INTRODUCCIÓN A LA GEOCOMPUTACIÓN CON R
EJERCICIOS NO. 7 (WHAT FOR()?)

INDER TECUAPETLA-GÓMEZ

1. Escribe un ciclo *for* que imprima en consola el cubo de los números $1, 2, \ldots, 10$. **Hint:** `?cat`

2. Escribe una función que solicite al usuario ingresar un número natural positivo, y luego imprima en consola el cubo de cada número desde el 1 hasta el número ingresado. **Hint1:** considera el ejercicio anterior así como también el 3 de los Ejercicios No. 6. **Hint2:** posible solución

```
> getCubesTill()
---Calcula el cubo del 1 hasta el numero ingresado---
---El numero ingresado debe ser positivo---
Ingresa un numero: 5
El cubo de  1 es:  1
El cubo de  2 es:  8
El cubo de  3 es:  27
El cubo de  4 es:  64
El cubo de  5 es:  125
```

3. Sin codificarla, ¿puedes deducir cuál es el *output* de la siguiente función?

```
agregar_a_vector <- function(n) {
  output <- c()
  for (i in seq_len(n)) {
    output <- c(output, i)
  }
  output
}
```

4. Sin codificarla, ¿puedes deducir cuál es el *output* de la siguiente función?

```
agregar_a_vector_2 <- function(n) {
  output <- vector("integer", n)
  for (i in seq_len(n)) {
    output[[i]] <- i
  }
  output
}
```

5. Después de cargar a tu sesión de trabajo las funciones `agregar_a_vector()` y `agregar_a_vector2()` ejecuta el siguiente codigo:

Octubre 12, 2023

```
> install.packages(microbenchmark)
> library(microbenchmark)
> timings <- microbenchmark(agregar_a_vector(10000), agregar_a_vector_2(10000), times = 10)
>
> timings
```

El resultado puede variar dependiendo de tu sistema, pero básicamente `agregar_a_vector_2()` es 350 veces más rápido que `agregar_a_vector()`. Nota que la gran diferencia entre estas funciones es la *asignación* de memoria que se realiza en `agregar_a_vector_2()`. Concretamente, la instrucción `output <- vector("integer", n)` asigna/crea/guarda/reserva espacio en memoria para el objeto `output`, un vector de clase `integer` de tamaño `n`; nota que esto es hecho una sola vez. Por el contrario, en `agregar_a_vector()` optamos por `output <- c()` para definir el objeto `output`. Esta forma *dinámica* de reservar espacio en memoria para este objeto posee el inconveniente de ser invocada repetidas veces dentro del ciclo `for` lo que puede explicar la sub optimalidad de `agregar_a_vector_2()`. Puedes leer la documentación de la función `vector()` para conocer su correcto uso. También se recomienda leer la documentación de `microbenchmark()`.

6. Probablemente el siguiente ejercicio puede resolverse con alguna combinación de funciones del `tidyverse()`, sin embargo, de momento, escribe un ciclo `for` para calcular el número de valores únicos (sin repetir) en cada columna del *dataset* `iris`. **Hint1:** `?unique`. **Hint2:** posible salida

```
## Sepal.Length  Sepal.Width Petal.Length  Petal.Width      Species
##           35           23           43           22            3
```

7. Probablemente el siguiente ejercicio puede resolverse con alguna combinación de funciones del `tidyverse()`, sin embargo, de momento, escribe un ciclo `for` para determinar el tipo de objeto en cada columna del *dataset* `nycflights13::flights`; `nycflights13` es un paquete. **Hint:** posible salida

```
## $year
## [1] "integer"
##
## $month
## [1] "integer"
##
## $day
## [1] "integer"
##
## $dep_time
## [1] "integer"
##
## $sched_dep_time
## [1] "integer"
##
## $dep_delay
## [1] "numeric"
##
## $arr_time
## [1] "integer"
##
## $sched_arr_time
```

```
## [1] "integer"
##
## $arr_delay
## [1] "numeric"
##
## $carrier
## [1] "character"
##
## $flight
## [1] "integer"
##
## $tailnum
## [1] "character"
##
## $origin
## [1] "character"
##
## $dest
## [1] "character"
##
## $air_time
## [1] "numeric"
##
## $distance
## [1] "numeric"
##
## $hour
## [1] "numeric"
##
## $minute
## [1] "numeric"
##
## $time_hour
## [1] "POSIXct" "POSIXt"
```

8. Probablemente el siguiente ejercicio puede resolverse con alguna combinación de funciones del `tidyverse()`, sin embargo, de momento, escribe un ciclo `for` para calcular la media de cada columna del *dataset* `mtcars`. **Hint:** solución

```
##        mpg        cyl       disp         hp       drat         wt       qsec
##  20.090625   6.187500 230.721875 146.687500   3.596563   3.217250  17.848750
##         vs         am       gear       carb
##   0.437500   0.406250   3.687500   2.812500
```

9. Escribe código para simular la tirada de 50 volados, almacenando el resultado (1 = águila, 0 = sol) en un vector. **Hint1:** `?rbinom`. **Hint2:** posible solución

```
##  [1] 0 1 0 1 0 0 1 1 1 0 1 1 0 0 0 0 0 1 1 0
```

10. Escribe una función para estudiar el comportamiento a largo plazo del experimento *tirar volados* siguiendo las siguientes especificaciones:

(a) En consola, solicita al usuario la cantidad de volados a generar

(b) En consola, despliega el resultado de los volados

(c) Calcula el promedio aritmético (también conocido como *media*) de los volados generados. Calcula también la desviación estándar de los volados generados. En consola, despliega el promedio y la desviación estándar. **Hint:** ejemplos

```
> getVolados()
--- Genera volados ---
Cuantos volados quieres generar? 5
Estos son tus volados:  1 0 0 1 0
El promedio de tus volados es:  0.4
La desviacion estandar de tus volados es:  0.5477226
> getVolados()
--- Genera volados ---
Cuantos volados quieres generar? 10
Estos son tus volados:  0 0 1 0 1 0 1 0 0 0
El promedio de tus volados es:  0.3
La desviacion estandar de tus volados es:  0.4830459
> getVolados()
--- Genera volados ---
Cuantos volados quieres generar? 20
Estos son tus volados:  1 1 1 0 0 0 0 0 1 1 0 1 0 0 1 0 1 0 0 0
El promedio de tus volados es:  0.4
La desviacion estandar de tus volados es:  0.5026247
```

Los siguiente ejercicios muestran el uso de ciclos `for` *anidados*.

11. Sin codificarlo, ¿puedes deducir cuál es la salida del siguiente código?

```
for (i in 1:2) {
    for (j in 1:3) {
        print(paste("i =", i, "j =", j))
    }
}
```

12. Ciclos `for` y matrices. Un ejemplo clásico es operar ciclos sobre los índices de los renglones, típicamente denotados con la letra i, y las columnas, denotados con la letra j. Considera la siguiente representación de una matriz con 2 renglones y 3 columnas:

$$M = \begin{pmatrix} x_{11} & x_{12} & x_{13} \\ x_{21} & x_{22} & x_{23} \end{pmatrix}.$$

En `R` esta representación es equivalente a:

$$\begin{pmatrix} M[1,1] & M[1,2] & M[1,3] \\ M[2,1] & M[2,2] & M[2,3] \end{pmatrix}$$

Entonces, para acceder a cada elemento de `M` (una vez), necesitamos ciclar sobre los valores de `i in 1:2` y `j in 1:3`; esto es justo lo que hace el código del ejercicio anterior. Consideremos acceder a los elementos de la siguiente matriz a través de un ciclo `for` anidado.

```
(x <- matrix(c(9, 0, 3, 17, 5, 2), ncol = 3))
```

```
##      [,1] [,2] [,3]
## [1,]    9    3    5
## [2,]    0   17    2
```

```
for (i in 1:2) {
    for (j in 1:3) {
        cat("El elemento x[", i, ", ", j, "] es ", x[i, j], "\n")
    }
}
```

```
## El elemento x[ 1 ,  1 ] es  9
## El elemento x[ 1 ,  2 ] es  3
## El elemento x[ 1 ,  3 ] es  5
## El elemento x[ 2 ,  1 ] es  0
## El elemento x[ 2 ,  2 ] es  17
## El elemento x[ 2 ,  3 ] es  2
```

Evita mezclar las dimensiones (renglones por columnas) y los índices correspondientes. Considera el siguiente código:

```
for (i in 1:3) {
    for (j in 1:2) {
        cat("El elemento x[", i, ", ", j, "] es ", x[i, j], "\n")
    }
}
```

```
## El elemento x[ 1 ,  1 ] es  9
## El elemento x[ 1 ,  2 ] es  3
## El elemento x[ 2 ,  1 ] es  0
## El elemento x[ 2 ,  2 ] es  17
## Error in x[i, j] : subscript out of bounds
```

El error se produce cuando el ciclo intenta acceder al elemento `x[3,1]` el cual no existe ya que `x` tiene 2 renglones y 3 columnas; el código de arriba *de forma incorrecta* especifica ciclar sobre `i=1:3` y `j=1:2`, es decir, este código especifica ciclar sobre 3 renglones y 2 columnas.

13. Nota sobre el indizado. En el ejercicio anterior funciona correctamente utilizar `i=1:2` y `j=1:3`, sin embargo para matrices cuyas dimensiones pueden variar, esta manera de indizar los ciclos es *sub óptima*. Una alternativa, es utilizar `1:nrow(x)` y `1:ncol(x)`,

```
(x <- matrix(1:6, nrow = 2))
```

```
##      [,1] [,2] [,3]
## [1,]    1    3    5
## [2,]    2    4    6
```

```
1:nrow(x)
```

```
## [1] 1 2
```

```
1:ncol(x)
```

```
## [1] 1 2 3
```

... o también `seq_len(nrow(x))` y `seq_len(ncol(x))` para evitar problemas si tenemos una matriz con cero renglones o cero columnas.

```
seq_len(nrow(x))
```

```
## [1] 1 2
```

```
seq_len(ncol(x))
```

```
## [1] 1 2 3
```

14. Uso de `names()` para indizar. Hasta ahora hemos utilizado números enteros para indizar los ciclos `for()`. Sin embargo, cuando tenemos una matriz cuyos renglones o columnas tienen nombres, podemos indizar un ciclo `for` usando `rownames()` o `colnames()`. Por ejemplo,

```
(x <- matrix(c(28,35,13,13,1.62,1.53,1.83,1.71), nrow=4,
             dimnames = list(c("Veronica", "Carlos", "Miriam", "Pedro"),
                             c("Edad", "Estatura"))))
```

```
##          Edad Estatura
## Veronica   28     1.62
## Carlos     35     1.53
## Miriam     13     1.83
## Pedro      13     1.71
```

```
for (rname in rownames(x)) {
    for (cname in colnames(x)) {
        cat("La", cname, "de", rname, "es", x[rname, cname], "\n")
    }
}
```

```
## La Edad de Veronica es 28
## La Estatura de Veronica es 1.62
## La Edad de Carlos es 35
## La Estatura de Carlos es 1.53
## La Edad de Miriam es 13
## La Estatura de Miriam es 1.83
## La Edad de Pedro es 13
## La Estatura de Pedro es 1.71
```

15. Queremos obtener el promedio de cada una de las 3 columnas del objeto `x` definido en el ejercicio anterior. Esto puede lograrse, por ejemplo, usando un ciclo `for`:

(a) Cicla sobre todas los columnas usando sus nombres.

(b) Extrae la columna correspondiente.

(c) Calcula el promedio aritmético

Hint: Posible solución

```
## La Edad promedio es 22.25
## La Estatura promedio es 1.6725
```