# Clase Recorridos en Grafos

Martín Amster y Dafne Yudcovsky

Universidad de Buenos Aires

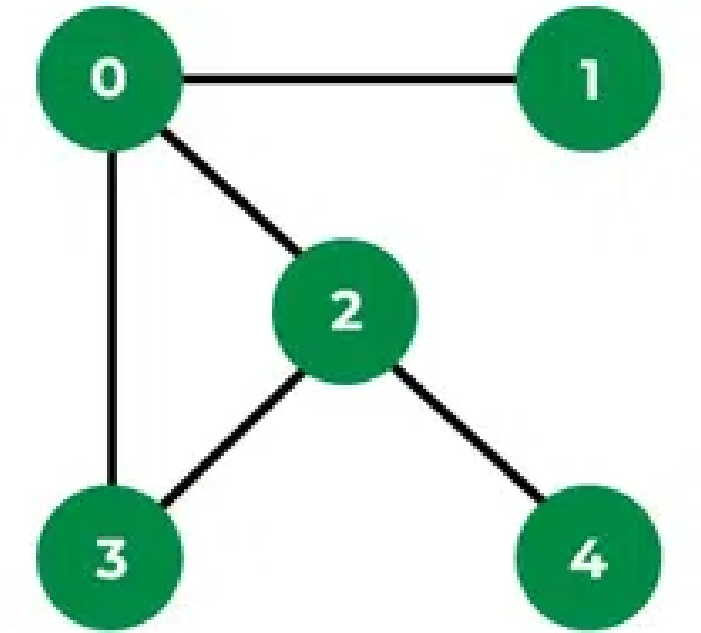
Lo importante no es llegar, lo importante es el RECORRIDO

# Definición DFS

DFS es un algoritmo recursivo que sigue la idea de backtracking para poder recorrer todos los nodos. Lo podemos usar tanto para grafos dirigidos como para no dirigidos.
Vamos a recorrer en profundidad (Depth): siempre vamos hasta el final de la rama y de ahí subimos.

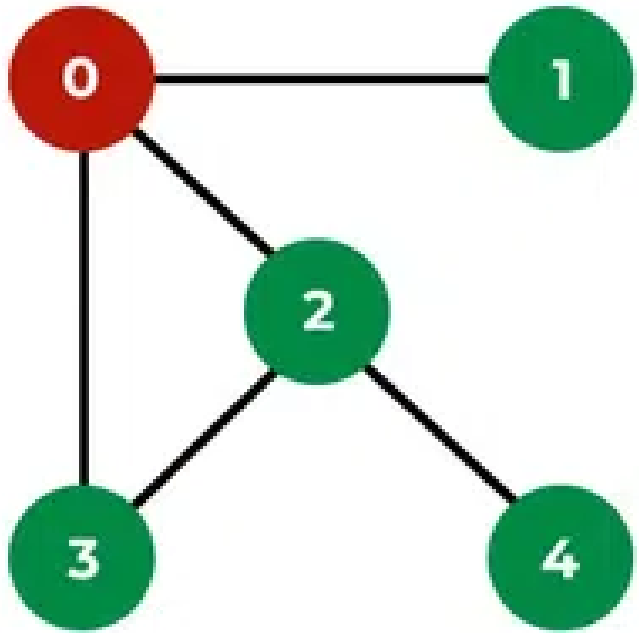
# Ejemplo DFS

Tomamos un grafo de ejemplo con 5 nodos. Verde es no visitado, rojo es visitado.



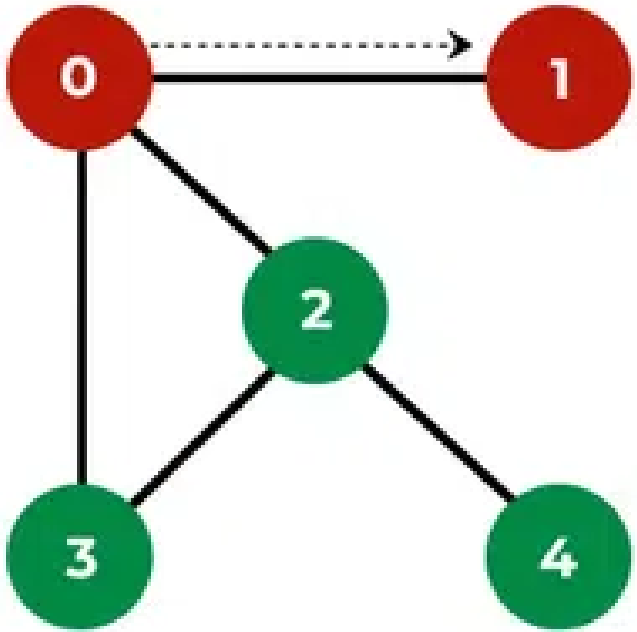
# Ejemplo DFS

Arrancamos desde el 0 y lo marcamos como visitado. Ahora podemos recorrer cualquiera de los adyacentes al 0: 1, 2 o 3.



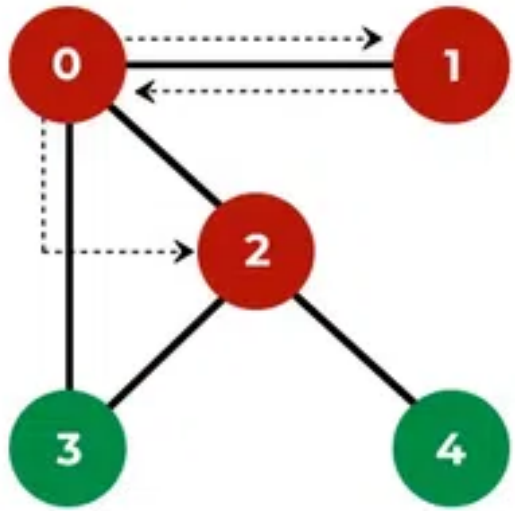
# Ejemplo DFS

Visitamos el 1 y ahora podemos visitar a los adyacentes del 1 no visitados.



# Ejemplo DFS

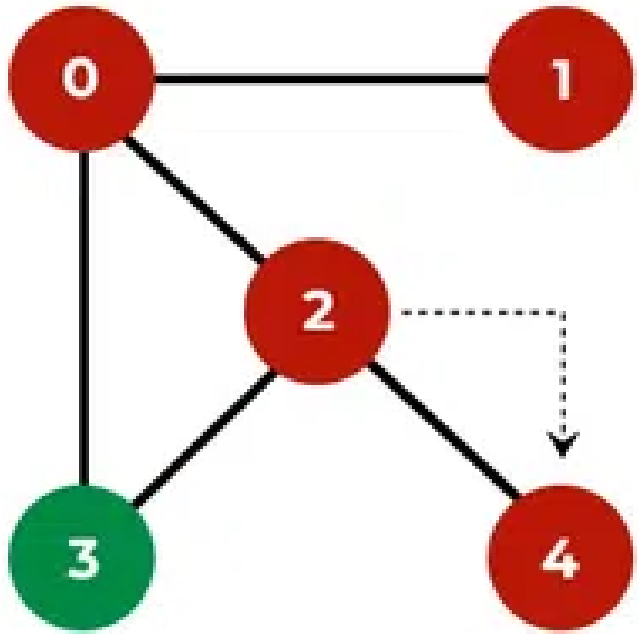
Como no hay adyacentes del 1 no visitados vuelve hacia arriba y sigue con los adyacentes no visitados del 0.




Ahora puedo seguir por cualquiera de los adyacentes del 2, que son el 3 o el 4.

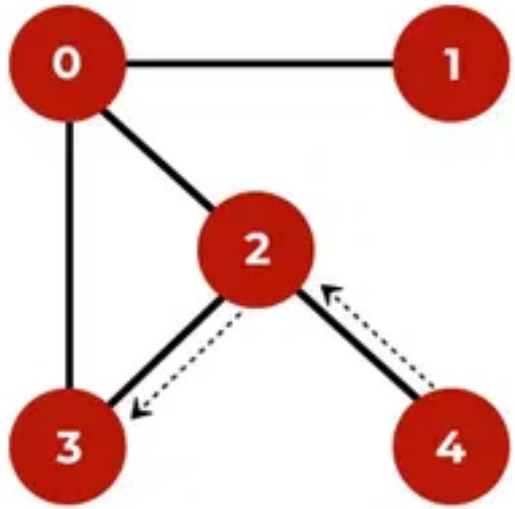
# Ejemplo DFS

Visito el 4



# Ejemplo DFS

Por el 4 no puedo seguir así que vuelvo al 2 y sigo en su lista de adyacentes, por lo que visito al 3.




Ahora vuelvo al 0 y ya no tengo más adyacentes que no fueron visitados.

# DFS

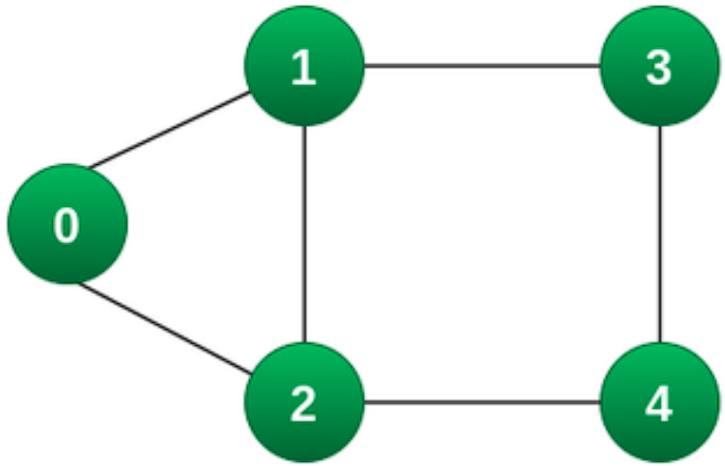
La complejidad de DFS es $O(m + n)$ ya que recorro todos los nodos una sola vez y reviso las aristas también una sola vez (aunque puede que no recorra todas).
Sin agregarle complejidad al algoritmo de DFS podemos pedirle que nos devuelva el árbol o bosque (como un vector de padres) y la lista de backedges, que son las aristas que generan ciclos.

# Definicion BFS

Con BFS también vamos a recorrer todos los nodos pero en lugar de recorrer en profundidad va recorriendo a lo ancho. Se suele implementar iterativo. La idea del algoritmo es arrancar en algún nodo y recorrer todos sus vecinos, luego los vecinos de sus vecinos...

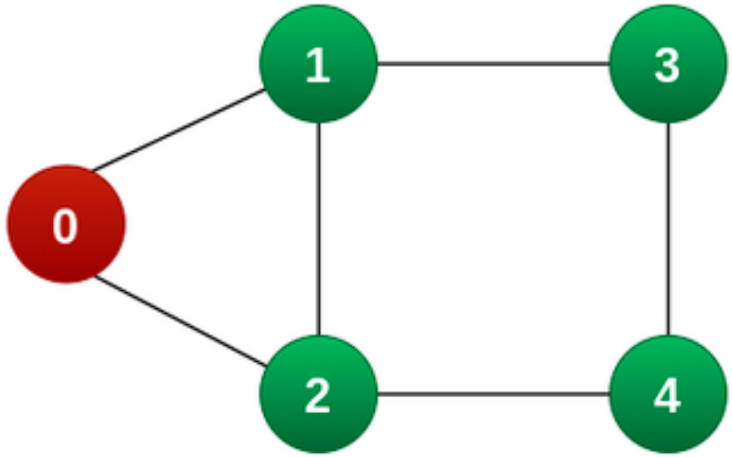
# Ejemplo BFS

Tomemos como ejemplo este grafo y comenzamos a recorrer desde el nodo 0.



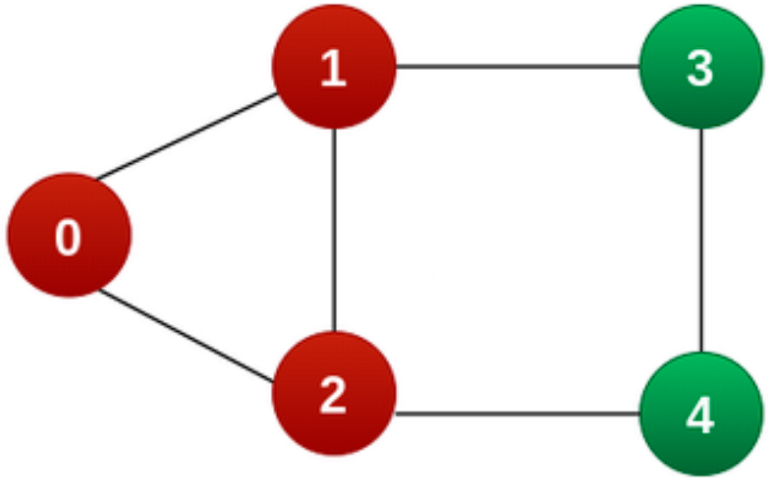
# Ejemplo BFS

Visitamos el 0 y vemos cuáles son sus vecinos



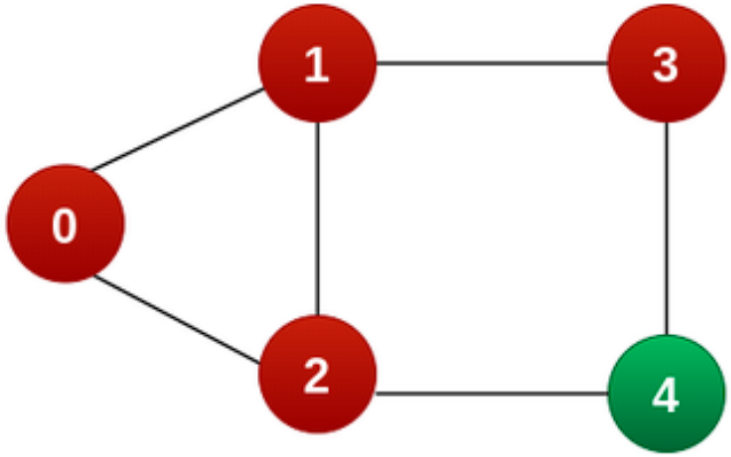
# Ejemplo BFS

Visitamos ambos vecinos y luego vamos a ver los adyacentes al 1.



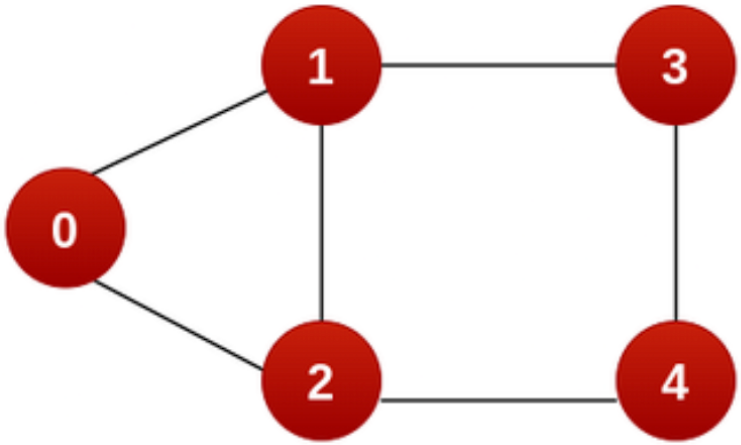
# Ejemplo BFS

El 3 era el único adyacente al 1 que no había recorrido.



# Ejemplo BFS

Sigo con los adyacentes del 2.




Luego el algoritmo se fijará en los vecinos del 3 y 4 pero como no quedan vecinos sin recorrer terminará la ejecución.

# BFS

La complejidad de BFS también es **O(m+n)**, al igual que DFS.
Nos devuelve un árbol $v$-geodésico, siendo $v$ el nodo desde el cual corremos el algoritmo. Puede devolver también las distancias de $v$ a todos en el grafo (que son iguales que en el árbol ya que es $v$-geodésico!!)

# Ejercicio 1

Vamos a arrancar con uno tranqui:

## Ejercicio 1: chequeo de conectividad

¿Cómo podemos ver que un grafo es conexo? Es decir, dado $G = (V, E)$, queremos chequear si $\forall u, w \in V$, $\exists$ camino entre $u$ y $w$, o que $d(u, w) = k$ finito.

# Ejercicio 1

Podríamos correr alguno de los algoritmos de recorrido y ver si todos los nodos fueron visitados (usando el vector visitados). Para esto usamos un código aún más simple que el visto en la teórica ya que si no puedo llegar a un nodo $w$ corriendo DFS (o BFS) desde otro nodo $v$ no me interesa correr de vuelta el algoritmo desde $w$, sino que ya sé que no es conexo. Si lo queremos resolver con DFS, podríamos usar esta implementación:

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

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

# Ejercicio 1

Ahora... ¿Cómo podemos ver si es conexo o no, luego del recorrido? Podemos recorrer el vector de visitados y analizar si están todos los valores en *true*. Significa que desde nuestro nodo inicial, pudimos encontrar un camino para todos los nodos del grafo, con lo cual lo hace conexo. Por el contrario, si encontramos alguna posición $i$ en *false*, significa que desde el nodo que arrancamos no pudimos llegar al vértice $i$, con lo cual el grafo no es conexo.
Veamos el código...

# Ejercicio 1

```
//Continuacion del codigo de arriba...

//Arranco el recorrido por el primer vertice
dfs(FIRST_NODE)

//Compruebo el vector de visitados
for(int i = 0; i < visitados.size(); i++){
    // Si encuentro alguno no visitado,
    // el grafo no es conexo
    if (!visitados[i]) return false;
}

// Si estan todos visitados, el grafo es conexo
return true;
```

# Ejercicio 2

## Ejercicio 2: componentes conexas

Dar un algoritmo que dado un grafo devuelva la cantidad de componentes conexas que tiene.

## Definition

Dos vértices $u, v$ pertenecen a la misma **componente conexa** si existe un camino entre ellos.

# Ejercicio 2

Podríamos resolverlo con la implementación usual de DFS o BFS agregando un contador de componentes conexas o bien usar la implementación del ejercicio anterior y agregarle algo asi:

```
int componentes = 0;
for (int i = 0;i < n;i++) {
    if (!visitado[i]) {
        componentes++;
        recorroAPartirDelVertice(i);
    }
}
```

Donde `recorroAPartirDelVertice` es la implementación del ejercicio anterior de DFS (o una similar de BFS).

# Ejercicio 3

## ¡Para combatir el bajón!

A Rasta le gusta mucho comer alfajores. Es por eso que quiere ir todas las mañanas a la fábrica de alfajores que está en una esquina cerca de su casa para comprárselos. Pero sucede que Rasta detesta enormemente repetir caminos, ¡y ni hablemos de caminar de más! Rasta anhela saber de cuántas maneras distintas puede ir de la esquina donde queda su casa a la fábrica de alfajores caminando siempre lo menor posible.

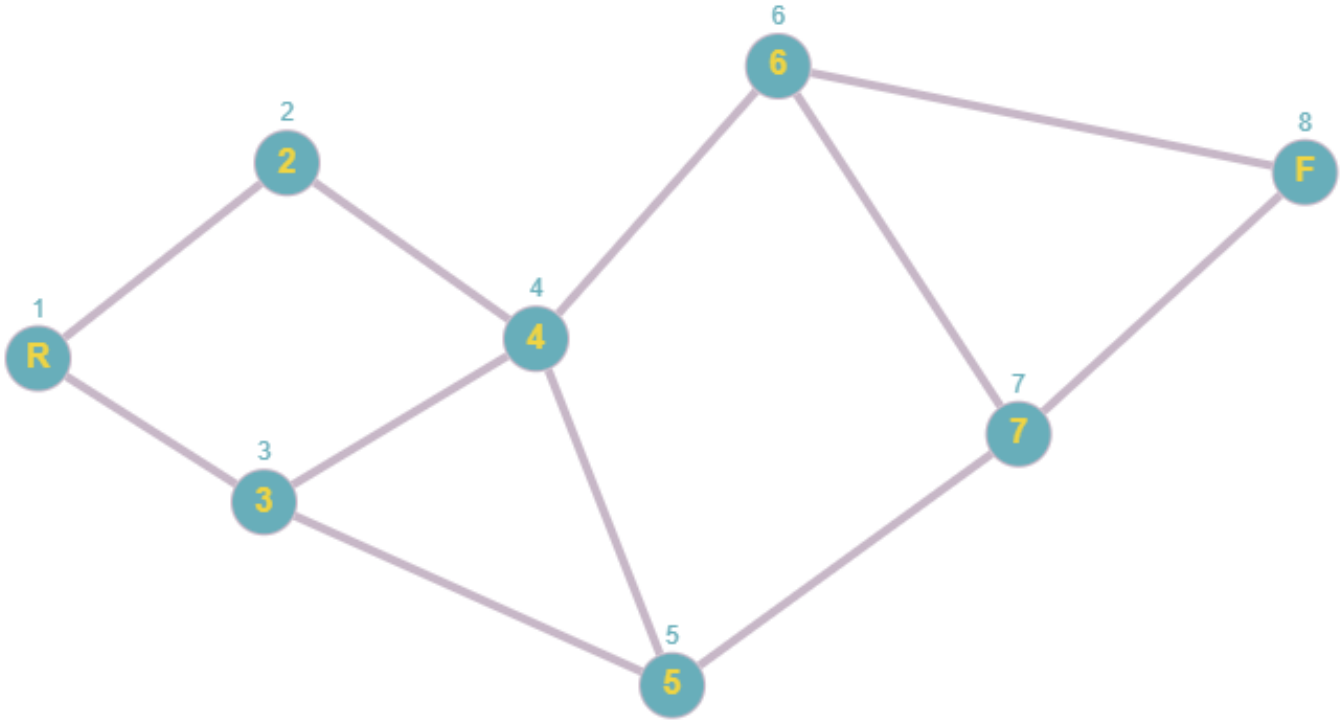
# Ejercicio 3

**Idea**

O sea, en resumidas cuentas, queremos calcular el número de caminos con mínima cantidad de cuadras entre la casa de Rasta (R) y la fábrica (F). ¿Cómo podemos hacer?

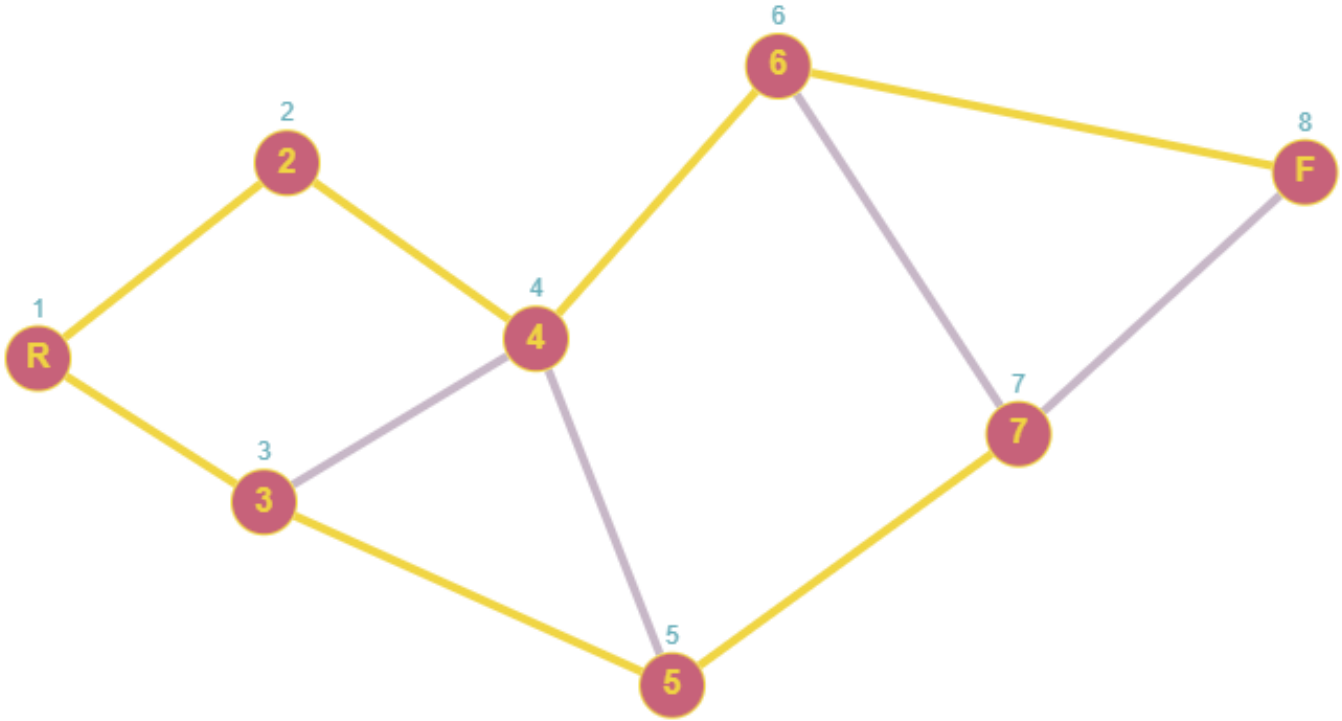
Primero modelemos el problema como un grafo. Sea $G = (V, E)$.

- ¿Quienes van a ser los vértices de $V$? Las esquinas por las que puede pasar Rasta.
- ¿Quienes van a ser las aristas de $E$? Las cuadras del barrio.


1
R
2
2
3
3
4
4
5
5
6
6
7
7
8
F

Vamos a resolverlo usando programación dinámica.

- Sabemos que BFS nos da un vector con las distancias de v a todos, así que comenzamos corriendo un BFS.
- Luego usando el vector de distancias hacemos un algoritmo que de abajo hacia arriba en el árbol chequee si para algún otro vecino de $v$ distancia[$w$] = distancia[adyacente a $w$] + 1
- Si pasa eso debo volver a correr el algoritmo desde el adyacente de $w$, ya que podría haber más de una manera de llegar a ese con distancia mínima también
- Lo hacemos con programación dinámica ya que tenemos superposición de problemas.


1
R
2
2
3
3
4
4
5
5
6
6
7
7
8
F

Planteamos la función #caminosHasta

$$\#caminosHasta(v) = \begin{cases} 1 & R = V \\ \sum_{w* \in N_v} \#caminosHasta(w) \end{cases} \quad (1)$$

donde $w* = \{w \in N_v : \delta(w) + 1 = \delta(v)\}$

Y lo inicializamos con #caminosHasta(F)

# Ejercicio 3

```
int cantidadDeCaminosHasta(v) {
    if (distancia[v] == 0) return 1;
    if (memo[v] != -1) return memo[v];
    int res = 0;
    for (int vecino : aristas[v]) {
        if (distancia[vecino] + 1 == distancia[v]) {
            res += cantidadDeCaminosHasta(vecino);
        }
    }
    memo[v] = res;
    return res;
}
```

¿Qué complejidad tiene este algoritmo?

- Corrimos BFS una vez:
- Programación dinámica: como la función y la memoización recibe los vértice, pero a cada uno accedo $|E|$ veces, entonces la complejidad de este paso es $O(|V| + |E|)$

# Ejercicio 4

## Nadie acepta nada: chequeo de bipartito

Actualmente, nos encontramos viviendo el ocaso de la televisión tradicional, donde cambiamos actores de años de trayectoria por nuevos personajes denominados "Streamers", que sin maquillaje ni escrúpulos hablan sobre tópicos variados de la vida. La plataforma MyTube genera un ambiente amigable para la transmisión en vivo, y además teniendo en cuenta que para el usuario promedio (Gen Z/Millenial) es una red social recurrente, Micky Granadina y Mico Bonniato quieren llamar a varios de estos Streamers para crear dos canales de Streaming. Estos van a ser Aglo y Zulu-Tv. Ambos estaban muy entusiasmados pero empezaron a encontrar problemas. La gran exposición de los Streamers generó algunos conflictos entre ellos, por lo que Micky y Mico quieren ver si pueden armar ambos canales sin conflictos internos.

# Ejercicio 4

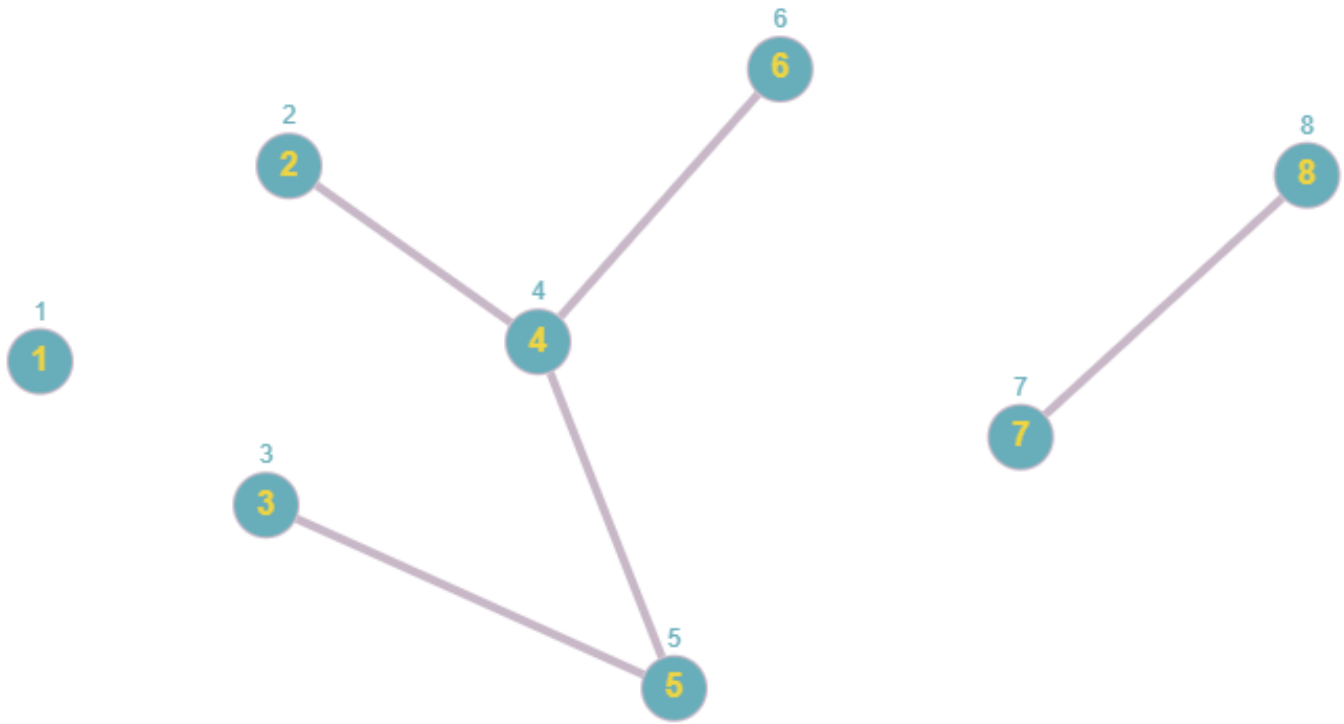
## Nadie acepta nada: chequeo de bipartito

Se cuenta con $N$ Streamers en total, y con $c$ pares de conflictos $(A, B)$, que significa que $A$ está peleado con $B$. Se quiere modelar el problema como un problema de grafos, en el que respondamos si es posible armar dos canales sin conflictos.

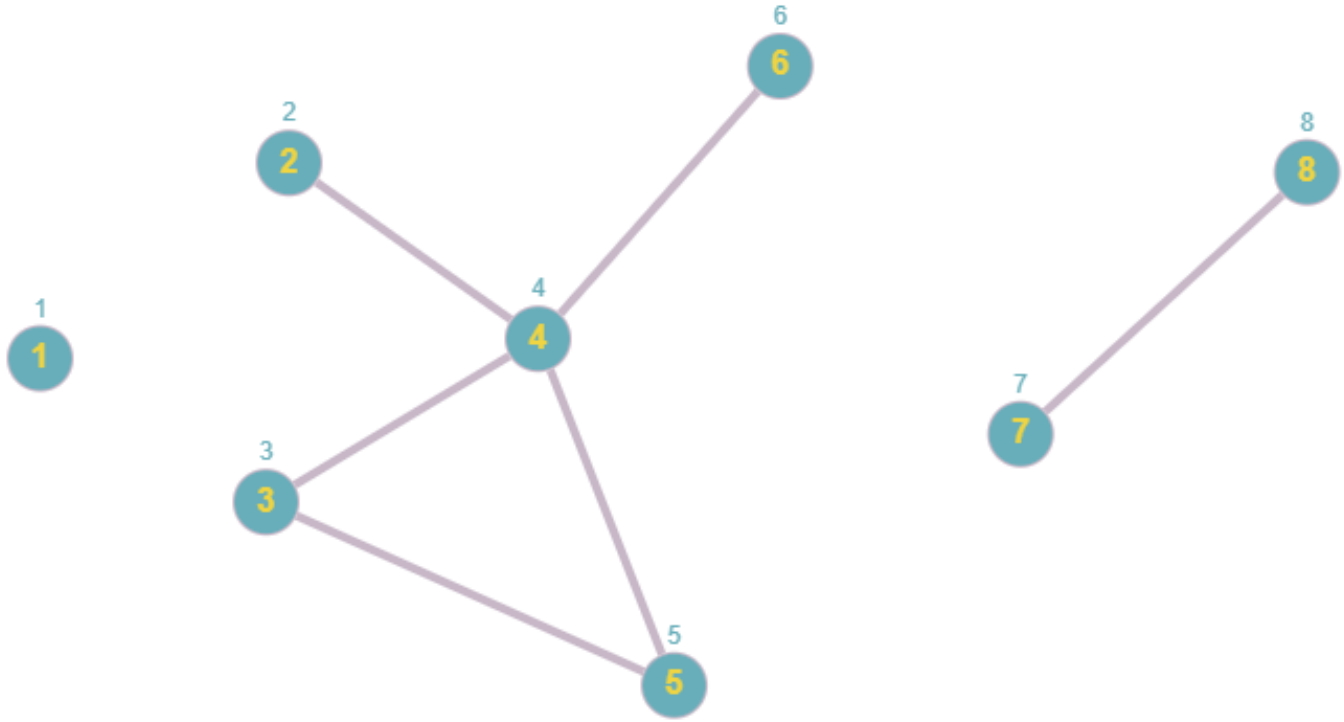
Primero modelemos el problema como un grafo. Sea $G = (V, E)$.

- ¿Quiénes van a ser los vértices de $V$? Los Streamers
- ¿Quiénes van a ser las aristas de $E$? Los conflictos

Con éste se puede?



Y con éste?



# Ejercicio 4

Idea:

- Puede haber streamers sin conflictos, entonces la solución tiene que preservarse para cada **componente conexa**.
- Recordemos que dos vértices $u, v$ pertenecen a la misma **componente conexa** si existe un camino entre ellos.

- A fin de cuentas queremos que cada componente conexa sea un **grafo bipartito**.

**Definition**

Un grafo es **bipartito** si existen dos conjuntos $V_1$ y $V_2$ tales que $V_1 \cup V_2 = V$, $V_1 \cap V_2 = \emptyset$ y $\forall v \in V_1, (u, v) \in E \implies u \in V_2$, y análogamente para $V_2$.
Es decir que los vértices en $V_1$ sólo son adyacentes a los de $V_2$ y los de $V_2$ sólo son adyacentes a los de $V_1$.

- En resumen, quiero verificar si todas las componentes conexas de $G$ son bipartitas.
- Puedo usar BFS, ya que al armar el árbol de BFS, lo que tengo que chequear es que los nodos que tengan nivel con la misma paridad no tengan aristas en común. Recordemos que por invariante de BFS, sabemos que el nivel es la distancia a la raíz, que puede ser cualquier nodo de la componente en principio.

**Definition (Lema)**

Sean $u, v \in V : (u, v) \in E$.
$G$ es bipartito $\iff$ no hay ningún ciclo impar

# Solución

```
bool isBipartiteComponent(int& graph[MAX_NODES][MAX_NODES], int n,
    int start, int& distance[]) {
  Queue q; distance[start] = 0; // Distancia inicial
  enqueue(&q, start);
  while (!isEmpty(&q)) {
    int vertex = dequeue(&q);
    for (int possible_neighbour in range(n)) {
    // Si no visite el nodo, lo visito y pongo su distancia
      if (graph[node][posible_neighbour]) {
        if (distance[posible_vecino] == -1) {
          distance[posible_vecino] = 1 + distance[vertex];
          enqueue(&q, possible_neighbour);
        } // No es bipartito si los nodos vecinos tienen distancia
            par o impar
        else if ((distance[possible_neighbour] - distance[vertex])
            % 2 == 0) {
          return false; // No es bipartita
        }
      }
    }
  }
  return true;
}
```

```
bool isBipartite(int& graph[MAX_NODES][MAX_NODES], int n) {
    int distance[MAX_NODES];

    // Inicializar todos los nodos como no coloreados
    for (int i = 0; i < n; i++) {
        distance[i] = -1;
    }

    // Verificar cada componente conexa
    for (int i = 0; i < n; i++) {
        if (distance[i] == -1) { // Si no ha sido visitado
            if (!isBipartiteComponent(graph, n, i, colors)) {
                return false;
            }
        }
    }

    return true;
}
```

# ¿Nos tomamos un break?

# Ejercicio 5

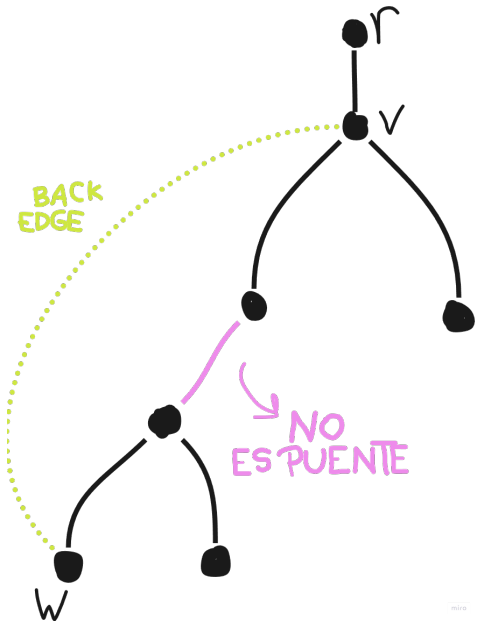
## Ejercicio 5: detectar aristas puente

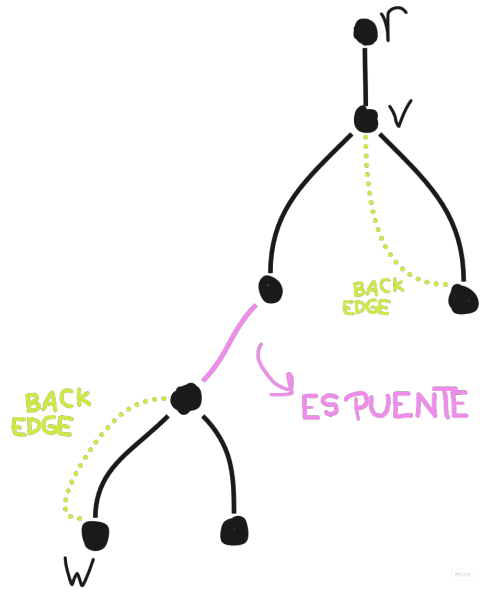
Dar un algoritmo lineal basado en DFS para encontrar todas las aristas puente de $G$ (2d de la guia).

# Ejercicio 5

A tener en cuenta:

- Una arista es puente si al sacarla aumenta la cantidad de componentes conexas
- Una back-edge nunca puede ser puente
- Las aristas que son puentes son aquellas tree-edges que no tienen una back-edge que las "cubra"



# Ejercicio 5

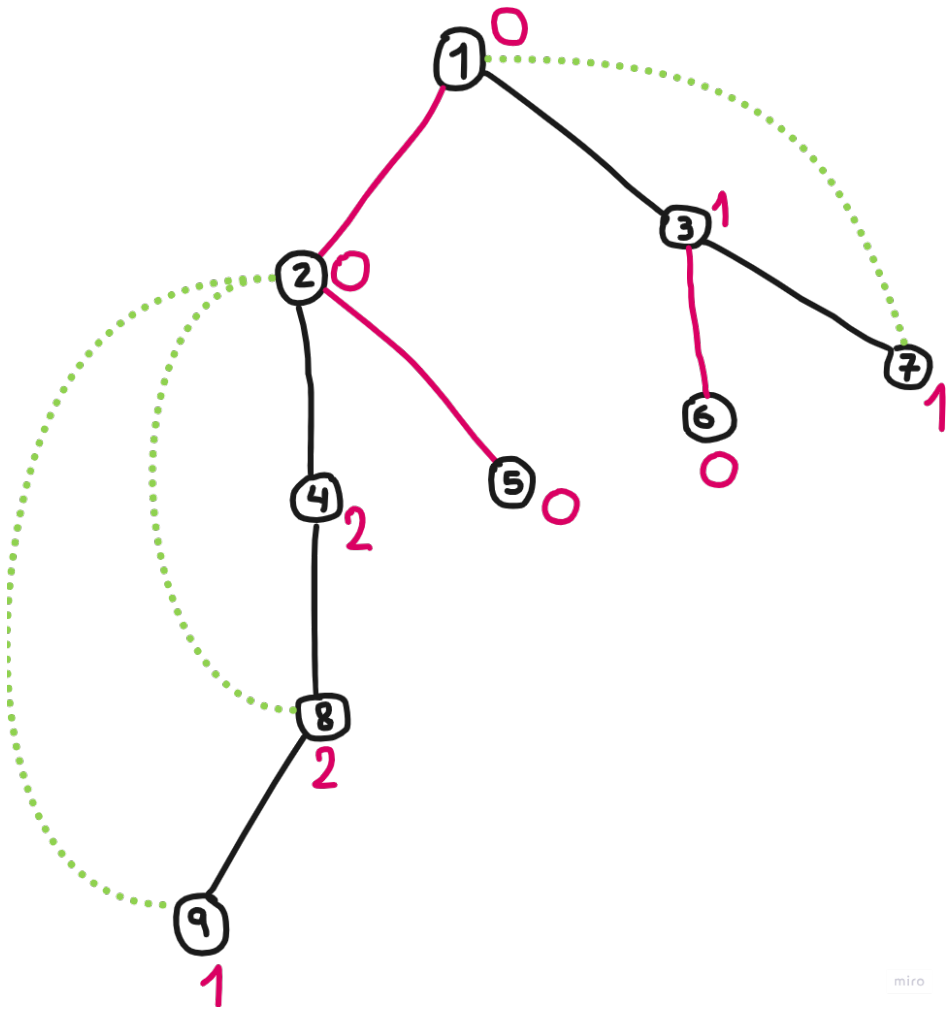
Primero debemos convencernos de que la cantidad de backedges que cubren la arista entre $v$ y su padre se puede calcular de la siguiente manera:

$$\begin{aligned} \text{cubren}(v) = \ & \text{backEdgesConExtremoInferiorEn}(v) \\ & - \text{backEdgesConExtremoSuperiorEn}(v) \\ & + \sum_{w \in \text{hijos}(v)} \text{cubren}(w) \end{aligned}$$

Veamos un ejemplo.

# Ejercicio 5

Los números al lado de los nodos indican el valor de cubren en dicho nodo.
Si cubren de un nodo $v$ es 0 significa que la arista de $v$ al padre es puente



# Ejercicio 5

Ahora si corremos un DFS que de alguna manera calcule para cada nodo `backConExtremoInferiorEn` y `backConExtremoSuperiorEn`. Puede ser algo asi:

```
vector<vector<int>> treeEdges; //init en []
...
void dfs(int v, int p = -1) {
    estado[v] = EMPECE_A_VER;
    for (int u : aristas[v]) {
        if (estado[u] == NO_LO_VI) {
            treeEdges[v].push_back(u);
            dfs(u, v);
        } else if (u != p) {
            backConExtremoInferiorEn[v]++;
            backConExtremoSuperiorEn[u]++;
        }
    }
    estado[v] = TERMINE_DE_VER;
}
```

En este DFS si ya pasé por un nodo y no es el padre entonces sumo una backedge.

Notar que en esta implementación de DFS se asume que los nodos tienen tres estados: no lo vi, empecé a ver y terminé de ver

# Ejercicio 5

Hacemos un algoritmo que calcule cubren con programación dinámica ya que tenemos superposición de subproblemas (siempre para cubren de $v$ necesito cubren de los hijos).

```
vector<int> memo; //init en -1
int cubren(int v, int p = -1) {
    if (memo[v] != -1) return memo[v];
    int res = 0;
    res += backConExtremoInferiorEn[v];
    res -= backConExtremoSuperiorEn[v];
    for (int hijo : treeEdges[v]) if (hijo != p)
        res += cubren(hijo, v);
    }

    memo[v] = res;
    return res;
}
```

# Ejercicio 5

Y ahora finalmente solo necesito calcular cuántos puentes nos quedan, que es la cantidad de nodos $v$ tal que cubren$(v) = 0$.

```
int puentes = 0;
for (int i = 0;i < n;i++) {
    if (cubren(i) == 0) {
        puentes++;
    }
}
```

**OJO** nos falta restar la cantidad de componentes ya que como vimos en los ejemplos la raíz de los árboles también tiene cubren $= 0$.
puentes -= componentesConexas

# Ejercicio 5

Complejidad?

- DFS: **O(n+m)**
- cubren: **O(n)** ya que como máximo llenamos el vector de memoización
- puentes: **O(n)** porque recorro el vector cubren que tiene longitud cant nodos

# Ejercicio luces

## Ejercicio luces

Igna compró una casa en el campo y las luces vinieron falladas: si bien hay luces en todas las habitaciones, los interruptores que las controlan están en otras habitaciones respectivamente. A Igna le da miedo la oscuridad, así que nos pidió que le busquemos la manera más corta de llegar hasta su habitación (sin que queden otras luces prendidas en la casa) desde el primer cuarto arrancando con las luces de los otros cuartos apagadas y nunca puede estar en una habitación a oscuras. Si bien no nos dio muchos detalles nos dijo que como máximo su casa tiene 10 habitaciones

# Ejercicio luces

- ¿Cómo podríamos modelar el grafo en este problema?
- En principio pensemos cada habitación como un nodo y que los nodos están conectados entre sí si las habitaciones son adyacentes. ¿Cómo vemos el tema de las luces?
- Con el grafo modelado de esa manera no tenemos en cuenta a las luces. Vamos a tener que modificar el modelo... Alguna idea?

# Ejercicio luces

- Vamos a incluir ahora las luces al modelo. Necesitamos un grafo donde cada nodo nos dé información sobre la posición de Igna y de las luces de toda la casa y cada arista represente una acción válida.
- Entonces en este nuevo modelo vamos a tener un nodo para cada *estado*. Un estado es una habitación con cierta configuración de luces particular. ¿Cuántos nodos son?
- Serían $h \cdot 2^h$, siendo h la cantidad de habitaciones. Parece muchísimo peeeeeero como máximo puede haber 10 habitaciones.

# Ejercicio luces

- ¿Y ahora como ponemos las aristas?
- Va a haber una arista de $A$ a $B$ si las habitaciones son adyacentes y con los interruptores de $A$ Igna puede poner la configuración de luces que tiene $B$.

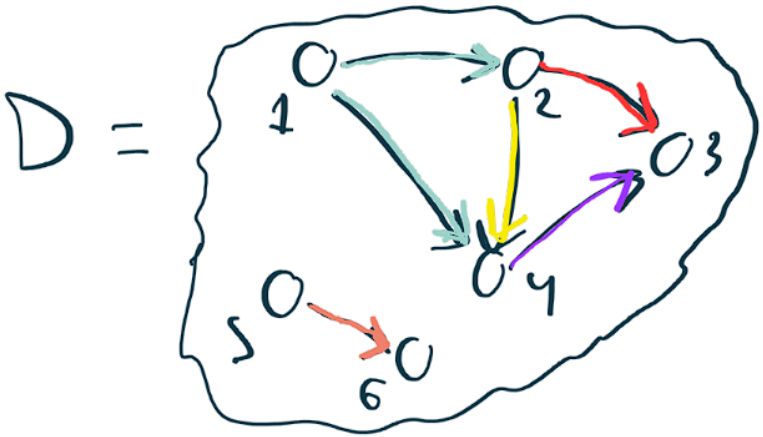
# Ejercicio luces

- Ya tenemos el modelado. ¿Ahora qué podemos hacer?
- Queremos minimizar las distancias entonces con correr BFS desde el *estado* inicial (primera habitación y luces apagadas) y luego chequear la distancia hasta el estado final (ultima habitación y luces apagadas) alcanza.

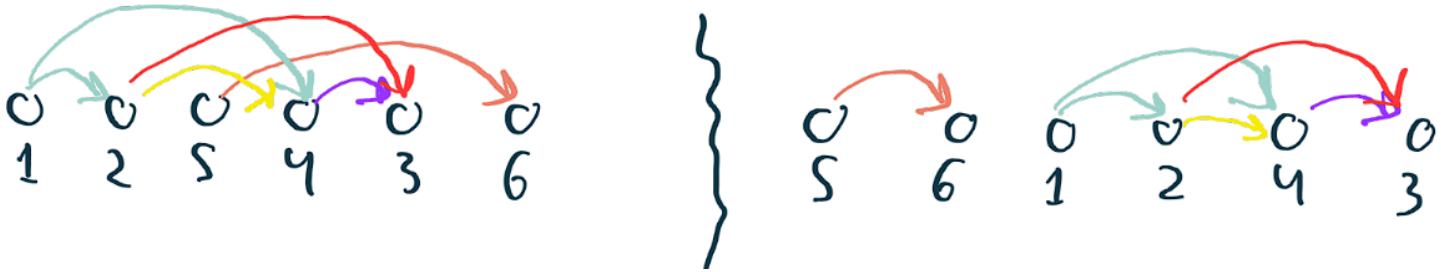
# Orden topológico

Dado un digrafo $D$, un orden topológico de $D$ es un ordenamiento $v_1 \dots v_n$ de sus nodos que cumple que toda arista queda de la forma $v_i v_j$ con $i < j$ (en el ordenamiento). Es decir, damos un orden a los nodos de tal forma que las aristas apuntan de izquierda a derecha ("no hay aristas para atrás").
Veamos un ejemplo...

# Orden topológico




Dos posibles órdenes topológicos



# Orden topológico

Idea para el algoritmo que nos da un orden topológico:

- Primero tenemos que verificar si el digrafo tiene ciclos (Ejercicio, sale con DFS). Si tiene ciclos no tiene orden topológico.
- Usamos la implementación de DFS de tres estados: no lo vi, empecé a ver y terminé de ver.
- Vamos a modificar DFS para que si terminé de procesar un nodo lo pushee a un stack finish.

El siguiente algoritmo nos da el orden topológico:

```
Topological_sort ( G ) :
|     DFS ( G )
|     return invertir finish
```

Tarea: convencerse de que funciona y demostrarlo