

# Introducción a la Programación

Guía de estudio - Lic. en Estadística - FCEyE - UNR

Mgs. Lic. Marcos Prunello (Prof. Tit. Ord.)

2023-03-21

# Índice general

<b>1 Introducción a la Programación</b> . . . . .	<b>5</b>
1.1 Qué es la programación . . . . .	5
1.2 Una breve reseña histórica sobre la programación . . . . .	5
1.3 Software y hardware . . . . .	9
1.4 Problemas, algoritmos y lenguajes de programación . . . . .	11
1.5 Errores de programación . . . . .	15
1.6 Procesador, ambiente y acciones . . . . .	17
1.7 R y RStudio . . . . .	18
1.8 Guía de estilo . . . . .	21
<b>2 Objetos y operadores</b> . . . . .	<b>23</b>
2.1 Objetos . . . . .	23
2.2 Operadores . . . . .	26
2.3 Entrada y salida de información . . . . .	31
2.4 Directorio de trabajo . . . . .	32
<b>3 Estructuras de control</b> . . . . .	<b>35</b>
3.1 Estructuras de control secuenciales . . . . .	35
3.2 Estructuras de control condicionales . . . . .	35
3.3 Estructuras de control iterativas . . . . .	38
3.4 Ejemplos . . . . .	42
<b>4 Descomposición algorítmica</b> . . . . .	<b>45</b>
4.1 Tipos de subalgoritmos . . . . .	45
4.2 Funciones . . . . .	46
4.3 Funciones en R . . . . .	48
4.4 Documentación de los subalgoritmos . . . . .	55
4.5 Pasaje de parámetros . . . . .	57
4.6 Ámbito de las variables . . . . .	59
4.7 Otras nociones importantes en R . . . . .	63
4.8 Otros tópicos de lectura opcional . . . . .	64
<b>5 Estructuras de datos</b> . . . . .	<b>69</b>
5.1 Arreglos . . . . .	69
5.2 Características particulares de las estructuras de datos en R . . . . .	80
5.3 Otras consideraciones (lectura opcional) . . . . .	99
5.4 Arreglos multidimensionales (lectura opcional) . . . . .	100
<b>6 Uso de archivos de datos</b> . . . . .	<b>103</b>
<b>7 Otros tópicos</b> . . . . .	<b>105</b>
7.1 La consola . . . . .	105
7.2 Uso de argumentos en la línea de comandos al ejecutar código de R . . . . .	112
7.3 Generación de secuencias . . . . .	116
<b>Bibliografía</b> . . . . .	<b>119</b>

## BIENVENIDA

¡Les damos la bienvenida a la asignatura Introducción a la Programación en la Facultad de Ciencias Económicas y Estadística, Universidad Nacional de Rosario! La presente guía resume los conceptos más importantes que vamos a desarrollar. La misma irá siendo revisada, completada y actualizada a lo largo del cuatrimestre y no está exenta de presentar errores o expresar ideas que puedan ser mejoradas. Avisanos si encontrás algo que deba ser cambiado. ¡Esperamos que juntos podamos pasarlo bien al dar nuestros primeros pasos en la programación!

Este material fue escrito por el Mgs. Lic. Marcos Prunello (profesor titular) y revisado por los integrantes de la cátedra, Tec. César Mignoni y Lic. Maite San Martín.

**NOTA:** La versión en PDF no es revisada en cuanto a su configuración, por lo tanto es probable que se encuentren páginas con espacios en blanco, tablas con filas cortadas, figuras muy grandes, etc. El archivo PDF es generado sin revisar su estado final, pero el texto es el mismo que la versión online (<https://mpru.github.io/introprog>)



## Unidad 1

# Introducción a la Programación

### 1.1. Qué es la programación

Las computadoras son una parte esencial de nuestra vida cotidiana. Casi todos los aparatos que usamos tienen algún tipo de computadora capaz de ejecutar ciertas tareas: lavarropas con distintos modos de lavado, consolas de juegos para momentos de entretenimiento, calculadoras súper potentes, computadoras personales que se usan para un montón de propósitos, teléfonos celulares con un sinfín de aplicaciones y miles de cosas más.

Todos estos dispositivos con computadoras de distinto tipo tienen algo en común: alguien “les dice” cómo funcionar, es decir, les indica cuáles son los pasos que deben seguir para cumplir una tarea. De eso se trata la *programación*: es la actividad mediante la cual las *personas* le entregan a una *computadora* un conjunto de instrucciones para que, al ejecutarlas, ésta pueda *resolver un problema*. Quienes realizan esta actividad reciben el nombre de *programadores*. Sin las personas que las programen, las computadoras dejarán de ser útiles, por más complejos que sean estos aparatos. Los conjuntos de instrucciones que reciben las computadoras reciben el nombre de *programas*.

La programación es un proceso creativo: en muchas ocasiones la tarea en cuestión puede cumplirse siguiendo distintos caminos y el programador es el que debe imaginar cuáles son y elegir uno. Algunos de estos caminos pueden ser mejores que otros, pero en cualquier caso la computadora se limitará a seguir las instrucciones ideadas por el programador.

Desafortunadamente, las computadoras no entienden español ni otro idioma humano. Hay que pasarles las instrucciones en un lenguaje que sean capaces de entender. Para eso debemos aprender algún *lenguaje de programación*, que no es más que un lenguaje artificial compuesto por una serie de expresiones que la computadora puede interpretar. Las computadoras interpretan nuestras instrucciones de forma muy literal, por lo tanto a la hora de programar hay que ser muy específicos. Es necesario respetar las reglas del lenguaje de programación y ser claros en las indicaciones provistas.

Ahora bien, ¿por qué debemos estudiar programación en la Licenciatura en Estadística? La actividad de los profesionales estadísticos está atravesada en su totalidad por la necesidad de manejar con soltura herramientas informáticas que nos asisten en las distintas etapas de nuestra labor, desde la recolección y depuración de conjuntos de datos, pasando por la aplicación de distintas metodologías de análisis, hasta la comunicación efectiva de los resultados. Por eso, en la asignatura *Introducción a la Programación* estudiaremos los conceptos básicos de esta disciplina, fomentando la ejercitación del pensamiento abstracto y lógico necesario para poder entendernos hábilmente con la computadora y lograr que la misma realice las tareas que necesitamos.

Para poner en práctica los conceptos sobre Programación que aprenderemos, vamos a emplear un lenguaje que ha sido desarrollado específicamente para realizar tareas estadísticas, llamado *R*. Sin embargo, debemos resaltar que éste no es un curso sobre *R*, es decir, no nos dedicaremos a aprender las herramientas que este lenguaje brinda para el análisis de datos. De hecho, frente a variados problemas vamos a dedicarnos a crear soluciones que ya existen y están disponibles en *R*, pero lo haremos con el fin de utilizar dicho lenguaje para aprender y ejercitarse en nociones básicas de programación.

### 1.2. Una breve reseña histórica sobre la programación

La historia de la programación está vinculada directamente con la de la computación. Esta palabra proviene del latín *computatio*, que deriva del verbo *computare*, cuyo significado es “enumerar cantidades”. Computación, en este

sentido, designa la acción y efecto de computar, realizar una cuenta, un cálculo matemático. De allí que antiguamente computación fuese un término usado para referirse a los cálculos realizados por una persona con un instrumento expresamente utilizado para tal fin (como el ábaco, por ejemplo) o sin él. En este sentido, la computación ha estado presente desde tiempos ancestrales, sin embargo debemos remontarnos al siglo XVII para encontrar los primeros dispositivos diseñados para automatizar cómputos matemáticos.

En 1617 el matemático escocés John Napier (el mismo que definió los logaritmos) inventó un sistema conocido como *los huesos de Napier* o *huesos neperianos* que facilitaba la tarea de multiplicar, dividir y tomar raíces cuadradas, usando unas barras de hueso o marfil que tenían dígitos grabados. Esta fue la base para otras ideas más avanzadas, entre ellas la que dio origen a la primera calculadora mecánica, inventada por el alemán Wilhelm Schickard en 1623, capaz de realizar cálculos aritméticos sencillos funcionando a base de ruedas y engranajes. Se componía de dos mecanismos diferenciados, un ábaco de Napier de forma cilíndrica en la parte superior y un mecanismo en la inferior para realizar sumas parciales de los resultados obtenidos con el aparato de la parte superior. Fue llamado *reloj calculador*. A partir de aquí se fueron desarrollando otros modelos, todos ellos teniendo en común el hecho de ser puramente mecánicos, sin motores ni otras fuentes de energía. El operador ingresaba números ubicando ruedas de metal en posiciones particulares y al girarlas otras partes de la máquina se movían y mostraban el resultado. Algunos ejemplos son las calculadoras del inglés William Oughtred en 1624, de Blaise Pascal en 1645 (llamada *pascalina*), la de Samuel Morland en 1666 y las de Leibniz, en 1673 y 1694.



Figura 1.1: De izquierda a derecha: los huesos de Napier (Museo Arqueológico Nacional de España), el reloj calculador de Schickard (Museo de la Ciencia de la Universidad Pública de Navarra) y una pascalina del año 1952

El siglo XVIII trajo consigo algunos otros diseños, pero un gran salto se dio a comienzos del siglo XIX de mano de un tejedor y comerciante francés, Joseph Jacquard. En 1801 creó un telar que tenía un sistema de tarjetas perforadas para controlar las puntadas del tejido, de forma que fuera posible *programar* una gran diversidad de tramas y figuras. Sin saberlo, Jacquard sentó una idea fundamental para la creación de las computadoras.



Figura 1.2: Un telar de Jacquard y sus tarjetas perforadas en el Museo de la ciencia y la industria en Mánchester.

En 1822 el matemático británico Charles Babbage publicó un diseño para la construcción de una *máquina diferencial*, que podía calcular valores de funciones polinómicas mediante el método de las diferencias. Este complejo sistema de ruedas y engranajes era el primero que podía trabajar automáticamente utilizando resultados de operaciones

previas. Si bien el diseño era viable, por motivos técnicos y económicos no lo pudo concretar (sólo construyó un modelo de menor escala). Sin embargo, Babbage no se dio por vencido y en 1837 presentó el diseño de una *máquina analítica*, un aparato capaz de ejecutar cualquier tipo de cálculo matemático y que, por lo tanto, se podría utilizar con cualquier propósito. Tal como el telar de Jacquard, la operación de esta máquina sería controlada por un patrón de perforaciones hechas sobre una tarjetas que la misma podría leer. Al cambiar el patrón de las perforaciones, se podría cambiar el comportamiento de la máquina para que resuelva diferentes tipos de cálculos. Para la salida de resultados, la máquina sería capaz de perforar tarjetas. Además, funcionaría con un motor a vapor y su tamaño hubiese sido de 30 metros de largo por 10 de ancho. Si bien Babbage tampoco llegó a concretar en vida este diseño que dejó plasmado en más de 300 dibujos y 2200 páginas por motivos políticos, se lo considera como la primera conceptualización de lo que hoy conocemos como computadora, por lo cual Babbage es conocido como *el padre de la computación*.

En 1843 Lady Ada Lovelace, una matemática y escritora británica, publicó una serie de notas sobre la máquina analítica de Babbage, en las que resaltaba sus potenciales aplicaciones prácticas, incluyendo la descripción detallada de tarjetas perforadas para que sea capaz de calcular los números de Bernoulli. Al haber señalado los pasos para que la máquina pueda cumplir con estas y otras tareas, Ada es considerada actualmente como la primera programadora del mundo, a pesar de que en la época no fue tomada en serio por la comunidad científica, principalmente por su condición de mujer.



Figura 1.3: Charles Babbage, Ada Lovelace y el algoritmo que publicó Ada para calcular los números de Bernoulli con la máquina analítica de Charles.

La utilidad de las tarjetas perforadas quedó confirmada en 1890, cuando Herman Hollerith las utilizó para automatizar la tabulación de datos en el censo de Estados Unidos. Las perforaciones en determinados lugares representaban información como el sexo o la edad de las personas, logrando que se pudieran lograr clasificaciones y conteos de forma muy veloz. Así, se tardaron sólo 3 años en procesar la información del censo, cinco años menos que en el anterior de 1880. Con el fin de comercializar esta tecnología, Hollerith fundó una compañía que terminaría siendo la famosa International Business Machine (IBM), empresa líder en informática hasta el día de hoy.

Sin embargo, la visión de Babbage de una computadora programable no se hizo realidad hasta los años 1940, cuando el advenimiento de la electrónica hizo posible superar a los dispositivos mecánicos existentes. John Atanasoff y Clifford Berry (Iowa State College, Estados Unidos) terminaron en 1942 en Iowa State College (Estados Unidos) una computadora electrónica capaz de resolver sistemas de ecuaciones lineales simultáneas, llamada *ABC* (por “Atanasoff Berry Computer”). La misma contaba con 300 tubos de vacío, unas bombillas de vidrio con ciertos componentes que podían recibir y modificar una señal eléctrica mediante el control del movimiento de los electrones produciendo una respuesta, que habían sido presentados por primera vez en 1906 por el estadounidense Lee De Forest. La *ABC* dio comienzo a la conocida como la *primera generación de computadoras* basadas en el empleo de tubos de vacío.

La primera computadora electrónica de propósito general fue la *ENIAC*, *Electronic Numerical Integrator and Computer*, completada por Presper Eckert y John Mauchly en la Universidad de Pensilvania. Podía realizar cinco mil operaciones aritmética por segundo y tenía más de 18000 tubos de vacío, ocupando una sala de 9x15 metros en un sótano de la universidad donde se montó un sistema de aire acondicionado especial.

Ni la *ABC* ni la *ENIAC* eran reprogramables: la *ABC* servía el propósito específico de resolver sistemas de ecuaciones

y la *ENIAC* era controlada conectando ciertos cables en un panel, lo que hacía muy compleja su programación. El siguiente gran avance se produjo en 1945, cuando el matemático húngaro-estadounidense John von Neumann (Universidad de Princeton) propuso que los programas, es decir, las instrucciones para que la máquina opere, y también los datos necesarios, podrían ser representados y guardados en una memoria electrónica interna. Así nació el concepto de *programa almacenado* (o *stored-program*), en contraposición con el uso de tableros de conexiones y mecanismos similares de los modelos vigentes. Los creadores de la ENIAC, bajo la consultoría de von Neumann, implementaron esto en el diseño de su sucesora, la *EDVAC*, terminada en 1949. También ya había experimentado con esta idea el alemán Konrad Zuse, quien entre 1937 y 1941 desarrolló la *Z3*, por lo cual es considerada por algunos como la primera máquina completamente automática y programable. En lugar de usar tubos de vacío, empleaba un conjunto de 2600 relés, unos dispositivos electromagnéticos inventados en 1835 y empleados, por ejemplo, en telegrafía. El modelo original de la *Z3* fue destruido en Berlín por un bombardeo durante la segunda guerra mundial.

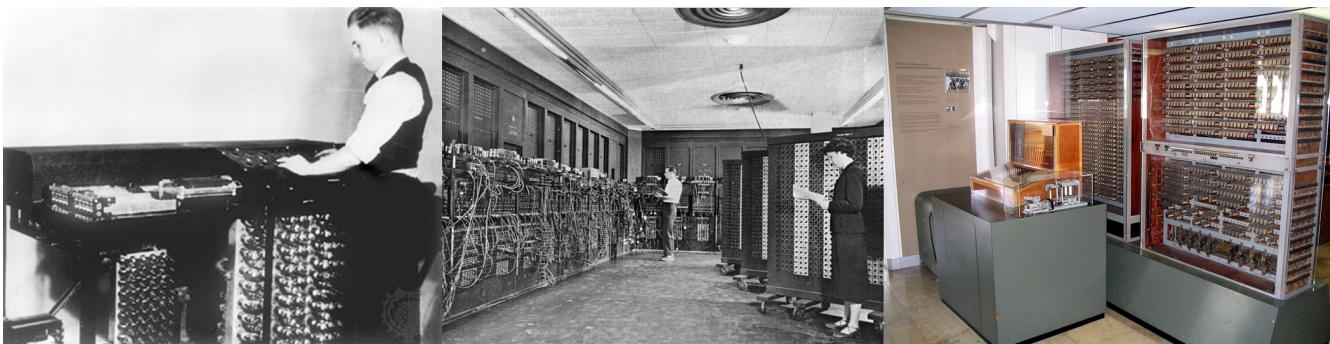


Figura 1.4: De izquierda a derecha: las computadoras ABC, ENIAC y Z3

Este nuevo paradigma cambió la historia de la computación, como también lo hizo la invención del *transistor* en 1947 en los Laboratorios Bell. Un *transistor* es un dispositivo electrónico semiconductor que entrega una señal de salida en respuesta a una señal de entrada, mucho más pequeño que los tubos de vacío y que consumen menos energía eléctrica. Así, una computadora podía tener cientos de miles de transistores, no obstante ocupando mucho espacio.

Desde entonces, la computación ha evolucionado muy rápidamente, con la introducción de nuevos sistemas y conceptos, que llegan a los complejos y poderosos diseños electrónicos que caracterizan la vida actual. En un intento de caracterizar y resumir esta impactante evolución, algunos historiadores dividen al desarrollo de las computadoras modernas en “generaciones” (esta clasificación no es única y existen diversas versiones de la misma):

- *Primera generación* (aprox. 1940-1958): se trata de las computadoras electrónicas que usaban tubos de vacío para su circuito interno. Los equipos eran enormes y ocupan habitaciones enteras. Consumían mucha electricidad y generaban demasiado calor. Podía llevar días o semanas modificar las conexiones para hacer que la computadora resuelva un problema diferente. Usaban tarjetas perforadas y cinta de papel para la lectura de datos e impresiones para mostrar las salidas.
- *Segunda generación* (aprox. 1958-1964): se caracteriza por el uso de *transistores* (inventados en 1947) en lugar de tubos de vacío, permitiendo que las computadoras tengan un consumo eléctrico más eficiente, sean más baratas, más pequeñas y más rápidas.
- *Tercera generación* (aprox. 1964-1971): se inició en 1959 con el desarrollo de un circuito integrado (“chip”) que se trata de una pequeña placa de silicio sobre el cual se imprime un gran número de transistores conectados. La primera computadora de este estilo fue de IBM en 1960. Al ser más pequeñas y baratas, su uso llegó a una mayor audiencia. Se pudo interactuar con la máquina mediante teclados, monitores y un sistema operativo, que posibilitaba ejecutar múltiples acciones a la vez bajo el monitoreo de un programa central.
- *Cuarta generación* (aprox. 1971-presente): los avances tecnológicos permitieron construir la unidad entera de procesamiento de una computadora sobre un único chip de silicio (*microprocesador*), incluyendo la memoria y los controles de entrada y salida de datos. Todo lo que en una computadora de la primera generación ocupaba una habitación entera, fue capaz de entrar en la palma de una mano. El primer microprocesador fue el Intel 4004 de 1971, mientras que la primera computadora de uso doméstico fue desarrollada por IBM en 1981. Surgieron también en esta generación el ratón (*mouse*) y las interfaces gráficas de usuario (como Windows, en 1985).



Figura 1.5: De derecha a izquierda: un tubo de vacío, un transistor y un chip.

### 1.3. Software y hardware

Como podemos ver, en la historia de la computación hubo dos aspectos que fueron evolucionando: las máquinas y los programas que las dirigen. Hacemos referencia a estos elementos como *hardware* y *software* respectivamente, y es la conjunción de ambos la que le da vida a la computación y hace posible la programación.



Figura 1.6: Representación de la diferencia entre hardware y software.

El *hardware* es el conjunto de piezas físicas y tangibles de la computadora. Existen diversas formas de clasificar a los elementos que componen al hardware, según distintos criterios:

Cuadro 1.1: Clasificación del hardware.

Criterio	Clasificación	Descripción	Ejemplos
Según su utilidad	Dispositivos de procesamiento	Son los que reciben las instrucciones mediante señales eléctricas y usan cálculos y lógica para interpretarlas y emitir otras señales eléctricas como resultado.	microprocesador, tarjeta gráfica, tarjeta de sonido, etc.
	Dispositivos de almacenamiento	Son capaces de guardar información para que esté disponible para el sistema.	disco duro, pen drive, DVD, etc.
	Dispositivos de entrada	Captan instrucciones por parte de los usuarios y las transforman en señales eléctricas interpretables por la máquina.	teclado, mouse, touch pad, etc.
	Dispositivos de salida	Transforman los resultados de los dispositivos de procesamiento para presentarlos de una forma fácilmente interpretable para el usuario.	monitor, impresora, etc
Según su ubicación	Dispositivos internos	Generalmente se incluye dentro de la carcasa de la computadora.	microprocesador, disco rígido, ventiladores, módem, tarjeta gráfica, fuente de alimentación, puertos, etc.
	Dispositivos externos o periféricos	No se incluye dentro de la carcasa de la computadora y está al alcance del usuario	monitor, teclado, mouse, joystick, micrófono, impresora, escáner, pen drive, lectores de código de barras, etc.
Según su importancia	Hardware principal	Dispositivos esenciales para el funcionamiento de la computadora	microprocesador, disco rígido, memoria RAM, fuente de alimentación, monitor, etc.
	Hardware complementario	Aquellos elementos no indispensables (claramente, dependiendo del contexto, alguna pieza del hardware que en alguna situación podría considerarse complementaria, en otras resulta principal).	

Por otro lado tenemos al *software*, que es el conjunto de todos los programas (es decir, todas las instrucciones que recibe la computadora) que permiten que el hardware funcione y que se pueda concretar la ejecución de las tareas. No tiene una existencia física, sino que es intangible. El software se puede clasificar de la siguiente forma:

Cuadro 1.2: Clasificación del software.

Clasificación	Descripción	Ejemplos
Software de sistema o software base	Son los programas informáticos que están escritos en lenguaje de bajo nivel como el de máquina o ensamblador y cuyas instrucciones controlan de forma directa el hardware	BIOS o UEFIs (sistemas que se encargan de operaciones básicas como el arranque del sistema, la configuración del hardware, etc), sistemas operativos (Linux, Windows, iOS, Android), controladores o *drivers*, etc.
Software de aplicación o utilitario	Son los programas o aplicaciones que usamos habitualmente para realizar alguna tarea específica.	procesadores de texto como Word, reproductor de música, Whatsapp, Guaraní, navegadores web, juegos, etc.
Software de programación o de desarrollo	Son los programas y entornos que nos permiten desarrollar nuestras propias herramientas de software o nuevos programas. Aquí se incluyen los lenguajes de programación	C++, Java, Python, R, etc.

## 1.4. Problemas, algoritmos y lenguajes de programación

Mencionamos anteriormente que la *programación* consistía en instruir a una computadora para que resuelva un problema y que la comunicación de esas instrucciones debe ser realizada de forma clara. Es por eso que, ante un problema que debe ser resuelto computacionalmente, el primer paso es pensar detalladamente cuál puede ser una forma de resolverlo, es decir, crear un *algoritmo*. Un *algoritmo* es una estrategia consistente de un conjunto ordenado de pasos que nos lleva a la solución de un problema o alcance de un objetivo. Luego, hay que traducir el algoritmo elegido al idioma de la computadora.

Entonces, podemos decir que la resolución computacional de un problema consiste de dos etapas básicas:

1. *Diseño algorítmico*: desarrollar un algoritmo, o elegir uno existente, que resuelva el problema.
2. *Codificación*: expresar un algoritmo en un lenguaje de programación para que la computadora lo pueda interpretar y ejecutar.

Al aprender sobre programación, comenzamos enfrentándonos a problemas simples para los cuales la primera etapa parece sencilla, mientras que la codificación se torna difícil ya que hay que aprender las reglas del lenguaje de programación. Sin embargo, mientras que con práctica rápidamente podemos ganar facilidad para la escritura de código, el diseño algorítmico se torna cada vez más desafiante al encarar problemas más complejos. Es por eso que haremos hincapié en el planteo y desarrollo de algoritmos como una etapa fundamental en la programación.

### 1.4.1. El diseño algorítmico

Cotidianamente, hacemos uso de algoritmos para llevar adelante casi todas las actividades que realizamos: preparar el desayuno, sacar a pasear la mascota, poner en la tele un servicio de *streaming* para ver una película, etc. Cada una de estas tareas requiere llevar adelante algunas acciones de forma ordenada, aunque no hagamos un listado de las mismas y procedamos casi sin pensar.

Sin embargo, cuando estamos pensando la solución para un problema que va a resolver una computadora, debemos ser claros y concretos, para asegurarnos de que al seguir los pasos del algoritmo se llegue a la solución y para que quien tenga que codificarlo, nosotros mismos u otras personas, lo pueda entender sin problemas. Por eso, el primer paso es idear un algoritmo para su solución y expresarlo por escrito, por ejemplo, en español, pero adaptando el lenguaje humano a formas lógicas que se acerquen a las tareas que puede realizar una computadora. En programación, el lenguaje artificial e informal que usan los desarrolladores en la confección de algoritmos recibe el nombre de *pseudocódigo*. Es la herramienta que utilizamos para describir los algoritmos mezclando el lenguaje común con instrucciones de programación. No es en sí mismo un lenguaje de programación, es decir, la computadora no es capaz de entenderlo, sino que el objetivo del mismo es que el programador se centre en la solución lógica y luego lo utilice como guía al escribir el programa.

El pseudocódigo, como cualquier otro lenguaje, está compuesto por:

- Un *léxico*: conjunto de palabras o frases válidas para escribir las instrucciones.

- Una *sintaxis*: reglas que establecen cómo se pueden combinar las distintas partes.
- Una *semántica*: significado que se les da a las palabras o frases.

El pseudocódigo sigue una *estructura secuencial*: define una acción o instrucción que sigue a otra en secuencia. Esta estructura puede representarse de la siguiente forma:

**ALGORITMO: "Ejemplo"**

COMENZAR

    Acción 1

    Acción 2

    ...

    Acción N

FIN

Se comienza con un título que describa el problema que el algoritmo resuelve, seguido por la palabra **COMENZAR**. Luego se detallan las acciones o instrucciones a seguir y se concluye con la palabra **FIN**. Por ejemplo, si nuestro problema es poner en marcha un auto, el algoritmo para resolverlo puede ser expresado mediante el siguiente pseudocódigo:

**ALGORITMO: "Arrancar el auto"**

COMENZAR

    INSERTAR la llave de contacto

    UBICAR el cambio en punto muerto

    GIRAR la llave hasta la posición de arranque

    SI el motor arranca

        ENTONCES

            DEJAR la llave en posición "encendido"

    SI NO

        LLAMAR al mecánico

    FINSI

FIN

Es importante destacar la presencia de sangrías (*sangrado*) en el ejemplo anterior, que facilitan la lectura.

Los algoritmos suelen ser representados también mediante *diagramas de flujo*, como el que se muestra en la siguiente figura<sup>1</sup>.

#### 1.4.2. Codificación

El algoritmo anterior está presentado en pseudocódigo utilizando el lenguaje español, una opción razonable para compartir esta estrategia entre personas que se comunicuen con este idioma. Claramente, si queremos presentarle nuestro algoritmo a alguien que sólo habla francés, el español ya no sería una buena elección, y mucho menos si queremos presentarle el algoritmo a una computadora. Para que una computadora pueda entender nuestro algoritmo, debemos traducirlo en un *lenguaje de programación*, que, como dijimos antes, es un idioma artificial diseñado para expresar cálculos que puedan ser llevados a cabo por equipos electrónicos, es decir es un medio de comunicación entre el humano y la máquina.

Si bien hay distintos lenguajes de programación, una computadora en definitiva es un aparato que sólo sabe *hablar* en *binario*, es decir, sólo interpreta señales eléctricas con dos estados posibles, los cuales son representados por los dígitos binarios 0 y 1. Toda instrucción que recibe la computadora se construye mediante una adecuada y larga combinación de ceros y unos<sup>2</sup>. Este sistema de código con ceros y unos que la computadora interpreta como instrucciones o conjuntos de datos se llama *lenguaje de máquina* (o código de máquina).

Programar en lenguaje de máquina es muy complejo y lento, y es fácil cometer errores pero es difícil arreglarlos. Por eso a principios de la década de 1950 se inventaron los *lenguaje ensambladores*, que usan palabras para representar simbólicamente las operaciones que debe realizar la computadora. Cada una de estas palabras reemplaza un código de máquina binario, siendo un poco más fácil programar. Imaginemos que deseamos crear un programa que permita

<sup>1</sup>En este curso no emplearemos diagramas de flujo

<sup>2</sup>Como vimos anteriormente, las computadoras de la primera generación no se manejaban con lenguajes de programación, sino que para introducir información e instrucciones en las primeras computadoras se usaban tarjetas perforadas, en las cuales los orificios representaban un “0” y las posiciones que no los tenían se entendían como un “1”, de modo que la máquina podía operar empleando el sistema binario.



Figura 1.7: Ejemplo del algoritmo .^arrancar el autorepresentado gráficamente con un diagrama de flujo.

sumar dos números elegidos por una persona. La computadora puede hacer esto si se lo comunicamos mediante un mensaje compuesto por una larga cadena de ceros y unos (lenguaje de máquina) que a simple vista no podríamos entender. Sin embargo, escrito en lenguaje ensamblador, el programa se vería así (por ejemplo):

```

1 mov ah,01h ;Se guarda 01h en el registro ah
2 int 21h    ;Se llama a la interrupción 21h
3 sub al,30h ;Se resta 30h para obtener el número ingresado
4 mov var1,al ;Se guarda el número ingresado en var1
5 mov ah,01h ;Se guarda 01h en el registro ah
6 int 21h    ;Se llama a la interrupción 21h
7 sub al,30h ;Se resta 30h para obtener el número ingresado
8 add al,var1 ;Se suma var1 y al y el resultado se guarda en el registro al
9 mov dl,al   ;Se guarda el contenido del registro al en el registro dl
10 add dl,30h ;Se suma 30h al registro dl para obtener la suma en ASCII
11 mov ah,02h ;Se guarda 02h en el registro ah
12 int 21h    ;Se llama a la interrupción 21h

```

Figura 1.8: Programa en lenguaje ensamblador para leer dos números, sumarlos y mostrar el resultado. Al final de cada línea hay una descripción de la operación realizada.

El programa que se encarga de traducir esto al código de máquina se llama *ensamblador*. A pesar de que no haya ceros y unos como en el lenguaje de máquina, probablemente el código anterior tampoco sea fácil de entender. Aparecen instrucciones que tal vez podemos interpretar, como *add* por sumar o *sub* por substraer, pero está lleno de cálculos hexadecimales, referencias a posiciones en la memoria de la computadora y movimientos de valores que no lo hacen muy amigable. Por eso, a pesar de que la existencia de los lenguajes ensambladores simplificó mucho la comunicación con la computadora, se hizo necesario desarrollar lenguajes que sean aún más sencillos de usar.

Por ejemplo, con el lenguaje que vamos a aprender, R, el problema de pedirle dos números a una persona y sumarlos se resumen en las siguientes líneas de código:

```
n1 <- scan()
n2 <- scan()
print(n1 + n2)
```

En las dos primeras líneas con la instrucción **scan()** (que quiere decir “escanear”, “leer”) se le pide a la persona que indique dos números y en la tercera línea se muestra el resultado de la suma, con la instrucción **print()** (“imprimir”, “mostrar”). Mucho más corto y entendible.

Esta simplificación es posible porque nos permitimos *ignorar* ciertos aspectos del proceso que realiza la computadora. Todas esas acciones que se ven ejemplificadas en la imagen con el código ensamblador se llevan a cabo de todas formas, pero no lo vemos. Nosotros sólo tenemos que aprender esas últimas tres líneas de código, de forma que nos podemos concentrar en el problema a resolver (ingresar dos números, sumarlos y mostrar el resultado) y no en las complejas operaciones internas que tiene que hacer el microprocesador.

En programación, la idea de simplificar un proceso complejo ignorando algunas de sus partes para comprender mejor lo que hay que realizar y así resolver un problema se conoce como *abstracción*<sup>3</sup>. Esto quiere decir que los lenguajes de programación pueden tener distintos niveles de abstracción:

- *Lenguajes de bajo nivel de abstracción*: permiten controlar directamente el *hardware* de la computadora, son específicos para cada tipo de máquina, y son más rígidos y complicados de entender para nosotros. El lenguaje ensamblador entra en esta categoría.
- *Lenguajes de alto nivel de abstracción*: diseñados para que sea fácil para los humanos expresar los algoritmos sin necesidad de entender en detalle cómo hace exactamente el hardware para ejecutarlos. El lenguaje que utilizaremos en este taller, R, es de alto nivel. Son independientes del tipo de máquina.
- *Lenguajes de nivel medio de abstracción*: son lenguajes con características mixtas entre ambos grupos anteriores.

<sup>3</sup>La abstracción no es una idea exclusiva de la programación. Se encuentra, también, por ejemplo, en el *arte abstracto*.



Figura 1.9: Distintos lenguajes de programación y sus logos.

Si bien podemos programar usando un lenguaje de alto nivel para que nos resulte más sencillo, *alguien* o *algo* debe traducirlo a lenguaje de máquina para que la computadora, que sólo entiende de ceros y unos, pueda realizar las tareas. Esto también es necesario incluso si programáramos en lenguaje ensamblador. Para estos procesos de traducción se crearon los *compiladores* e *intérpretes*.

Un *compilador* es un programa que toma el código escrito en un lenguaje de alto nivel y lo traduce a código de máquina, guardándolo en un archivo que la computadora ejecutará posteriormente (archivo ejecutable). Para ilustrar el rol del compilador, imaginemos que alguien que sólo habla español le quiere mandar una carta escrita en español a alguien que vive en Alemania y sólo habla alemán. Cuando esta persona la reciba, no la va a entender. Se necesita de un intermediario que tome la carta en español, la traduzca y la escriba en alemán y luego se la mande al destinatario, quien ahora sí la podrá entender. Ese es el rol de un *compilador* en la computadora. Ahora bien, el resultado de la traducción, que es la carta escrita en alemán, sólo sirve para gente que hable alemán. Si se quiere enviar el mismo mensaje a personas que hablen otros idiomas, necesitaremos hacer la traducción que corresponda. De la misma forma, el código generado por un compilador es específico para cada máquina, depende de su arquitectura.

Además de los compiladores, para realizar este pasaje también existen los *intérpretes*. Un intérprete es un programa que traduce el código escrito en lenguaje de alto nivel a código de máquina, pero lo va haciendo a medida que se necesita, es decir, su resultado reside en la memoria temporal de la computadora y no se genera ningún archivo ejecutable. Siguiendo con el ejemplo anterior, es similar a viajar a Alemania con un intérprete que nos vaya traduciendo en vivo y en directo cada vez que le queramos decir algo a alguien de ese país. En su implementación por defecto, el lenguaje R es interpretado, no compilado.

Concluyendo, gracias al concepto de la *abstracción* podemos escribir programas en un lenguaje que nos resulte fácil entender, y gracias al trabajo de los *compiladores* e *intérpretes* la computadora podrá llevar adelante las tareas necesarias.

Cada una de las acciones que componen al algoritmo son codificadas con una o varias *instrucciones*, expresadas en el lenguaje de programación elegido, y el conjunto de todas ellas constituye un *programa*. El programa se guarda en un *archivo* con un nombre generalmente dividido en dos partes por un punto, por ejemplo: `mi_primer_programa.R`. La primera parte es la *raíz* del nombre con la cual podemos describir el contenido del archivo. La segunda parte es indicativa del uso del archivo, por ejemplo, `.R` indica que contiene un programa escrito en el lenguaje R. El proceso general de ingresar o modificar el contenido de un archivo se denomina *edición*.

## 1.5. Errores de programación

Apenas iniciemos nuestro camino en el mundo de la programación nos daremos cuenta que tendremos siempre ciertos compañeros de viaje: los *errores*. Muchas veces nos pasará que queremos ejecutar nuestro código y el mismo no anda o no produce el resultado esperado. No importa cuán cuidadosos seamos, ni cuánta experiencia tengamos,

los errores están siempre presentes. Con el tiempo y práctica, vamos a poder identificarlos y corregirlos con mayor facilidad, pero probablemente nunca dejemos de cometerlos.

A los errores en programación se los suele llamar *bugs* (insecto o bicho en inglés) y el proceso de la corrección de los mismos se conoce como *debugging* (depuración)<sup>4</sup>. Se dice que esta terminología proviene de 1947, cuando una computadora en la Universidad de Harvard (la *Mark II*) dejó de funcionar y finalmente se descubrió que la causa del problema era la presencia de una polilla en un relé electromagnético de la máquina. Sin embargo, otros historiadores sostienen que el término ya se usaba desde antes.



Figura 1.10: La polilla (bug) encontrada por la científica de la computación Grace Hooper en la *Mark II* fue pegada con cinta en un reporte sobre el malfuncionamiento de la máquina.

A continuación se presenta una de las posibles clasificaciones de los errores que se pueden cometer en programación:

- *Errores de sintaxis.* Tal como el lenguaje humano, los lenguajes de programación tienen su propio vocabulario y su propia sintaxis, que es el conjunto de reglas gramaticales que establecen cómo se pueden combinar las distintas partes. Estas reglas sintácticas determinan que ciertas instrucciones están correctamente construidas, mientras que otras no. Cuando ejecutamos un programa, el compilador o el intérprete chequea si el mismo es sintácticamente correcto. Si hemos violado alguna regla, por ejemplo, nos faltó una coma o nos sobra un paréntesis, mostrará un mensaje de error y debemos editar nuestro programa para corregirlo. En estos casos, hay que interpretar el mensaje de error, revisar el código y corregir el error.
- *Errores lógicos.* Se presentan cuando el programa puede ser compilado sin errores pero arroja resultados incorrectos o ningún resultado. El software no muestra mensajes de error, debido a que, por supuesto, no sabe cuál es el resultado deseado, sino que sólo se limita a hacer lo que hemos programado. En estos casos hay que revisar el programa para encontrar algún error en su lógica. Este tipo de errores suelen ser los más problemáticos. Algunas ideas para enfrentarlos incluyen volver a pensar paso por paso lo que se debería hacer para solucionar el problema y compararlo con lo que se ha programado, agregar pasos para mostrar resultados intermedios o emplear herramientas especializadas de *debugging* (llamadas *debugger*) para explorar el código paso a paso hasta identificar el error.
- *Errores en la ejecución (runtime errors).* Se presentan cuando el programa está bien escrito, sin errores lógicos ni sintácticos, pero igualmente se comporta de alguna forma incorrecta. Se dan a pesar de que el programa anda bien en el entorno de desarrollo del programador, pero no cuando algún usuario lo utiliza en algún

<sup>4</sup>Algunos usan el término bug para referirse exclusivamente a errores lógicos

contexto particular. Puede ser que se intente abrir un archivo que no existe, que el proceso supere la memoria disponible, que tomen lugar operaciones aritméticas no definidas como la división por cero, etc.

Los errores en la programación son tan comunes, que un científico de la computación muy reconocido, Edsger Dijkstra, dijo una vez: "si la depuración es el proceso de eliminar errores, entonces la programación es el proceso de generarlos". Ante la presencia de uno, no hay más que respirar profundo y con paciencia revisar hasta encontrarlo y solucionarlo.



Figura 1.11: Encontrando un bug en un programa.

## 1.6. Procesador, ambiente y acciones

Hemos definido a un algoritmo como una lista de instrucciones para resolver un problema. En este contexto, se entiende por *procesador* a todo agente capaz de comprender los pasos de un algoritmo y ejecutar el trabajo indicado por el mismo. Para cumplir con el objetivo, el procesador emplea ciertos recursos que tiene a disposición. Todos los elementos disponibles para ser utilizados por el procesador constituyen su *entorno* o *ambiente*. Cada una de las instrucciones que componen el algoritmo modifican el entorno de alguna manera y se denominan *acciones*.

Ejemplificaremos estos conceptos con los siguientes ejemplos:

### Ejemplo 1

- *Problema*: preparar una tortilla de 6 huevos.
- *Entorno*: una mesa, una hornalla, una sartén, un plato, un tenedor, aceite, una fuente con huevos, un tarro de basura.
- *Procesador*: una persona adulta.
- *Acciones comprensibles por el procesador*: agarrar un huevo, romper un huevo en un plato, batir huevos, poner aceite, poner en la sartén, poner al fuego, retirar del fuego, tirar las cáscaras, encender el fuego.

¿Cuál es un algoritmo adecuado para solucionar este problema? Podría ser:

ALGORITMO: "Preparar una tortilla de 6 huevos"

COMENZAR

```

ROMPER seis huevos en un plato
TIRAR las cáscaras en el tacho
BATIR los huevos
CALENTAR aceite en la sartén
PONER el contenido del plato en la sartén
ESPERAR diez minutos
RETIRAR la tortilla del fuego
APAGAR el fuego

```

FIN

### Ejemplo 2

- *Problema:* calcular el factorial del número 5.
- *Entorno:* se dispone de una calculadora común.
- *Procesador:* una persona adulta.
- *Acciones comprensibles por el procesador:* pulsar teclas de la calculadora.

¿Cuál es un algoritmo adecuado para solucionar este problema? Podría ser:

ALGORITMO: "Calcular 5!"

COMENZAR

```
PULSAR [ON]
PULSAR [1]
PULSAR [X]
PULSAR [2]
PULSAR [X]
PULSAR [3]
PULSAR [X]
PULSAR [4]
PULSAR [X]
PULSAR [5]
PULSAR [=]
MOSTRAR la pantalla
```

FIN

Las acciones del algoritmo pueden clasificarse en función de su complejidad:

- *Acción primitiva:* acción sencilla directamente realizable por el procesador sin necesidad de contar con mayor información.
- *Acción compuesta:* acción integrada por una sucesión de acciones primitivas.

La descripción de un algoritmo en términos de acciones compuestas puede facilitar su comprensión, pero al desarrollar el programa será necesario descomponerlas en acciones primitivas que son las que realmente ejecuta el procesador. Por ejemplo, la acción compuesta en el ejemplo de la tortilla de “romper seis huevos en un plato” puede descomponerse en acciones más simples:

REPETIR 6 VECES

```
TOMAR un huevo
GOLPEAR el huevo para generar una fractura en la cáscara
EJERCER presión sobre la cáscara
DERRAMAR la clara y la yema sobre el plato
```

En el contexto de la resolución de un problema computacional, el *procesador* será la computadora; las acciones estarán dadas por las instrucciones disponibles en el lenguaje de programación utilizado o las que podamos crear en base al mismo; y el *ambiente* estará constituido por un conjunto de variables, estructuras de datos, funciones y otros elementos que serán de utilidad en la programación.

## 1.7. R y RStudio

Como dijimos anteriormente, pondremos en práctica los conceptos sobre programación utilizando **R**, un lenguaje orientado a la realización de procesos estadísticos y gráficos. A diferencia de muchos otros, este es un software libre y gratuito: se distribuye bajo la licencia *GNU GPLv2* que establece la libertad de usarlo con cualquier propósito, de ver cómo fue hecho, cómo funciona y modificarlo, de distribuir copias y crear contribuciones y agregados para que estén disponibles para otros<sup>5</sup>.

Si bien R será nuestro medio de comunicación con la computadora, vamos a usar otro programa que brinda algunas herramientas para facilitar nuestro trabajo de programación, es decir, vamos a usar un **entorno de desarrollo integrado** (o *IDE*, por *integrated development environment*). Un IDE es un programa que hace que la codificación sea más sencilla porque permite manejar varios archivos de código, visualizar el *ambiente* de trabajo, utilizar resaltado con colores para distintas partes del código, emplear autocompletado para escribir más rápido, explorar páginas de ayuda, implementar estrategias de depuración e incluso intercalar la ejecución de instrucciones con la

<sup>5</sup>R tiene una comunidad de usuarios muy activa, de las más diversas áreas, con muchos blogs, portales y encuentros en persona para buscar y recibir ayuda. Por ejemplo, existe el grupo de R en Rosario: (<https://renrosario.rbind.io>)

visualización de los resultados mientras avanzamos en el análisis o solución del problema. El IDE más popularmente empleado para programar con R es **RStudio** y será el programa que estaremos usando todo el tiempo.

Para instalar estos programas, se debe visitar las páginas oficiales de R y de RStudio, descargar los instaladores y ejecutarlos. En este enlace se presenta un video con la instalación completa.

### 1.7.1. Organización de RStudio

Cuando se abre RStudio se pueden visualizar cuatro paneles:

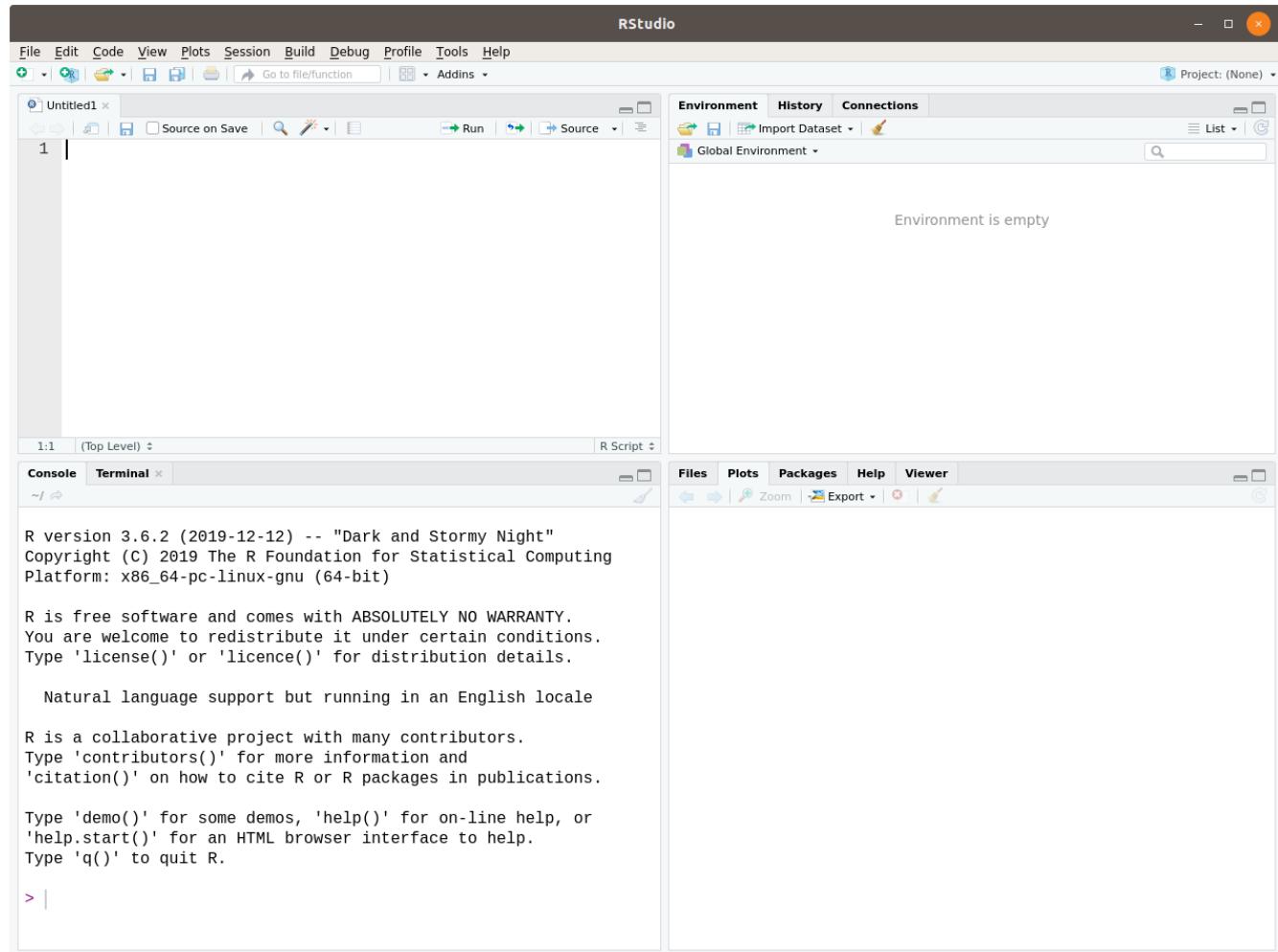


Figura 1.12: Captura de pantalla de RStudio

- A la izquierda:

- Arriba está el **editor de archivos**. Es el lugar donde escribiremos y editaremos nuestros programas, para luego guardarlos (una especie de *Bloc de Notas* o de un *Word* muy simple). La primera vez que se abre RStudio después de su instalación puede ser que este panel no esté presente, pero aparece si vamos a *File > New File > R Script*, con el atajo **Ctrl + Shift + N** o haciendo clic en el primer ícono de la barra de herramientas. Podemos abrir varios archivos a la vez.
- Abajo está la **consola**. Es la ventana que se comunica con R. A través de ella se envían las instrucciones a R para que las evalúe (también decimos, que las ejecute o corra) y se visualizan los resultados.

- Arriba a la derecha hay un panel con algunas pestañas:

- **Environment (ambiente)**: muestra todos los elementos que componen al ambiente o entorno.
- **History (historial)**: lista todas las instrucciones que R ha corrido anteriormente.
- **Otras más que no nos interesan por ahora**

- Abajo a la derecha hay otro panel con más pestañas:

- **Files:** explorador de archivos de la computadora
- **Plots:** ventana donde aparecen los gráficos si es que nuestro código produce alguno
- **Packages:** listado de los “paquetes” que tenemos instalados (ver más adelante)
- **Help:** manual de ayuda sobre todas las funciones de R
- **Viewer:** espacio para ver salidas de los programas con algún componente dinámico o interactivo.

### 1.7.2. Uso de la consola

Podemos usar la consola de R que encontramos en el panel de la izquierda para introducir allí nuestras instrucciones y al hacer **Enter** serán evaluadas, produciendo algún resultado. Por ejemplo, podemos hacer algunos cálculos matemáticos como dividir, multiplicar, sumar, restar, calcular potencias, logaritmos, raíces y mucho más:

```
1 + 2
[1] 3
5 * 3
[1] 15
exp(2)
[1] 7.389056
sqrt(100)
[1] 10
1 / 0
[1] Inf
(2 + 3i) * (3 + 6i)
[1] -12+21i
1i ^ 2
[1] -1+0i
```

Si bien podemos escribir nuestras instrucciones en la consola y dar **Enter** para que se ejecuten, en general queremos que queden escritas y guardadas en el archivo de código, por eso vamos a escribir nuestros programas en el panel de arriba a la izquierda. Una vez que escribimos una instrucción en el script, podemos *correrla* (es decir, enviarla a la consola para que se execute) haciendo clic en el ícono *Run* o con el atajo *Ctrl + Enter*. De esta forma, se *corre* la línea en la cual está el cursor o las líneas que hayamos seleccionado.

En todo lenguaje de programación existe un carácter especial que, al ser colocado al comienzo de una línea de código, le indica al software que dicha línea no debe ser evaluada. Esto se utiliza para incluir **comentarios**, es decir, líneas que expresan en español explicaciones o aclaraciones para nosotros mismos u otros que puedan utilizar nuestro código. También se utiliza para añadir encabezados con descripciones sobre el script, o indicar distintas secciones o partes en el programa. En R, este carácter especial es el símbolo numera (#). Si *corremos* líneas que empiezan con #, R no hará nada con ellas, las saltará. Por ejemplo

```
5^1
[1] 5
# 5^2
5^3
[1] 125
```

### 1.7.3. Diseño del sistema R

R Se divide en dos partes:

- La **base** (*R Base*), que se instala cuando descargamos el programa desde CRAN (“Comprehensive R Archive Network”). Contiene, entre otras cosas, una serie de herramientas básicas y fundamentales de R.
- **Paquetes adicionales.** Un paquete es un conjunto de archivos que se descarga de forma opcional desde CRAN u otros repositorios y que sirven para hacer alguna tarea especial. Por ejemplo, para poder hacer gráficos lindos, se puede usar un paquete que se llama `ggplot2`.

Como dijimos antes, en la pestaña *Packages* del panel de abajo a la derecha tiene el listado de todos los paquetes que ya están instalados (muchos vienen con R Base). Allí también hay un botón para instalar nuevos, aunque otra opción es correr la instrucción `install.packages("nombredelpaquete")`, por ejemplo, `install.packages("ggplot2")`. También es posible instalar paquetes publicados en otros repositorios<sup>6</sup>.

Un paquete se instala una sola vez, pero cada vez que lo queramos usar debemos *cargarlo* para que las herramientas que trae queden a nuestra disposición. Eso se hace con la instrucción `library("nombredelpaquete")`, por ejemplo, `library("ggplot2")`.

## 1.8. Guía de estilo

Es sumamente importante mantener la prolijidad en la escritura tanto del pseudocódigo como de los programas, para facilitar la lectura de los mismos, especialmente cuando estamos trabajando con problemas largos. Siempre hay que tener en cuenta de que cuando escribimos un programa, tenemos dos públicos potenciales: integrantes de nuestro equipo de trabajo que tienen leer el código y hacer sus propios aportes y nosotros mismos en el futuro, cuando retomemos código hecho en el pasado y necesitemos interpretar qué es lo que hicimos hacer.

Es por eso que se establecen conjuntos de reglas para controlar y unificar la forma de escribir programas, que se conocen como **guía de estilo**. Estas reglas cubren aspectos como, por ejemplo, la forma de escribir comentarios en el código, la utilización de espacios o renglones en blanco, el formato de los nombres para los elementos que creamos nosotros mismos (como las funciones) y para los archivos que generamos, etc. Una *guía de estilo* no indica la única forma de escribir código, ni siquiera la forma correcta de hacerlo, sino que establece una convención para que todos trabajen de la misma forma, basándose en costumbres que sí se ha visto que pueden tener más ventajas que otras.

Por ejemplo, para programar en R, existe una guía de estilo llamada The tidyverse style guide, que es la que utilizan la gente de Google y de RStudio. En este curso vamos a adherir a sus recomendaciones. Si bien es una lectura muy interesante, particularmente si tenés intenciones de profundizar tus conocimientos sobre programación en R, no es necesario que lean dicha guía completa. Por ahora es suficiente con que imiten con atención el estilo que usamos en los ejemplos provistos en esta guía.

Recordemos siempre que seguir un buen estilo para programar es como hacer uso de una correcta puntuación cuando escribimos, podemos entendernos sin ella, pero es muchísimo más difícil leerlo que escribirlo si no la respetamos no?

---

<sup>6</sup>Por ejemplo, es muy común descargar paquetes en desarrollo o en experimentación que estén disponibles en la plataforma Github. Estos paquetes se instalan especificando el nombre de la cuenta de Github de quien lo haya publicado y el nombre del paquete, por ejemplo: `devtools::install_github("mpru/karel")`. Para esto previamente hay que haber instalado desde CRAN el paquete `devtools` con `install.packages("devtools")`.



## Unidad 2

# Objetos y operadores

Hemos mencionado que para resolver un problema computacional se necesita de un *procesador*, capaz de entender y ejecutar ciertas *acciones* a partir de ciertos elementos disponibles en el *ambiente* o *entorno*. En este capítulo vamos a aprender acerca de dichos elementos, a los que llamamos *objetos*.

### 2.1. Objetos

Los **objetos** son las distintas piezas de información, o más sencillamente, *datos*, que componen al *entorno* y que el procesador debe manipular para resolver una tarea. Las distintas acciones del algoritmo van creando o modificando a los objetos del entorno. A medida que avancemos, veremos que hay distintas clases de objetos, algunos con estructuras más simples y otros más complejos. Es más, cada lenguaje de programación propone su propio catálogo de clases de objetos y cada programador puede crear otras nuevas. Sin embargo, en general todos los lenguajes tienen en común el hecho de que sus objetos pueden almacenar los siguientes tres tipos de datos básicos, conocidos como **tipos de datos primitivos**:

- **Datos de tipo numérico:** representan valores escalares de forma numérica y permiten realizar operaciones aritméticas comunes. Ejemplo: 230, 2.
- **Datos de tipo carácter:** representan texto y no es posible hacer operaciones matemáticas con ellos. Representamos estos valores entre comillas. Ejemplo: "hola", "chau123"
- **Datos de tipo lógico:** pueden tomar dos valores (*VERDADERO* o *FALSO*), ya que representan el resultado de alguna comparación entre otros objetos. En R, estos valores son **TRUE** y **FALSE**, escritos sin comillas.

De manera general, al nombre de un objeto se le dice **identificador**, el cual es una secuencia de caracteres alfanuméricos que sirve para identificarlo a lo largo del algoritmo. Nombrar los objetos hace posible referirse a los mismos. La elección de los identificadores es una tarea del programador, pero cada lenguaje tiene sus propias reglas. Por ejemplo, en R los nombres de los objetos:

- Deben empezar con una letra o un punto (no pueden empezar con un número).
- Sólo pueden contener letras, números, guiones bajos y puntos (se puede forzar a R para que acepte nombres con otros caracteres, pero no es aconsejable).
- No se pueden usar las siguientes palabras como nombres, ya que son palabras claves reservadas para R: **break**, **else**, **FALSE**, **for**, **function**, **if**, **Inf**, **NA**, **NaN**, **next**, **repeat**, **return**, **TRUE**, **while**.

Es aconsejable elegir un nombre que sea representativo de la información que va a guardar el objeto, ya que esto facilita la lectura y la comprensión tanto del algoritmo como del programa. Por ejemplo, si se necesita un objeto para guardar el valor numérico del precio de algún producto, el identificador **p** sería una mala elección, mientras que **precio** sería mejor. Si se necesitan varios identificadores para distinguir los precios de diversos productos, podríamos usar algo como **precio\_manzana**, **precio\_banana**, etc. Otra opción podría ser **preciomanzana** o **precioManzana**, pero en este curso seguiremos la convención de usar guiones bajos para facilitar la lectura de los nombres elegidos. No sería posible usar como identificador a **precio manzana**, puesto que un nombre no puede tener espacios.

Ciertos objetos almacenan temporalmente un valor durante la ejecución de un proceso y su contenido puede cambiar mientras corre el programa. Este tipo de objetos reciben el nombre de **variables**. Por ejemplo, en un programa creado para un comercio puede existir un objeto llamado **stock** (identificador) de tipo numérico que representa la cantidad de artículos disponibles y cuyo valor se modifica cada vez que se registra una nueva venta. Podemos pensar a una variable como una caja etiquetada con un nombre (su identificador) y que guarda un valor (numérico, lógico

o de carácter).

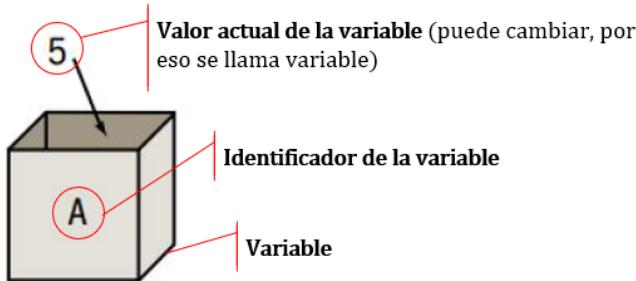


Figura 2.1: La variable A contiene al valor 5.

Si un objeto tiene siempre necesariamente un valor fijo, se dice que es una **constante**. El valor es siempre el mismo para cualquier ejecución del programa, es decir, no puede cambiar de una ejecución a otra. Son ejemplos de constantes el número de meses del año, las constantes matemáticas tales como el número  $\pi$ , los factores de conversión de unidades de medida, etc. Las constantes pueden usarse literalmente, es decir, usando explícitamente el valor, o también a través de un objeto que asocie un identificador al valor constante (por ejemplo, asociar el nombre pi a la constante 3.14159265).

### 2.1.1. Declaración de variables y constantes

Al expresar nuestros algoritmos en pseudocódigo tomaremos la costumbre de declarar al inicio del mismo las variables y constantes necesarias para resolver el problema, explicitando su identificador y determinando el tipo de valor que guarda. Muchos lenguajes de programación utilizan esta declaración para reservar en la memoria de la computadora un espacio para almacenar la información correspondiente de manera adecuada.

Veamos un ejemplo:

```
ALGORITMO: "Calcular área de un círculo"
COMENZAR
    CONSTANTE numérica pi
    VARIABLE numérica radio, area
    ...restantes acciones del algoritmo...
FIN
```

Una vez que una variable o constante ha sido definida con un tipo particular, es incorrecto intentar asignarle un valor de otro tipo, por ejemplo, asignarle a la variable `radio` del ejemplo anterior un valor de tipo carácter. Esto generaría ambigüedad, además de que en ciertos lenguajes de programación produciría un error.

R es un lenguaje dinámico que no requiere la declaración previa de las variables que serán utilizadas, sino que estas pueden definirse dinámicamente a lo largo del programa. Por eso, incluiremos declaración de variables y constantes sólo en los algoritmos y no cuando estos sean traducidos a R. Al no necesitar una declaración previa, en R es posible asignar, por ejemplo, un valor de tipo carácter a un identificador que anteriormente contenía un valor de tipo numérico, pero trataremos de evitar esto.

### 2.1.2. Acción de asignación

Para hacer que una variable guarde un determinado valor se recurre a una **acción de asignación**. Mediante asignaciones podemos dar valores iniciales a las variables, modificar el valor que tenían hasta el momento o guardar en ellas resultados intermedios o finales del algoritmo.

En pseudocódigo expresaremos a la asignación como se muestra en estos ejemplos:

```
ALGORITMO: "Calcular área de un círculo"
COMENZAR
    CONSTANTE numérica pi
    VARIABLE numérica radio, area
    pi <- 3.14159265358979
```

```

radio <- 3
...restantes acciones del algoritmo...
FIN

```

Si intervienen variables o constantes en la expresión a la derecha de una acción de asignación, se usará el valor que tengan las mismas en ese momento. Por ejemplo, la siguiente secuencia de acciones en un algoritmo:

```

var1 <- 2
var2 <- var1
var1 <- 7
var3 <- var1

```

resultará en que las variables `var1`, `var2` y `var3` tengan almacenados los valores 7, 2 y 7 respectivamente. Un caso particular se da cuando a una variable se le asigna el valor de una operación de la que forma parte la misma variable. Por ejemplo:

```

stock <- 43
ventas <- 29
compras <- 12
stock <- stock - ventas + compras

```

### 2.1.3. Creación y manejo de objetos en R

En R también se usa el operador “flechita” para asignar valores a variables. Como lo vamos a usar muchísimas veces, es conveniente recordar su *shortcut*: `Alt + -` (teclas `Alt` y guion medio). Como ya mencionamos antes, R es un lenguaje dinámico, lo cual significa que no tenemos que declarar las variables antes de asignarles un valor. Por ejemplo, si necesitamos registrar el peso y lugar de nacimiento de una persona, solamente tenemos que hacer:

```

lugar_nacimiento <- "Bombal"
peso_nacimiento <- 3.423

```

En este caso, `lugar_nacimiento` es un objeto de tipo carácter, ya que la información que contiene es “Bombal”, y `peso_nacimiento` es un objeto numérico. Vale aclarar que en R el punto decimal se representa con el punto `.` y no con la coma `,`, como solemos escribir habitualmente, por lo que el número indicado se lee “tres coma cuatrocientos veintitrés” y no “tres mil cuatrocientos veintitrés”. Asimismo, no se usan separadores de miles al escribir los números.

Por otro lado, R provee algunas funciones para poder averiguar en cualquier momento qué tipo de dato está almacenado en alguna variable. Todavía no hemos definido formalmente a las *funciones*, pero por ahora nos alcanza con saber que son otro tipo de objetos que cuando las usamos se encargan de cumplir con alguna tarea específica. Las funciones también tienen un nombre (*identificador*) y están seguidas por un par de paréntesis, dentro de los cuales se escriben opciones para que ellas operen. La función `class()` es la que nos dice qué tipo de dato hay en una variable:

```

var1 <- 2
var2 <- "Hola Mundo"
var3 <- TRUE
class(var1)

```

```
[1] "numeric"
```

```
class(var2)
```

```
[1] "character"
```

```
class(var3)
```

```
[1] "logical"
```

También hay algunas funciones que devuelven `TRUE` o `FALSE` a modo de respuesta cuando le preguntamos a R si una variable tiene un dato de tipo numérico, carácter o lógico:

```
is.numeric(var1)
```

```
[1] TRUE
```

```
is.numeric(var2)
```

```
[1] FALSE
```

```
is.character(var3)
```

```
[1] FALSE
```

```
is.logical(var3)
```

```
[1] TRUE
```

Todos los objetos que vamos definiendo en nuestro algoritmo y, posteriormente, en nuestro programa, forman parte del *ambiente*. En Rstudio podemos ver listados todos los objetos presentes en el ambiente en la pestaña *Environment* del panel superior derecho. También podemos ver en la consola un listado de todos los nombres de los objetos que existen en el ambiente con la función `ls()`, por ejemplo:

```
ls()
```

```
[1] "var1" "var2" "var3"
```

Es probable que mientras estamos escribiendo el código, necesitemos probar si algunas partes funcionan y para eso corremos algunas líneas de código, creando objetos en el ambiente. Entre intento e intento, probablemente necesitemos borrar alguno o todos esos objetos que se crearon, para poder comenzar con un ambiente libre. Si deseamos borrar todos los objetos del ambiente podemos correr la sentencia `rm(list = ls())` o hacer clic en el ícono de la escoba en el panel *Environment*. Si queremos eliminar sólo un objeto debemos ejecutar la función `rm()`, indicando entre paréntesis el identificador del objeto que deseamos borrar, por ejemplo:

```
rm(var1)
```

Todos los objetos generados viven temporalmente en la memoria de la computadora mientras dure la sesión de R en la que estamos trabajando. Si cerramos R, toda esa información desaparecerá. Muchas veces eso es algo deseable: una vez finalizado el programa, guardamos algún resultado que nos interese (por ejemplo, un conjunto de datos en un archivo de Excel) y todos los objetos del ambiente que fuimos necesitando en el camino son descartados. Sin embargo, en otras oportunidades nos interesa guardar de forma permanente en la computadora una copia de todo lo que se encuentra en el ambiente en un archivo dentro de alguna carpeta. Los archivos que contienen los objetos creados en R tienen extensión **.RData**. Estos archivos se generan con el ícono de guardar en la pestaña *Environment* o usando la función `save.image()`, que necesita que escribamos entre los paréntesis la carpeta donde guardaremos el archivo y el nombre elegido para el mismo, por ejemplo:

```
save.image("C:/Users/Marcos/Documentos/Facultad/objetos.RData")
```

Si por el contrario necesitamos *importar* al ambiente objetos que estén guardados en algún lugar de nuestra computadora en un archivo **.RData**, podemos usar el ícono de abrir en la pestaña *Environment* o la función `load()`, por ejemplo:

```
load("C:/Users/Marcos/Documentos/Facultad/objetos.RData")
```

## 2.2. Operadores

El desarrollo de un algoritmo involucra la necesidad de efectuar operaciones de distinto tipo entre los valores guardados en los objetos: suma, resta, concatenación de caracteres, comparaciones, etc. Los elementos que describen el tipo de operación a realizar entre dos objetos se denominan **operadores**.

### 2.2.1. Operadores aritméticos

Los operadores aritméticos permiten realizar operaciones matemáticas con datos de tipo numérico. A continuación presentamos su simbología más comúnmente empleada a la hora de expresarlos en pseudocódigo, junto con sus

equivalentes en el lenguaje R:

Cuadro 2.1: Operadores aritméticos.

Operación	Operador en pseudocódigo	Operador en R	Ejemplo	Rtdo para x <- 7, y <- 3
Suma	+	+	x + y	10
Resta	-	-	x - y	4
Multiplicación	*	*	x * y	21
División	/	/	x / y	2.33
Potenciación	$\hat{}$	$\hat{}$	x $\hat{}$ y	343
División entera	DIV	% / %	x % / %	2
División módular (resto de la división)	MOD	% %	x % % y	1

Los operadores aritméticos actúan con un orden de prioridad establecido, también conocido como *orden de evaluación* u *orden de precedencia*, tal como estamos acostumbrados en matemática. Las expresiones entre paréntesis se evalúan primero. Si hay paréntesis anidados se evalúan desde adentro hacia afuera. Dentro de una misma expresión, en R los operadores se evalúan en este orden:

1. Potenciación ( $\hat{}$ )
2. División entera y módulo (% / %, % %, y cualquier otro operador especial del tipo % . . . %)
3. Multiplicación y división (\*, /)
4. Suma y resta (+, -)

Si la expresión presenta operadores con igual nivel de prioridad, se evalúan de izquierda a derecha. Veamos algunos ejemplos:

Cuadro 2.2: Ejemplos de operaciones aritméticas según el orden de precedencia de R.

Operación	Resultado
4 + 2 * 4	12
23 * 2 / 5	9.2
3 + 5 * (10 - (2 + 4))	23
2.1 * 1.5 + 12.3	15.45
2.1 * (1.5 + 12.3)	28.98
1 % % 4	1
8 * (7 - 6 + 5) % % (1 + 8 / 2) - 1	7

## 2.2.2. Operadores relacionales o de comparación

Los operadores relacionales sirven para comparar dos valores de cualquier tipo y dan como resultado un valor lógico: *VERDADERO* (T o TRUE en R) o *FALSO* (F o FALSE en R).

Cuadro 2.3: Operadores relacionales o de comparación.

Comparación	Operador en pseudocódigo	Operador en R	Ejemplo	Rtdo para x <- 7, y <- 3
Mayor que	>	>	x > y	TRUE
Menor que	<	<	x < y	FALSE
Mayor o igual que	>=	>=	x >= y	TRUE
Menor o igual que	<=	<=	x <= y	FALSE
Igual a	= o ==	==	x == y	FALSE
Distinto a	!= o !=	!=	x != y	TRUE

Otros ejemplos:

```
a <- 3
b <- 4
d <- 2
e <- 10
f <- 15
(a * b) == (d + e)
```

[1] TRUE

```
(a * b) != (f - b)
```

[1] TRUE

Es interesante notar que primero se evalúan las operaciones a cada lado de los operadores relacionales y luego se hace la comparación. Es decir, **los operadores aritméticos preceden a los relacionales en el orden de prioridad**. Por eso, en los ejemplos anteriores no eran necesarios los paréntesis y podríamos poner directamente:

```
a * b == d + e
```

[1] TRUE

```
a * b != f - b
```

[1] TRUE

Si bien en pseudocódigo podemos usar tanto = o == para probar la igualdad entre dos elementos, **en R no debemos usar =** para este fin, puesto que = no es un operador de comparación sino de asignación, parecido al <-.

### 2.2.3. Operadores lógicos

Mientras que los operadores relacionales comparan cualquier tipo de valores, los operadores lógicos sólo toman operandos de tipo lógico y producen también un resultado lógico. Los operadores lógicos que utilizaremos son:

Cuadro 2.4: Operadores lógicos.

Operación	Operador en pseudocódigo	Operador en R	Ejemplo	Rtdo para x <- T, y <- F
Conjunción	Y	&&	x && y	FALSE
Disyunción	O		x    y	TRUE
Negación	NO o $\neg$	!	!x	FALSE

- La operación de conjunción (Y) devuelve un valor **VERDADERO** sólo si son verdaderas **ambas** expresiones que vincula. Ejemplo: (3 > 2) Y (3 > 5) resulta en VERDADERO Y FALSO y esto es FALSO.
- La operación de disyunción (O) devuelve un valor **VERDADERO** si **al menos una** de las dos expresiones que vincula es verdadera. Ejemplo: (3 > 2) O (3 > 5) resulta en VERDADERO O FALSO y esto es VERDADERO.
- La operación de negación (NO) niega un valor lógico, es decir, devuelve el opuesto. Ejemplo: NO (3 > 2) resulta en NO VERDADERO y esto es FALSO.

Tanto para la conjunción como para la disyunción, R provee dos operadores, repitiendo o no el símbolo correspondiente: **&&** vs **;** **||** vs **!**. Hay una diferencia entre ellos que por ahora no viene al caso, pero vamos a señalar que por ahora estaremos usando las versiones presentadas anteriormente: **&&** y **||**.

La **tabla de verdad** o **tabla de valores de verdad** se utiliza para mostrar los resultados de estas operaciones lógicas:

Cuadro 2.5: Operadores relacionales o de comparación.

Valor 1	Operador	Valor 2	Resultado
F	Y	F	F
F	Y	V	F

Valor 1	Operador	Valor 2	Resultado
V	Y	F	F
V	Y	V	V
F	O	F	F
F	O	V	V
V	O	F	V
V	O	V	V
	NO	F	V
	NO	V	F

Con estos operadores es posible construir evaluaciones lógicas más elaboradas como los siguientes ejemplos:

- Evaluar si el valor de `ancho` está entre 5 y 7: (`ancho > 5`) Y (`ancho < 7`).

```
ancho <- 6.4
(ancho > 5) && (ancho < 7)
```

[1] TRUE

- Establecer si una persona estudia Estadística o Economía: (`carrera == "Estadística"`) O (`carrera == "Economía"`).

```
carrera <- "Administración"
(carrera == "Estadística") || (carrera == "Economía")
```

[1] FALSE

- Determinar si una persona no estudia Estadística: NO (`carrera == "Estadística"`).

```
carrera <- "Administración"
!(carrera == "Estadística")
```

[1] TRUE

- Verificar que el valor guardado en `x` no sea igual a 2 ni a 3:

- Opción correcta 1: (`x != 2`) Y (`x != 3`)

```
# Da verdadero porque x no es ni 2 ni 3
x <- 10
(x != 2) && (x != 3)
```

[1] TRUE

```
# Da falso porque x es igual a 3
x <- 3
(x != 2) && (x != 3)
```

[1] FALSE

- Opción correcta 2: NO ((`x == 2`) O (`x == 3`))

```
# Da verdadero porque x no es ni 2 ni 3
x <- 10
!(x == 2 || x == 3)
```

[1] TRUE

```
# Da falso porque x es igual a 3
x <- 3
!(x == 2 || x == 3)
```

[1] FALSE

- Opción incorrecta: (`x != 2`) O (`x != 3`)

```
# Da verdadero, porque al ser x igual a 3, es distinto de 2,
# haciendo que la primera parte sea verdadera
x <- 3
(x != 2) || (x != 3)
```

[1] TRUE

Este último ejemplo se relaciona con las **Leyes de Morgan**: siendo b y c valores lógicos, se tiene:

- $\text{NO } (b \text{ O } c)$  es equivalente a  $\text{NO } b \text{ Y } \text{NO } c$ .
- $\text{NO } (b \text{ Y } c)$  es equivalente a  $\text{NO } b \text{ O } \text{NO } c$ .

Es importante notar que todos los paréntesis usados en el código de R del ejemplo 4 son innecesarios, puesto que **los operadores relacionales preceden a los lógicos en el orden de prioridad**. Sin embargo, a veces preferimos usar paréntesis para que la lectura sea más sencilla. En el siguiente ejemplo, ambas expresiones son equivalentes:

```
ancho <- 6.4
(ancho > 5) && (ancho < 7)
```

[1] TRUE

```
ancho > 5 && ancho < 7
```

[1] TRUE

Para pensar: predecir el resultado de las siguientes operaciones y luego verificar:

```
x <- 2
y <- -2
x > 0 && y < 0
x > 0 || y < 0
!(x > 0 && y < 0)
```

### 2.2.3.1. Evaluación en cortocircuito

Para evaluar la operación de conjunción  $x \&& y$ , en R se comienza por evaluar la expresión del primer operando  $x$  y si su resultado es FALSE ya no se evalúa la expresión  $y$  del segundo operando. Esto es porque si  $x$  es FALSE, el resultado de  $x \&& y$  ya no depende de  $y$ , será siempre FALSE. Por este motivo se dice que el operador **&&** se evalúa en *cortocircuito*. La evaluación en cortocircuito evita realizar operaciones innecesarias<sup>1</sup>.

Por ejemplo:

```
x <- 1
y <- 2

# La primera parte da TRUE, se continúa con la segunda, pero da error porque no
# existe un objeto llamado z
(y > x) && (x > z)
```

Error in eval(expr, envir, enclos): object 'z' not found

```
# La primera parte da FALSE, entonces toda la operación será FALSE, no se
# continúa con la segunda parte, con lo cual no se intenta usar el objeto
# inexistente z y no hay error
(y < x) && (x > z)
```

[1] FALSE

La operación de disyunción también se evalúa en cortocircuito, es decir, si se encuentra que uno de los operandos es TRUE, no hace falta evaluar los restantes, puesto que el resultado general será TRUE:

<sup>1</sup>El otro operador de conjunción, **&**, no evalúa en cortocircuito, además de poseer otras diferencias.

```
# Es TRUE porque la primera parte es TRUE, sin evaluar la segunda, que daría
# error
(y > x) || (x > z)
```

```
[1] TRUE
```

```
# Como la primera parte es FALSE, debe evaluar la segunda, no encuentra a z y da
# error
(x > y) || (x > z)
```

```
Error in eval(expr, envir, enclos): object 'z' not found
```

### 2.2.3.2. Orden de precedencia completo en R

Resumiendo la información anterior, a continuación se presenta el orden completo de precedencia de los operadores en R que utilizaremos (hay algunos más que pueden ver en `?Syntax`):

Cuadro 2.6: Orden de precedencia de los operadores en R.

Orden	Operaciones	Operadores
1	Potenciación	$\wedge$
2	Signo de un número (ej: -3)	+ -
3	División entera y resto	% / % % %
4	Multiplicación y división	* /
5	Suma y resta	+ -
6	Operadores de comparación	< > <= >= == !=
7	Negación	!
8	Conjunción	&& &
9	Disyunción	
10	Asignación (izquierda a derecha)	<-

Dentro de una misma expresión, operadores con igual prioridad se evalúan de izquierda a derecha.

## 2.3. Entrada y salida de información

En la resolución de problemas puede ser necesario que alguna fuente externa (como un usuario del programa) provea información. En estos casos se debe registrar dicha información como un valor que debe ser asignado a una variable. Cuando escribamos nuestros algoritmos en pseudocódigo, para esto utilizaremos la acción **LEER**. Cuando deseamos mostrar un resultado en un mensaje empleamos la acción **ESCRIBIR**. Las palabras o frases literales que se desean mostrar en el mensaje deben estar encerradas entre comillas porque son cadenas de texto, mientras que si se desea mostrar el valor de una variable se debe escribir su identificador sin comillas.

Por ahora, en R ejecutaremos a la acción **LEER** mediante la asignación directa de un valor a una variable a través del operador `<-`. La acción **ESCRIBIR** puede ser ejecutada a través de la función `cat()` si se quiere mostrar una frase compuesta. En los casos en los que sólo interesa mostrar un valor (sin escribir una frase) no será necesario usar `cat()`, ya que sencillamente al correr el nombre de un objeto, su valor es mostrado en la consola. En otros casos usaremos la función `print()`. Ya iremos viendo la utilidad de cada una de estas opciones.

Vamos a completar el ejemplo del algoritmo para el cálculo del área de un círculo, integrando todo lo mencionado anteriormente:

**Pseudocódigo:**

```
ALGORITMO: "Calcular área de un círculo"
COMENZAR
    CONSTANTE numérica pi
    VARIABLE numérica radio, area
    pi <- 3.14159265358979
```

```

LEER radio
area <- pi * radio^2
ESCRIBIR "El área del círculo es " area
FIN

```

En R:

```

# PROGRAMA: "Calcular área de un círculo" -----
pi <- 3.14159265358979
radio <- 5
area <- pi * radio^2
cat("El área del círculo es", area)

```

El área del círculo es 78.53982

*Nota:* pi ya es una constante incorporada en R Base, en realidad no es necesario crear esta constante y asignarle valor.

## 2.4. Directorio de trabajo

Antes de terminar este capítulo vamos a presentar un concepto fundamental acerca de cómo se relaciona R con el sistema operativo de nuestra computadora para poder tener acceso a nuestros archivos o para generar otros nuevos. Al finalizar la sección 2.1.3 vimos ejemplos en los que se utilizó R para, en primer lugar, generar un archivo llamado `objetos.RData` con todos los objetos existentes en nuestro ambiente de trabajo y, en segundo lugar, cargar la información que dicho archivo contiene:

```

save.image("C:/Users/Marcos/Documentos/Facultad/objetos.RData")
load("C:/Users/Marcos/Documentos/Facultad/objetos.RData")

```

En ambos casos, dentro de las funciones `save.image()` y `load()` se tuvo que escribir la dirección completa que representa cuál es la ubicación exacta de dicho archivo en la computadora: `C:/Users/Marcos/Documentos/Facultad/objetos.RData`. Esto significa que el archivo se encuentra en la carpeta `Facultad`, que a su vez está dentro de la carpeta `Documentos`, dentro de `Marcos` y dentro de `Users`, en el disco C de la computadora.

Expresiones como `C:/Users/Marcos/Documentos/Facultad/objetos.RData` reciben el nombre de **ruta informática** o *path* y sirven para referenciar de manera exacta la localización ya sea de una carpeta o de un archivo en particular dentro del sistema de archivos que maneja el sistema operativo de la computadora. Un *path* está compuesto por todos los nombres de los directorios que ordenadamente nos permiten llegar hasta la carpeta o archivo de interés, separados por un carácter que dependiendo del sistema operativo puede ser una barra diagonal / o una barra inversa \.

R siempre está mirando a alguna carpeta en particular dentro de la computadora, la cual recibe el nombre **directorio de trabajo** (o *working directory, wd*). Por ejemplo, en este momento y en mi computadora, R está posando su atención en una carpeta que se llama `introprog` y cuya ruta puedo descubrir con la función `getwd()`, que significa “obtener (get) el directorio de trabajo (wd)”:

```
getwd()
```

```
[1] "/home/marcos/GitProjects/introprog"
```

Esto quiere decir que R puede ver y acceder de manera directa a todos los archivos que hay allí, sin necesidad de escribir la ruta completa. Por ejemplo, si en lugar de ejecutar:

```
save.image("C:/Users/Marcos/Documentos/Facultad/objetos.RData")
```

ejecuto sencillamente:

```
save.image("objetos.RData")
```

lo que ocurre es que el nuevo archivo se generará en mi directorio de trabajo (carpeta `introprog`) y no en la carpeta `Facultad`. Del mismo modo, si ejecuto:

```
load("objetos.RData")
```

el software va a buscar el archivo `objetos.RData` en mi directorio de trabajo (carpeta `introprog`) y va a cargar su contenido al ambiente de trabajo. Si en dicha carpeta no existe un archivo con ese nombre, obtendremos un mensaje de error muy famoso:

```
cannot find file 'objetos.RData', probable reason 'No such file or directory'
```

El directorio de trabajo por default suele ser la carpeta *Documentos* o alguna equivalente y es la que vemos en el panel **Files** de RStudio. Podemos cambiar el directorio de trabajo por cualquier otra carpeta en la que queramos estar trabajando con la función `setwd()` (“setear el working directory”):

```
setwd("/home/marcos/documents/introprog/tp1")
getwd()
"/home/marcos/documents/introprog/tp1"
```

Por ejemplo, si estamos resolviendo un trabajo práctico para el cual tenemos varios archivos necesarios guardados en la carpeta `tp1`, tenemos dos opciones:

1. Sin importar cuál es nuestro directorio de trabajo, hacer referencia a dichos archivos con *paths* completos, por ejemplo:

```
load("/home/marcos/documents/introprog/tp1/objetos.RData")
```

2. Setear como directorio de trabajo a la carpeta del trabajo práctico y hacer un uso directo de los archivos que se encuentren allí:

```
setwd("/home/marcos/documents/introprog/tp1")
load("objetos.RData")
```

Es importante recordar lo siguiente: al escribir *paths*, R sólo reconoce como caracteres delimitadores entre los nombres de carpetas a la barra diagonal / o a dos barras invertidas \\\. Quienes usan el sistema operativo Windows notarán que en el explorador de archivos, las rutas están delimitadas con una sola barra invertida \. En el contexto de R, debe ser reemplazada por dos barras invertidas o por una sola barra diagonal.



# Unidad 3

## Estructuras de control

Como mencionamos anteriormente, un algoritmo está compuesto por una sucesión ordenada de comandos que se ejecutan uno detrás de otro. Sin embargo, con frecuencia es necesario recurrir a comandos especiales que alteran o controlan el orden en el que se ejecutan las acciones. Llamamos **estructuras de control del flujo de las acciones** al conjunto de reglas que permiten controlar el flujo de las acciones de un algoritmo o programa. Las mismas pueden clasificarse en **secuenciales, condicionales e iterativas**.

### 3.1. Estructuras de control secuenciales

Las **estructuras secuenciales** están compuestas por un número definido de acciones que se ubican en un orden específico y se suceden una tras otra. Los ejemplos que hemos discutido anteriormente están conformados por este tipo de estructura.

### 3.2. Estructuras de control condicionales

En algunas partes de un algoritmo puede ser útil detenerse a hacer una pregunta porque se llegó a una situación en la que puede haber una o más opciones disponibles para continuar. Dependiendo de la respuesta a la pregunta, que siempre deberá ser **VERDADERO (TRUE)** o **FALSO (FALSE)**, el algoritmo seguirá ciertas acciones e ignorará otras. Estas preguntas y respuestas representan procesos de toma de decisión que conducen a diferentes caminos dentro del algoritmo, permitiéndonos que la solución para el problema en cuestión sea flexible y se adapte a distintas situaciones. Este tipo de estructuras de control de las acciones reciben el nombre de **condicionales** (o *estructuras de selección*) y pueden ser **simples, dobles y múltiples**.

#### 3.2.1. Estructuras condicionales simples

Postulan una evaluación lógica y, si su resultado es **VERDADERO**, se procede a ejecutar las acciones encerradas por esta estructura. Se describen en pseudocódigo con la siguiente sintaxis:

```
SI <condición> ENTONCES
    Acción/es
FIN SI
```

La palabra **SI** indica el comando de evaluación lógica, **<condición>** indica la condición a evaluar y **Acción/es** son las instrucciones que se realizarán sólo si se cumple la condición, es decir, si la evaluación resulta en **VERDADERO**. Si la condición no se verifica, no se ejecuta ninguna acción y el algoritmo sigue su estructura secuencial a continuación del **FIN SI**.

En R, la estructura que nos permite realizar esto es:

```
if (<condición>) {
  ...código para ejecutar acciones...
}
```

Por ejemplo, el siguiente algoritmo registra la edad de una persona y, en el caso de que sea mayor de edad, avisa que puede votar en las elecciones provinciales de Santa Fe:

ALGORITMO: "Analizar edad para votar"

COMENZAR

```
VARIABLE numérica edad
LEER edad
SI edad >= 18 ENTONCES
    ESCRIBIR "Edad = " edad " años: puede votar"
FIN SI
FIN
```

```
# Programa: "Analizar edad para votar" -----
edad <- 21
if (edad >= 18) {
  cat("Edad =", edad, "años: puede votar")
}
```

Edad = 21 años: puede votar

Notar que si bien el uso de sangrías en el código es opcional, decidimos emplearlo para facilitar su lectura. Mantener la prolijidad en nuestros programas es esencial.

### 3.2.2. Estructuras condicionales dobles

Este tipo de estructura añade una acción a ejecutarse en el caso de que la condición evaluada no se verifique (es decir, devuelve el valor FALSO). La sintaxis es:

```
SI <condición> ENTONCES
  Acción/es
SI NO
  Acción/es
FIN SI
```

La palabra ENTONCES antecede a las acciones que se realizan si se cumple la condición y la expresión SI NO a las que se realizan si no se verifica la misma.

En R se utiliza el comando `else`:

```
if (<condición>) {
  ...código para ejecutar acciones...
} else {
  ...código para ejecutar acciones...
}
```

Retomando el ejemplo anterior:

ALGORITMO: "Analizar edad para votar"

COMENZAR

```
VARIABLE numérica edad
LEER edad
SI edad >= 18 ENTONCES
    ESCRIBIR "Edad = " edad " años: puede votar"
SI NO
    ESCRIBIR "Edad = " edad " años: no puede votar"
FIN SI
FIN
```

```
# Programa: "Analizar edad para votar" -----
edad <- 21
if (edad >= 18) {
```

```

cat("Edad =", edad, "años: puede votar")
} else {
    cat("Edad =", edad, "años: no puede votar")
}

```

Edad = 21 años: puede votar

### 3.2.3. Estructuras condicionales múltiples o anidadas

Permiten combinar varias estructuras condicionales para establecer controles más complejos sobre el flujo de las acciones, representando una toma de decisión múltiple. Podemos exemplificar la sintaxis de la siguiente forma:

```

SI <condición 1> ENTONCES
    Acción 1
SI NO
    SI <condición 2> ENTONCES
        Acción 2
    SI NO
        Acción 3
    FIN SI
FIN SI

```

En la estructura anterior, hay una primera evaluación lógica en la cual si el resultado es **VERDADERO**, se ejecuta la Acción 1 y nada más. En cambio, si su resultado es **FALSO**, se procede a realizar una segunda evaluación lógica, que da lugar a la ejecución de la Acción 2 o de la Acción 3 si su resultado es **VERDADERO** o **FALSO**, respectivamente.

Se debe notar que luego del primer **SI NO** comienza una nueva estructura completa de **SI/ENTONCES/SI NO/FIN SI**. Cada **SI** termina con su propio **FIN SI**. Al traducir esto a R, se vuelve algo más sencillo:

```

if (<condición 1>) {
    ...Acción 1...
} else if (<condición 2>) {
    ...Acción 2...
} else {
    ...Acción 3...
}

```

El último bloque de acciones (**...Acción 3...**) se evaluará si ninguna de las condiciones lógicas anteriores fue **VERDADERO**.

En el ejemplo de la edad:

```

ALGORITMO: "Analizar edad para votar"
COMENZAR
    VARIABLE numérica edad
    LEER edad
    SI edad < 18 ENTONCES
        ESCRIBIR "Edad = " edad " años: no puede votar"
    SI NO
        SI edad >= 70 ENTONCES
            ESCRIBIR "Edad = " edad " años: puede votar opcionalmente"
        SI NO
            ESCRIBIR "Edad = " edad " años: debe votar obligatoriamente"
        FIN SI
    FIN SI
FIN

# Programa: "Analizar edad para votar" -----
edad <- 21
if (edad < 18) {
    cat("Edad =", edad, "años: no puede votar")
}

```

```

} else if (edad >= 70) {
  cat("Edad =", edad, "años: puede votar opcionalmente")
} else {
  cat("Edad =", edad, "años: debe votar obligatoriamente")
}

```

Edad = 21 años: debe votar obligatoriamente

### 3.3. Estructuras de control iterativas

Las estructuras de control iterativas son útiles cuando la solución de un problema requiere que se ejecute repetidamente un determinado conjunto de acciones. El número de veces que se debe repetir dicha secuencia de acciones puede ser fijo o puede variar dependiendo de algún dato o condición a evaluar en el algoritmo.

#### 3.3.1. Estructuras de control iterativas con un número fijo de iteraciones

Se aplican cuando se conoce de antemano el número exacto de veces que se debe repetir una secuencia de acciones. También se conocen como *bucles (loops) controlados por un conteo*, ya que el algoritmo va contando la cantidad de repeticiones haciendo uso de una variable que recibe el nombre de **variable de iteración, índice o conteo**.

Por ejemplo, imaginemos que queremos escribir un algoritmo que permita calcular la quinta potencia de cualquier número. Para esto, se debe tomar dicho número y multiplicarlo por sí mismo 5 veces. Por lo tanto, una posible solución es:

ALGORITMO: "Calcular la quinta potencia"

COMENZAR

```

  VARIABLE numérica x, resultado
  LEER x
  resultado <- 1
  resultado <- resultado * x
  ESCRIBIR x "elevado a la quinta es igual a" resultado
FIN

```

Ya que sabemos que la multiplicación se debe repetir 5 veces, podemos resumir lo anterior con la siguiente estructura:

ALGORITMO: "Calcular la quinta potencia"

COMENZAR

```

  VARIABLE numérica x, resultado
  LEER x
  resultado <- 1
  PARA i DESDE 1 HASTA 5 HACER
    resultado <- resultado * x
  FIN PARA
  ESCRIBIR x "elevado a la quinta es igual a" resultado
FIN

```

La letra *i* es la variable de iteración. Podríamos haber elegido otra letra u otra palabra en su lugar, pero emplear *i* es una elección bastante común. En este ejemplo, su única función es ir contando la cantidad de veces que se repiten las acciones encerradas dentro de la estructura PARA/FIN PARA. El bloque de instrucciones se repite tantas veces como *i* tarde en llegar a 5 partiendo desde 1. Por convención, a la variable de iteración no la declaramos junto con las otras variables numéricas (como *x* y *resultado*).

En R, el ejemplo anterior se implementa así:

```
# Programa: "Calcular la quinta potencia" -----
x <- 4
```

```

resultado <- 1
for (i in 1:5) {
  resultado <- resultado * x
}
cat(x, "elevado a la quinta es igual a", resultado)

```

4 elevado a la quinta es igual a 1024

Dado que la variable de iteración toma un valor numérico que va cambiando en cada repetición del bloque, se puede aprovechar para hacer cuentas con el mismo. Por ejemplo, el siguiente algoritmo muestra la tabla del ocho:

ALGORITMO: "Mostrar tabla del 8"

COMENZAR

```

  VARIABLE numérica resultado
  PARA i DESDE 0 HASTA 10 HACER
    resultado <- 8 * i
    ESCRIBIR "8 x" i "=" resultado
  FIN PARA
FIN
```

# Programa: "Mostrar tabla del 8" -----

```

for (i in 0:10) {
  resultado <- 8 * i
  cat("8 x", i, "=", resultado, "\n")
}
```

```

8 x 0 = 0
8 x 1 = 8
8 x 2 = 16
8 x 3 = 24
8 x 4 = 32
8 x 5 = 40
8 x 6 = 48
8 x 7 = 56
8 x 8 = 64
8 x 9 = 72
8 x 10 = 80

```

En lo anterior, \n es un carácter especial que indica “salto de línea”. Si no lo agregamos, los mensajes se imprimirían uno al lado del otro en el mismo renglón:

# Programa: "Mostrar tabla del 8" -----

```

for (i in 0:10) {
  resultado <- 8 * i
  cat("8 x", i, "=", resultado)
}
```

```

8 x 0 = 08 x 1 = 88 x 2 = 168 x 3 = 248 x 4 = 328 x 5 = 408 x 6 = 488 x 7 = 568 x 8 = 648 x 9 = 728 x 10 =

```

De manera general, la sintaxis para este tipo de estructuras es:

```

PARA <variable> DESDE <valor1> HASTA <valor2> CON PASO <valor3> HACER
  Acción/es
FIN PARA
```

Dado un valor inicial <valor1> asignado a la <variable>, esta se irá aumentando o disminuyendo según el paso <valor3> hasta llegar a tomar el valor <valor2>. Si no se indica el paso se asume que la variable de iteración aumenta de uno en uno. En R:

```

for (<variable> in seq(<valor1>, <valor2>, <valor3>)) {
  ...Acción/es...
}
```

```
}
```

Notar en el ejemplo de la quinta potencia que `1:5` es lo mismo que `seq(1, 5, 1)`, pero podemos usar la función `seq()` en otros contextos más complejos, donde la variable de iteración puede pegar otros saltos en lugar de uno en uno.

### 3.3.2. Estructuras de control iterativas con un número indeterminado de iteraciones

En otras circunstancias se puede necesitar repetir un bloque de acciones sin conocer con exactitud cuántas veces, si no que esto depende de algún otro aspecto del algoritmo. Las iteraciones pueden continuar *mientras que* o *hasta que* se verifique alguna condición, dando lugar a dos tipos de estructuras. Estos casos también se conocen como *bucles (loops) controlados por una condición*.

#### 3.3.2.1. Mientras que

El conjunto de sentencias se repite mientras que se siga evaluando como **VERDADERO** una condición declarada al inicio del bloque. Cuando la condición ya no se cumple, el proceso deja de ejecutarse. La sintaxis es:

```
MIENTRAS QUE <condición> HACER
    Acción/es a repetir
FIN MIENTRAS
```

En R:

```
while (<condición>) {
  ...Acción/es a repetir...
}
```

Observaciones:

- La evaluación de la condición se lleva a cabo antes de cada iteración, incluso antes de ejecutar el código dentro del bloque por primera vez. Si la condición es **FALSO** inicialmente, entonces las acciones en el cuerpo de la estructura no se ejecutan nunca.
- La evaluación de la condición **sólo** se lleva a cabo al inicio de cada iteración. Si la condición se vuelve **FALSO** en algún punto durante la ejecución de un bloque, el programa no lo nota hasta que se termine de ejecutar el bloque y la condición sea evaluada antes de comenzar la próxima iteración.

Veamos un ejemplo:

ALGORITMO: "Dividir un número por 2 hasta encontrar un valor menor que 0.01"  
COMENZAR

```
VARIABLE numérica x
LEER x
MIENTRAS QUE x >= 0.01 HACER
  x <- x / 2
  ESCRIBIR x
FIN MIENTRAS
FIN
```

```
x <- 100
while (x >= 0.01) {
  x <- x / 2
  cat(x, "\n")
}
```

```
50
25
12.5
6.25
3.125
1.5625
0.78125
```

```
0.390625
0.1953125
0.09765625
0.04882812
0.02441406
0.01220703
0.006103516
```

### 3.3.2.2. Hasta que

A diferencia de la estructura *MIENTRAS QUE*, la estructura *HASTA QUE* repite el bloque de acciones hasta que se cumpla una condición, es decir, se ejecuta mientras que dicha condición sea evaluada como **FALSA**. La sintaxis es:

```
REPETIR
    Acción/es
HASTA QUE <condición>
```

Observación: con la estructura *MIENTRAS QUE* podría ser que el conjunto de sentencias nunca llegue a ejecutarse si desde partida la condición evaluada ya es falsa. Por el contrario, en la estructura *HASTA QUE* el proceso se realiza al menos una vez, dado que la condición se evalúa al final.

El ejemplo anterior empleando este tipo de estructura:

ALGORITMO: "Dividir un número por 2 hasta encontrar un valor menor que 0.01"  
COMENZAR

```
VARIABLE numérica x
LEER x
REPETIR
    x <- x / 2
    ESCRIBIR x
HASTA QUE x < 0.01
FIN
```

En R este tipo de estructura se implementa con la sentencia `repeat {}`. Si bien a continuación se muestra el correspondiente ejemplo, no vamos a utilizar esta estructura, debido a que su escritura es más compleja y a que generalmente es posible obtener el mismo resultado con un `while () {}`.

```
x <- 100
repeat {
    x <- x / 2
    cat(x, "\n")
    if (x < 0.01) break
}
```

```
50
25
12.5
6.25
3.125
1.5625
0.78125
0.390625
0.1953125
0.09765625
0.04882812
0.02441406
0.01220703
0.006103516
```

### 3.3.2.3. Loops infinitos

Con las sentencias de tipo **MIENTRAS QUE** se debe tener mucha precaución, puesto que si la evaluación lógica no está bien especificada o nunca deja de ser evaluada como TRUE, se incurre en un *loop* infinito: el programa nunca deja de repetir el bloque (al menos hasta que la máquina se tilde o se produzca un error por desbordamiento de memoria, por ejemplo).

La siguiente situación ilustra esto:

```
var <- 9
while (var < 10) {
  var <- var - 1
  cat("var =", var, "No puedo parar!!!\n")
}

var = 8 No puedo parar!!!
var = 7 No puedo parar!!!
var = 6 No puedo parar!!!
var = 5 No puedo parar!!!
var = 4 No puedo parar!!!
var = 3 No puedo parar!!!
var = 2 No puedo parar!!!
var = 1 No puedo parar!!!
var = 0 No puedo parar!!!
var = -1 No puedo parar!!!
...

```

En R se puede usar la instrucción **break** para forzar la detención del proceso iterativo si se presenta alguna condición en particular<sup>1</sup>:

```
var <- 9
while (var < 10) {
  var <- var - 1
  cat("var =", var, "No puedo parar!!!\n")
  if (var == -3) break
}

var = 8 No puedo parar!!!
var = 7 No puedo parar!!!
var = 6 No puedo parar!!!
var = 5 No puedo parar!!!
var = 4 No puedo parar!!!
var = 3 No puedo parar!!!
var = 2 No puedo parar!!!
var = 1 No puedo parar!!!
var = 0 No puedo parar!!!
var = -1 No puedo parar!!!
var = -2 No puedo parar!!!
var = -3 No puedo parar!!!
```

## 3.4. Ejemplos

A continuación se presentan algunos otros ejemplos

- No necesariamente tiene que ser **i** la variable iteradora, podemos darle cualquier nombre:

```
for (guau in 1:5) {
  print(guau)
}
```

<sup>1</sup>También podemos usar **break** dentro de una estructura **for**

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

La sentencia `print()` también sirve para mostrar resultados. La ventaja con respecto a `cat()` es que no necesitamos agregar `\n` para que el siguiente mensaje se escriba en un nuevo renglón, ya que lo agrega por sí sola sin que lo pidamos. La desventaja es que no nos permite crear un mensaje combinando elementos separados entre comas, tal como se puede hacer con `cat()` (por ejemplo, `cat(.el valor de x es igual a", x)`).

- Acá tenemos un ejemplo de dos estructuras `for` anidadas. En primer lugar, `i` toma el valor 1, y entonces `j` varía de 1 a 2, generando las combinaciones `i = 1, j = 1`; `i = 1, j = 2`. Luego de que el *loop* de `j` finalice habiendo recorrido todo su campo de variación, comienza la segunda iteración del *loop* de `i`, actualizándose su valor a 2 y comenzando otra vez el *loop* de `j`, que varía de 1 a 2. Así, se generan las combinaciones `i = 2, j = 1`; `i = 2, j = 2`. Finalmente, se actualiza `i` y pasa a valer 3, generando las combinaciones `i = 3, j = 1`; `i = 3, j = 2`. Para cada combinación, se muestra el valor de la suma:

```
for (i in 1:3) {
  for (j in 1:2) {
    suma <- i + j
    cat("i vale", i, "y j vale", j, ". La suma es igual a", suma, "\n")
  }
}

i vale 1 y j vale 1 . La suma es igual a 2
i vale 1 y j vale 2 . La suma es igual a 3
i vale 2 y j vale 1 . La suma es igual a 3
i vale 2 y j vale 2 . La suma es igual a 4
i vale 3 y j vale 1 . La suma es igual a 4
i vale 3 y j vale 2 . La suma es igual a 5
```

- Sumar los números naturales del 1 al 5:

```
suma <- 0
for (i in 1:5) {
  suma <- suma + i
}
suma
```

```
[1] 15
```

- Sumar números naturales hasta que la suma pase el valor 100 y detenerse:

```
suma <- 0
i <- 1
while (suma < 100) {
  suma <- suma + i
  i <- i + 1
}
suma
```

```
[1] 105
```

- Escribir todos los múltiplos de 8 menores que 150:

ALGORITMO: "Múltiplos de 8 menores a 150"  
 COMENZAR  
   VARIABLE numérica multiplo  
   multiplo <- 8  
   MIENTRAS QUE multiplo < 150 HACER

```
ESCRIBIR multiplo
multiplo <- multiplo + 8
FIN MIENTRAS
FIN

# Programa: "Múltiplos de 8 menores a 150" -----
multiplo <- 8
while (multiplo < 150) {
    print(multiplo)
    multiplo <- multiplo + 8
}

[1] 8
[1] 16
[1] 24
[1] 32
[1] 40
[1] 48
[1] 56
[1] 64
[1] 72
[1] 80
[1] 88
[1] 96
[1] 104
[1] 112
[1] 120
[1] 128
[1] 136
[1] 144
```

## Unidad 4

# Descomposición algorítmica

Un principio fundamental en la resolución de un problema es intentar descomponerlo en partes más pequeñas, que puedan ser más fáciles de afrontar. Este concepto también se aplica en la programación. Nuestros algoritmos pueden descomponerse en **subalgoritmos** que den solución a un aspecto del problema, de menor extensión. Este proceso se conoce como **descomposición algorítmica** o **descomposición modular**. Cada subalgoritmo debe ser independiente de los demás y a su vez podría seguir descomponiéndose en partes más sencillas en lo que se conoce como **refinamiento sucesivo**. Si un programa es muy largo se corre el riesgo de que sea muy difícil de entender como un todo, pero siempre se lo puede dividir en secciones más simples y manejables. Un subalgoritmo se escribe una vez y luego es utilizado por todos aquellos algoritmos que lo necesiten.

Cada vez que, como parte de la resolución de un problema, desde un algoritmo se solicita que se realicen las acciones establecidas por un subalgoritmo, se dice que se lo está **invocando** o **llamando**. Al algoritmo que invoca a los subalgoritmos a veces le decimos *algoritmo principal* para darle un mayor énfasis a la idea que, desde el curso de acción principal, cada tanto se delega la ejecución de algunas tareas al subalgoritmo.

El empleo de subalgoritmos, desarrollando por separado ciertas partes del problema, resulta especialmente ventajoso en los casos siguientes:

- **En algoritmos complejos:** si el algoritmo, y luego el programa, se escribe todo seguido y en un único archivo de código, resulta muy complicado de entender, porque se pierde la visión de su estructura global dada la gran cantidad de operaciones que lo conforman. Aislando ciertas partes como subalgoritmos separados se reduce la complejidad.
- **Cuando se repiten operaciones análogas:** si la resolución de un problema requiere realizar una tarea que se repite varias veces en el algoritmo, podemos definir dicha tarea como un subalgoritmo por separado. De esta manera, su código se escribirá sólo una vez aunque se use en muchos puntos del programa.

En este capítulo hay algunas secciones indicadas como “opcionales” y algunos comentarios agregados como notas al pie. Estas partes añaden información para quienes estén interesados en saber un poco más, pero su contenido no será requerido en la práctica ni en las evaluaciones.

### 4.1. Tipos de subalgoritmos

En el mundo de la programación existen muchos términos para definir distintos tipos de subalgoritmos: subrutinas, funciones, procedimientos, métodos, subprogramas, etc. No es posible obtener una definición que capture todas las variantes que existen en el uso de estos términos debido a que el significado de cada uno de ellos varía según el paradigma<sup>1</sup> y el lenguaje de programación escogidos.

Sin embargo, suele haber bastante consenso en distinguir, dentro de los subalgoritmos, a las **funciones** y a los **procedimientos** de esta forma:

- Una **función** es un subalgoritmo que al ser evaluado devuelve un único resultado (por ejemplo, un valor numérico) que es utilizado en el algoritmo principal que lo invoca.
- Un **procedimiento** es un subalgoritmo que al ser evaluado no devuelve un valor, sino que produce *efectos secundarios* en el ambiente del algoritmo principal que lo invoca. Persigue el objetivo de ayudar en la

---

<sup>1</sup>Se usa el término paradigma de programación para clasificar a los lenguajes según sus características. En este link se puede encontrar una breve descripción de los principales paradigmas de programación

modularidad del programa.

## 4.2. Funciones

Una **función** es un subalgoritmo que devuelve un único resultado a partir de otros valores provistos. El valor que la función devuelve define su **tipo**, de modo que una función puede ser de tipo **numérica**, **carácter** o **lógica**<sup>2</sup>.

Para exemplificar, podemos decir que la noción de *función* en programación se asemeja a la idea matemática de *función de una o más variables*. Podemos pensar en la función  $f(x, y) = x^2 + 3y$  (ejemplo 1). Si queremos saber cuál es el valor numérico de la función  $f$  cuando  $x$  toma el valor 4 e  $y$  toma el valor 5, reemplazamos en la expresión anterior las variables por los valores mencionados y obtenemos:  $f(4, 5) = 4^2 + 3 \times 5 = 31$ .

Podemos definir dicha función en pseudocódigo de la siguiente manera:

```
FUNCIÓN f(x: numérico, y: numérico): numérico
COMENZAR
    DEVOLVER x^2 + 3 * y
FIN FUNCIÓN
```

El primer renglón de la definición comienza con la palabra clave **FUNCIÓN** y termina, luego de los dos puntos, con la palabra **numérico** para indicar que esta función devuelve como resultado un valor numérico.

En el medio se encuentra el nombre elegido para la función (**f**), seguido por la declaración entre paréntesis de los **parámetros** o **argumentos** que la función necesita para operar, es decir, el *input* o información de entrada con la cual se realizarán las operaciones. Se dice que  $x$  e  $y$  son los **parámetros formales** o **ficticios**, ya que no tienen un valor asignado en sí mismos sino que permiten expresar de manera general las acciones que la función ejecuta. Describen lo que uno diría en palabras: “hay que tomar a  $x$ , elevarlo al cuadrado y sumarle la  $y$  multiplicada por 3”. Entre los paréntesis también se aclara que estos parámetros formales son de tipo numérico.

Los valores en los cuales se quiere evaluar la función se llaman **parámetros actuales** o **reales**. Por ejemplo, si nos interesa calcular  $f(4, 5)$ , los valores 4 y 5 son los parámetros actuales y se establece una correspondencia entre el parámetro formal  $x$  y el actual 4, así como entre la  $y$  y el 5. El resultado que se obtiene, como observamos antes, es 31 y este es el valor que la función *devuelve*.

La definición anterior también puede ser expresada como:

```
FUNCIÓN f(x: numérico, y: numérico): numérico
COMENZAR
    VARIABLE numérica resultado
    resultado <- x^2 + 3 * y
    DEVOLVER resultado
FIN FUNCIÓN
```

Aquí notamos que no debemos declarar a  $x$  e  $y$  puesto que son los parámetros de la función (quedan declarados entre los paréntesis en la primera línea). Sin embargo, sí declaramos cualquier otra nueva variable que sea creada dentro de la función, por ejemplo, la variable **resultado**.

De manera general, la definición de una función es:

```
FUNCIÓN nombre(lista de parámetros formales): tipo de resultado
COMENZAR
    Declaración de variables
    Acciones
    DEVOLVER valor
FIN FUNCIÓN
```

La palabra clave **DEVOLVER** provoca la inmediata finalización de la ejecución de la función e indica cuál es el resultado de la misma, cuyo tipo debe coincidir con el tipo de función declarado en el encabezado. La acción **DEVOLVER** se puede insertar en cualquier punto del cuerpo de la función y, además, es posible utilizar más de una sentencia **DEVOLVER** en una misma función, aunque sólo una llegue a ejecutarse. Esto puede verse en el siguiente ejemplo (ejemplo 2):

---

<sup>2</sup>No siempre se pueden clasificar de esta forma a las funciones, ya que hay algunas que pueden devolver distintos tipos de valores según el caso o que devuelven otras clases de objetos.

```

FUNCIÓN maximo(num1: numérico, num2: numérico): numérico
COMENZAR
    SI num1 >= num2 ENTONCES
        DEVOLVER num1
    SI NO
        DEVOLVER num2
    FIN SI
FIN FUNCIÓN

```

Para usar una función en un algoritmo, se la invoca escribiendo su nombre seguido por los valores actuales entre paréntesis, separados por coma. Esta invocación representa un valor que puede ser usado como operando en otra expresión. Por ejemplo:

```

ALGORITMO: "Hallar el máximo entre dos valores y restarle 100"
COMENZAR
    VARIABLE numérica x, y, rtdo
    LEER x, y
    rtdo <- maximo(x, y) - 100
    ESCRIBIR "El resultado es " rtdo
FIN

```

Al invocar una función es obligatorio que los valores suministrados para los argumentos actuales entre los paréntesis correspondan en cantidad, tipo y orden con los argumentos formales de la definición de la función. Es por esto que los siguientes casos son ejemplos de un **uso incorrecto** de funciones en el algoritmo principal:

```

ALGORITMO: "Incorrecto por proveer pocos argumentos para la función"
COMENZAR
    VARIABLE numérica x, y, rtdo
    LEER x, y
    rtdo <- maximo(x) - 100
    ESCRIBIR "El resultado es " rtdo
FIN

```

```

ALGORITMO: "Incorrecto por proveer valores de tipo carácter para la función"
COMENZAR
    VARIABLE numérica rtdo
    VARIABLE carácter x, y
    x <- "chau"
    y <- "holá"
    rtdo <- maximo(x, y) - 100
    ESCRIBIR "El resultado es " rtdo
FIN

```

```

ALGORITMO: "Incorrecto por no proveer argumentos para la función"
COMENZAR
    VARIABLE numérica x, y, rtdo
    LEER x, y
    rtdo <- maximo - 100
    ESCRIBIR "El resultado es " rtdo
FIN

```

Para motivar el uso de una buena práctica que ayude a distinguir entre las acciones de los subalgoritmos y del algoritmo, vamos a escribir los subalgoritmos **antes y por fuera** del algoritmo principal<sup>3</sup>. Consideremos el ejemplo 3 mostrado a continuación. Primero definimos los subalgoritmos que necesitaremos (son los de los ejemplos 1 y 2) y luego escribiremos un algoritmo principal que hace uso de ellos para resolver un problema en particular:

---

#### SUBALGORITMOS

<sup>3</sup>Es más, podríamos escribirlos y guardarlos en archivos distintos.

---

```

FUNCIÓN f(x: numérico, y: numérico): numérico
COMENZAR
    DEVOLVER x^2 + 3 * y
FIN FUNCIÓN

FUNCIÓN maximo(num1: numérico, num2: numérico): numérico
COMENZAR
    SI num1 >= num2 ENTONCES
        DEVOLVER num1
    SI NO
        DEVOLVER num2
    FIN SI
FIN FUNCIÓN

```

---

## ALGORITMO PRINCIPAL

---

```

ALGORITMO: "Realizar operaciones matemáticas muy importantes"
COMENZAR
    VARIABLE numérica rtdo1, rtdo2, rtdo3
    rtdo1 <- f(2, 5)
    rtdo2 <- f(3, 10)
    rtdo3 <- maximo(rtdo1, rtdo2) + 20
    ESCRIBIR "El resultado es " rtdo3
FIN

```

¿Qué mensaje escribe el algoritmo anterior?

## 4.3. Funciones en R

En la sección anterior vimos cómo definir funciones en pseudocódigo. Antes de pasar a ver cómo programar nuestras funciones en R, vamos a comentar algunas cuestiones acerca de las funciones que R ya trae disponibles como parte de su funcionalidad básica.

### 4.3.1. Funciones predefinidas de R

R, como todo lenguaje de programación, tiene **funciones predefinidas**, es decir, sentencias que se encargan de realizar alguna actividad. Ya estuvimos usando algunas de ellas, por ejemplo, cuando hemos necesitado mostrar algún mensaje usamos las funciones `cat()` o `print()`<sup>4</sup>. Además, existen muchas otras funciones predefinidas, como todas aquellas que se necesitan para realizar ciertas operaciones matemáticas:

```
# Raíz cuadrada
sqrt(100)
```

```
[1] 10
```

```
# Valor absoluto
abs(100)
```

```
[1] 100
```

```
# Función exponencial
exp(100)
```

<sup>4</sup>Sentencias como `for`, `while` o `if` también son funciones, aunque con una estructura muy particular.

```
[1] 2.688117e+43
```

```
# Logaritmo natural
log(100)
```

```
[1] 4.60517
```

En los ejemplos anteriores, 100 representa un valor numérico que se pasa como argumento a la función para que la misma opere. Algunas funciones predefinidas en R pueden trabajar con más de un argumento, en cuyo caso hay que enumerarlos dentro de los paréntesis, separados con comas. Por ejemplo, si en lugar de calcular el logaritmo natural (cuya base es la constante matemática  $e$ ), queremos calcular un logaritmo en base 10, podemos hacer lo siguiente:

```
# Logaritmo de 100 en base 10
log(100, 10)
```

```
[1] 2
```

¿Cómo sabemos que la función `log()` se puede usar de esa forma, cambiando el valor de la base con respecto a la cual toma el logaritmo? Lo aprendemos al leer el manual de ayuda de R. Toda función predefinida de R viene con un instructivo que detalla cómo se usa, qué argumentos incluye y otras aclaraciones. Lo encontramos en la pestaña de Ayuda (*Help*) en el panel de abajo a la derecha en RStudio. Otras formas de abrir la página de ayuda sobre una función es correr en la consola alguna de estas sentencias:

```
help(log)
?log
```

Esa página de ayuda tiene bastante información, porque reúne información sobre muchas funciones relacionadas con logaritmos y exponentiales, pero podemos detenernos en la parte resaltada que se muestra a continuación:

### Usage

```
log(x, base = exp(1))
logb(x, base = exp(1))
log10(x)
log2(x)

log1p(x)

exp(x)
expm1(x)
```

### Arguments

**x** a numeric or complex vector.

**base** a positive or complex number: the base with respect to which logarithms are computed. Defaults to `e=exp(1)`.

Figura 4.1: Captura de pantalla de la ayuda sobre la función `log()`

En la sección *Usage* (“uso”) descubrimos que la función `log()` puede usarse con dos argumentos: `x` y `base`. En la sección *Arguments* entendemos que `x` es el número al cual le vamos a sacar el logaritmo y `base` es la base con respecto a la cual se toma el logaritmo. Por eso, al correr `log(100, 10)`, estamos calculando el logaritmo de `x = 100` con `base = 10`.

Vemos, además, una diferencia en la forma en que `x` y `base` aparecen en la descripción: `log(x, base = exp(1))`. Cuando un argumento tiene un signo `=` significa que tiene asignado un **valor por default** y que no es necesario usarlo. Por eso, cuando corremos `log(100)` estamos calculando el logaritmo de `x = 100` con la base elegida por R por defecto: `base = exp(1)`, que es la forma que tiene R de nombrar a la constante  $e = 2.718282\dots$  (es el logaritmo natural). Si quiero cambiar la base, debo proveer un valor, por ejemplo, `log(100, 10)`. Por el contrario, el argumento `x` no tiene asignado un valor por default. Eso significa que obligatoriamente tenemos que proveer un valor para el mismo.

R también permite usar una función escribiendo los nombres de los argumentos (es decir, detallando tanto los

parámetros formales como los actuales), lo cual muchas veces es muy esclarecedor:

```
log(x = 100, base = 10)
```

```
[1] 2
```

Es más, si escribimos los nombres de los parámetros explícitamente, podemos cambiar su orden, sin alterar el resultado:

```
log(base = 10, x = 100)
```

```
[1] 2
```

Si no escribimos los nombres, el orden importa:

```
log(100, 10)
```

```
[1] 2
```

```
log(10, 100)
```

```
[1] 0.5
```

Al no tener los nombres indicados explícitamente, R hace corresponder los parámetros formales `x` y `base` con los valores provistos en ese orden: en el primer caso `x` recibe el valor 100 y `base`, el valor 10, mientras que en el segundo caso es al revés.

Finalmente, se debe observar que no es necesario invocar a la función escribiendo de forma directa los valores entre los paréntesis, sino que en su lugar pueden ir variables:

```
x <- 100
y <- x / 2
z <- 4
log(x - y, 4)      # Log en base 4 de x - y
```

```
[1] 2.821928
```

### 4.3.2. Definición de nuevas funciones en R

Ahora que ya hemos visto cómo se trabaja con funciones en R de manera general, vamos a aprender a definir nuestras propias funciones. Recordemos el subalgoritmo del ejemplo 1:

```
FUNCIÓN f(x: numérico, y: numérico): numérico
COMENZAR
    VARIABLE numérica resultado
    resultado <- x^2 + 3 * y
    DEVOLVER resultado
FIN FUNCIÓN
```

En R, definimos esta función así:

```
f <- function(x, y) {
  resultado <- x^2 + 3 * y
  return(resultado)
}
```

La estructura general es:

```
nombre <- function(argumentos) {
  ... sentencias de R ...
}
```

Debemos:

1. Elegir un nombre

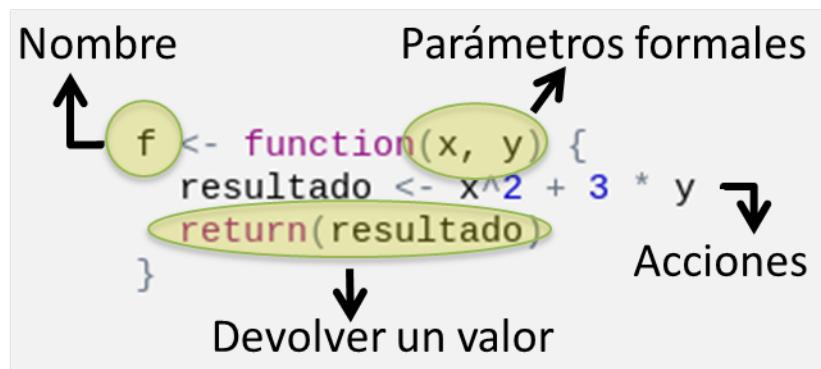


Figura 4.2: Estructura de una función en R

2. Al lado del nombre, colocar el operador de asignación (`<-`) para asociar a ese nombre la definición de una función.
3. Escribir la sentencia `function(...){...}`, donde entre paréntesis se definen todos los parámetros formales separados con coma y entre llaves el conjunto de acciones a englobar.
4. El valor que la función debe arrojar como resultado se encierra dentro de la sentencia `return()`, que indica el fin de la ejecución.

Una vez que la definición de la función es ejecutada, pasa a formar parte de los elementos que conforman al ambiente, como se puede apreciar al verla listada como un objeto más en el panel *Environment* de RStudio<sup>5</sup>. A partir de este momento, podemos utilizarla, como parte de otro programa. Para invocarla, escribimos el nombre de la función y entre paréntesis los valores que nos interesan para el cálculo (parámetros actuales). Por ejemplo:

```
# Ejemplos de uso de la función f
f(4, 5)
```

```
[1] 31
```

```
f(6, -5)
```

```
[1] 21
```

```
f(0, 0)
```

```
[1] 0
```

Recordando lo discutido en la sección anterior, podemos apreciar que los siguientes usos de la función `f()` son equivalentes:

```
f(4, 5)
```

```
[1] 31
```

```
f(x = 4, y = 5)
```

```
[1] 31
```

```
f(y = 5, x = 4)
```

```
[1] 31
```

Sin embargo, no son equivalentes los siguientes:

<sup>5</sup>Sí, las funciones que creamos también son *objetos* para R, ya que son una pieza que guarda algún tipo de información. Las funciones en R son objetos de clase “function”. Ver, por ejemplo, `class(f)`

```
# Siguiendo el orden de definición, x recibe el valor 4, y recibe el 5:  
f(4, 5)
```

[1] 31

```
# Siguiendo el orden de definición, x recibe el valor 5, y recibe el 4:  
f(5, 4)
```

[1] 37

A continuación, podemos ver casos que generan error por hacer un uso incorrecto de la función (¿por qué?):

```
# Error por omitir un argumento de uso obligatorio (x recibe 4, falta y)  
f(4)
```

Error in f(4): argument "y" is missing, with no default

```
# Error por proveer más argumentos de los declarados en la definición  
f(4, 5, 6)
```

Error in f(4, 5, 6): unused argument (6)

Retomemos ahora el ejemplo 3. Mencionamos que es importante distinguir entre la definición de los subalgoritmos y la de un algoritmo principal que los invoca:

#### SUBALGORITMOS

FUNCIÓN f(x: numérico, y: numérico): numérico

COMENZAR

    DEVOLVER  $x^2 + 3 * y$

FIN FUNCIÓN

FUNCIÓN maximo(num1: numérico, num2: numérico): numérico

COMENZAR

    SI num1  $\geq$  num2 ENTONCES

        DEVOLVER num1

    SI NO

        DEVOLVER num2

    FIN SI

FIN FUNCIÓN

#### ALGORITMO PRINCIPAL

ALGORITMO: "Realizar operaciones matemáticas muy importantes"

COMENZAR

    VARIABLE numérica rtdo1, rtdo2, rtdo3

    rtdo1 <- f(2, 5)

    rtdo2 <- f(3, 10)

    rtdo3 <- maximo(rtdo1, rtdo2) + 20

    ESCRIBIR "El resultado es " rtdo3

FIN

Esta distinción también es importante en R: la definición de las funciones debe ejecutarse antes de que las mismas sean llamadas desde el programa principal. Así, para traducir el pseudocódigo anterior a R, podríamos crear un archivo de código (llamado, por ejemplo, `ejemplo3.R`) con el siguiente contenido:

```

# -----
# DEFINICIÓN DE FUNCIONES
# -----


f <- function(x, y) {
  resultado <- x^2 + 3 * y
  return(resultado)
}

maximo <- function(num1, num2) {
  if (num1 > num2) {
    return(num1)
  } else {
    return(num2)
  }
}

# -----
# PROGRAMA PRINCIPAL
# -----


rtdo1 <- f(2, 5)
rtdo2 <- f(3, 10)
rtdo3 <- maximo(rtdo1, rtdo2) + 20
cat("El resultado es", rtdo3)

```

El resultado es 59

#### 4.3.3. NULL vs NA vs NaN

Generalmente los lenguajes de programación poseen un valor conocido como NULO, para representar un objeto vacío, sin información. El mismo suele emplearse como valor devuelto por funciones cuando no corresponde devolver otro tipo de resultado. En pseudocódigo podemos usar esta estrategia escribiendo DEVOLVER NULO, si deseamos que nuestra función no devuelva nada. La representación en R de este tipo de objeto es NULL, que se trata de un objeto vacío que generalmente devuelven las funciones cuando el resultado es indefinido. Es decir, podemos crear funciones que terminen con un `return(NULL)`, como en el siguiente caso donde interesa emitir un mensaje, pero no devolver ningún objeto:

```

g <- function(x, y) {
  resultado <- x^2 + 3 * y
  cat("El resultado de esta cuenta es:", resultado)
  return(NULL)
}
g(4, 5)

```

El resultado de esta cuenta es: 31

NULL

El objeto NULL no debe confundirse con otros dos valores existentes en el lenguaje R: NA y NaN:

- NA son las siglas de *Not Available* y es un valor lógico (como TRUE y FALSE) que generalmente representa datos faltantes.
- NaN son las siglas de *Not a Number* y es un valor numérico que generalmente surge como resultado de operaciones aritméticas imposibles de calcular, como indeterminaciones, raíces negativas, etc. (correr `0/0`, `log(-1)` o `sqrt(-1)` para verlo). Es un valor establecido por IEEE, el estándar con el cual se rige la representación numérica en la computadora.

Otro valor numérico muy especial es Inf (y su contrapartida negativa, -Inf), que es el resultado de almacenar un número muy grande o de una división por cero. No es semejante a NA, porque además de no ser de tipo lógico, no

representa que hay un dato faltante sino que se trata de un valor numérico.

```
a <- NULL
b <- NA
d <- NaN
e <- Inf
f <- "NULL"

class(a)

[1] "NULL"

class(b)

[1] "logical"

class(d)

[1] "numeric"

class(e)

[1] "numeric"

class(f)

[1] "character"
```

#### 4.3.4. Función `return()`

En R, la función `return()` puede omitirse, ya que si no está presente se devuelve el resultado de la última expresión analizada. Por eso, las siguientes funciones son equivalentes:

```
g1 <- function(x, y) {
  resultado <- x^2 + 3 * y
  return(resultado)
}
g1(4, 5)
```

```
[1] 31

g2 <- function(x, y) {
  x^2 + 3 * y
}
g2(4, 5)
```

```
[1] 31
```

De todos modos, es aconsejable usar `return()` para evitar ambigüedades y ganar en claridad. Además, en funciones más complejas, su uso puede ser indispensable para indicar el término de la evaluación de la función.

En el caso particular donde interese que nuestra función emita un mensaje, sin necesariamente devolver un objeto en particular, podemos proceder como se mencionó en la sección anterior:

```
g3 <- function(x, y) {
  resultado <- x^2 + 3 * y
  cat("El resultado de esta cuenta es:", resultado)
  return(NULL)
}
g3(4, 5)
```

```
El resultado de esta cuenta es: 31
```

NULL

O de esta otra forma:

```
g4 <- function(x, y) {
  resultado <- x^2 + 3 * y
  cat("El resultado de esta cuenta es:", resultado)
}
g4(4, 5)
```

El resultado de esta cuenta es: 31

En ambos casos, la función escribe el mensaje y devuelve como resultado un objeto NULL: en g3 porque se lo pedimos explícitamente y en g4 porque la función `cat()`, que es lo último en evaluarse, además de escribir un mensaje, devuelve un NULL:

```
x <- g3(4, 5)
```

El resultado de esta cuenta es: 31

x

NULL

```
y <- g4(4, 5)
```

El resultado de esta cuenta es: 31

y

NULL

## 4.4. Documentación de los subalgoritmos

En el contexto de la programación, documentar significa escribir indicaciones para que otras personas puedan entender lo que queremos hacer en nuestro código o para que sepan cómo usar nuestras funciones. Por ejemplo, como vimos antes todas funciones predefinidas de R están documentadas para que podamos buscar ayuda si la necesitamos. Cuando estamos creando nuestras propios subalgoritmos, es importante que también incluyamos comentarios para guiar a otras personas (y a nosotros mismos en el futuro si nos olvidamos) para qué y cómo se usa lo que estamos desarrollando.

Para ilustrar esto, vamos a recordar que en la práctica 2 escribimos un algoritmo para el cálculo de factoriales. Dado que los mismos son muy útiles en variadas aplicaciones, podemos escribir un subalgoritmo que se encargue de obtenerlos. Luego, escribiremos un algoritmo para mostrar todos los factoriales de los números 1 a 10.

---

### SUBALGORITMOS

---

```
#
# Función fact
# Calcula el factorial de números enteros no negativos
# Entrada:
#   - n, entero no negativo
# Salida:
#   - el factorial de n
#
FUNCIÓN fact(n: numérico): numérico
COMENZAR
  VARIABLE numérica resultado
  resultado <- 1
```

```

SI n > 0 ENTONCES
    PARA i DESDE 1 HASTA n HACER
        resultado <- resultado * i
    FIN PARA
FIN SI
DEVOLVER resultado
FIN FUNCIÓN

```

---

ALGORITMO PRINCIPAL

---

ALGORITMO: "Mostrar los factoriales de los 10 primeros naturales"  
COMENZAR

```

PARA j DESDE 1 HASTA 10 HACER
    ESCRIBIR "El factorial de " j " es igual a " fact(j)
    FIN PARA
FIN

```

En R:

```

# -----
# DEFINICIÓN DE FUNCIONES
# ----

#-----
# Función fact
# Calcula el factorial de números enteros no negativos
# Entrada:
#     - n, entero no negativo
# Salida:
#     - el factorial de n
#-----

fact <- function(n) {
  resultado <- 1
  if (n > 0) {
    for (i in 1:n) {
      resultado <- resultado * i
    }
  }
  return(resultado)
}

# -----
# PROGRAMA PRINCIPAL: Mostrar los factoriales de los 10 primeros naturales
# -----
for (j in 1:10) {
  cat("El factorial de", j, "es igual a", fact(j), "\n")
}

```

El factorial de 1 es igual a 1  
El factorial de 2 es igual a 2  
El factorial de 3 es igual a 6  
El factorial de 4 es igual a 24  
El factorial de 5 es igual a 120  
El factorial de 6 es igual a 720  
El factorial de 7 es igual a 5040  
El factorial de 8 es igual a 40320

El factorial de 9 es igual a 362880  
 El factorial de 10 es igual a 3628800

## 4.5. Pasaje de parámetros

Los algoritmos y subalgoritmos comunican información entre sí a través de los parámetros o argumentos. Esta comunicación recibe el nombre de **pasaje de argumentos** y se puede realizar de dos formas: *por valor* o *por referencia*. Algunos lenguajes de programación trabajan con uno u otro sistema, mientras que otros lenguajes permiten el uso de ambos.

### 4.5.1. Pasaje por valor

En este caso, los argumentos representan valores que se transmiten **desde** el algoritmo **hacia** el subalgoritmo. El **pasaje por valor** implica que los objetos del algoritmo provistos como argumentos en la llamada al subalgoritmo no serán modificados por la ejecución del mismo. Este sistema funciona de la siguiente forma:

1. Se evalúan los argumentos actuales usados en la invocación al subalgoritmo.
2. Los valores obtenidos se *copian* en los argumentos formales dentro del subalgoritmo.
3. Los argumentos formales se usan como variables dentro del subalgoritmo. Aunque los mismos sean modificados (por ejemplo, se les asignen nuevos valores), no se modifican los argumentos actuales en el algoritmo, sólo sus copias dentro del subalgoritmo.

Veamos un ejemplo:

---

SUBALGORITMOS

---

```
FUNCIÓN fun(x: numérico, y: numérico): numérico
COMENZAR
  x <- x + 1
  y <- y * 2
  DEVOLVER x + y
FIN FUNCIÓN
```

---

ALGORITMO PRINCIPAL

---

```
ALGORITMO: "Ejemplo de pasaje de argumentos"
COMENZAR
```

```
  VARIABLE numérica a, b, d
  a <- 3
  b <- 5
  d <- fun(a, b)
  ESCRIBIR a b d
FIN
```

Si el pasaje de argumentos se hace por valor, los cambios producidos en el cuerpo de la función sobre los parámetros formales no son transmitidos a los parámetros actuales en el algoritmo principal. Esto significa que los formales son una “copia” de los actuales. Los pasos que sigue el algoritmo son:

1. En el algoritmo principal, se asignan los valores:  $a = 3$ ,  $b = 5$ .
2. Al invocar la función, se establece la correspondencia:  $x = 3$ ,  $y = 5$ .
3. Primera línea de la función:  $x = 3 + 1 = 4$ .
4. Segunda línea de la función:  $y = 5 * 2 = 10$ .
5. La función devuelve el valor  $x + y = 4 + 10 = 14$ .
6. De regreso en el algoritmo principal:  $d$  recibe el valor 14.
7. El algoritmo escribe: 3 5 14.

En R, el pasaje de argumentos es **por valor**. Por lo tanto, este tipo de comportamiento es lo que vemos cuando implementamos el ejemplo discutido<sup>6</sup>:

```
# -----
# DEFINICIÓN DE FUNCIONES
# -----

fun <- function(x, y) {
  x <- x + 1
  y <- y * 2
  return(x + y)
}

# -----
# PROGRAMA PRINCIPAL
# -----

a <- 3
b <- 5
d <- fun(a, b)
cat(a, b, d)
```

3 5 14

#### 4.5.2. Pasaje por referencia

En este caso, los argumentos no sólo representan valores que se transmiten desde el algoritmo hacia el subalgoritmo, sino también desde el subalgoritmo al algoritmo. Esto sirve en las situaciones en las que se quiere que el subalgoritmo pueda modificar las variables del algoritmo principal que se pasaron como argumentos. De esta manera, un subalgoritmo puede producir uno o varios efectos secundarios en el ambiente del algoritmo.

Si un parámetro se pasa por referencia, todos los cambios que experimente dentro del subalgoritmo se producirán también en la variable externa pasada como argumento. Esto se debe a que la información que es pasada desde el algoritmo al subalgoritmo es la dirección en la memoria de la computadora donde se halla almacenado el parámetro actual, es decir, se pasa una referencia a la variable, no el valor que contiene.

Este sistema funciona de la siguiente forma:

1. Se seleccionan las variables usadas como argumentos actuales.
2. Se asocia cada variable con el argumento formal correspondiente.
3. Los cambios que experimenten los argumentos formales se reflejan también en los argumentos actuales de origen.

Retomemos el ejemplo anterior:

---

#### SUBALGORITMOS

---

```
FUNCIÓN fun(x: numérico, y: numérico): numérico
COMENZAR
  x <- x + 1
  y <- y * 2
  DEVOLVER x + y
FIN FUNCIÓN
```

---

#### ALGORITMO PRINCIPAL

---

<sup>6</sup>**Nota:** En general, se desalienta la reasignación de valor a los parámetros de la función por resultar confuso. Esto quiere decir que en el ejemplo anterior, para evitar ambigüedades, sería recomendable reemplazar  $x \leftarrow x + 1$  por algo como  $z \leftarrow x + 1$  y operar con  $z$ , de modo que no se sobreescriba el valor del parámetro  $x$ . También sería aconsejable hacer algo similar para la  $y$ .

---

ALGORITMO: "Ejemplo de pasaje de argumentos"

COMENZAR

```
VARIABLE numérica a, b, d
a <- 3
b <- 5
d <- fun(a, b)
ESCRIBIR a b d
```

FIN

Si el pasaje de argumentos se hace por referencia, los pasos que sigue el algoritmo son:

1. En el algoritmo principal, se asignan los valores:  $a = 3$ ,  $b = 5$ .
2. Al invocar la función, se establece la correspondencia:  $x = 3$ ,  $y = 5$ .
3. Primera línea de la función:  $x = 3 + 1 = 4$ . El parámetro actual asociado con  $x$ ,  $a$ , es en realidad el que sufre dicho cambio y recibe el valor 4 ( $a = 4$ ).
4. Segunda línea de la función:  $y = 5 * 2 = 10$ . El parámetro actual asociado con  $y$ ,  $b$ , es en realidad el que sufre dicho cambio y recibe el valor 10 ( $b = 10$ ).
5. La función devuelve el valor  $x + y = 4 + 10 = 14$ .
6. De regreso en el algoritmo principal:  $d$  recibe el valor 14.
7. El algoritmo escribe: 4 10 14.

Debe notarse que los resultados difieren dependiendo del tipo de pasaje de argumentos empleado <sup>7</sup>. R no trabaja con pasaje por referencia (aunque es posible forzar a que haga algo similar, si así se lo desea).

## 4.6. Ámbito de las variables

En todo lenguaje de programación se le dice **ámbito** o **scope** a la región del programa donde una variable definida existe y es visible, tal que fuera de dicha región no se puede acceder a la misma <sup>8</sup>. Según el ámbito en el que existen, las variables pueden considerarse *locales* o *globales*.

### 4.6.1. Variables locales

Las variables declaradas dentro de un subalgoritmo (por ejemplo, dentro de una función) se llaman **variables locales**. Sólo pueden ser usadas por las instrucciones que están dentro de esa función, mientras que el programa principal u otros subalgoritmos desconocen su existencia y no las pueden usar. Las *variables locales* residen en el *ambiente local* de un subalgoritmo y no tienen nada que ver con las variables que puedan ser declaradas con el mismo nombre en otros lugares<sup>9</sup>. En el siguiente ejemplo, las variables  $a$  y  $b$  son locales a la función  $f1$  y no se pueden usar desde el programa principal, porque dejan de existir una vez que termina la ejecución de  $f1$ :

---

SUBALGORITMOS

---

FUNCTION f1(x: numérico): numérico

COMENZAR

```
VARIABLE numérica a, b
a <- x - 10
b <- x + 10
DEVOLVER a + b
```

FIN FUNCIÓN

---

<sup>7</sup>Para diferenciar subalgoritmos con pasaje por referencia, algunos autores sugieren distinguir la declaración de los parámetros formales con algún símbolo, por ejemplo &.

<sup>8</sup>No sólo las variables pertenecen a un ámbito, sino todos los objetos que se puedan crear, sean estos variables, constantes o subalgoritmos.

<sup>9</sup>Cuando otro subalgoritmo utiliza el mismo nombre se refiere a una posición diferente en memoria.

## ALGORITMO PRINCIPAL

---

ALGORITMO: "Ejemplo"  
 COMENZAR  
 VARIABLE numérica z  
 $z \leftarrow f1(50)$   
 ESCRIBIR z  
 ESCRIBIR  $z + a$  ---LÍNEA CON ERROR---  
 FIN

---

# -----  
*# DEFINICIÓN DE FUNCIONES*  
# -----

```
f1 <- function(x) {  

  a <- x - 10  

  b <- x + 10  

  return(a + b)
}
```

---

# -----  
*# PROGRAMA PRINCIPAL*  
# -----

```
z <- f1(50)  

z
```

```
[1] 100  

z + a
```

```
Error in eval(expr, envir, enclos): object 'a' not found
```

El error se genera porque el algoritmo principal quiere usar a la variable *a*, la cual es local a la función *f1()* y sólo existe dentro de la misma.

El uso de *variables locales* tiene muchas ventajas. Permiten independizar al subalgoritmo del algoritmo principal, ya que las variables definidas localmente en un subalgoritmo no son reconocidas fuera de él. La comunicación entre el subalgoritmo y el algoritmo principal se da exclusivamente a través de la lista de parámetros. Esta característica hace posible dividir grandes proyectos en piezas más pequeñas y que, por ejemplo, diferentes programadores puedan trabajar independientemente en un mismo proyecto.

#### 4.6.2. Variables globales

Las variables globales son las que se definen en el algoritmo principal y pueden ser usadas dentro de los subalgoritmos, aún cuando no se las pase como argumento. En el ejemplo anterior *z* es una variable global<sup>10</sup>. Las *variables globales* residen en el *ambiente global* del algoritmo.

El siguiente ejemplo muestra cómo la función *f2* puede hacer uso de una variable global y que fue definida fuera de ella, en el programa principal<sup>11</sup>:

---

SUBALGORITMOS

---

<sup>10</sup>*f1* también es global: todo tipo de objeto, incluso las funciones, pertenecen a un determinado ambiente

<sup>11</sup>Algunos autores sugieren agregar la palabra *GLOBAL* o *LOCAL* en la declaración de las variables para distinguir su ambiente, por ejemplo, poner dentro de *f2* VARIABLE LOCAL numérica *a* y en el algoritmo VARIABLE GLOBAL *y*, pero no seguiremos esta práctica para ganar en sencillez de escritura. También algunos lenguajes de programación requieren señalar de alguna manera especial a las variables globales.

```

FUNCIÓN f2(x: numérico): numérico
COMENZAR
    VARIABLE numérica a
    a <- x * y
    DEVOLVER a
FIN FUNCIÓN

```

---

ALGORITMO PRINCIPAL

---

ALGORITMO: "Ejemplo"

```

COMENZAR
    VARIABLE numérica y
    y <- 20
    ESCRIBIR f2(2)
    y <- 18
    ESCRIBIR f2(2)
FIN

```

```

# -----
# DEFINICIÓN DE FUNCIONES
# -----

```

```

f2 <- function(x) {
    a <- x * y
    return(a)
}

```

```

# -----
# PROGRAMA PRINCIPAL
# -----

```

```

y <- 20
f2(2)

```

```
[1] 40
```

```

y <- 18
f2(2)

```

```
[1] 36
```

La función pudo hacer uso de la variable global `y` sin haberse comunicado con el programa principal a través de los argumentos. Esta práctica no es recomendable: si bien evaluemos `f2(2)` dos veces, el resultado no fue el mismo, porque depende de cuánto vale `y` en el ambiente global en el momento que `f2` es invocada. Además de ser confuso, esto es una violación al principio de *transparencia referencial*: un subalgoritmo sólo debe utilizar elementos mencionados en la lista de argumentos o definidos localmente, sin emplear variables globales. En particular, si hablamos de una función donde el pasaje de parámetros es por valor, esta práctica garantiza que la misma siempre devuelva el mismo resultado cada vez que sea invocada con los mismos valores en los argumentos de entrada, sin producir ningún efecto secundario en el algoritmo principal. El uso de variables globales permite escribir subalgoritmos que carecen de transparencia referencial.

Un algoritmo puede usar el mismo nombre para variables locales y globales, pero dentro de una función toma precedencia la variable local. En el siguiente ejemplo, hay una variable global `a` en el programa principal que recibe el valor 70. Y hay otra variable `a` que es local a la función `f3`. Cuando `f3` calcula `a + b`, lo hace con el valor de su variable local (`x - 10`) y no con el valor de la variable global (70):

## SUBALGORITMOS

---

```

FUNCIÓN f3(x: numérico): numérico
COMENZAR
    VARIABLE numérica a, b
    a <- x - 10
    b <- x + 10
    ESCRIBIR "Acá, dentro de la f3, el valor de a es", a
    DEVOLVER a + b
FIN FUNCIÓN

```

---

## ALGORITMO PRINCIPAL

---

```

ALGORITMO: "Ejemplo"
COMENZAR
    VARIABLE numérica a, z
    a <- 70
    z <- f3(50)
    ESCRIBIR z
    ESCRIBIR "Acá, en el programa principal, el valor de a es", a
    ESCRIBIR a + z
FIN

```

---

```

# -----
# DEFINICIÓN DE FUNCIONES
# -----

```

```

f3 <- function(x) {
  a <- x - 10
  b <- x + 10
  cat("Acá, dentro de la f3, el valor de a es", a)
  return(a + b)
}

```

```

# -----
# PROGRAMA PRINCIPAL
# -----

```

```

a <- 70
z <- f3(50)

```

Acá, dentro de la f3, el valor de a es 40

z

[1] 100

```

cat("Acá, en el programa principal, el valor de a es", a)

```

Acá, en el programa principal, el valor de a es 70

a + z

[1] 170

Se debe prestar atención que con la sentencia ESCRIBIR o la función cat() en R se muestra en pantalla un mensaje

en el momento en el que se ejecuta esa acción. Si el mensaje incluye mostrar valores guardados en objetos, se mostrarán los valores que los mismos tienen en ese momento. Por otro lado, lo devuelto por la sentencia **DEVOLVER** o la función **return()** es el resultado de la ejecución de la función: el valor que la función entrega puede ser asignado a otro objeto en el algoritmo principal, como ocurre en la línea de ***z <- f3(50)***.

## 4.7. Otras nociones importantes en R

### 4.7.1. La función **source()**

Cuanto más grande o complejo es el problema a resolver, más funciones deben ser programadas y no es necesario escribirlas a todas en el mismo archivo de código del programa principal. Para ser más ordenados, podemos escribir nuestras funciones en uno o más archivos separados. Si hacemos esto, en el comienzo del script del programa principal debemos incluir una sentencia para que en primer lugar se ejecute el código guardado en esos otros archivos, de modo que las funciones sean definidas y formen parte del ambiente global.

Consideremos otra vez el ejemplo de la función para el cálculo de factoriales. Podemos guardar el código de esta función (y otras si hubiese) en un archivo llamado **funciones.R**, con el siguiente contenido:

```
# -----
# Función fact
# Calcula el factorial de números enteros no negativos
# Entrada:
#   - n, entero no negativo
# Salida:
#   - el factorial de n
# -----
fact <- function(n) {
  resultado <- 1
  if (n > 0) {
    for (i in 1:n) {
      resultado <- resultado * i
    }
  }
  return(resultado)
}
```

Luego, en cualquier problema que requiera el cálculo de factoriales, vamos a pedirle a R que ejecute el código guardado en el archivo **funciones.R** con la sentencia **source()**, como paso inicial en el archivo donde estemos escribiendo el programa principal. Por ejemplo:

```
# -----
# PROGRAMA PRINCIPAL: Mostrar los factoriales de los 10 primeros naturales
# -----
```

```
source("C:/Documentos/Facultad/IALP/funciones.R")

for (j in 1:10) {
  cat("El factorial de", j, "es igual a", fact(j), "\n")
```

Gracias a **source()** todas las funciones definidas en el archivo **funciones.R** aparecerán en el entorno y no hay necesidad ni siquiera de abrirlo. Notar que **C:/Documentos/Facultad/IALP/** es la dirección o *path* de la carpeta en la computadora donde está guardado el archivo **funciones.R**.

### 4.7.2. Argumentos con valores asignados por defecto

Hemos visto que algunos argumentos de las funciones predefinidas de R tienen valores asignados por defecto, como es el caso de la función **log()**, que a menos que indiquemos otra cosa opera con la base natural. Cuando definimos nuestras propias funciones, también es posible asignarle un valor por defecto a uno o más de sus argumentos.

Tomemos el primer ejemplo de este capítulo:

```
f <- function(x, y) {
  resultado <- x^2 + 3 * y
  return(resultado)
}
f(4, 5)
```

[1] 31

Esta función también podría ser definida así:

```
nueva_f <- function(x, y = 100) {
  resultado <- x^2 + 3 * y
  return(resultado)
}
```

Esto significa que si no proveemos un valor para el argumento *y*, a este se le asignará por default el valor 100. Luego:

```
nueva_f(4)
```

[1] 316

En el caso anterior, se hace corresponder el 4 al primer argumento de la función, *x*, y como no hay ningún otro parámetro actual que le estemos pasando a la función, la misma le asigna a *y* el valor 100 y calcula:  $x^2 + 3 * y = 16 + 300 = 316$ . Sin embargo, podemos, como antes, proveer cualquier otro valor para *y*, de modo que no se use el valor por default:

```
nueva_f(4, 5)
```

[1] 31

Como *x* no tiene valor asignado por default en la función *nueva\_f()*, siempre debemos pasarle un valor. En caso contrario, recibiremos un error:

```
nueva_f()
```

Error in *nueva\_f()*: argument "x" is missing, with no default

```
nueva_f(y = 5)
```

Error in *nueva\_f(y = 5)*: argument "x" is missing, with no default

## 4.8. Otros tópicos de lectura opcional

### 4.8.1. Modificar una variable global desde el cuerpo de una función en R

Hemos dicho que una función recibe información desde el programa principal a través de sus parámetros, y envía información al mismo mediante el valor que devuelve. Sin embargo, es posible alterar el comportamiento para que sea capaz de producir efectos secundarios, por ejemplo, modificando el valor de una variable global, violando así el principio de transparencia referencial.

Los siguientes ejemplos definen dos funciones con un único argumento, *x*, pero que en su cuerpo hacen uso de una variable global, *y*, definida el algoritmo principal (estos casos violan el principio de transparencia referencial, su práctica no es recomendable). La diferencia entre ellas es que *g1()* modifica el valor de *y* dentro de la función, pero el valor de *y* en el ambiente global no es alterado; mientras que *g2()* cambia el valor de *y* no sólo localmente, sino también en el ambiente global. Esto se logra mediante el uso del operador  $\ll<^{12}$ .

<sup>12</sup>En realidad, el operador  $\ll<$  trabaja de forma más compleja que lo mencionado en esta guía. En programas más elaborados de R, pueden haber ambientes anidados y este operador iniciará una búsqueda desde el ambiente actual hacia los superiores hasta encontrar una variable que se llame *y* para asignarle un valor. En este caso sencillo, sólo hay dos ambientes, el de la función *g2* y el global. Por lo tanto el operador  $\ll<$  hace que se le asigne un valor a la variable *y* en el global.

```

# -----
# DEFINICIÓN DE FUNCIONES
# -----

g1 <- function(x) {
  y <- y + 100
  return(x / y)
}

g2 <- function(x) {
  y <<- y + 100
  return(x / y)
}

# -----
# PROGRAMA PRINCIPAL
# -----



# Caso 1: el valor de y en el ambiente global no es modificado por g1
x <- 500
y <- 50
z <- g1(x)
cat(x, y, z)

500 50 3.333333

# Caso 2: el valor de y en el ambiente global es modificado por g2
x <- 500
y <- 50
z <- g2(x)
cat(x, y, z)

500 150 3.333333

```

Nuevamente, esta forma de trabajo no es aconsejable porque estamos produciendo *efectos secundarios* desde la función en el ambiente global que pueden pasar desapercibidos si no estamos muy atentos. Así como la mejor práctica es pasar toda la información desde el programa principal hacia la función a través de sus parámetros, también es recomendable que toda comunicación desde la función hacia el programa principal se realice a través del valor (u objeto) que la función devuelve, sin producir efectos secundarios (transparencia referencial).

#### 4.8.2. Procedimientos

Un **procedimiento** es un subalgoritmo que agrupa una acción o conjunto de acciones, dándoles un nombre por el que se las puede identificar posteriormente. Se diferencia de la función en que no tiene como objetivo, en general, devolver un valor, sino sólo contribuir a la descomposición o modularidad del programa. R no trabaja con el concepto de *procedimiento*, sino que todo tipo de subalgoritmo se genera con la misma estructura de `function(...){...}`<sup>13</sup>.

Como en las funciones, desde el algoritmo principal se pasan valores al procedimiento utilizando **parámetros** o **argumentos**, aunque también puede haber procedimientos que carezcan de los mismos. Para usar un procedimiento hay que invocarlo, escribiendo su nombre y a continuación, si los hay, los valores de los argumentos actuales para esa llamada, separados por comas. Aquí también los argumentos actuales deben ser compatibles en cuanto a la cantidad, tipo y orden que los argumentos formales declarados en la definición del procedimiento.

En el siguiente ejemplo podemos identificar los argumentos actuales **a** (con el valor 5), **b** (con el valor 2), **c** y **d** (sin valores asignados inicialmente). Cuando el procedimiento `proced1` es invocado, se establece una correspondencia con los argumentos formales **n1**, **n2**, **n3** y **n4**, respectivamente. **n1** toma el valor 5, **n2** toma el valor 2 y el procedimiento le asigna los valores 7 a **n3** y 1 a **n4**. Al finalizar, este procedimiento habrá dejado sin cambios a las variables **a** y

<sup>13</sup>En la jerga de R, se habla de *funciones puras* o *no puras*. Las primeras no producen ningún efecto secundario en el ambiente global, las segundas sí. Hay muy pocas funciones no puras en R

b, mientras que le habrá asignado los valores 7 a c y 1 a d. Como resultado, el algoritmo escribe “5 2 7 1”.

## SUBALGORITMOS

```
PROCEDIMIENTO proced1(n1: numérico, n2: numérico, n3: numérico, n4: numérico)
    n3 <- n1 + n2
    n4 <- n2 - 1
FIN PROCEDIMIENTO
```

## ALGORITMO PRINCIPAL

ALGORITMO: Primer ejemplo de procedimiento

```
COMENZAR
    VARIABLE numérica a, b, c, d
    a <- 5
    b <- 2
    proced1(a, b, c, d)
    ESCRIBIR a b c d
FIN
```

En el siguiente ejemplo, el procedimiento **proced2** modifica las variables que actúan como argumentos actuales. Al ser invocado, se establece una correspondencia entre los argumentos actuales **a** (con el valor 5) y **b** (con el valor 2), y los argumentos formales **n1** y **n2**, respectivamente. De esta forma, la primera acción del procedimiento le asigna el valor 7 a **n1** y 1 a **n2**. De esta manera, al finalizar **a** vale 7 y **b** vale 1 y el algoritmo escribe “7 1”.

## SUBALGORITMOS

```
PROCEDIMIENTO proced2(n1: numérico, n2: numérico)
    n1 <- n1 + n2
    n2 <- n2 - 1
FIN PROCEDIMIENTO
```

## ALGORITMO PRINCIPAL

ALGORITMO: Segundo ejemplo de procedimiento  
COMENZAR

```
VARIABLE numérica a, b
a <- 5
b <- 2
proced2(a, b)
ESCRIBIR a b
FIN
```

Analicemos ahora el tipo de pasaje de argumentos en el contexto de un procedimiento:

## SUBALGORITMOS

```
PROCEDIMIENTO miProc(x: numérico, y: numérico)
    x <- x * 2
```

```

y <- x - y
FIN PROCEDIMIENTO

```

ALGORITMO PRINCIPAL

ALGORITMO: Tercer ejemplo de procedimiento

COMENZAR

```

VARIABLE numérica a, b
a <- 8
b <- 4
miProc(a, b)
ESCRIBIR a b

```

FIN

Si el pasaje es por referencia, los pasos que sigue el algoritmo serían:

1. En el algoritmo se asignan los valores:  $a = 8$ ,  $b = 4$ .
2. Al invocar la función:  $x = 8$ ,  $y = 4$ .
3. Primera línea de la función:  $x = 8 * 2 = 16$ . Lo mismo sucede con el parámetro actual  $a$ :  $a = 16$ .
4. Segunda línea de la función:  $y = 16 - 4 = 12$ . Lo mismo sucede con el parámetro actual  $b$ :  $b = 12$ .
5. Al regresar al algoritmo principal, la sentencia ESCRIBIR produce: 16 12.

Si el pasaje hubiese sido por valor,  $a$  y  $b$  no hubiesen cambiado y la sentencia ESCRIBIR mostraría 8, 4. Como en un procedimiento los resultados regresan en los mismos parámetros, no pueden ser todos pasados por valor, porque en ese caso el procedimiento nunca realizaría ninguna acción.

Si el parámetro  $x$  se pasa por valor mientras que  $y$  se pasa por referencia, los pasos serían:

1.  $a = 8$ ,  $b = 4$ .
2. Al invocar la función:  $x = 8$ ,  $y = 4$ .
3. Primera línea de la función:  $x = 8 * 2 = 16$ .
4. Segunda línea de la función:  $y = 16 - 4 = 12$ . Lo mismo sucede con el parámetro actual  $b$ :  $b = 12$ .
5. Al regresar al algoritmo principal, la sentencia ESCRIBIR produce: 8 12.

Dado que en R el pasaje de argumentos es **siempre** por valor y no por referencia, no existen los procedimientos. Por esta razón, esta sección no se ejemplifica con código de R.



# Unidad 5

## Estructuras de datos

Hasta ahora todos los algoritmos que hemos desarrollado hacen uso de objetos que guardan datos individuales, los cuales representan un número, una cadena de texto o un valor lógico. Sin embargo, la verdadera utilidad de la computación radica en poder trabajar con conjuntos de datos, organizados de acuerdo a ciertas reglas que permitan su manipulación y acceso. Definimos entonces como **estructura de datos** a un conjunto de datos que cuentan con un sistema de organización.

### 5.1. Arreglos

Un **arreglo** se define como una colección de valores individuales con dos características fundamentales:

- *Ordenamiento*: los valores individuales pueden ser enumerados en orden, es decir, debe ser posible identificar en qué posición del arreglo se encuentra cada valor.
- *Homogeneidad*: los valores individuales almacenados en un arreglo son todos del mismo tipo (numérico, carácter, lógico).

Los arreglos son muy útiles para almacenar información en la memoria de la computadora, organizando valores que estén relacionados entre sí de alguna manera, por ejemplo, una conjunto de precios, los meses del año, el listado de calificaciones de estudiantes en distintos parciales, etc.

Los componentes individuales del conjunto se llaman **elementos**. Para indicar qué posición ocupa cada elemento en el arreglo se emplean uno o más **índices**. Dependiendo de cuántos índices se deban utilizar para acceder a cada elemento dentro de los arreglos, estos se clasifican en **unidimensionales** (*vectores*) o **bidimensionales** (*matrices*). También existen los arreglos **multidimensionales** y están presentados al final de este capítulo, pero como no trabajaremos la lectura de esa sección es opcional.

#### 5.1.1. Arreglos unidimensionales o vectores

Un **arreglo unidimensional** o **vector** tiene  $n$  elementos todos del mismo tipo. Por ejemplo, el siguiente es un vector de tipo numérico llamado **x** con 5 elementos:

x	-4.5	12	2.71	-6	25
---	------	----	------	----	----

Figura 5.1: Ejemplo de un vector numérico

Cada uno de los elementos ocupa una posición determinada en el vector. Por ejemplo, el elemento 3 del vector **x** es el número 2.71. Se puede *acceder* o hacer referencia a cada elemento mediante el uso de *índices*, expresados entre corchetes al lado del nombre del vector. De esta forma, si escribimos **x[3]** hacemos referencia a la tercera posición del vector, que actualmente guarda al valor 2.71. Como podemos ver, sólo hace falta un índice para hacer referencia a cada elemento de un vector.

Los siguientes son ejemplos de vectores de tipo carácter y lógico, con distintas cantidades de elementos:

Al igual que todas las variables que empleamos en nuestros algoritmos, los vectores que serán utilizados deben ser

x	-4.5	12	2.71	-6	25
	x[1]	x[2]	x[3]	x[4]	x[5]

Figura 5.2: Ejemplo de un vector numérico: índices para señalar cada posición.

y	"ARG"	"correo@gmail.com"	"Ok"	"chau"
	y[1]	y[2]	y[3]	y[4]

z	VERDADERO	VERDADERO	FALSO
	z[1]	z[2]	z[3]

Figura 5.3: Ejemplo de un vector carácter y un vector lógico

declarados en el pseudocódigo, eligiendo un *identificador* (nombre) e indicando su tipo y su tamaño, es decir, la cantidad de posiciones que contienen. Esto último se señala entre paréntesis al lado del nombre elegido. Por ejemplo, el vector x visto anteriormente puede ser creado de la siguiente forma:

```
VARIABLE numérica x(5)
x[1] <- -4.5
x[2] <- 12
x[3] <- 2.71
x[4] <- -6
x[5] <- 25
```

Si bien la declaración de un vector sólo tiene como objetivo permitirle a la computadora que reserve internamente el espacio necesario en memoria para el mismo, para escribir pseudocódigo de una manera sencilla estableceremos la siguiente **convención**. Cuando declaramos un vector de tipo numérico con la expresión VARIABLE numérica x(5) asumiremos que, además de reservar espacio en memoria para el vector, se le asigna un 0 (cero) en cada posición. Es decir, el vector x es iniciado con ceros, que más tarde pueden ser reemplazados por otros valores. Del mismo modo, asumiremos que cuando declaramos vectores de tipo carácter, todos sus elementos son iniciados con valores (una cadena de texto vacía) y cuando declaramos vectores de tipo lógico, con el valor FALSO.

En R, los vectores se construyen de forma dinámica por lo cual no es necesario declararlos antes de comenzar a utilizarlos. La función **c()** (de *combinar*) permite crear vectores, por ejemplo, los mencionados anteriormente:

```
x <- c(-4.5, 12, 2.71, -6, 25)
y <- c("ARG", "correo@gmail.com", "Ok", "chau")
z <- c(TRUE, TRUE, FALSE)
```

Cuando ejecutamos dichas líneas, se crean en el ambiente global los objetos x, y y z, como podemos notar en la pestaña Environment de RStudio. Es decir, los vectores, así como cualquier otro tipo de arreglo, son **objetos** que constituyen entidades en sí mismas y que pueden ser manipulados al hacer referencia a sus indicadores. Además, RStudio nos muestra en la pestaña mencionada qué tipo de vector es cada uno (**num**, **chr**, **logi**), cuántos elementos tiene ([1:5], [1:4], [1:3]) y una previsualización de sus primeros elementos.

Dado que la función **c()** resulta, en consecuencia, muy importante al programar en R, es recomendable que evitemos usar la letra **c** como nombre para otros objetos<sup>1</sup>.

Podemos emplear estructuras iterativas para recorrer todas las posiciones de un vector y realizar operaciones con ellas, por ejemplo:

```
PARA i DESDE 1 HASTA 5 HACER
  ESCRIBIR "La posición " i "de x está ocupada por el valor " x[i]
FIN PARA
```

<sup>1</sup>Aunque a veces me olvido de esta recomendación y por eso en algunos ejemplos de la asignatura hay variables llamadas c

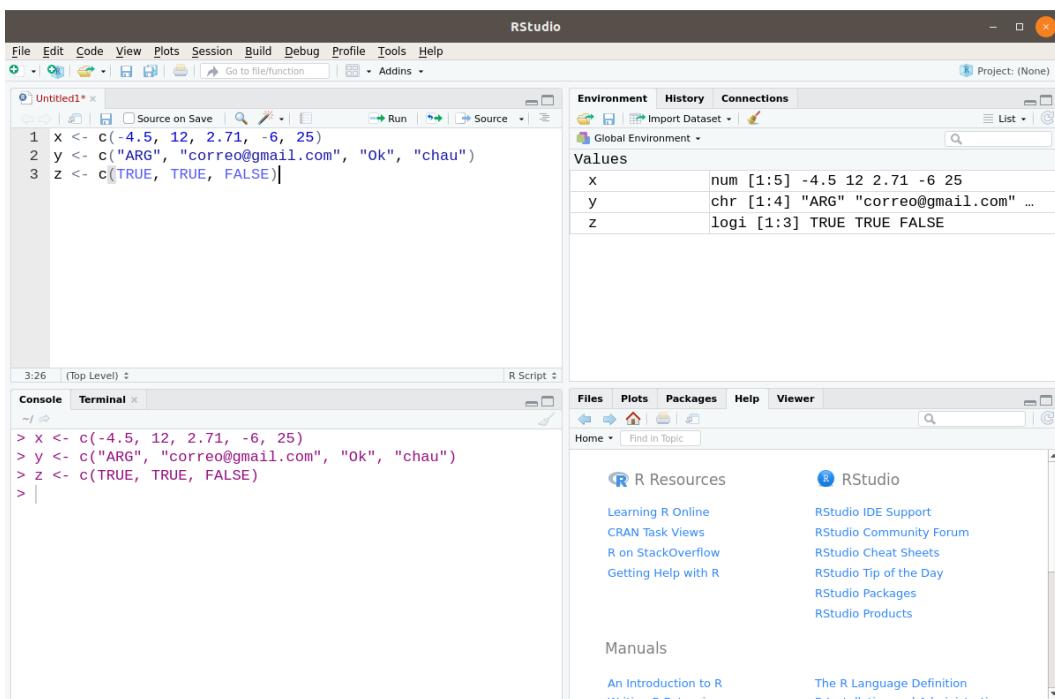


Figura 5.4: Creación de vectores en R

```

for (i in 1:5) {
  cat("La posición ", i, "de x está ocupada por el valor", x[i], "\n")
}

```

La posición 1 de x está ocupada por el valor -4.5  
 La posición 2 de x está ocupada por el valor 12  
 La posición 3 de x está ocupada por el valor 2.71  
 La posición 4 de x está ocupada por el valor -6  
 La posición 5 de x está ocupada por el valor 25

Todos los lenguajes de programación incluyen, además, alguna función para determinar cuántos elementos tiene un vector que ya fue creado. Para esto emplearemos la expresión LARGO() en el pseudocódigo y la función `length` de R:

```

ESCRIBIR "El vector x tiene " LARGO(x) " elementos."
ESCRIBIR "El vector y tiene " LARGO(y) " elementos."
ESCRIBIR "El vector z tiene " LARGO(z) " elementos.

cat("El vector x tiene", length(x), "elementos.")

```

El vector x tiene 5 elementos.

```
cat("El vector y tiene", length(y), "elementos.")
```

El vector y tiene 4 elementos.

```
cat("El vector z tiene", length(z), "elementos.")
```

El vector z tiene 3 elementos.

Entonces, para recorrer todos los elementos del vector podemos hacer también:

```

PARA i DESDE 1 HASTA LARGO(x) HACER
  ESCRIBIR "La posición " i "de x está ocupada por el valor " x[i]

```

FIN PARA

O bien:

```
tam <- LARGO(x)
PARA i DESDE 1 HASTA tam HACER
    ESCRIBIR "La posición " i "de x está ocupada por el valor " x[i]
FIN PARA
```

Antes comentamos que en R los vectores se crean con expresiones como `x <- c(-4.5, 12, 2.71, -6, 25)`, donde sus elementos están listados de forma literal. También podemos crear vectores de un largo determinado dejando que cada posición quede ocupada por algún valor asignado por defecto. Por ejemplo, el siguiente código crea un vector tipo numérico con 10 posiciones, uno carácter con 7 y otro lógico con 2. En cada caso, R rellena todas las posiciones con el mismo valor: ceros en el vector numérico, caracteres vacíos en el vector de tipo carácter y valores FALSE en el vector lógico:

```
a <- numeric(10)
b <- character(7)
d <- logical(2)

a
[1] 0 0 0 0 0 0 0 0 0 0

b
[1] "" "" "" "" "" ""

d
[1] FALSE FALSE
```

Se pueden asignar valores a una, varias o todas las posiciones de un vector en cualquier parte del algoritmo. Además, en pseudocódigo emplearemos la palabra clave MOSTRAR cuando deseamos que se escriba en pantalla todo el contenido de un vector. Por ejemplo:

```
VARIABLE numérica a(10)
...algunas acciones...
PARA i DESDE 1 HASTA LARGO(a) HACER
    SI i MOD 3 == 0 ENTONCES
        a[i] <- i * 100
    FIN SI
FIN PARA
MOSTRAR a
```

```
a <- numeric(10)
for (i in 1:length(a)) {
    if (i %% 3 == 0) {
        a[i] <- i * 100
    }
}
a
```

```
[1] 0 0 300 0 0 600 0 0 900 0
```

En los ejemplos anteriores, declaramos los vectores explicitando su tamaño con un número: VARIABLE numérica x(5) o VARIABLE numérica a(10). Sin embargo, el tamaño del vector podría estar guardado en otra variable, cuyo valor se determina en cada ejecución del programa mediante información externa o como resultado de algún cálculo anterior. En el siguiente ejemplo se deja que el usuario determine la dimensión del vector y que provea cada uno de los valores para el mismo. Antes de poder declarar la existencia del nuevo vector llamado `mi_vector`, se “lee” su tamaño:

```
VARIABLE numérica tam
```

```

LEER tam
VARIABLE numérica mi_vector(tam)
PARA i DESDE 1 HASTA tam HACER
    LEER mi_vector[i]
FIN PARA

```

Por ahora, toda instrucción de *leer* en el pseudocódigo será traducida en R mediante la asignación directa de valores. Por ejemplo, LEER *tam* se reemplaza por *tam* <- 5 (o el número que necesitemos).

Antes de terminar esta sección haremos una última observación. En R todos los objetos que hemos considerado como “variable” y que guardan un único valor (como *tam* en el ejemplo anterior), son también considerados como vectores, cuyo largo es 1, como podemos verificar en el siguiente ejemplo:

```

x <- 25
length(x)

[1] 1

is.vector(x) # Esta función lógica le pregunta a R si el objeto x es un vector

```

#### Ejemplo: invertir los elementos de un vector

Nos planteamos el problema de dar vuelta los elementos pertenecientes a un vector, de manera que el primer elemento pase a ser el último, el segundo pase al penúltimo lugar, etcétera. Por ejemplo, dado el vector de tipo carácter *v*:

V	“Estadística”	“en”	“Licenciatura”	“la”	“Aguante”
---	---------------	------	----------------	------	-----------

Figura 5.5: Vector v original

queremos modificarlo para obtener:

V	“Aguante”	“la”	“Licenciatura”	“en”	“Estadística”
---	-----------	------	----------------	------	---------------

Figura 5.6: Vector v reordenado

Si bien podemos pensar en distintas formas para resolver este problema, probablemente la más sencilla requiere que intercambiemos de a dos los valores en ciertas posiciones del vector, por ejemplo, empezando por intercambiar el primero con el último. Para esto podemos emplear una variable auxiliar que guarde el valor de alguna de las celdas temporalmente (por eso lo vamos a llamar *tmp*):

Ahora sólo resta realizar el mismo procedimiento para los valores de las posiciones 2 y 4. Como el número de elementos en el vector es impar, el valor en la posición central queda en su lugar. Podemos definir el siguiente algoritmo para resolver este problema de manera general. En el siguiente pseudocódigo, primero declaramos una variable numérica *n* que puede tomar cualquier valor y que servirá para declarar cuántos espacios necesita el vector. Luego, se itera para leer cada elemento del vector. Finalmente, se implementa la estrategia de reordenamiento:

**ALGORITMO: "Invertir (dar vuelta) los elementos de un vector"**  
COMENZAR

```

# Declarar variables
VARIABLE numérica n
VARIABLE carácter tmp
LEER n
VARIABLE carácter v(n)

# Asignar valores al vector

```

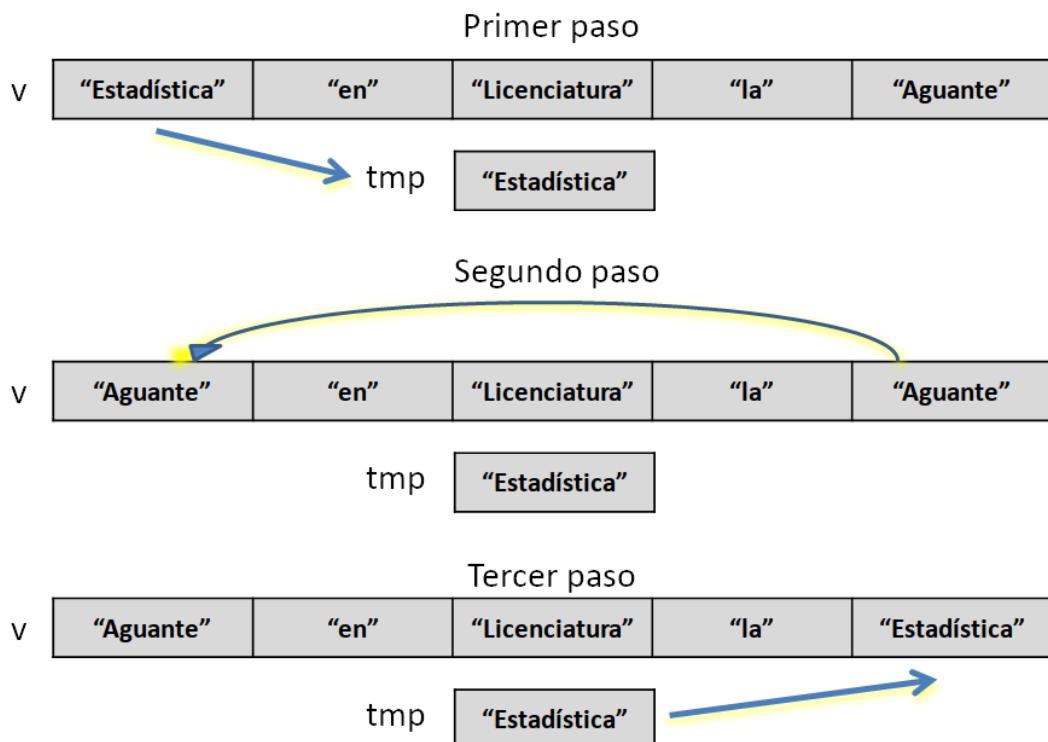


Figura 5.7: Pasos para intercambiar valores

```

PARA i DESDE 1 HASTA n HACER
    LEER v[i]
FIN PARA

# Reordenar
PARA i DESDE 1 HASTA n DIV 2 HACER
    tmp <- v[i]                      # Paso 1
    v[i] <- v[n - i + 1]              # Paso 2
    v[n - i + 1] <- tmp              # Paso 3
FIN PARA

# Mostrar el vector reordenado
MOSTRAR v

FIN

```

En el ejemplo anterior hemos incorporado el uso de comentarios en el pseudocódigo para describir el objetivo de cada parte. Imitando lo que hacemos en R, señalamos la presencia de comentarios con el carácter `#` (podríamos usar otra cosa, pero adheriremos a esta convención). Se usó el operador `DIV` para obtener la división entera entre `n` y 2 (por ejemplo, `5 DIV 2 = 2`). En R reemplazamos todas las instrucciones `LEER` por una asignación directa de valores y empleamos el operador de división entera `%/%`:

```

v <- c("Estadística", "en", "Licenciatura", "la", "Aguante")
n <- length(v)
for (i in 1:(n%/%2)) {
    tmp <- v[i]
    v[i] <- v[n - i + 1]
    v[n - i + 1] <- tmp
}
v

```

```
[1] "Aguante"      "la"          "Licenciatura" "en"      "Estadística"
```

### 5.1.2. Arreglos bidimensionales o matrices

Un **arreglo bidimensional** representa lo que habitualmente conocemos en matemática como **matriz** y por eso también lo llamamos de esa forma. Podemos *imaginar* que en una matriz los elementos están organizados en *filas* y *columnas* formando una tabla. Por ejemplo, la siguiente es una matriz llamada **x**:

	<b>8</b>	<b>13</b>	<b>18</b>	<b>23</b>
<b>x</b>	<b>11</b>	<b>16</b>	<b>21</b>	<b>26</b>
	<b>14</b>	<b>19</b>	<b>24</b>	<b>29</b>

Figura 5.8: Ejemplo de una matriz numérica

A diferencia de los vectores, las matrices requieren dos índices para señalar la posición de cada elemento, el primero para indicar la fila y el segundo para indicar la columna. Los mismos se colocan entre corchetes, separados por una coma, al lado del identificador de la matriz. De esta forma, si hablamos de **x[1, 3]** hacemos referencia a la posición ocupada por el valor 18, mientras que si mencionamos **x[3, 1]** nos referimos al valor 14.

	<b>8</b> x[1, 1]	<b>13</b> x[1, 2]	<b>18</b> x[1, 3]	<b>23</b> x[1, 4]	Fila 1
<b>x</b>	<b>11</b> x[2, 1]	<b>16</b> x[2, 2]	<b>21</b> x[2, 3]	<b>26</b> x[2, 4]	Fila 2
	<b>14</b> x[3, 1]	<b>19</b> x[3, 2]	<b>24</b> x[3, 3]	<b>29</b> x[3, 4]	Fila 3

Columna 1      Columna 2      Columna 3      Columna 4

Figura 5.9: Ejemplo de una matriz numérica: índices para señalar cada posición

Al **tamaño** de una matriz, es decir, cuántas filas y columnas tiene, se le dice **dimensión**. La matriz anterior es de dimensión  $3 \times 4$ .

Como hicimos con los vectores, debemos declarar las matrices que vamos a usar en el pseudocódigo, indicando su identificador, tipo y dimensión: **VARIABLE numérica x(3, 4)**. También vamos a asumir que todas las posiciones de una matriz son iniciadas con el valor 0, o **FALSO** si la misma es numérica, carácter o lógica, respectivamente. La matriz **x** puede ser generada en pseudocódigo de esta forma:

```
VARIABLE numérica x(3, 4)
x[1, 1] <- 8
x[1, 2] <- 13
x[1, 3] <- 18
x[1, 4] <- 23
x[2, 1] <- 11
x[2, 2] <- 16
x[2, 3] <- 21
x[2, 4] <- 26
x[3, 1] <- 14
x[3, 2] <- 19
x[3, 3] <- 24
x[3, 4] <- 29
```

En R, no es necesario declarar las matrices con anterioridad y las mismas pueden ser creadas de manera literal con la función `matrix()`. Su primer argumento, `data`, es un vector con todos los elementos que queremos guardar en la matriz. Luego, se indica la cantidad de filas para la misma con `nrow` y la cantidad de columnas con `ncol`:

```
x <- matrix(data = c(8, 11, 14, 13, 16, 19, 18, 21, 24, 23, 26, 29),
             nrow = 3, ncol = 4)
x
```

```
[,1] [,2] [,3] [,4]
[1,]    8    13    18    23
[2,]   11    16    21    26
[3,]   14    19    24    29
```

Notar que R ubicó a los valores provistos llenando primero la columna 1, luego la 2, etc. Ese comportamiento puede ser modificado con el argumento `byrow`, que por default es `FALSE`. Si lo cambiamos a `TRUE` los elementos son ubicados por fila. Además, podemos usar saltos de líneas (`enter`) para visualizar las diferentes filas de la matriz. Esto no tiene ningún impacto en R, sólo sirve para que el código sea más fácil de leer. Dado que hemos provisto 12 valores e indicamos que queremos 3 filas, el argumento `ncol` no es necesario (es obvio que quedarán 4 columnas). Por eso, las siguientes sentencias son equivalentes a la anterior:

```
x <- matrix(c( 8, 13, 18, 23,
              11, 16, 21, 26,
              14, 19, 24, 29),
              nrow = 3, byrow = TRUE)

x <- matrix(c( 8, 13, 18, 23,
              11, 16, 21, 26,
              14, 19, 24, 29),
              ncol = 4, byrow = TRUE)
```

Si colocamos un único valor como primer argumento en la función `matrix()`, el mismo se repetirá en todas las posiciones. En este caso sí o sí tenemos que indicar cuántas filas y columnas deseamos:

```
y <- matrix(0, nrow = 2, ncol = 5)
y
```

```
[,1] [,2] [,3] [,4] [,5]
[1,]    0    0    0    0    0
[2,]    0    0    0    0    0
```

Una vez que la matriz ya existe, en el pseudocódigo haremos referencia al número de filas y columnas de la misma con las expresiones `NFILA(x)` y `NCOL(x)`. En R tenemos las siguientes funciones para analizar el tamaño de las matrices:

```
dim(x)
```

```
[1] 3 4
```

```
nrow(x)
```

```
[1] 3
```

```
ncol(x)
```

```
[1] 4
```

```
dim(y)
```

```
[1] 2 5
```

```
nrow(y)
```

```
[1] 2
```

```
ncol(y)
```

```
[1] 5
```

Podemos recorrer todas las posiciones de una matriz con una estructura iterativa doble: nos situamos en la primera fila y recorremos cada columna, luego en la segunda fila y recorremos todas las columnas y así sucesivamente:

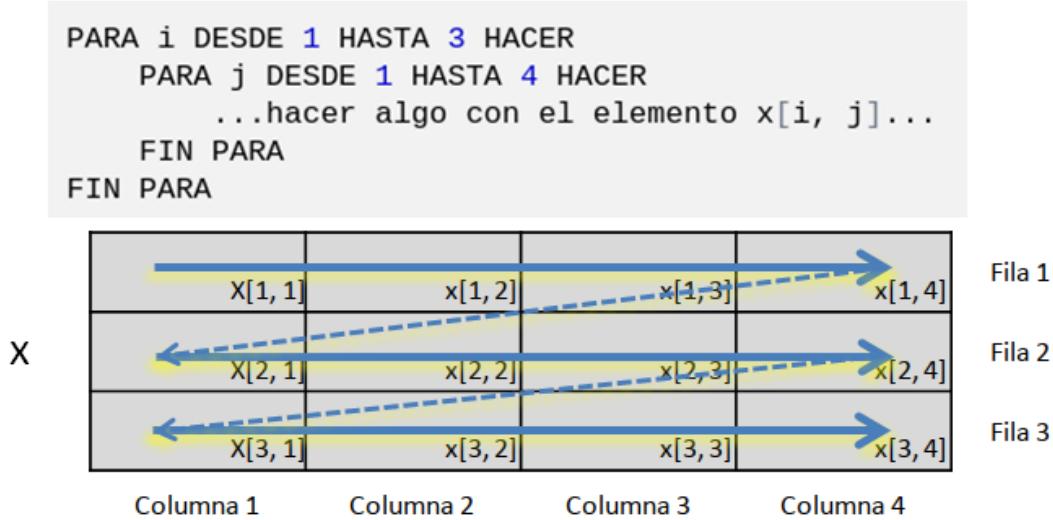


Figura 5.10: Recorrer una matriz por fila

También se puede recorrer la matriz por columna, si modificamos ligeramente las estructuras iterativas:

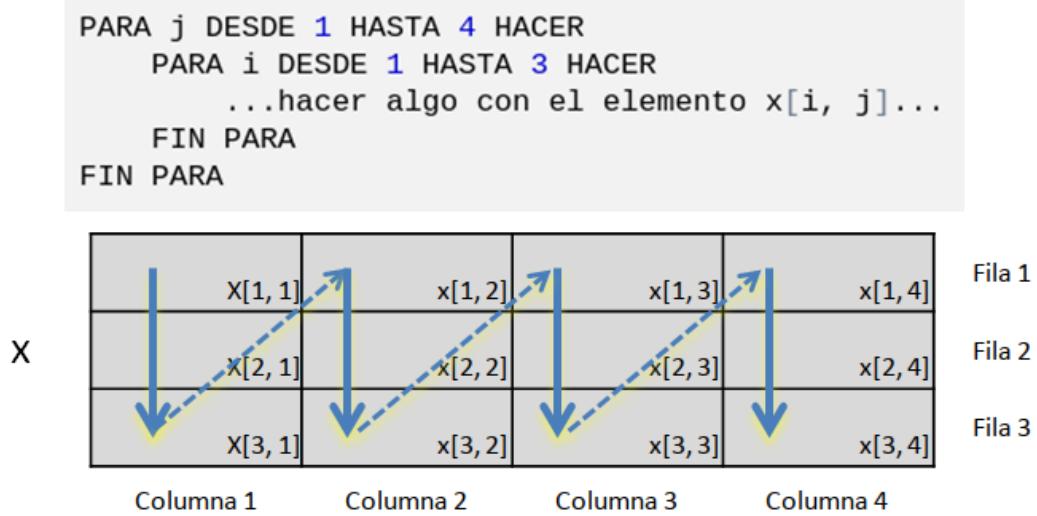


Figura 5.11: Recorrer una matriz por columna

Se puede usar cualquier letra o palabra como variables iteradoras, pero el uso de **i** para las filas y de **j** para las columnas es bastante común.

También podemos asignar valores en cada celda mientras recorremos la matriz. De hecho, la matriz **x** del ejemplo puede ser generada así, donde los índices **i** y **j** no sólo señalan una posición en particular dentro de la matriz, sino que además se usan para hacer el cálculo del valor a asignar:

```
VARIABLE numérica x(3, 4)
PARA i DESDE 1 HASTA NFILA(x) HACER
    PARA j DESDE 1 HASTA NCOL(x) HACER
        x[i, j] <- 3 * i + 5 * j
    FIN PARA
FIN PARA
MOSTRAR x
```

```
x <- matrix(0, nrow = 3, ncol = 4)
for (i in 1:nrow(x)) {
    for (j in 1:ncol(x)) {
        x[i, j] <- 3 * i + 5 * j
    }
}
x
```

```
[,1] [,2] [,3] [,4]
[1,] 8 13 18 23
[2,] 11 16 21 26
[3,] 14 19 24 29
```

Si queremos dejar que el valor en cada posición sea determinado por una fuente de información externa a la hora de correr el programa, empleamos la sentencia LEER en el pseudocódigo:

```
VARIABLE numérica x(3, 4)
PARA i DESDE 1 HASTA NFILA(x) HACER
    PARA j DESDE 1 HASTA NCOL(x) HACER
        LEER x[i, j]
    FIN PARA
FIN PARA
```

#### Ejemplo: trasponer una matriz

En Álgebra, trasponer una matriz de dimensión  $m \times n$  significa generar una nueva matriz de dimensión  $n \times m$ , donde los elementos se intercambian de este modo:

<b>Matriz x</b>	8	13	18	23
	11	16	21	26
	14	19	24	29

<b>Matriz traspuesta de x</b>	8	11	14
	13	16	19
	18	21	24
	23	26	29

Figura 5.12: Matriz traspuesta

Podemos formalizar el algoritmo que permite generar la matriz traspuesta, teniendo en cuenta que cada elemento que originalmente ocupa la posición  $[i, j]$  en la matriz original, debe pasar a ocupar la posición  $[j, i]$  en la matriz traspuesta:

ALGORITMO: Trasponer matriz  
COMENZAR

```
# Declarar objetos
VARIABLE numérica nf, nc
LEER nf, nc
VARIABLE numérica x(nf, nc), traspuesta(nc, nf)

# Leer los valores de la matriz
PARA i DESDE 1 HASTA nf HACER
    PARA j DESDE 1 HASTA nc HACER
        LEER x[i, j]
    FIN PARA
FIN PARA

# Trasponer
PARA i DESDE 1 HASTA nf HACER
    PARA j DESDE 1 HASTA nc HACER
        traspuesta[j, i] <- x[i, j]
    FIN PARA
FIN PARA

# Mostrar ambas matrices
ESCRIBIR "Matriz original"
MOSTRAR x
ESCRIBIR "Matriz traspuesta"
MOSTRAR traspuesta
```

FIN

Dado que en R vamos a asignar valores en la matriz de manera literal, primero la creamos y luego usamos `nrow()` y `ncol()` para obtener los correspondientes valores de `nf` y `nc`. En el siguiente ejemplo, además, todas las posiciones de la matriz traspuesta son iniciadas con el valor `NA`.

```
x <- matrix(c( 8, 13, 18, 23,
              11, 16, 21, 26,
              14, 19, 24, 29),
             nrow = 3, byrow = TRUE)
nf <- nrow(x)
nc <- ncol(x)
traspuesta <- matrix(NA, nc, nf)
for (i in 1:nf) {
  for (j in 1:nc) {
    traspuesta[j, i] <- x[i, j]
  }
}
cat("Matriz original\n")
x
cat("Matriz traspuesta\n")
traspuesta
```

```
Matriz original
 [,1] [,2] [,3] [,4]
[1,]    8   13   18   23
```

```
[2,]   11   16   21   26
[3,]   14   19   24   29
Matriz traspuesta
 [,1] [,2] [,3]
[1,]   8   11   14
[2,]  13   16   19
[3,]  18   21   24
[4,]  23   26   29
```

## 5.2. Características particulares de las estructuras de datos en R

Los vectores y matrices son estructuras que están bien representadas en casi cualquier lenguaje de programación. Por esta razón, ante diversos problemas computacionales podemos escribir algoritmos que empleando arreglos y operando con cada uno de sus elementos alcancen los objetivos propuestos.

No obstante, cada lenguaje de programación propone formas particulares de operar con los arreglos e incluso otros tipos de estructuras de datos. En esta sección nos dedicaremos a conocer cuáles son las herramientas que R nos ofrece para trabajar con vectores y matrices. Como son específicas de R, no hay convenciones generales para representarlas en pseudocódigo.

### 5.2.1. Elementos con nombre

Además de guardar información, los objetos de R pueden poseer ciertos **atributos**, que consisten en información adicional sobre el objeto. Uno de ellos es el atributo **names**, que permite que cada elemento dentro de un vector o una lista pueda tener su propio nombre, así como también que cada fila o columna de una matriz tenga su propio nombre, independientemente del identificador general del objeto.

#### Vectores

A cada elemento de un vector se le puede, opcionalmente, asignar un nombre. Esto se realiza de alguna de estas formas:

- Opción 1: después de crear el vector

```
# El vector se llama "frutas" y tiene 4 elementos
frutas <- c(3, 7, 2, 1)
frutas

[1] 3 7 2 1

# Cada uno de estos elementos no tienen nombres
names(frutas)

NULL

# Le doy un nombre a cada elemento
names(frutas) <- c("manzanas", "naranjas", "bananas", "peras")
frutas

manzanas naranjas bananas    peras
      3         7        2        1
```

- Opción 2: en el momento de crear el vector

```
frutas <- c(manzanas = 3, naranjas = 7, bananas = 2, peras = 1)
frutas

manzanas naranjas bananas    peras
      3         7        2        1
```

Los nombres son útiles porque permiten indexar al vector, sin necesidad de usar como índice la posición del elemento:

```
frutas[2]
```

```
naranjas
7
```

```
frutas["naranjas"]
```

```
naranjas
7
```

No todos los elementos de un vector deben tener nombre:

```
frutas <- c(manzanas = 3, 7, bananas = 2, 1)
frutas
```

manzanas	bananas
3	7
2	1

```
names(frutas)
```

```
[1] "manzanas" ""           "bananas"   "
```

## Matrices

En el caso de las matrices, se le puede asignar nombres a sus filas y columnas:

- Opción 1: después de crear la matriz

```
x <- matrix(c( 8, 13, 18, 23,
              11, 16, 21, 26,
              14, 19, 24, 29),
              nrow = 3, byrow = TRUE)
rownames(x) <- c("A", "B", "C")
colnames(x) <- c("col1", "grupo2", "grupo3", "grupo4")
x

col1 grupo2 grupo3 grupo4
A     8      13      18      23
B    11      16      21      26
C    14      19      24      29
```

- Opción 2: al crear la matriz

```
x <- matrix(c( 8, 13, 18, 23,
              11, 16, 21, 26,
              14, 19, 24, 29),
              nrow = 3, byrow = TRUE,
              dimnames = list(Categorias = c("A", "B", "C"),
                               Grupos = c("grupo1", "grupo2", "grupo3", "grupo4")))
x
```

```
Grupos
Categorias grupo1 grupo2 grupo3 grupo4
A     8      13      18      23
B    11      16      21      26
C    14      19      24      29
```

En este último ejemplo, se han elegido arbitrariamente los nombres **Categorías** y **Grupos** para llamar al conjunto completo de las filas y de las columnas, respectivamente. Esos nombres pueden ser cambiados por otros. Además, los nombres fueron encerrados en una *lista*, una estructura de datos que estudiaremos en breve.

Al igual que con los vectores, podemos usar los nombres de filas y columnas para indexar:

```
x["B", "grupo2"]
```

```
[1] 16
```

### 5.2.2. Operaciones vectorizadas

Con los conocimientos compartidos hasta aquí en esta unidad seremos capaces de escribir interesantes algoritmos para operar con vectores y matrices (por ejemplo: ordenar, buscar el mínimo, realizar cálculos algebraicos, etc.) y también de programarlos en R. En este proceso de aprendizaje, en la práctica de esta unidad vamos a encarar la tarea de escribir muchas funciones que, por lo general, ya forman parte de la sintaxis básica de cualquier lenguaje de programación. Sí... trabajaremos de más, ¡pero es para poder aprender! No obstante, ahora vamos a mencionar algunos ejemplos de funciones que ya están disponibles en R y que evitan que tengamos que trabajar tanto.

La mayoría de las funciones de R están **vectorizadas**. Esto quiere decir que están diseñadas para operar al mismo tiempo con todos los elementos de los vectores y matrices y no es necesario recorrer cada posición, una por una, como aprendimos para incorporar nuestros primeros conocimientos sobre algoritmos. Las funciones operan en todos los elementos sin tener que usar estructuras iterativas, haciendo que el código sea más conciso, fácil de leer y con menos chances de cometer errores.

#### Vectores

Por ejemplo, supongamos que queremos sumar dos vectores, como en el **\*ejercicio 2 de la práctica 4**. Gracias a que la suma en R está vectorizada, esto se logra haciendo sencillamente:

```
u <- c(5, 8, 2)
v <- c(2, 3, -1)
suma <- u + v
suma
```

```
[1] 7 11 1
```

Sin vectorización, deberíamos diseñar y programar un algoritmo como el siguiente:

```
suma <- numeric(length(u))
for (i in 1:length(u)) {
  suma[i] <- u[i] + v[i]
}
suma
```

```
[1] 7 11 1
```

Como podemos notar, al ejecutar `u + v` R realiza la suma elemento a elemento entre los dos vectores. Esto también sucede con los otros operadores aritméticos:

```
u - v
```

```
[1] 3 5 3
```

```
u * v
```

```
[1] 10 24 -2
```

```
u / v
```

```
[1] 2.500000 2.666667 -2.000000
```

```
u %% v
```

```
[1] 1 2 0
```

Estas operaciones también funcionan con vectores de distinto largo. En este caso, R aplica la **regla del reciclaje**: el vector más corto se recicla (se repiten sus elementos) hasta alcanzar la longitud del más largo y luego se opera elemento a elemento. Como es raro que queramos operar con dos vectores de distinto largo, R por las dudas nos tira una advertencia:

```
z <- c(1, 2)
u + z
```

Warning in u + z: longer object length is not a multiple of shorter object length

```
[1] 6 10 3
```

Si hacemos operaciones que involucran a una constante y a un vector, R repetirá tal operación con cada elemento del vector:

```
u + 5
```

```
[1] 10 13 7
```

```
1 / v
```

```
[1] 0.5000000 0.3333333 -1.0000000
```

```
10 * z
```

```
[1] 10 20
```

```
(u + v) / 100
```

```
[1] 0.07 0.11 0.01
```

Si le aplicamos funciones matemáticas como `log()` o `sqrt()` a un vector, obtendremos como resultado el valor de dicha función en cada uno de los elementos del vector:

```
log(u)
```

```
[1] 1.6094379 2.0794415 0.6931472
```

```
sqrt(z)
```

```
[1] 1.000000 1.414214
```

Hay funciones que cuando se aplican a un vector, logran resumirlo siguiendo algún criterio:

- Sumar todos los elementos de un vector:

```
sum(u)
```

```
[1] 15
```

- Multiplicar todos los elementos de un vector:

```
prod(u)
```

```
[1] 80
```

- Calcular el promedio de los elementos de un vector:

```
mean(u)
```

```
[1] 5
```

- Encontrar el valor mínimo y su ubicación en el vector (como en el **ejercicio 4 de la práctica 4**):

```
x <- c(40, 70, 20, 90, 20)
min(x)
```

```
[1] 20
```

```
which.min(x) # si el mínimo se repite, esta es la posición del primero
```

```
[1] 3
```

```
which(x == min(x)) # si el mínimo se repite, esto muestra todas sus posiciones
```

```
[1] 3 5
```

- Encontrar el valor máximo y su ubicación en el vector:

```
max(x)
```

```
[1] 90
```

```
which.max(x) # si el mínimo se repite, esta es la posición del primero
```

```
[1] 4
```

```
which(x == max(x)) # si el mínimo se repite, esto muestra todas sus posiciones
```

```
[1] 4
```

Combinando las ideas anteriores, podemos resolver de forma muy rápida ciertos problemas, como el de calcular el producto escalar entre dos vectores (**ejercicio 5 de la práctica 4**):

```
u
```

```
[1] 5 8 2
```

```
v
```

```
[1] 2 3 -1
```

```
sum(u * v)
```

```
[1] 32
```

En lo anterior, `u * v` hace la multiplicación elemento a elemento entre los vectores `u` y `v` y luego sumamos esos valores con `sum()`. Sin las operaciones vectorizadas, deberíamos hacer algo como lo siguiente:

```
rtdo <- 0
for (i in 1:length(u)) {
  rtdo <- rtdo + u[i] * v[i]
}
```

```
[1] 32
```

En el **ejercicio 3** de la **Práctica 4** creamos las funciones `ordenar_asc()` y `ordenar_des()` para ordenar los elementos de un vector. Con las funciones disponibles en R, esto se puede hacer así:

```
x
```

```
[1] 40 70 20 90 20
```

```
sort(x)
```

```
[1] 20 20 40 70 90
```

```
sort(x, decreasing = TRUE)
```

```
[1] 90 70 40 20 20
```

## Matrices

Los casos anteriores tienen sus equivalentes cuando operamos con matrices. Por ejemplo, en el **ejercicio 7 de la práctica 4** programamos una función para hacer la suma entre dos matrices. Sin vectorización, esto involucra pasos como los siguientes:

```
a <- matrix(c(5, 8, 2, 2, 3, 1), nrow = 3)
b <- matrix(c(0, -1, 3, 1, 2, 4), nrow = 3)
a
```

```
[,1] [,2]
[1,]    5    2
[2,]    8    3
[3,]    2    1
```

```
b
```

```
[,1] [,2]
[1,]    0    1
[2,]   -1    2
[3,]    3    4
```

```
suma <- matrix(NA, nrow(a), ncol(a))
for (i in 1:nrow(a)) {
  for (j in 1:ncol(a)) {
    suma[i, j] <- a[i, j] + b[i, j]
  }
}
suma
```

```
[,1] [,2]
[1,]    5    3
[2,]    7    5
[3,]    5    5
```

Gracias a las operaciones vectorizadas de R, esto se puede resumir en:

```
a + b
```

```
[,1] [,2]
[1,]    5    3
[2,]    7    5
[3,]    5    5
```

A continuación, otros ejemplos de operaciones realizadas elemento a elemento con matrices:

```
a + b
```

```
[,1] [,2]
[1,]    5    3
[2,]    7    5
[3,]    5    5
```

```
a - b
```

```
[,1] [,2]
[1,]    5    1
[2,]    9    1
[3,]   -1   -3
```

```
a * b
```

```
[,1] [,2]
[1,]    0    2
[2,]   -8    6
[3,]    6    4
```

```
a / b
```

```
[,1] [,2]  
[1,]      Inf 2.00  
[2,] -8.0000000 1.50  
[3,] 0.6666667 0.25
```

```
a^2
```

```
[,1] [,2]  
[1,] 25     4  
[2,] 64     9  
[3,] 4      1
```

```
sqrt(a)
```

```
[,1]      [,2]  
[1,] 2.236068 1.414214  
[2,] 2.828427 1.732051  
[3,] 1.414214 1.000000
```

También podemos resumir la información contenida en una matriz:

- Suma de todos los elementos:

```
a
```

```
[,1] [,2]  
[1,] 5     2  
[2,] 8     3  
[3,] 2     1
```

```
sum(a)
```

```
[1] 21
```

- Promedio de todos los elementos:

```
mean(a)
```

```
[1] 3.5
```

- Suma de los elementos por fila:

```
rowSums(a)
```

```
[1] 7 11 3
```

- Suma de los elementos por columna:

```
colSums(a)
```

```
[1] 15 6
```

- Promedio de los elementos por fila:

```
rowMeans(a)
```

```
[1] 3.5 5.5 1.5
```

- Promedio de los elementos por columna:

```
colMeans(a)
```

```
[1] 5 2
```

- Búsqueda de mínimos y máximos en una matriz:

```
d <- matrix(sample(100, 20), nrow = 5)

# Valor máximo
max(d)

[1] 81

# Posición (arr.ind = TRUE para que nos indique fila y columna)
which(d == max(d), arr.ind = TRUE)

      row col
[1,]   2   3

# Valor mínimo
min(d)

[1] 5

# Posición
which(d == min(d), arr.ind = TRUE)

      row col
[1,]   4   4
```

Como aprenderán en Álgebra, las matrices numéricas son muy útiles en diversos campos y por eso existen distintas operaciones que se pueden realizar con las mismas. Veamos algunos ejemplos de la aplicación del álgebra matricial en R:

- Transpuesta de una matriz:

```
a

[,1] [,2]
[1,]    5    2
[2,]    8    3
[3,]    2    1

t(a)

[,1] [,2] [,3]
[1,]    5    8    2
[2,]    2    3    1
```

- Producto entre dos matrices:

```
e <- matrix(1:4, nrow = 2)

a

[,1] [,2]
[1,]    5    2
[2,]    8    3
[3,]    2    1

e

[,1] [,2]
[1,]    1    3
[2,]    2    4

a %*% e

[,1] [,2]
[1,]    9   23
```

```
[2,]   14   36
[3,]    4   10
```

- Inversa de la matriz:

```
solve(e)
```

```
[,1] [,2]
[1,] -2  1.5
[2,]  1 -0.5
```

- Obtener los elementos de la diagonal principal:

```
diag(d)
```

```
[1] 6 80 14 5
```

### 5.2.3. Operaciones lógicas vectorizadas

Cuando dos vectores o matrices se vinculan a través de una comparación, se opera elemento a elemento obteniendo un vector o matriz de valores lógicos:

```
x <- c(40, 70, 20, 90, 20)
```

```
y <- c(10, 70, 30, 15, 21)
```

```
x > y
```

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

```
x < y * 5
```

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

```
a <- matrix(c(5, 8, 2, 2, 3, 1), nrow = 3)
```

```
b <- matrix(c(0, -1, 3, 1, 2, 4), nrow = 3)
```

```
a
```

```
[,1] [,2]
[1,]    5    2
[2,]    8    3
[3,]    2    1
```

```
b
```

```
[,1] [,2]
[1,]    0    1
[2,]   -1    2
[3,]    3    4
```

```
a != b
```

```
[,1] [,2]
[1,] TRUE TRUE
[2,] TRUE TRUE
[3,] TRUE TRUE
```

```
a > b
```

```
[,1] [,2]
[1,] TRUE TRUE
[2,] TRUE TRUE
[3,] FALSE FALSE
```

Si un vector o matriz de valores lógicos y queremos saber si todos o al menos uno de los elementos es igual a TRUE,

podemos usar las funciones `all()` y `any()`, respectivamente:

```
all(a != b)
```

```
[1] TRUE
```

```
any(a != b)
```

```
[1] TRUE
```

```
all(a > b)
```

```
[1] FALSE
```

```
any(a > b)
```

```
[1] TRUE
```

Las operaciones de comparación pueden hacerse entre cada elemento de un vector o matriz y un único valor:

```
x < 50
```

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

```
a == 3
```

```
[,1] [,2]
[1,] FALSE FALSE
[2,] FALSE TRUE
[3,] FALSE FALSE
```

```
b > 0
```

```
[,1] [,2]
[1,] FALSE TRUE
[2,] FALSE TRUE
[3,] TRUE TRUE
```

Los operadores lógicos que se utilizan para realizar cálculos elemento a elemento con vectores y matrices son `&`, `\` y `!`. Ellos nos permiten crear expresiones aún más complejas:

```
x < 50 & y > 50
```

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

```
a < 0 | b > 0
```

```
[,1] [,2]
[1,] FALSE TRUE
[2,] FALSE TRUE
[3,] TRUE TRUE
```

```
!(x <= 50)
```

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

#### 5.2.4. Uso de vectores para indexar vectores y matrices

Como ya sabemos, indexar es hacer referencia a uno o más elementos particulares dentro de una estructura de datos. Vimos que para indexar a un vector, hace falta sólo un índice:

```
x <- c(10.4, 5.6, 3.1, 6.4, 21.7)
```

```
x[3]
```

[1] 3.1

Y que para indexar matrices, son necesarios dos índices:

```
a <- matrix(c(4,-2, 1, 20, -7, 12, -8, 13, 17), nrow = 3)
a
```

```
[,1] [,2] [,3]
[1,]    4   20   -8
[2,]   -2   -7   13
[3,]    1   12   17
```

```
a[2, 3]
```

[1] 13

Pero también podemos indexar a múltiples elementos de un vector o una matriz a la vez:

### Vectores

```
# Mostrar los primeros tres elementos del vector x
x[1:3]
```

[1] 10.4 5.6 3.1

```
# Mostrar los elementos en las posiciones 2 y 4
x[c(2, 4)]
```

[1] 5.6 6.4

```
# Mostrar el último elemento
x[length(x)]
```

[1] 21.7

```
# Indexar con valores lógicos. Obtenemos sólo las posiciones indicadas con TRUE:
x[c(F, F, T, T, F)]
```

[1] 3.1 6.4

```
# Sabiendo que la siguiente operación devuelve TRUE o FALSE para cada posición de x:
x > 10
```

[1] TRUE FALSE FALSE FALSE TRUE

```
# ...la podemos usar para quedarnos con aquellos elementos de x mayores a 10:
x[x > 10]
```

[1] 10.4 21.7

```
#Mostrar todos los elementos menos el cuarto
x[-4]
```

[1] 10.4 5.6 3.1 21.7

### Matrices

```
# Toda la fila 3
a[3, ]
```

[1] 1 12 17

```
# Toda la columna 2
a[, 2]
```

```
[1] 20 -7 12
```

```
# Submatriz con las columnas 1 y 2
a[, 1:2]
```

```
[,1] [,2]
[1,]    4    20
[2,]   -2   -7
[3,]    1    12
```

```
# Submatriz con las columnas 1 y 3
a[, c(1, 3)]
```

```
[,1] [,2]
[1,]    4   -8
[2,]   -2   13
[3,]    1   17
```

```
# Asignar el mismo valor en toda la fila 3
```

```
a[3, ] <- 10
a
```

```
[,1] [,2] [,3]
[1,]    4    20   -8
[2,]   -2   -7   13
[3,]   10   10   10
```

### 5.2.5. La función apply()

Supongamos que queremos encontrar el máximo valor en cada fila de una matriz. Podemos lograrlo de la siguiente forma. Creamos un vector `maximos` con lugar para guardar el máximo de cada fila. Luego, iteramos para recorrer cada fila de la matriz, buscar el mínimo y guardarlo en el vector `maximos`:

```
a
```

```
[,1] [,2] [,3]
[1,]    4    20   -8
[2,]   -2   -7   13
[3,]   10   10   10
```

```
maximos <- numeric(nrow(a))
for (i in 1:nrow(a)) {
  maximos[i] <- max(a[i, ])
}
maximos
```

```
[1] 20 13 10
```

En R existe una forma más práctica y eficiente de conseguir el mismo resultado:

```
apply(a, 1, max)
```

```
[1] 20 13 10
```

La función `apply()` sirve para aplicar una misma operación a cada fila o columna de una matriz. En el ejemplo anterior:

- el primer argumento, `a`, es la matriz a analizar.
- el segundo argumento, `1`, indica que la operación se realizará fila por fila (para que se haga por columna, debemos indicar `2`)
- el tercer argumento, `max`, es el nombre de la función que se le aplica a cada fila.

De manera similar, podemos encontrar el mínimo valor de cada columna:

```
apply(a, 2, min)
```

```
[1] -2 -7 -8
```

### 5.2.6. Generación de vectores con secuencias numéricas

A continuación mostramos cómo generar algunos vectores numéricos en R:

```
# Enteros de 1 a 5
1:5
```

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

```
# Números de 1 a 10 cada 2
seq(1, 10, 2)
```

```
[1] 1 3 5 7 9
```

```
# Números de 0 a -1 cada -0.1
seq(0, -1, -0.1)
```

```
[1] 0.0 -0.1 -0.2 -0.3 -0.4 -0.5 -0.6 -0.7 -0.8 -0.9 -1.0
```

```
# Siete números equiespaciados entre 0 y 1
seq(0, 1, length.out = 7)
```

```
[1] 0.0000000 0.1666667 0.3333333 0.5000000 0.6666667 0.8333333 1.0000000
```

```
# Repetir el 1 tres veces
rep(1, 3)
```

```
[1] 1 1 1
```

```
# Repetir (1, 2, 3) tres veces
rep(1:3, 3)
```

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

```
# Repetir cada número tres veces
rep(1:3, each = 3)
```

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

```
# Generar una matriz diagonal
diag(c(3, 7, 1, 5))
```

```
[,1] [,2] [,3] [,4]
[1,]    3    0    0    0
[2,]    0    7    0    0
[3,]    0    0    1    0
[4,]    0    0    0    5
```

```
# Generar una matriz identidad
diag(rep(1, 5))
```

```
[,1] [,2] [,3] [,4] [,5]
[1,]    1    0    0    0    0
[2,]    0    1    0    0    0
[3,]    0    0    1    0    0
[4,]    0    0    0    1    0
[5,]    0    0    0    0    1
```

### 5.2.7. Concatenación de vectores y matrices

Los vectores pueden combinarse entre sí para crear nuevos vectores con `c()`:

```
x <- 1:5
y <- c(10, 90, 87)
z <- c(x, y, x)
z
```

[1] 1 2 3 4 5 10 90 87 1 2 3 4 5

Matrices que tienen la misma cantidad de filas pueden concatenarse una al lado de la otra con `cbind()`:

```
a <- matrix(c(5, 8, 2, 2, 3, 1), nrow = 3)
b <- matrix(c(0, -1, 3, 1, 2, 4), nrow = 3)
a
```

```
[,1] [,2]
[1,] 5 2
[2,] 8 3
[3,] 2 1
```

b

```
[,1] [,2]
[1,] 0 1
[2,] -1 2
[3,] 3 4
```

```
d <- cbind(a, b)
d
```

```
[,1] [,2] [,3] [,4]
[1,] 5 2 0 1
[2,] 8 3 -1 2
[3,] 2 1 3 4
```

Matrices que tienen la misma cantidad de columnas pueden concatenarse una debajo de la otra con `rbind()`:

```
e <- rbind(a, b)
e
```

```
[,1] [,2]
[1,] 5 2
[2,] 8 3
[3,] 2 1
[4,] 0 1
[5,] -1 2
[6,] 3 4
```

Estas funciones también permiten unir matrices con vectores:

```
x <- 1:6
cbind(e, x)
```

```
x
[1,] 5 2 1
[2,] 8 3 2
[3,] 2 1 3
[4,] 0 1 4
[5,] -1 2 5
[6,] 3 4 6
```

### 5.2.8. Listas

Una de las principales características de los **arreglos** es la **homogeneidad**: todos los elementos que contienen deben ser del mismo tipo. No se puede, por ejemplo, mezclar en una matriz valores numéricos y lógicos. Sin embargo, en muchos problemas resulta útil contar con alguna estructura de datos que permita agrupar objetos de diversos tipos. Esa es, justamente, la definición de una **lista**. Podemos imaginarla como una bolsa en la cual podemos meter todo tipo de objetos, incluyendo vectores, matrices y, por qué no, otras bolsas (es decir, bolsas dentro de una bolsa o listas dentro de una lista). Todos los lenguajes de programación proveen algún tipo de estructura con estas características, aunque no todos las llaman igual. Otros posibles nombres con los que se conocen pueden ser *tupla* o *agregado*. En R se llaman **listas** o **vectores recursivos**. El siguiente diagrama presenta una lista (recuadro con puntas redondeadas) que contiene:

1. Un vector numérico de largo 3.
2. Un vector carácter de largo 2.
3. Una matriz numérica de dimensión 2x2.
4. Un valor lógico.



Figura 5.13: Ejemplo de una lista

La creación de esta lista se realiza mediante la función `list()`, cuyos argumentos son los elementos que queremos guardar en la lista, separados por comas:

```
mi_lista <- list(
  c(-4.5, 12, 2.71),
  c("hola", "chau"),
  matrix(c(8, 11, 13, 16), nrow = 2),
  TRUE
)
mi_lista

[[1]]
[1] -4.50 12.00 2.71

[[2]]
[1] "hola" "chau"

[[3]]
[1] [,1] [,2]
```

```
[1,]    8   13
[2,]   11   16
```

```
[[4]]
[1] TRUE
```

Luego de correr la sentencia anterior, podemos ver que `mi_lista` es un nuevo objeto disponible en el ambiente global y como tal está listado en el panel *Environment*. Allí se nos indica que se trata de una lista y, además, podemos previsualizar su contenido al hacer clic en el círculo celeste que antecede a su nombre:

The screenshot shows the RStudio interface with the 'Environment' tab selected. In the 'Global Environment' list, there is an entry for 'mi\_lista' which is highlighted with a blue circle. The details for 'mi\_lista' are shown in a table:

	mi_lista	List of 4
\$ : num	[1:3]	-4.5 12 2.71
\$ : chr	[1:2]	"hola" "chau"
\$ : num	[1:2, 1:2]	8 11 13 16
\$ : logi		TRUE

Figura 5.14: La lista en la pestaña Environment de RStudio

Usamos dobles corchetes `[[ ]]` para referenciar a cada objeto que forma parte de la lista. Además, si queremos indicar un elemento dentro de un objeto que forma parte de la lista, agregamos otro conjunto de corchetes como hacemos con vectores y matrices. Por ejemplo:

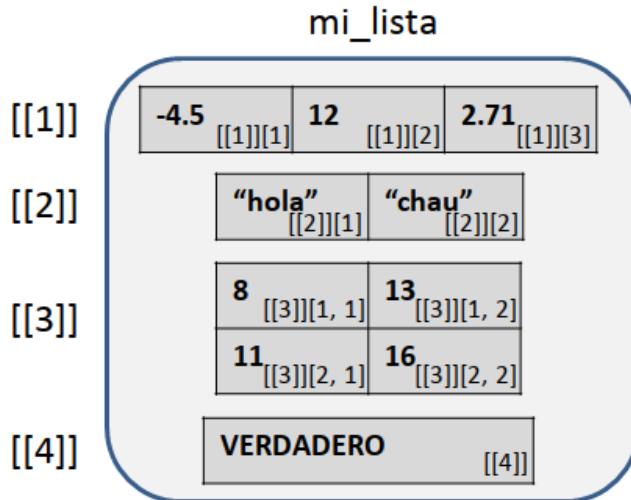


Figura 5.15: Ejemplo de una lista

```
mi_lista[[1]]
[1] -4.50 12.00 2.71

mi_lista[[1]][3]
[1] 2.71

mi_lista[[2]]
[1] "hola" "chau"
```

```
mi_lista[[2]][2]
```

```
[1] "chau"
```

```
mi_lista[[3]]
```

```
[,1] [,2]
```

```
[1,] 8 13
```

```
[2,] 11 16
```

```
mi_lista[[3]][2, 1]
```

```
[1] 11
```

```
mi_lista[[4]]
```

```
[1] TRUE
```

```
mi_lista[[4]][1]
```

```
[1] TRUE
```

Podemos asignar valor a algún elemento usando los índices de esa misma forma:

```
mi_lista[[1]][3] <- 0
```

```
mi_lista
```

```
[[1]]
```

```
[1] -4.5 12.0 0.0
```

```
[[2]]
```

```
[1] "hola" "chau"
```

```
[[3]]
```

```
[,1] [,2]
```

```
[1,] 8 13
```

```
[2,] 11 16
```

```
[[4]]
```

```
[1] TRUE
```

Cada uno de los elementos de una lista puede tener un nombre propio. Podemos asignarle un nombre a uno, algunos o todos los integrantes en una lista:

```
mi_lista <- list(
  w = c(-4.5, 12, 2.71),
  x = c("hola", "chau"),
  y = matrix(c(8, 11, 13, 16), nrow = 2),
  z = TRUE
)
mi_lista
```

```
$w
```

```
[1] -4.50 12.00 2.71
```

```
$x
```

```
[1] "hola" "chau"
```

```
$y
```

```
[,1] [,2]
```

```
[1,]    8   13
[2,]   11   16
```

```
$z
[1] TRUE
```

Esto amplía las opciones para hacer referencia a cada objeto y elemento allí contenido. Las siguientes sentencias son todas equivalentes y sirven para acceder al tercer elemento de la lista, cuyo nombre es y:

```
mi_lista[[3]]
```

```
[,1] [,2]
[1,]    8   13
[2,]   11   16
```

```
mi_lista[["y"]]
```

```
[,1] [,2]
[1,]    8   13
[2,]   11   16
```

```
mi_lista$y
```

```
[,1] [,2]
[1,]    8   13
[2,]   11   16
```

Finalmente, consideremos la situación en la cual queremos aplicarle la misma función a cada uno de los elementos que integran una lista. Para esto podemos usar `lapply()` o `sapply()`, parientes de la función `apply()` que vimos antes. Por ejemplo, tenemos una lista con varios vectores y queremos saber el largo de cada uno de ellos:

```
mi_lista <- list(x = c(1, 8, 9, -1), y = c("uno", "dos", "tres"), z = c(3, 2))
mi_lista
```

```
$x
[1] 1 8 9 -1
```

```
$y
[1] "uno"  "dos"  "tres"
```

```
$z
[1] 3 2
```

Podemos ver el largo de cada elemento de la lista, uno por uno:

```
length(mi_lista$x)
```

```
[1] 4
```

```
length(mi_lista$y)
```

```
[1] 3
```

```
length(mi_lista$z)
```

```
[1] 2
```

O podemos hacerlo así:

```
lapply(mi_lista, length)
```

```
$x
```

```
[1] 4
```

```
$y  
[1] 3
```

```
$z  
[1] 2
```

```
sapply(mi_lista, length)
```

```
x y z  
4 3 2
```

Ambas funciones le aplican la función elegida como segundo argumento (`length()`) a cada elemento de la lista indicada en el primer argumento (`mi_lista`). La diferencia entre ellas es que `lapply()` devuelve una nueva lista con los resultados, mientras que `sapply()` los devuelve acomodados en un vector<sup>2</sup>.

#### Ejemplo: función que devuelve una lista

En el capítulo anterior, dijimos que las funciones son subalgoritmos que podían devolver exactamente un objeto como resultado. Esto puede ser una limitación, ya que en algunos casos tal vez necesitemos devolver varios elementos de distinto tipo<sup>3</sup>. La solución consiste en devolver una lista que englobe a todos los objetos que nos interese que la función le entregue como resultado al algoritmo principal que la invocó. Como una lista es un único objeto, ¡la función puede devolverla sin ningún problema!

Para ejemplificar, recordemos el siguiente ejercicio de la práctica 3: escribir un programa para la creación de la función `triangulos(a, b, c)` que a partir de la longitud de los tres lados de un triángulo `a`, `b` y `d` (valores positivos) lo clasifica con los siguientes resultados posibles: no forman un triángulo (un lado mayor que la suma de los otros dos), triángulo equilátero, isósceles o escaleno. Vamos a modificar la función para que tenga el siguiente comportamiento: la función debe devolver el tipo de triángulo como cadena de texto y el valor numérico del perímetro del mismo (o un 0 si no es triángulo). Es decir, la función debe devolver tanto un objeto de tipo carácter y otro de tipo numérico. Para lograrlo los encerraremos en una lista:

```
#-----  
# Función triangulos  
# Clasifica un triángulo según la longitud de sus lados  
# Entrada:  
#      - a, b, d, números reales positivos  
# Salida:  
#      - una lista cuyo primer elemento es un carácter que indica el tipo de  
#        triángulo y cuyo segundo elemento es el perímetro del triángulo o el valor 0  
#        si los lados provistos no corresponden a un triángulo  
#-----  
triangulos <- function(a, b, d) {  
  perímetro <- a + b + d  
  if (a > b + d || b > a + d || d > a + b) {  
    tipo <- "no es triángulo"  
    perímetro <- 0  
  } else if (a == b & a == d) {  
    tipo <- "equilátero"  
  } else if (a == b || a == d || b == d) {  
    tipo <- "isósceles"  
  } else {  
    tipo <- "escaleno"  
  }  
  return(list(tipo = tipo, perímetro = perímetro))
```

<sup>2</sup>Los ejemplos presentados aquí son muy generales. Dependiendo del tipo de estructura de datos a considerar y de la función a aplicar, la forma en que se presentan los resultados al aplicar alguna de estas funciones puede variar, pero por ahora eso no importa.

<sup>3</sup>Si fuesen elementos del mismo tipo, los podríamos devolver dentro de un vector, por ejemplo, las dos soluciones reales distintas de una ecuación cuadrática.

```
# atención con tipo = tipo: la primera vez es el nombre que le estamos dando
# al elemento de la lista, la segunda vez es la variable que guardamos en la lista
}
```

Ejemplos del uso de esta función:

```
# Guardamos el resultado devuelto (una lista) en el objeto resultado
resultado <- triangulos(2, 3, 4)
# Miramos el primer elemento de la lista (carácter que indica el tipo)
```

```
resultado[[1]]
```

```
[1] "escaleno"
```

```
resultado$tipo
```

```
[1] "escaleno"
```

```
# Miramos el primer elemento de la lista (perímetro)
```

```
resultado[[2]]
```

```
[1] 9
```

```
resultado$perimetro
```

```
[1] 9
```

```
# Miramos todo junto
```

```
resultado
```

```
$tipo
```

```
[1] "escaleno"
```

```
$perimetro
```

```
[1] 9
```

```
# Otro ejemplo:
```

```
resultado2 <- triangulos(2, 3, 10)
resultado2[[1]]
```

```
[1] "no es triángulo"
```

```
resultado2[[2]]
```

```
[1] 0
```

### 5.3. Otras consideraciones (lectura opcional)

Dependiendo de cómo se almacenan los datos que componen a una estructura en el *hardware* de la computadora, las mismas se pueden clasificar en **contiguas** o **enlazadas**. En las estructuras contiguas, los datos se sitúan en áreas adyacentes de memoria y cada uno de ellos se puede localizar partiendo de la posición en memoria del primer elemento de la estructura. En las estructuras enlazadas, los datos no se sitúan necesariamente de forma continua en la memoria sino que existen *punteros* (otro tipo de dato que *apunta* hacia determinada posición de memoria) que permite identificar cuál es el orden de los elementos dentro de la estructura.

Por otro lado, las estructuras también se pueden clasificar en **dinámicas** o **estáticas**, según si su tamaño puede cambiar o no durante la ejecución de un programa, respectivamente. Cuando se emplea una estructura estática, se declara con anterioridad el tamaño que ocupará en memoria y su dimensión no podrá variar, aún cuando no se ocupen todas las posiciones reservadas. En contraposición, una estructura dinámica puede cambiar en tamaño.

Antes se mencionó que las listas de R pueden contener objetos de distintos tipos. Para ser más rigurosos, una lista es un tipo especial de vector que agrupa punteros hacia distintos objetos. Técnicamente, todos los elementos de una

lista son del mismo tipo (punteros), pero hacen referencia a distintos objetos, dándonos la impresión de que en una lista de R, sencillamente, podemos meter cualquier tipo de objeto.

## 5.4. Arreglos multidimensionales (lectura opcional)

Un **arreglo multidimensional** contiene más de dos dimensiones, es decir, requiere más de dos índices para identificar a cada uno de sus elementos. La representación matemática o visual ya no es tan sencilla como la de los vectores o matrices. Para interpretarlos o saber cuándo usarlos, pensamos que cada una de las dimensiones representa una característica de los elementos.

Por ejemplo, imaginemos que en un local comercial se quiere registrar cuántos clientes se atendieron en cada una de las tres cajas disponibles (primer dimensión del arreglo: caja 1, caja 2 o caja 3), ya sea en el turno mañana o tarde (segunda dimensión: 1 para la mañana o 2 para la tarde) en cada día hábil de una semana (tercera dimensión: 1 lunes, 2 martes, 3 miércoles, 4 jueves o 5 viernes). Si queremos registrar, por ejemplo, que la caja 1 en el turno tarde del día jueves atendió 12 clientes, tenemos que guardar el valor 12 en la posición [1, 2, 4] del arreglo.

El arreglo de 3 dimensiones que permite acomodar toda la información del ejemplo en una sola estructura puede definirse en R así:

```
clientes <- array(0, dim = c(3, 2, 5))
clientes

, , 1

[,1] [,2]
[1,]    0    0
[2,]    0    0
[3,]    0    0

, , 2

[,1] [,2]
[1,]    0    0
[2,]    0    0
[3,]    0    0

, , 3

[,1] [,2]
[1,]    0    0
[2,]    0    0
[3,]    0    0

, , 4

[,1] [,2]
[1,]    0    0
[2,]    0    0
[3,]    0    0

, , 5

[,1] [,2]
[1,]    0    0
[2,]    0    0
[3,]    0    0
```

Luego, si queremos registrar que la caja 1 en el turno tarde del día jueves atendió 12 clientes, hacemos:

```
clientes[1, 2, 4] <- 12
```

```
clientes
```

```
, , 1
```

```
[,1] [,2]
[1,] 0 0
[2,] 0 0
[3,] 0 0
```

```
, , 2
```

```
[,1] [,2]
[1,] 0 0
[2,] 0 0
[3,] 0 0
```

```
, , 3
```

```
[,1] [,2]
[1,] 0 0
[2,] 0 0
[3,] 0 0
```

```
, , 4
```

```
[,1] [,2]
[1,] 0 12
[2,] 0 0
[3,] 0 0
```

```
, , 5
```

```
[,1] [,2]
[1,] 0 0
[2,] 0 0
[3,] 0 0
```

En R, podemos ponerle un nombre a cada una de las dimensiones del arreglo (“caja”, “turno”, “dia”), para poder identificar mejor a qué aspecto hace referencia y, a su vez, un nombre a cada una de sus modalidades (por ejemplo, “caja 1”, “caja 2” o “caja 3”). Esto se logra de la siguiente forma:

```
dimnames(clientes) <- list(caja = c("caja 1", "caja 2", "caja 3"),
                             turno = c("mañana", "tarde"),
                             dia = c("lun", "mar", "mie", "jue", "vie"))
```

```
clientes
```

```
, , dia = lun
```

	turno	
caja	mañana	tarde
caja 1	0	0
caja 2	0	0
caja 3	0	0

```
, , dia = mar
```

```
turno
```

```
caja    mañana tarde
caja 1      0      0
caja 2      0      0
caja 3      0      0
```

, , dia = mie

```
turno
caja    mañana tarde
caja 1      0      0
caja 2      0      0
caja 3      0      0
```

, , dia = jue

```
turno
caja    mañana tarde
caja 1      0      12
caja 2      0      0
caja 3      0      0
```

, , dia = vie

```
turno
caja    mañana tarde
caja 1      0      0
caja 2      0      0
caja 3      0      0
```

Para situaciones como la anterior, resulta más útil guardar los valores en otros tipos de estructuras especializadas en conjuntos de datos, como veremos más adelante. Por esta razón, en esta asignatura no profundizaremos en el estudio de arreglos multidimensionales y nos bastaremos con vectores y matrices. Sin embargo, debemos saber de su existencia porque son estructuras útiles para ciertas tareas de programación.

## Unidad 6

# Uso de archivos de datos

Página en revisión



# Unidad 7

## Otros tópicos

Las reglas que aprendimos para escribir el pseudocódigo nos permiten traducir nuestros algoritmos a cualquier lenguaje de computación de manera muy general y sencilla. Sin embargo, cada lenguaje de programación tiene diseñado su propio conjunto de funciones y estructuras de datos que facilitan algunas tareas. Una vez que hemos incorporado los conceptos básicos de la programación, podemos dedicarnos a aprender las profundidades de un lenguaje en particular.

En este capítulo vamos mencionar algunas cosas útiles sobre R y otras cuestiones.

### 7.1. La consola

Cuando prendemos nuestra computadora nos encontramos con una interfaz gráfica implementada por el sistema operativo para que podamos hacer lo que necesitemos de manera sencilla usando ventanas y menús, interactuando con el sistema a través del mouse, teclado, micrófono o pantalla táctil. Sin embargo, es posible usar la compu de otra forma, escribiendo comandos especiales en una ventanita, llamada **consola** que es capaz de interpretarlos para hacer cualquier tipo de actividad, sin utilizar la interfaz gráfica. Años atrás, esta era la única manera disponible de usar la computadora (por ejemplo, con sistema operativo MS-DOS de Windows).

Saber usar la consola es muy útil para automatizar actividades, realizar tareas administrativas, manipular varios archivos u objetos en simultáneo, lanzar a correr proyectos grandes, conectarse de manera remota a un servidor, etc. La primera vez que uno se mete en esto puede ser intimidante, pero no hay que olvidar que ya estamos acostumbrados a usar la consola de R, con lo cual esto de escribir comandos e interpretar respuestas es algo familiar.

Los términos *terminal*, *consola*, *shell* y *línea de comandos* son términos que a veces se usan como sinónimos sin demasiada preocupación, como si fuesen distintas formas de llamar a lo mismo: una ventanita donde puedo escribir comandos y hacer que sucedan cosas en la computadora. Sin embargo, hay pequeñas diferencias entre estos conceptos, que acá tratamos de resumir (aunque ni siquiera entre informáticos hay mucho acuerdo en las definiciones):

- **Shell** (intérprete de línea de comandos): es software, es un programa que corre otros programas, procesa los comandos que recibe y devuelve resultados Ejemplo: Bash (la más común en sistemas Linux), sh, PowerShell, etc.
- **Terminal**: un programa que propicia la transferencia de input/output entre el usuario y la shell. No ejecuta comandos, pero recibe los comandos que el shell va a procesar. Ejemplos: Command prompt, guake, gnome-terminal, etc. Antiguamente, se trataba de las múltiples estaciones con un monitor y un teclado que una gran computadora tenía para ser usada por varias personas.
- **Consola**: un tipo particular de terminal, con una ventana escribir inputs y leer outputs. Históricamente, se trataba de un panel físico (consola) con controles. Siri o Cortana podrían considerarse terminales, pero no son consolas, ya que no hay que escribir para mandar comandos. Terminal y consola se usan prácticamente como sinónimos.

Para abrir una terminal en Linux se puede usar el atajo **ctrl + alt + t** y en Windows se puede escribir **cmd** en Inicio.

Así luce la línea de comandos de Windows:

Los comandos se escriben en la última línea a continuación del símbolo **>**, que a su vez está precedido por la ruta a la carpeta que es el directorio de trabajo actual de la terminal, en este caso, **C:\Users\Marcos** (carpeta **Marcos**, dentro de la carpeta **Users**, en el disco **C**).

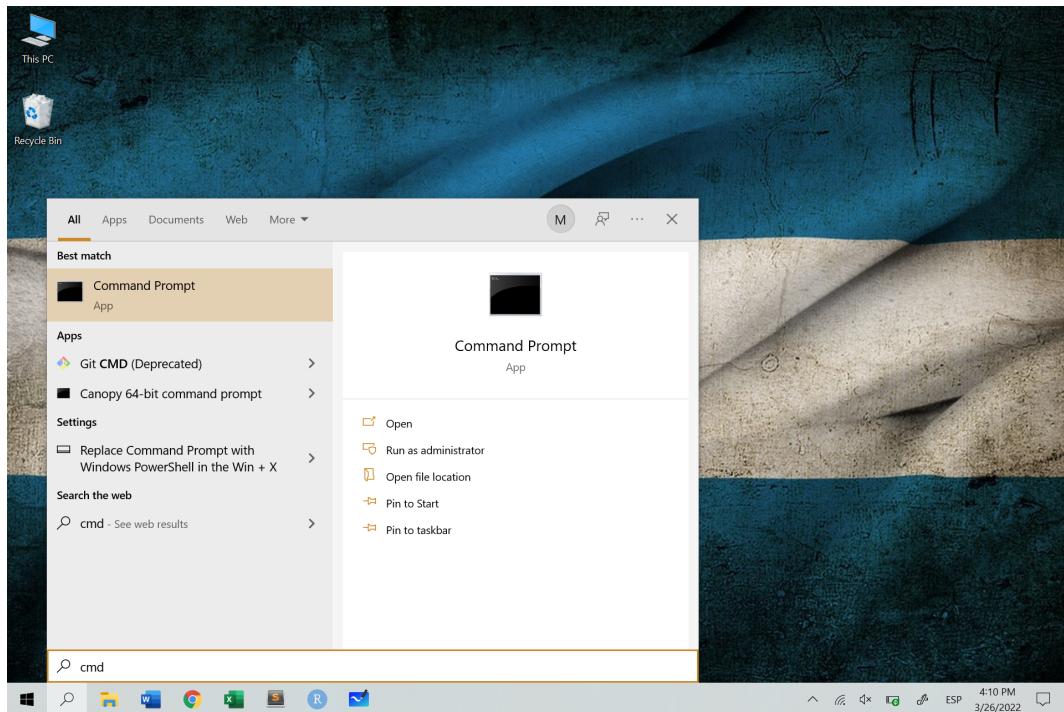


Figura 7.1: Abrir la terminal en Windows. En computadoras con Windows en español, en lugar de Command Prompt dice Símbolo del sistema.



Figura 7.2: Línea de comandos de Windows.

Si bien hay muchísimos comandos para utilizar en la terminal, acá vamos a mencionar algunos como ejemplo:

- Mostrar en qué carpeta (directorio) de la compu estamos situados: *pwd* en Linux o *cd* en Windows
- Listar todos los archivos y carpetas que tenemos en el directorio actual: *ls* en Linux o *dir* en Windows
- Entrar a una subcarpeta desde el directorio en el que estamos: *cd nombresubcarpeta*
- Ver la ayuda de los comandos: *help*
- Limpiar la consola: *cls* en Windows o *clear* en Linux
- Cerrar la consola: *exit*

Por ejemplo, podemos ver todos los archivos que existen en el directorio actual con *ls*:

```
Microsoft Windows [Version 10.0.19044.1586]
(c) Microsoft Corporation. All rights reserved.

C:\Users\Marcos>ls
'3D Objects'
'AppData'
'Application Data'
'Canopy'
'Contacts'
'Cookies'
'Desktop'
'Documents'
'Downloads'
```

Figura 7.3: Contenido del directorio actual (Windows).

Para los siguientes ejemplos, trabajaremos en una carpeta llamada **Ejemplos**, cuyo *path* es C:\Users\Marcos\Trabajo\Ejemplos que tiene la siguiente composición:

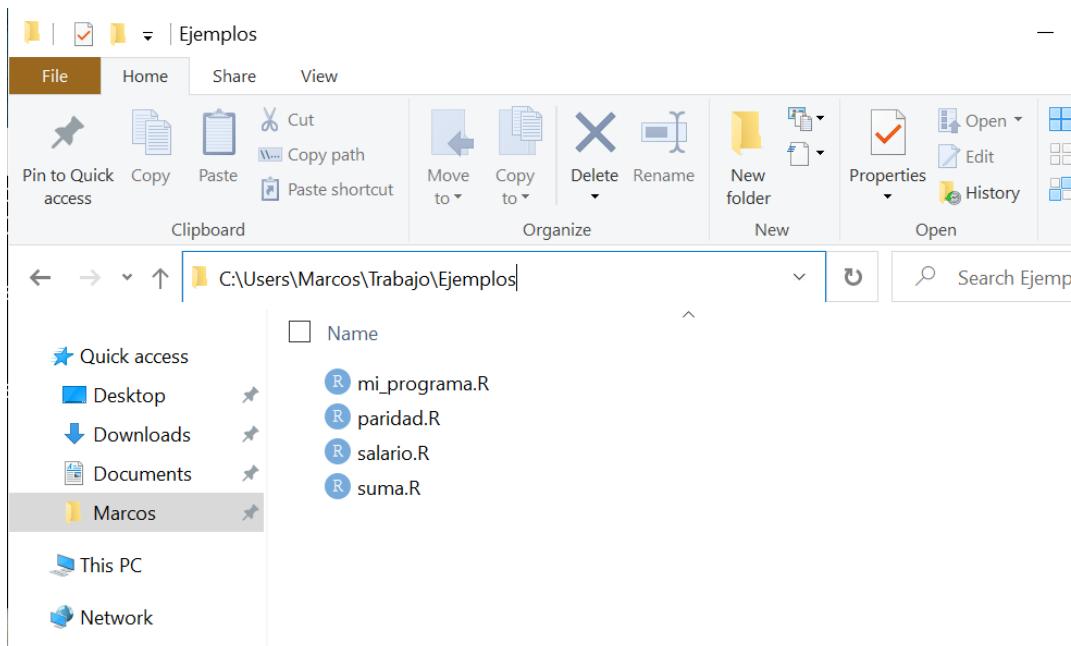


Figura 7.4: Carpeta en la cual deseamos trabajar.

Podemos convertir a dicha carpeta como nuestro nuevo directorio de trabajo con el comando *cd* (*change directory*): En Windows hay una forma más directa de abrir la terminal y que ya tenga seteada como directorio de trabajo a una carpeta deseada. Antes de abrir la terminal, vamos con el *Explorador de archivos* a la carpeta en cuestión, nos posicionamos en la barra del explorador, escribimos *cmd* y le damos **ENTER**. Automáticamente se abrirá la terminal, con esta carpeta como directorio de referencia.

Desde la terminal podemos correr nuestros programas de R. Hacer esto es necesario cuando tenemos que programar alguna tarea de gran escala que se ejecutará de manera remota en algún servidor o cuando necesitamos encapsular



```
Command Prompt
Microsoft Windows [Version 10.0.19044.1586]
(c) Microsoft Corporation. All rights reserved.

C:\Users\Marcos>cd Trabajo\Ejemplos
C:\Users\Marcos\Trabajo\Ejemplos>ls
mi_programa.R  paridad.R  salario.R  suma.R

C:\Users\Marcos\Trabajo\Ejemplos>
```

Figura 7.5: Cambio de directorio de trabajo y listado de archivos en el mismo.

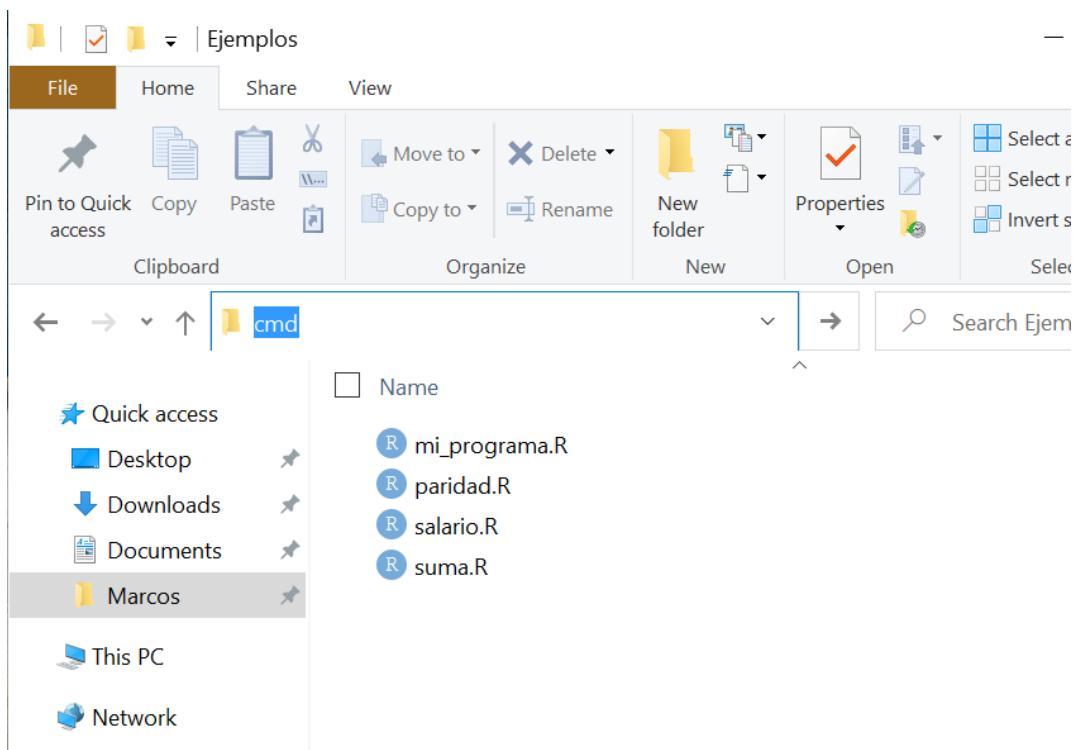


Figura 7.6: Abrir la terminal desde una carpeta en particular en Windows.

nuestro programa para que otros lo puedan correr sin siquiera saber nada de R.

Veamos un ejemplo. En la carpeta C:\Users\Marcos\Trabajo\Ejemplos tengo guardado el siguiente script, en un archivo llamado `mi_programa.R` que tiene este contenido:

```
a <- "¡Hola, Mundo!"
b <- 3
d <- 5
cat("=====\\n")
cat("          RESULTADOS          \\n")
cat("=====\\n\\n")
cat("El valor de b es ", b, ", mientras que d vale ", d, ".\\n\\n", sep = "")
cat("La suma entre ellos es igual a ", b + d, ".\\n\\n", sep = "")
cat("Este es un saludo:", a)
```

Para ejecutar este programa desde la terminal, sin abrir RStudio o R, utilizo el comando `Rscript`, que le indica a la computadora que el contenido del archivo `mi_programa.R` debe ser evaluado por R. Esto es lo que se observa en la consola:

```
C:\Windows\System32\cmd.exe
Microsoft Windows [Version 10.0.19044.1586]
(c) Microsoft Corporation. All rights reserved.

C:\Users\Marcos\Trabajo\Ejemplos>Rscript mi_programa.R
=====
          RESULTADOS
=====

El valor de b es 3, mientras que d vale 5.

La suma entre ellos es igual a 8.

Este es un saludo: ¡Hola, Mundo!
C:\Users\Marcos\Trabajo\Ejemplos>
```

Figura 7.7: Correr el programa de R desde la consola en Windows.

Todo lo que en el programa estaba encerrado en una llamada a la función `cat()` es lo que se muestra como mensajes en la terminal. Notar que la instrucción `RScript mi_programa.R` funcione, debemos tener como directorio de trabajo aquella carpeta que aloja al archivo `mi_programa.R`, en caso contrario el sistema nos alertará que el mismo no está disponible.

Para que lo anterior funcione en Windows, hay que indicarle al sistema operativo que `Rscript` es un comando que se instaló con R y que lo puede encontrar en la carpeta de los archivos del programa R. Esto hay que hacerlo una sola vez editando las **variables de entorno** de Windows, que son cadenas de texto que contienen información acerca del sistema para determinar, por ejemplo, dónde buscar algunos archivos. Esto se logra siguiendo estos pasos:

1. Fijarse en qué carpeta de la compu está instalado R. Seguramente lo encuentres si, abriendo el explorador de archivo, vas siguiendo este camino: `Este equipo > Windows (C:) > Archivos de programa > R > R-version > bin`. En esta carpeta tiene que haber dos archivos, llamados `R.exe` y `Rscript.exe`. Si es así, hacé clic con el botón derecho del mouse sobre cualquiera de ellos, luego en “Propiedades” y copiá el path que aparece en “Ubicación” (deberías copiar algo como `C:\Program Files\R\R-3.6.0\bin`).
2. En Inicio, escribir “Entorno” y hacer clic en la opción “Editar las variables de entorno del sistema (panel de control)”.
3. Hacer clic en el botón “Variables de entorno”.

4. En el cuadro “Variables del sistema”, hacer clic en la variable “Path” y luego en “Editar”.
5. Hacer clic en “Nuevo”, pegar la dirección C:\Program Files\R\R-3.6.0\bin y dar Enter. Luego, hacer clic en “Aceptar” tres veces para cerrar todo.
6. ¡Listo! Ya podés correr tus programas desde la consola con el comando *Rscript*.

Lo bueno de esto es que si corremos nuestros programas desde la terminal, podemos hacer cosas interactivas. Por ejemplo, para todo lo que pusimos LEER en nuestros pseudocódigos, ahora podemos hacer verdaderamente que la persona usuaria del programa provea los valores correspondientes.

Veamos algunos ejemplos.

### Práctica 2, Ejercicio 1: paridad de un número

La función `scan()` es la que permite *escanear* o *leer* valores que los usuarios ingresen por la terminal. Entre sus argumentos tenemos a `file`, que si lo seteamos como `file = "stdin"` indica que vamos a leer información desde la consola. Otros argumentos que son de utilidad incluyen a `n = 1`, que indica que sólo leeremos un valor y `quiet = TRUE` que le pide a esta función que no emita ningún mensaje. Por ejemplo, si el siguiente código se guarda en el archivo `paridad.R` y es ejecutado desde la consola, le va a pedir a la persona que lo esté usando que indique cualquier número y luego le va a comunicar si es par o impar:

```
cat("=====\\n")
cat("          PARIDAD DE UN NÚMERO          \\n")
cat("=====\\n\\n")
cat("Ingrese un número entero y presione enter:\\n")
n <- scan(file = "stdin", n = 1, quiet = TRUE)
if (n %% 2 == 0) {
  cat(n, "es par\\n")
} else {
  cat(n, "es impar\\n")
}
```

Esto es lo que ocurre en la terminal:

```
C:\Windows\System32\cmd.exe
C:\Users\Marcos\Trabajo\Ejemplos>Rscript paridad.R
=====
          PARIDAD DE UN NUMERO
=====

Ingrese un número entero y presione enter:
5
5 es impar

C:\Users\Marcos\Trabajo\Ejemplos>Rscript paridad.R
=====
          PARIDAD DE UN NUMERO
=====

Ingrese un número entero y presione enter:
120
120 es par

C:\Users\Marcos\Trabajo\Ejemplos>
```

Figura 7.8: Programa `paridad.R`.

### Práctica 2, Ejercicio 3: salario

En este ejemplo, tenemos que leer tres valores, dos de los cuales son de tipo carácter. Para esto tenemos que agregar

en la función `scan()` el argumento `what =`, que admite el ingreso de caracteres alfanuméricos (por default `scan()` sólo espera recibir valores numéricos). Si el siguiente código se guarda en el archivo `salario.R` y se lo ejecuta desde la consola, produce el resultado que se muestra en la imagen:

```
cat("=====\\n")
cat("          CÁLCULO DEL SALARIO      \\n")
cat("=====\\n\\n")
cat("Ingrese la cantidad de horas trabajadas:\\n")
horas <- scan("stdin", n = 1, quiet = TRUE)
cat("\\nIngrese el día de la semana (DOM LUN MAR MIE JUE VIE SAB):\\n")
dia <- scan("stdin", what = "", n = 1, quiet = TRUE)
cat("\\nIngrese el turno (M T N):\\n")
turno <- scan("stdin", what = "", n = 1, quiet = TRUE)

salario <- horas * 400
if (turno == "N") {
  salario <- salario + horas * 200
}
if (turno == "DOM") {
  salario <- salario + horas * 100
}
cat("\\nEl salario que se debe abonar es $", salario, "\\n", sep = "")
```

```
C:\Windows\System32\cmd.exe
C:\Users\Marcos\Trabajo\Ejemplos>Rscript salario.R
=====
          CÁLCULO DEL SALARIO
=====

Ingrese la cantidad de horas trabajadas:
8

Ingrese el día de la semana (DOM LUN MAR MIE JUE VIE SAB):
LUN

Ingrese el turno (M T N):
T

El salario que se debe abonar es $3200

C:\Users\Marcos\Trabajo\Ejemplos>
```

Figura 7.9: Programa `salario.R`.

#### Práctica 4, Ejercicio 1: suma de elementos de un vector<sup>1</sup>

En este ejercicio escribimos una función para sumar los elementos de un vector. Vamos a ver cómo hacer para que un usuario nos diga cuáles son los valores que quiere sumar desde la consola. Primero preguntamos cuántos números se desean sumar y luego los recibimos en el vector `v`. Si el siguiente código queda guardado en el archivo `suma.R` y se lo corre desde la terminal, produce el resultado que se muestra en la imagen.

<sup>1</sup>Si estás leyendo esto antes de que hayamos visto en clase la Unidad 5, podés omitir este último ejemplo y retomarlo más adelante en el cursado.

```

cat("=====\\n")
cat("          SUMA DE NÚMEROS          \\n")
cat("=====\\n\\n")
cat("¿Cuántos números va a ingresar?\\n")
n <- scan("stdin", n = 1, quiet = TRUE)
cat("\\nIngrese los números, presionando Enter luego de cada uno:\\n")
v <- scan("stdin", n = n, quiet = TRUE)
suma <- 0
for (i in 1:length(v)) {
  suma <- suma + v[i]
}
cat("\\nLa suma de los números es:", suma, "\\n")

```

```

C:\Windows\System32\cmd.exe
C:\Users\Marcos\Trabajo\Ejemplos>Rscript suma.R
=====
SUMA DE NUMEROS
=====

¿Cuántos números va a ingresar?
5

Ingrese los números, presionando Enter luego de cada uno:
6
-1
2
10
4

La suma de los números es: 21

C:\Users\Marcos\Trabajo\Ejemplos>

```

Figura 7.10: Programa suma.R.

## 7.2. Uso de argumentos en la línea de comandos al ejecutar código de R

En ejemplos anteriores hemos visto cómo capturar distintas piezas de información de forma interactiva mediante la función `scan()` mientras estamos ejecutando un programa de R desde la línea de comandos.

En otras ocasiones, en lugar de pausar la ejecución del programa a la espera de que el usuario ingrese algún valor, es conveniente especificar algunas opciones directamente en la instrucción `Rscript` que ejecuta el código.

Por ejemplo, imaginemos que tenemos un programa llamado `resumen.R` que se encarga de hacer un análisis descriptivo de un conjunto de datos que están guardados en un archivo de texto de nombre `02_05_22.txt`. Al ejecutar este programa desde la terminal, podemos indicar el nombre del archivo como un argumento adicional de esta forma:

```
Rscript resumen.R 02_05_22.txt
```

Ahora supongamos que este mismo tipo de análisis se repite todos los días con datos nuevos. En lugar de modificar nuestro script `resumen.R`, ejecutamos lo anterior con el nombre del archivo que corresponda y listo:

```
Rscript resumen.R 03_05_22.txt
Rscript resumen.R 04_05_22.txt
Rscript resumen.R 05_05_22.txt
```

Para que esto funcione, el programa que está guardado en `resumen.R` debe ser capaz de capturar el nombre del archivo que tiene leer y que el usuario se lo está pasando como un argumento adicional en la instrucción `Rscript`. La función que se encarga de capturar los argumentos adicionales que enviamos desde la terminal es `commandArgs()`. Toma todos los elementos que escribamos y los reúne en un vector de tipo carácter. Por ejemplo, el archivo `ejemplo1.R` tiene el siguiente contenido:

```
# Capturar los argumentos pasados desde la terminal en un vector
args <- commandArgs(trailingOnly = TRUE)

# Contar cuántos argumentos nos pasaron
cat("Nos pasaron", length(args), "argumentos.\n\n")

# Mostrar los argumentos que nos pasaron
cat("Los argumentos que nos pasaron son:\n")
cat(args, "\n")

# Aunque los argumentos sean números, son tomados como carácter
cat("\nLos argumentos se toman como valores de tipo:\n")
class(args)
```

Al ejecutarlo desde la línea de comandos con los argumentos “hola”, “chau” y “4” obtenemos:

```
Rscript ejemplo1.R hola chau 4
```

```
WARNING: ignoring environment value of R_HOME
Nos pasaron 3 argumentos.
```

```
Los argumentos que nos pasaron son:
hola chau 4
```

```
Los argumentos se toman como valores de tipo:
[1] "character"
```

Si lo ejecutamos sin argumentos:

```
Rscript ejemplo1.R
```

```
WARNING: ignoring environment value of R_HOME
Nos pasaron 0 argumentos.
```

```
Los argumentos que nos pasaron son:
```

```
Los argumentos se toman como valores de tipo:
[1] "character"
```

Ahora vamos a suponer que el programa `ejemplo2.R` tiene como objetivo contar un chiste o decir un refrán, según lo que se le pida en el único argumento que se le pasa al correrlo desde la terminal. Si el argumento es igual “chiste”, se cuenta el chiste; si es igual a “refran” se cuenta el refrán; y en otro caso no se hace nada. El contenido del archivo es:

```
# Capturar los argumentos pasados desde la terminal en un vector
args <- commandArgs(trailingOnly = TRUE)

if (args[1] == "chiste") {
  cat("- Juan, cómo has cambiado.\n- Yo no soy Juan.\n- Más a mi favor.\n\n")
```

```

} else if (args[1] == "refran") {
  cat("No por mucho madrugar amanece más temprano.\n\n")
} else {
  # Genero un error para que el programa se detenga, avisando lo que pasa
  stop("El argumento provisto debe ser igual a chiste o refran.\n")
}

```

Ejecutamos este archivo pasando distintos valores para su argumento:

```
Rscript ejemplo2.R refran
```

```

WARNING: ignoring environment value of R_HOME
No por mucho madrugar amanece más temprano.

```

```
Rscript ejemplo2.R chiste
```

```

WARNING: ignoring environment value of R_HOME
- Juan, cómo has cambiado.
- Yo no soy Juan.
- Más a mi favor.

```

```
Rscript ejemplo2.R hola
```

```

WARNING: ignoring environment value of R_HOME
Error: El argumento provisto debe ser igual a chiste o refran.
Execution halted

```

Podemos controlar la cantidad de argumentos admitidos generando errores en el código para aquellas situaciones donde el usuario envíe menos o más que la cantidad deseada. Por ejemplo, en el caso anterior, es obligatorio enviar uno y sólo un argumento:

```

# Capturar los argumentos pasados desde la terminal en un vector
args <- commandArgs(trailingOnly = TRUE)

# Controlar la cantidad de argumentos
if (length(args) == 0 || length(args) > 1) {
  stop("Debe proveer exactamente un argumento, que debe ser igual a chiste o refran.\n")
}

if (args[1] == "chiste") {
  cat("- Juan, cómo has cambiado.\n- Yo no soy Juan.\n- Más a mi favor.\n\n")
} else if (args[1] == "refran") {
  cat("No por mucho madrugar amanece más temprano.\n\n")
} else {
  # Genero un error para que el programa se detenga, avisando lo que pasa
  stop("El argumento provisto debe ser igual a chiste o refran.\n")
}

```

Veamos lo que pasa si cumplimos o no con la cantidad exacta de argumentos que hay que pasarle al código de R:

```
Rscript ejemplo3.R
```

```

WARNING: ignoring environment value of R_HOME
Error: Debe proveer exactamente un argumento, que debe ser igual a chiste o refran.
Execution halted

```

```
Rscript ejemplo3.R chiste refran
```

```

WARNING: ignoring environment value of R_HOME
Error: Debe proveer exactamente un argumento, que debe ser igual a chiste o refran.

```

Execution halted

Rscript ejemplo3.R chiste

```
WARNING: ignoring environment value of R_HOME
- Juan, cómo has cambiado.
- Yo no soy Juan.
- Más a mi favor.
```

Imaginemos por último que es obligatorio pasar un primer argumento (“chiste” o “refran”) y que opcionalmente se puede pasar un segundo argumento, que se va a tratar de un número para indicar cuántas veces queremos que el chiste o el refrán se repita. Como todos los argumentos se pasan como datos de tipo carácter, para poder usar el número tendremos que convertirlo a dato de tipo numérico.

```
# Capturar los argumentos pasados desde la terminal en un vector
args <- commandArgs(trailingOnly = TRUE)

# Si no proveyó argumentos, generar un error y que se detenga el programa
if (length(args) == 0) {
  stop("Debe proveer al menos un argumento (chiste o refran).")
}

# Si proveyó más de 2 argumentos, generar un error y que se detenga el programa
if (length(args) > 2) {
  stop("No debe proveer más de 2 argumentos. El primero es obligatorio (chiste o refran) y el segundo es")
}

# Si no hay segundo argumento, args[2] es NA
if (is.na(args[2])) {
  n <- 1
} else {
  n <- as.numeric(args[2])
}

# Repetir n veces
for (i in 1:n) {
  if (args[1] == "chiste") {
    cat("- Juan, cómo has cambiado.\n- Yo no soy Juan.\n- Más a mi favor.\n\n")
  } else if (args[1] == "refran") {
    cat("No por mucho madrugar amanece más temprano.\n\n")
  } else {
    # Genero un error para que el programa se detenga, avisando lo que pasa
    stop("El argumento provisto debe ser igual a chiste o refran.\n")
  }
}
```

Veamos ahora cómo funciona:

Rscript ejemplo4.R refran 5

```
WARNING: ignoring environment value of R_HOME
No por mucho madrugar amanece más temprano.
```

No por mucho madrugar amanece más temprano.

No por mucho madrugar amanece más temprano.

No por mucho madrugar amanece más temprano.

Rscript ejemplo4.R chiste 3

WARNING: ignoring environment value of R\_HOME

- Juan, cómo has cambiado.
- Yo no soy Juan.
- Más a mi favor.

- Juan, cómo has cambiado.

- Yo no soy Juan.
- Más a mi favor.

- Juan, cómo has cambiado.

- Yo no soy Juan.
- Más a mi favor.

Rscript ejemplo4.R refran

WARNING: ignoring environment value of R\_HOME

No por mucho madrugar amanece más temprano.

Rscript ejemplo4.R

WARNING: ignoring environment value of R\_HOME

Error: Debe proveer al menos un argumento (chiste o refran).

Execution halted

Si querés probar estos ejemplos, podés crear los archivos de código mencionados copiando y pegando las instrucciones o descargarlos de este enlace.

Además de la función `commandArgs()` existen paquetes de R para poder trabajar con argumentos y opciones de formas mucho más elaboradas, como los paquetes `argparse` y `optparse`, entre otros.

### 7.3. Generación de secuencias

A continuación mostramos cómo generar algunos vectores numéricos en R:

```
# Generar vectores con secuencias numéricas
```

```
# Enteros de 1 a 5
```

```
1:5
```

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

```
# Números de 1 a 10 cada 2
```

```
seq(1, 10, 2)
```

```
[1] 1 3 5 7 9
```

```
# Números de 0 a -1 cada -0.1
```

```
seq(0, -1, -0.1)
```

```
[1] 0.0 -0.1 -0.2 -0.3 -0.4 -0.5 -0.6 -0.7 -0.8 -0.9 -1.0
```

```
# Siete números equiespaciados entre 0 y 1
```

```
seq(0, 1, length.out = 7)
```

```
[1] 0.0000000 0.1666667 0.3333333 0.5000000 0.6666667 0.8333333 1.0000000
```

```
# Repetir el 1 tres veces  
rep(1, 3)
```

```
[1] 1 1 1
```

```
# Repetir (1, 2, 3) tres veces  
rep(1:3, 3)
```

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

```
# Repetir cada número tres veces  
rep(1:3, each = 3)
```

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

```
# Generar una matriz diagonal  
diag(c(3, 7, 1, 5))
```

```
[,1] [,2] [,3] [,4]  
[1,] 3 0 0 0  
[2,] 0 7 0 0  
[3,] 0 0 1 0  
[4,] 0 0 0 5
```

```
# Generar una matriz identidad  
diag(rep(1, 5))
```

```
[,1] [,2] [,3] [,4] [,5]  
[1,] 1 0 0 0 0  
[2,] 0 1 0 0 0  
[3,] 0 0 1 0 0  
[4,] 0 0 0 1 0  
[5,] 0 0 0 0 1
```



# Bibliografía

## Bibliografía de la asignatura

- Casale, Juan Carlos (2012). *Introducción a la Programación*. Buenos Aires: Editorial Fox Andina.
- Cerrada Somolinos, José y Collado Machuca, Manuel (2015). *Fundamentos De Programación*. Madrid: Editorial Universitaria Ramón Areces.
- Martínez López, Pablo (2013). *Las bases conceptuales de la Programación: una nueva forma de aprender a programar*. La Plata: Editorial de la Universidad Nacional de Quilmes.
- Quetglás, Gregorio; Toledo Lobo, Francisco; Cerverón Lleó, Vicente (1995). *Fundamentos de informática y programación*. Valencia: Editorial V.J.
- Wicham, Hadley (2019). *Advanced R*. Florida: Editorial Chapman and Hall/CRC.

## Textos consultados para la reseña histórica de la programación:

- Ada Lovelace - Wikipedia. Consultado el 21/04/21.
- Difference engine - Wikipedia. Consultado el 21/04/21.
- Generaciones de las computadoras. Consultado el 21/04/21.
- Generaciones de ordenadores. Consultado el 21/04/21.
- Historia de la programación - Wikipedia. Consultado el 21/04/21.
- Konrad Zuse - people.idsia.ch. Consultado el 21/04/21.
- Los huesos de Napier, la multiplicación árabe y tú. Consultado el 21/04/21.
- Significado de computación. Consultado el 21/04/21.

## Otros artículos consultados

- Aprender programación - Curso de Java. Consultado el 25/04/21.
- Fundamentos de programación. Consultado el 25/04/21.
- Hardware y software: definiciones y conceptos. Consultado el 25/04/21.
- Introduction to programming. Consultado el 25/04/21.
- Programación estructurada. Consultado el 25/04/21.
- The 7 Most Common Types of Errors in Programming and How to Avoid Them. Consultado el 25/04/21.
- What is abstraction in programming?. Consultado el 25/04/21.