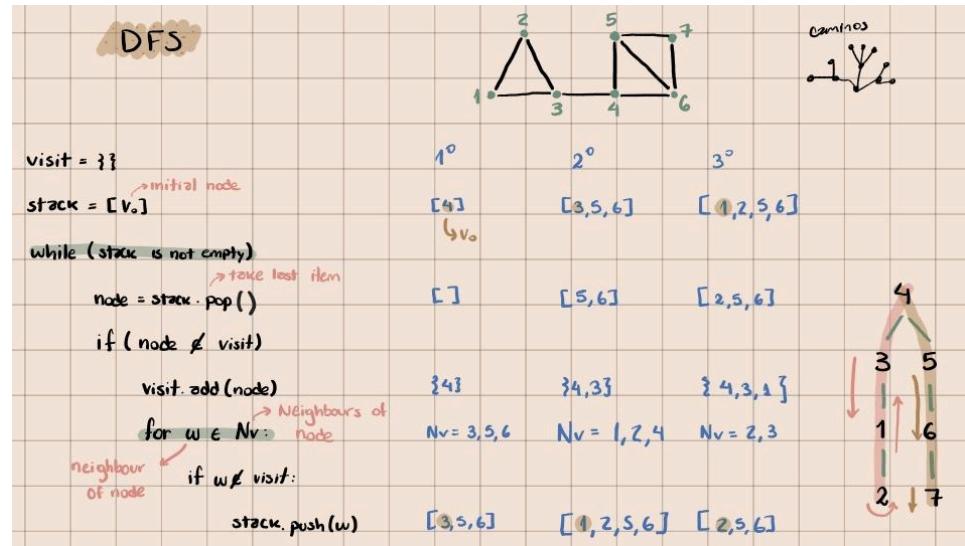
[Depth-first search in 4 minutes](#)

[Depth First Search Algorithm | Graph Theory](#)

Código Recursivo del algoritmo

```
vector<vector<int>> aristas;
vector<bool> visitado;

void dfs(int v) {
    visitado[v] = true;
    for (int u : aristas[v]) {
        if (!visitado[u])
            dfs(u);
    }
}
```



 Para ver si un grafo tiene ciclos podemos chequear si en la etapa de ver los vecinos de un nodo no hay repetidos que ya hayan sido visitados (sin contar el nodo del cual venimos)

Una **back edge** en un DFS es una **arista que conecta un vértice con uno de sus ancestros en el árbol DFS**. (En el ejemplo el nodo 6 y 4)

La complejidad de DFS es **O(m+n)** ya que **re corro todos los nodos una sola vez y reviso las aristas también una sola vez** (aunque puede que no recorra todas).

Complejidad

Los algoritmos para recorrer grafos (DFS y BFS), tienen complejidad **O(n+m)**, donde **n** es el **número de nodos (vértices)** del grafo y **m** es el **número de aristas** del grafo. Esta complejidad se debe a la forma en que exploramos los nodos y aristas en un grafo representado mediante listas de adyacencia.

Los algoritmos de DFS y BFS visitan:

- **Cada nodo una vez** → Se marca cada nodo como visitado una sola vez.
- **Cada arista una vez (en grafos dirigidos) o dos veces (en grafos no dirigidos)** → En BFS y DFS, al recorrer el grafo, procesamos cada arista en la lista de adyacencia al menos una vez.

ALGORITMOS DE CAMINOS CORTOS

Bellman-Ford y Floyd-Warshall, que son **algoritmos de programación dinámica** fundamentales para grafos con pesos, especialmente cuando hay pesos negativos, que **permiten resolver problemas de caminos más cortos entre otros problemas**.

💡 Funciona en grafos dirigidos y no dirigidos, donde las aristas pueden tener pesos negativos y positivos.

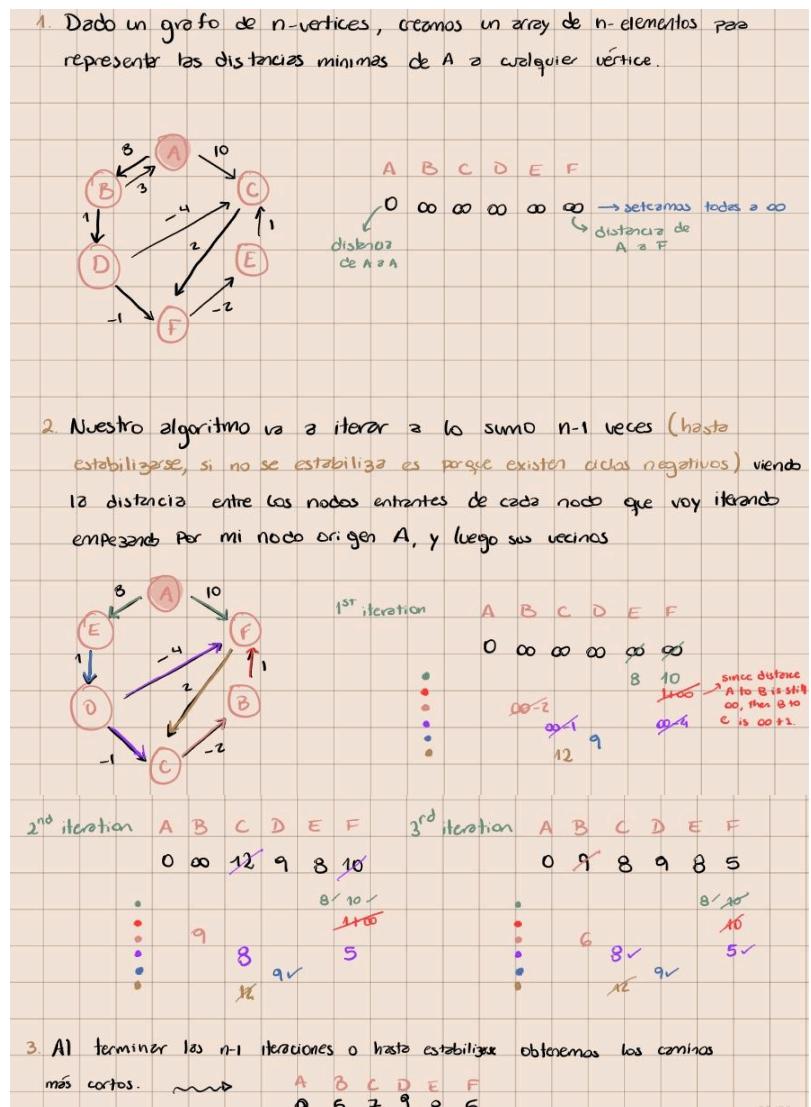
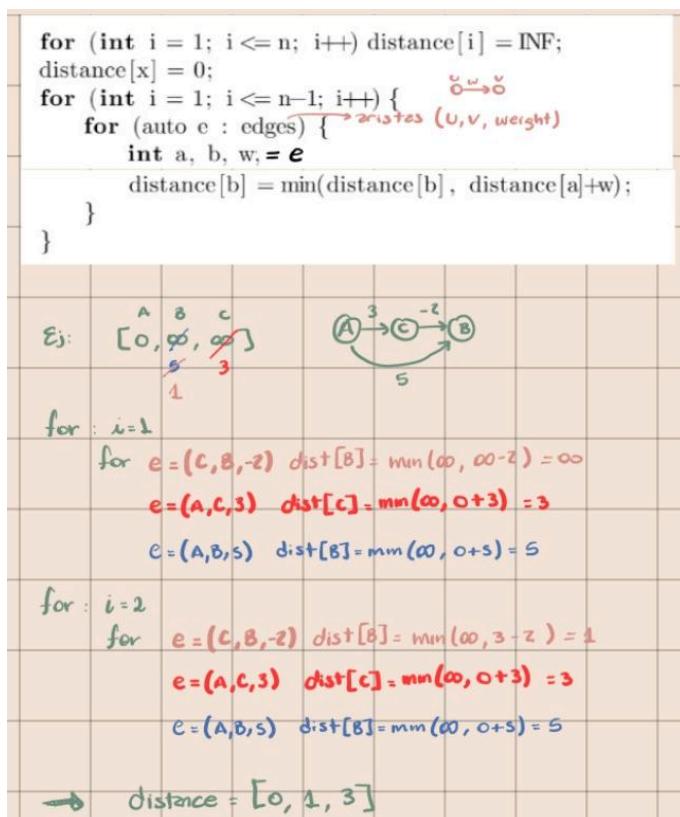
BELLMAN-FORD

Algoritmos de programación dinámica que se utiliza para cuando querés encontrar la **distancia más corta desde un solo nodo origen a los demás y cuando se quiere detectar ciclos negativos** (ciclo cuya suma total de pesos es negativa, sin importar si algunas aristas tienen peso positivo).

La idea de este algoritmo es:

- Inicializar las distancias desde el nodo origen (0 para el origen, ∞ para el resto).
- **Ver si pasar por un nodo intermedio u mejora la distancia conocida hasta v.** Esto lo repetimos $n-1$ veces (camino máximo entre dos nodos). Si sí, **actualizo** la distancia a v.
- Si en una iteración extra (la número n o posteriores) se puede mejorar la distancia, entonces hay un **ciclo negativo**
- **Si un grafo no tiene ciclos negativos, entonces podemos encontrar las distancias mínimas para la iteración $n-1$ o antes.**

La complejidad del algoritmo es $O(n \cdot m)$, donde $n = \text{número de nodos}$ y $m = \text{número de aristas}$.



💡 La idea es simple: al principio, solo conocés la distancia al nodo de inicio (cero), y el resto es infinito. Luego, en cada iteración, probás si llegar a un nodo b a través de otro nodo a mejora la distancia. Hacés esto $n-1$ veces (porque el camino más largo sin ciclos tiene $n-1$ aristas).

Bellman-Ford in 5 minutes — Step by step example

Bellman Ford Algorithm | Shortest path & Negative cycles | Graph Theory

Bellman Ford's Algorithm

Propiedad: Sea $D = (V, X)$ fuertemente conexo (dirigido y conexo), es decir, cualquier nodo es alcanzable desde cualquier otro. Entonces si existe un ciclo negativo en el grafo, se podrá identificar arrancando el algoritmo desde cualquier nodo.

Esta propiedad es muy útil porque si no tuviéramos un grafo fuertemente conexo, entonces deberíamos correr Bellman Ford para cada nodo individual y ver si existe un ciclo negativo. Entonces la complejidad se vuelve $O(n^2 \cdot m)$ y se vuelve más útil usar FW si $m > n$.

Además, funciona bien si el grafo es **disperso** (pocos arcos comparado con los nodos), porque sino $m = n-1$.

Otro tip para este lema, es que si usamos un “nodo fantasma” (agregamos un nodo) que esté conectado a todos de forma bidireccional tal que el grafo se vuelva fuertemente conexo, entonces podemos usar el lema de Bellman Ford y ver si tenemos ciclos negativos en $O(n \cdot m)$, en lugar de ver si cada nodo tiene un ciclo negativo nodo por nodo.

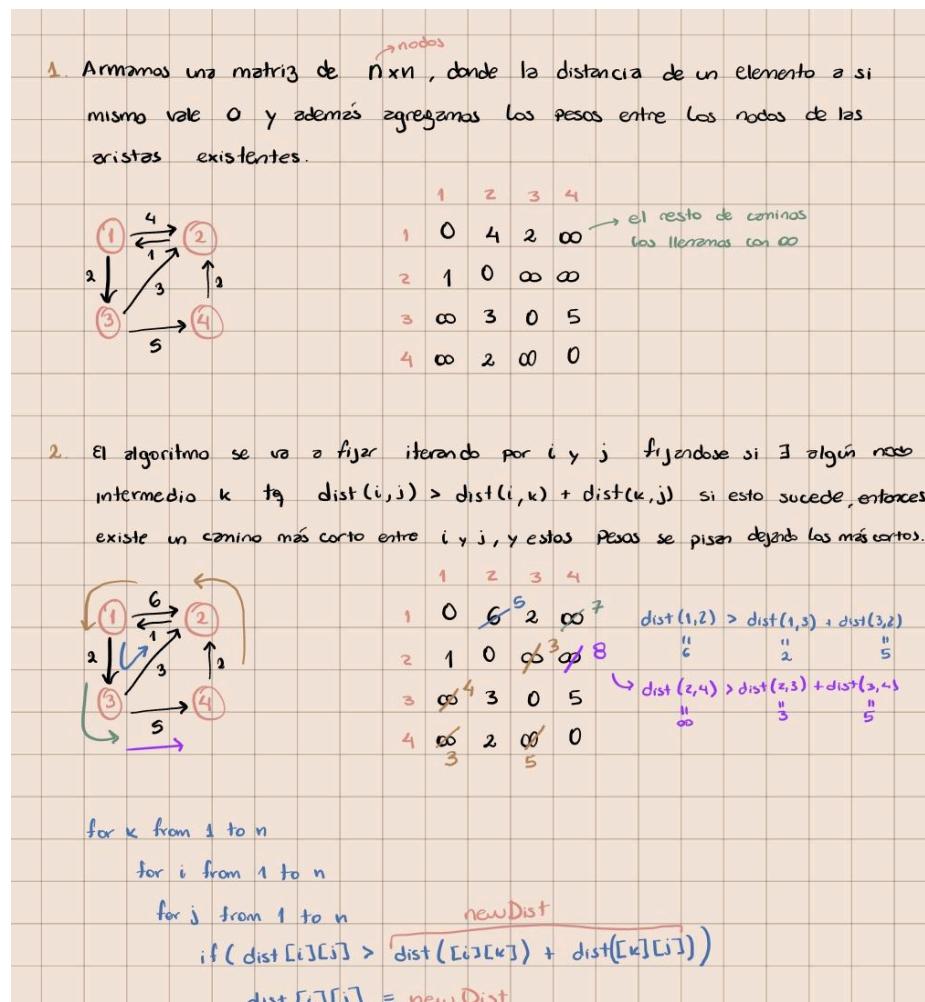
FLOYD-WARSHALL

Se utiliza cuando queremos **encontrar los caminos más cortos entre todos los pares de vértices**. Ideal para grafos pequeños a medianos (**porque usa mucha memoria y tiempo**). **Floyd-Warshall Algorithm is an algorithm for finding the shortest path between all the pairs of vertices in a weighted graph.**

 Este algoritmo de programación dinámica tiene muchas superposiciones de problemas.

Se utiliza una matriz de nxn nodos con las distancias de i a j de todos los nodos, luego se itera sobre todos los nodos intermedios posibles y se van reemplazando las distancias por las más cortas si existen caminos por nodos intermedios que acotan el camino

La complejidad temporal y espacial del algoritmo es $O(n^3)$ con n la cantidad de nodos en el grafo.



 [Floyd-Warshall algorithm in 4 minutes](#)

 [Floyd Warshall All Pairs Shortest Path Algorithm | Graph Theory | Dynamic Programming](#)

[Floyd-Warshall Algorithm](#)

```

n = no of vertices
A = matrix of dimension n*n
for k = 1 to n
    for i = 1 to n
        for j = 1 to n
            Ak-1[i, j] = min (Ak-1[i, j], Ak-1[i, k] + Ak-1[k, j])
return A

```

Floyd(G)
entrada: $G = (V, E)$ de n vértices
salida: D matriz de distancias de G

```

D ← L
para k desde 1 a n hacer
    para i desde 1 a n hacer
        para j desde 1 a n hacer
            d[i][j] ← min(d[i][j], d[i][k] + d[k][j])
        fin para
    fin para
retornar D

```

 La idea es "¿Puedo llegar de i a j más barato pasando por el nodo k?". En cada iteración de k, el algoritmo permite el uso del nodo k como intermedio para los caminos i→j.

La fórmula: $A[i][j] = \min(A[i][j], A[i][k] + A[k][j])$ significa "el mejor camino de i a j es el que ya conocía (el directo en caso de la primera iteración), o el que pasa por k , si es más corto." k es la iteración (nodo intermedio).

	Shortest Path (SP) Algorithms			
	BFS	Dijkstra's	Bellman Ford	Floyd Warshall
Complexity	$O(V+E)$	$O((V+E)\log V)$	$O(VE)$	$O(V^3)$
Recommended graph size	Large	Large/Medium	Medium/Small	Small
Good for APSP?	Only works on unweighted graphs	Ok	Bad	Yes
Can detect negative cycles?	No	No	Yes	Yes
SP on graph with weighted edges	Incorrect SP answer	Best algorithm	Works	Bad in general
SP on graph with unweighted edges	Best algorithm	Ok	Bad	Bad in general

ORDEN TOPOLOGICO

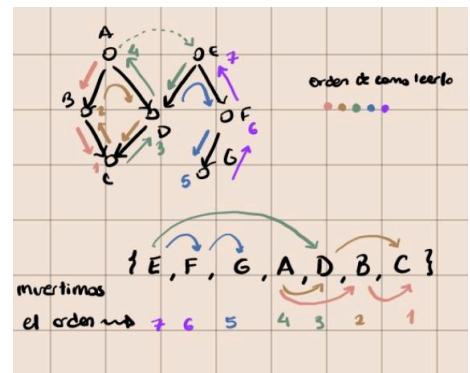
El **orden topológico** es una **propiedad que se da en grafos dirigidos acíclicos (DAG)** es una **enumeración de sus vértices** tal que para cada arista $u \rightarrow v$, el vértice u aparece **antes** que v en el orden.



💡 Si hay un ciclo, no existe un orden topológico válido (ya que no habría un nodo de donde iniciar).

Existen dos enfoques comunes para obtener este orden topológico:

- Kahn's algorithm (con BFS y grados de entrada).
- **DFS (Depth-First Search):** agregás los nodos al final de una lista al terminar su DFS, y luego la invertís. Su complejidad es correr DFS: $O(|V| + |E|)$ e invertir un array: $O(|V|)$, es decir, $O(|V| + |E|)$



UNIDAD 2 - COMPLEJIDAD

COMPLEJIDAD COMPUTACIONAL

La complejidad computacional es una rama de la informática que estudia la eficiencia de los algoritmos en **términos de tiempo** (cuánto tarda un algoritmo en ejecutarse) y **espacio** (cuánta memoria necesita).

La complejidad de un algoritmo es una función que representa el tiempo de ejecución en función del tamaño de la entrada.

La complejidad computacional se usa para:

- Seleccionar el algoritmo más eficiente para resolver un problema.
- Estimar los recursos necesarios para ejecutar un algoritmo en un sistema.
- Comparar diferentes algoritmos en términos de rendimiento.

💡 Una operación elemental es aquella cuyo tiempo de ejecución está acotado superiormente por un valor constante.

Instancia → Una instancia de un problema es un **conjunto válido de datos de entrada**.

1. **Entrada:** Un número n entero no negativo.
 2. **Salida:** ¿El número n es primo?
- En este ejemplo, una instancia está dada por un número entero no negativo.

💡 El tamaño de una instancia equivale a la cantidad de bits para un dato primitivo, o contando la cantidad de elementos para estructuras más complejas como arrays, grafos, matrices, etc.

TIEMPO DE EJECUCIÓN

► **Tiempo de ejecución de un algoritmo A:**
 $T_A(I)$ = suma de los tiempos de ejecución de las instrucciones realizadas por el algoritmo con la *instancia I*.

► Dada una instancia I , definimos $|I|$ como la cantidad de bits necesarios para almacenar los datos de entrada de I .

1. Si b está fijo y la entrada ocupa n celdas de memoria, entonces $|I| = bn = O(n)$. *El módulo de una instancia equivale al número de celdas.*

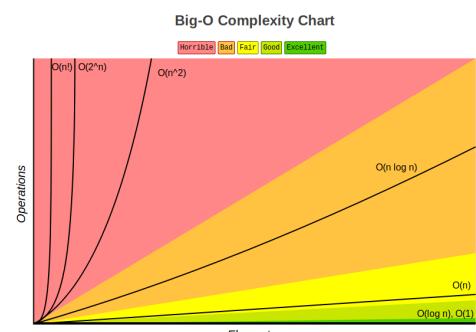
► **Complejidad de un algoritmo A:**
 $f_A(n) = \max_{I:|I|=n} T_A(I)$.

↳ mayor tiempo de ejecución dado el peor n , es decir, la instancia cuyo módulo es más grande.

Hay n celdas de memoria, cada una compuesta por b bits.
n celdas
...
b bits
...
Conjunto de valores

❖ Notación O , Ω y Θ

- Si un algoritmo es $O(\log n)$, se dice **logarítmico**.
- Si un algoritmo es $O(n)$, se dice **lineal**.
- Si un algoritmo es $O(n^2)$, se dice **cuadrático**.
- Si un algoritmo es $O(n^3)$, se dice **cúbico**.
- Si un algoritmo es $O(n^k)$, $k \in \mathbb{N}$, se dice **polinomial**.
- Si un algoritmo es $O(d^n)$, $d \in \mathbb{R}_{>1}$, se dice **exponencial**.



 La función logarítmica es mejor que la función lineal (no importa la base), es decir “ $\log n$ ” es “ $O(n)$ ” pero no vale la inversa.

 Cualquier función exponencial es peor que cualquier función polinomial.

Un algoritmo polinomial es aquel cuya complejidad puede expresarse como un polinomio en función del tamaño de la entrada, como $O(n^2)$ o $O(n^3)$. Estos algoritmos son considerados satisfactorios porque su tiempo de ejecución crece de manera controlada y predecible a medida que aumenta el tamaño de la entrada.

Por otro lado, los algoritmos supra-polinomiales tienen una complejidad que crece más rápido que cualquier polinomio, como $O(2^n)$ o $O(n!)$. Estos algoritmos son considerados no satisfactorios porque, para entradas grandes, el tiempo de ejecución crece muy rápidamente, volviéndose impráctico.

Dadas dos funciones $T, g : \mathbb{N} \rightarrow \mathbb{R}$, decimos que:

- ▶ $T(n) = \mathcal{O}(g(n))$ si existen $c \in \mathbb{R}_+$ y $n_0 \in \mathbb{N}$ tales que

$$T(n) \leq c g(n) \text{ para todo } n \geq n_0.$$

T no crece más rápido que g , $T \preceq g$.

- ▶ $T(n) = \Omega(g(n))$ si existen $c \in \mathbb{R}_+$ y $n_0 \in \mathbb{N}$ tales que

$$T(n) \geq c g(n) \text{ para todo } n \geq n_0.$$

T crece al menos tan rápido como g , $T \succeq g$.

- ▶ $T(n) = \Theta(g(n))$ si

$$T = \mathcal{O}(g(n)) \text{ y } T = \Omega(g(n)).$$

T crece al mismo ritmo que g , $T \approx g$.

Ejemplos:

- ▶ 3^n no es $\mathcal{O}(2^n)$.

▶ Vamos a demostrarlo por el absurdo.

▶ Supongamos que sí, es decir que 3^n es $\mathcal{O}(2^n)$.

▶ Entonces, por definición, existirían $c \in \mathbb{R}_+$ y $n_0 \in \mathbb{N}$ tales que $3^n \leq c 2^n$ para todo $n \geq n_0$.

▶ Por lo tanto, $(\frac{3}{2})^n \leq c$ para todo $n \geq n_0$.

▶ Esto genera un absurdo porque c debe ser una constante y no es posible que una constante siempre sea mayor que $(\frac{3}{2})^n$ cuando n crece.

- ▶ Si $a, b \in \mathbb{R}_+$, entonces $(\log_a(n))$ es $\Theta(\log_b(n))$.

Es decir, todas las funciones logarítmicas crecen de igual forma sin importar la base.

$$\text{▶ Como } \log_a(n) = \frac{\log_b(n)}{\log_b(a)}$$

▶ la constante $\frac{1}{\log_b(a)}$ sirve tanto para ver que $(\log_a(n))$ es $\mathcal{O}(\log_b(n))$

▶ como para $(\log_a(n))$ es $\Omega(\log_b(n))$.

ALGORITMO DE FUERZA BRUTA

Un algoritmo de fuerza bruta o búsqueda exhaustiva se enfoca en generar todas las posibles soluciones (todas las soluciones factibles generadas por un problema) hasta encontrar la correcta.

Estos métodos exploran exhaustivamente el espacio de soluciones posibles. Si bien es simple de implementar, puede ser muy inefficiente, especialmente cuando el número de posibles soluciones es grande, ya que, un algoritmo de fuerza bruta tiene una complejidad exponencial.

Los algoritmos de fuerza bruta se usan cuando:

- El espacio de soluciones es pequeño y manejable.
- Se necesita una solución garantizada y no hay mejor método conocido.
- Se desea un punto de partida básico para comparar con otros algoritmos más eficientes.

 Esta técnica siempre funciona y es fácil de implementar e inventar, pero son inefficientes, sin embargo esta técnica es útil para comparar respuestas con las soluciones de otras técnicas, ya que este método explora todas las soluciones y podemos ver si nuestras otras técnicas funcionan o no.

Problema: Hallar todas las formas posibles de colocar n reinas en un tablero de ajedrez de $n \times n$ casillas, de forma que ninguna reina amenace a otra.

- ▶ Solución por FB: hallar *todas* las formas posibles de colocar n reinas en un tablero de $n \times n$ y luego seleccionar las que satisfagan las restricciones.
- ▶ En cada configuración tenemos que elegir las n posiciones donde ubicar a las reinas de los n^2 posibles casilleros.
- ▶ Entonces, el número de configuraciones que analizará el algoritmo es:

$$\binom{n^2}{n} = \frac{n^2!}{(n^2 - n)!n!}$$

- ▶ Pero fácilmente podemos ver que la mayoría de las configuraciones que analizaríamos no cumplen las restricciones del problema y que trabajamos de más.

BACKTRACKING

El backtracking es una **técnica de algoritmos** similar a fuerza bruta, probando diferentes posibilidades y retrocediendo (haciendo "backtrack") cuando se alcanza un callejón sin salida (esta es una mejora). Es una mejora sobre la fuerza bruta porque evita explorar soluciones que no pueden llevar a una solución válida.

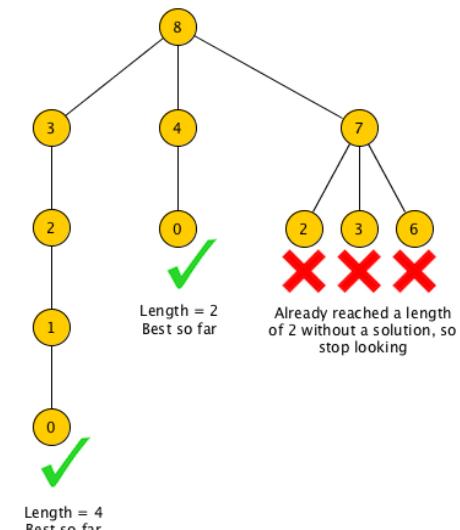
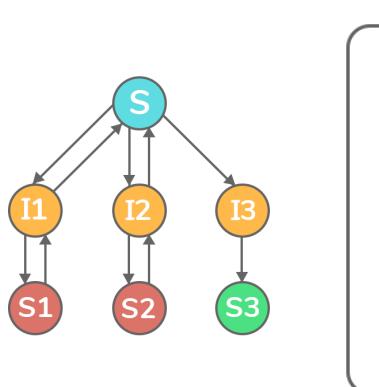
💡 La idea es recorrer todas las posibles configuraciones del espacio de soluciones de manera recursiva.

💡 La cantidad de nodos del árbol las calculamos como 2^N en caso de tener dos hijos por nodo y una cantidad de N elementos.

Dentro del backtracking tenemos 3 tipos de soluciones:

- **Soluciones Parciales** → Son soluciones incompletas que se construyen paso a paso durante la exploración del algoritmo para obtener soluciones candidatas. (todos los nodos)
- **Soluciones Candidatas** → Son soluciones finales que podrían convertirse en una solución válida o no válida según si se cumplen las restricciones del problema. (todas las hojas)
- **Soluciones Válidas** → Son soluciones completas que satisfacen todas las restricciones del problema.

💡 Las hojas son soluciones candidatas completas que pueden ser válidas o no, pero pueden haber nodos, no hojas (soluciones parciales), que ya cumplan y sean solución válida, entonces podemos cortar ahí la recursión.



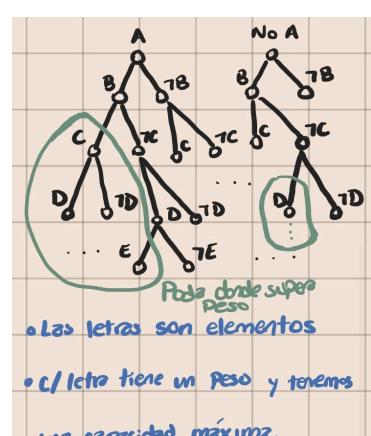
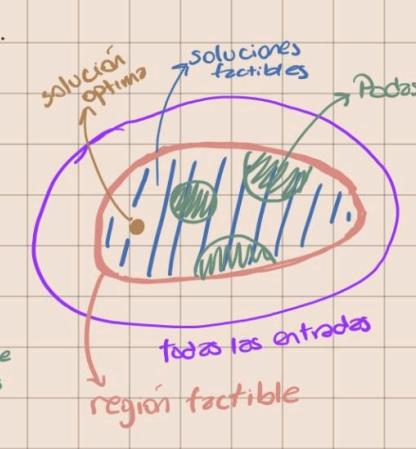
- Soluciones parciales son todos los nodos *
- Soluciones candidatas, son los nodos rojos y verdes *
- Soluciones válidas, son los nodos verdes *

💡 El máx o mín depende del problema a resolver.

$$z^* = \max_{x \in S} f(x) \quad \text{o bien} \quad z^* = \min_{x \in S} f(x)$$

- ▶ La función $f : S \rightarrow \mathbb{R}$ se denomina **función objetivo** del problema. *Conjunto de entradas que cumplen con todas las restricciones de un Problema.*
- ▶ El conjunto S es la **región factible** y los elementos $x \in S$ se llaman **soluciones factibles**. *cualquier entrada que cumple con las restricciones.*
- ▶ El valor $z^* \in \mathbb{R}$ es el **valor óptimo** del problema, y cualquier solución factible $x^* \in S$ tal que $f(x^*) = z^*$ se llama un **óptimo** del problema. *Resultado que mejor responde al problema en base a las entradas*

💡 Entrada que cumple que $f(x) = z^*$



PODA

Una poda es una condición o criterio que se usa para evitar explorar ramas del árbol de decisiones que no llevan a una solución válida y se retrocede hasta encontrar un vértice con un hijo válido por donde seguir la exploración. Esto es una optimización para el algoritmo de Backtracking, ya que nos ahorra analizar soluciones que no son válidas.

 En cada paso, se evalúa si la solución parcial puede llegar a ser una solución completa válida. Si se determina que no puede serlo, el algoritmo "poda" esa rama, es decir, abandona esa dirección y retrocede al último punto de decisión para probar otra opción.

 En este caso se mantiene la complejidad de peor caso “O grande” (no se poda nada), pero si mejora la complejidad del mejor caso “Ω” (se podan muchas ramas).

 Las podas no se suelen poner en las funciones recursivas (muchas veces no se puede), estas se deben agregar en el código.

- **Poda por factibilidad** → Elimina ramas del árbol de búsqueda **donde las soluciones parciales violan restricciones básicas del problema** (si esta ruta ya no cumple las reglas, no sigas por aquí).
Por ejemplo (en N-Reinas): Si colocas una reina en una casilla amenazada, poda esa rama (no es factible seguir explorando).
- **Poda por Optimización** → Elimina ramas donde las soluciones parciales no pueden mejorar la mejor solución actual (si esta ruta ya es peor que la mejor que tenemos, descártala)
Por ejemplo (en Mochila 0/1): Si el valor acumulado de los ítems + el máximo valor posible restante es menor que el valor óptimo actual, poda la rama.

 La factibilidad se basa en restricciones ("¿Es válido?"), mientras que la Optimalidad se basa en optimización ("¿Puede ser mejor?").

"The first version of a backtracking algorithm does not contain any optimizations. We simply use backtracking to generate all possible paths, optimizations help reduce the amount of possible paths by cutting those who don't follow the conditions given"

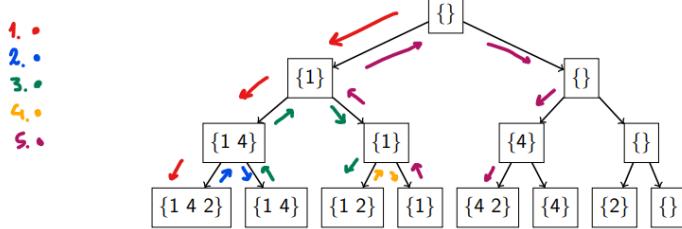
Para demostrar que una poda es válida se utiliza la **propiedad del efecto dominó**, si una solución parcial $a = (a_1, a_2, \dots, a_k)$ no cumple las propiedades deseadas, tampoco lo hará cualquier extensión posible de ella.

Ejemplo: Dado el problema de las 8 reinas si una solución parcial (a_1, \dots, a_k) donde cada reina está en una columna distinta y los a_i indican la fila donde cada reina se ubica en dicha columna, con $k \leq 8$, entonces si una solución parcial tiene reinas que se amenazan, entonces toda extensión de ella seguro tendrá reinas que se amenazan. Por lo tanto es correcto podar una solución parcial que tiene reinas que se amenazan.

 Para demostrar la correctitud de un algoritmo de backtracking con o sin podas, debemos demostrar que se pueden llegar a todas las soluciones válidas.

RECORRIDO DE SOLUCIONES

- Cada elemento puede o no estar en el subconjunto.



- ¿Cuántos nodos tiene el árbol que estamos recorriendo?
- Tiene $\sum_{i=0}^N 2^i = O(2^N)$ nodos.

Para hallar el conjunto S de posibles soluciones usamos **backtracking**, esta técnica nos permite generar espacios de búsqueda “recursivos”, usando **dfs** (algo que veremos luego, pero básicamente toma una rama y la profundiza antes de pasar a la siguiente), mediante la extensión de soluciones parciales.

ALGORITMO

La idea básica es tratar de extender una solución parcial del problema hasta, eventualmente, llegar a obtener una solución completa, que podrá ser válida o no. Habitualmente, **se utiliza un vector $a = (a_1, a_2, \dots, a_n)$ para representar una solución candidata**, cada a_i pertenece a un dominio/conjunto finito A_i , donde en cada paso se extienden las soluciones parciales $a = (a_1, a_2, \dots, a_k)$, $k < n$, agregando un elemento más, $a_{k+1} \in S_{k+1} \subseteq A_{k+1}$, al final del vector a .

El espacio de soluciones es el producto cartesiano $A_1 \times \dots \times A_n$.

Se puede pensar este espacio de búsqueda como un árbol dirigido donde cada vértice representa una solución parcial, donde la raíz del árbol se corresponde con el vector vacío (la solución parcial vacía).

El algoritmo toma dos parámetros, a y k .

$a \rightarrow$ Una **solución parcial** que se está construyendo..

$k \rightarrow$ Índice o contador que ayuda a gestionar la construcción de la **solución**. Por ejemplo, si nos piden tomar 4 elementos de un conjunto de 10, nuestro k máximo será 4.

```

algoritmo BT(a, k)
    si a es solución entonces
        procesar(a)
        retornar
    sino
        para cada a' ∈ Sucesores(a, k)
            BT(a', k + 1)
        fin para
    fin si
    retornar
  
```

Si a es solución:

Aquí el algoritmo verifica si la solución parcial a es una solución completa (**candidata**) al problema. Si es una solución entonces, **procesar(a)**, verifica si es una solución válida y lo almacena, lo cuenta, o lo imprime. **Después de procesar la solución, el algoritmo termina esta rama de ejecución y retorna para probar otras posibilidades.**

Si a no es solución:

Si a no es una solución completa, entonces se procede a expandirla. Para cada $a' \in \text{Sucesores}(a, k)$: El algoritmo genera todos los posibles "sucesores" de la solución parcial a . Cada sucesor a' es una versión extendida de a que incluye un nuevo elemento o paso. $BT(a', k + 1)$: El algoritmo llama recursivamente a sí mismo para cada sucesor a' , incrementando k en 1 para reflejar el avance en la construcción de la solución.

Ejemplo (Reinas): <https://www.youtube.com/watch?v=s7kBjMOMWq0&t=23s>

Un ejemplo común es el problema del n -reinas, donde se deben colocar n reinas en un tablero de ajedrez de $n \times n$ de manera que ninguna reina ataque a otra. El algoritmo de *backtracking* coloca una reina y luego procede a colocar la siguiente. Si se coloca una reina en una posición donde no se puede colocar ninguna más, el algoritmo retrocede y prueba una nueva posición para la reina anterior.

- ▶ Por ejemplo, para $n = 8$ una implementación directa consiste en generar **todos los subconjuntos** de casillas.
 $2^{64} = 18,446,744,073,709,551,616$ combinaciones!
- ▶ Sabemos que dos damas no pueden estar en la misma casilla.
 $\binom{64}{8} = 4,426,165,368$ combinaciones.
- ▶ Sabemos que cada columna debe tener exactamente una dama. Cada solución parcial puede estar representada por (a_1, \dots, a_k) , $k \leq 8$, con $a_i \in \{1, \dots, 8\}$ indicando la fila de la dama que está en la columna i .
Tenemos ahora $8^8 = 16,777,216$ combinaciones.
- ▶ Adicionalmente, cada fila debe tener exactamente una dama.
Se reduce a $8! = 40,320$ combinaciones.

Se cumple la propiedad dominó: Si una solución parcial (a_1, \dots, a_k) , $k \leq 8$, tiene reinas que se amenazan, entonces toda extensión de ella seguro tendrá reinas que se amenazan. Por lo tanto es correcto podar una solución parcial que tiene reinas que se amenazan.

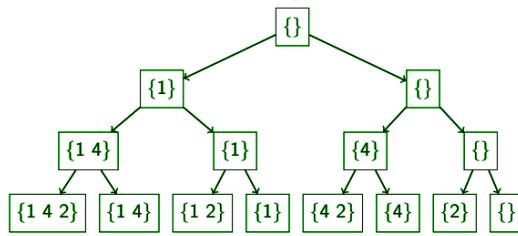
Ejemplo (CD):

Enunciado

Tenemos un CD que soporta hasta P minutos de música, y dado un conjunto de N canciones de duración p_i (con $1 \leq i \leq m$, y $p_i \in \mathbb{N}$) queremos encontrar la mayor cantidad de minutos de música que podemos escuchar.

Con $P = 5$ y una lista de $N = 3$ canciones con duraciones $[1, 4, 2]$ la solución es 5.

- Cada elemento puede o no estar en el subconjunto.



Algorithm $BT_{CD}(a, i)$ // a es una solución parcial

```

1: if  $i = N$  then
2:   if  $suma(a) \leq P$  &  $suma(a) > mejorSuma$  then
3:      $mejorSuma \leftarrow suma(a)$ 
4:   end if
5: else
6:    $BT_{CD}(a \cup p_i, i + 1)$ 
7:    $BT_{CD}(a, i + 1)$ 
8: end if
  
```

- Cada nodo interno del nivel i representa un subconjunto de los primeros i elementos. Por ende para extender cada solución se agrega o no el elemento $i + 1$.

- ¿Qué podas podemos usar en CD?
- En CD podemos dejar de avanzar si la solución parcial ya superó N (**factibilidad**).
- También podemos ver si poniendo todas las canciones restantes no nos excedemos de k . Si ese es el caso, la mejor solución desde donde estamos es agregar todo (**optimalidad**).

Otra forma de pensarla:

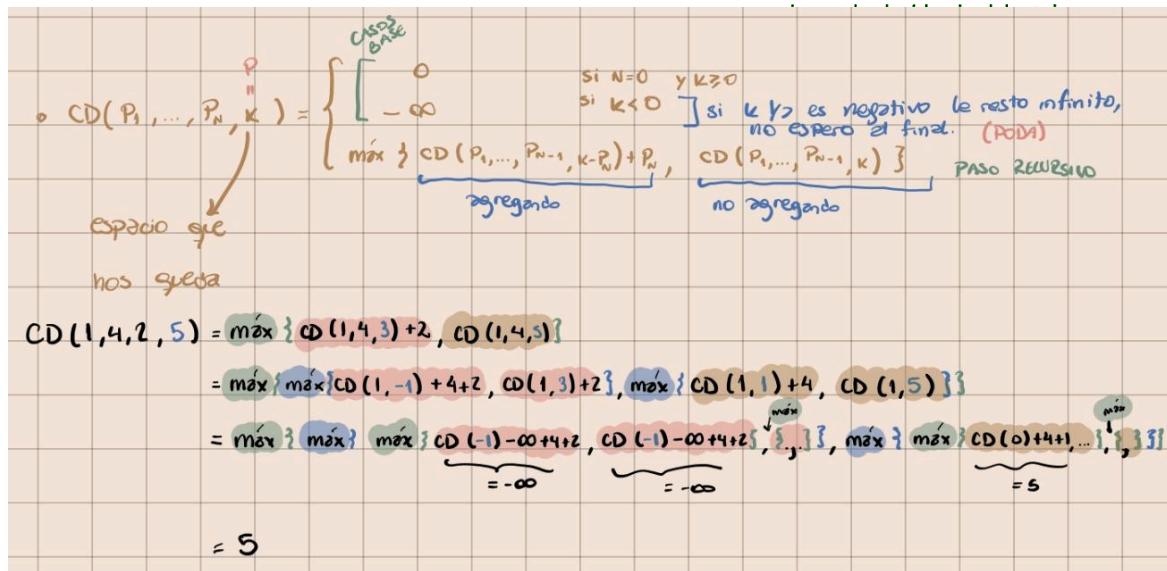
Plantear funciones de esta pinta nos va a ser muy útil cuando hagamos programación dinámica. Intuitivamente, el árbol de recursión de esta función es el mismo que el de los subconjuntos. Sin embargo, en esta formulación queda claro que no importan qué elementos se fueron eligiendo, sino la suma de los pesos.

El problema se puede pensar de otra forma: si quiero llegar a P , y agrego el primer elemento al CD...

- ¿A cuánto se quiere llegar con el resto de los temas?
- ¿Y si no se lo agrega?
- Si no quedan temas ($N=0$), ¿Qué valores de P son válidos?

$$CD(i, k) = \begin{cases} -\infty & \text{si } i = N \text{ y } k < 0 \\ 0 & \text{si } i = N \text{ y } k \geq 0 \\ \max(CD(i+1, k), CD(i+1, k-p_i) + p_i) & \text{cc} \end{cases}$$

'La máxima cantidad de música que puedo obtener sin exceder k minutos empleando las

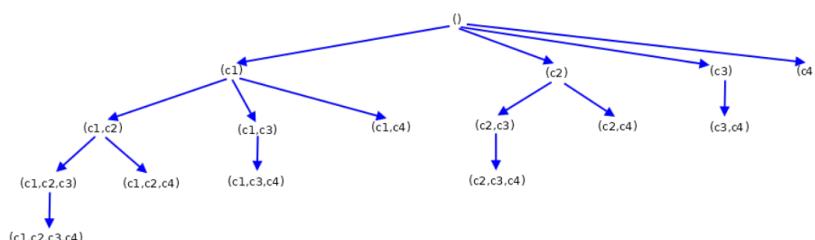


Ejemplo (Sumatoria de Subconjuntos):

Problema: Dado un conjunto de naturales $C = \{c_1, \dots, c_n\}$ (sin valores repetidos) y $k \in \mathbb{N}$, queremos encontrar un subconjunto de C (o todos los subconjuntos) cuyos elementos sumen k .

- ▶ Una solución puede estar representada por $a = (a_1, \dots, a_r)$, con $r \leq n$, $a_i \in C$ y $a_i \neq a_j$ para $0 \leq i, j \leq r$.
- ▶ Es decir, el vector a contiene los elementos del subconjunto de C que representa.
- ▶ En este caso, todos los vértices de árbol corresponden a posibles soluciones (no necesariamente válidas), no sólo las hojas. Hay que adaptar a esto el esquema general que presentamos previamente.
- ▶ Como las soluciones son conjuntos, no nos interesa el orden en que los elementos fueron agregados a la solución.
- ▶ Entonces (c_4, c_5, c_2) y (c_2, c_4, c_5) representarían la misma solución, el subconjunto $\{c_2, c_4, c_5\}$.
- ▶ Queremos evitar esto porque estaríamos agrandando aún más el árbol de búsqueda.
- ▶ Los vértices de nivel 1 del árbol de búsqueda se corresponderán a las n soluciones de subconjuntos de un elemento, (c_i) , para $i = 1, \dots, n$.
- ▶ En el segundo nivel, no quisiera generar los vectores (c_i, c_j) y (c_j, c_i) , sino sólo uno de ellos porque representan la misma solución.
- ▶ Para esto, para la solución parcial (c_i) podemos sólo crear los hijos (c_i, c_j) con $j > i$. Así evitaríamos una de las dos posibilidades.
- ▶ De forma general, cuando extendemos una solución, podemos pedir que el nuevo elemento tenga índice mayor que el último elemento de a .

Si $n = 4$, el árbol de búsqueda es:



- Para cada vértice del árbol podemos ir guardando la suma de los elementos que están en la solución.
- Si encontramos un vértice que sume k , la solución correspondiente será una solución válida.
- Se cumple la propiedad dominó: Si una solución parcial (a_1, \dots, a_r) suma más que k , entonces toda extensión de ella seguro también sumará más que k . Es decir, las soluciones no se *arreglan* al extenderlas.
- Esto no sería cierto si hay elementos negativos en C .
- Si la suma excede k , esa rama se puede podar, ya que los elementos de C son positivos.

Podemos definir los sucesores de una solución como:

$$\text{Sucesores}(a, k) = \{(a, a_k) : a_k \in \{c_{s+1}, \dots, c_n\}, \text{ si } a_{k-1} = c_s \text{ y } \sum_{i=1}^k a_i \leq k\}.$$

Ejemplo (Prime Ring):

Prime Ring

Dados N números naturales p_0, \dots, p_{N-1} , con $1 < p_i < 10N$, queremos saber cuántas permutaciones j de ellos hay que cumplan que $p_{j_i} + p_{(j_{i+1} \bmod n)}$ sea primo para todo $0 \leq i \leq n - 1$

- Lo vamos a resolver con backtracking.
- ¿Cuál es el espacio de búsqueda? Todas las permutaciones de los naturales p_0, \dots, p_{N-1} .
- ¿Cuáles son las soluciones parciales? Los prefijos de las permutaciones.
- ¿Cuál es la operación de extensión? Agregar un elemento al prefijo de permutación.

La función que hay que implementar entonces es:

$$\text{primeRing}(I) = \begin{cases} \text{esValida}(I) & \text{si } |I| = N \\ \sum_{p_i \notin I} \text{primeRing}(I \oplus p_i) & \text{cc} \end{cases}$$

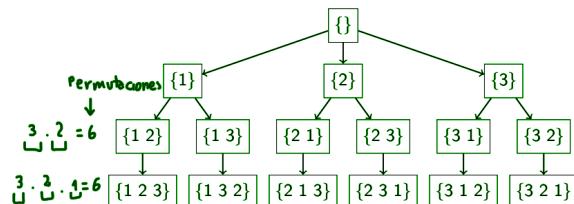
'La cantidad de permutaciones que extienden a I , usan todos los elementos de p y generan un anillo de primos'

La solución al problema es $\text{primeRing}(\{\})$

Ejemplo:

- Permutación (1, 2, 3):
 - $1 + 2 = 3$ (primo)
 - $2 + 3 = 5$ (primo)
 - $3 + 1 = 4$ (no es primo)
 - Esta permutación no cumple con la condición.
- Permutación (2, 3, 1):
 - $2 + 3 = 5$ (primo)
 - $3 + 1 = 4$ (no es primo)
 - $1 + 2 = 3$ (primo)
 - Esta permutación no cumple con la condición.

Árbol para $n = 3$, si $p = [1, 2, 3]$.



Si agregamos podas

- Podríamos verificar la condición de primalidad durante la selección de sucesores.
- El árbol cambia: ahora las soluciones parciales son las permutaciones de los subconjuntos que cumplen la condición de primalidad.
- ¿Hay que verificar algo en las hojas?
- En las hojas solo tenemos que verificar que cierre bien el anillo.

La función queda entonces como

$$\text{primeRing}(I) = \begin{cases} \text{esPrimo}(\text{ultimo}(I) + \text{primero}(I)) & \text{si } |I| = N \\ \sum_{p_i \notin I} \text{primeRing}(I \oplus p_i) & \text{cc} \\ \text{esPrimo}(p_i + \text{ultimo}(I)) & \end{cases}$$

Solo verifico el primero y último porque en cada paso siempre que agregó un elemento antes de continuar con su sucesor, me fijo que la suma de primo.

Ejemplo (Sudoku):

Enunciado

Dado un tablero de Sudoku de $N \times N$ con algunas casillas ocupadas hay que decidir si se lo puede completar siguiendo las reglas del Sudoku.

- ¿Cuáles son las soluciones parciales? Tableros incompletos.
- ¿Cuál es la función de extensión? Colocar un valor en algún casillero.
- ¿Qué verificamos en las hojas? Revisamos si es un tablero válido.

$$sudoku(T, (i, j)) = \begin{cases} \text{esValido}(T) & \text{si estoy en un casillero ya completo} \\ \text{sudoku}(T, \underline{\text{sig}(i, j)}) & \text{si } i = N \\ \bigvee_{1 \leq k \leq N} \text{sudoku}(T \oplus ((i, j) \rightarrow k), \underline{\text{sig}(i, j)}) & \text{si } T[i][j] \neq 0 \\ \text{cc} & \end{cases}$$

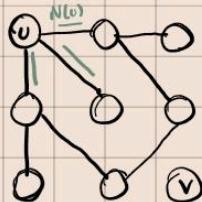
← Las casillas son elegidas al azar

Esta función toma un tablero (parcialmente completado) y un índice del mismo, y “prueba” todas las formas de llenar ese casillero y pasa al siguiente (donde el siguiente es el casillero a su derecha o bien el primero de la siguiente fila, si nos encontramos al borde).
La solución es $sudoku(T, (0, 0))$

- ¿Qué podías podemos implementar?
- No pongamos números que ya están prohibidos. Para eso revisamos las filas, columnas y subcuadrados en cada paso.
- ¿Hace falta ir en orden?
- No, usemos siempre la posición mas condicionada (o sea, la posición en la cual hay una menor cantidad de opciones válidas restantes).

BACKTRACKING EN GRAFOS

- Ejemplo:
 - $u \rightarrow$ casa de perro
 - $v \rightarrow$ Facultad
 - $P =$ perros
- $E =$ conexiones entre perros = $\{(P_i, P_j) / \exists$ puede viajar entre $P_i y P_j\}$



$G = (P, E)$ ¿ Hay recorrido que empiece en u y termine en V ?

→ me fijo si estando en los vecinos de u puedo llegar a los vecinos de v

sea $f(G, u, v) = 0$

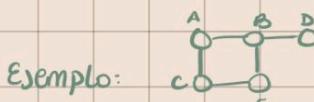
$\begin{cases} \text{true } u=v \\ \text{false } N(u)=\emptyset \wedge u \neq v \\ s \in N(u) \end{cases}$	$\vee f(G, v, s, v)$ <small>cuando sacamos un vértice asumimos que tamb se sacan sus aristas.</small>	<small>① si busco un recorrido no debo eliminar los nodos que visita, si busco un camino entonces si los debo eliminar.</small>
<small>Disjunción (ó-logico) mientras haya un camino devolver true</small>		

Depth-First Search

- Depth-First Search (DFS) - Búsqueda en profundidad.

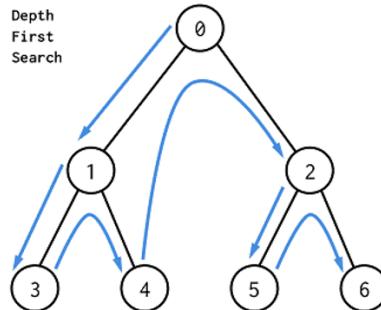
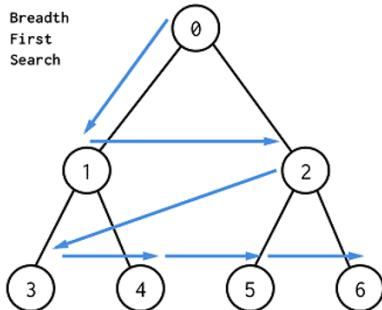
DFS es un algoritmo de recorrido de grafos que explora hasta el fondo de una rama antes de retroceder (backtracking) y probar caminos alternativos.

Podemos pensar como que DFS avanza por un camino en un laberinto hasta llegar a un callejón sin salida, luego vuelve atrás para explorar rutas no visitadas



Recorrido DFS desde A : $A \rightarrow B \rightarrow D$ (backt.) $B \rightarrow E \rightarrow C$ (backtrack)
 Orden de visita : A, B, D, E, C

4/5



PROGRAMACIÓN DINÁMICA

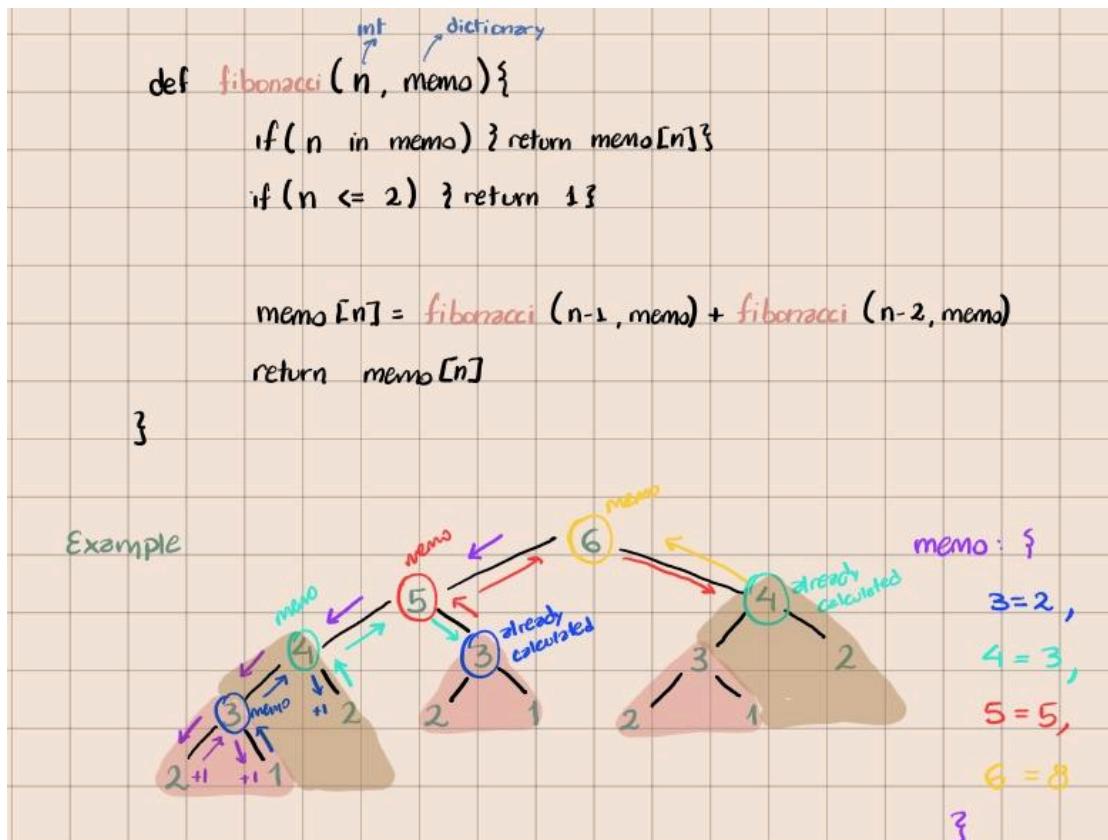
La programación dinámica es una **técnica de diseño de algoritmos** utilizada para resolver problemas dividiéndolos en **subproblemas más simples** y almacenando los resultados de estos subproblemas para evitar cálculos repetidos. Es especialmente útil en problemas de optimización donde hay una superposición de subproblemas, es decir, donde los mismos subproblemas se resuelven varias veces.

La idea principal es descomponer el problema original en partes más pequeñas, resolver cada una de esas partes solo una vez, y **almacenar sus soluciones en una estructura de datos como una tabla (o matriz)**, de modo que si se necesita resolver el mismo subproblema nuevamente, se pueda obtener la solución directamente de la tabla, en lugar de recalcularla.

💡 Cuando se hace una llamada recursiva a un subproblema primero se chequea si este subproblema ya ha sido resuelto. Si lo fue, se utiliza el resultado almacenado. En caso contrario, se realiza la llamada recursiva.

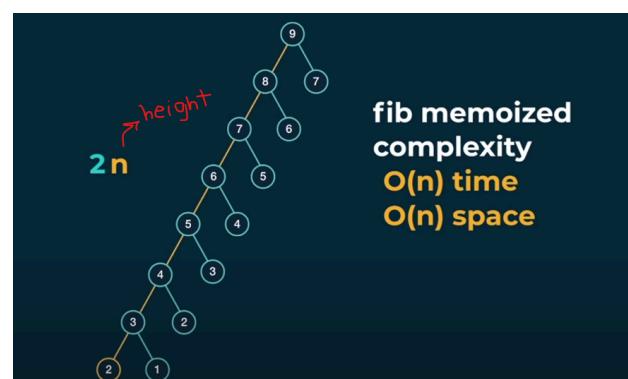
💡 Se memorizan los parámetros de las llamadas recursivas como índices de la matriz o vector, donde se almacenan los valores calculados.

Superposición de estados → El árbol de llamadas recursivas resuelve el mismo problema varias veces.



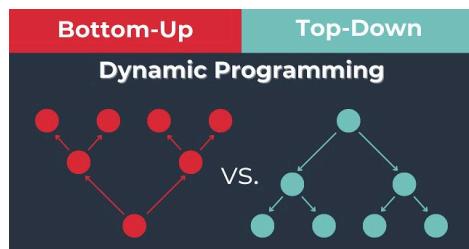
💡 Tenemos mapeados todos los valores de la función (bottom-up), entonces en lugar de calcular cada coeficiente binomial, solamente le enviamos dos parámetros a la matriz, y este me devuelve el valor calculado.

💡 La complejidad de un algoritmo usando programación dinámica no es 2^n , si no que la cantidad de casos, la cantidad de pasos que evaluamos sin contar repetidos (los que se superponen)



TOP-DOWN & BOTTOM-UP

En programación dinámica, los dos **enfoques principales para evitar la repetición de cálculos** son **top-down** y **bottom-up**. Ambos enfoques tienen el mismo objetivo: resolver el problema original de manera eficiente, pero difieren en la forma en que abordan los subproblemas.



- **Top-Down (memorización)** es útil cuando el problema tiene una estructura recursiva natural y cuando **algunos subproblemas no se necesitan calcular, ahorrando tiempo**. **Resuelve problemas recursivamente y almacena resultados para evitar repeticiones**.

💡 Usualmente es recursivo, porque no se cuando termino de resolver cada rama. En cada paso recursivo voy guardando en un vector o matriz los resultados y los uso cada vez que necesite calcular algo ya calculado.

- **Bottom-Up (tabulación)** es preferible cuando **todos los subproblemas deben resolverse**, y puede ser más eficiente en términos de espacio porque no requiere mantener la pila de recursión. **Resuelve iterativamente desde los subproblemas más pequeños hasta el problema original**.

💡 Usualmente es iterativo, porque se desde donde quiero empezar y donde quiero terminar. En BottomUp calculo todo (desde el caso inicial hasta el final) y voy guardando cada cálculo en un vector o matriz y luego devuelvo el valor guardado en la matriz o vector con los parámetros.

💡 Inclusive, en bottom up, podemos liberar memoria, si algunos cálculos no van a volver a ser usados, lo cual permite guardar la información en variables fijas o estructuras como colas.

<https://www.log2base2.com/algorithms/dynamic-programming/dynamic-programming.html> (another explanation)

- ▶ Un algoritmo de programación dinámica evita estas repeticiones con alguno de estos dos esquemas:

1. **Enfoque top-down.** Se implementa recursivamente, pero se guarda el resultado de cada llamada recursiva en una estructura de datos (**memorización**). Si una llamada recursiva se repite, se toma el resultado de esta estructura.
2. **Enfoque bottom-up.** Resolvemos primero los subproblemas más pequeños y guardamos (habitualmente en una tabla) todos los resultados.

Considera nuevamente el problema de la serie de Fibonacci:

- **Top-Down:** Comenzarías intentando calcular $F(n)$. Si $F(n)$ depende de $F(n - 1)$ y $F(n - 2)$, calcularías esos valores recursivamente. Si ya has calculado $F(n - 1)$, lo reutilizarías desde la memoria en lugar de recalcularlo.
- **Bottom-Up:** Comenzarías calculando $F(0)$ y $F(1)$, almacenando esos valores. Luego calcularías $F(2)$ utilizando $F(0)$ y $F(1)$, y seguirías hasta llegar a $F(n)$.

CUANDO USAR CADA TÉCNICA

