Introducción

En el siguiente material abordaremos:

- Tipo de operadores: operacionales, relacionales y lógicos
- Estructuras de control: Composición secuencial. Instrucción compuesta.
- Selección condicional. Bucles e iteraciones.

01. Operadores

Muchas veces deseamos hacer distintos tipos de relaciones entre objetos.
Ya en el documento "Primeros pasos", vimos ejemplos de aplicación de los operadores aritméticos. Como éstos, existen 2 tipos de operadores: operadores relacionales y operadores lógicos.

Operadores Aritméticos	
Operador	Descripción
+	Suma
-	Resta
*	Multiplicación
/	División
^	Exponencial
%%	Módulo o resto de la división
%/%	División Entera

02.Operadores relacionales

Los operadores relacionales son símbolos mediante los cuales podemos realizar operaciones. Como resultado obtenemos objetos de clase *logical*, que pueden adoptar los valores TRUE o FALSE.

Por ejemplo, si introducimos en la consola:

```
> 5 == 10/2
```

En este caso estamos proponiendo la igualdad entre los dos términos. Como la respuesta es verdadera, al dar click en Enter, R da como resultado "TRUE".

Operadores Relacionales	
Operador	Descripción
<	Menor que
>	Mayor que
<=	Menor o igual que
>=	Mayor o menor que
==	Igual que
!=	Distinto que

A continuación, algunos ejemplos del uso de estos operadores:

```
> 8 > 1          # TRUE  (¿ocho es mayor que uno?)
> 3 < 2          # FALSE (¿tres es menor que dos?)
> 3 == 2         # FALSE (¿tres es igual a dos?)
> 7 == 7         # TRUE (¿siete es igual a siete?)
> 5 != 5         # FALSE (¿cinco es distinto de cinco?)
```

Veamos qué sucede cuando trabajamos con vectores:

```
> x <- 10:15
> x < 13      # Compara cada elemento de x con el valor 13
```

Si generamos un vector y luego lo comparamos con un valor, obtenemos como resultado la comparativa de cada elemento del vector con el parámetro que introducimos.

```
[1]  TRUE  TRUE  TRUE FALSE FALSE FALSE
```

En cambio, cuando comparamos 2 vectores

```
> y <- 16:11
> x == y   # Compara cada elemento de 'x' con el de 'y'
```

Obtenemos la comparación de cada elemento del vector x con el elemento correspondiente en el vector y.

```
[1] FALSE FALSE FALSE  TRUE FALSE FALSE
```

Esto es lo mismo que decir: x[i] ==y[i], para todo i comprendido entre 1 y el largo del vector. Como la comparación se realiza por ubicación de los elementos, si los vectores tienen distintos números de elementos, R nos devolverá un mensaje de advertencia.

```
> z <- 11:19
> x == z
```

```
[1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
Warning message:
In x == y : longer object length is not a multiple of shorter object length
```

Una aplicación muy importante es su uso para identificar valores dentro de un vector que cumplen con una condición de interés.

Veamos ésto con un ejemplo:

Podemos querer buscar todos los valores dentro del vector x que sean menores a 5:

```
> cond <- x < 13
> cond
```

Por tanto, el vector *cond* devuelve como "TRUE" las posiciones del vector x en donde se cumple que los valores son menores a 5.

```
[1]  TRUE  TRUE  TRUE FALSE FALSE FALSE
```

Esto nos permite usar el vector lógico *cond* para obtener los valores de *x* que cumplen la condición indicada en su asignación.

```
> x[cond]
```

```
[1] 10 11 12
```

03.Operadores lógicos

Quizás algunos recuerden aquellas clases de lógica del colegio secundario, en donde veíamos cómo obtener el resultado de distintas proposiciones compuestas utilizando una tabla de verdad. ¡Bueno, es hora de desempolvar ese antiguo conocimiento!

La proposición
"Una proposición (o enunciado) es una oración con valor referencial o informativo, de la cual se puede predicar su veracidad o falsedad."

Continuando con lo visto, x <13 es una proposición, puesto que su resultado tiene un valor referencial en términos de verdadero/falso. Las proposiciones, en la práctica, son las condiciones que planteamos utilizando los operadores relacionales.
Los operadores lógicos sirven para evaluar y/o combinar varias proposiciones.

<table>
<tr><th colspan="4">Operadores Lógicos</th></tr>
<tr><th>Operador</th><th>Significado</th><th colspan="2">Descripción</th></tr>
<tr><td>!</td><td>NOT (No)</td><td colspan="2">Negación. Opuesto.</td></tr>
<tr><td>&</td><td rowspan="2">AND (Y)</td><td rowspan="2">Conjunción. Cumplimiento en forma simultánea</td><td>Toma en cuenta cada elemento del vector</td></tr>
<tr><td>&&</td><td>Toma en cuenta el primer elemento</td></tr>
<tr><td>||</td><td rowspan="2">OR (O)</td><td rowspan="2">Disyunción. Alternativa.</td><td>Toma en cuenta cada elemento del vector</td></tr>
<tr><td>|</td><td>Toma en cuenta el primer elemento</td></tr>
<tr><td>xor(a,b)</td><td>OR - AND</td><td colspan="2">Disyunción exclusiva</td></tr>
</table>

04. La conjunción: &, &&

Dadas dos proposiciones, al plantear la expresión A && B estamos evaluando el cumplimiento simultáneo de ambas condiciones: A y B.

Esta expresión da como resultado TRUE si y sólo si ambas operaciones tienen como resultado TRUE.

A	B	A && B
TRUE	TRUE	TRUE
FALSE	FALSE	FALSE
TRUE	FALSE	FALSE
FALSE	TRUE	FALSE

No debe confundirse el operador de asignación = con el de comparación ==

Ejemplo:

```
> A <- 3 > 2      # TRUE
> B <- 4/5 < 7/8   # TRUE
> C <-  8 == 9    # FALSE
> A & B          # TRUE
> A && B         # TRUE
> A & C          # FALSE
```

Como en este caso A, B y C son vectores de un solo elemento, las variantes & y && no muestran resultados distintos. Sin embargo, **para vectores lógicos de varios elementos es necesario utilizar & para realizar la operación elemento por elemento**.

Por ejemplo, al utilizar el vector *x* que definimos anteriormente, podemos generar vectores lógicos de varios elementos:

```
> E <- x > 12
> F <- x < 12
```

Al aplicar el operador lógico &:

```
> E & F
```

Obtenemos un vector lógico con valores de TRUE para las posiciones que son TRUE en las dos comparaciones (en este caso, sólo en la posición 6, en la que tanto x como y valen 0).

```
[1]  TRUE  TRUE FALSE FALSE FALSE FALSE
```

Si en su lugar usamos && para este caso, R simplemente evalúa el primer elemento de cada uno de los vectores lógicos e ignora el resto del vector.

```
> E && F
```

```
[1] TRUE
```

05. La disyunción: |, ||

Dadas dos proposiciones, al plantear la expresión A || B estamos evaluando el cumplimiento de una de las dos condiciones: A o B.

En este caso, basta con que una de las dos proposiciones sea TRUE para que el resultado también lo sea TRUE.

A	B	A || B
TRUE	TRUE	TRUE
FALSE	FALSE	FALSE
TRUE	FALSE	TRUE
FALSE	TRUE	TRUE

Veamos esto con ejemplos:

```
> A | B    # TRUE
> A || C    # TRUE
```

Para el caso de operaciones que involucren vectores, se sigue con | y || el mismo criterio que con & y &&:

- | para evaluar pares de valores en los vectores. Ejemplo: E || F
- || para evaluar el primer valor de cada vector. Ejemplo: E || F

Desafío: seleccionar a partir de una condición

Tenemos un vector compuesto por los valores: 42, 67, 23, 65 , 87, 91, 12, 31 y queremos seleccionar los valores que cumplan las siguientes 2 condiciones: ser mayores a 30 y menores a 80.

1. confeccionamos el vector:

```
> vector <- c(42, 67, 23, 65 , 87, 91, 12, 31)
```

2) establecemos las condiciones lógicas:

```
> condicion_1 <- vector > 30
> condicion_2 <- vector < 80
```

3) seleccionamos el operador lógico. Para este caso el AND

```
condicion_1 & condicion_2
```

4) utilizamos el conjunto de condiciones para buscar la posición en el vector:

```
> vector[condicion_1 & condicion_2]
```

06.La disyunción exclusiva: xor(a, b)

Dados a y b vectores lógicos, **xor() devuelve como resultado TRUE solamente en las posiciones en las que solo uno de los dos contiene TRUE**. Es decir, solo evalúa como verdadero si se cumple en forma exclusiva una de las dos condiciones.

Conceptualmente es equivalente a sustraer de los resultados verdaderos de una operación con OR los resultados que sean, además, verdaderos en una operación con AND.

A	B	xor(A, B)
TRUE	TRUE	FALSE
FALSE	FALSE	FALSE
TRUE	FALSE	TRUE
FALSE	TRUE	TRUE

La sintaxis es diferente a lo visto para AND u OR ya que utiliza la función **xor()**.

Ejemplo:

```
> xor(A,B) #FALSE
> xor(A,C) #TRUE
```

07.Negación: !

Si E es un vector lógico, entonces !E es el opuesto de este vector. Es decir, para todo valor TRUE de E, !E tendrá un valor FALSE y viceversa.

Por ejemplo, compare los siguiente vectores lógicos:

```
> E
> !E
> F
> !F
```

EXTRA

- La función **any()** devuelve TRUE si hay algún valor TRUE dentro de un vector lógico.
- La función **all()** devuelve TRUE si todos los valores son TRUE dentro de un vector lógico.

En general && es preferido para el control de flujo, como son los condicionales *if* o *while*.

08. Estructuras de control

Hasta ahora en el curso nos hemos limitado a trabajar con secuencias simples de comandos, scripts o funciones que se ejecutan de forma lineal, una tras otra. Si bien se pueden lograr cosas muy útiles y sofisticadas con este sistema, aún falta conocer las herramientas más potentes que tiene la programación: *las estructuras de control*.

Las estructuras de control nos permiten controlar el flujo de ejecución de nuestra secuencia de comandos. Permiten realizar cosas como saltear líneas, repetir líneas, hacer bifurcaciones y mucho más.

Tanto las estructuras de control como el uso de funciones son aspectos fundamentales de la programación, más allá de cuál sea el lenguaje que escojamos para nuestro trabajo.

¡Si por momentos sentís que no entendés algo, no desesperes! ***A programar se aprende haciendo.*** Con la práctica vas a ir afianzando esas pequeñas dudas que puedan surgir.

09.Composición secuencial

Como hemos visto hasta acá, cuando queremos ejecutar varias líneas seguidas, las escribimos en orden, una debajo de la otra.

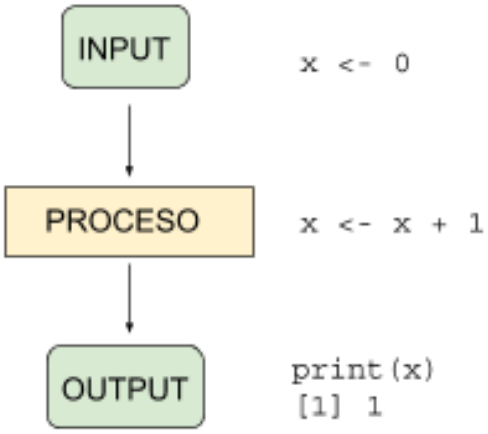
Si tenemos la secuencia:

```
> x <- 0      # INPUT o ENTRADA
> x <- x + 1  # PROCESO
> print(x)    # OUTPUT o SALIDA
```

En la primer línea generamos una variable *x* y le asignamos el valor 0. Luego a la variable *x* le sumamos 1 y su resultado lo asignamos a la variable *x*. Por tanto, *x* ahora vale 1 y con la función ***print ( )*** imprimimos el resultado en la consola.

Podemos tomar el primer paso como un input, una entrada o punto de inicio, donde puede cambiar el valor que le asignemos a *x* siempre que x sea una variable numérica para que el script tenga sentido. x + 1 es el proceso que llevamos adelante en el script. Y finalmente, nuestra salida es la impresión del valor que toma x tras el proceso.

En este ejemplo, la ejecución de las sentencias es secuencial: se ejecuta línea tras línea. A su vez, el flujo de información es lineal. No hay nada que indique una condición o la repetición de alguna de las sentencias.



10. Instrucción compuesta

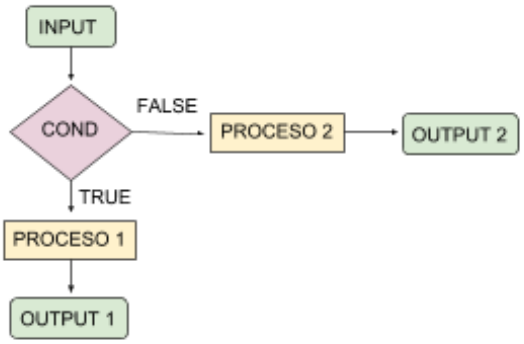
Una secuencia de instrucciones se vuelve una **instrucción compuesta** cuando utilizamos las llaves { }.

```
> x <- 0
> {x <- x +1
  print(x)}
```

Las llaves agrupan las sentencias y hace que se ejecuten como un bloque. Por otro lado, las variables que creamos dentro del bloque sólo sobreviven a la duración del bloque, permitiéndonos crear variables temporales.

11. Selección condicional

Como ya mencionamos, hasta el momento sólo vimos situaciones donde teníamos un flujo de información lineal, pero ¿qué pasa cuando queremos hacer una cosa si A y otra si B?



Cuando programamos es posible condicionar la ejecución u omisión de una instrucción a través del uso de la estructura de control condicional ***if.***

La sintaxis para utilizar un ***if*** es la siguiente:

if(condición){ proceso 1 } else { proceso 2 }

La condición debe ser una expresión del tipo booleana y siempre debe ir dentro de un paréntesis. A continuación, debemos escribir el proceso (instrucción compuesta) que queremos que ocurra en caso de que la sentencia utilizada como condición sea verdadera. Agregamos un ***else***, y finalmente escribimos el proceso a utilizarse cuando la sentencia es falsa.

De esta forma, cuando la ejecución alcanza el condicional, primero se evalúa la condición, si el resultado es TRUE, se ejecuta el proceso 1 y, si es FALSE, ejecuta el proceso 2.

Esto lo podemos leer como: "*Si la condición es verdadera realizar el proceso 1, de lo contrario, utilizar el proceso 2*"

Por ejemplo, veamos el ejemplo siguiente:

```
> if(nota < 7){
   print("desaprobado")
   }else{
   print("aprobado")
   }
```

si la variable nota es menor a 7, R nos devuelve "desaprobado". Si por el contrario, nota es mayor a 7, entonces imprime "aprobado".

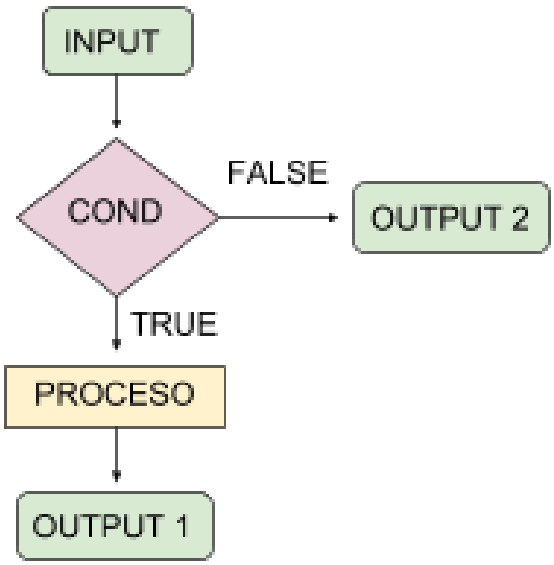
Mantener el indentado (la sangría) tal como se muestra en el ejemplo es parte de las normas de estilo. Si bien R no reconoce el indentado, colocarlo facilita su lectura. Lo mismo sucede con los símbolos de apertura y cierre de bloque, los cuales se recomienda escribirlos en líneas distintas.

Una variante para la selección condicional es el caso en que sólo queramos realizar un proceso si la condición es verdadera:

if(condición){ proceso }

Por ejemplo, muchas veces tenemos tablas con valores cero que en verdad no son ceros absolutos, sino que son niveles menores al límite de detección (<LD) de la técnica. Si quisiéramos reemplazar los ceros de forma automática, podríamos hacerlo con el siguiente script:

```
> if(x == 0){
    x <- "<LD"
    print(x)}
```



Desafío 1: Desarrollar un programa que identifique si un número es par o impar
1- Como primer paso necesitamos identificar nuestro input. En nuestro caso es un número introducido por el usuario. Supongamos que el usuario elige el 4

x <- 4

2- Podemos utilizar el operador aritmético %% que nos devuelve el resto de la división. Entonces, si el resultado es 0 nuestro número es par, si no es impar.

Por tanto, la condición lógica sería:

x%%2 == 0

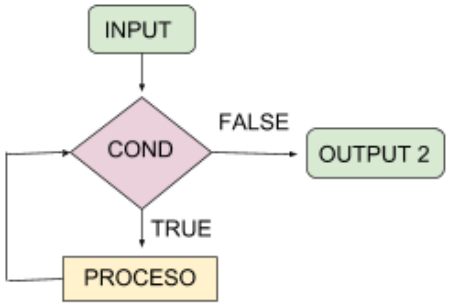
3- Entonces, si (x%%2 == 0) es TRUE queremos que en la consola aparezca "numero par", y si la salida es FALSE que nos imprima "numero impar". Esto quedaría:

```
> x <- 4
> if(x%%2 == 0){
   print("numero par")
   }else{
   print("numero impar")
   }
```

Para probar si el condicional está bien programado, conviene probar con un número par y otro impar y de esta forma explorar todas las posibilidades.

12. Bucles e iteraciones

Un bucle es una composición compuesta que se repite *n* veces hasta alcanzar un límite definido. Cada repetición del bucle es una nueva iteración. Podemos decir que un bucle está compuesto por *n* iteraciones.



En los bucles utilizamos una condición para determinar cuántas veces queremos ejecutar una instrucción. De esta forma, mientras la condición sea verdadera el proceso se repetirá hasta que ya no lo sea más.

En R hay varios comandos que permiten implementar bucles: *for, while y repeat*.

Dependiendo del problema a resolver será más sencilla la implementación de uno u otro tipo de bucle.

13. FOR

El bucle más sencillo es el bucle ***for*** cuya sintaxis tiene la forma:

for(i in secuencia){ expresión }

Veamos un ejemplo sencillo:

```
> for( i in 1:4 ){
   print(i)
}
```

```
[1] 1
[1] 2
[1] 3
[1] 4
```

Esto se puede leer como "*para i entre 1 y 4 imprimir el valor de i*". Por lo tanto, en la primer iteración i vale 1 entonces imprime el 1. En la segunda iteración, i vale 2 entonces imprime el dos, etc.. hasta que i vale 4 e imprime 4. En ese momento, se termina de ejecutar el bucle.

Los bucles se suelen utilizar para recorrer los elementos de un objeto. Recordemos que cuando tenemos un *vector* de *n* elementos, *vector[i]* nos devuelve el elemento en la i-ésima posición.

En el siguiente ejemplo vamos a crear un vector sobre el cual iterar, y vamos a utilizar un bucle para imprimir cada uno de sus valores. Para esto, nos vamos a ayudar con la función **length()** que nos devuelve el largo del vector:

```
> x <- c("a", "b", "c", "d") # Vector sobre el cual vamos a iterar
> length(x)
```

```
[1] 4
```

Incluyendo la función ***length()***:

```
> for( i in 1:length(x) ){
    print(x[i])
 }
```

```
[1] "a"
[1] "b"
[1] "c"
[1] "d"
```

Son posibles otras formas de obtener el mismo resultado. Podríamos haber utilizado la función ***seq_along()***, la cual genera una secuencia numérica con los índices de los elementos en el vector.

```
> seq_along(x)
```

```
[1] 1 2 3 4
```

Incorporando **seq_along()** en el bucle:

```
> for( i in seq_along(x) ){
  print(x[i])
 }
```

Por último, otra forma es acceder a estos directamente:

```
> for( i in x ){
    print(i)
 }
```

En este último ejemplo, *i* ya no representa las posiciones de los elementos, sino los elementos en el vector.

La variable *i* también puede tener cualquier modo: numérico, carácter, lógico o una combinación de estos.

```
> for( i in c(3,2,9,6) ){print (i^2)}
```

```
[1] 9
[1] 4
[1] 81
[1] 36
```

14. WHILE

En casos en donde queremos realizar un ciclo pero no conocemos el número de ciclos que se precisan, se usa **while** que permite iterar hasta que cierto criterio se cumpla.

La sintaxis del bucle ***while*** es de la forma:

while(condición){ expresión }

Por ejemplo, queremos generar un bucle que sume todos los enteros positivos hasta que la suma sea mayor o igual a 1000:

```
> n <- 0
> suma <- 0
> while (suma <= 1000){
      n <- n + 1
      suma <- suma + n }
> c(suma, n)
```

```
[1] 1035   45
```

En el ejemplo, n es un contador de las iteraciones que realiza el bucle, pasando por todos los números enteros positivos. Y suma es la variable que acumula los procesos de las iteraciones.

Este bucle lo podemos leer como: "mientras la variable *suma* sea menor o igual a 1000, a *n* sumarle 1 y asignarlo a *n* y tras esto, a *suma* sumarle el nuevo *n* generado."

Como resultado, R nos muestra que el primer valor de la suma que pasa de 1000 es 1035 y que hicieron falta 45 iteraciones para llegar a este valor.

15. REPEAT

La tercera alternativa para hacer bucles en un programa es la instrucción ***repeat***. Este comando repite un conjunto de instrucciones hasta que se satisfaga un criterio de parada. En R se usa la palabra break para indicar el momento de parar.

La sintaxis de un bucle repeat es, pues, de la forma:

repeat { expresión if (condición) break }

Como ejemplo vamos a utilizar el problema anterior, pero usando la instrucción repeat:

```
> n <- 0
> suma <- 0
> repeat{
      n <- n + 1
      suma <- suma + n
      if ( suma > 1000) break }
> c(suma, n)
```

16. NEXT

En cualquiera de los bucles anteriores es posible introducir el comando **next**, que produce un salto inmediato a la siguiente iteración.

Por ejemplo, si estamos dentro de un bucle ***for*** lo que sucedería al momento de ejecutar la función ***next*** es que se salta directo al siguiente elemento de la iteración.

Veámoslo en un ejemplo. Supongamos que queremos imprimir los numero impares entre 1 y 10:

```
> for ( i in 1:10 ) {
    if (i %% 2 == 0) next
    print(i)
 }
```

A medida que *i* avance se evaluará si el valor que toma es divisible por 2. En caso de que así lo sea, se pasará al siguiente valor de *i*. En el caso contrario, se imprime el valor de *i* en la consola.

17. BREAK

Otra función muy útil en los bucles es la función ***break*** que produce la salida inmediata del bucle.

En el siguiente ejemplo, cuando x[i] toma el valor “c” se detiene el bucle:

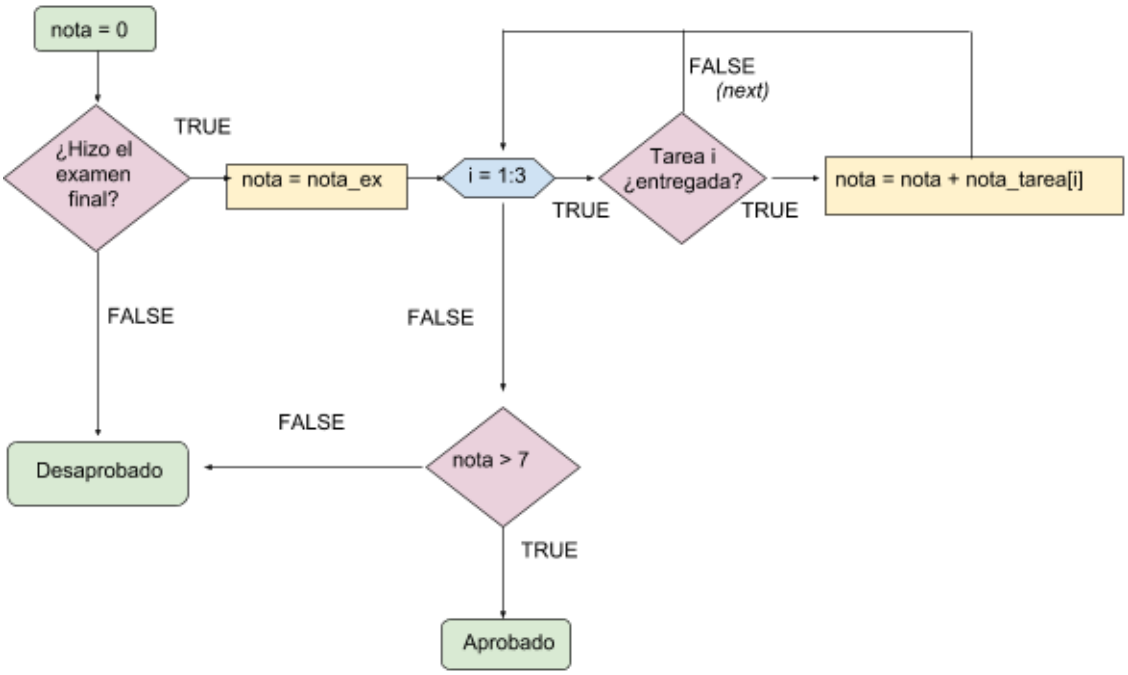
```
> x <- c("a", "b", "c", "d")
> for ( i in x ){
   print(i)
   if( i == "c") break
 }
```

Desafío 2: Algoritmo para obtener la nota final del curso

Para poner en práctica lo que hemos visto, queremos generar un algoritmo que nos devuelva la nota final del curso.
Sabemos que:

- El curso cuenta con 3 instancias evaluatorias: 3 tareas y una evaluación final
- La evaluación final es obligatoria.
- Las notas de las tareas y del examen se suman sin ponderación

¿Cómo harías el algoritmo para calcular la nota final? Te proponemos el siguiente:



En programación se busca que el algoritmo sea lo más eficiente posible. Esto es, con la menor cantidad de estructuras de control. ¿se te ocurre cómo hacerlo más sencillo? Compartilo !

También, podés contarnos si encontraste útiles las estructuras de control o si se te ocurrió en qué podrías utilizarlas.

¿Quién se anima a subir un boceto del flujo del trabajo que le gustaría realizar?