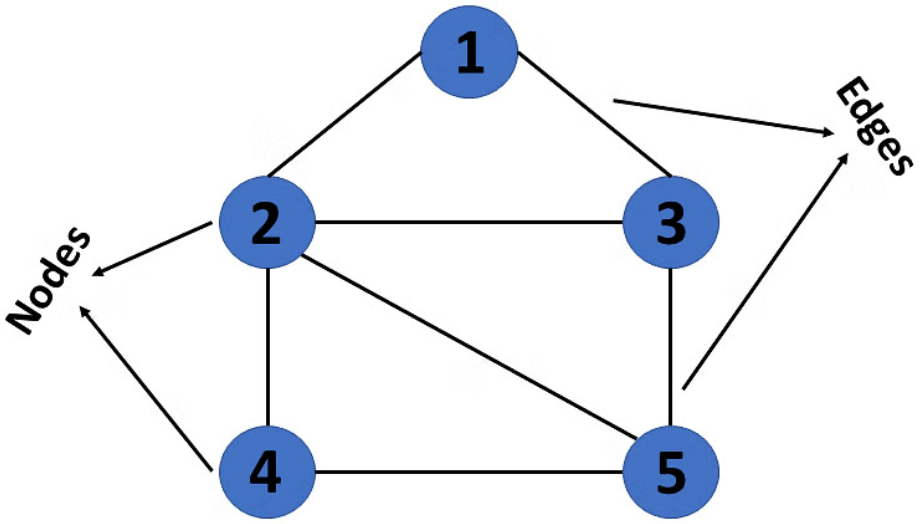
Técnicas y Diseños de Algoritmos

Tomás Agustín Hernández

Grafos

Un grafo es una estructura de datos compuesta por **V un conjunto de vértices (nodos)** y **E un conjunto de aristas que conectan pares de vértices**.
Definimos un grafo como: $G = (V, E)$ donde $E \subseteq VXV$.



Nótese que tiene sentido que el conjunto de aristas E esté formado por un par VXV. De lo contrario no existiría una arista.
Good to Know

- Si dos vértices están conectados por una arista, entonces estos se llaman **adyacentes**.
- La relación entre dos vértices define a $e = (u, v) \in E$. En este caso, decimos que **e** es incidente a **u** y **v**.
- La cantidad de vértices la notamos con la letra **n** y la cantidad de aristas con la letra **m**.

¿Por qué le decimos vértices a los nodos?

Porque los grafos tienen un fundamento mucho más matemático.

Self Loop (Bucle)

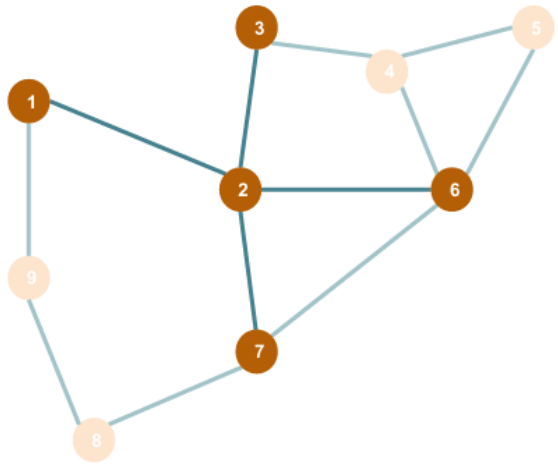
En un grafo, hablamos de self loop cuando un vértice se relaciona consigo mismo.

Cantidad de vertices y cantidad de aristas

Definimos la cantidad de vértices de un grafo como $|V| = n$ y la cantidad de aristas como $|E| = m$.
Good to Know: En un grafo T (árbol) **siempre** sucede que $|V| > |E|$ en una unidad. Lo veremos más adelante, pero está bueno notarlo.

Vecindad de un vértice

Llamamos vecindad de un vértice v a todos los vértices adyacentes de v.
Formalmente, lo definimos como: $N(v) = \{u \in V : (v, u) \in E\}$



Ej.: $N(2) = \{1, 3, 7, 6\}$
Good to Know: Se utiliza la letra N de Neighbor (vecino) en inglés.

Grafo Simple

Un Grafo Simple es un grafo que no tiene más de una arista entre dos mismos nodos. Nótese que aquí no importa la relación entre ellos, sino que, no debería estar repetido el mismo par (sin importar el orden).

- $E = \{(A, B), (A, B)\}$ NO es un Grafo Simple.
- $E = \{(A, B), (B, A)\}$ o $E = \{(B, A), (A, B)\}$ NO es un Grafo Simple.
- $E = \{(A, B)\}$ SÍ es un Grafo Simple.

Grafo no Simple

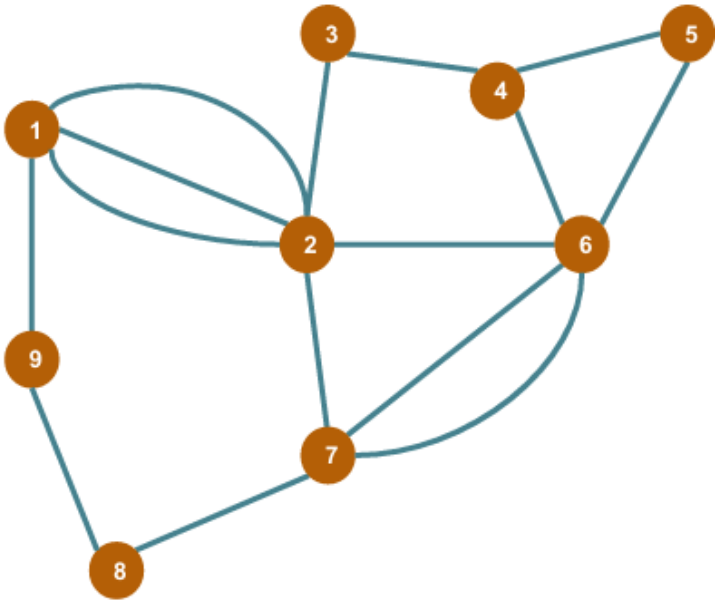
Un Grafo no Simple es un grafo que no tiene ninguna restricción con respecto a la relación entre dos vértices (nodos).

- $E = \{(A, B), (A, B)\}$ es un Grafo no Simple.
- $E = \{(A, B), (B, A)\}$ o $E = \{(B, A), (A, B)\}$ es un Grafo no Simple.
- $E = \{(A, B)\}$ SÍ es un Grafo no Simple.

Good to Know: Los grafos simples, son subconjuntos de grafos no simples. Por ende, si tenemos un grafo no simple que cumple las propiedades de un grafo simple, consideramos que es un **grafo simple** al ser más restrictivo.
En los ejemplos anteriores, entonces, $E = \{(A, B)\}$ sería mejor considerado como Grafo Simple.

Multigrafo

Un Multigrafo es un tipo de **grafo no simple** el cual puede haber más de una arista entre el mismo par de vértices.



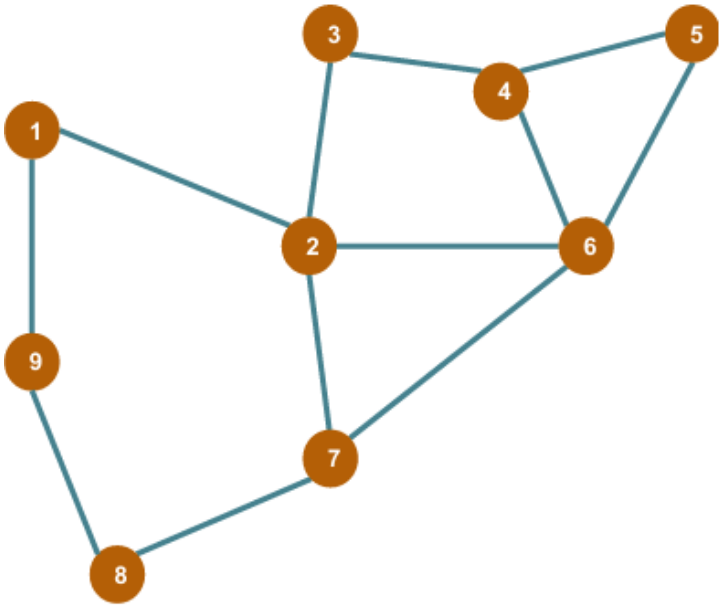
En el gráfico anterior, se definió el siguiente multigrafo: $G = \{\mathbf{(1,\ 2)}, \mathbf{(1,\ 2)}, \mathbf{(1,\ 2)}, (2,3), (3,4), (3,2), (4,3)...\}$

Pseudografo

Un Pseudografo es un tipo de **grafo no simple** el cual tiene la particularidad de que puede haber más de una arista entre el mismo par de vértices, y, además, los vértices pueden tener self-loops.
Good to Know: Puede haber más de un self-loop de un vértice.

Grado de un vértice (nodo)

El grado de un vértice es la cantidad de conexiones que tiene un nodo.
Definimos formalmente al grado de un vértice como $deg(v) = \#\{e \in E : v \in e\}$
Good to Know: deg refiere a degree.
Good to Know 2: **e** es una arista particular $e = (v, w)$.
Good to Know 3: En un grafo G, al grado mínimo lo notamos como $\delta(G)$ mientras que al grado máximo lo notamos como $\triangle(G)$



En este caso particular, tenemos que $\delta(G) = 2$ y $\triangle(G) = 4$, y, a modo de ejemplo, $deg(2) = 4$
Good to Know 4: Por cada par de vértices, tenemos una suma de grado 2. Ej.: $\{(1, 2)\}$ nos da $deg(1) + deg(2) = 2$

Grado de un vértice en un Multigrafo

Exactamente igual que en un Grafo, pero acá hay que aclarar que como podemos tener una relación entre dos vértices más de una vez tenemos que contar cada arista. Por ejemplo, si tenemos $\{(1,2),(1,2)\}$ entonces $deg(1) = 2$.

Grado de un vértice en un Pseudografo

Exactamente igual que en un Grafo, pero acá el self-loop cuenta 2 veces. Por ejemplo, si tenemos $\{(1,2),(1,2),(1,1),(1,1)\}$ entonces $deg(1) = 6$. Si en algún momento te maréas, el self-loop vale dos porque tenés el mismo número en el par.

Suma de los grados de los vértices

Dado un grafo G = (V, E) la **la suma de los grados** de sus vértices es igual a 2 veces el número de aristas. Es decir,

$$\sum_{v \in V} deg(v) = 2 \times \#\{e \in E\} = 2m$$

Para ver acerca de cómo demostrarlo, véase **anexo**
Corolario: En un grafo, siempre hay una cantidad par de vértices que tienen grado impar. Ej.: $\{(1,2),(2,3)\}, deg(1) = 1, deg(2) = 2, deg(3) = 1$

Grafo dirigido (digrafo)

Un grafo dirigido es un tipo de grafo (que puede ser simple o no simple) pero importa quién se relaciona con quién, es decir, el orden en que guardamos la relación. En dibujos, lo notamos con una flecha.

- $E = \{(A,B)_{\rightarrow}, (B,A)_{\leftarrow}\}$ es un grafo dirigido, esto quiere decir que $(A,B) \neq (B,A)$

Good to Know: A nivel estructura de datos, parece que E es igual en un Grafo Dirigido que un Grafo No Dirigido. No obstante, optamos, a nivel de notación incluir una fecha para indicar la relación entre ambos. A nivel código, deberías manejar alguna información extra para entender si es dirigido o no.

Grado de un vértice en un grafo dirigido

En un Grafo Dirigido, el grado de un vértice se calcula como $deg(v) = indeg(v) + outdeg(v)$
¿A qué nos referimos con indeg(v) y outdeg(v)? Como es un grafo dirigido, la relación entre los vértices importa. Por lo tanto, aquellas aristas entrantes al nodo v origen las agrupamos en indeg(v) mientras que aquellas aristas salientes desde el nodo v las agrupamos en outdeg(v).

- $E = \{(A,B)_{\rightarrow}, (B,A)_{\leftarrow}, (A,C)_{\rightarrow}\}$ es un grafo dirigido. Siendo indeg(A) = 1, outdeg(A) = 2 $\implies deg(A) = 3$

Cantidad de Aristas de un Grafo Dirigido Simple

Como acá importa la relación los vértices pues contamos todas las aristas pues $(A,B) \neq (B,A)$

$$0 \leq |E| \leq n(n-1)$$

Cantidad de Aristas de un grafo dirigido no simple

No hay límite, pueden ser infinitas. No hay fórmula concreta.

Grafo no dirigido

Un Grafo no dirigido es un grafo el cual no existe la dirección en una relación entre dos vértices.

- $E = \{(A,B),(B,A)\}$ es un grafo no dirigido, esto quiere decir que $(A,B) = (B,A)$

Good to Know: Los grafos dirigidos o grafos no dirigidos pueden ser grafos simples o no.

Cantidad de aristas de un grafo no dirigido simple

Como no importa la relación entre los vértices pues $(A,B) = (B,A)$, contamos solo una vez esa arista

$$0 \leq |E| \leq \frac{n(n-1)}{2} = \binom{n}{2}$$

Cantidad de aristas de un grafo no dirigido no simple

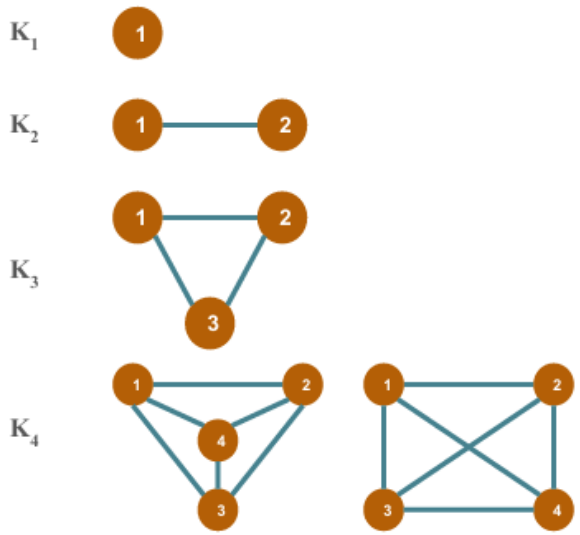
No hay límite, pueden ser infinitas. No hay fórmula concreta.

Grado de un vértice en un grafo no dirigido

Es la misma cuenta que hacemos en un grafo dirigido, pero sin distinguir en grupos a las aristas **(porque no tienen dirección)**.

Grafo completo

Llamamos Grafo Completo a un Grafo que tiene todos sus vértices están conectados entre sí.
Formalmente, lo notamos como K_n donde **n** es la cantidad de vértices que tiene el grafo.



Grafo complemento

Un Grafo Complemento (G^C) es el Grafo que tiene todas las **aristas** que no estaban en G.
Ej.: $G = \{(1,2),(3,2)\}$ entonces $G^c = \{(1,3),(2,3)\}$

Clausura transitiva y grafo transitivo

La **clausura transitiva** de un grafo dirigido $G = (V, E)$, denotada $G^+ = (V, E^+)$, es otro grafo que contiene una arista (u, v) si existe un camino desde u hasta v en el grafo original. Es decir:

$$(u, v) \in E^+ \iff \text{existe un camino de } u \text{ a } v \text{ en } G.$$

Un grafo se dice **transitivo** si ya contiene todas esas conexiones, es decir:

$$G = G^+.$$

Ejemplo: Si $E = \{(1,2),(2,3)\}$, entonces su clausura transitiva es $E^+ = \{(1,2),(2,3),(1,3)\}$.

Cantidad de aristas de un Grafo Complemento

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

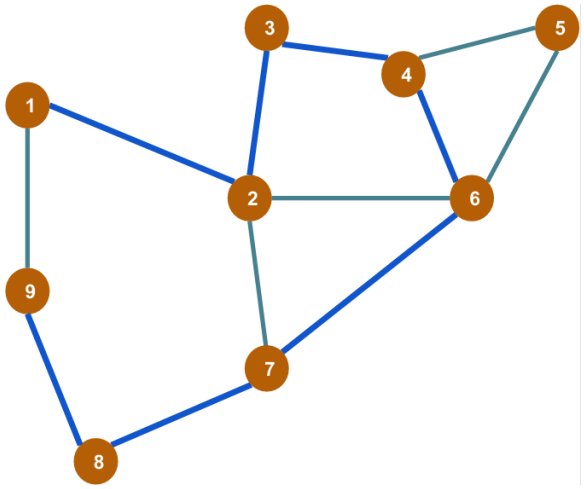
Creo que es necesario que sea un Grafo simple no dirigido

Recorrido

Un recorrido 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 $i = 1...k : P = v_0 e_1 v_1 e_2 ... e_k v_k$ (vértice, par de vértices, vértice, par de vértices, vértice...).
Ej.: $P = 1 \to (1,2) \to 2 \to (2,3) \to 3 \to (3,4) \to 4 \to (4,6) \to 6 \to (2,6) \to 2...$
Es decir, el nodo anterior (si hay) debe estar en la siguiente arista en alguna de las posiciones.

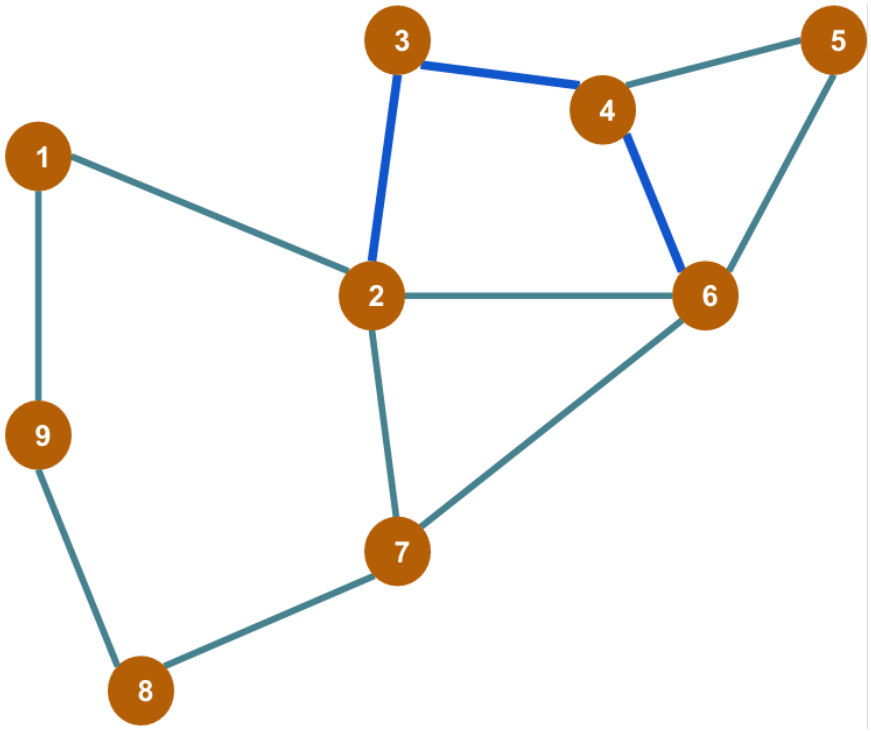
Camino

Un camino es un recorrido **que no pasa dos veces** por el mismo vértice.
Ej.: $P = 1 \to 2 \to 3 \to 4 \to 5 \to 6 \to 7 \to 8 \to 9$



Sección (subconjunto/parte de un recorrido)

Una sección es un tramo del recorrido P, se nota $P_{v_i v_j}$. Nótese que P es el recorrido, y los subíndices es aquello que acota al mismo.
Ej.: $P_{2,6} = 2 \to 3 \to 4 \to 6$. En este ejemplo, $v_i = 2$ y $v_j = 6$.

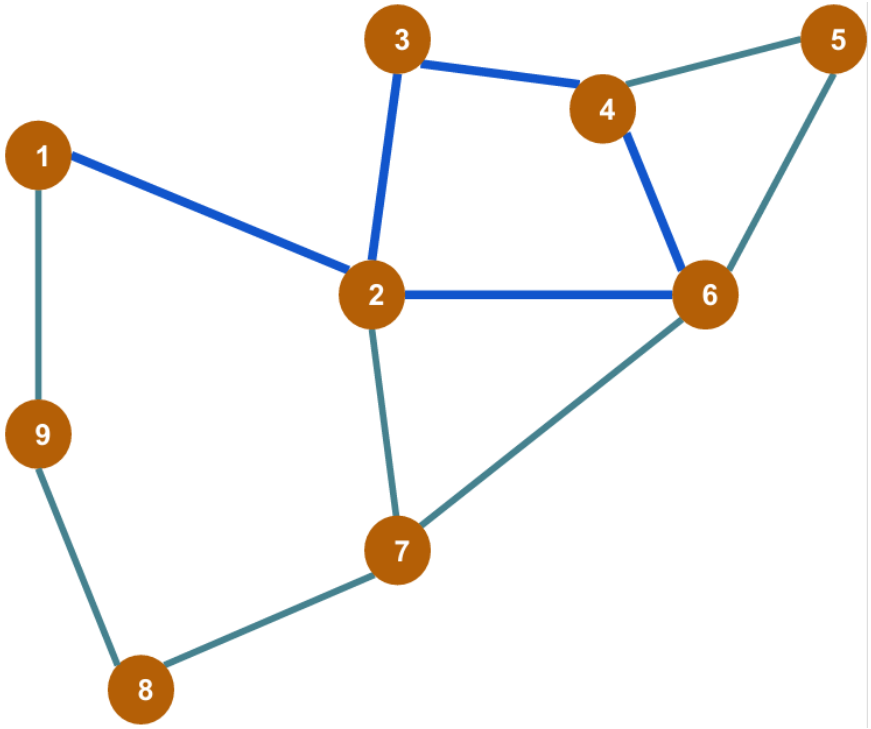


Good to Know

- Una sección es un subconjunto de un recorrido P.

Circuito

Un circuito es un recorrido que empieza y termina en el mismo vértice, se nota $P_{v_i v_j}$
Ej.: $P_{1,1} = 1 \to 2 \to 3 \to 4 \to 6 \to 2 \to 1$

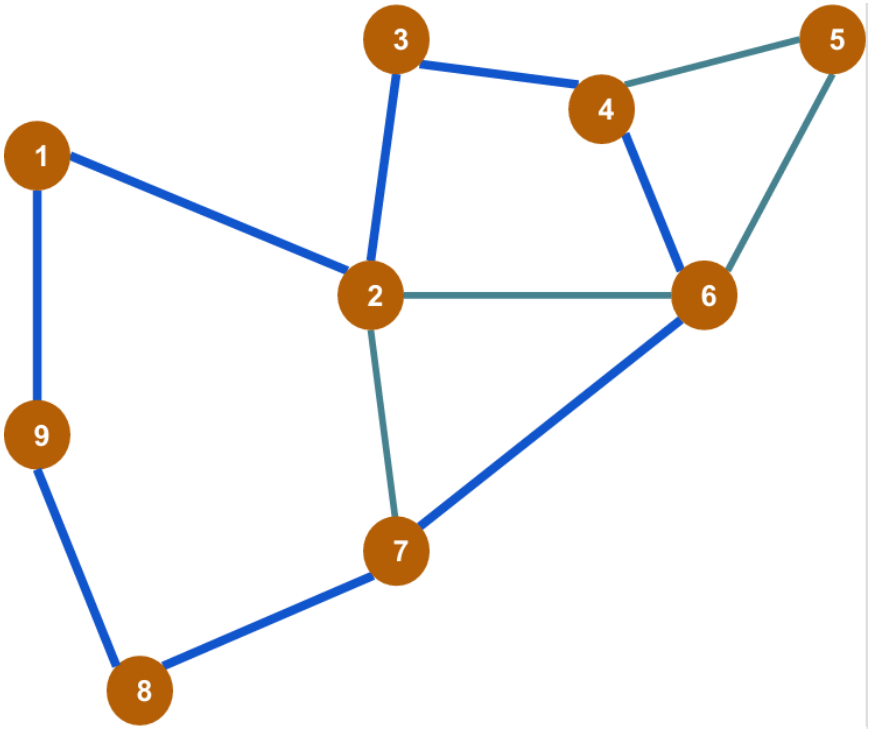


Good to Know:

- Un circuito es un tipo de recorrido.
- Un circuito es opuesto a un camino. Porque un camino no tiene repetidos.

Ciclo (o circuito simple)

Un ciclo o circuito simple es un circuito (de tres o más vértices) que no repite vértices, se nota $P_{v_i v_j}$. Ej.: $P_{1,1} = 1 \to 2 \to 3 \to 4 \to 6 \to 7 \to 8 \to 9 \to 1$

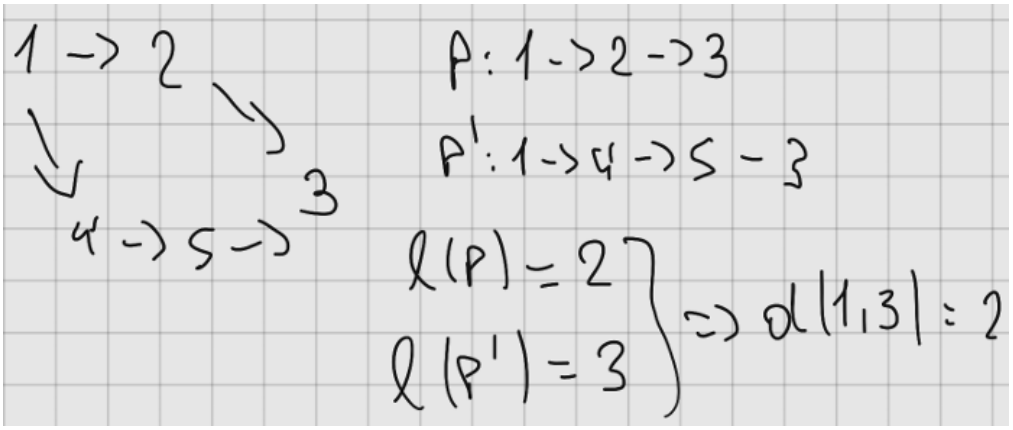


Longitud

La longitud de un recorrido P denotada l(P), es la **cantidad de aristas** que tiene.
Ej. : $P = 1 \to 2 \to 3 \to 4 \to 6 \to 2 \to 7$ donde $l(P) = 6$ **Good to Know**: Esta definición es muy importante, porque tranquilamente podría haber dos recorridos diferentes entre cada par de vértices.

Distancia entre dos vértices

La distancia entre dos vértices u y v se define como la **longitud del recorrido más corto** entre u y v. Se denota $d(u, v)$



Good to Know

- Precondiciones
  - Tener dos vectores: u y v que pertenezcan a un grafo G.
  - Tener uno o más recorridos, que incluyan los nodos de G.
- Postcondiciones
  - d(u,v) es el recorrido más corto desde u a v.
  - Si $u = v \iff d(u, v) = 0$
  - Si no existe un recorrido entre u y v entonces $d(u, v) = \infty$

Véase **anexo** para ver como demostrarlo por absurdo.

Camino en un grafo dirigido

Importa quién se relaciona con quien. Ej.: $E = \{(A, B)_{\to}, (B, C)_{\to}\}$

Camino en un Grafo no Dirigido

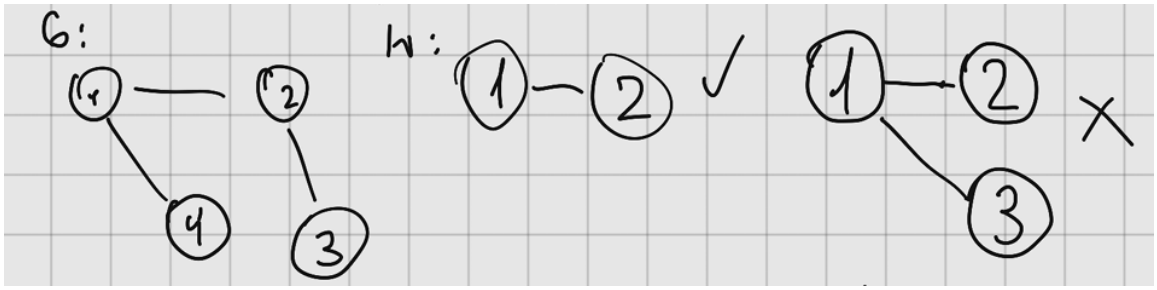
No importa quién se relaciona con quien. Ej.: $E = \{(A, B), (B, C)\}$. El camino podría ser en cualquier sentido.

Subgrafo

Sea $G = (V_G, E_G)$ y $H = (V_H, E_H)$

Subgrafo (base)

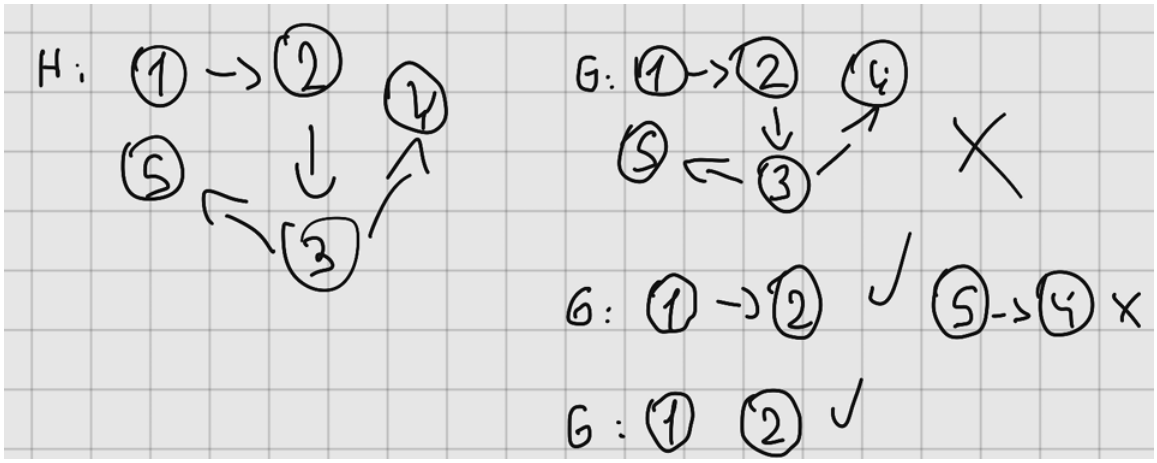
H es un subgrafo de G si todos los vértices y aristas de H están en G.
Esto quiere decir que con que no haya aristas (conexiones entre vértices) o vértices que no estén en G, H es un subgrafo de G.



A partir de acá, los demás subgrafos cumplen mínimamente estas condiciones

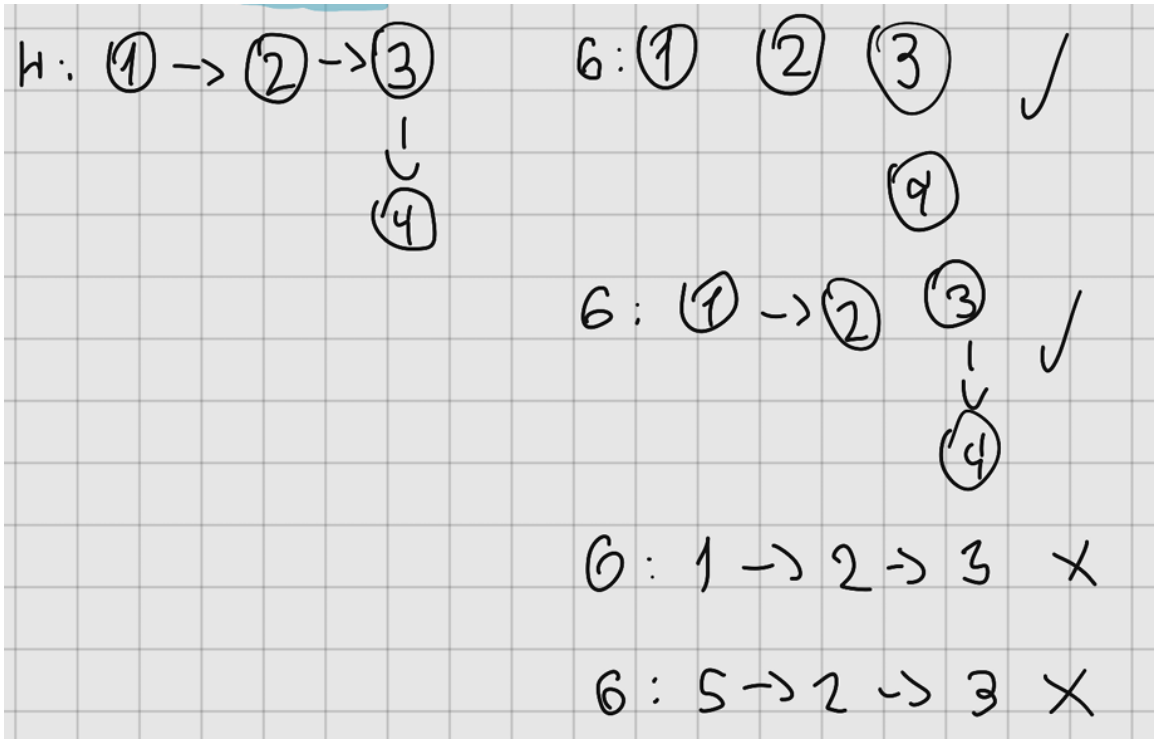
Subgrafo propio

H es un subgrafo propio de G si todos los vértices y aristas de H están en G siempre y cuando **no sea la totalidad**. Es decir, no debe suceder que sea exactamente $H = G$.



Subgrafo generador

H es un subgrafo generador de G si H **tiene exactamente los mismos vértices** que G.



Un subgrafo generador es más fuerte que un subgrafo propio porque nos hacen usar todos los nodos pero conectarlos como queramos (siempre y cuando esas conexiones existan en el grafo original). Un subgrafo propio solo tiene una condición, no ser igual al grafo original.

Subgrafo inducido

H es un subgrafo inducido de G si todos los nodos que están en H tienen todas las aristas de G que contienen exactamente a los nodos de H. Ej.: $G = ((1, 2, 3), \{(1, 3), (1, 2)\})$ si $H = (1, 2)$ entonces está obligado a tener las aristas que contengan exactamente a los elementos 1 y 2. Por lo tanto serían: $\{1, 2\}$

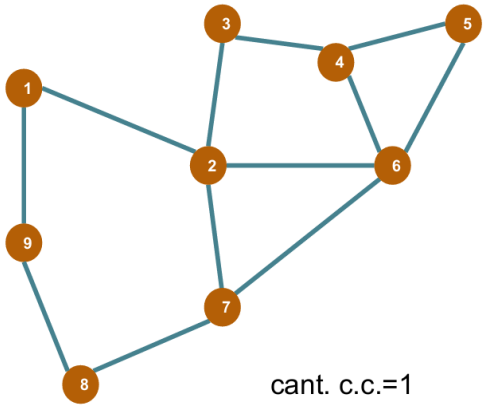
Grafo conexo

Un Grafo Conexo es un grafo el cual existe un **camino** que conecta todos los vértices entre sí.

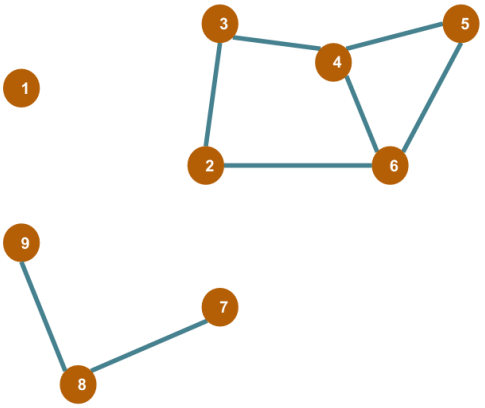
- $E = \{(A, B)_{\rightarrow}, (B, C)_{\rightarrow}, \}$ es un grafo conexo.
- $E = \{(A, B)_{\rightarrow}, (C, D)_{\rightarrow}\}$ no es un grafo conexo.

Componente conexa

Una componente conexa es un subgrafo.



cant. c.c.=1



cant. c.c.=3

Normalmente se consiguen cuando desconectás alguna arista de un grafo conexo. Entonces, te quedan subgrafos. La cantidad de subgrafitos son los componentes conexos.

Grafo fuertemente conexo

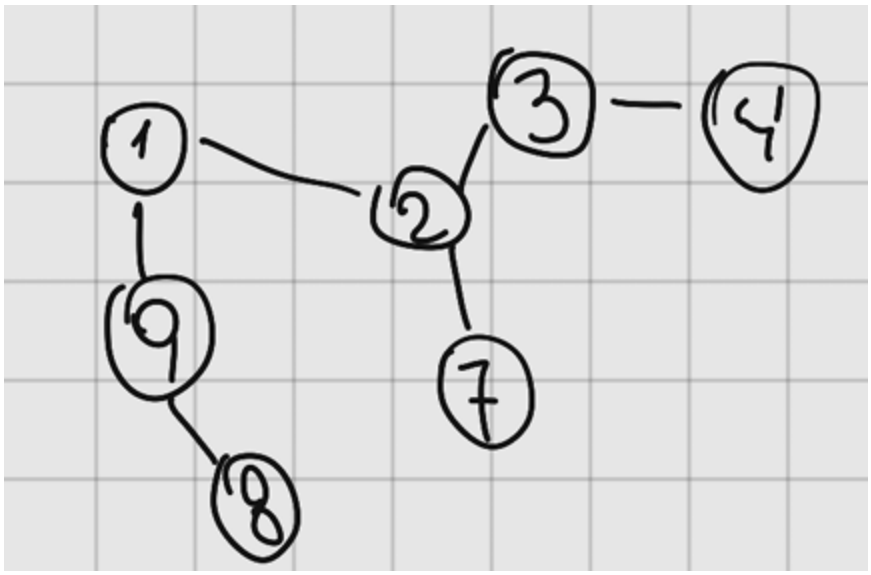
Un Grafo fuertemente conexo es un grafo dirigido y conexo. Todos los nodos deben tener una relación simétrica entre ellos.

- $E = \{(A, B)_{\rightarrow}, (B, C)_{\rightarrow}\}$ es un grafo conexo.
- $E = \{(A, B)_{\rightarrow}, (B, C)_{\rightarrow}, (C, B)_{\rightarrow}, (B, A)_{\rightarrow}, \}$ es un grafo fuertemente conexo.

Árboles

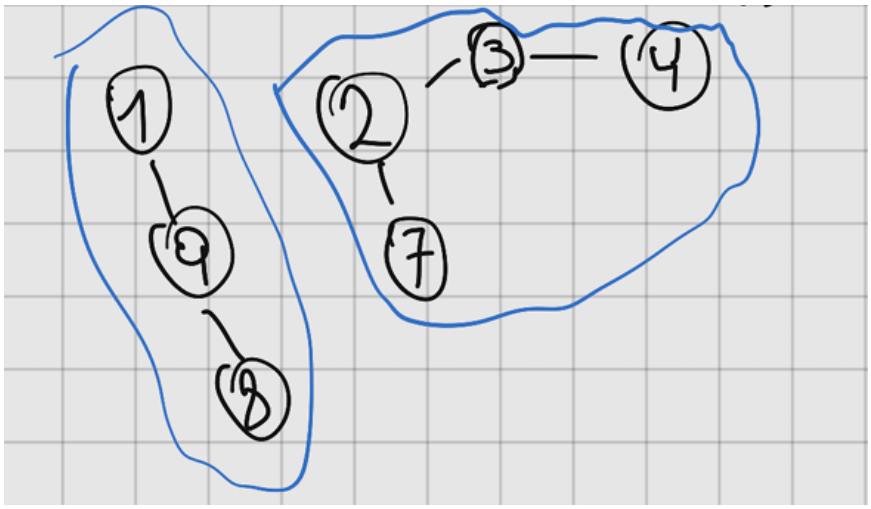
Base

Es un grafo conectado, sin ciclos (circuitos simples).



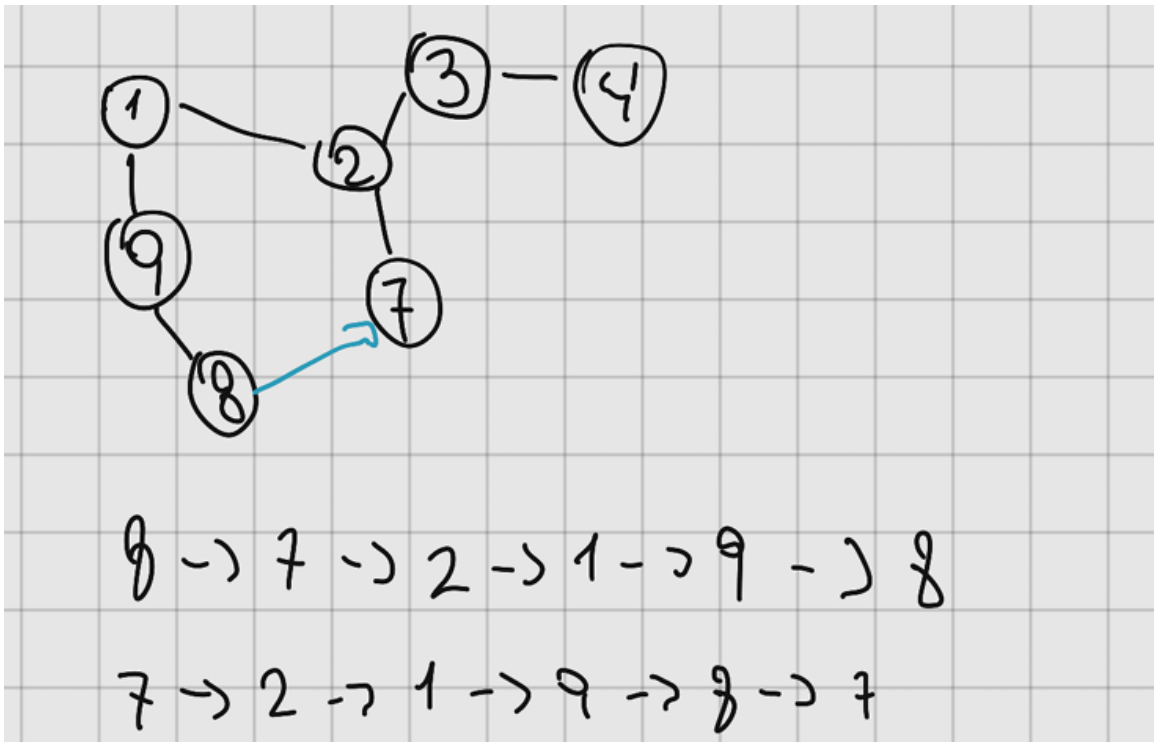
Quitar Arista

Si quitamos una arista cualquiera de un árbol, tenemos dos subgrafos conexos. Esto porque inicialmente un árbol no tiene ninguna componente suelta en el aire.



Agregar Arista

Si agrego una arista cualquiera se forma un ciclo (circuito simple).



Grafo T

Es un árbol que cumple con las siguientes propiedades

- Es conexo.
- No tiene ciclos.

Grafo Pesado

Definimos un Grafo Pesado G como $G = (V, E, W)$ donde W es una función que recibe dos vértices y devuelve el peso. Ahora, en cada arista almacenamos ambos vértices y el resultado del peso.

$$E_i = (v_i, v_j, w(v_i, v_j))$$

Good to Know: En inglés, decimos **weighted graphs** mientras que en español también los conocemos por ser **grafos ponderados**.

Camino en Grafo Pesado

En grafos pesados, el camino más corto entre dos nodos u y v es el que tiene la menor suma de pesos en sus aristas, sin importar cuántas aristas lo componen.

Ciclos Negativos en Grafos Pesados

Un grafo pesado G tiene un ciclo negativo si la **suma de los pesos de las aristas** es negativa.

```
A -> B (peso 2)
B -> C (peso -4)
C -> A (peso 1)

Suma total del ciclo: 2 + (-4) + 1 = -1
```

Entonces, $A \to B \to C \to A$ es un ciclo negativo.

Grafo Acíclico Dirigido (DAG)

Un Grafo Acíclico Dirigido es un grafo dirigido que no tiene ciclos. Es decir, no existe ningún vértice que es capaz de empezar y terminar un recorrido.
Ej.: Una Single Linked List es un caso particular de un DAG.

Orden Topológico

- Es una lista de nodos ordenados de forma que, si existe una arista $u \to v$, entonces u aparece antes que v en la lista.
- Es útil para resolver problemas donde ciertas tareas deben ejecutarse antes que otras, como en compiladores, donde un archivo puede depender de otros.
- Este orden no es único. ¿Por qué? Porque si hay nodos que no dependen de nadie, su posición relativa puede variar sin romper la condición del orden.
- **Ejemplo**: Dado el grafo con aristas $A \to C$ y $B \to C$, hay dos órdenes topológicos posibles:

$$A, B, C \quad \text{o} \quad B, A, C$$

- Esto se debe a que tanto A como B no dependen de ningún otro nodo, por lo que pueden aparecer en cualquier orden. El nodo C depende de ambos, así que debe aparecer después de A y B.

Véase **Topological Sort** para un caso de uso real con DFS y ejemplos.

Grafos Bipartito

Dado un grafo $G = (V, E)$, G es bipartito si

- Existe V_1 y V_2 tal que $V_1 \cup V_2 = V$
- $V_1 \cap V_2 = \emptyset$
- $e = (u, v) \in E, u \in V_1, v \in V_2$. Esto significa que el vértice de V_1 se relaciona con el de V_2 o en el otro sentido.
- La única restricción es que no podés conectar un vértice de V_1 con otro de V_1 ni tampoco uno de V_2 con otro de V_2

Grafo Bipartito Completo

Lo mismo que un Grafo Bipartito pero acá todos los vértices de V_1 están conectados con los de V_2 y viceversa. Básicamente, el producto cartesiano.

Cosas a releer / preguntar

- Un grafo es bipartito, $V = (V_1, V_2) \iff$ no tiene ciclos de longitud impar.
- Un grafo es bipartito $\iff$ todas sus c.c. lo son
- Un grafo no tiene ciclos impares $\iff$ cada una de sus c.c. no tienen ciclos impares.

Isomorfismo de Grafos

Dados $G = (V, E)$ y $G' = (V', E')$. G y G' son isomorfos si existe una función biyectiva $f : V \to V$ tal que $\forall u, v \in V$

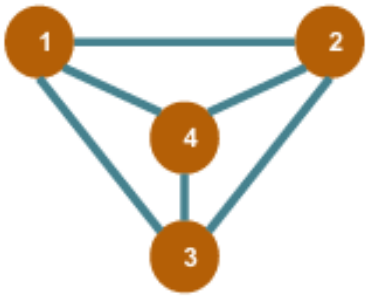
$$(u, v) \in E \iff (f(u), f(v)) \in E'$$

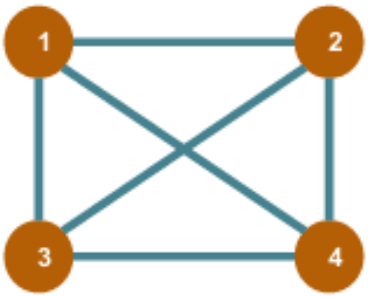
Es decir, **si parto desde E y agarro el par que está en una arista, puedo poner cada componente en una función f y me lo transforma en una arista que está en E'**.
Lo mismo con el caso opuesto.
Good to Know: En nuestro caso, si G y G'son isomorfos los vamos a notar como $G = G'$ porque son lo mismo pero cambian la forma en que se nombran los nodos y el orden de las aristas.

Propiedades

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

- Tienen el mismo número de vértices (sí, porque en la biyectividad se cumple la inyectividad y la sobreyectividad).
- Tienen el mismo número de aristas (sí, porque cada una debería traducirse a una única específica, de lo contrario no sería inyectiva).
- Para todo k, $1 \leq k \leq n - 1$, tienen el mismo número de vértices de grado k (la misma distribución de grado).
- Tienen la misma cantidad de componentes conexas.
- Para todo k, $1 \leq k \leq n - 1$, tienen el mismo número de caminos simples de longitud k.





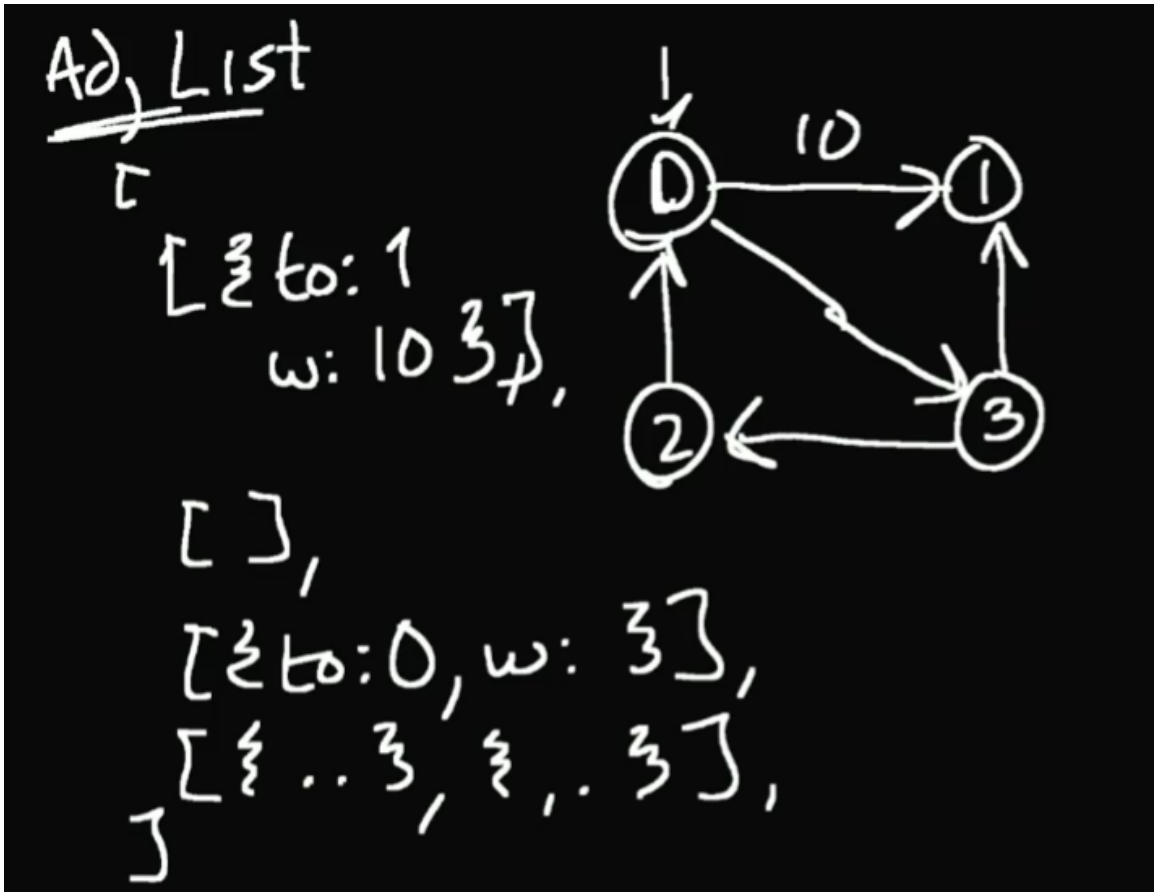
Representación de grafos

Good to Know: Si busca uso de grafos en la práctica, y cómo representarlos véase **anexo**. **Good to Know**: En las diapositivas se usa la d para representar tanto como el grado de un vértice o la distancia entre dos vértices. Acá, optamos por usar deg para el grado y d para distancia entre dos vértices.

Lista de Adyacencia

Es la más común y la más eficiente.
Es una lista de listas, donde cada lista corresponde al vértice y las aristas con las que se conecta se representa con una lista

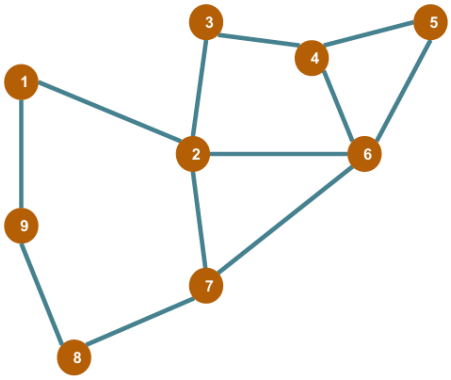
de diccionarios.



Matriz de Adyacencia

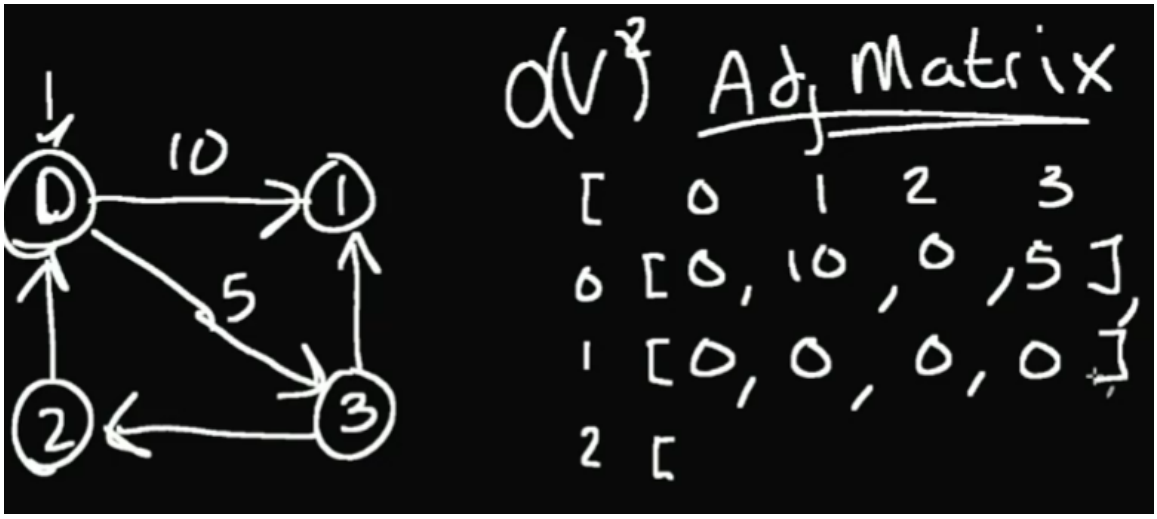
Cada número está representado por el índice en donde está. Vale 1 **(o el valor de peso de la arista)** si existe la arista y 0 si no.
La cantidad de filas y columnas es igual a la cantidad de nodos que hay, es por eso que espacialmente cuesta $O(n^2)$

Matriz de adyacencia (a_{ij})



	1	2	3	4	5	6	7	8	9
1	0	1	0	0	0	0	0	0	1
2	1	0	1	0	0	1	1	0	0
3	0	1	0	1	0	0	0	0	0
4	0	0	1	0	1	1	0	0	0
5	0	0	0	1	0	1	0	0	0
6	0	1	0	1	1	0	1	0	0
7	0	1	0	0	0	1	0	1	0
8	0	0	0	0	0	0	1	0	1
9	1	0	0	0	0	0	0	1	0

Ej.: El nodo 3 (fila 3), está relacionado con todos los nodos que están en la columna 3. En el dibujo, el nodo 3 está relacionado con el 2 y con el 4. Por eso, $a_{3,2} = 1$ y $a_{3,4}$.



Good to Know

- En el dibujo vemos que empieza todo desde el índice 1 pero en realidad empezamos desde el 0.
- **Lo que todavía no entiendo es, qué pasaría si los nodos guardan información compleja, como elijo cual va en el índice 0, cual en el 1, etc.**
- La matriz es simétrica si el grafo es no dirigido. Es decir, ver desde fila/columna o columna/fila es lo mismo.

Esta forma de trabajar con grafos es **súper ineficiente**.
¿Esto significa que nunca usaremos matrices? ¡No! Véase **anexo** para ver casos de uso.

Algunas propiedades útiles (solo grafos no dirigidos)

- 1. La suma de los elementos de la fila i (o columna j) es igual a $deg(u_i)$, es decir: $\sum_{i=0}^{j} a[i][0] = \sum_{j=0}^{i} a[j][0]$
- 2. Los elementos de la diagonal principal de A^2 indican los grados de los vértices $a^2_{ii} = deg(u_i)$. **Nota**: usa ii porque hay misma cantidad de filas que de columnas, de lo contrario, se podría decir ij con $i = j$. Esto es más que nada para hablar de los índices de la diagonal principal de una matriz.

1. Nótese que en la columna $a_{i,ultColumna+1}$ o en la fila $a_{ultFila+1,j}$ se encuentran las sumas de esa fila o columna.

Proposición 4:

Si A es la matriz de adyacencia del grafo G, entonces

1. la suma de los elementos de la fila *i* (o columna *j*) es igual a $d(u_i)$
2. los elementos de la diagonal de A^2 indican los grados de los vértices $a^2_{ii} = d(u_i)$

$\Sigma = \Sigma$

	1	2	3	4	5	6	7	8	9	
1	0	1	0	0	0	0	0	0	1	**2**
2	1	0	1	0	0	1	1	0	0	**4**
3	0	1	0	1	0	0	0	0	0	**2**
4	0	0	1	0	1	1	0	0	0	**3**
5	0	0	0	1	0	1	0	0	0	**2**
6	0	1	0	1	1	0	1	0	0	**4**
7	0	1	0	0	0	1	0	1	0	**3**
8	0	0	0	0	0	0	1	0	1	**2**
9	1	0	0	0	0	0	0	1	0	**2**
	2	**4**	**2**	**3**	**2**	**4**	**3**	**2**	**2**	

2. Nótese que ahora lo que estaban en la fila / columna extra, ahora se pasaron al a_{ii} (o a_{ij} con $i = j$) correspondiente.

	1	2	3	4	5	6	7	8	9
1	2	0	0	0	0	0	0	0	0
2	0	4	0	0	0	0	0	0	0
3	0	0	2	0	0	0	0	0	0
4	0	0	0	3	0	0	0	0	0
5	0	0	0	0	2	0	0	0	0
6	0	0	0	0	0	4	0	0	0
7	0	0	0	0	0	0	3	0	0
8	0	0	0	0	0	0	0	2	0
9	0	0	0	0	0	0	0	0	2

Matriz de Adyacencia con Grafos Dirigidos

- La matriz de adyacencia de un grafo dirigido NO es simétrica como sí pasaba con un grafo común.
- G = (V, E) donde E es un conjunto de pares ordenados. Esto significa que el **orden de los elementos del par** importan.
- $e = (u, v)$ también se llama **arco**. A u se le llama **cola** y a v se le llama **cabeza**. (medio raro, pero bueno).
- Los digrafos tienen grados de entrada (deg_{in}) y grados de salida (deg_{out})
- El grafo subyacente (G^s) es el que resulta de remover las direcciones.
- La suma de los elementos de la fila i es igual a $d_{OUT(u_i)}$.
- La suma de los elementos de la columna j es igual a $d_{IN(u_j)}$

Algoritmos de Búsqueda en Grafos

Recorrer grafos nos da mucha información acerca de su estructura.

BFS (Técnica Greedy)

Breadth-First Search o mejor conocido como BFS es un algoritmo de búsqueda para grafos.
Casos de uso:

- Vale para cualquier tipo de grafo ($*^1$)
- **Sirve** para buscar el **camino mínimo** de un nodo a otro.
- Se usa para asegurarnos que siempre pasamos por todos los nodos del mismo nivel antes de pasar a otro.
- Explora el grafo o árbol nivel a nivel, primero horizontalmente y luego verticalmente. Es decir, primero agarra todos los vecinos, después del primer vecino baja al siguiente nivel, después del segundo vecino baja al siguiente nivel y así sucesivamente.
- A nivel estructura de datos se implementa con una **queue** (FIFO).
- Se almacena en una lista los nodos visitados para no repetirlos. Es decir, que antes de pasar por uno hay que chequear si ya está.

Good to Know ($*^1$): Si usamos BFS en un Grafo G pesado **NO NOS GARANTIZA** que sea el camino mínimo, porque BFS calcula el camino con menor cantidad de aristas, y no con menor peso.

Complejidad temporal de BFS

La complejidad temporal de BFS es $O(|V| + |E|)$.

DFS

Depth-First Search o mejor conocido como DFS es un algoritmo de búsqueda para grafos.
Casos de uso:

- Explora todos los nodos a profundidad, primero verticalmente y luego horizontalmente.
- A nivel estructura de datos se implementa con un **stack** (LIFO).
- **NO sirve** para buscar el camino más corto de un nodo al otro. Esto es debido a que ingresa a las ramas en profundidad.
- Vale para cualquier tipo de grafo ($*^2$)

Un ejemplo claro de DFS sería buscar la altura de un árbol porque tendrías que buscar la rama más larga.
Good to Know ($*^2$): Si usamos DFS en un Grafo G pesado **NO NOS GARANTIZA** que sea el camino mínimo, porque DFS calcula el camino con menor cantidad de aristas, y no con menor peso.

Complejidad temporal de DFS

La complejidad temporal de DFS es $O(|V| + |E|)$.
Good to Know: Pese a que se implementa recursivamente o iterativamente, la complejidad podría parecer que es $O(|V|*|E|)$ pero es $O(|V| + |E|)$ porque cada arista se visita una sola vez.

Algoritmos de Búsqueda de Camino Mínimo (Grafos Pesados)

- Single-Source-Shortest-Paths: Para encontrar el camino mínimo desde un nodo particular a todos los demás nodos. Resuelve **uno a muchos**.
  - Dijkstra, Bellman-Ford
- All-Pairs-Shortest-Paths: Para encontrar el camino mínimo desde todos los nodos a todos los demás nodos. Resuelve **muchos a muchos**.
  - Floyd-Warshall

Single-Source-Shortest-Paths

Bellman-Ford (Técnica DP)

Es un algoritmo que nos permite **dado un nodo encontrar el camino mínimo desde ese nodo hasta los demás**. Este es útil para detectar ciclos negativos en el grafo.
Su complejidad temporal es de $O(V * E)$ y su complejidad espacial es de $O(V + E)$

Requisitos de uso de Bellman-Ford

- **Grafo ponderado**
- Puede trabajar con grafos dirigidos o no dirigidos
- Puede trabajar con aristas de peso positivo o negativo.

¿Cuándo usar Bellman-Ford?

- Si el grafo es grande.
- **Si necesitás el camino más corto desde un nodo particular a todos los demás nodos**. Resuelve **uno a muchos**.
- Si tenés que detectar ciclos negativos en el grafo.
- Si tenés pesos negativos en las aristas.

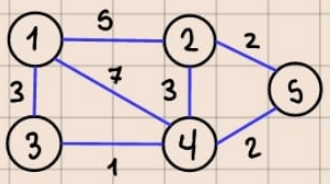
Orden de operaciones

- Se crea una matriz de longitud NxN donde se inicializa el primer nodo con valor 0 y el resto en ∞.
- Tomamos los vecinos del primer nodo, y guardamos el peso que hay entre el nodo y sus vecinos.
- Tomamos uno de los vecinos del nodo inicial y calculamos la distancia a todos sus nodos vecinos.
- Así sucesivamente, vamos a encontrar el camino más corto (con menos peso) de un nodo a todos los demás nodos.

BELLMAN-FORD: PERMITE HALLAR EL CAMINO MÁS CORTO ENTRE UN NODO Y TODOS LOS DEMÁS.

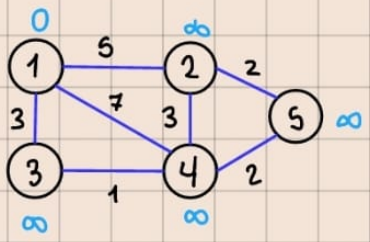
- ACEPTA SOLO GRAFOS PONDERADOS.
- USA DP.
- TRABAJA CON GRAFOS DIRIGIDOS Y NO DIRIGIDOS
- PERMITE HALLAR CICLOS NEGATIVOS.
- ACEPTA PESOS NEGATIVOS Y CICLOS NEGATIVOS.

INPUT EJ:

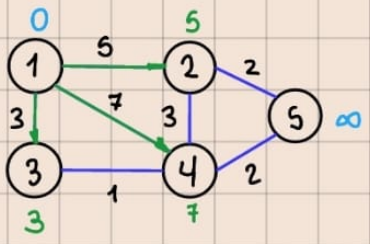


PASOS:

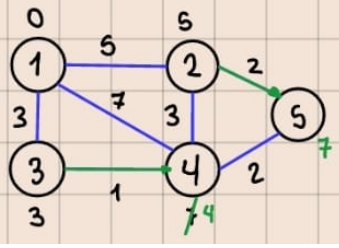
1. INICIALIZO EL NODO OBJETIVO CON DISTANCIA A SÍ MISMO COMO 0, ∞ CON LOS DEMÁS.

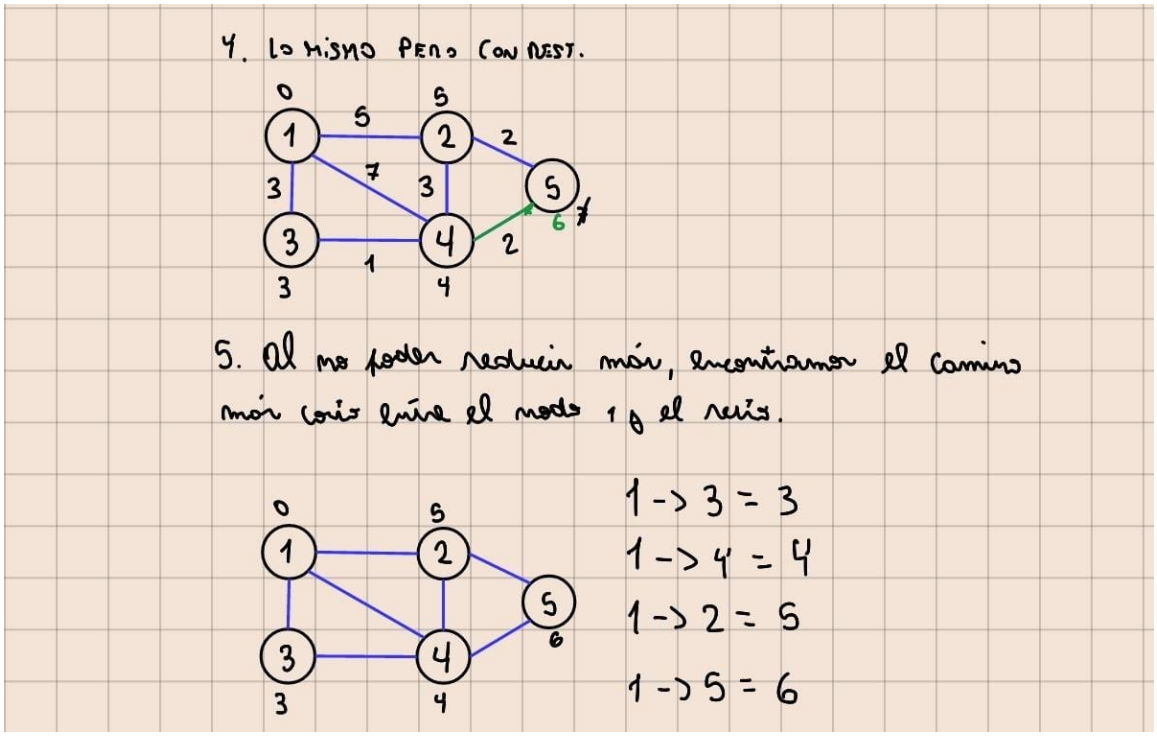


2. GUARDO LA DISTANCIA ENTRE MI NODO Y SUS VECINOS



3. GUARDO LA DISTANCIA ENTRE LOS VECINOS DE MI NODO.





Algoritmo

```
    for (int i = 1; i <= n; i++) distance[i] = INF;
    distance[x] = 0;
    for (int i = 1; i <= n-1; i++) {
        for (auto e : edges) {
            int a, b, w;
            tie(a, b, w) = e;
            distance[b] = min(distance[b], distance[a]+w);
        }
    }
```

All-Pairs-Shortest-Paths

Good to Know: Es posible conseguir el mismo resultado haciendo $|V|$ operaciones pero con algoritmos de Single-Source-Shortest-Paths. ¿Vale la pena?, depende. Véase **Tradeoff entre Algoritmos de Búsqueda de Camino Mínimo**

Floyd-Warshall (Técnica DP)

Es un algoritmo que nos permite **en una sola operación** encontrar el **camino mínimo entre todos los pares de nodos** en un grafo ponderado.
Su implementación algorítmica es muy simple aunque este método es muy costoso dado que su complejidad temporal es $O(n^3)$ y su complejidad espacial es $O(n^2)$ que es el costo de armar la matriz.
¿Por qué tal complejidad temporal? Bueno... Porque se implementa con 3 for anidados.
Se basa en el uso de Programación Dinámica de forma bottom-up.

Requisitos de uso de Floyd-Warshall

- **Grafo ponderado**
- Puede trabajar con grafos dirigidos o no dirigidos
- El grafo puede tener pesos negativos, pero no debe haber ciclos negativos

¿Cuándo usar Floyd-Warshall?

- Si necesitás todos los caminos más cortos entre **todos los pares de nodos**. Resuelve **muchos a muchos**.
- Si tenes pesos negativos pero no ciclos negativos (no es necesario tener pesos negativos)
- Puede trabajar con aristas de peso positivo o negativo.
- Cuando el grafo NO es demasiado grande.

Orden de operaciones

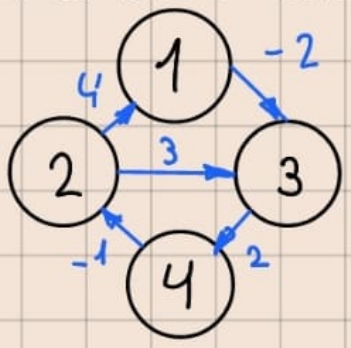
- Se arma una matriz, donde la distancia de todos los nodos a sí mismos es 0 (i = j =¿0)
- Por cada nodo, se buscan las aristas que los conectan con los demás nodos, y se coloca en esa relación (ij), el peso de la arista.
- Se crean 3 for anidados. k, i, j que comienzan desde $i = 1$ y van hasta la cantidad de nodos.
- En cada iteración, se chequea que si la distancia actual de (ij) es mayor que agregando un nodo intermedio, entonces, la nueva distancia más corta es la de agregando ese nodo intermedio.
    - Hay 2 casos que chequear al final.
        - Si existía distancia inicial entre dos nodos y el valor final cambió, significa que se encontró un camino más corto.
        - Si existía distancia inicial entre dos nodos y el valor final no cambió, significa que era el camino más corto.
        - Si no existía distancia inicial (∞) y el valor final cambió, significa que se encontró un camino entre esos nodos.
        - Si no existía distancia inicial (∞) y el valor final no cambió, significa que no se encontró un camino entre esos nodos.

RECORDATORIO: EN UN GRAFO PESADO, EL CAMINO MÁS CORTO ES EL QUE MENOS PESO TIENE.

FLOYD-WARSHALL: Encuentra el CAMINO MÍNIMO entre TODOS LOS PARES DE VÉRTICES

- ACEPTA SOLO GRAFOS PONDERADOS.
- USARLO EN GRAFOS PEQUEÑOS
- USA DP.
- ACEPTA PESOS NEGATIVOS. NO DEBE HABER CICLOS NEGATIVOS EN EL GRAFO.
  ↳ A->B = 2 B->C = 1 C->A = -4 ⇝ A->B->C->A = -1
- TRABAJA CON GRAFOS DIRIGIDOS Y NO DIRIGIDOS

INPUT EJ:



$G = (\{1,2,3,4\}, \{(1,3,-2), (3,4,2), (4,2,-1), (2,1,4), (2,3,3)\})$

PASOS:

1. Llena MATRIZ en [i][j] con i=j de 0, $\forall v \in V_G$

	1	2	3	4
1	0			
2		0		
3			0	
4				0

2. Coloca el PESO de la arista $\forall e:(u,v) \in E_G$.

	1	2	3	4
1	0		-2	
2	4	0	3	
3			0	2
4		-1		0

*

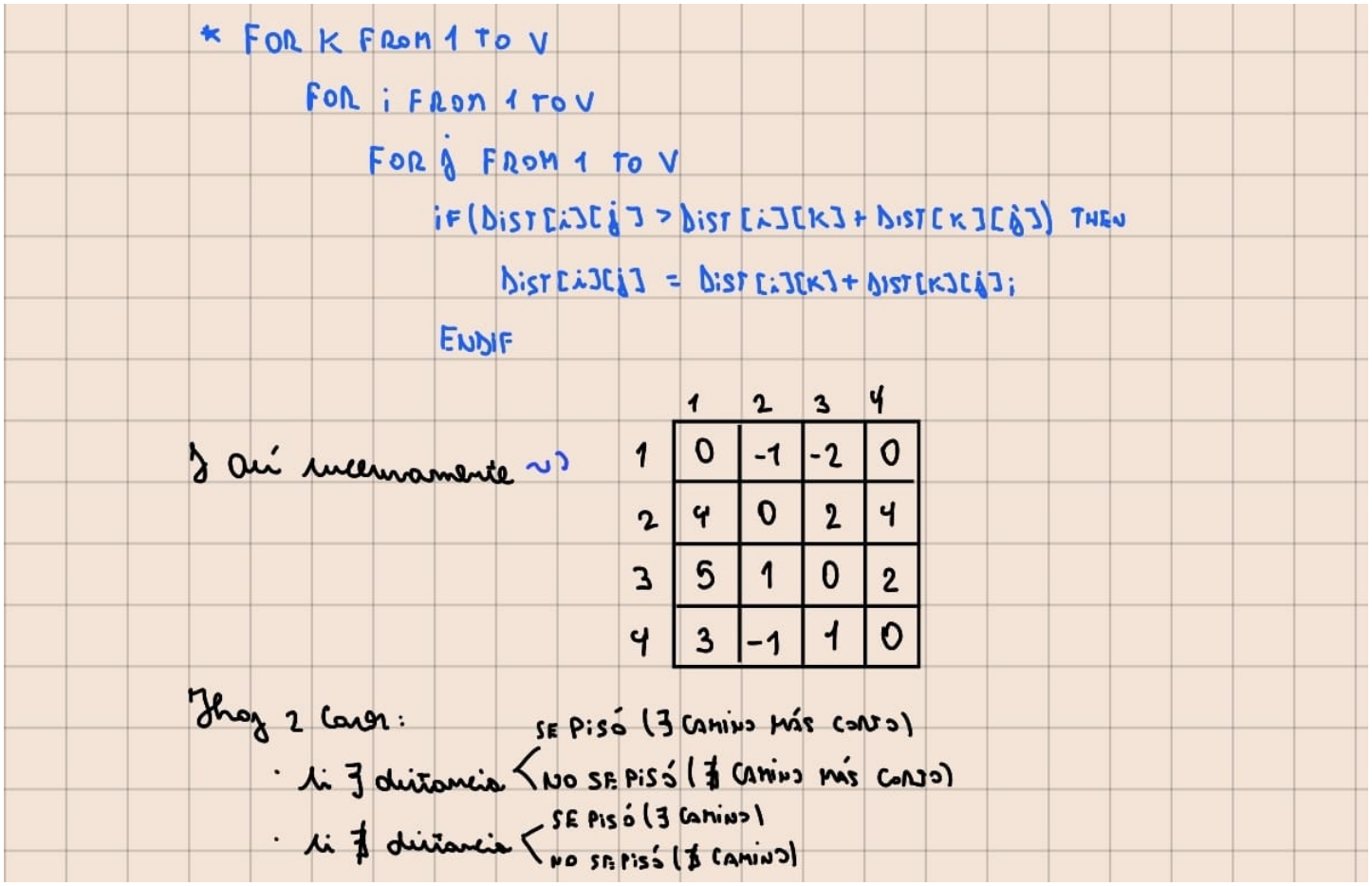
3. Buscar preguntando: "¿Existe un camino más corto de i a j pasando por K?" Si lo hay, actualiza la distancia.

Veamos paso a paso.

	1	2	3	4
1	0		-2	
2	4	0	2	
3			0	2
4		-1		0

- K = 1 2 3 4
  i = 1 2 3 4
  j = 1 2 3 4

- K=2, i=4, j=1
  G[i][j] = ∞
  G[i][k] = -1
  G[k][j] = 4
  G[i][k] + G[k][j] = 3 < ∞



Algoritmo

```
1   for (int i = 1; i <= n; i++) {
2       for (int j = 1; j <= n; j++) {
3           if (i == j) distance[i][j] = 0;
4               else if (adj[i][j]) distance[i][j] = adj[i][j];
5               else distance[i][j] = INF;
6           }
7   }
8
9   for (int k = 1; k <= n; k++) {
10      for (int i = 1; i <= n; i++) {
11          for (int j = 1; j <= n; j++) {
12              distance[i][j] = min(distance[i][j],
13              distance[i][k]+distance[k][j]);
14          }
15      }
16  }
```

Tradeoff entre Algoritmos de Búsqueda de Camino Mínimo

Es cierto que si tenemos un grafo pesado, podemos encontrar todas las relaciones de todos los nodos usando un método que sea **Single-Source Shortest Path**, resolviendo una vez por nodo, pero esto es muy ineficiente cuando tenemos un grafo grande.
No obstante, hay casos donde podría servir si no analizamos bien el problema.
Ej.: En **grafos dispersos**, sin **pesos negativos** es más óptimo usar implementaciones de Dijkstra (Min-Heap, Fibonacci Heap, etc.) $|V|$ veces antes que Floyd-Warshall.
Ej. 2: Tenemos un **grafo pesado acíclico** y aristas positivas. Queremos saber el **mínimo camino de un nodo a otros**. Si usamos Dijkstra la complejidad es $O(V^3)$ y Bellman-Ford es $O(V^2E)$. ¿Esto significa que Bellman-Ford es mejor? ¡No! Porque si el grafo fuese denso (no lo sabemos) entonces con Bellman-Ford la complejidad sería $O(V^4)$

Topological Sort

Topological Sort es un caso de uso real de DFS. Es un **ordenamiento lineal** de los nodos de un **grafo dirigido acíclico (DAG)** tal que, para cada arista $u \to v$, el nodo u aparece antes que el nodo v.

Un orden topológico nos sirve para asegurarnos de **qué cosas deben cumplirse antes de que podamos hacer otras**. Imaginemos que estamos creando un compilador, y tenemos muchos archivos que dependen de diversas librerías. Por ejemplo, una de las librerías es **Pow**, que depende de **Math**, y **Main** también necesita de **Math** para funcionar correctamente.
En este caso, tenemos dos dependencias claras:

- **Pow** no funciona sin **Math**, porque depende de funciones dentro de esa librería. Por lo tanto, primero tenemos que compilar **Math** antes de poder acceder a **Pow**.
- **Main** no funciona sin **Math**, porque necesita funciones de allí. Así que también tenemos que compilar **Math** antes de poder acceder a **Main**.

¿Qué tiene que ver esto con grafos? Bueno, podemos tratar a las entidades (**Pow**, **Math** y **Main**) como nodos, y las dependencias entre ellas como aristas del grafo. Es decir:

$$V = \{\text{Pow}, \text{Math}, \text{Main}\}, \quad E = \{(\text{Math}, \text{Main}), (\text{Math}, \text{Pow})\}$$

Ahora viene la parte interesante: el **ordenamiento topológico**. ¿Dónde está? La idea es que podemos cambiar el orden de los nodos siempre y cuando respetemos el orden de las dependencias. Es decir, **Main** no puede ser compilado antes de **Math**. Sin embargo, no importa si **Pow** o **Main** se compilan antes de lo otro, siempre y cuando **Math** esté compilado primero.
Por lo tanto, tenemos dos posibles representaciones topológicas del problema:

- Math $\to$ Pow $\to$ Main
- Math $\to$ Main $\to$ Pow

Ambas cumplen con la condición de que **Math** debe compilar antes que **Pow** y **Main**, pero no importa si **Pow** o **Main** se compilan antes que el otro.
¿Qué pasa si las dependencias cambian? Imaginemos el siguiente conjunto de dependencias:

$$V = \{\text{Pow}, \text{Math}, \text{Main}\}, \quad E = \{(\text{Math}, \text{Main}), (\text{Math}, \text{Pow}), (\text{Pow}, \text{Main})\}$$

En este caso, solo existe **un único orden topológico válido**:

- Math $\to$ Pow $\to$ Main

Esto es porque la arista $(\text{Pow}, \text{Main})$ obliga a que **Pow** se compile antes que **Main**, eliminando las opciones de reordenar **Pow** y **Main**.
¿Para qué me sirve esto? Para asegurarnos de que no hay **dependencias cíclicas** y que todas las dependencias se resuelven en el orden correcto, manteniendo la integridad de las relaciones entre las entidades.
Implementación con DFS:

```
1   topoSort(graph G)
2       for each vertex u in V[G] do
3           state[u] = UNVISITED
4           parent[u] = NULL
5       end for
6       time = 0
7       for each vertex u in V[G] do
8           if state[u] = UNVISITED then
9               TOP_VISIT(u)
10          end if
11      end for
12  end function
```

Good to Know: Si existen varios nodos que no tienen dependencias entre sí, hay más de una forma válida de realizar el orden topológico. En estos casos, el orden en que aparecen esos nodos no tiene importancia, y puedes elegir su disposición según tus preferencias al implementar el algoritmo. El orden topológico solo se asegura de que los nodos con dependencias se ubiquen en el orden correcto, pero no especifica el orden de los nodos independientes.
La complejidad es la misma que DFS, $\theta(|V| + |E|)$

Fuerza Bruta

Consiste en analizar y listar todas las posibilidades y quedarme con las que me interesan.
En la fuerza bruta, existen problemas de factibilidad y optimalidad.

- En los problemas de factibilidad lo que queremos son soluciones que cumplan ciertas características.
- En los problemas de optimizaciones combinatorias, teniendo todas las soluciones factibles me quedo con la mejor (según el criterio que considero qué es mejor).

Los algoritmos por fuerza bruta son muy fáciles de implementar pero son muy ineficientes; son útiles cuando tenemos instancias súmamente pequeñas.
Ej.: Si tenemos que completar un tablero de ajedrez tendríamos que poner todas las posibilidades en todos los casilleros y si hay pocas soluciones válidas estaríamos tardando demasiado tiempo por nada.

Backtracking

Es una modificación de la técnica de fuerza bruta que aprovecha propiedades del problema para evitar analizar todas las configuraciones.
Para que el algoritmo sea correcto debemos estar seguros de no dejar de examinar configuraciones que estamos buscando.
Las soluciones candidatas se representan con un **vector a** $= (a_1, a_2, \ldots, a_n)$

- Soluciones parciales: Son las soluciones que se van armando en cada paso.
  - Cada nodo es una solución parcial.
  - La raíz del árbol es el vector vacío (solución parcial vacía).
- Soluciones candidatas: Depende el problema. A veces son todas las hojas (si necesitás gastar todos los elementos), a veces te basta con cumplir X cosa y no necesitás recorrer todo el árbol.
- Soluciones válidas: Son aquellas soluciones que cumplen con todas las características que buscamos. A veces son las hojas del árbol, a veces no es necesario llegar hasta las hojas, porque capaz se cumple en uno, o dos pasos tempranos.

¿Todas las soluciones candidatas son soluciones parciales? Porque si una posible solución válida llegás en el primer nivel del árbol, entonces esa solución era candidata, pero también era parcial.
Ej.: Ejercicio de armar subconjuntos que sumen n.

Podas

En el backtracking, recorremos el árbol en profundidad.
Cuando podemos ver que una solución parcial no nos llevará a una solución válida, no es necesario seguir explorando esa rama del árbol de búsqueda (se poda el árbol) y se retrocede hasta encontrar un vértice con un hijo válida por donde seguir.

- Poda por factibilidad: Ninguna extensión de la solución parcial derivará en una solución valida del problema.
  - Digamos que quiero llegar a armar subconjuntos que sumen 9, si con el nodo actual la suma es 10, entonces no me sirve seguir hacia adelante. Podo de una. Esto se llama **propiedad dominó**, si una solución parcial no vale, la generada a partir de ella menos todavía.
  - **¿Qué pasaría, si llegase a haber algún caso donde capaz nuestra solución parcial ya no es válida, pero si seguíamos adelante capaz teníamos un factor que la volvía a hacer válida? Por ej.: En un conjunto que tenes números positivos y negativos tenés que armar subconjuntos que sumen k. Si te pasaste de k, no te sirve podar de una porque capaz encontrás algun negativo que te ayude a llegar a k de vuelta.**. Respuesta: No te sirve esa poda para ese problema.
- Poda por optimalidad (problemas de optimización): Ninguna extensión de la solución parcial derivará en una solución del problema óptima.

Un problema de backtracking que no tiene podas es un problema de fuerza bruta.

Correctitud en Backtracking

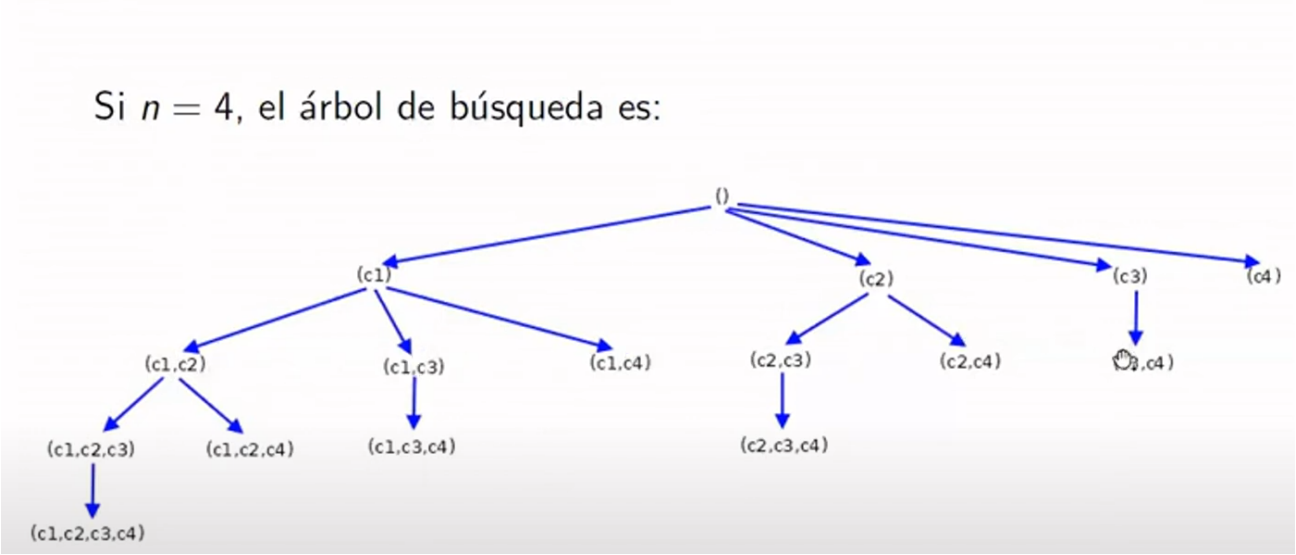
Para demostrar la correctitud de un algoritmo de backtracking, debemos demostrar que se enumeran todas las configuraciones válidas. Si hacés una poda, me tenés que garantizar que aunque el rendimiento haya mejorado, me sigas trayendo las mismas soluciones válidas.
Ej.: Si lo hacemos por fuerza bruta, podemos conseguir todas las soluciones válidas. Luego, si lo hacemos por backtracking y añadimos podas, nos fijamos que tengamos los mismos resultados que si lo hacemos por fuerza bruta.

Complejidad en Backtracking

Consideramos el costo de procesar cada nodo. Cada nodo aparece cada vez que hacemos un llamado recursivo.

Tips Backtracking

- Si tengo conjuntos, solo me interesan tener una única ocurrencia. Es decir, $\{1, 3\} = \{3, 1\}$ por lo tanto, siempre priorizo que el elemento de la izquierda pueda tener a todos los mayores de la derecha pero no al revés. Esto gana mucho tiempo porque estamos eliminando soluciones repetidas.



Programación Dinámica (DP)

Se basa en solucionar y memorizar subproblemas que se repetirán para ahorrar tiempo.
Esa **superposición de problemas** es que estamos llamando usando algo que **ya resolvimos anteriormente**, entonces, para evitar calcular todo nuevamente, usamos la información almacenada en memoria.
Es **bastante común** usar este tipo de estrategias para afrontar problemas con matrices, o casos recursivos para cosas altamente costosas de calcular.
Importante: ¿Cuándo hay veces que la entrada no es exactamente igual pero vale la memorización para ese subproblema? Cuando la dirección de los argumentos no importa, se puede utilizar ese subproblema memorizado, es decir, en casos donde, por ejemplo $(a, b) = (b, a)$.

Propiedades de la Programación Dinámica

Existen varios conceptos importantes que debemos tener en cuenta para entender el uso de programación dinámica:

- Estado: Representa una subparte del problema original. Se define por las variables mínimas necesarias para describir un subproblema.
- Caso base: Es la condición inicial del problema. Define cuándo detener la recursión y que valor devolver en ese caso.
- Caso recursivo: Es la definición del problema en función de subproblemas más pequeños.
- Transición: Es la regla que te lleva de un estado a otro, es decir, cómo se construye una solución a partir de otras. Qué hacés con el llamado recursivo.
- Orden Topológico: Es el orden obligatorio con que se realizan las operaciones.

Veamos un ejemplo de un código real para categorizar los ítems anteriormente mencionados

```
1  long long fibo(int n, std::vector<long long>& memo){ //Estado: n, es lo que importa para memorizar.
2      if(memo[n] != 0) return memo[n];
3
4      if(n == 0) return 0; // Caso base
5      if(n == 1) return 1; // Caso base
6
7      auto state1 = fibo(n-1, memo); //Caso recursivo
8      auto state2 = fibo(n-2, memo); //Caso recursivo
9      memo[n] = state1 + state2;   //Transición
```

```
    return memo[n];

    Orden topológico: calcular fibo(0), luego fibo(1), luego fibo(2), luego fibo(3), ... etc.

    DAG: Una cadena de dependencias 0 -> 1 -> 2 ... -> n (como resuelve en base al param n)

    }
```

Veamos unos ejemplos más

```
    vector<long long> fact(n+1, 1);
    vector<long long> sum(n+1, 0);
    for(int i=1; i<=n; ++i){
        fact[i] = i * fact[i-1];
        sum[i] = sum[i-1] + fact[i];
    }
    return sum[n];

    Transición:
        fact[i] = i * fact[i-1];
        sum[i] = sum[i-1] + fact[i];
    (como sum[n] es la respuesta, y sum[i] depende de sum y fact, eso cambia el estado).

        Orden topológico: primero calculo fact[0..n], y luego sum[0..n].

    DAG: fact[0] -> sum[0] -> fact[1] -> sum[1] -> fact[2] -> sum[2] ... -> fact[n] -> sum[n]
```

Formas de realizar programación dinámica

Se puede hacer tanto de forma Top-Down (recursivamente) o Bottom-Up (iterativamente).
En Top-Down hablamos de **memorización** mientras que en Bottom-Up hablamos de **tabulación**.

Bottom-Up vs Top-Down

- Top Down (memorización): Usamos recursión, guardamos los resultados en una caché (generalmente diccionario o array). Solo resolvemos los subproblemas que necesitamos.
- Bottom Up (tabulación): Usamos un enfoque iterativo, contruimos la solución desde los casos base hacia arriba mientras que llenamos una tabla paso a paso.

Memoization (Top-down)

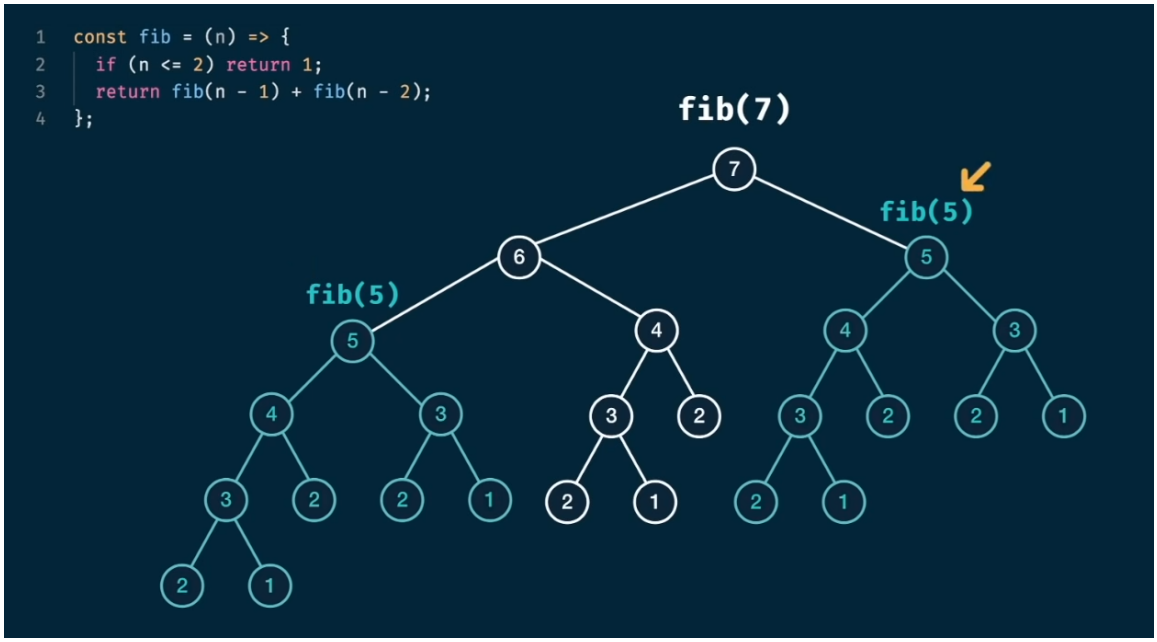
```js
function fib(n, memo = {}) {
  if (n <= 1) return n;
  if (memo[n]) return memo[n];
  memo[n] = fib(n - 1, memo) + fib(n - 2, memo);
  return memo[n];
}
```

Tabulation (Bottom-up)

```js
function fib(n) {
  if (n <= 1) return n;
  const dp = [0, 1];
  for (let i = 2; i <= n; i++) {
    dp[i] = dp[i - 1] + dp[i - 2];
  }
  return dp[n];
}
```

Programación Dinámica Top-Down

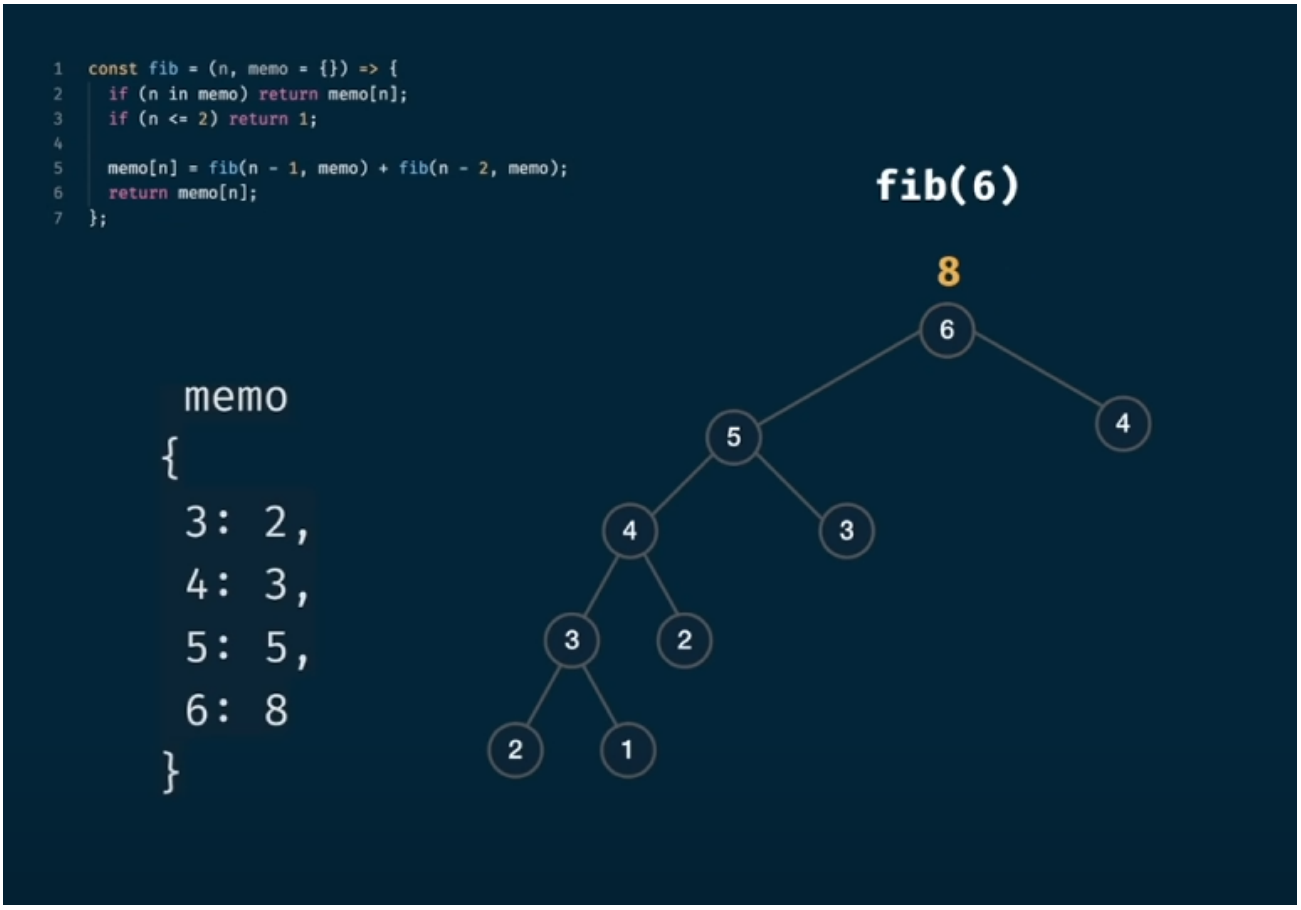
Fibonacci y su relación con la Programación Dinámica



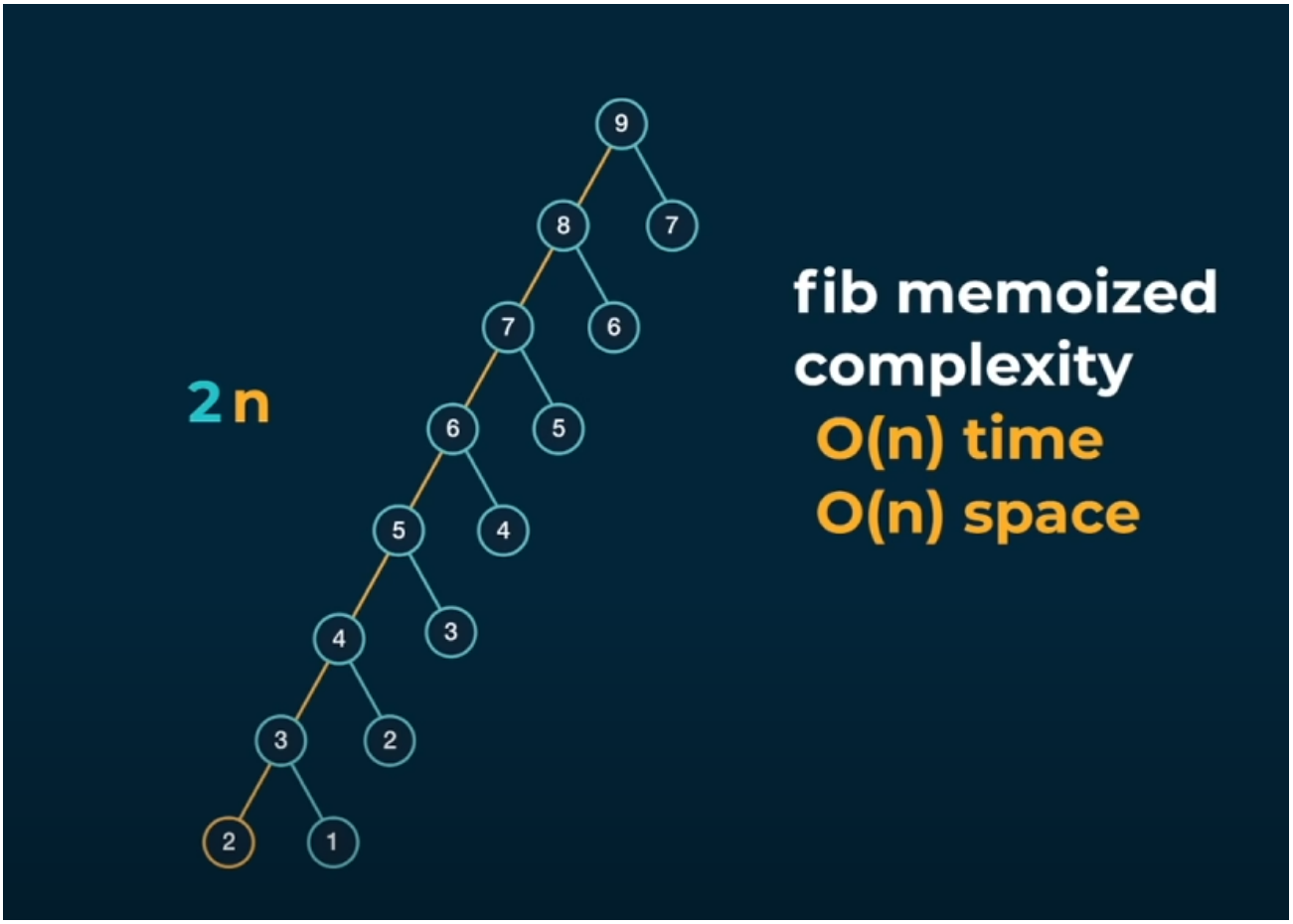
¿Donde está la superposición de problemas acá? Bueno, veamos que para **fib 5** se encuentra repetida una vez.
Entonces, la idea, es que guardemos el resultado para el primer **fib** que salga, y luego reutilizarla.
Nótese que al ser **fib(n-1) + fib(n-2)** en algún momento, indudablemente el **n-1** va a llegar a ser el valor que fue **n-2**, y ahí justamente, entra la memorización.
A nivel código, no es más que guardar en un array, map o una estructura eficiente.
Sin ningún tipo de memorización, la complejidad que da en tiempo es $O(n^2)$ mientras que en espacio da $O(n)$.
En espacio da $O(n)$ porque a nivel stack, lo más profundo que baja en una rama es n veces.

Con Programación Dinámica

Sabiendo que tenemos superposición de problemas basado en n, optimizado quedaría así

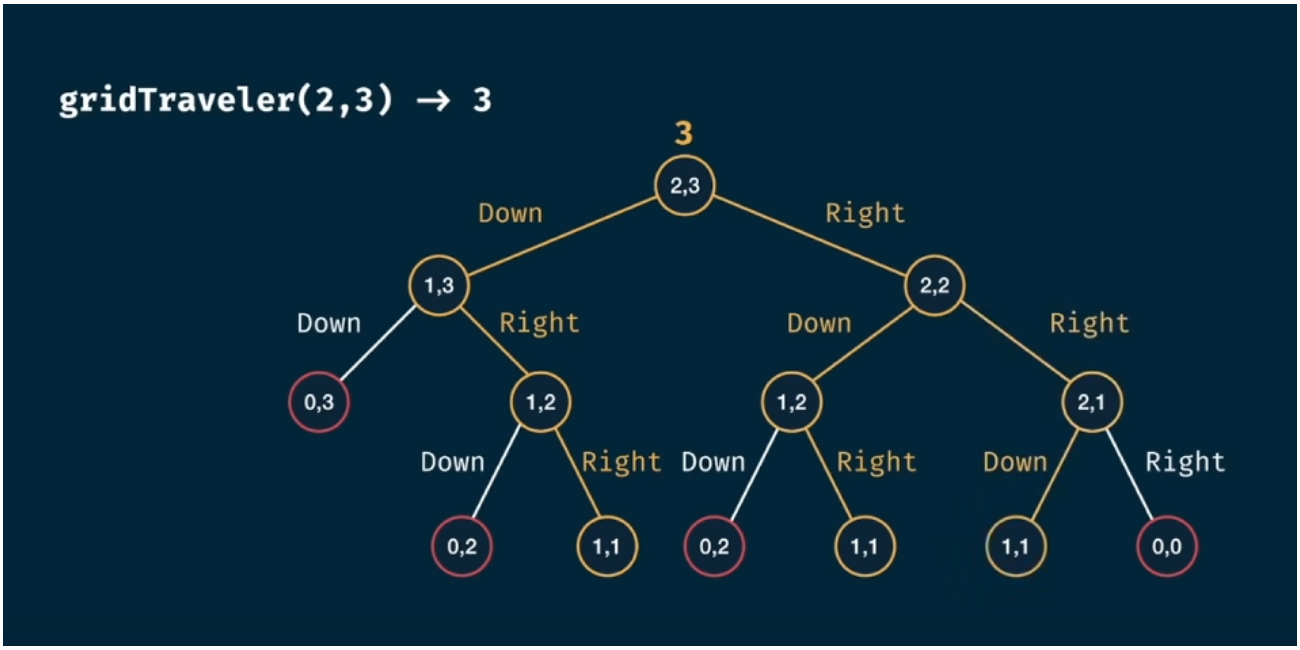


¿Cómo quedó la complejidad temporal y espacial? Bueno, espacialmente seguimos teniendo $O(n)$ porque bajamos hasta el caso base, al menos una vez, pero temporalmente ahora también tenemos $O(n)$



Llegando a un punto particular en una grilla

Digamos que queremos movernos a un punto particular fijo y solo podemos movernos a la derecha y para abajo. Es sabido que van a existir muchas formas de llegar a ese punto.
Ahora bien, a medida que nos vamos acercando, el problema se achica. Así que, seguramente, vaya a existir algún otro camino por el que lleguemos al punto (o no).

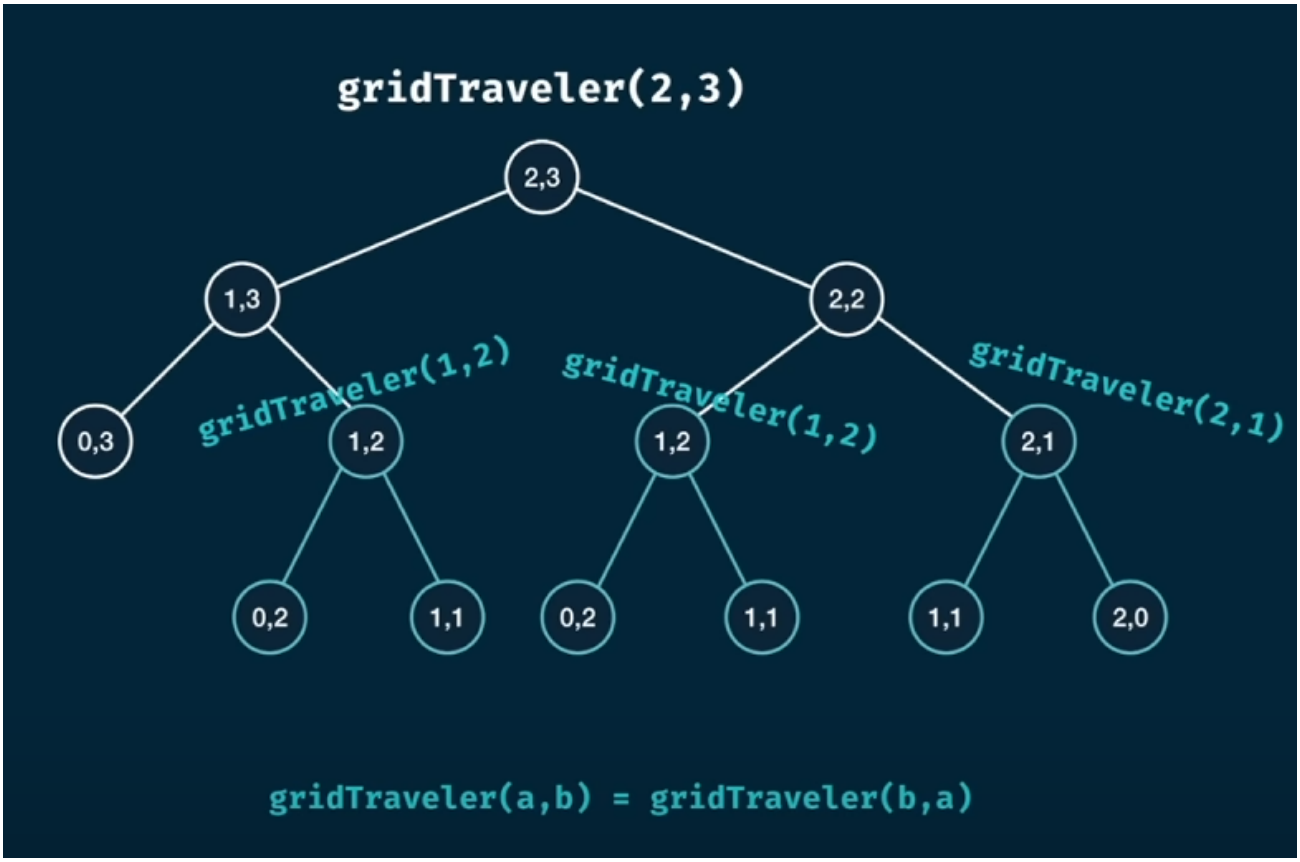


Sin ningún tipo de memorización, la complejidad que da en tiempo es $O(2^{n+m})$ porque para cada caso, tengo dos opciones que dependen de dos variables mientras que en complejidad espacial da $O(n+m)$ que es lo que nos cuesta bajar hasta el caso base gastando todas las opciones.

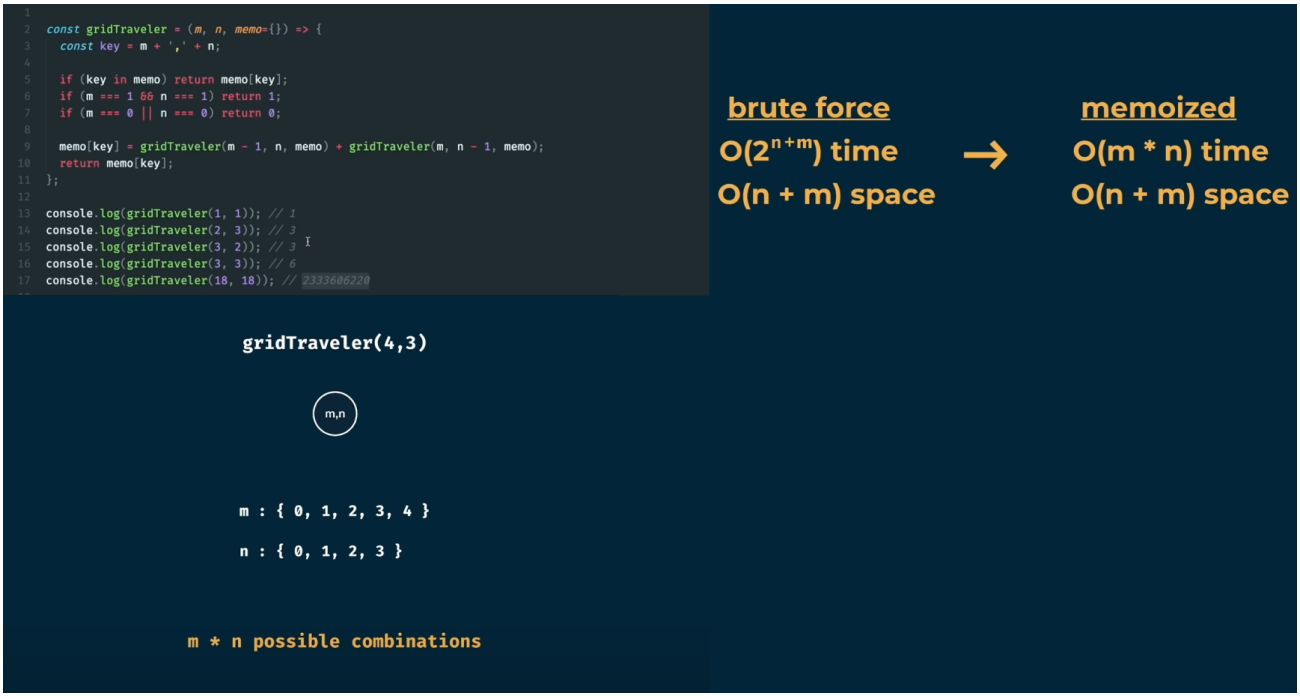
Con Programación Dinámica

¿Cómo hacemos para guardar en una estructura de memorización algo que tiene más de un argumento? Normalmente, lo ideal es guardar algo de la especie **m + ', '+ n**.
Es realmente importante NO guardar todos de forma continua y sí distinguirlos por algún símbolo **(,)** por si en algún

momento, tenemos que considerar los elementos separados pero que valgan por simetría.



Por último, veamos como mejoró nuestra solución



Receta de Programación Dinámica (Top-Down)

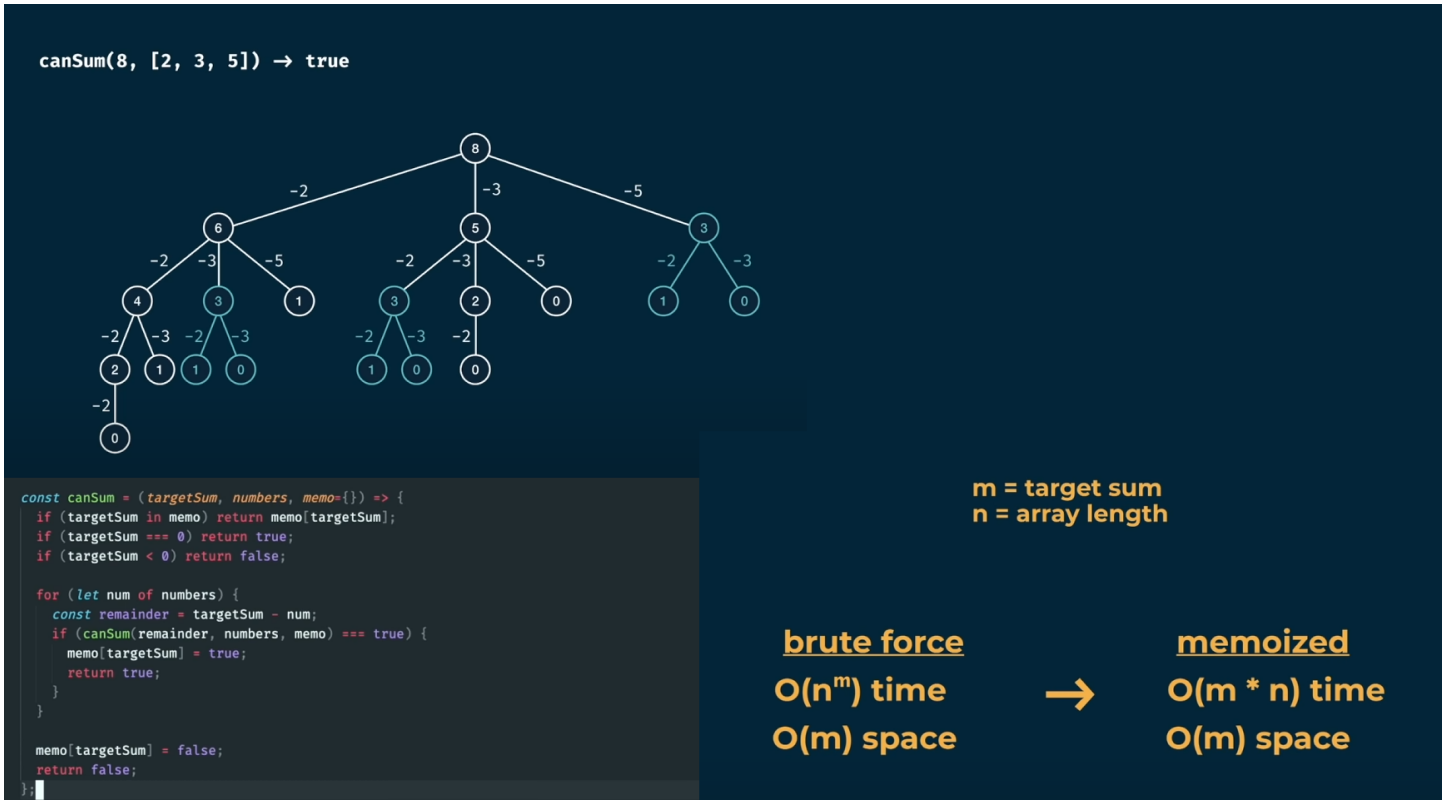
- Hacer que la solución sin programación dinámica funcione
  - Visualizar el problema como un árbol.
  - Implementar ese árbol usando recursión.
  - Testear si la solución funciona.
- Optimizarlo

- Agregar un objeto de memo.
- El objeto memo debe pasarse a todos los llamados recursivos. Normalmente lo pasamos por referencia o usamos variables globales.
- Agregar casos base para los valores memorizados.
- Guardar los valores de los return dentro del objeto memo.

Para resolver problemas de programación dinámica se recomienda ser meticuloso, hacer dibujos, armar buenas soluciones de fuerza bruta y recién luego pensar la estrategia de memorización. Memorizar es sencillo, qué memorizar es complicado.
No traten de implementar una solución óptima desde el principio. Make it work first.

canSum problem

La idea es que dado un array, se diga si existe algún subconjunto de elementos que suman un valor n particular.
Tip importante: En problemas donde hay que llegar a determinado valor k, siempre, pero siempre conviene disminuir ese valor de target y no ir sumando.



Algo que siempre hay que tener en cuenta es que, si llegamos a determinado valor, el árbol siempre va a tener el mismo comportamiento para ese valor. Nótese que acá, en este problema particular tampoco nos interesa qué numeros suman nuestro target, sino, simplemente si hay un camino o no.

Programación Dinámica Bottom-Up

Anexo

Demostraciones de Grafos

Normalmente se hace inducción sobre las aristas, aunque en algunos casos, optamos sobre los vértices.
Lo más común es tener un caso base del tipo $n = 1, m = 0$ o $n = 2, m = 1$.
Good to Know: Usar a favor las propiedades.

Distancia entre dos vértices

Si P es el recorrido entre u y v tiene longitud d(u,v) $\implies$ P es un camino.
Recordemos algunas definiciones

- Un recorrido es una sucesión de vértices y aristas, que no tiene ninguna limitación.
- d(u, v) es el recorrido más corto entre u y v. Por lo que, podría haber otros.

- P es un camino: Un camino es un subconjunto de un recorrido donde no puede haber vértices repetidos.

Probando por el absurdo:

$$\neg Q \implies R$$

En este caso

- $\neg Q$ = P no es un camino
- R = P es el recorrido entre u y v y tiene longitud d(u,v)

La idea es llegar a algo falso.

$$V \implies F = F$$

Suponemos que P NO es un camino $\neg Q = V$, entonces hay, al menos un vértice en P que se repite.
Recorrido P

$$u \to z \to z \to v$$

¿Podemos encontrar algún camino que sea más corto que P? (esto haría falso el consecuente).
Defino un camino T

$$u \to z \to v$$

Luego, la distancia mínima entre ambos recorridos

$$min(dP(u,v), dT(u,v)) = min(3,2) = 2 = dT(u,v)$$

Finalmente, llegamos a un absurdo, porque la distancia mínima la tenemos con el recorrido T y no con P.
Si hubiesemos llegado a que con $\neg Q$, P es el camino más corto, hicimos mal la demostración.

Representación de grafos en la práctica

Resumen: Si el grafo es denso $|E|$ es igual de grande que $|V|^2$ y necesitás saber si existe una arista entre dos vértices lo más rápido posible, usá una matriz de adyacencia. Caso contrario, una lista de adyacencia.

Numeración de los vértices

Numeramos los vértices desde 0 hasta n-1 (donde n es la cantidad de vértices). Esto hace que sea más fácil de trabajar con arrays y matrices.

Ejemplo

Queremos ver si podemos llegar desde la ciudad A hasta la ciudad D. Un cliente nos trae la siguiente información donde la primer letra corresponde al saliente y la segunda el destino.

```
{
  "rutas": [
    ["A", "B"],
    ["B", "C"],
    ["C", "D"],
    ["A", "E"],
    ["E", "F"]
  ],
  "origen": "A",
  "destino": "D"
}
```

Acá es donde tenemos que pensar cómo armamos nuestro grafo.

Forma 1 (Lista de Adyacencia)

Sabemos que tenemos que ir desde una ciudad a otra, asì que estarìa bueno tener la lista de conexiones de una ciudad con todas las demás. Es decir, meter todas las ciudades con las que se conecta una en una lista.
En este caso, nos conviene una lista de adyacencia (conjunto vecindario) y ahora veremos por qué.
Nos quedaría entonces, algo así

```
    graph = {
        "A": ["B", "E"],
        "B": ["C"],
        "C": ["D"],
        "E": ["F"],
    }
```

El problema entonces sería algo así:

- Empiezo en A, tomo cada vecino de él y me fijo con quién se conectan.
  - Llegué a B
    - B se conecta con C, C se conecta con D, entonces A se conecta con D.
    - B se conecta con E, E se conecta con F y F no se conecta con nadie.
  - Respuesta final: Sí existe un camino, y ese es A -¿B -¿C -¿D y es un circuito simple.

¿Cómo quedaría de esta manera, la complejidad temporal y espacial en el armado del grafo?

- Temporal:O(E) porque el input inicial es una lista de aristas y recorremos toda esa lista.
- Espacial: O(V + E) porque por cada nodo, guardamos una lista de sus vecinos.

¿Y en la búsqueda?

- Espacial: O(V) que es lo que cuesta ir guardando los nodos visitados.
- Temporal: O(V + E) porque por cada nodo, vemos todas sus aristas.

¿Cuál es la complejidad total del algoritmo? En tiempo es O(V + E) y en espacio O(V + E).
¿Cuando usar listas de adyacencias?

- El grafo es disperso, es decir, tiene más aristas que nodos.
- Es importante optimizar el espacio.
- Buscás los vecinos frecuentemente.

Forma 2 (Matriz)

Es importante recordar que en la matriz, la dimensión es siempre como si todos los nodos estarían relacionados con todos. En este caso tenemos 6 nodos, y en el peor caso se conectan todos con todos. Así que la matriz sería de 6x6.

```
    matrix = [
    [0, 1, 0, 0, 1, 0],  //A
    [0, 0, 1, 0, 0, 0],  //B
    [0, 0, 0, 1, 0, 0],  //C
    [0, 0, 0, 0, 0, 0],  //D
    [0, 0, 0, 0, 0, 1],  //E
    [0, 0, 0, 0, 0, 0],  //F
]
```

¿Cómo quedaría de esta manera, la complejidad temporal y espacial en el armado del grafo?

- Temporal: O(V^2).
- Espacial: O(V^2).

¿Y en la búsqueda?

- Buscar todos los vecinos de A - Tiempo: O(V).
  - Buscar todos los vecinos B - Tiempo: O(V). Buscar todos los vecinos de C - Tiempo: O(V).
  - Buscar todos los vecinos de E - Tiempo: O(V). Buscar todos los vecinos de F - Tiempo: O(V).

Como esto se hace varias veces, sería algo de O(cantVecinosNodo * V) = O(V).
¿Cuál es la complejidad total del algoritmo? En tiempo y espacio es O(V^2).
¿Cuando usar matrices de adyacencia?

- El grafo es denso ($|E| \approx |V|^2$).
- Necesitás verificar si existe rápidamente una conexión entre dos nodos.
- El grafo es estático, es decir, no se agregan ni se quitan aristas frecuentemente.

Comparación de las dos estructuras

Operación	**Lista de Adyacencia**	**Matriz de Adyacencia**
Espacio (Construcción)	O(N + E)	$O(N^2)$
Tiempo (Construcción)	O(N + E)	O(E)
Acceder a los Vecinos (por Nodo)	O(1)	O(N)
Tiempo para Buscar Camino	O(N + E) (DFS/BFS)	$O(N^2)$
Espacio Temporal (Búsqueda)	O(N)	O(N)
Acceder a la existencia de una Arista	O(k) (conjunto de vecinos)	O(1)

Tabla 1: Comparativa de complejidad temporal y espacial de las representaciones de grafos.