Existe secuencia de amigos

En este ejercicio nos piden encontrar una secuencia de amigos entre los diferentes usuarios de una red:

Código a revisar:

1.

```
existeSecuenciaDeAmigos :: RedSocial -> Usuario -> Usuario -> Bool
existeSecuenciaDeAmigos rsRed uU1 uU2
    | (pertenece uU1 (usuarios rsRed) && pertenece uU2 (usuarios rsRed)) == False  = False
    | (cantidadDeAmigos rsRed uU1 == 0 || cantidadDeAmigos rsRed uU2 == 0) == True = False
    | (relacionadosDirecto uU1 uU2 rsRed) == True                                  = True
    | otherwise = cadenaDeAmigos (amigosDe rsRed uU1) uU2 rsRed [uU1]
```

2.

```
cadenaDeAmigos :: [Usuario] -> Usuario -> RedSocial -> [Usuario] -> Bool
cadenaDeAmigos [] _ _ _ = False
cadenaDeAmigos [amU1] un rsRed _ = (relacionadosDirecto amU1 un rsRed || interseccion (amigosDe rsRed amU1) (amigosDe rsRed un))
cadenaDeAmigos (amU1:amU1s) un rsRed yaTesteado
    | pertenece amU1 yaTesteado = cadenaDeAmigos amU1s un rsRed yaTesteado
    | (cantidadDeAmigos rsRed amU1) == 1                                  = cadenaDeAmigos amU1s un rsRed (yaTesteado ++ [amU1])
    | cadenaDeAmigos (amigosDe rsRed amU1) un rsRed (yaTesteado ++ [amU1]) == False = cadenaDeAmigos amU1s un rsRed (yaTesteado ++ [amU1])
    | otherwise                                                           = cadenaDeAmigos (amigosDe rsRed amU1) un rsRed (yaTesteado ++ [amU1])
```

3.

```
relacionadosDirecto :: Usuario -> Usuario -> RedSocial -> Bool
relacionadosDirecto uU1 uU2 rsRed = pertenece (uU1, uU2) (relaciones rsRed) || pertenece (uU2, uU1) (relaciones rsRed)
```

4.

```
interseccion :: (Eq t) => [t] -> [t] -> Bool
interseccion _ []    = False
interseccion [] _    = False
interseccion [x] liY = pertenece x liY
interseccion (x:xs) liY
    | pertenece x liY = True
    | otherwise       = interseccion xs liY
```

Voy a explicarlo desde el que tiene menor dependencia hacia el que tiene mayor dependencia, empezando por 4.

4. Intersección:

Esta función verifica si existe una intersección entre dos listas, funciona exactamente igual que en algebra.

Por ejemplo, si hacemos

```
interseccion [1,2,3] [3,4,5]
```

Obtenemos como resultado True, pues algebraicamente $\{1,2,3\} \cap \{3,4,5\} = \{3\}$

En cambio

```
interseccion [1,2,3] [9,8,7]
```

Nos devolverá False, pues $\{1,2,3\} \cap \{9,8,7\} = \emptyset$

En resumen, analiza el cardinal de $\#(A \cap B)$ y devuelve True si $\#(A \cap B) \geq 1$, o devuelve False si $\#(A \cap B) = 0$

3. RelacionadosDirecto:

Esta función verifica si dos usuarios son amigos en la red.

Supongamos que tengo una red definida como:

```
redA = ([usuariosA], [relacionesA], [])
```

Donde:

```
usuariosA = [u1, u2]
relacionesA = [u1_u2]
u1 = (1, "Juan")
u2 = (2, "Miguel")
u1_u2 = (u1, u2)
```

En este caso hacer:

```
relacionadosDirecto u1 u2 redA
```

Nos devuelve un True, pues existe una tupla (en este caso u1_u2) en redA que satisface las condiciones de relacionados directo, es decir $(u1, u2) \in relaciones(rsA) \vee (u2, u1) \in relaciones(rsA)$, donde ∈ simboliza la función pertenece.

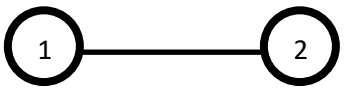
1. existeSecuanciaDeAmigos

Esta función vamos a analizarla por partes:

```
existeSecuenciaDeAmigos :: RedSocial -> Usuario -> Usuario -> Bool
existeSecuenciaDeAmigos rsRed uU1 uU2
    | (pertenece uU1 (usuarios rsRed) && pertenece uU2 (usuarios rsRed)) == False  = False
    | (cantidadDeAmigos rsRed uU1 == 0 || cantidadDeAmigos rsRed uU2 == 0) == True = False
    | (relacionadosDirecto uU1 uU2 rsRed) == True                                  = True
    | otherwise = cadenaDeAmigos (amigosDe rsRed uU1) uU2 rsRed [uU1]
```

¿Primero que nada, de que se trata esta función? Bien, esta función es de tipo indicadora: indica si existe o no una secuencia de amigos que encadene a dos usuarios que son parte de una red. Hagamos algunos ejemplos para entender de manera más practica esto:

Supongamos que tengo una red con dos usuarios y una relación entre estos dos usuarios:



¿Existe una secuencia de amigos entre 1 y 2? La respuesta es sí, pues, están directamente relacionados con esto podemos resumir las primeras tres guardas de la función:

Primera guarda:

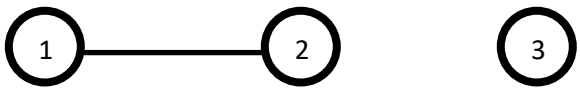
```
    | (pertenece uU1 (usuarios rsRed) && pertenece uU2 (usuarios rsRed)) == False  = False
```

Verifica que los usuarios a los que vamos a testear pertenezcan a la red, no tendría sentido por ejemplo, en nuestra red, buscar una relación entre un usuario 1 y un usuario 3, en caso de que alguno de los dos usuarios no este en la red se devuelve False, es decir no existe tal secuencia de amigos.

Segunda guarda:

```
    | (cantidadDeAmigos rsRed uU1 == 0 || cantidadDeAmigos rsRed uU2 == 0) == True = False
```

Verifica si alguno de los usuarios que pasamos como parámetro tiene cero amigos y si alguno de los dos no tiene amigos devolvemos False ¿Por qué verificamos la cantidad de amigos?, volvamos a nuestro ejemplo:



Esta vez tenemos tres usuarios, pero el usuario número tres no tiene ningún amigo, no podrá existir nunca una cadena de amigos entre, por ejemplo, el usuario 1 y el usuario 3, pues el usuario 3 tiene 0 amigos.

Tercera guarda

---

[1] amU1 significa amigo de U1

[2] un indica a al último usuario de la secuencia pues en algebra se nota $[u1, u2, \dots, un]$, donde un es el último elemento de la secuencia

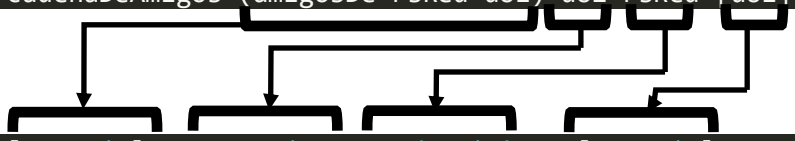
```
    | (relacionadosDirecto uU1 uU2 rsRed) == True                                    = True
```

Se explica por sí misma, el usuario 1 y el usuario 2 son amigos, entonces hay una secuencia de amigos (ellos 2).

Cuarte guarda:

Si no pudimos ver si hay una secuencia de amigos con las pruebas anteriores caemos sobre una función auxiliar, en este caso se llama cadenaDeAmigos, analicemos parámetro a parámetro que es lo que está pasando:

```
    | otherwise = cadenaDeAmigos (amigosDe rsRed uU1) uU2 rsRed [uU1]
```



```
cadenaDeAmigos :: [Usuario] -> Usuario -> RedSocial -> [Usuario] -> Bool
```

Primer parámetro: como primer parámetro tenemos un *[Usuario]*: Vemos que como primer parámetro estamos pasando la lista de amigos de *uU1*.

Segundo parámetro: como segundo parámetro le pasamos *uU2*

Tercer parámetro: como tercer parámetro le pasamos la red con la que sestamos trabajando.

Cuarto parámetro: le pasamos una lista de usuarios que esta inicializada con *uU1* como elemento.

Pasemos ahora a cadenaDeAmigos:

```
cadenaDeAmigos :: [Usuario] -> Usuario -> RedSocial -> [Usuario] -> Bool
cadenaDeAmigos [] _ _ _ = False
cadenaDeAmigos [amU1] un rsRed _ = (relacionadosDirecto amU1 un rsRed || interseccion (amigosDe rsRed amU1) (amigosDe rsRed un))
cadenaDeAmigos (amU1:amU1s) un rsRed yaTesteado
    | pertenece amU1 yaTesteado = cadenaDeAmigos amU1s un rsRed yaTesteado
    | (cantidadDeAmigos rsRed amU1) == 1                                    = cadenaDeAmigos amU1s un rsRed (yaTesteado ++ [amU1])
    | cadenaDeAmigos (amigosDe rsRed amU1) un rsRed (yaTesteado ++ [amU1]) == False = cadenaDeAmigos amU1s un rsRed (yaTesteado ++ [amU1])
    | otherwise                                                             = cadenaDeAmigos (amigosDe rsRed amU1) un rsRed (yaTesteado ++ [amU1])
```

Empezamos por las primeras dos líneas:

```
cadenaDeAmigos [] _ _ _ = False
cadenaDeAmigos [amU1] un rsRed _ = (relacionadosDirecto amU1 un rsRed || interseccion (amigosDe rsRed amU1) (amigosDe rsRed un))
```

Primera línea: Vemos que solo importa el primer parámetro. Verifica si el primer parámetro esta vacío y devuelve False, recordemos que el primer parámetro son la lista de amigos de uU1.

Segunda línea: El último parámetro no importa. En caso de que la lista de amigos de uU1 tenga un único usuario (amU1[1]), verifica si el usuario es amigo de un[2] o si hay una intersección entre la lista de amigos de amU1 y un.

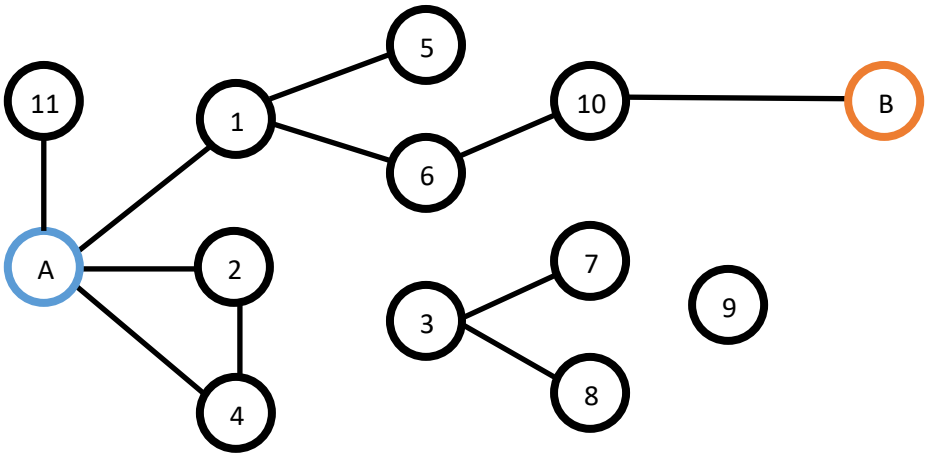
¿Por qué verificamos la intersección entre la lista de amU1 y un, en lugar de solo ver si son amigos? Porque buscamos un amigo que ambos tengan en común para no tener que estar verificando uno por uno los usuarios:

Supongamos que $amU1$ tiene como amigos a $\{u1, u2, u3\}$ y un tiene como amigos a $\{u2, u4\}$, entonces al hacer la intersección obtenemos $\{u1, u2, u3\} \cap \{u2, u4\} = u2$, luego obtenemos True, en este caso particular.

Guardas:

```
cadenaDeAmigos (amU1:amU1s) un rsRed yaTesteado
    | pertenece amU1 yaTesteado = cadenaDeAmigos amU1s un rsRed yaTesteado
    | (cantidadDeAmigos rsRed amU1) == 1                                    = cadenaDeAmigos amU1s un rsRed (yaTesteado ++ [amU1])
    | cadenaDeAmigos (amigosDe rsRed amU1) un rsRed (yaTesteado ++ [amU1]) == False = cadenaDeAmigos amU1s un rsRed (yaTesteado ++ [amU1])
    | otherwise = cadenaDeAmigos (amigosDe rsRed amU1) un rsRed (yaTesteado ++ [amU1])
```

La idea: Supongamos que tengo la siguiente red



Donde cada circulo es un usuario y cada línea es una relación entre esos usuarios. Digamos que estamos buscando una secuencia de amigos entre A y B, se ve que la secuencia de amigos es: $[A, 1,6,10, B]$

¿Ahora como hacemos eso en nuestro programa?

En nuestro caso el primer parámetro son los amigos de A, es decir 1,2, 4 y 11. Por una cuestión de simplicidad empecemos por la segunda guarda:

```
    | (cantidadDeAmigos rsRed amU1) == 1                    = cadenaDeAmigos amU1s un rsRed (yaTesteado ++ [amU1])
```

Primero, amU1 como ya dijimos es un amigo de A, en este caso puede ser 1, 2, 4 u 11. Imaginemos que amU1 es 11 y tiene un solo amigos, el mismo A, evidentemente por ese camino no puede haber una secuencia hasta B.

¿Cómo descartamos a ese 1? Observando la igualdad vemos que llamamos a cadenaDeAmigos con que tiene como primer parámetro a $amU1S$ (no confundir como $amU1$), $amU1S$ es la lista de amigos de A sin incluir a 11, es decir:

$(amU1: amU1S) = [11,2,4]$

$(amU1S) = [2,4]$

El segundo y el tercer parámetro es un (el último elemento que debe estar en nuestra secuencia), y $rsRed$ (la red en la que estamos trabajando) respectivamente.

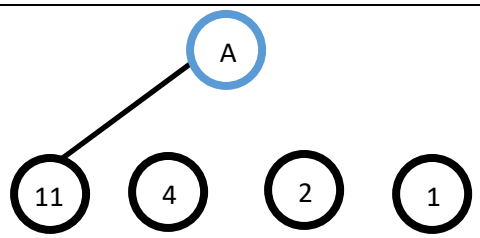
Y el último parámetro: Este se usa para descartar es una lista de los usuarios que ya verificamos, más adelante veremos porque es útil mantener esta lista. Por ahora recordar que la lista $yaTesteado$ vale $yaTesteado = \{A, 11\}$

Primera, tercera y cuarta guarda:

```
| pertenece amU1 yaTesteado = cadenaDeAmigos amU1s un rsRed yaTesteado
| cadenaDeAmigos (amigosDe rsRed amU1) un rsRed (yaTesteado ++ [amU1]) == False = cadenaDeAmigos amU1s un rsRed (yaTes
teado ++ [amU1])
| otherwise = cadenaDeAmigos (amigosDe rsRed amU1) un rsRed (yaTesteado ++ [amU1])
```

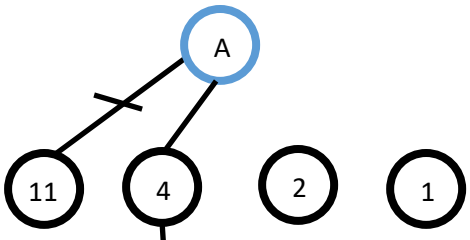
Esta es la más complicada de entender, pero es prácticamente la que hace el mayor esfuerzo, lo que hacemos gráficamente es:

$yaTestados = [A]$



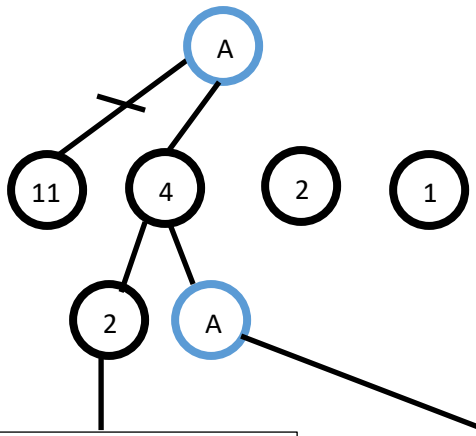
¿Pertenece a yaTesteados?, No
¿Tiene un solo amigo? Si

$yaTestados = [A, 11]$



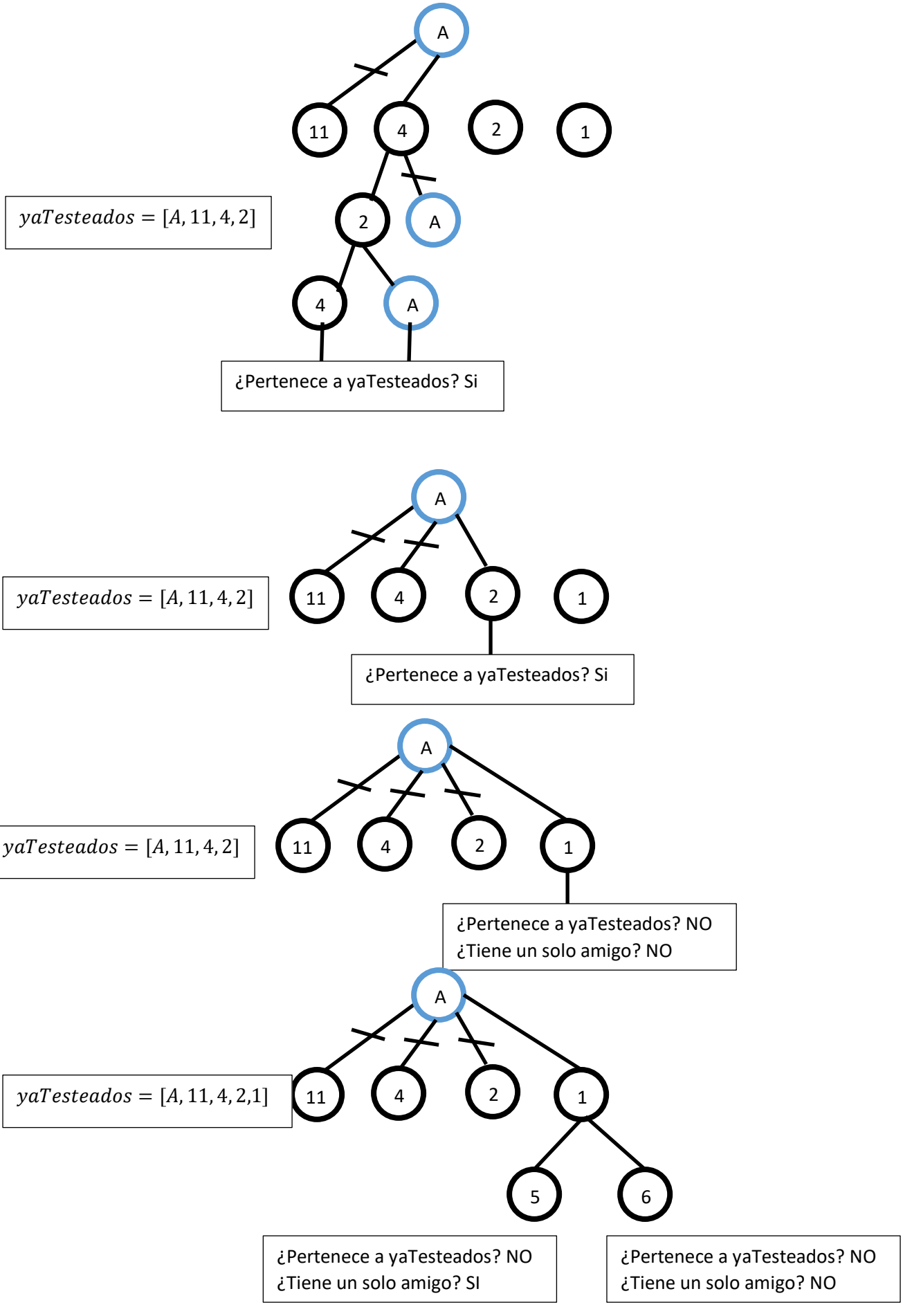
¿Pertenece a yaTesteados?
No
¿Tiene un solo amigo? No

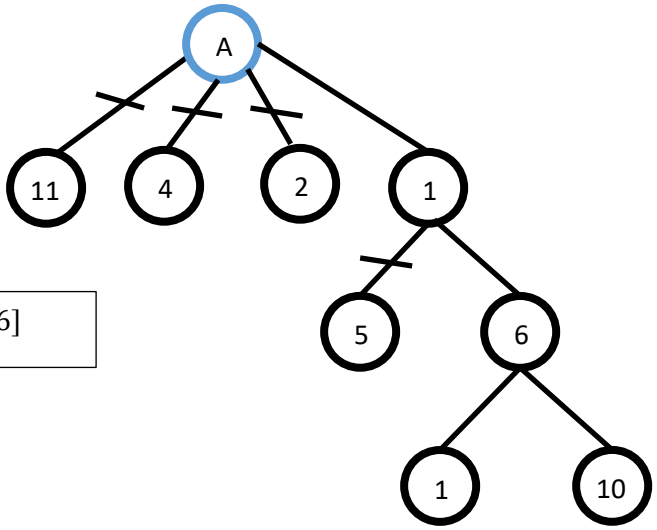
$yaTesteados = [A, 11,4\ ]$



¿Pertenece a yaTesteados?
No
¿Tiene un solo amigo? No

¿Pertenece a yaTesteados? Si

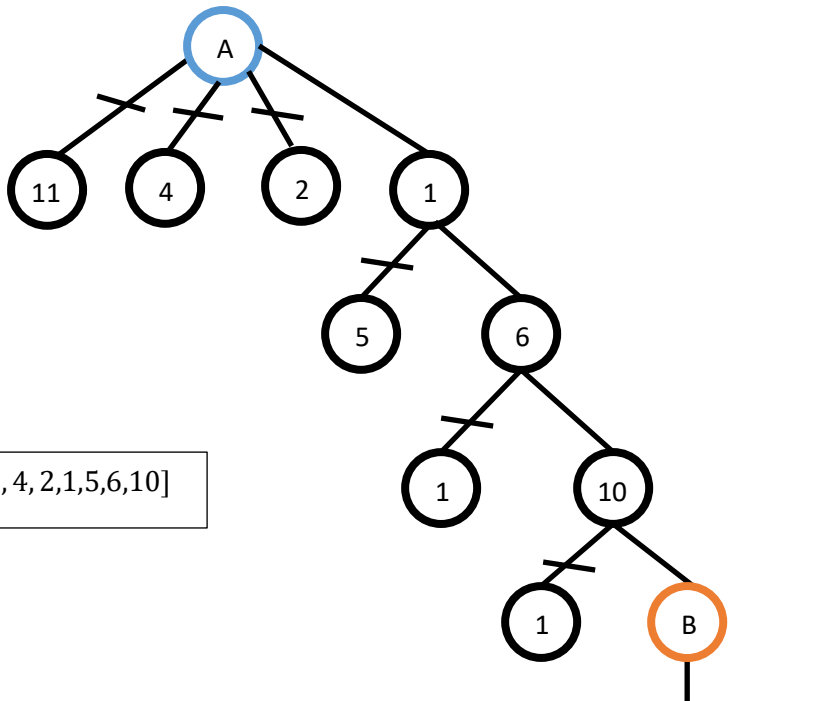

A
11
4
2
1
$yaTesteados = [A, 11, 4, 2]$
2
A
4
A
¿Pertenece a yaTesteados? Si
A
11
4
2
1
$yaTesteados = [A, 11, 4, 2]$
¿Pertenece a yaTesteados? Si
A
11
4
2
1
$yaTesteados = [A, 11, 4, 2]$
¿Pertenece a yaTesteados? NO
¿Tiene un solo amigo? NO
A
11
4
2
1
$yaTesteados = [A, 11, 4, 2,1]$
5
6
¿Pertenece a yaTesteados? NO
¿Tiene un solo amigo? SI
¿Pertenece a yaTesteados? NO
¿Tiene un solo amigo? NO



$yaTesteados = [A, 11, 4, 2, 1, 5, 6]$

¿Pertenece a yaTesteados? SI

¿Pertenece a yaTesteados? NO
¿Tiene un solo amigo? SI



$yaTesteados = [A, 11, 4, 2, 1, 5, 6, 10]$

Devuelvo True

En resumen, nuestro algoritmo hace lo siguiente:

1. Toma dos usuarios y una red, si U0 y UN son amigos, entonces devuelvo True

2. Si U0 o U0 no tienen amigos devuelvo False

3. Si U0 y UN tienen amigos, pero no son directamente amigos, buscamos algún U1 que es amigo de U0, y si ese U1 es amigo de U2 o tiene algún amigo en común con U2 devolvemos True, en caso contrario:

4. Repetir paso 1 pero hacer: Toma dos usuarios y una red, si U1 y UN son amigos, entonces devuelvo True

Matemáticamente:

Sean $m, n \in \mathbb{N}_0$, $m < n$, con $m = 0$