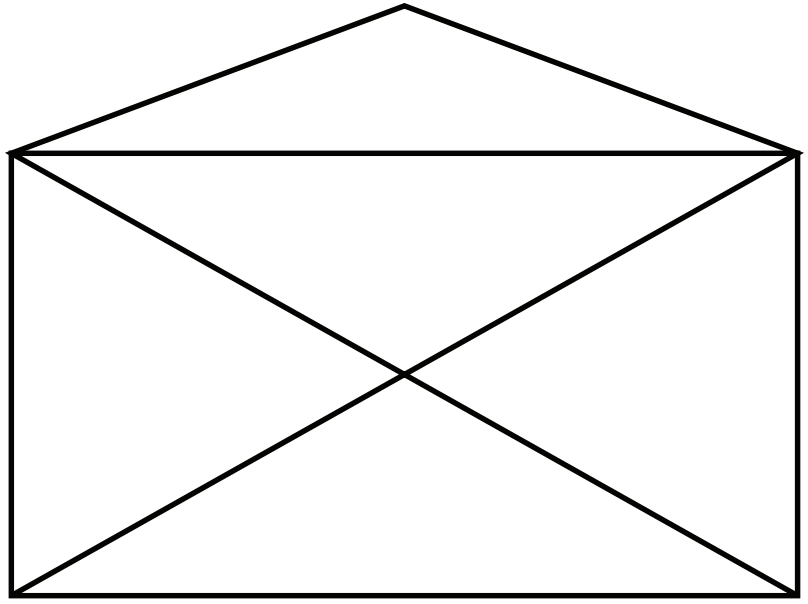
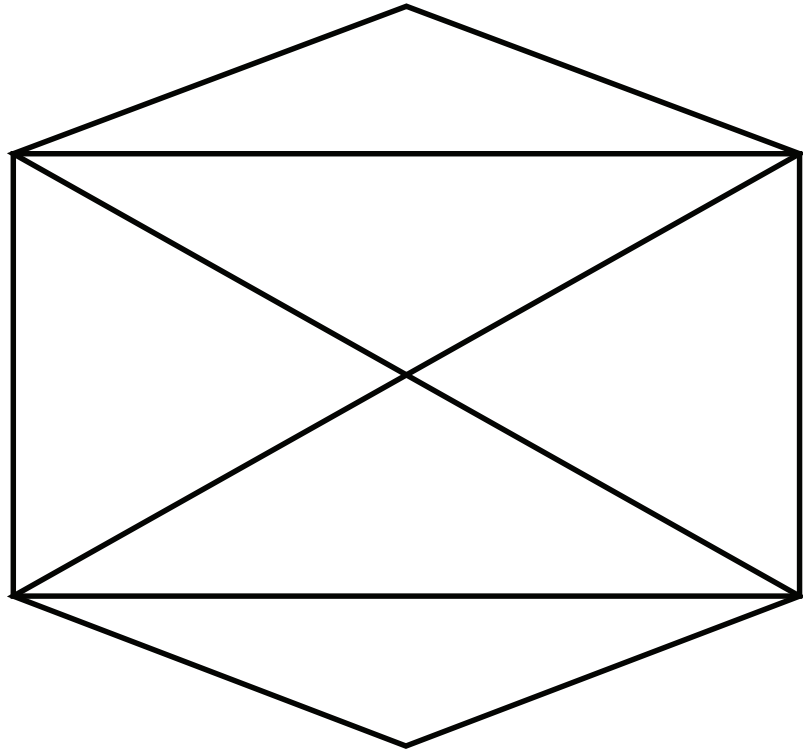
¿Se puede dibujar la siguiente figura, empezando y terminando en el mismo punto, sin levantar e lápiz del papel?



¿Y esta otra?

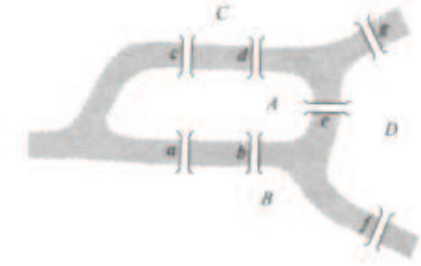


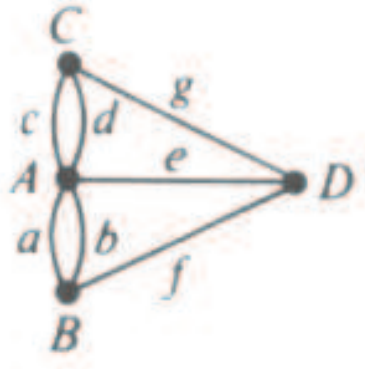
Los puentes de Königsberg

Königsberg es famosa por ser la ciudad natal de Immanuel Kant, pero también es famosa por sus siete puentes y por el problema que consistía en saber si una persona podría cruzar todos los puentes una sola vez, volviendo al lugar de donde partió.

Los puentes de Königsberg

Este problema fue resuelto por Euler en 1736, quien demostró que no era posible. Para eso modeló el problema como un problema de grafos: recorrer todas las aristas de un grafo una y solo una vez, volviendo al vértice inicial.

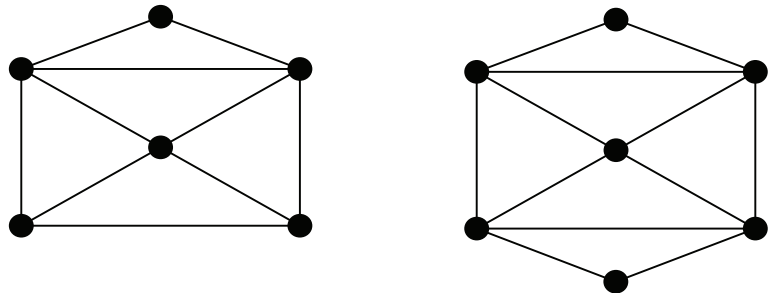




Definiciones

- Un camino Euleriano en un grafo o multigrafo G es un camino que recorre cada arista una y sólo una vez.
- Un circuito Euleriano en un grafo o multigrafo G es un circuito que recorre cada arista una y sólo una vez.
- Un grafo o multigrafo es Euleriano si tiene un circuito Euleriano.

Ej:



Obs: Si un grafo tiene al menos dos componentes conexas no triviales, no puede tener camino ni circuito Euleriano.

El Teorema de Euler

Teorema

Son equivalentes, para G conexo:

1. G es Euleriano.
2. Todo vértice de G tiene grado par.
3. Las aristas de G pueden particionarse en circuitos.

Teorema

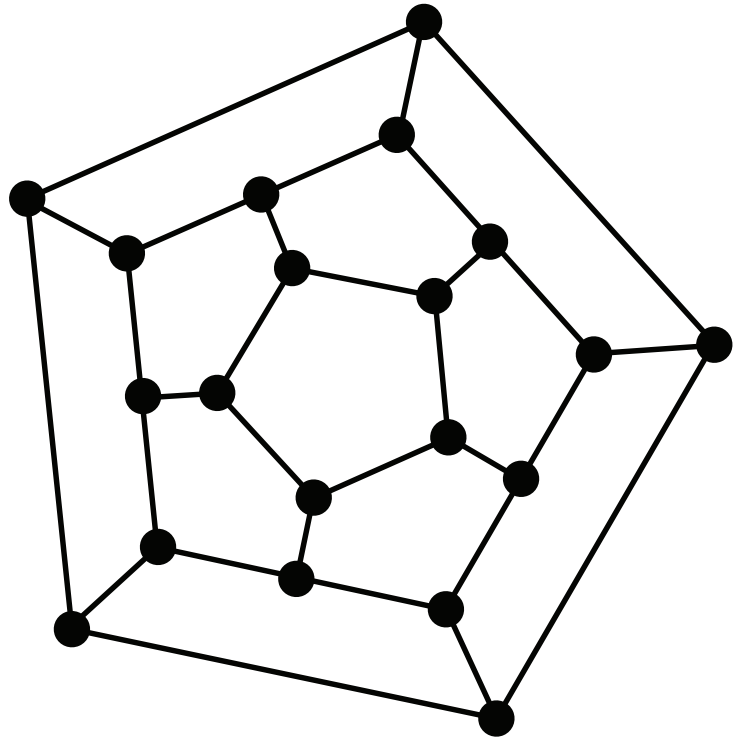
Un grafo conexo tiene un camino Euleriano si y sólo si todos sus vértices tienen grado par salvo dos.

- Obs: Los teoremas anteriores valen para multigrafos.

- En base a los teoremas anteriores, ¿cuál es la complejidad computacional de saber si un grafo es o no Euleriano?

El juego de Hamilton

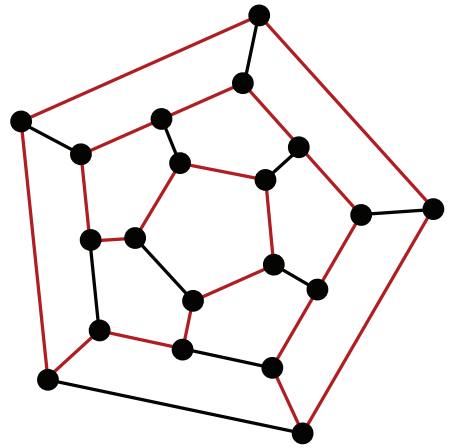
En 1859 Hamilton inventó un juego que consistía en encontrar un recorrido de todos los vértices de un dodecaedro sin repetir vértices y volviendo al original.



Definiciones

- Un camino Hamiltoniano en un grafo G es un camino que recorre cada vértice una y sólo una vez.
- Un circuito Hamiltoniano en un grafo G es un circuito que recorre cada vértice una y sólo una vez.
- Un grafo es Hamiltoniano si tiene un circuito Hamiltoniano.

Ej:



Obs: Si un grafo no es conexo, no puede tener camino ni circuito Hamiltoniano.

Proposición

Si G es Hamiltoniano, entonces no tiene puentes ni puntos de corte.

Teorema

El problema de saber si un grafo es Hamiltoniano es NP-completo.

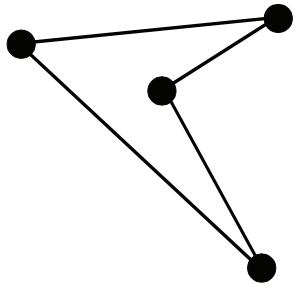
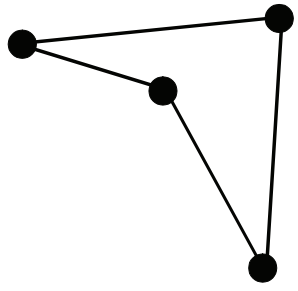
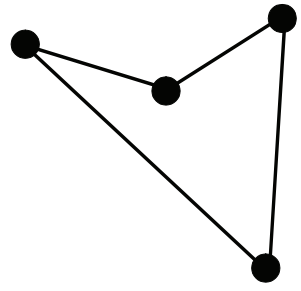
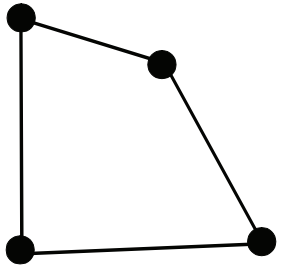
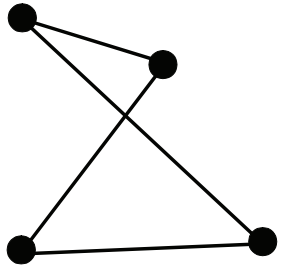
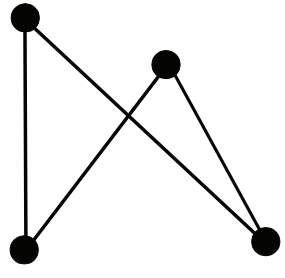
Ejercicio: Probar que el problema de saber si un grafo tiene un camino Hamiltoniano es NP-completo.

El problema del viajante de comercio (TSP)

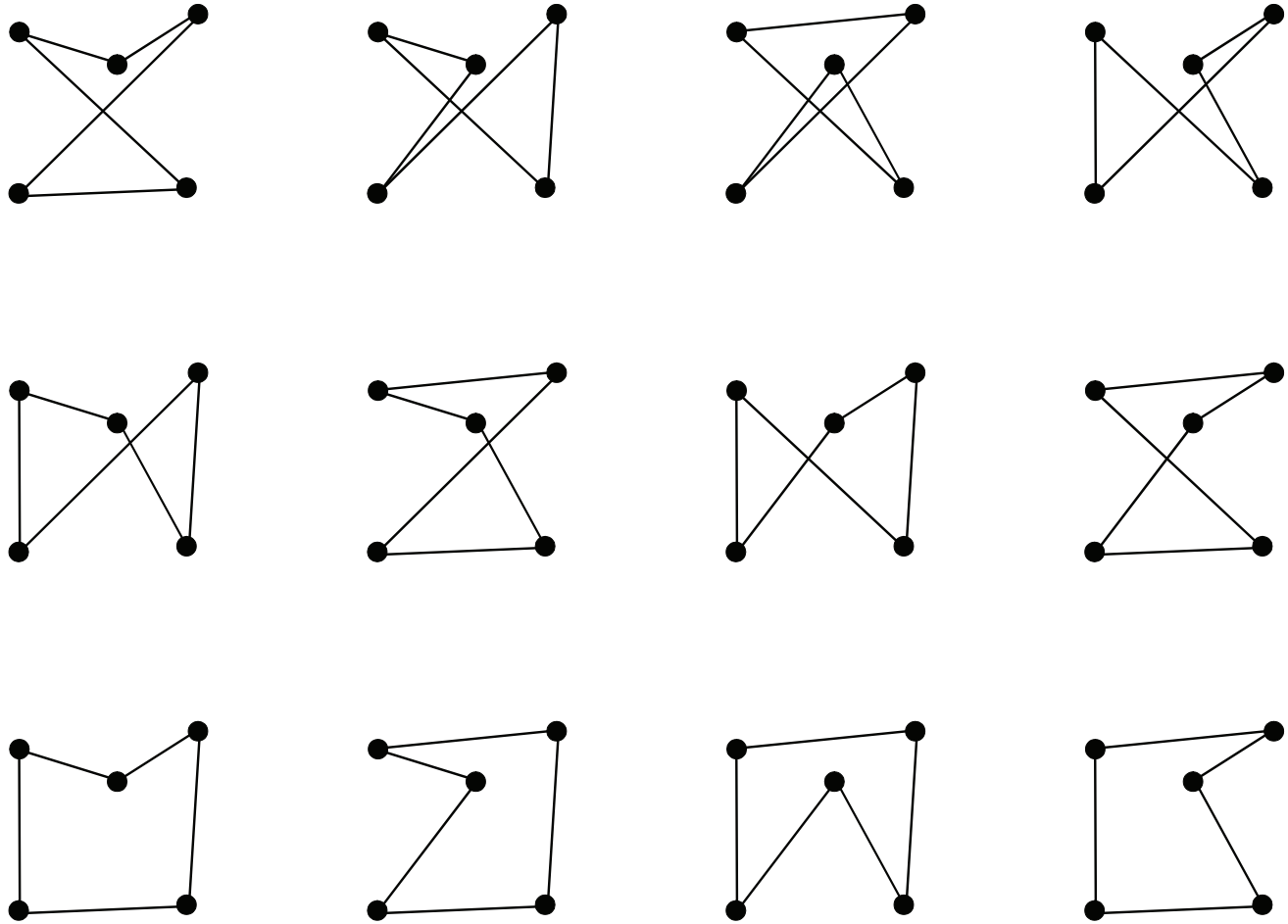
- El problema del viajante de comercio se trata de un viajante que debe recorrer lo más pronto posible cierta cantidad de ciudades y volver finalmente a la ciudad donde vive.
- En términos de grafos, es encontrar un camino Hamiltoniano de longitud mínima en un grafo completo con longitudes asociadas a sus aristas.
- En su versión de decisión, la entrada es un grafo completo G con longitudes asociadas a sus aristas y un número k, y la pregunta es "Existe en G un circuito Hamiltoniano de longitud $\leq k$?".

- Ejercicio: Demostrar que el problema del viajante de comercio es NP-completo.

Recorridos con cuatro ciudades

Recorridos con cinco ciudades



Recorridos con más de cinco ciudades

- ¿Cuántos recorridos tengo en un caso con 10 ciudades?

  $$181440$$

- ¿Cuántos recorridos tengo en un caso con 50 ciudades?

  30414093201713378043612608166064768844377641568960512000000000

- ¿Cuántos recorridos tengo en un caso con 100 ciudades?

  46663107721972076340849619428133350245357984132190810734296481947608799996614957804470731988078259143126848960413611879125592605458432000000000000000000000

- ¿Cuántos recorridos tengo en un caso con n ciudades?

  $$\frac{(n-1)!}{2}$$

Soluciones óptimas para el TSP

- En 1954 Dantzig, Fulkerson y Johnson resolvieron un caso de 49 ciudades del TSP.
- "Resolvieron" significa que D,F&J demostraron que la solución que presentaban era la mejor de un conjunto de 60 decillones de soluciones posibles.

Solución record (en 2001) de 15112 ciudades de Alemania

- Resuelta en una red de 110 máquinas en las universidades de Rice y Princeton, por Applegate, Bixby, Chvátal y Cook.
- Tiempo total de cómputo de 22.6 años de una PC de 500 MHz.
- Longitud total de aproximadamente 66.000 Km (Un poco más de una vuelta y media a la tierra por el ecuador).

Solución record (en 2004) de 24978 ciudades de Suecia

- Resuelta por Applegate, Bixby, Chvátal, Cook y Helsgaun.
- Longitud total de aproximadamente 72.500 Km.

Solución record actual (2005)

- Cook, Espinoza y Goycoolea: 33810 ciudades!

Unidad 2: Problemas de camino mínimo

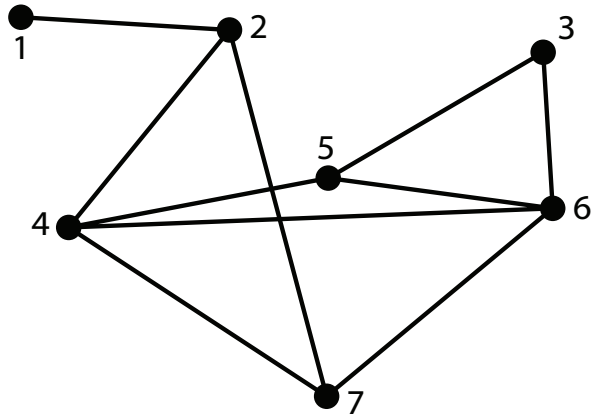
Representación de grafos

- Matriz de adyacencia
- Matriz de incidencia
- Listas de vecinos (para cada vértice, se indica la lista de sus vecinos).
- Cantidad de vértices y lista de aristas (se asumen los vértices numerados de 1 a n).

Matriz de adyacencia

Dado un grafo G cuyos vértices están numerados de 1 a n, definimos la matriz de adyacencia de G como $M \in \{0,1\}^{n \times n}$ donde $M(i,j) = 1$ si los vértices i y j son adyacentes y 0 en otro caso.

Ej:



$$\begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 & 0 \end{pmatrix}$$

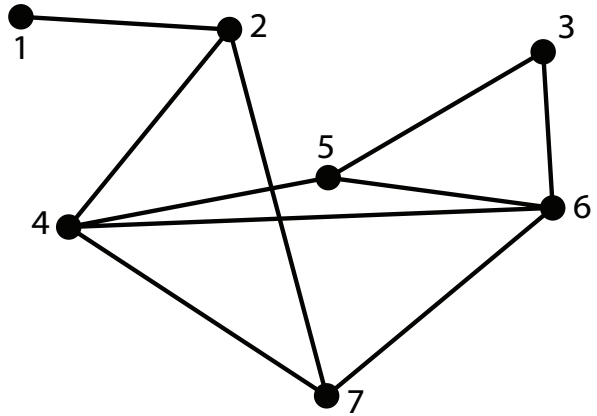
Propiedades de la matriz de adyacencia

- Si A_G es la matriz de adyacencia de G, entonces $A_G^k[i,j]$ es la cantidad de caminos (no necesariamente simples) distintos de longitud k entre i y j.
- En particular, $A_G^2[i,i] = d_G(i)$.
- Y además, un grafo G es bipartito si y sólo si $A_G^k[i,i] = 0$ para todo k impar, o sea, si la diagonal de la matriz $\sum_{j=0}^{n/2} A_G^{2j+1}$ tiene la diagonal nula.
- Un grafo G es conexo si y sólo si la matriz $\sum_{j=1}^{n} A_G^j$ no tiene ceros.

Matriz de incidencia

Numerando las aristas de G de 1 a m, definimos la matriz de incidencia de G como $M \in \{0,1\}^{m \times n}$ donde $M(i,j) = 1$ si el vértice j es uno de los extremos de la arista i y 0 en otro caso.

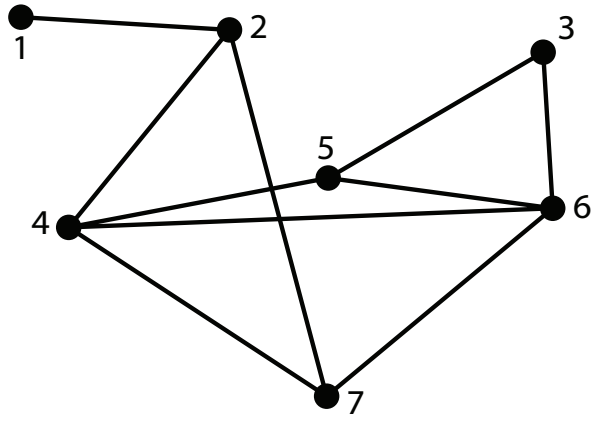
Ej:



$$\begin{pmatrix} 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 \end{pmatrix}$$

Listas de vecinos

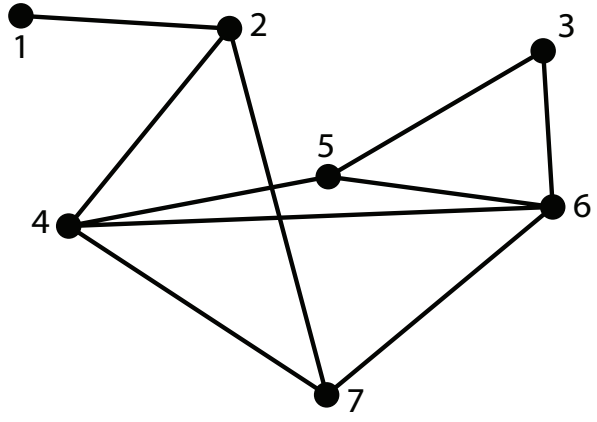
Ej:



$L_1 : 2$
$L_2 : 1 \longrightarrow 4 \longrightarrow 7$
$L_3 : 5 \longrightarrow 6$
$L_4 : 2 \longrightarrow 5 \longrightarrow 6 \longrightarrow 7$
$L_5 : 3 \longrightarrow 4 \longrightarrow 6$
$L_6 : 3 \longrightarrow 4 \longrightarrow 5 \longrightarrow 7$
$L_7 : 2 \longrightarrow 4 \longrightarrow 6$

Lista de aristas

Ej:



Cant. vértices: 7

Aristas: (1,2),(2,4),(2,7),(3,5),
(3,6),(4,5),(4,6),(4,7),(5,6),(6,7)

Es una estructura útil para el algoritmo de Kruskal, por ejemplo. También es la que se usa en general para almacenar un grafo en un archivo de texto.

Algoritmos para recorrido de grafos

Descripción del problema a grandes rasgos:

Tengo un grafo descripto de cierta forma y quiero recorrer sus vértices para:

- encontrar un vértice que cumpla determinada propiedad.
- calcular una propiedad de los vértices, por ejemplo la distancia a un vértice dado.
- calcular una propiedad del grafo, por ejemplo sus componentes conexas.
- obtener un orden de los vértices que cumpla determinada propiedad.

Estructuras de datos

Una pila (*stack*) es una estructura de datos donde el último en entrar es el primero en salir. Las tres operaciones básicas de una pila son:

- apilar (*push*), que coloca un objeto en la parte superior de la pila,
- tope (*top*) que permite ver el objeto superior de la pila y
- desapilar (*pop*), que elimina de la pila el tope.

Estructuras de datos

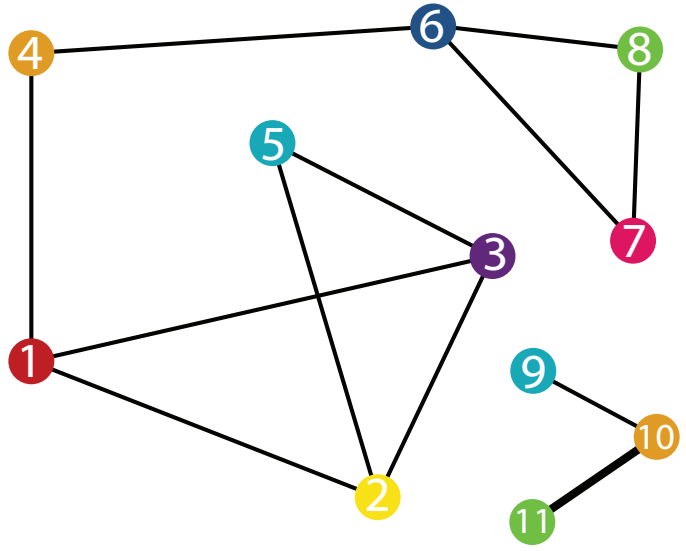
Una cola (*queue*) es una estructura de datos donde el primero en llegar es el primero en salir.
Las tres operaciones básicas de una cola son:

- encolar (*push*), que coloca un objeto al final de la cola,
- primero (*first*) que permite ver el primer objeto de la cola y
- desencolar (*pop*), que elimina de la cola el primer objeto.

Recorrido BFS

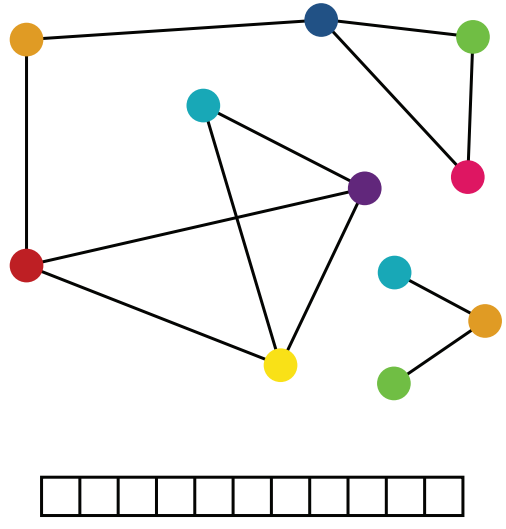
BFS: *Breadth-First Search*, o sea, recorrido a lo ancho. A partir de un vértice, recorre sus vecinos, luego los vecinos de sus vecinos, y así sucesivamente.... si al finalizar quedan vértices sin visitar, se repite a partir de un vértice no visitado.

Ejemplo:



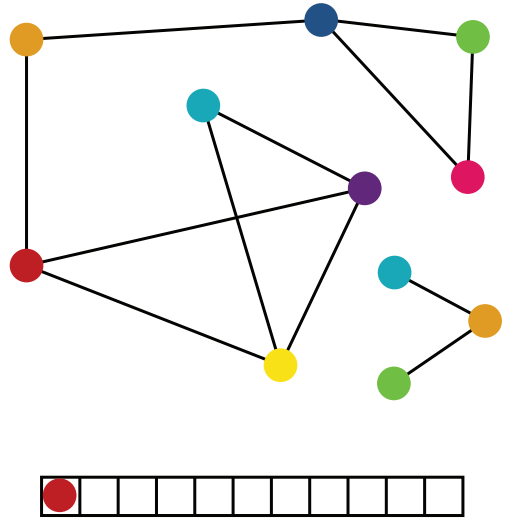
Recorrido BFS: pseudocódigo

```
cola C; vértices v, w; ord = 0;
Mientras queden vértices sin visitar
    v = elegir un vértice no visitado;
    marcar v;
    C.vaciar();
    C.encolar(v);
    Mientras C no sea vacía
        w = C.primero();
        ord++;
        orden(w)=ord;
        C.desencolar();
        Para cada vecino z de w
            Si z no está visitado
                marcar z;
                C.encolar(z);
```



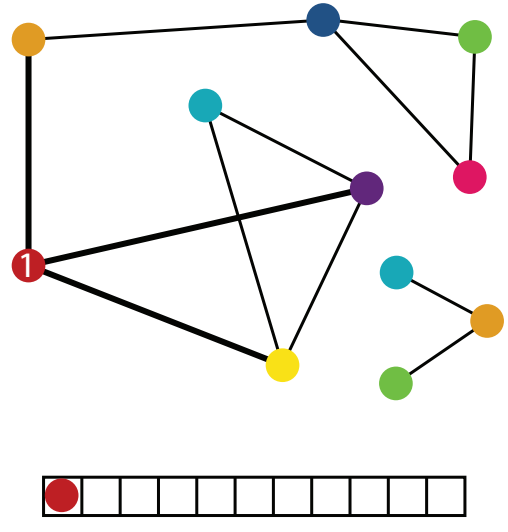
Recorrido BFS: pseudocódigo

```
cola C; vértices v, w; ord = 0;
Mientras queden vértices sin visitar
    v = elegir un vértice no visitado;
    marcar v;
    C.vaciar();
    C.encolar(v);
    Mientras C no sea vacía
        w = C.primero();
        ord++;
        orden(w)=ord;
        C.desencolar();
        Para cada vecino z de w
            Si z no está visitado
                marcar z;
                C.encolar(z);
```



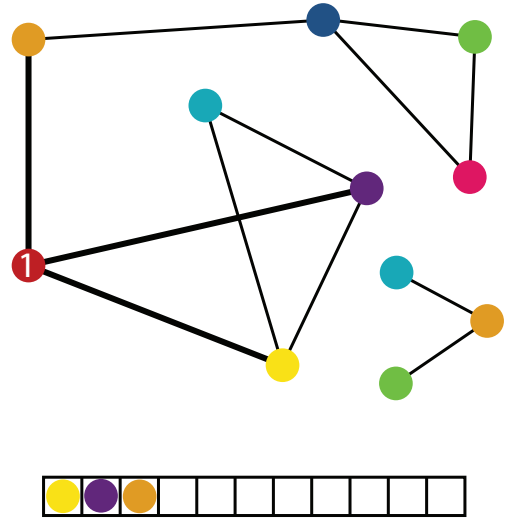
Recorrido BFS: pseudocódigo

```
cola C; vértices v, w; ord = 0;
Mientras queden vértices sin visitar
    v = elegir un vértice no visitado;
    marcar v;
    C.vaciar();
    C.encolar(v);
    Mientras C no sea vacía
        w = C.primero();
        ord++;
        orden(w)=ord;
        C.desencolar();
        Para cada vecino z de w
            Si z no está visitado
                marcar z;
                C.encolar(z);
```



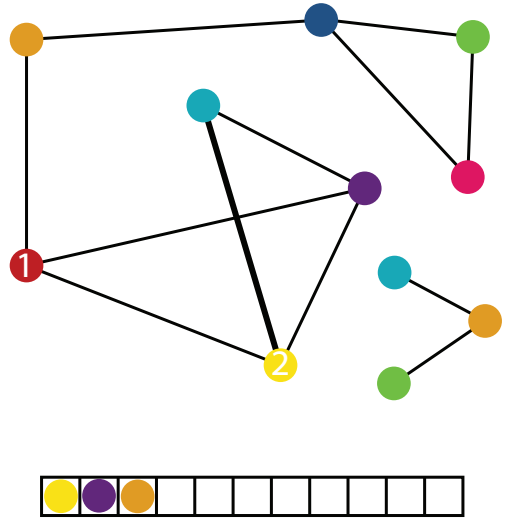
Recorrido BFS: pseudocódigo

```
cola C; vértices v, w; ord = 0;
Mientras queden vértices sin visitar
    v = elegir un vértice no visitado;
    marcar v;
    C.vaciar();
    C.encolar(v);
    Mientras C no sea vacía
        w = C.primero();
        ord++;
        orden(w)=ord;
        C.desencolar();
        Para cada vecino z de w
            Si z no está visitado
                marcar z;
                C.encolar(z);
```



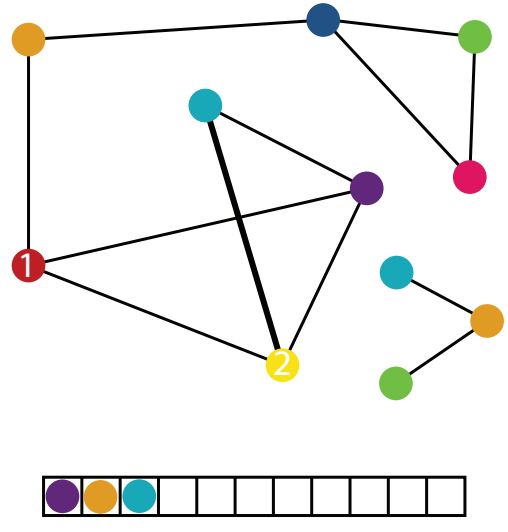
Recorrido BFS: pseudocódigo

```
cola C; vértices v, w; ord = 0;
Mientras queden vértices sin visitar
    v = elegir un vértice no visitado;
    marcar v;
    C.vaciar();
    C.encolar(v);
    Mientras C no sea vacía
        w = C.primero();
        ord++;
        orden(w)=ord;
        C.desencolar();
        Para cada vecino z de w
            Si z no está visitado
                marcar z;
                C.encolar(z);
```



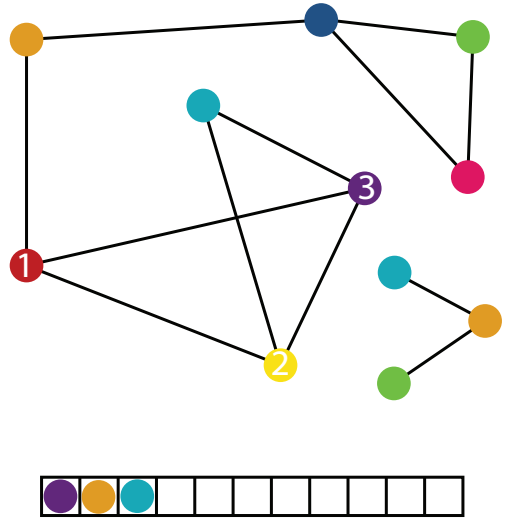
Recorrido BFS: pseudocódigo

```
cola C; vértices v, w; ord = 0;
Mientras queden vértices sin visitar
    v = elegir un vértice no visitado;
    marcar v;
    C.vaciar();
    C.encolar(v);
    Mientras C no sea vacía
        w = C.primero();
        ord++;
        orden(w)=ord;
        C.desencolar();
        Para cada vecino z de w
            Si z no está visitado
                marcar z;
                C.encolar(z);
```



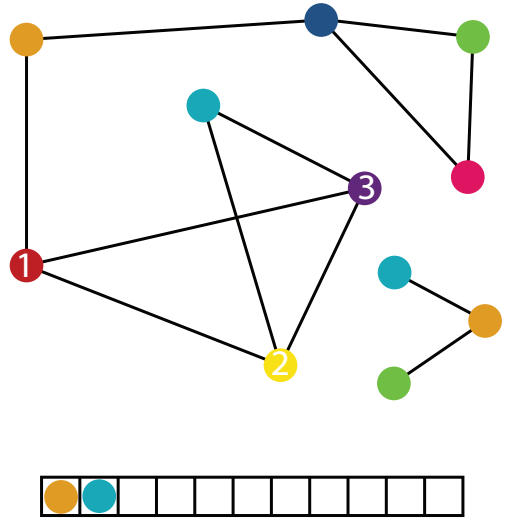
Recorrido BFS: pseudocódigo

```
cola C; vértices v, w; ord = 0;
Mientras queden vértices sin visitar
    v = elegir un vértice no visitado;
    marcar v;
    C.vaciar();
    C.encolar(v);
    Mientras C no sea vacía
        w = C.primero();
        ord++;
        orden(w)=ord;
        C.desencolar();
        Para cada vecino z de w
            Si z no está visitado
                marcar z;
                C.encolar(z);
```



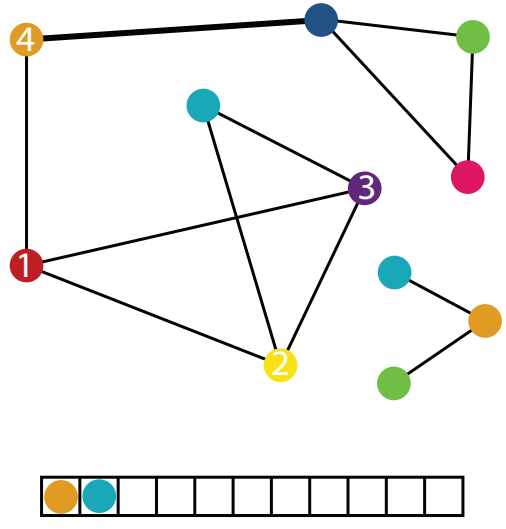
Recorrido BFS: pseudocódigo

```
cola C; vértices v, w; ord = 0;
Mientras queden vértices sin visitar
    v = elegir un vértice no visitado;
    marcar v;
    C.vaciar();
    C.encolar(v);
    Mientras C no sea vacía
        w = C.primero();
        ord++;
        orden(w)=ord;
        C.desencolar();
        Para cada vecino z de w
            Si z no está visitado
                marcar z;
                C.encolar(z);
```



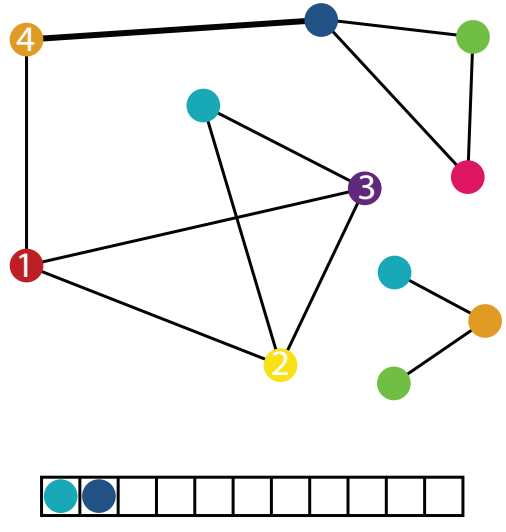
Recorrido BFS: pseudocódigo

```
cola C; vértices v, w; ord = 0;
Mientras queden vértices sin visitar
    v = elegir un vértice no visitado;
    marcar v;
    C.vaciar();
    C.encolar(v);
    Mientras C no sea vacía
        w = C.primero();
        ord++;
        orden(w)=ord;
        C.desencolar();
        Para cada vecino z de w
            Si z no está visitado
                marcar z;
                C.encolar(z);
```



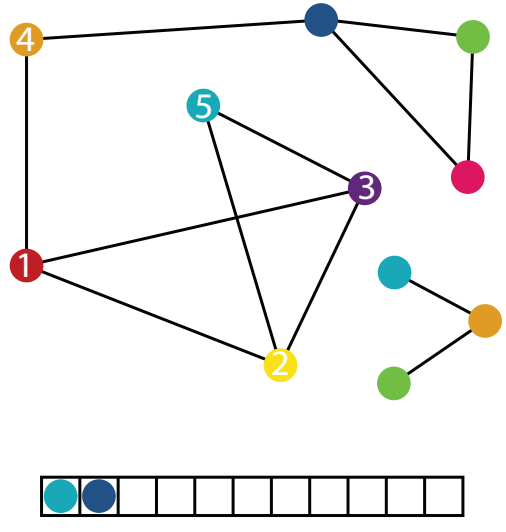
Recorrido BFS: pseudocódigo

```
cola C; vértices v, w; ord = 0;
Mientras queden vértices sin visitar
    v = elegir un vértice no visitado;
    marcar v;
    C.vaciar();
    C.encolar(v);
    Mientras C no sea vacía
        w = C.primero();
        ord++;
        orden(w)=ord;
        C.desencolar();
        Para cada vecino z de w
            Si z no está visitado
                marcar z;
                C.encolar(z);
```



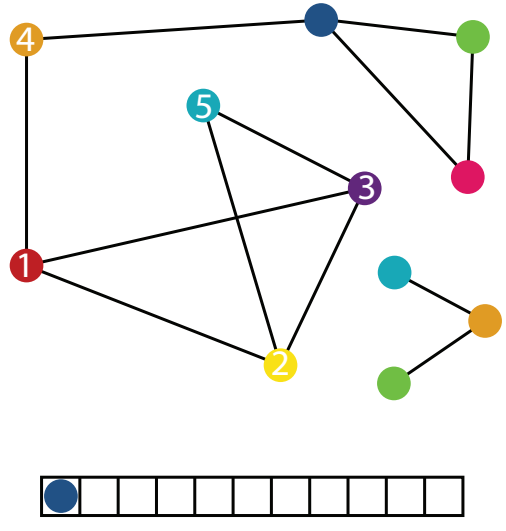
Recorrido BFS: pseudocódigo

```
cola C; vértices v, w; ord = 0;
Mientras queden vértices sin visitar
    v = elegir un vértice no visitado;
    marcar v;
    C.vaciar();
    C.encolar(v);
    Mientras C no sea vacía
        w = C.primero();
        ord++;
        orden(w)=ord;
        C.desencolar();
        Para cada vecino z de w
            Si z no está visitado
                marcar z;
                C.encolar(z);
```



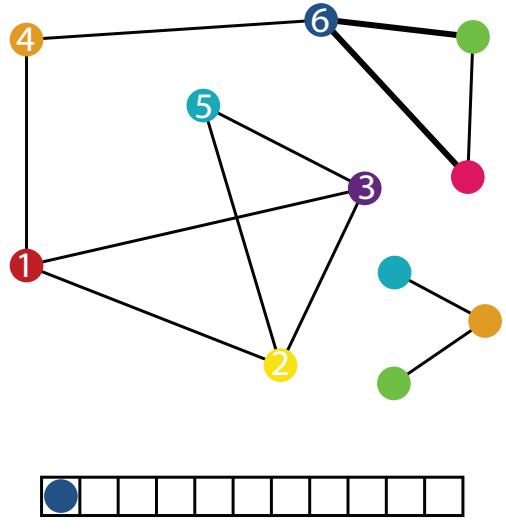
Recorrido BFS: pseudocódigo

```
cola C; vértices v, w; ord = 0;
Mientras queden vértices sin visitar
    v = elegir un vértice no visitado;
    marcar v;
    C.vaciar();
    C.encolar(v);
    Mientras C no sea vacía
        w = C.primero();
        ord++;
        orden(w)=ord;
        C.desencolar();
        Para cada vecino z de w
            Si z no está visitado
                marcar z;
                C.encolar(z);
```



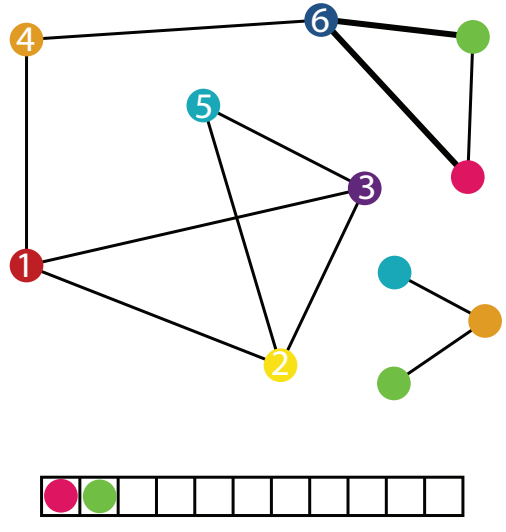
Recorrido BFS: pseudocódigo

```
cola C; vértices v, w; ord = 0;
Mientras queden vértices sin visitar
    v = elegir un vértice no visitado;
    marcar v;
    C.vaciar();
    C.encolar(v);
    Mientras C no sea vacía
        w = C.primero();
        ord++;
        orden(w)=ord;
        C.desencolar();
        Para cada vecino z de w
            Si z no está visitado
                marcar z;
                C.encolar(z);
```



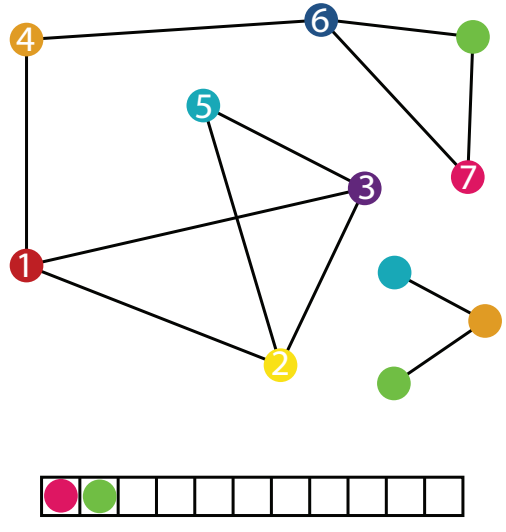
Recorrido BFS: pseudocódigo

```
cola C; vértices v, w; ord = 0;
Mientras queden vértices sin visitar
    v = elegir un vértice no visitado;
    marcar v;
    C.vaciar();
    C.encolar(v);
    Mientras C no sea vacía
        w = C.primero();
        ord++;
        orden(w)=ord;
        C.desencolar();
        Para cada vecino z de w
            Si z no está visitado
                marcar z;
                C.encolar(z);
```



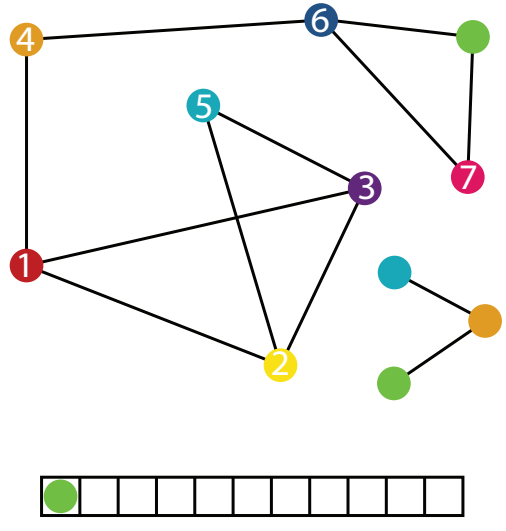
Recorrido BFS: pseudocódigo

```
cola C; vértices v, w; ord = 0;
Mientras queden vértices sin visitar
    v = elegir un vértice no visitado;
    marcar v;
    C.vaciar();
    C.encolar(v);
    Mientras C no sea vacía
        w = C.primero();
        ord++;
        orden(w)=ord;
        C.desencolar();
        Para cada vecino z de w
            Si z no está visitado
                marcar z;
                C.encolar(z);
```



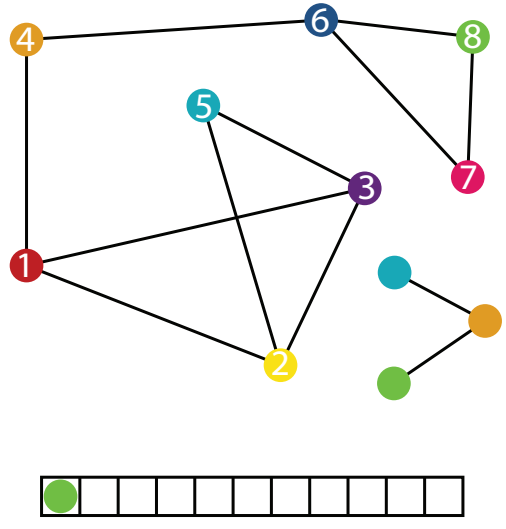
Recorrido BFS: pseudocódigo

```
cola C; vértices v, w; ord = 0;
Mientras queden vértices sin visitar
    v = elegir un vértice no visitado;
    marcar v;
    C.vaciar();
    C.encolar(v);
    Mientras C no sea vacía
        w = C.primero();
        ord++;
        orden(w)=ord;
        C.desencolar();
        Para cada vecino z de w
            Si z no está visitado
                marcar z;
                C.encolar(z);
```



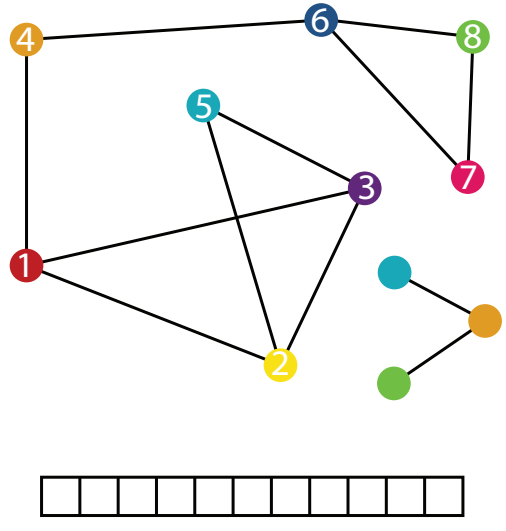
Recorrido BFS: pseudocódigo

```
cola C; vértices v, w; ord = 0;
Mientras queden vértices sin visitar
    v = elegir un vértice no visitado;
    marcar v;
    C.vaciar();
    C.encolar(v);
    Mientras C no sea vacía
        w = C.primero();
        ord++;
        orden(w)=ord;
        C.desencolar();
        Para cada vecino z de w
            Si z no está visitado
                marcar z;
                C.encolar(z);
```



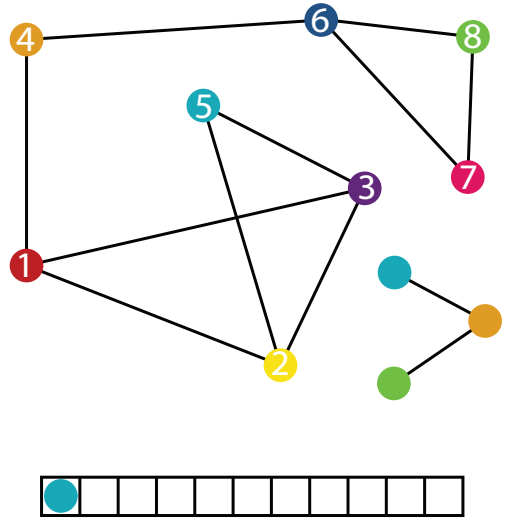
Recorrido BFS: pseudocódigo

```
cola C; vértices v, w; ord = 0;
Mientras queden vértices sin visitar
    v = elegir un vértice no visitado;
    marcar v;
    C.vaciar();
    C.encolar(v);
    Mientras C no sea vacía
        w = C.primero();
        ord++;
        orden(w)=ord;
        C.desencolar();
        Para cada vecino z de w
            Si z no está visitado
                marcar z;
                C.encolar(z);
```



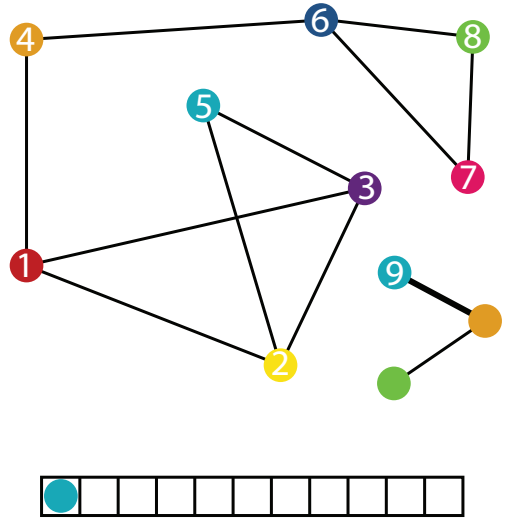
Recorrido BFS: pseudocódigo

```
cola C; vértices v, w; ord = 0;
Mientras queden vértices sin visitar
    v = elegir un vértice no visitado;
    marcar v;
    C.vaciar();
    C.encolar(v);
    Mientras C no sea vacía
        w = C.primero();
        ord++;
        orden(w)=ord;
        C.desencolar();
        Para cada vecino z de w
            Si z no está visitado
                marcar z;
                C.encolar(z);
```



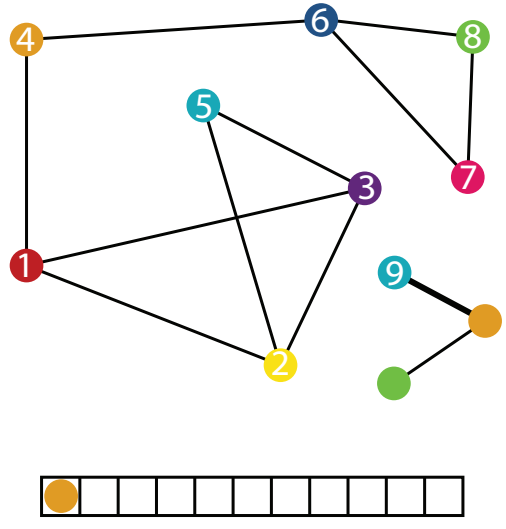
Recorrido BFS: pseudocódigo

```
cola C; vértices v, w; ord = 0;
Mientras queden vértices sin visitar
    v = elegir un vértice no visitado;
    marcar v;
    C.vaciar();
    C.encolar(v);
    Mientras C no sea vacía
        w = C.primero();
        ord++;
        orden(w)=ord;
        C.desencolar();
        Para cada vecino z de w
            Si z no está visitado
                marcar z;
                C.encolar(z);
```



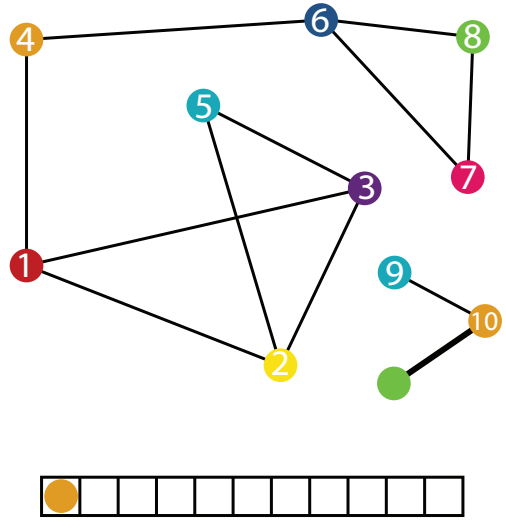
Recorrido BFS: pseudocódigo

```
cola C; vértices v, w; ord = 0;
Mientras queden vértices sin visitar
    v = elegir un vértice no visitado;
    marcar v;
    C.vaciar();
    C.encolar(v);
    Mientras C no sea vacía
        w = C.primero();
        ord++;
        orden(w)=ord;
        C.desencolar();
        Para cada vecino z de w
            Si z no está visitado
                marcar z;
                C.encolar(z);
```



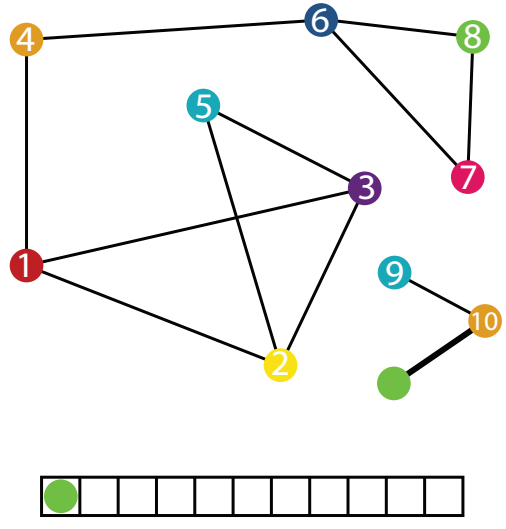
Recorrido BFS: pseudocódigo

```
cola C; vértices v, w; ord = 0;
Mientras queden vértices sin visitar
    v = elegir un vértice no visitado;
    marcar v;
    C.vaciar();
    C.encolar(v);
    Mientras C no sea vacía
        w = C.primero();
        ord++;
        orden(w)=ord;
        C.desencolar();
        Para cada vecino z de w
            Si z no está visitado
                marcar z;
                C.encolar(z);
```



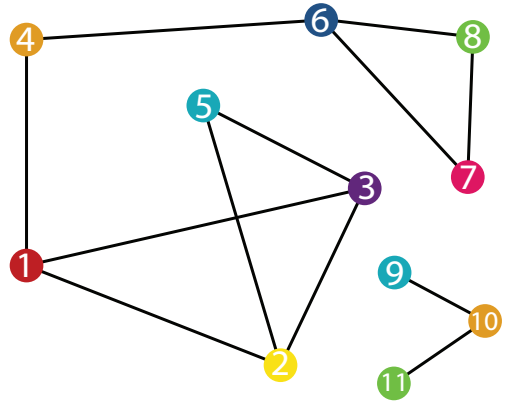
Recorrido BFS: pseudocódigo

```
cola C; vértices v, w; ord = 0;
Mientras queden vértices sin visitar
    v = elegir un vértice no visitado;
    marcar v;
    C.vaciar();
    C.encolar(v);
    Mientras C no sea vacía
        w = C.primero();
        ord++;
        orden(w)=ord;
        C.desencolar();
        Para cada vecino z de w
            Si z no está visitado
                marcar z;
                C.encolar(z);
```



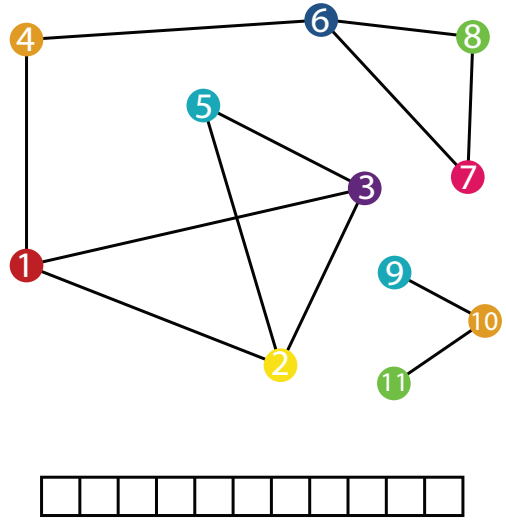
Recorrido BFS: pseudocódigo

```
cola C; vértices v, w; ord = 0;
Mientras queden vértices sin visitar
    v = elegir un vértice no visitado;
    marcar v;
    C.vaciar();
    C.encolar(v);
    Mientras C no sea vacía
        w = C.primero();
        ord++;
        orden(w)=ord;
        C.desencolar();
        Para cada vecino z de w
            Si z no está visitado
                marcar z;
                C.encolar(z);
```



Recorrido BFS: pseudocódigo

```
cola C; vértices v, w; ord = 0;
Mientras queden vértices sin visitar
    v = elegir un vértice no visitado;
    marcar v;
    C.vaciar();
    C.encolar(v);
    Mientras C no sea vacía
        w = C.primero();
        ord++;
        orden(w)=ord;
        C.desencolar();
        Para cada vecino z de w
            Si z no está visitado
                marcar z;
                C.encolar(z);
```



Recorrido BFS: propiedades

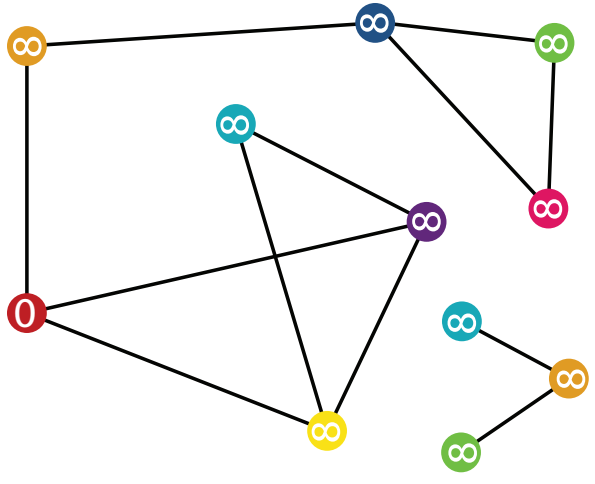
Notemos que cada vez que comenzamos desde un vértice nuevo, es porque los vértices que quedan sin visitar no son alcanzables desde los vértices visitados. Entonces lo que estamos encontrando son las diferentes componentes conexas del grafo.

Si ponemos un contador que se incremente luego de la operación *elegir un vértice no visitado*, lo que obtenemos es la cantidad de componentes conexas del grafo (y con algunas leves modificaciones podemos obtener las componentes en si).

Recorrido BFS: propiedades

Si partimos desde un vértice v, modificando levemente el código podemos calcular para cada vértice su distancia a v. Inicialmente todos los vértices tienen rótulo ∞. Rotulamos el vértice v con 0 y luego, al visitar los vecinos de cada vértice w que aún tienen rótulo ∞, los rotulamos con el rótulo de w más 1. (Notemos que los que no sean alcanzados desde v, o sea, no pertenecen a la componente conexa de v, tienen distancia infinita a v.)

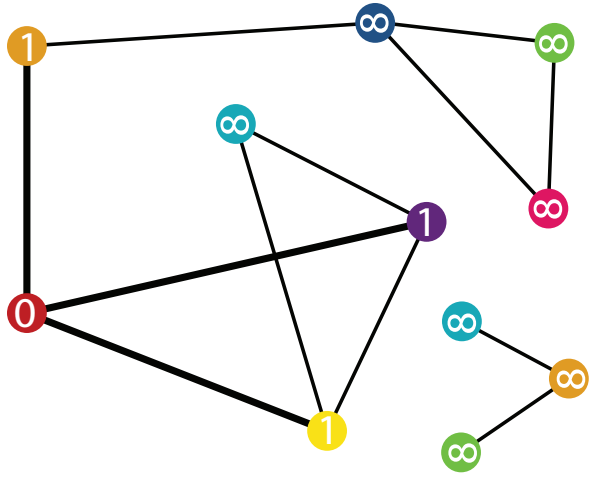
Ejemplo:



Recorrido BFS: propiedades

Si partimos desde un vértice v, modificando levemente el código podemos calcular para cada vértice su distancia a v. Inicialmente todos los vértices tienen rótulo ∞. Rotulamos el vértice v con 0 y luego, al visitar los vecinos de cada vértice w que aún tienen rótulo ∞, los rotulamos con el rótulo de w más 1. (Notemos que los que no sean alcanzados desde v, o sea, no pertenecen a la componente conexa de v, tienen distancia infinita a v.)

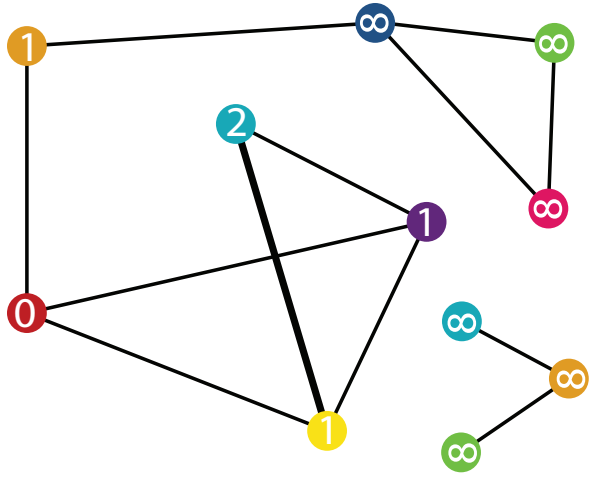
Ejemplo:



Recorrido BFS: propiedades

Si partimos desde un vértice v, modificando levemente el código podemos calcular para cada vértice su distancia a v. Inicialmente todos los vértices tienen rótulo ∞. Rotulamos el vértice v con 0 y luego, al visitar los vecinos de cada vértice w que aún tienen rótulo ∞, los rotulamos con el rótulo de w más 1. (Notemos que los que no sean alcanzados desde v, o sea, no pertenecen a la componente conexa de v, tienen distancia infinita a v.)

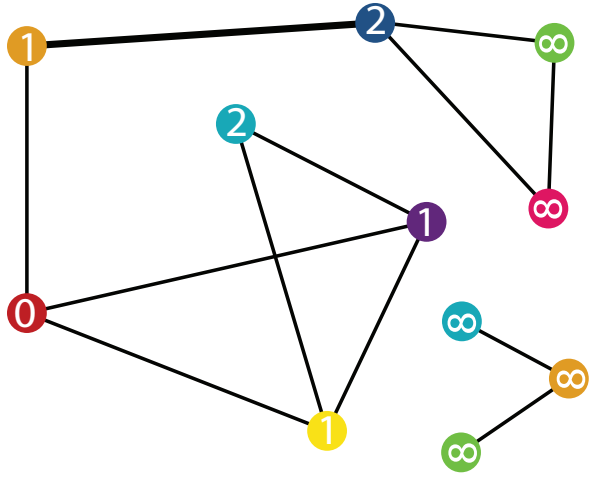
Ejemplo:



Recorrido BFS: propiedades

Si partimos desde un vértice v, modificando levemente el código podemos calcular para cada vértice su distancia a v. Inicialmente todos los vértices tienen rótulo ∞. Rotulamos el vértice v con 0 y luego, al visitar los vecinos de cada vértice w que aún tienen rótulo ∞, los rotulamos con el rótulo de w más 1. (Notemos que los que no sean alcanzados desde v, o sea, no pertenecen a la componente conexa de v, tienen distancia infinita a v.)

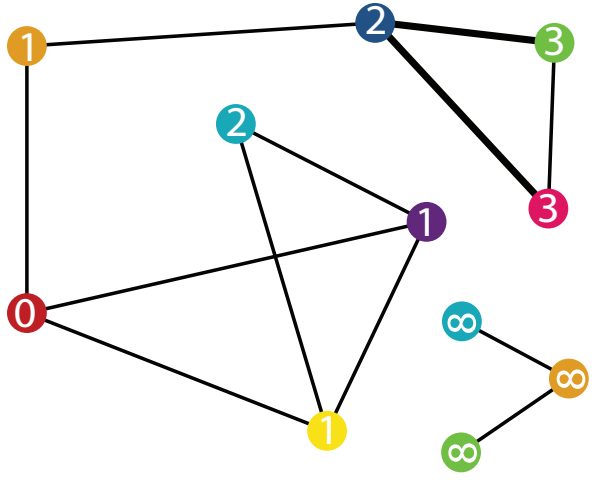
Ejemplo:



Recorrido BFS: propiedades

Si partimos desde un vértice v, modificando levemente el código podemos calcular para cada vértice su distancia a v. Inicialmente todos los vértices tienen rótulo ∞. Rotulamos el vértice v con 0 y luego, al visitar los vecinos de cada vértice w que aún tienen rótulo ∞, los rotulamos con el rótulo de w más 1. (Notemos que los que no sean alcanzados desde v, o sea, no pertenecen a la componente conexa de v, tienen distancia infinita a v.)

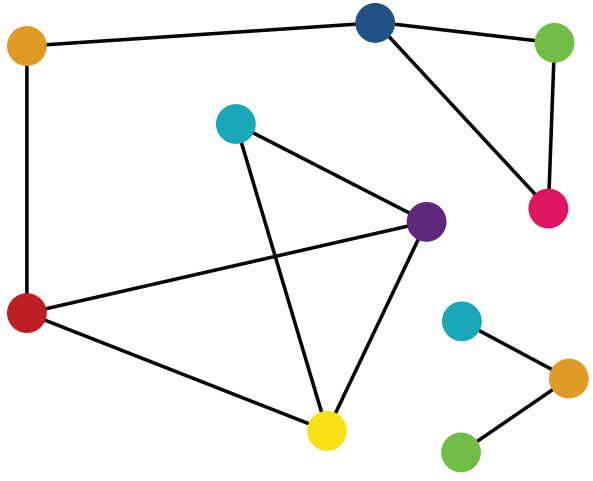
Ejemplo:



Recorrido BFS: reconocimiento de grafos bipartitos

Utilizamos una idea parecida a la del algoritmo anterior en cada componente conexa: rotulamos el vértice v con 0 y luego, al visitar los vecinos de cada vértice w que aún no tienen rótulo, los rotulamos con 1 menos el rótulo de w. De esta manera queda determinada la partición, de ser bipartito el grafo. Queda entonces verificar que ningún par de vértices con el mismo rótulo sean adyacentes (ese chequeo se puede ir haciendo en cada paso).

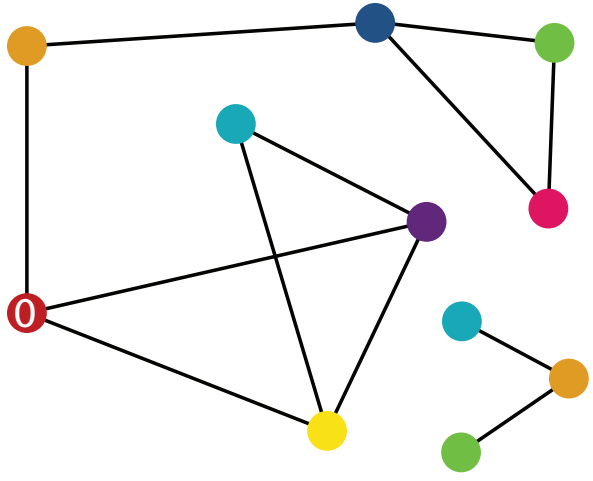
Ejemplo:



Recorrido BFS: reconocimiento de grafos bipartitos

Utilizamos una idea parecida a la del algoritmo anterior en cada componente conexa: rotulamos el vértice v con 0 y luego, al visitar los vecinos de cada vértice w que aún no tienen rótulo, los rotulamos con 1 menos el rótulo de w. De esta manera queda determinada la partición, de ser bipartito el grafo. Queda entonces verificar que ningún par de vértices con el mismo rótulo sean adyacentes (ese chequeo se puede ir haciendo en cada paso).

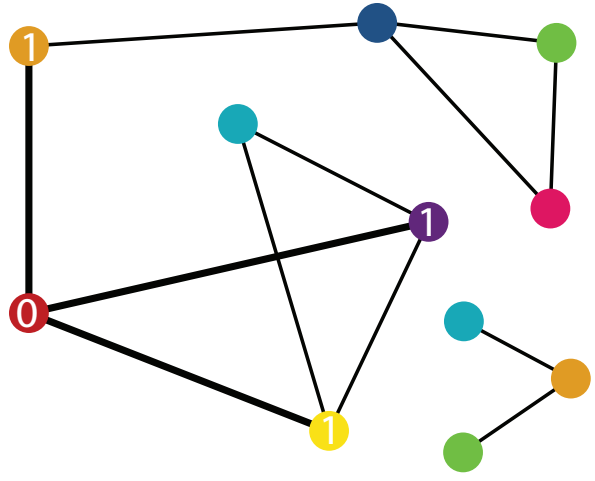
Ejemplo:



Recorrido BFS: reconocimiento de grafos bipartitos

Utilizamos una idea parecida a la del algoritmo anterior en cada componente conexa: rotulamos el vértice v con 0 y luego, al visitar los vecinos de cada vértice w que aún no tienen rótulo, los rotulamos con 1 menos el rótulo de w. De esta manera queda determinada la partición, de ser bipartito el grafo. Queda entonces verificar que ningún par de vértices con el mismo rótulo sean adyacentes (ese chequeo se puede ir haciendo en cada paso).

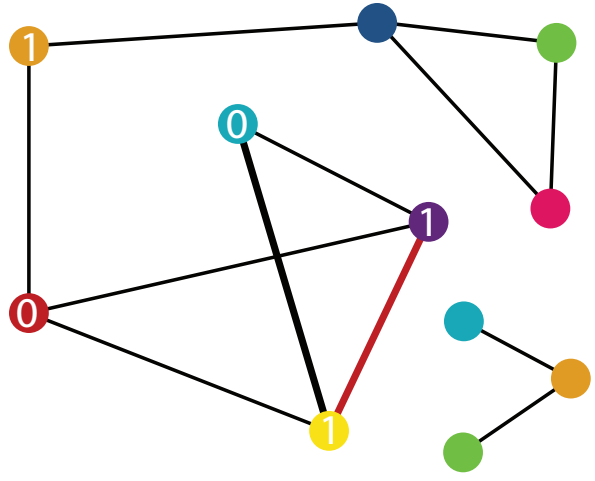
Ejemplo:



Recorrido BFS: reconocimiento de grafos bipartitos

Utilizamos una idea parecida a la del algoritmo anterior en cada componente conexa: rotulamos el vértice v con 0 y luego, al visitar los vecinos de cada vértice w que aún no tienen rótulo, los rotulamos con 1 menos el rótulo de w. De esta manera queda determinada la partición, de ser bipartito el grafo. Queda entonces verificar que ningún par de vértices con el mismo rótulo sean adyacentes (ese chequeo se puede ir haciendo en cada paso).

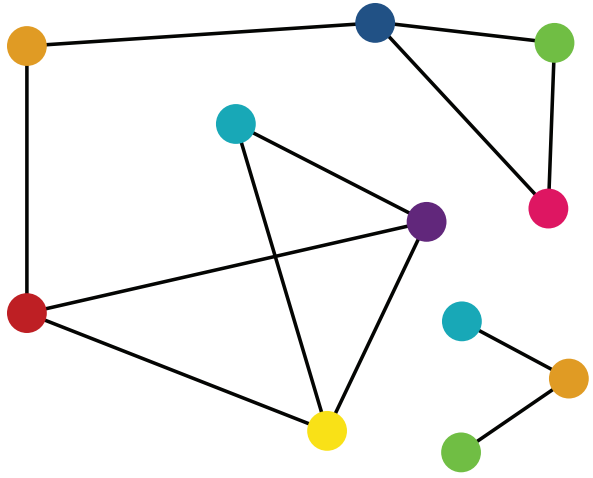
Ejemplo:



Recorrido DFS

DFS: *Depth-First Search*, o sea, recorrido en profundidad. A partir de un vértice, comienza por alguno de sus vecinos, luego un vecino de éste, y así sucesivamente hasta llegar a un vértice que ya no tiene vecinos sin visitar. Sigue entonces por algún vecino del último vértice visitado que aún tiene vecinos sin visitar.
Igual que en BFS, si al finalizar quedan vértices sin visitar, se repite a partir de un vértice no visitado.

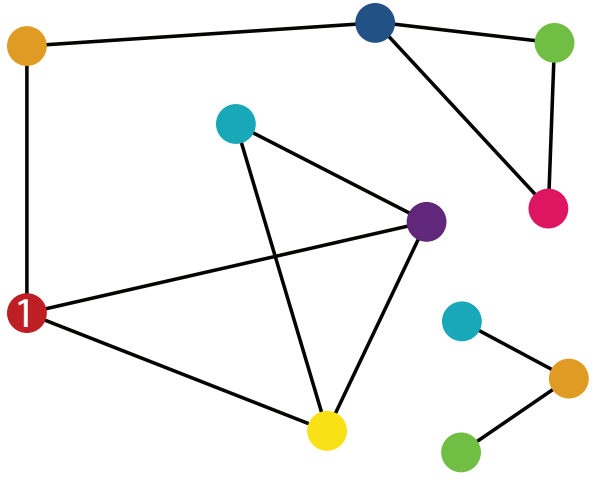
Ejemplo:



Recorrido DFS

DFS: *Depth-First Search*, o sea, recorrido en profundidad. A partir de un vértice, comienza por alguno de sus vecinos, luego un vecino de éste, y así sucesivamente hasta llegar a un vértice que ya no tiene vecinos sin visitar. Sigue entonces por algún vecino del último vértice visitado que aún tiene vecinos sin visitar.
Igual que en BFS, si al finalizar quedan vértices sin visitar, se repite a partir de un vértice no visitado.

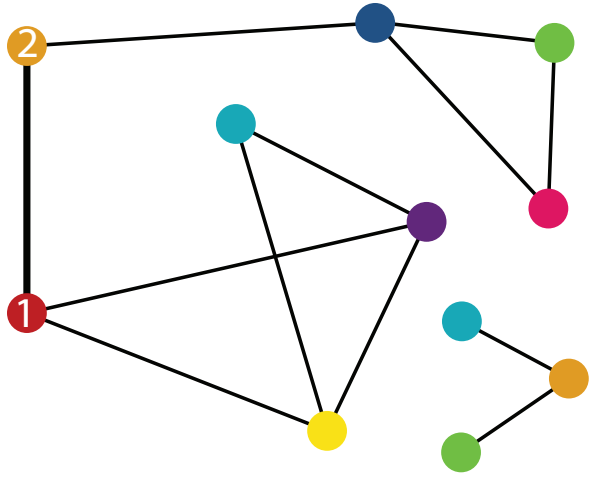
Ejemplo:



Recorrido DFS

DFS: *Depth-First Search*, o sea, recorrido en profundidad. A partir de un vértice, comienza por alguno de sus vecinos, luego un vecino de éste, y así sucesivamente hasta llegar a un vértice que ya no tiene vecinos sin visitar. Sigue entonces por algún vecino del último vértice visitado que aún tiene vecinos sin visitar.
Igual que en BFS, si al finalizar quedan vértices sin visitar, se repite a partir de un vértice no visitado.

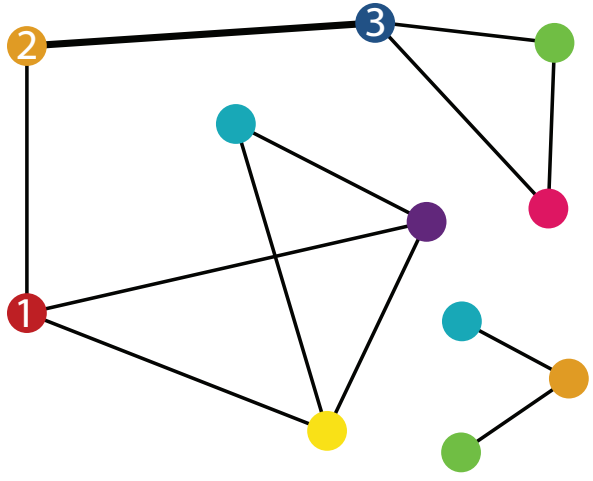
Ejemplo:



Recorrido DFS

DFS: *Depth-First Search*, o sea, recorrido en profundidad. A partir de un vértice, comienza por alguno de sus vecinos, luego un vecino de éste, y así sucesivamente hasta llegar a un vértice que ya no tiene vecinos sin visitar. Sigue entonces por algún vecino del último vértice visitado que aún tiene vecinos sin visitar.
Igual que en BFS, si al finalizar quedan vértices sin visitar, se repite a partir de un vértice no visitado.

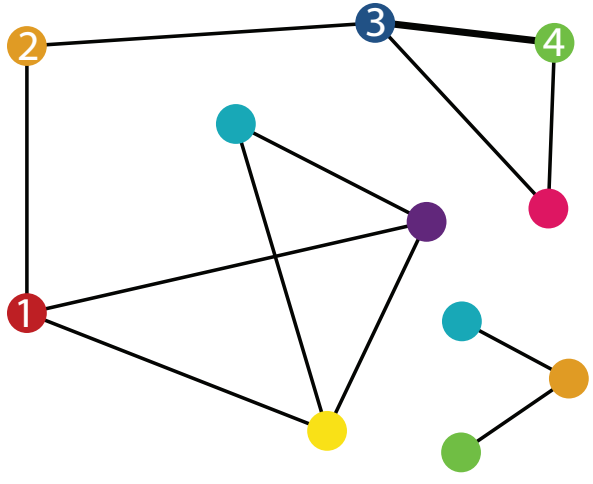
Ejemplo:



Recorrido DFS

DFS: *Depth-First Search*, o sea, recorrido en profundidad. A partir de un vértice, comienza por alguno de sus vecinos, luego un vecino de éste, y así sucesivamente hasta llegar a un vértice que ya no tiene vecinos sin visitar. Sigue entonces por algún vecino del último vértice visitado que aún tiene vecinos sin visitar.
Igual que en BFS, si al finalizar quedan vértices sin visitar, se repite a partir de un vértice no visitado.

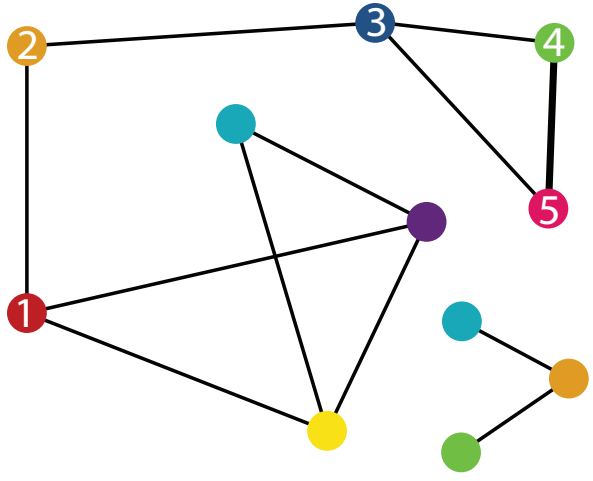
Ejemplo:



Recorrido DFS

DFS: *Depth-First Search*, o sea, recorrido en profundidad. A partir de un vértice, comienza por alguno de sus vecinos, luego un vecino de éste, y así sucesivamente hasta llegar a un vértice que ya no tiene vecinos sin visitar. Sigue entonces por algún vecino del último vértice visitado que aún tiene vecinos sin visitar.
Igual que en BFS, si al finalizar quedan vértices sin visitar, se repite a partir de un vértice no visitado.

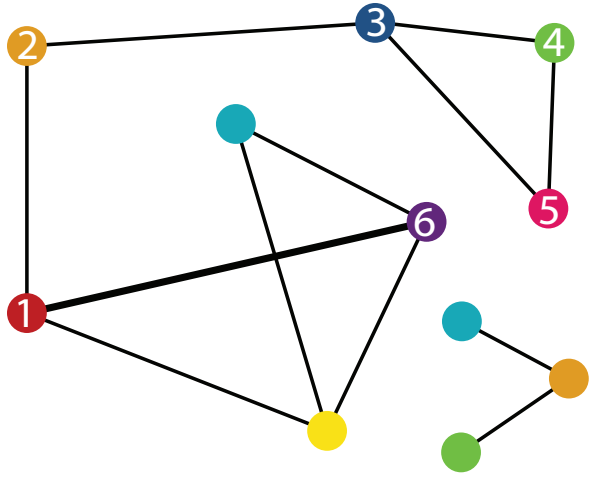
Ejemplo:



Recorrido DFS

DFS: *Depth-First Search*, o sea, recorrido en profundidad. A partir de un vértice, comienza por alguno de sus vecinos, luego un vecino de éste, y así sucesivamente hasta llegar a un vértice que ya no tiene vecinos sin visitar. Sigue entonces por algún vecino del último vértice visitado que aún tiene vecinos sin visitar.
Igual que en BFS, si al finalizar quedan vértices sin visitar, se repite a partir de un vértice no visitado.

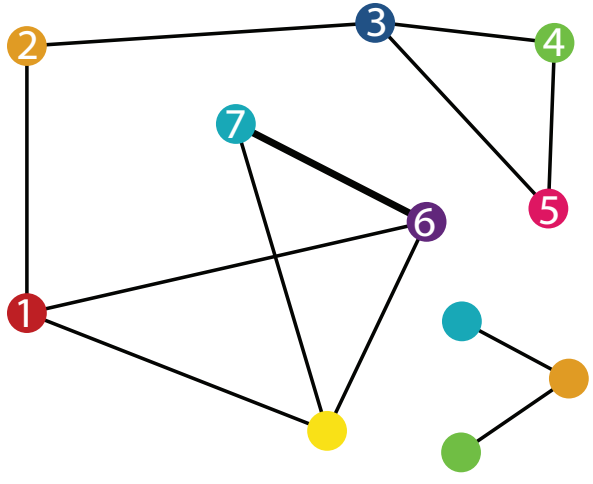
Ejemplo:



Recorrido DFS

DFS: *Depth-First Search*, o sea, recorrido en profundidad. A partir de un vértice, comienza por alguno de sus vecinos, luego un vecino de éste, y así sucesivamente hasta llegar a un vértice que ya no tiene vecinos sin visitar. Sigue entonces por algún vecino del último vértice visitado que aún tiene vecinos sin visitar.
Igual que en BFS, si al finalizar quedan vértices sin visitar, se repite a partir de un vértice no visitado.

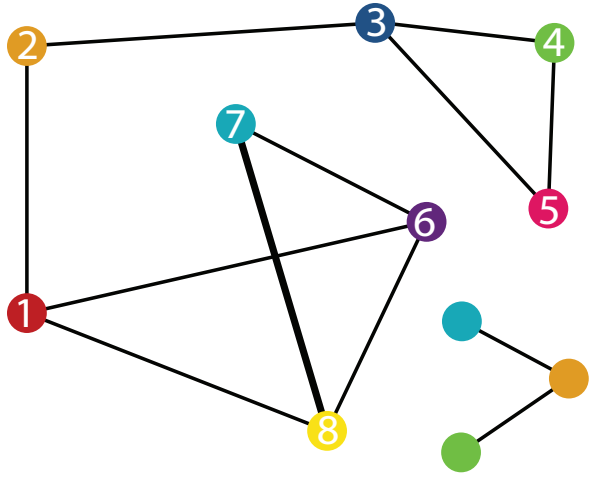
Ejemplo:



Recorrido DFS

DFS: *Depth-First Search*, o sea, recorrido en profundidad. A partir de un vértice, comienza por alguno de sus vecinos, luego un vecino de éste, y así sucesivamente hasta llegar a un vértice que ya no tiene vecinos sin visitar. Sigue entonces por algún vecino del último vértice visitado que aún tiene vecinos sin visitar.
Igual que en BFS, si al finalizar quedan vértices sin visitar, se repite a partir de un vértice no visitado.

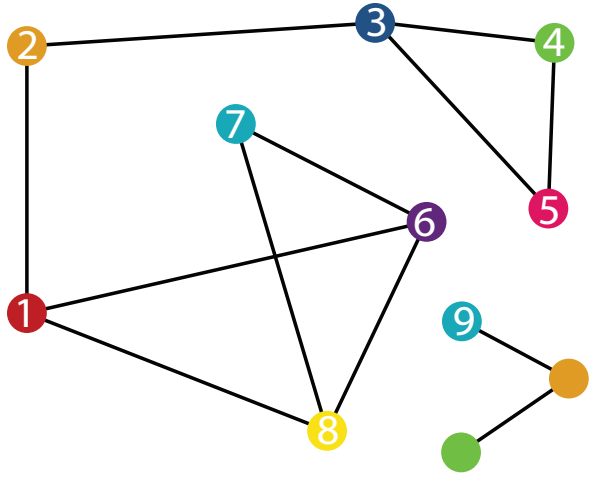
Ejemplo:



Recorrido DFS

DFS: *Depth-First Search*, o sea, recorrido en profundidad. A partir de un vértice, comienza por alguno de sus vecinos, luego un vecino de éste, y así sucesivamente hasta llegar a un vértice que ya no tiene vecinos sin visitar. Sigue entonces por algún vecino del último vértice visitado que aún tiene vecinos sin visitar.
Igual que en BFS, si al finalizar quedan vértices sin visitar, se repite a partir de un vértice no visitado.

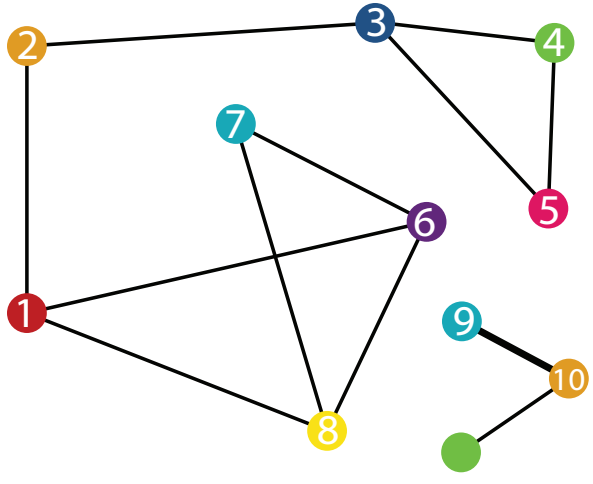
Ejemplo:



Recorrido DFS

DFS: *Depth-First Search*, o sea, recorrido en profundidad. A partir de un vértice, comienza por alguno de sus vecinos, luego un vecino de éste, y así sucesivamente hasta llegar a un vértice que ya no tiene vecinos sin visitar. Sigue entonces por algún vecino del último vértice visitado que aún tiene vecinos sin visitar.
Igual que en BFS, si al finalizar quedan vértices sin visitar, se repite a partir de un vértice no visitado.

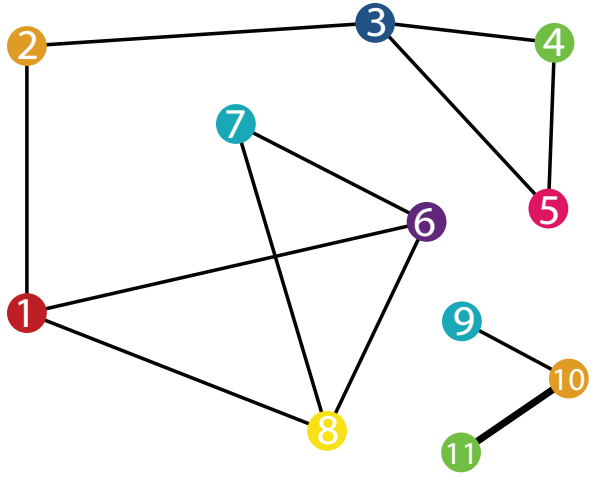
Ejemplo:



Recorrido DFS

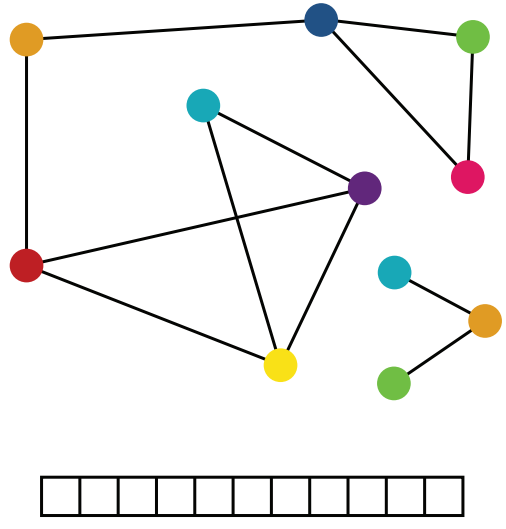
DFS: *Depth-First Search*, o sea, recorrido en profundidad. A partir de un vértice, comienza por alguno de sus vecinos, luego un vecino de éste, y así sucesivamente hasta llegar a un vértice que ya no tiene vecinos sin visitar. Sigue entonces por algún vecino del último vértice visitado que aún tiene vecinos sin visitar.
Igual que en BFS, si al finalizar quedan vértices sin visitar, se repite a partir de un vértice no visitado.

Ejemplo:



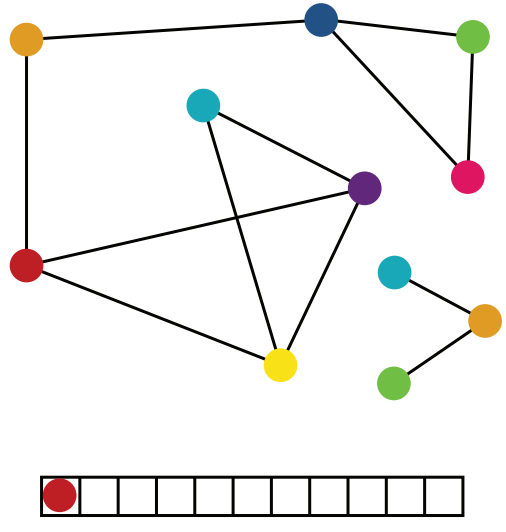
Recorrido DFS: pseudocódigo

```
pila P; vértices v, w; ord = 0;
Mientras queden vértices sin visitar
    v = elegir un vértice no visitado;
    marcar v;
    P.vaciar();
    P.apilar(v);
    Mientras P no sea vacía
        w = P.tope();
        ord++;
        orden(w)=ord;
        P.desapilar();
        Para cada vecino z de w
            Si z no está visitado
                marcar z;
                P.apilar(z);
```



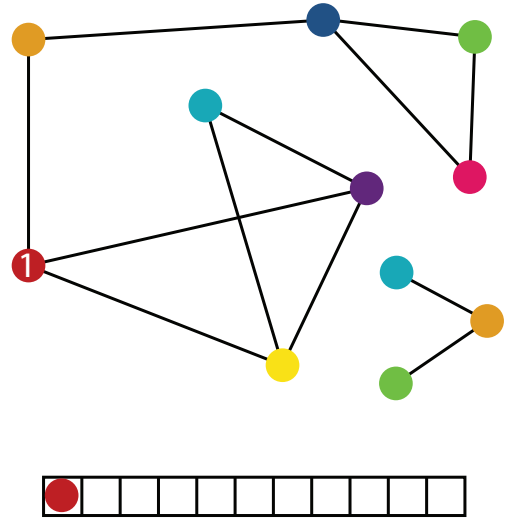
Recorrido DFS: pseudocódigo

```
pila P; vértices v, w; ord = 0;
Mientras queden vértices sin visitar
    v = elegir un vértice no visitado;
    marcar v;
    P.vaciar();
    P.apilar(v);
    Mientras P no sea vacía
        w = P.tope();
        ord++;
        orden(w)=ord;
        P.desapilar();
        Para cada vecino z de w
            Si z no está visitado
                marcar z;
                P.apilar(z);
```



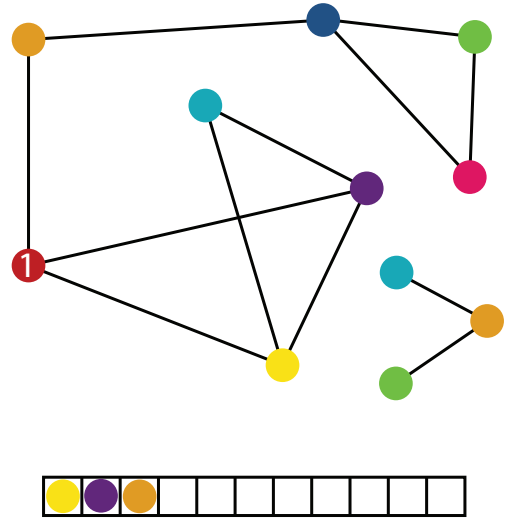
Recorrido DFS: pseudocódigo

```
pila P; vértices v, w; ord = 0;
Mientras queden vértices sin visitar
    v = elegir un vértice no visitado;
    marcar v;
    P.vaciar();
    P.apilar(v);
    Mientras P no sea vacía
        w = P.tope();
        ord++;
        orden(w)=ord;
        P.desapilar();
        Para cada vecino z de w
            Si z no está visitado
                marcar z;
                P.apilar(z);
```



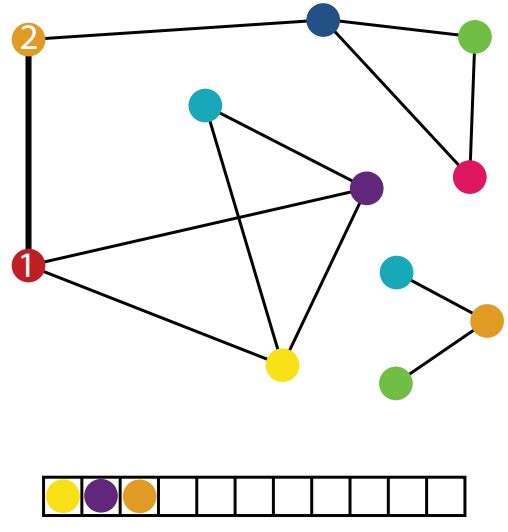
Recorrido DFS: pseudocódigo

```
pila P; vértices v, w; ord = 0;
Mientras queden vértices sin visitar
    v = elegir un vértice no visitado;
    marcar v;
    P.vaciar();
    P.apilar(v);
    Mientras P no sea vacía
        w = P.tope();
        ord++;
        orden(w)=ord;
        P.desapilar();
        Para cada vecino z de w
            Si z no está visitado
                marcar z;
                P.apilar(z);
```



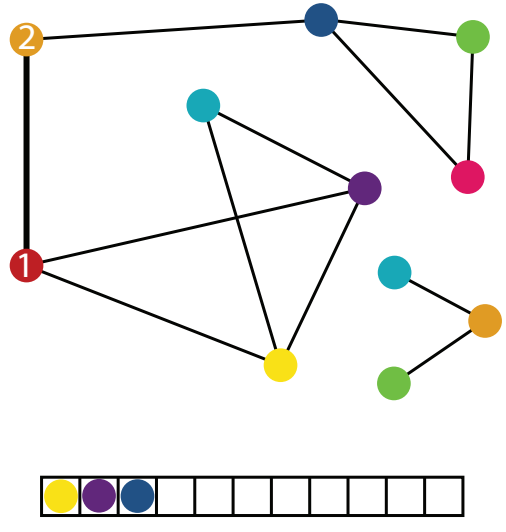
Recorrido DFS: pseudocódigo

```
pila P; vértices v, w; ord = 0;
Mientras queden vértices sin visitar
    v = elegir un vértice no visitado;
    marcar v;
    P.vaciar();
    P.apilar(v);
    Mientras P no sea vacía
        w = P.tope();
        ord++;
        orden(w)=ord;
        P.desapilar();
        Para cada vecino z de w
            Si z no está visitado
                marcar z;
                P.apilar(z);
```



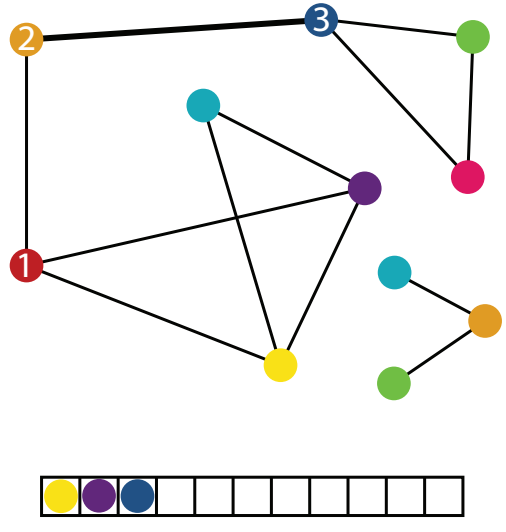
Recorrido DFS: pseudocódigo

```
pila P; vértices v, w; ord = 0;
Mientras queden vértices sin visitar
    v = elegir un vértice no visitado;
    marcar v;
    P.vaciar();
    P.apilar(v);
    Mientras P no sea vacía
        w = P.tope();
        ord++;
        orden(w)=ord;
        P.desapilar();
        Para cada vecino z de w
            Si z no está visitado
                marcar z;
                P.apilar(z);
```



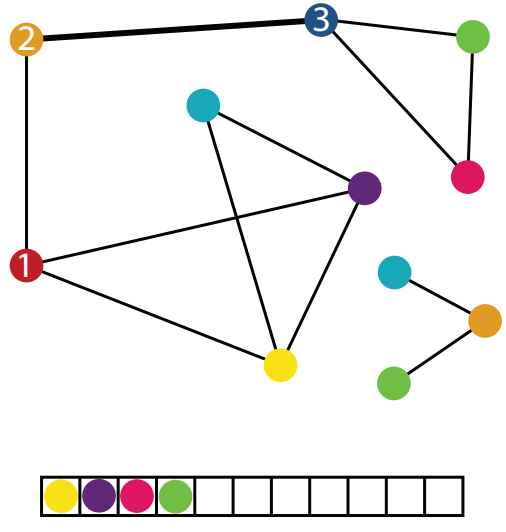
Recorrido DFS: pseudocódigo

```
pila P; vértices v, w; ord = 0;
Mientras queden vértices sin visitar
    v = elegir un vértice no visitado;
    marcar v;
    P.vaciar();
    P.apilar(v);
    Mientras P no sea vacía
        w = P.tope();
        ord++;
        orden(w)=ord;
        P.desapilar();
        Para cada vecino z de w
            Si z no está visitado
                marcar z;
                P.apilar(z);
```



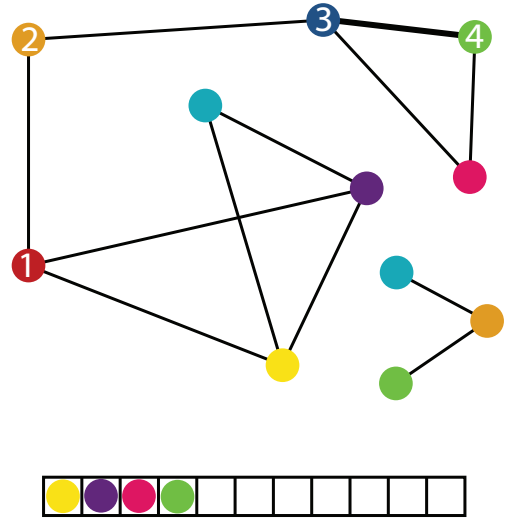
Recorrido DFS: pseudocódigo

```
pila P; vértices v, w; ord = 0;
Mientras queden vértices sin visitar
    v = elegir un vértice no visitado;
    marcar v;
    P.vaciar();
    P.apilar(v);
    Mientras P no sea vacía
        w = P.tope();
        ord++;
        orden(w)=ord;
        P.desapilar();
        Para cada vecino z de w
            Si z no está visitado
                marcar z;
                P.apilar(z);
```



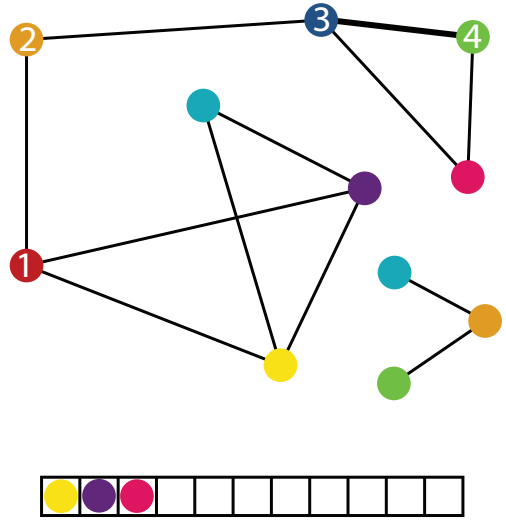
Recorrido DFS: pseudocódigo

```
pila P; vértices v, w; ord = 0;
Mientras queden vértices sin visitar
    v = elegir un vértice no visitado;
    marcar v;
    P.vaciar();
    P.apilar(v);
    Mientras P no sea vacía
        w = P.tope();
        ord++;
        orden(w)=ord;
        P.desapilar();
        Para cada vecino z de w
            Si z no está visitado
                marcar z;
                P.apilar(z);
```



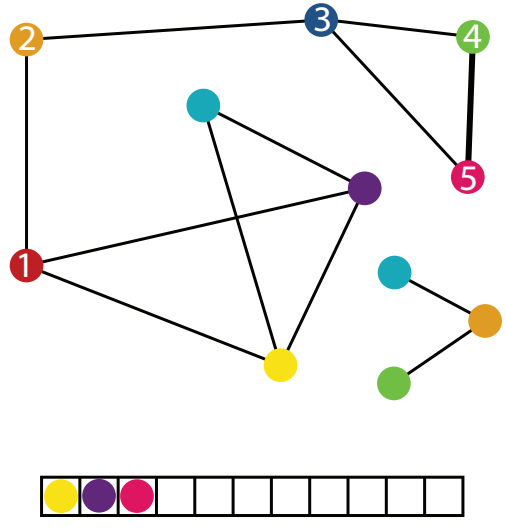
Recorrido DFS: pseudocódigo

```
pila P; vértices v, w; ord = 0;
Mientras queden vértices sin visitar
    v = elegir un vértice no visitado;
    marcar v;
    P.vaciar();
    P.apilar(v);
    Mientras P no sea vacía
        w = P.tope();
        ord++;
        orden(w)=ord;
        P.desapilar();
        Para cada vecino z de w
            Si z no está visitado
                marcar z;
                P.apilar(z);
```



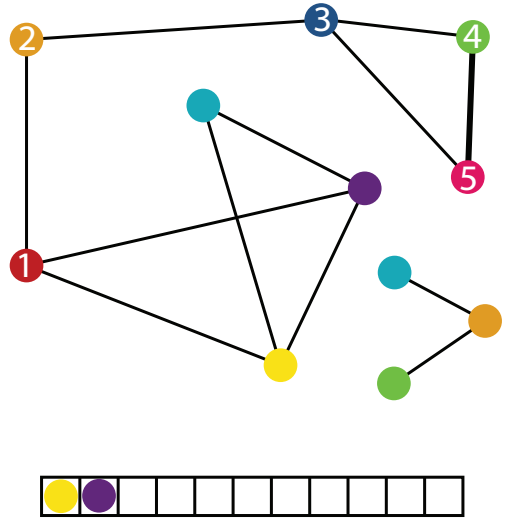
Recorrido DFS: pseudocódigo

```
pila P; vértices v, w; ord = 0;
Mientras queden vértices sin visitar
    v = elegir un vértice no visitado;
    marcar v;
    P.vaciar();
    P.apilar(v);
    Mientras P no sea vacía
        w = P.tope();
        ord++;
        orden(w)=ord;
        P.desapilar();
        Para cada vecino z de w
            Si z no está visitado
                marcar z;
                P.apilar(z);
```



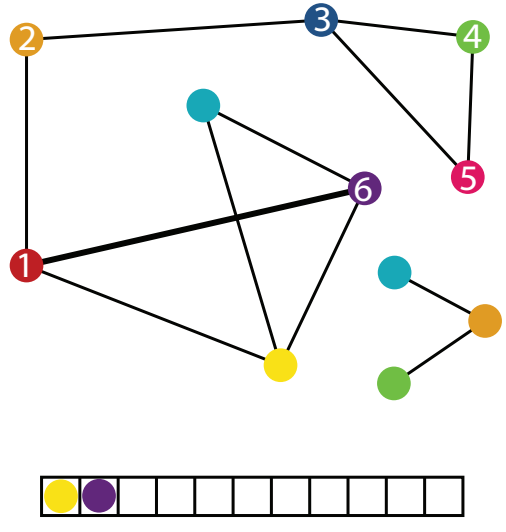
Recorrido DFS: pseudocódigo

```
pila P; vértices v, w; ord = 0;
Mientras queden vértices sin visitar
    v = elegir un vértice no visitado;
    marcar v;
    P.vaciar();
    P.apilar(v);
    Mientras P no sea vacía
        w = P.tope();
        ord++;
        orden(w)=ord;
        P.desapilar();
        Para cada vecino z de w
            Si z no está visitado
                marcar z;
                P.apilar(z);
```



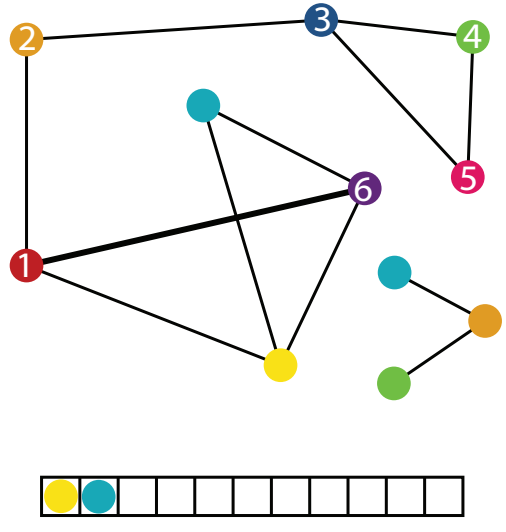
Recorrido DFS: pseudocódigo

```
pila P; vértices v, w; ord = 0;
Mientras queden vértices sin visitar
    v = elegir un vértice no visitado;
    marcar v;
    P.vaciar();
    P.apilar(v);
    Mientras P no sea vacía
        w = P.tope();
        ord++;
        orden(w)=ord;
        P.desapilar();
        Para cada vecino z de w
            Si z no está visitado
                marcar z;
                P.apilar(z);
```



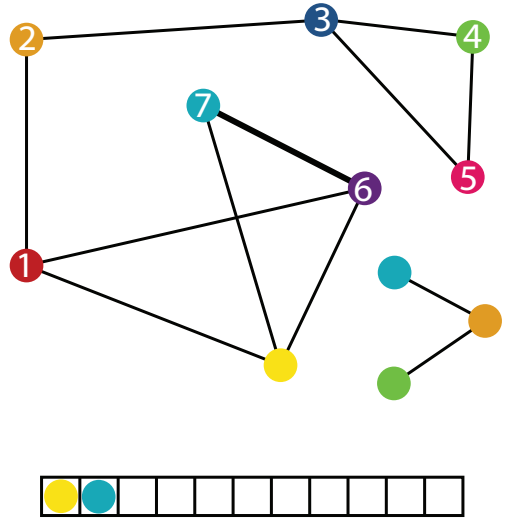
Recorrido DFS: pseudocódigo

```
pila P; vértices v, w; ord = 0;
Mientras queden vértices sin visitar
    v = elegir un vértice no visitado;
    marcar v;
    P.vaciar();
    P.apilar(v);
    Mientras P no sea vacía
        w = P.tope();
        ord++;
        orden(w)=ord;
        P.desapilar();
        Para cada vecino z de w
            Si z no está visitado
                marcar z;
                P.apilar(z);
```



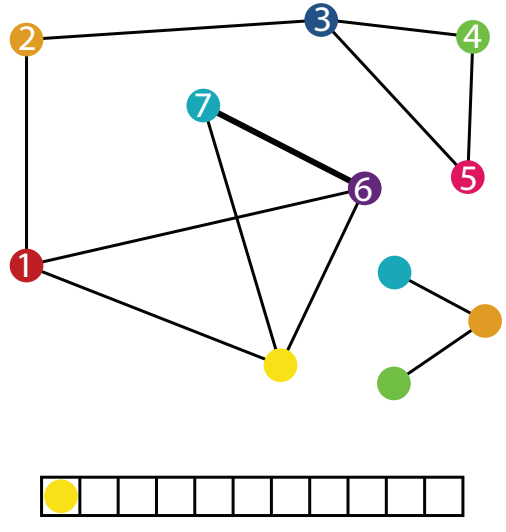
Recorrido DFS: pseudocódigo

```
pila P; vértices v, w; ord = 0;
Mientras queden vértices sin visitar
    v = elegir un vértice no visitado;
    marcar v;
    P.vaciar();
    P.apilar(v);
    Mientras P no sea vacía
        w = P.tope();
        ord++;
        orden(w)=ord;
        P.desapilar();
        Para cada vecino z de w
            Si z no está visitado
                marcar z;
                P.apilar(z);
```



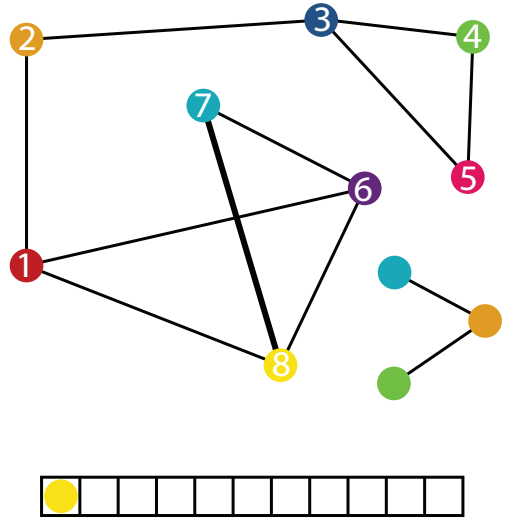
Recorrido DFS: pseudocódigo

```
pila P; vértices v, w; ord = 0;
Mientras queden vértices sin visitar
    v = elegir un vértice no visitado;
    marcar v;
    P.vaciar();
    P.apilar(v);
    Mientras P no sea vacía
        w = P.tope();
        ord++;
        orden(w)=ord;
        P.desapilar();
        Para cada vecino z de w
            Si z no está visitado
                marcar z;
                P.apilar(z);
```



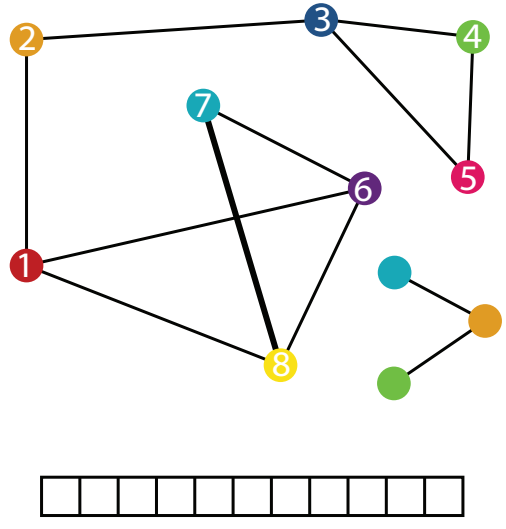
Recorrido DFS: pseudocódigo

```
pila P; vértices v, w; ord = 0;
Mientras queden vértices sin visitar
    v = elegir un vértice no visitado;
    marcar v;
    P.vaciar();
    P.apilar(v);
    Mientras P no sea vacía
        w = P.tope();
        ord++;
        orden(w)=ord;
        P.desapilar();
        Para cada vecino z de w
            Si z no está visitado
                marcar z;
                P.apilar(z);
```



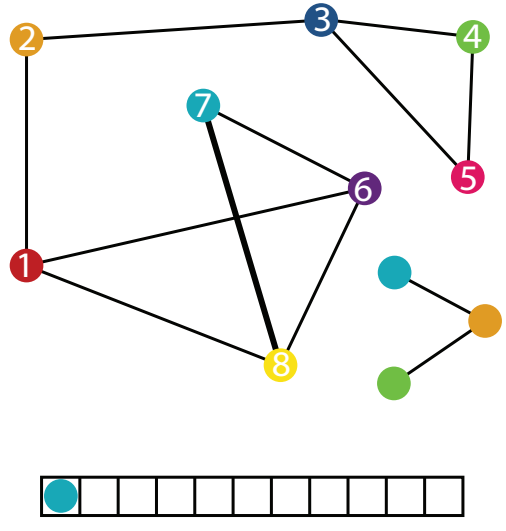
Recorrido DFS: pseudocódigo

```
pila P; vértices v, w; ord = 0;
Mientras queden vértices sin visitar
    v = elegir un vértice no visitado;
    marcar v;
    P.vaciar();
    P.apilar(v);
    Mientras P no sea vacía
        w = P.tope();
        ord++;
        orden(w)=ord;
        P.desapilar();
        Para cada vecino z de w
            Si z no está visitado
                marcar z;
                P.apilar(z);
```



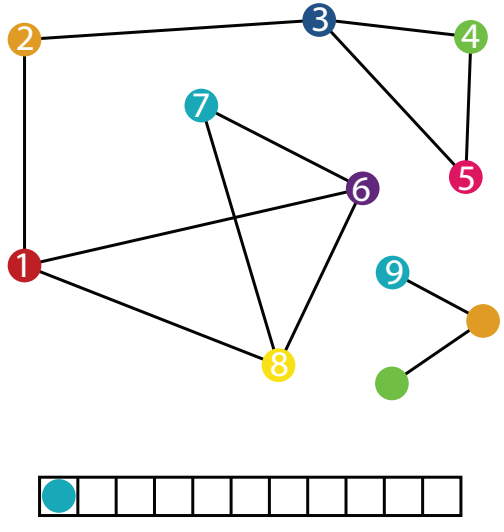
Recorrido DFS: pseudocódigo

```
pila P; vértices v, w; ord = 0;
Mientras queden vértices sin visitar
    v = elegir un vértice no visitado;
    marcar v;
    P.vaciar();
    P.apilar(v);
    Mientras P no sea vacía
        w = P.tope();
        ord++;
        orden(w)=ord;
        P.desapilar();
        Para cada vecino z de w
            Si z no está visitado
                marcar z;
                P.apilar(z);
```



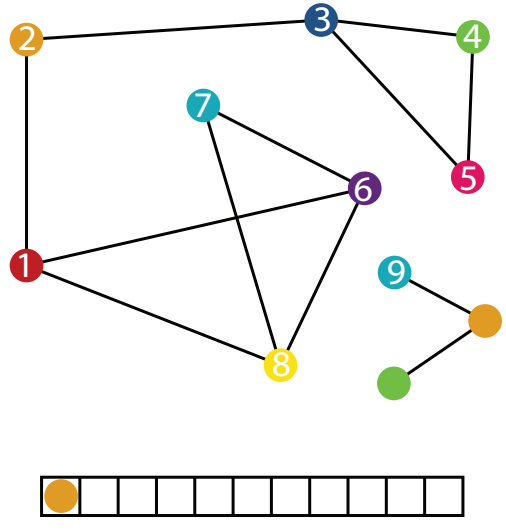
Recorrido DFS: pseudocódigo

```
pila P; vértices v, w; ord = 0;
Mientras queden vértices sin visitar
    v = elegir un vértice no visitado;
    marcar v;
    P.vaciar();
    P.apilar(v);
    Mientras P no sea vacía
        w = P.tope();
        ord++;
        orden(w)=ord;
        P.desapilar();
        Para cada vecino z de w
            Si z no está visitado
                marcar z;
                P.apilar(z);
```



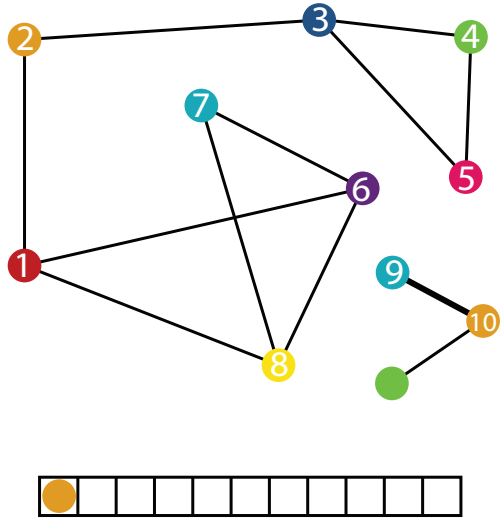
Recorrido DFS: pseudocódigo

```
pila P; vértices v, w; ord = 0;
Mientras queden vértices sin visitar
    v = elegir un vértice no visitado;
    marcar v;
    P.vaciar();
    P.apilar(v);
    Mientras P no sea vacía
        w = P.tope();
        ord++;
        orden(w)=ord;
        P.desapilar();
        Para cada vecino z de w
            Si z no está visitado
                marcar z;
                P.apilar(z);
```



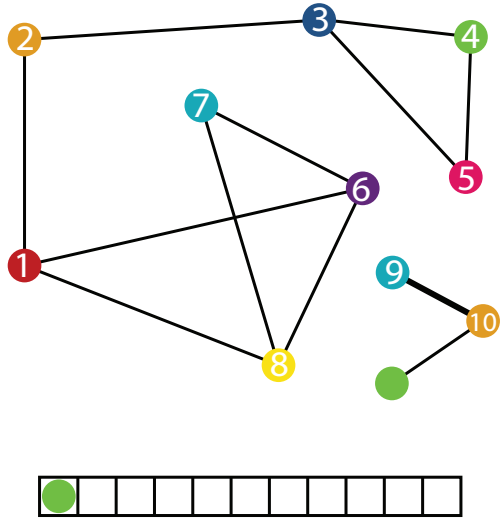
Recorrido DFS: pseudocódigo

```
pila P; vértices v, w; ord = 0;
Mientras queden vértices sin visitar
    v = elegir un vértice no visitado;
    marcar v;
    P.vaciar();
    P.apilar(v);
    Mientras P no sea vacía
        w = P.tope();
        ord++;
        orden(w)=ord;
        P.desapilar();
        Para cada vecino z de w
            Si z no está visitado
                marcar z;
                P.apilar(z);
```



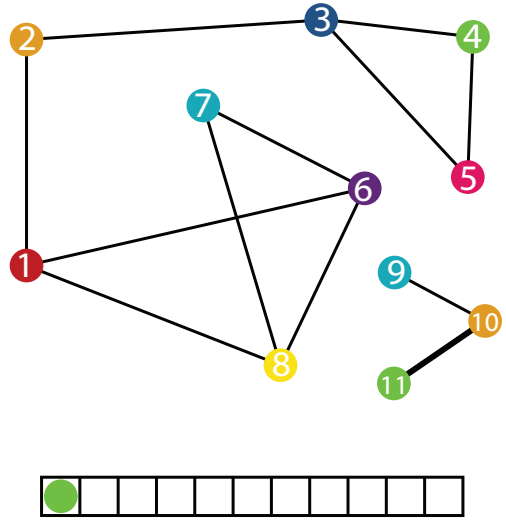
Recorrido DFS: pseudocódigo

```
pila P; vértices v, w; ord = 0;
Mientras queden vértices sin visitar
    v = elegir un vértice no visitado;
    marcar v;
    P.vaciar();
    P.apilar(v);
    Mientras P no sea vacía
        w = P.tope();
        ord++;
        orden(w)=ord;
        P.desapilar();
        Para cada vecino z de w
            Si z no está visitado
                marcar z;
                P.apilar(z);
```



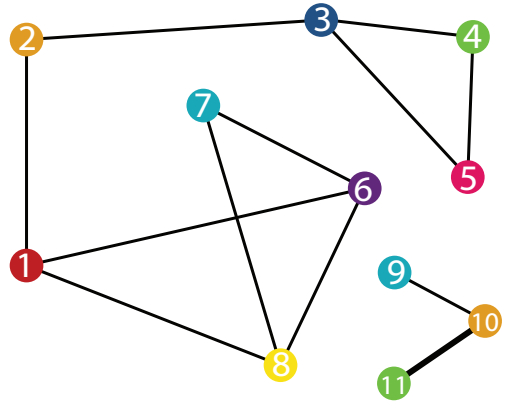
Recorrido DFS: pseudocódigo

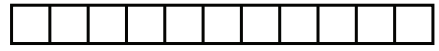
```
pila P; vértices v, w; ord = 0;
Mientras queden vértices sin visitar
    v = elegir un vértice no visitado;
    marcar v;
    P.vaciar();
    P.apilar(v);
    Mientras P no sea vacía
        w = P.tope();
        ord++;
        orden(w)=ord;
        P.desapilar();
        Para cada vecino z de w
            Si z no está visitado
                marcar z;
                P.apilar(z);
```



Recorrido DFS: pseudocódigo

```
pila P; vértices v, w; ord = 0;
Mientras queden vértices sin visitar
    v = elegir un vértice no visitado;
    marcar v;
    P.vaciar();
    P.apilar(v);
    Mientras P no sea vacía
        w = P.tope();
        ord++;
        orden(w)=ord;
        P.desapilar();
        Para cada vecino z de w
            Si z no está visitado
                marcar z;
                P.apilar(z);
```





Propiedades

- Al igual que BFS, este algoritmo puede modificarse para calcular las componentes conexas de un grafo y/o su cantidad.
- El grafo en si puede no estar totalmente almacenado, pero tener una descripción implícita que permita generar bajo demanda los vecinos de un vértice. Un ejemplo típico de este caso es cuando el grafo es el árbol de posibilidades de un problema de búsqueda exhaustiva. El recorrido DFS es el que se utiliza habitualmente para hacer *backtracking*, ya que permite encontrar algunas soluciones rápido y usarlas para "podar" (es decir, no recorrer) ciertas ramas del árbol.
- BFS puede ser útil en ese contexto para encontrar el camino más corto hacia la salida de un laberinto, por ejemplo, aún si el laberinto viene descripto de manera implícita.

Estructuras de datos y complejidad

- Para estos algoritmos conviene en general tener el grafo dado por listas de adyacencia.
- De esta manera, el *Para cada vecino z de w* puede ser realizado en orden $O(d(w))$.
- Con lo cual, si el marcar, asignar orden y las operaciones de cola y pila se realizan en orden constante, la complejidad total será de $O(n+m)$, para un grafo de n vértices y m aristas. El n viene del recorrido secuencial que va identificando si quedan vértices no marcados.
- Si en cambio el grafo está dado por matriz de adyacencia, la complejidad será $O(n^2)$, porque buscar los vecinos cuesta $O(n)$.