

Análisis y solución de problemas aplicando algoritmos

**Breve descripción:**

El avance tecnológico lleva implícito la frontera del conocimiento. Esto, a su vez, genera nuevos conceptos, terminologías, notaciones, nomenclaturas y su representación a través de diagramas, planos o artefactos que terminan convirtiéndose en un lenguaje natural de la tecnología en cuestión. En este componente se presentan los artefactos, diagramas y nomenclaturas con los que se describen los algoritmos.

**Septiembre 2023**

Tabla de contenido

[Introducción 4](#_Toc145350418)

[1. Metodología de algoritmos 6](#_Toc145350419)

[1.1. Análisis 7](#_Toc145350420)

[1.2. Diseño 8](#_Toc145350421)

[2. Diagramas de flujo 14](#_Toc145350424)

[2.1. Elaborar diagramas de flujo 20](#_Toc145350425)

[2.2. Herramientas digitales para creación y prueba de algoritmos 20](#_Toc145350426)

[3. Uso de identificadores y palabras reservadas 23](#_Toc145350427)

[4. Operadores y jerarquía en los operadores 26](#_Toc145350428)

[5. Estructuras de control secuencial 30](#_Toc145350430)

[5.1. Ejemplo estructura de control secuencial 31](#_Toc145350431)

[5.2. Estructura condicional 33](#_Toc145350432)

[5.3. Estructuras de iteración o repetitivas 37](#_Toc145350433)

[6. Estructura de datos 46](#_Toc145350436)

[6.1. Vectores 46](#_Toc145350437)

[6.2. Matrices 50](#_Toc145350438)

[7. Programación modular 52](#_Toc145350439)

[8. Conceptos básicos de programación 57](#_Toc145350440)

[9. Entornos de codificación e instalación JavaScript 60](#_Toc145350441)

[10. Sintaxis del lenguaje JavaScript 62](#_Toc145350442)

[10.1. Funciones y operaciones matemáticas 62](#_Toc145350443)

[10.2. Tipos de datos, operadores y orden de evaluación 67](#_Toc145350444)

[10.3. Expresiones y comentarios 74](#_Toc145350445)

[10.4. Estructuras de selección 75](#_Toc145350446)

[10.5. Estructuras de repetición 79](#_Toc145350447)

[10.6. Estructura de datos 85](#_Toc145350450)

[10.7. Estructuras de salto 90](#_Toc145350455)

[10.8. Métodos de ordenamiento y búsqueda 93](#_Toc145350456)

[11. Depuración y fallas de sintaxis 100](#_Toc145350461)

[12. Fallas de lógica 101](#_Toc145350462)

[13. Manejo de errores y excepciones 105](#_Toc145350468)

[Síntesis 108](#_Toc145350469)

[Material complementario 110](#_Toc145350470)

[Glosario 112](#_Toc145350471)

[Referencias bibliográficas 113](#_Toc145350472)

[Créditos 115](#_Toc145350473)

Introducción

Aquí comienza el estudio del componente formativo “**Análisis y solución de problemas aplicando algoritmos**”; explore la información del video que se muestra enseguida, la cual le contextualiza sobre los aspectos más importantes de los temas por desarrollar. ¡**Adelante**!

1. Introducción, codificación de algoritmos



[**Enlace de reproducción de video**](https://youtu.be/L2KbJaJx8WM?si=bEilOTyonRGy3EEX)

|  |
| --- |
| **Síntesis del video: Introducción, codificación de algoritmos** |
| El producto principal de la industria del “software” es un conjunto de instrucciones de computador que, correctamente implantadas, pueden llamarse programa, módulo de programa, aplicación, servicio o, simplemente, producto “software”.  Codificar este conjunto de instrucciones es la tarea que trata de convertir algoritmos en un programa o módulo que compone un programa; y que una computadora o teléfono inteligente (“smartphone”) es capaz de ejecutar. Estas instrucciones están inscritas en un lenguaje de programación que puede ser interpretado o compilado por una máquina.  El lenguaje de programación es como un sistema de comunicación entre la persona (programador) y la máquina, que consiste en un conjunto de reglas gramaticales bien definidas que le ofrecen al programador la capacidad de escribir (programar) los algoritmos de los que está compuesto un sistema informático.  En particular se usa el lenguaje Javasrcipt ya que casi todos los navegadores lo interpretan y se puede usar para publicar servicios y aplicaciones en internet. |

# Metodología de algoritmos

De manera implícita, en las distintas áreas productivas de la sociedad se usan algoritmos y un par de ejemplos pueden ser que, en la construcción de una casa, el ingeniero o el arquitecto analiza el terreno, diseña unos planos, realiza maquetas y así continúan siguiendo una secuencia lógica de procesos, para la resolución del problema; o un abogado que antes de defender a un cliente analiza las pruebas, elabora un plan, practica sus argumentos y sigue un método para lograr la defensa de su cliente. Se puede concluir, entonces, que una solución a la medida de un problema requiere una metodología que lleve a la solución final.

En el proceso de desarrollo de programas informáticos, aparte de conocer esta metodología, se necesita conocer las funciones que puede realizar un computador y la representación de las mismas.

En general, existen tres (3) fases secuenciales para la elaboración de un algoritmo: análisis, diseño e implementación. Aunque se abordarán las dos primeras, a continuación, se detallan las tres:

1. **Análisis**:

* Planteamiento del problema.
* Identificación de datos.
* Identificación de procesos y operaciones.
* Identificación de salidas y resultados.

1. **Diseño**:

* Desarrollo de algoritmos.
* Verificación de algoritmos.

1. **Implementación**:

* Codificación de algoritmo.
* Prueba y depuración.
* Documentación.
* Mantenimiento.

## Análisis

La fase de análisis trata de responder a la pregunta: ¿Qué es lo que se debe hacer?, y esto es necesario para entender el problema. Por ello, es preciso relacionar los datos de entrada, salida o resultados con los procesos a ejecutar con esta información y los datos auxiliares que se puedan requerir en las tareas.

Hasta este punto parece que esas son las actividades propias del análisis, sin embargo, el éxito de un algoritmo consiste en reducir a cero la brecha de lo que el cliente necesita y los resultados que el algoritmo entrega. Para lograrlo en su totalidad, es importante indagar otros aspectos que corresponden a la naturaleza del problema, las siguientes preguntas modeladoras son de referencia casi obligante a tener en cuenta:

* ¿Qué es?
* ¿Quiénes intervienen?
* ¿Cómo interviene o se comporta?
* ¿Cuándo?
* ¿Cómo lo hace?
* ¿Por qué? Y ¿Para qué?

## Diseño

En esta etapa se responde, básicamente, a la pregunta ¿Cómo se va a hacer?, refiriéndose a la manera en la que el sistema va a realizar la tarea solicitada. Lo importante en este punto, es que debe quedar bien definido el problema, sin dar espacio a más de una interpretación de la acción o acciones que debe hacer el sistema.

En el proceso de desarrollo del algoritmo, las siguientes actividades son obligatorias, sobre todo si se está empezado en el diseño de soluciones basadas en algoritmos y no se cuenta con mucha experiencia:

* Definir las partes del algoritmo
* Pasos para el desarrollo del algoritmo
* La representación del algoritmo (notación, diagrama o seudocódigo)

Para la verificación del algoritmo, se aplica una técnica denominada pruebas de escritorio o trazas, que trata de la validación y verificación del algoritmo mediante la ejecución de las sentencias u operaciones que lo componen (proceso), y así determinar sus resultados de salida a partir de una base determinada de elementos en la entrada del proceso.

Como el resultado del proceso de diseño es un algoritmo, es obligatorio saber representarlo de alguna manera, y se empleará inicialmente con la notación de pseudocódigo para hacer este tipo de representaciones.

### Notación de diseño: pseudocódigo

Para expresar el contenido de un programa de computadora, se necesita de un lenguaje de programación que las computadoras pueden interpretar. Sin embargo, se puede expresar ese mismo contenido en el lenguaje natural lengua materna (español), empleando expresiones sencillas que lo simplifican y abrevian. Cuando se emplea esta técnica, se dice que está escrito en pseudocódigo.

**Ejemplo**: el cliente pide que se desarrolle un programa de computadora que le solicite al usuario un número, luego el sistema debe mostrar en pantalla el número digitado.

A continuación, se aplica la metodología explicada:

1. **Análisis**: el dato de entrada debe ser un número ingresado por el usuario, pero antes de eso el sistema debe solicitarle al usuario que lo ingrese (digite) al programa, si no, ¿de qué otra manera el usuario sabría que el programa está esperando el ingreso de ese dato? Por lo tanto, una posible operación inicial dentro del algoritmo sería escribir en pantalla que “por favor se ingrese un número”.

La segunda operación es la de recibir ese número desde el teclado, para ingresarlo al programa, pero esto implica que se debe definir un dato intermedio donde se guardará ese valor y el cual identificará el valor ingresado (a este dato se denomina variable porque puede tener cualquier valor entero según digite el usuario). Ya ocurre el primer problema: para guardar el dato (variable) se debe definir dónde se almacenará o como podría referirse al dato ingresado dentro del programa, por lo tanto, antes de empezar el algoritmo se debe indicar al computador que reserve un espacio para guardar un número entero.

Una vez definida la variable, es posible referirse a ella, leer desde el teclado el número entero y almacenarlo en dicha variable.

Por último, el sistema debe imprimir el contenido de la variable (dato de salida), con lo cual se puede dar final al programa.

1. **Diseño**: en anterior análisis se contemplaron los datos de entrada, datos auxiliares (variable) operaciones (impresión en pantalla, lectura de datos e impresión en pantalla de resultados). Pero son muchas las palabras usadas en el análisis por lo tanto lo abreviamos con el siguiente código:

El código de la imagen inmediatamente anterior se explica así:

En ALGORITMO se indica el nombre del algoritmo, y después se declaran las variables que serán usadas en él (con VAR) indicando su tipo de dato (un valor de tipo ENTERO, para este ejemplo).

Entre INICIO y FIN se escriben las diferentes acciones que realizará el algoritmo y usamos ESCRIBIR() para indicar con esta palabra que se mostrará un mensaje por pantalla (el texto que se va a mostrar 'tal cual' debe ir entre paréntesis dobles), y LEER() para recibir el dato desde el teclado.

Se debe tener en cuenta que cada línea termina en punto y coma (excepto INICIO y FIN), el punto y coma indica que lo que se ejecuta es una sentencia que hace una única operación, mientras que lo que está entre INICIO y FIN agrupa varias sentencias que se ejecutan en un orden determinado.

### Pruebas de escritorio o trazas

Las pruebas de escritorio son las simulaciones de ejecución de un algoritmo que permite determinar la validez o efectividad del mismo. Se trata de escribir en una tabla con tantas columnas como variables tiene el algoritmo y seguir las instrucciones poniendo los valores correspondientes.

Descargue y explore el PDF denominado **Prueba de escritorios o trazas**, que se encuentra en la carpeta Anexos y amplíe sus conocimientos sobre este punto.

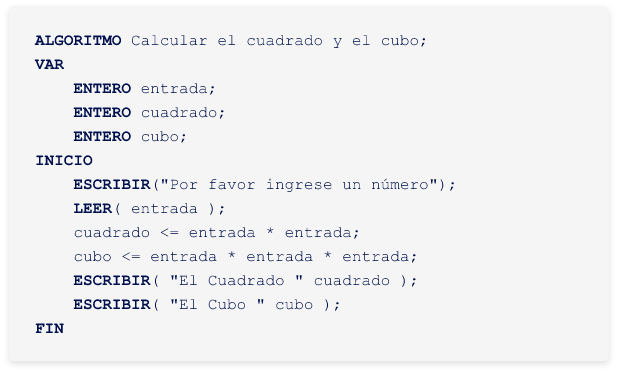
**Nuevo ejemplo**: escribir un algoritmo que calcule el cuadrado y el cubo de un número introducido por teclado y mostrar los resultados para el cubo y el cuadrado.

1. **Análisis**: nuevamente el dato de entrada debe ser un número ingresado por el usuario, y al igual que en el ejemplo anterior se debe solicitar al usuario que lo digite.

Antes de realizar las operaciones de elevar al cuadrado y al cubo, se necesita definir las dos variables que recibirán estos datos (el cuadrado, y el cubo), por lo tanto, el algoritmo debe definirlas antes de realizar las operaciones y asignarles a ellas los resultados.

Una vez realizadas las operaciones el algoritmo debe mostrar en pantalla el contenido de estas variables, con un mensaje texto que indique si es el resultado del cuadrado o del cubo.

1. **Diseño**: la siguiente imagen tiene la representación del resultado al análisis anterior:



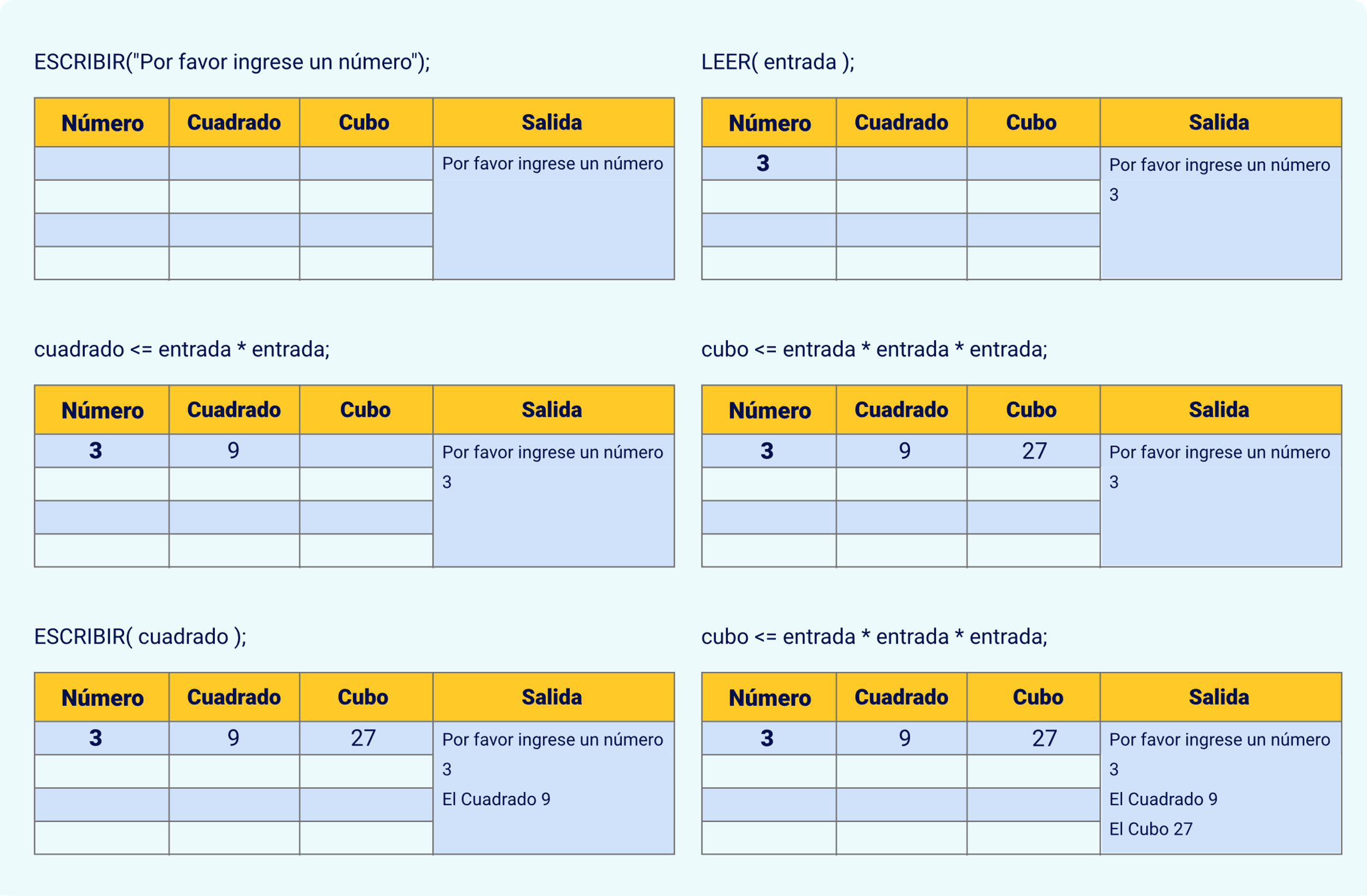
Las pruebas de escritorio permiten verificar que el pseudocódigo realice lo que se espera al resolver el problema. Sin embargo, por ahora es importante que a todos los pseudocódigos se les aplique, al menos, una de prueba de escritorio o las que se consideren necesarias.

La mayoría de las pruebas de escritorio se representan en una tabla, donde en cada columna se anotan las variables y condiciones si las hay, en el orden en que aparecen en el algoritmo, para entenderlo se tiene que recorrer línea a línea el pseudocódigo y así identificar tanto las variables como condiciones e ir anotándolas en la tabla de la prueba de escritorio.

Es importante que en la tabla no aparezcan variables o condiciones repetidas y en la última columna se pueden anotar los datos de salida para entonces también anotar “Salida”.

Ejecutando, línea a línea, el algoritmo que propone el ejemplo anterior, se va llenando la tabla como se muestra enseguida:

1. Tablas de traza del algoritmo



Cada tabla de traza expresa la cantidad de columnas necesarias según las variables que contenga el algoritmo.

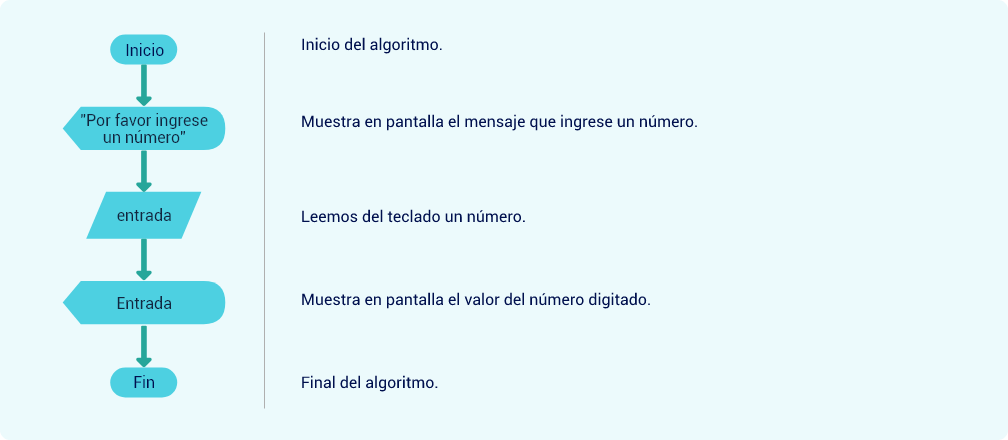
También es posible realizar pruebas de escritorio a otras representaciones de algoritmos, como los diagramas de flujo, expuestos a continuación, y que se representan con algoritmos recomendando que a cada ejemplo presentado se le realice la prueba de escritorio correspondiente.

# Diagramas de flujo

Una forma común de presentar los algoritmos es con diagramas de flujo, que consiste en una representación por medio de un gráfico. Mientras en seudocódigo las sentencias se representaban con preposiciones en diagramas de flujo la preposición es reemplazada por una figura.

Tomando el ejemplo propuesto en la figura 1, la representación del algoritmo sería la siguiente:

1. Diagrama de flujo imprimir un entero



La estructura del diagrama de flujo imprimir un entero es:

* **inicio**: inicio del algoritmo.
* **“por favor ingrese un número”**: muestra en pantalla el mensaje que ingrese un número.
* **entrada**: leemos del teclado un número.
* **entrada**: muestra, en pantalla, el valor del número digitado.
* **fin**: final del algoritmo.

Chequee el contenido del PDF denominado **Ventajas de los diagramas de flujo**, que se encuentra en la carpeta a Anexos y asimile tales ventajas, allí expuestas.

Retomando los ejemplos dados a lo largo de este componente formativo, la representación del algoritmo que calcula el cuadrado y el cubo de un número introducido por teclado y mostrar los resultados del cubo y del cuadrado, es:

1. Diagrama de flujo imprimir el cuadrado y el cubo de un entero



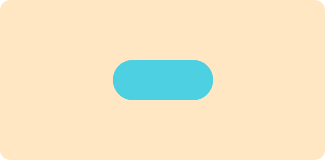
El diagrama de flujo Imprimir el cuadrado y el cubo de un entero presenta una estructura así:

* **inicio**: inicio del algoritmo
* **“por favor ingrese un número”:** muestra en pantalla el mensaje que ingrese un número.
* **entrada**: leemos del teclado un número.
* **cuadrado = entrada X entrada**: calcula el cuadrado del número digitado.
* **cubo = entrada X entrada X entrada**: calcula el cubo del número digitado.
* **“El cuadrado” cuadrado**: muestra el cuadrado del número digitado.
* **“El cuadrado” cuadrado**: muestra el cubo del número digitado.
* **fin**: final del algoritmo.

La Organización Internacional para la Estandarización (ISO, por su sigla en inglés) y el Instituto Nacional Americano de Estandarización (ANSI, por su sigla en inglés), estandarizaron los símbolos que han sido mayormente aceptados en 1985. Con el fin de evitar la utilización de símbolos diferentes para representar procesos iguales (López, 2009).

Los siguientes son los principales símbolos para crear diagramas de flujo:

* **Inicio/Final**. Se utiliza para indicar el inicio y el final de un diagrama; del Inicio solo puede salir una línea de flujo y al Final solo debe llegar una línea (López, 2009).



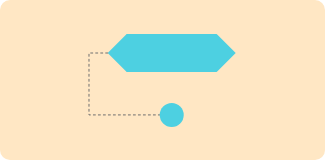
* **Decisión**. Indica la comparación de dos datos y dependiendo del resultado lógico (falso o verdadero) se toma la decisión de seguir un camino del diagrama u otro (López, 2009).



* **Entrada General**. Entrada/Salida de datos en General (en esta guía, solo se usará para la Entrada) (López, 2009).



* **Iteración**. Indica que una instrucción o grupo de instrucciones deben ejecutarse varias veces (López, 2009).



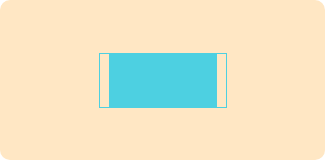
* **Entrada por teclado**. Instrucción de entrada de datos por teclado. Indica que el computador debe esperar a que el usuario teclee un dato que se guardará en una variable o constante (López, 2009).



* **Salida Impresa**. Indica la presentación de uno o varios resultados en forma impresa (López, 2009).



* **Llamada a subrutina**. Indica la llamada a una subrutina o procedimiento determinado (López, 2009).



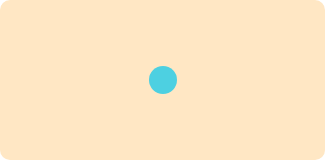
* **Salida en Pantalla**. Instrucción de presentación de mensajes o resultados en pantalla (López, 2009).



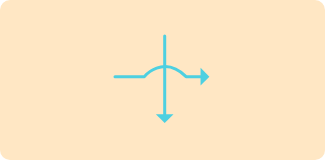
* **Acción/Proceso General**. Indica una acción o instrucción general que debe realizar el computador (cambios de valores de variables, asignaciones, operaciones aritméticas, etc.) (López, 2009).



* **Conector**. Indica el enlace de dos partes de un diagrama dentro de la misma página (López, 2009).



* **Flujo**. Indica el seguimiento lógico del diagrama. También indica el sentido de ejecución de las operaciones (López, 2009).



* **Conector**. Indica el enlace de dos partes de un diagrama en páginas diferentes (López, 2009).



## Elaborar diagramas de flujo

Cuando el algoritmo se desea representar como diagrama de flujo, se deben tener en cuenta algunos principios para su elaboración (Rojas y Ñacato, 1980).

Conozca tales principios y generalidades, descargando y explorando la infografía contenida en el PDF denominado **Diagramas de Flujo**, que se encuentra en la carpeta Anexos.

## Herramientas digitales para creación y prueba de algoritmos

Para aprender a reconocer los elementos que componen los diagramas de flujo se puede hacer uso de la herramienta denominada Draw.io a través del enlace: <https://app.diagrams.net/> con la que puede usar una cuenta de correo de Google, o misena.edu.co para acceder y crear un nuevo diagrama.

Es ideal ingresar y seleccionar los diagramas que se muestran en la siguiente figura:

1. Seleccionar formas



Nota.Tomada de https://app.diagrams.net/

Como se muestra en la figura anterior, la selección de formas contempla opciones como: formas básicas, esquemas de diagramas, entre otras.

En caso de contar con intermitencia de internet se puede realizar el ejercicio propuesto a través del programa de libre distribución disponible en <http://dia-installer.de/> o puede acceder a la versión de escritorio de Draw.io disponible en <https://app.diagrams.net/>

Consulte el video que se propone en el siguiente enlace y afiance sus conocimientos en cómo descargar draw io 2 para dibujar diagramas de flujo:

<https://www.youtube.com/watch?v=encz3h8TLWA>

El programa de mayor uso para el diseño y depuración de algoritmos en pseudocódigo es PSeInt disponible en <http://pseint.sourceforge.net/>

# Uso de identificadores y palabras reservadas

En la elaboración de diagramas de flujo o algoritmos muchas veces se necesitan datos auxiliares, a continuación, se definen algunos de estos tipos de datos auxiliares.

Los identificadores son nombres que se dan a los elementos utilizados para resolver un problema y poder diferenciar unos de otros. De acuerdo con López (2009), al asignar nombres (identificadores) a variables, constantes y procedimientos se deben tener en cuenta algunas reglas, a saber:

* Pueden estar formados por una combinación de letras y números (saldoMes, salario, fecha2, etc.).
* El primer carácter de un nombre debe ser una letra.
* La mayoría de los lenguajes de programación diferencian las mayúsculas de las minúsculas.
* Los nombres deben ser nemotécnicos, con solo leerlos se puede entender lo que contienen.
* Deben ser muy descriptivos; no utilizar abreviaturas, a menos que se justifique.
* Es conveniente utilizar una sola palabra para nombrar páginas, controles, variables, etc.
* No utilizar caracteres reservados (%, +, /, >, etc.) tampoco letras acentuadas (á, é, í, ó, ú).
* No utilizar palabras reservadas por los lenguajes de programación.

**Ejemplo**: listar los identificadores usados en los algoritmos previos.

**Solución**: en el algoritmo de imprimir un número entero usando el identificador de entrada para representar el número digitado.

En el algoritmo de imprimir el cuadrado y el cubo de un número entero empleando el identificador de entrada para representar el número digitado, el identificador cuadrado para representar el valor del cuadrado de entrada y el identificador cubo para almacenar el cubo entrada.

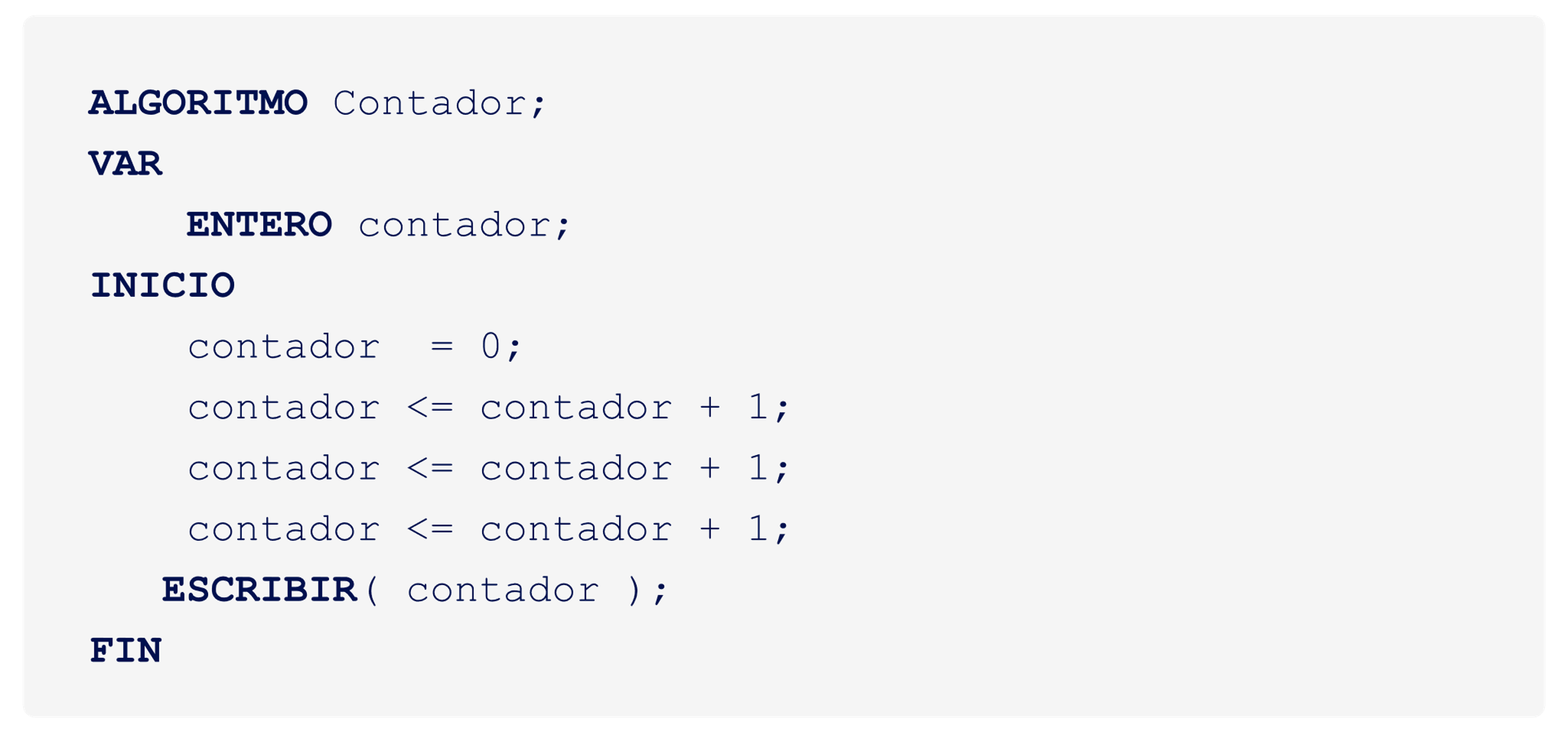
Todos los lenguajes de programación tienen unas palabras para nombrar sus comandos, instrucciones y funciones; estas palabras no se pueden usar como indicadores.

* **Variables**. Para los algoritmos con diversos conjuntos de datos iniciales, se debe determinar una independencia entre los datos iniciales de un problema y la estructura de su solución. Esto se hace mediante la utilización de variables (cantidades que se suelen emplear y que durante la ejecución del algoritmo pueden tomar cualquier valor de un intervalo de valores posibles).

Las variables son espacios de trabajo (contenedores) dispuestos para almacenar datos (valores). El valor de una variable puede ir cambiando en algún paso del algoritmo o permanecer sin cambios; por lo tanto, el valor de una variable es el del último valor o dato asignado a esta.

* **Constantes**. Las constantes se crean de la misma forma que las variables, a diferencia que se procura su definición o creación a inicio del algoritmo. Las constantes consisten en datos que, luego de ser asignados, no cambian en ninguna instrucción del algoritmo; pueden contener constantes matemáticas (pi).
* **Contadores**. Un contador es una variable que está en ambos lados de una asignación (a la derecha y a la izquierda) interna, a la que se le suma un valor constante. Es necesario haberla iniciado al principio del algoritmo, ya que va a ser leído su contenido.

Ejemplo: definir un contador llamado contador que se incremente en uno cada vez tres veces.



Como muestra el algoritmo, tres veces se incrementa en 1 (uno) el valor del contador y al final de la ejecución su valor es de 3. Es importante notar que contador inicia con un valor, el valor 0, si esta asignación inicial no se hace el computador pondría un valor aleatorio y el valor al final del algoritmo sería indeterminado.

* **Acumuladores**. Los acumuladores son una versión ampliada de contadores, tienen las mismas características que un contador, excepto el valor que se va incrementando; no es una constante, sino que es un valor variable.

# Operadores y jerarquía en los operadores

Los operadores son símbolos que sirven para manipular datos. Y en un algoritmo es necesario dejar un espacio en blanco a cada lado del signo aritmético. Los operadores y las funciones que realizan se clasifican como se indica a continuación:

1. **Aritméticos**: permiten realizar operaciones entre datos de tipo numérico y dan como resultado otro valor de tipo numérico. Ejemplo: producto o multiplicación (\*); división (/); suma (+); resta (-); asignación (=).
2. **Alfanuméricos**: permiten operar con datos de tipo carácter (letras) o cadenas de texto. La mayoría de los lenguajes de programación admiten el operador + para realizar la concatenación (unión) de caracteres o cadenas.
3. **Relacionales**: permiten la comparación entre datos del mismo tipo de dato y dan como resultado dos valores posibles: verdadero o falso. Ejemplo: igual a (=); menor que (<); mayor que (>), mayor e igual que (>=).
4. **Lógicos**: posibilitan la evaluación lógica de dos expresiones de tipo lógico. Dan como resultado uno de dos valores posibles: verdadero o falso. Ejemplo: negación (no); conjunción (y); disyunción(o).

### Jerarquía en los operadores

Las computadoras ejecutan los operadores en un orden establecido y el siguiente es el orden (jerarquía) para ejecutar operadores:

1. Paréntesis (se ejecutan primero los más internos).
2. Signo (-2), si un valor es positivo o negativo.
3. Potencias (^) y Raíces (sqrt); Productos y Divisiones (\* y /), en este mismo orden.
4. Sumas y Restas (+ y -).
5. Concatenación (+).
6. Relaciónales (=, <, >).
7. Negación (no).
8. Conjunción (y).
9. Disyunción (o).

El siguiente ejemplo permite comprender mejor la jerarquía de operadores:

1. Orden de ejecución de una expresión matemática

| Orden | Operación | Explicación y expresión resultante |
| --- | --- | --- |
| 1 | 7 + 8 | Primero que todo lo que más valor tiene en la jerarquía son los paréntesis de manera que y en particular los más interiores de esta forma el computador se interesará en resolver primero el contenido de (7 + 8). Y el resultado será 15.  De esta forma, la expresión queda reducida a:  (2 + 15 \* 16 + 32) /4 +5 |
| 2 | 32 | De la expresión resultante luego de la operación 1, se observa que nuevamente hay un paréntesis por reducir, de manera que la computadora intentará resolver su contenido primero, como dentro del paréntesis está la expresión de elevar al cuadrado el número 3 (32) que tiene como resultado 9, la expresión resultante sería:  (2 + 15 \* 16 + 9) /4 +5 |
| 3 | 15 \* 16 | De los paréntesis de la expresión resultante la operación de más jerarquía es la multiplicación 15 \* 16 que tiene como resultado 240, la expresión resultante sería:  (2 + 240 + 9) /4 +5 |
| 4 | 2+24+9 | De la expresión resultante el que tiene mayor precedencia es el paréntesis por lo tanto la suma aritmética de su contenido data como expresión resultante:  251/4 +5 |
| 5 | 251/4 | De la expresión resultante el operador de mayor jerarquía es la división y da como resultado 62.75 que en la siguiente operación se le sumará el valor de 5 la expresión resultante será:  62,75 + 5 |
| 6 | 62,75 + 5 | 67,75 |

A continuación, se revisan dos ejemplos:

* **Ejemplo 1**. Una computadora debe ejecutar las siguientes operaciones, se debe escribir en el orden que la computadora ejecutará las operaciones de la siguiente expresión matemática: (2 + (7 + 8) \* 16 + 32) /4 +5.

**Solución**: primero que todo, lo que más valor tiene en la jerarquía son los paréntesis de manera que y en particular los más interiores, de esta forma la computadora se interesará en resolver primero el contenido de (7 + 8).

* **Ejemplo 2**. Escribir la siguiente ecuación como una expresión que la computadora pueda interpretar en el orden adecuado:

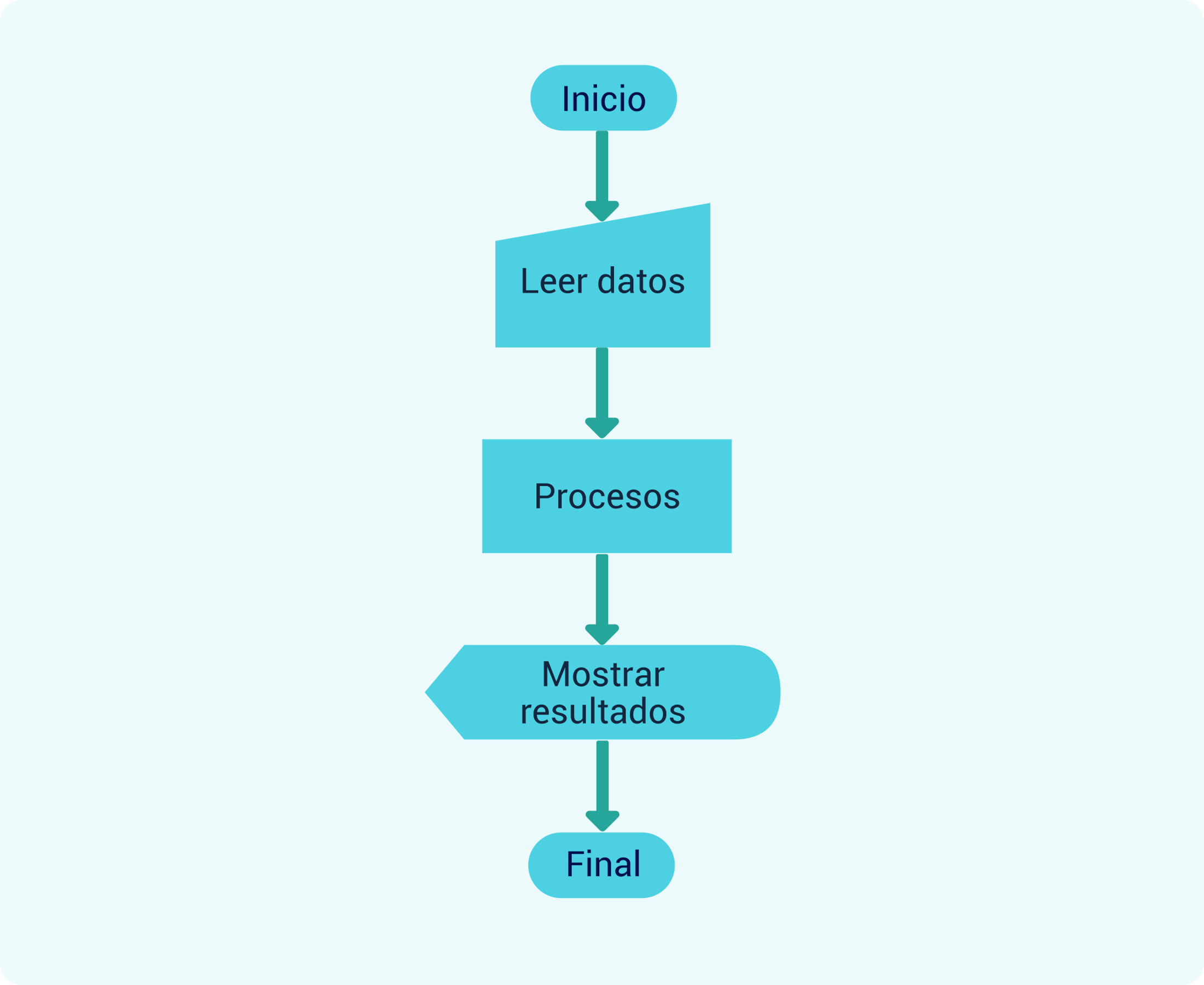
sqrt (62/2 + 7) / 82

# Estructuras de control secuencial

Una estructura es definida como un esquema con determinada distribución y orden que permite representar una idea de forma simplificada. La estructura de control secuencial es la más sencilla y es conocida como estructura lineal. Se compone de instrucciones que deben ejecutarse en forma consecutiva, una tras otra, siguiendo una línea de flujo. Los problemas más sencillos pueden resolverse haciendo uso únicamente de esta estructura. Por lo general, la estructura secuencial forma parte de soluciones a problemas complejos, en las que se la utilizan otras estructuras iterativas (repetir varias veces un conjunto de instrucciones) y condicionales (López, 2009).

La siguiente figura muestra la estructura general del algoritmo más básico.

1. Modelo de estructura secuencial



La estructura de algoritmo básico secuencial se ordena así: Inicio, leer datos, procesos, mostrar resultados, final.

## Ejemplo estructura de control secuencial

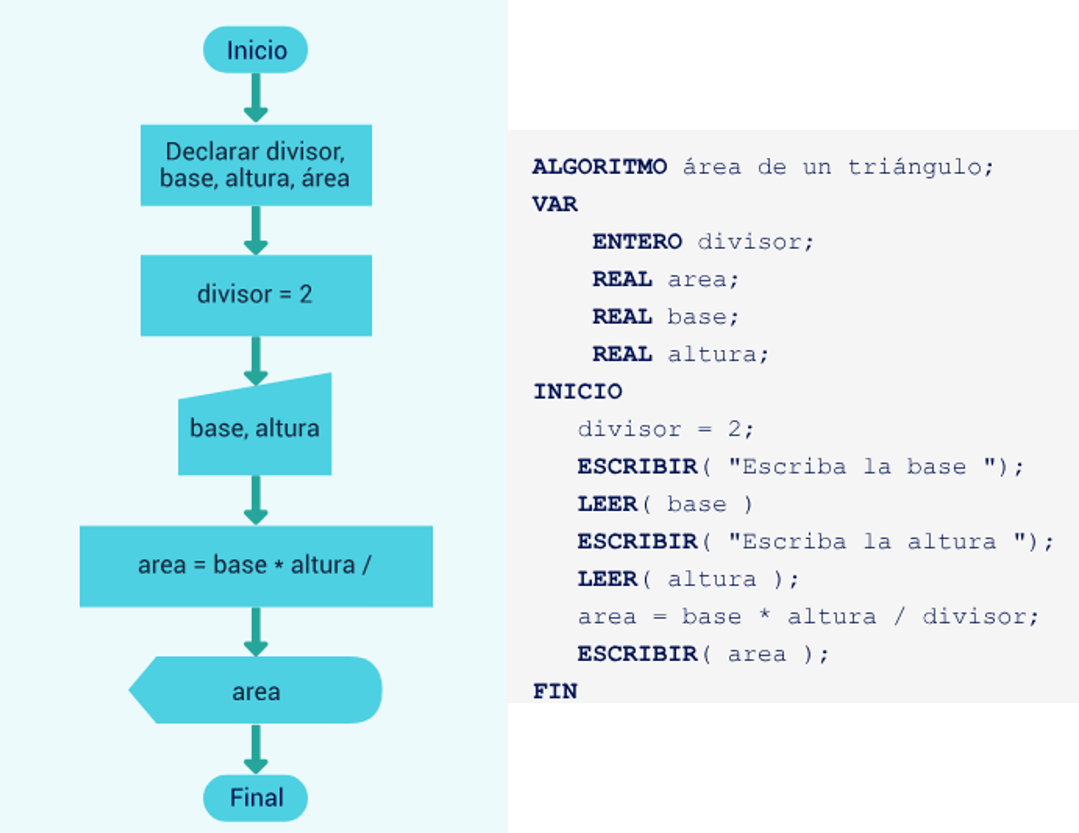
Escribir un algoritmo para calcular el área de cualquier triángulo rectángulo, en él se debe pedir al usuario que ingrese los valores de la altura y la base del triángulo:

1. **Análisis**.

* Formular el problema: está claramente planteado y consiste en calcular el área de un rectángulo.
* Resultados esperados: se espera que el algoritmo sea capaz de calcular el área del rectángulo y mostrar en la pantalla el resultado de ese cálculo.
* Datos disponibles: dos datos números que son la base y la altura del triángulo (se deben solicitar al usuario). El aprendiz debe preguntarse si sus conocimientos actuales de matemáticas le permiten resolver este problema, de no ser así, tendrá que plantear una estrategia para obtener los conocimientos requeridos. Para este ejemplo recordar que un triángulo rectángulo es aquel que uno de sus lados tiene un ángulo recto (90 grados).
* Restricciones: los valores de base y altura son variables y se deben solicitar al usuario.
* Procesos necesarios: definir variables; asignar el valor 2 a la constante usando para dividir (divisor); solicitar al usuario el valor de la altura del triángulo; solicitar al usuario el valor de la base; aplicar la fórmula de área; mostrar el resultado.

1. **Diseño**.

Mientras se está aprendiendo a diseñar algoritmos, es importante hacerlos en diagrama de flujo y en la medida en que se va adquiriendo experiencia con la representación de pseudocódigo, es más conveniente hacer esta representación, en razón de la velocidad a la hora de diseñarlo.



La estructura de este diagrama en concreto está integrada por:

* Punto de inicio
* Acción: declarar divisor, base, altura, área.
* Acción: divisor = 2
* Entrada por teclado: base, altura
* Acción: base \* altura /
* Salida en pantalla: área
* Punto final

Del diagrama de flujo y el pseudocódigo se puede notar lo siguiente: el identificador de área no tiene tilde por la razón antes mencionada, el tipo de dato REAL hace referencia a que es un número que no solamente puede contener un número entero, sino también un valor con cifras decimales.

## Estructura condicional

Es importante prestar atención a las estructuras que utilizan para resolver problemas y poderlas reconocer para lograr mayor control sobre la solución planteada.

La estructura condicional se emplea para indicarle al computador que debe evaluar una condición y, según el resultado, ejecutar el bloque de instrucciones. La forma más común está compuesta por una condición que se evalúa y dos bloques de instrucciones que se ejecutan: uno cuando la condición es verdadera y otro cuando la condición es falsa.

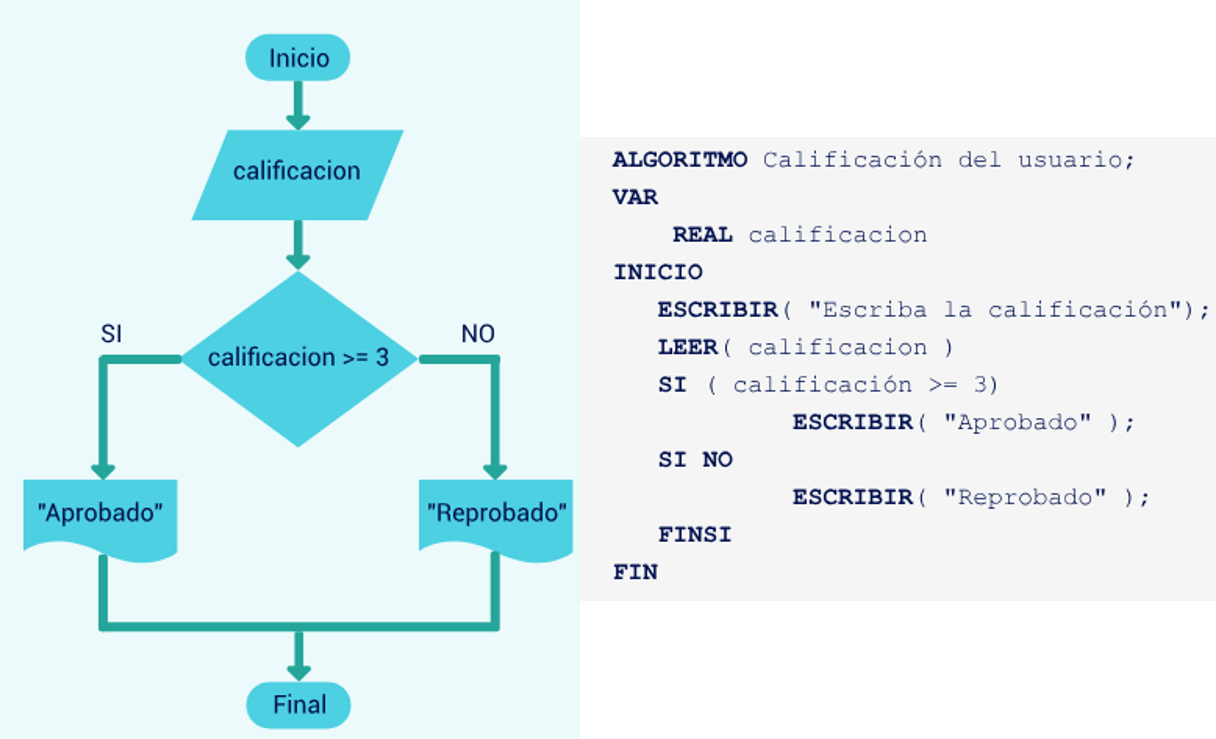
**Ejemplo**: un aprendiz aprueba un examen cuando la calificación de este es mayor o igual a 3. Elaborar un algoritmo que pida al usuario una calificación, aplicar el criterio de aprobación e imprimir “Aprobado” o “Reprobado”, según sea el caso.

1. **Análisis**.

* Formular el problema: es un problema sencillo de condición con dos operaciones o condición doble.
* Resultados esperados: un aviso que reporte si el estudiante obtiene “Aprobado” o “Reprobado” el examen.
* Datos disponibles: la calificación ingresada por el usuario. Para aprobar, la nota debe ser mayor o igual a 3.
* Restricciones: aplicar el criterio de aprobación (calificación mayor o igual a 3).
* Procesos necesarios: solicitar al usuario que ingrese la calificación. Evaluar si la calificación es igual o superior a 3; en caso de ser verdadero, reportar “Aprobado”; en caso contrario, reportar “Reprobado”.

1. **Diseño**.

En anterior análisis se contemplaron los datos de entrada, datos auxiliares (variable) operaciones (impresión en pantalla, lectura de datos e impresión en pantalla de resultados). Pero son muchas las palabras usadas en el análisis por lo tanto lo abreviamos con el siguiente código que, posteriormente, se explicará.



La estructura de este ejemplo se configura de la siguiente manera:

* Punto de inicio
* Entrada general: calificación
* Decisión: calificación mayor o igual a 3
* Salida impresa: Aprobado (cuando la decisión cumple la condición)
* Salida impresa: Reprobado (cuando la decisión no cumple la condición)
* Punto final

En el ejemplo anterior, se introduce la calificación por teclado, tanto en el diagrama de flujo (con la representación de trapecio) o el pseudocódigo con la preposición LEER, luego este valor se compara en un condicional, en el diagrama de flujo con la representación de un rombo, mientras que en el pseudocódigo se una la preposición SI, seguido de las preposiciones que se han de ejecutar en caso de que la condición sea verdadera (note que está escrita más a la derecha). También se usa la preposición SI NO para identificar el inicio de bloque de instrucciones a ejecutar si no se cumple la condición y la preposición FINSI para indicar que no hay más intrusiones que ejecutar si se cumple o no la condición.

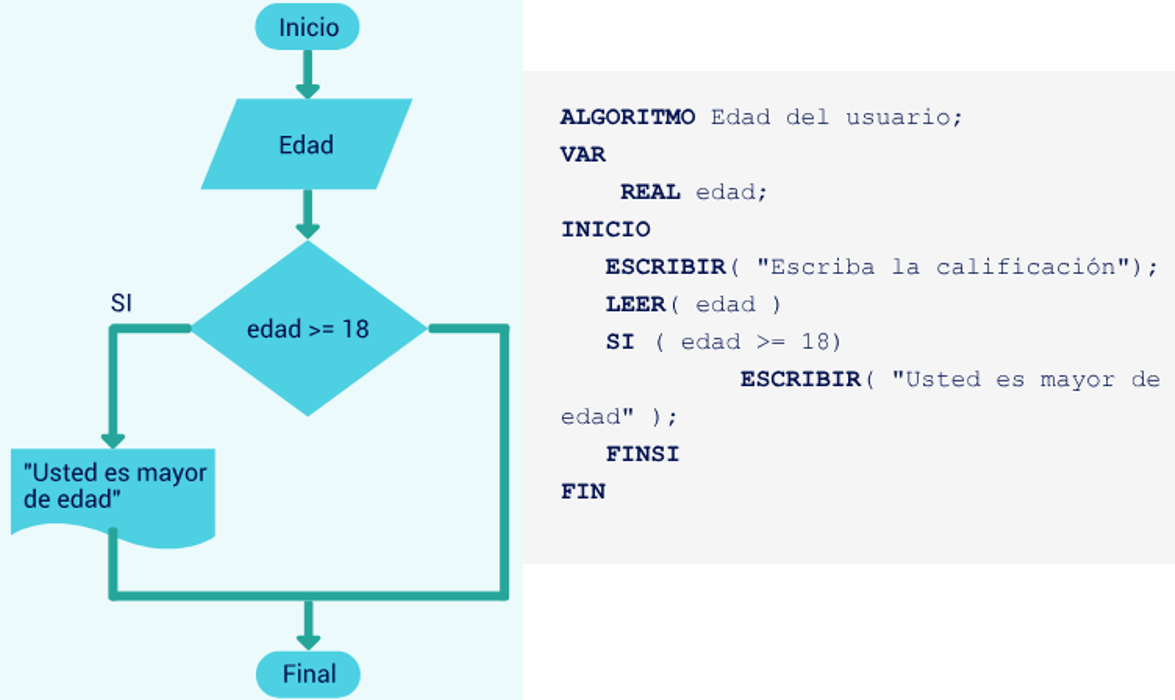
Como la condición evalúa dos condiciones si se cumple o no, y en cada caso existe un bloque de instrucciones (o alimenta una instrucción) a seguir se llama: ESTRUCTURA CONDICIONAL SIMPLE, ya que si la condición se cumple escribe algo y si no se cumple también hace otra operación.

**Nuevo ejemplo**: escribir un algoritmo en donde se pida la edad del usuario; si es mayor de edad debe aparecer un mensaje indicándolo.

1. **Análisis**.

* Formular el problema: es un problema sencillo solo consiste en usar una sentencia de repetición.
* Resultados esperados: el mensaje “Esto es un algoritmo” 7 veces impreso en pantalla.
* Datos disponibles: constante con el número de repeticiones (7).
* Restricciones: ninguna.
* Procesos necesarios: hacer que se imprima en pantalla el mensaje 7 veces.

1. **Diseño**.



Para el ejemplo donde se pide la edad del usuario y se generará una nota cuando sea mayor de edad, la estructura se configura así:

* Punto de inicio
* Entrada general: Edad
* Decisión: Edad mayor o igual a 18
* Salida impresa: Usted es mayor de edad (cuando la decisión sí cumple la condición).
* Punto final

En este ejemplo concreto, no se ejecuta nada en caso de que NO se cumpla la condición. Este tipo de estructura condicional se denomina ESTRUCTURA CONDICIONAL SIMPLE; además, todo el proceso (pedir la edad, comprobar la condición, mostrar o no el mensaje) se realiza una única vez y finaliza la ejecución del algoritmo, si se deseara que el programa repitiera estas mismas operaciones más de una vez, se tendría que usar una estructura de iteración como se verá a continuación.

## Estructuras de iteración o repetitivas

Las estructuras iterativas o de repetición permiten ejecutar una o varias instrucciones, un número determinado de veces o indefinidamente hasta que se cumpla una condición. Esta estructura ayuda a simplificar los algoritmos.

En programación existen, como mínimo, dos tipos de estructuras repetitivas las cuales tienen variantes en los diferentes lenguajes de programación.

La característica común es que ambos tipos permiten ejecutar una o varias instrucciones:

* Un número conocido de veces (Estructura PARA o FOR en inglés).
* Mientras se cumpla una condición (Estructura MIENTRAS o WHILE).

### Estructura PARA (FOR)

Esta estructura se usa cuando se sabe el número de veces que se debe ejecutar un conjunto de operaciones.

**Ejemplo**: escribir un procedimiento que muestre siete (7) veces en pantalla la frase “Esto es un algoritmo”. Mostrar en pantalla la salida del en el computador.

1. **Análisis**.

* Formular el problema: es un problema sencillo solo consiste en usar una sentencia de repetición.
* Resultados esperados: el mensaje “Esto es un algoritmo” 7 veces impreso en pantalla.
* Datos disponibles: constante con el número de repeticiones (7).
* Restricciones: ninguna.
* Procesos necesarios: hacer que se imprima en pantalla el mensaje 7 veces.

1. **Diseño**.



Para el algoritmo donde se requiere que se muestre, siete veces, la frase “esto es un algoritmo”, la estructura se configura así:

* Punto de inicio
* Iteración: Para i = 1 hasta 7
* Salida impresa: “Esto es un algoritmo”
* Punto final

La salida en pantalla sería:

Esto es un algoritmo

Esto es un algoritmo

Esto es un algoritmo

Esto es un algoritmo

Esto es un algoritmo

Esto es un algoritmo

Esto es un algoritmo

Existe una variable i que tendrá el valor 1, 2, 3 ... hasta 7, esta variable, se puede usar dentro del el bloque PARA hasta FINPARA.

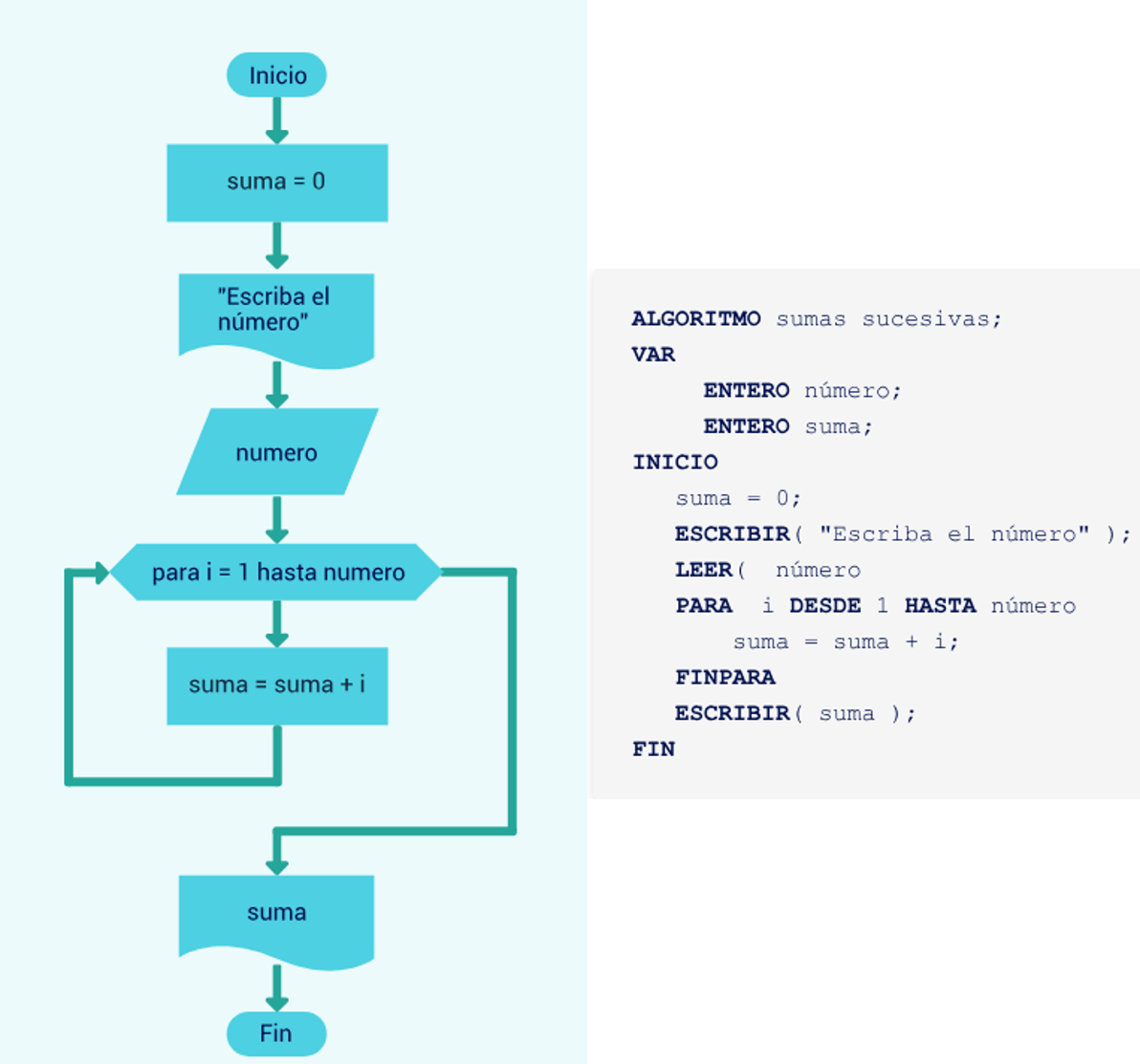
Ahora se presenta un ejemplo un poco más elaborado.

**Ejemplo**: se necesita elaborar un algoritmo que solicite al usuario un número entero y sume todos los números naturales que hay hasta ese número. Por ejemplo, si el usuario digita 3, el programa debe sumar: 1 + 2 + 3, si el usuario digita 5 el programa debe sumar 1+2+3+4+5. Al finalizar debe imprimir el resultado.

1. **Análisis**.

* Formular el problema: necesita la sumatoria de los números en secuencia natural.
* Resultados esperados: la suma de los números en sucesivos de uno hasta el número digitado por el usuario.
* Datos disponibles: el dato entero que digita el usuario es la entrada.
* Datos adicionales: se necesita un acumulador donde ir guardando el valor de las sumas sucesivas.
* Restricciones: ninguna.
* Procesos necesarios: definir un acumulador con valor inicial cero, imprimir para que el usuario sepa que debe digitar un número (“Escriba el número”), Leer del teclado el número del usuario, sumar cada vez el dato que se va incrementando de uno en uno al valor digitado por el usuario.

1. **Diseño**.



Para este caso del ejemplo, la estructura se configura así:

* Punto de inicio
* Acción: Suma = 0
* Salida Impresa: “Escriba el número”
* Entrada general: Número
* Iteración: para i = 1 hasta número
* Acción: Suma = suma + i
* Salida impresa: Suma
* Punto final

En la anterior figura se muestra el algoritmo en sus dos representaciones más comunes, se propone en secciones posteriores hacer una prueba de escritorio a este problema.

### Estructura MIENTRAS (WHILE)

La instrucción "Mientras", puede impedir la ejecución de un conjunto de instrucciones, si la evaluación de una condición lógica es falsa. Esto significa que es repetitiva únicamente cuando la evaluación de la condición es verdadera.

**Ejemplo**: realice un programa que permita obtener la factorial de un número dado por el usuario.

**Análisis**.

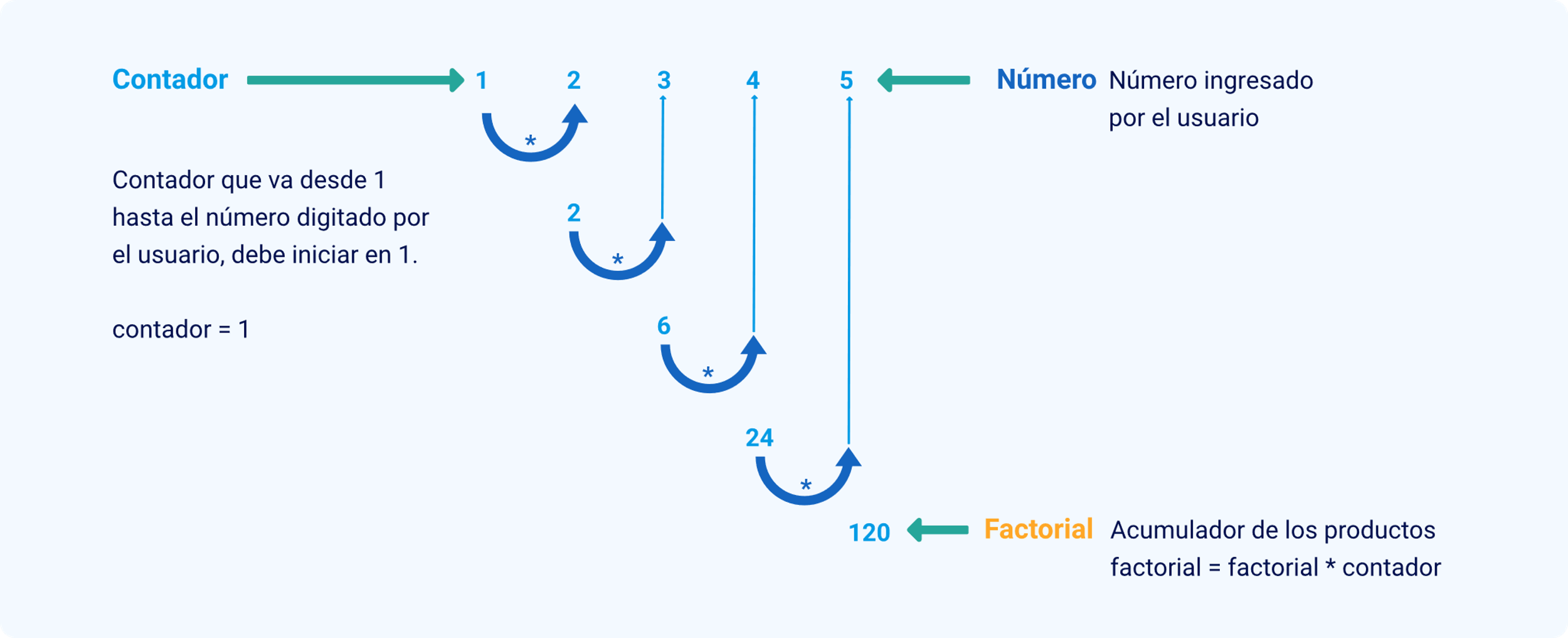
* **Formular el problema**: la factorial de un entero positivo n, se define como el producto de todos los números enteros positivos desde 1 hasta n.

Ejemplo: ¡Matemáticamente 5! = 5\*4\*3\*2\*1 o a la inversa: 5! = 1\*2\*3\*4\*5.

* **Resultados esperados**:

La factorial del número digitado por el usuario.

1. Proceso factorial

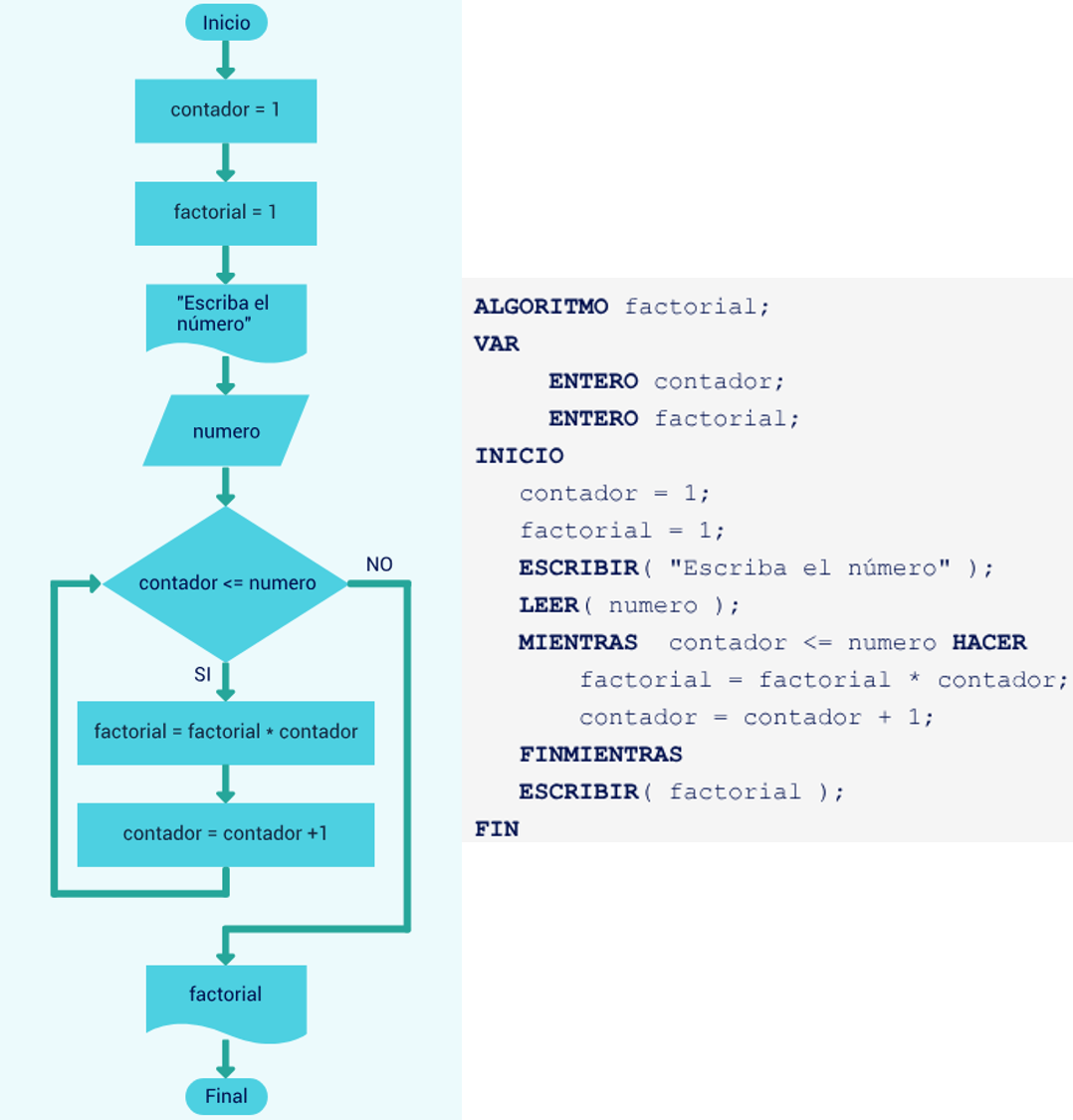


En la anterior figura se observa que es preciso ir incrementando el valor en 1 (valor constante - contador -) hasta que llegue al número 5 (datos suministrados por el usuario); es decir, 1\*2 = 2, este resultado se multiplica por 3 (2 \* 3 = 6), 6 se multiplica por 4 (6 \* 4 = 24), finalmente también 24 se multiplica por 5 y se obtiene la factorial (24 \* 5 = 120), de esta forma, los resultados de las multiplicaciones se van acumulando hasta obtener la factorial.

1. **Análisis**.

* Datos disponibles: el dato entero que digita el usuario es la entrada.
* Datos adicionales: se requiere de un acumulador donde ir guardando el valor de las multiplicaciones sucesivas, también un contador para ir contando el número secuencial (como lo hace el número i en el ciclo PARA).
* Restricciones: la repetición se detiene cuando
* Procesos necesarios: definir un acumulador con valor inicial cero, imprimir para que el usuario sepa que debe digitar un número (“Por favor el número”), leer del teclado el número del usuario, multiplicar cada vez el dato que se va incrementando de uno en uno al valor digitado por el usuario.

1. **Diseño**.



Para el caso propuesto del proceso factorial, la estructura se configura así:

* Punto de inicio
* Acción: Contador = 1
* Acción: Factorial = 1
* Salida impresa: “Escriba el número”
* Entrada general: Número
* Decisión: Contador < = número
* Acción: Factorial = Factorial \* contador
* Acción: Contador = Contador + 1
* Salida impresa: Factorial (cuando la decisión no cumple la condición)
* Punto final

# Estructura de datos

Es importante conocer el concepto de arreglos e identificar cuándo usarlos en la aplicación en los algoritmos. Así mismo, ser capaz de resolver problemas básicos mediante diagramas de flujo y pseudocódigo. De otra parte, existen varios tipos de arreglos y de estos los más comunes son los vectores y las matrices.

## Vectores

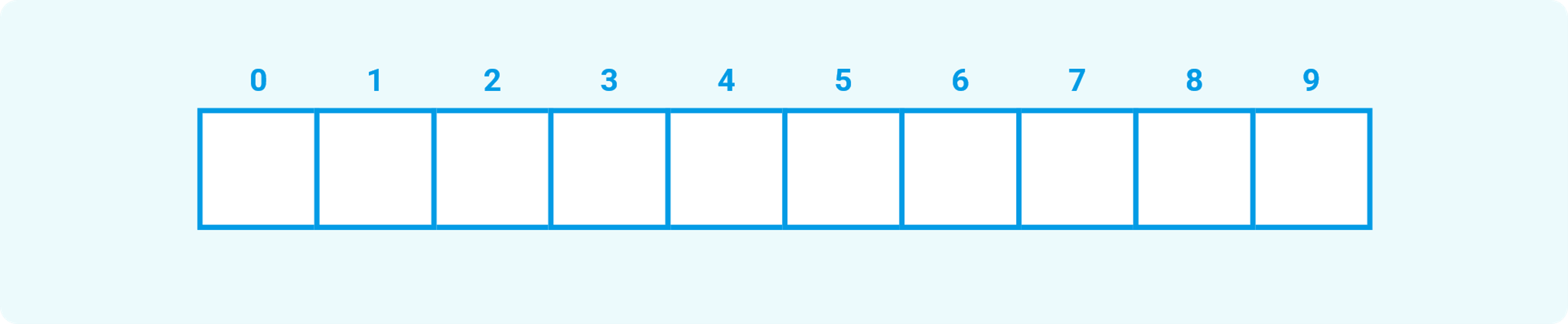
Los arreglos o “array”, en inglés, (matriz, arreglo o vector) se definen como un conjunto finito y ordenado de elementos del mismo tipo. La propiedad “ordenado” quiere decir que el elemento primero, segundo, tercero, …, enésimo de un “array” puede ser identificado y accedido. Cuando se dice que los elementos de un arreglo son homogéneos, significa que son del mismo tipo de datos. Un arreglo puede estar compuesto de todos sus elementos de tipo letras, otro puede tener todos sus elementos de tipo entero, etc. Los arreglos se conocen como matrices en matemáticas, y tablas en temas financieros (Vázquez, 2012).

Según sus dimensiones, los arreglos se clasifican en:

**Unidimensionales**. De una dimensión, comparables con una lista de elementos.

**Multidimensionales**. Son arreglos de varias dimensiones o los más comunes son bidimensionales (dos dimensiones o tablas) o arreglos multidimensionales (tridimensionales, por ejemplo).

1. Matriz unidimensional con 10 elementos



Cada vector está compuesto de un número de elementos ya definidos, esto hace que sea una estructura estática (no puede aumentarse ni reducirse). Cada elemento está referenciado por la posición que ocupa dentro del vector. Estas posiciones son llamadas índices; nótese que empieza a contar desde 0.

Para referirse un vector se usa un identificador como se hace con las variables o constantes, pero queda entre corchetes [ ], por ejemplo si el vector se llama lista, y se quiere acceder al quinto elemento del vector, se debe hacer así lista[4], es de notar que la quinta posición se accede con el índice 4, porque se empezó a contar desde 0.

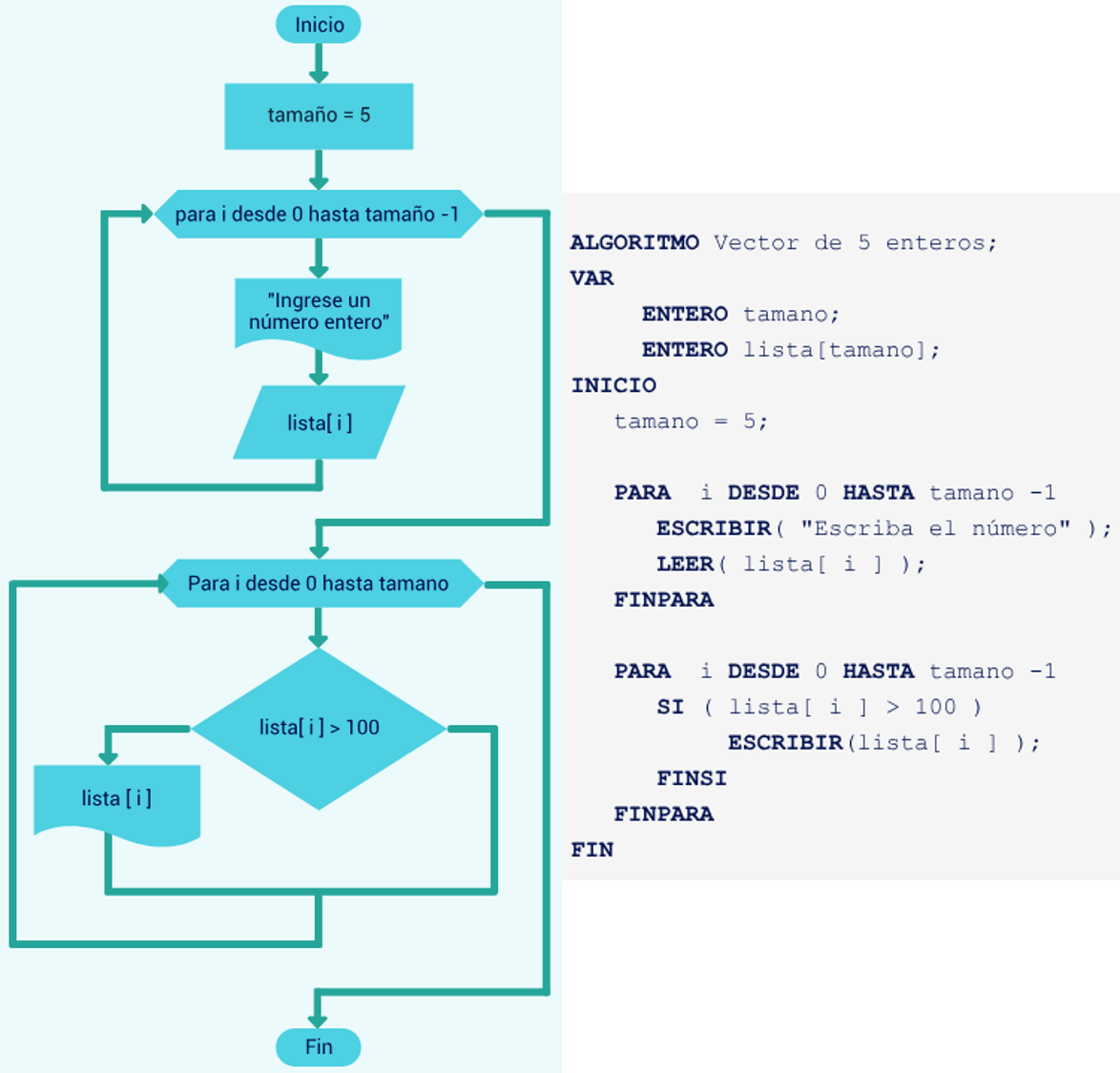
Para realizar las operaciones de asignación, lectura/escritura sobre un vector, se recomienda usar las estructuras repetitivas para, a través de ellas, manipular los índices correspondientes a las posiciones del arreglo.

**Ejemplo**: hacer un programa que pida 5 números enteros al usuario y los guarde en un arreglo de números en un arreglo, luego que imprima los números mayores de 100.

1. **Análisis**.

* Formular el problema: pedirle al usuario 5 números enteros almacenarlos en un arreglo y luego recorrer el arreglo para buscar los números mayores a 100.
* Resultados esperados: imprimir en pantalla de los 5 números ingresados por el usuario solamente los mayores a 100.
* Datos disponibles: 5 números ingresados por el usuario.
* Datos adicionales: por cada uno de esos números validar si son mayores que 100 y mostrar los que lo sean en la pantalla.
* Restricciones: ninguna.
* Procesos necesarios: se requiere definir un arreglo de tamaño 5, se necesita pedir 5 veces un entero y almacenarlos en el vector. Luego se debe recorrer las 5 posiciones del vector para comparar en cada uno de los datos si es mayor que 100, si el número comparador es lo que se debe imprimir. Como 5 es un número invariante, el programa lo define como una constante de nombre tamano (no se usa la ñ porque no permite ni acentos latinos ni tildes), se debe usar la sentencia repetitiva PARA porque se sabe el número de veces que se debe repetir cada acción.

1. **Diseño**.



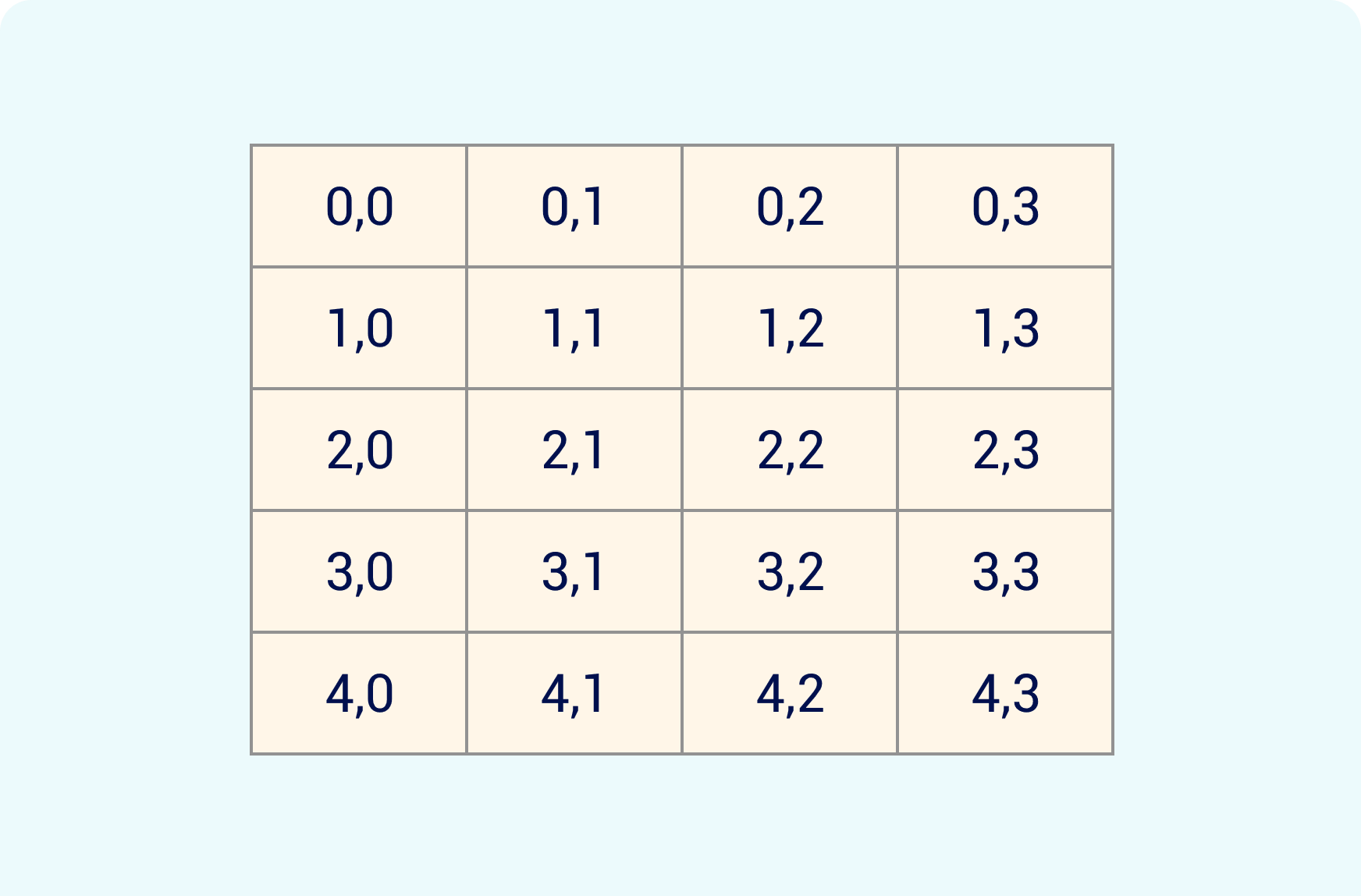
Para el ejemplo de hacer un programa que pida al usuario 5 números y estos sean guardados en un arreglo de números, la estructura se configura así:

* Punto de inicio
* Acción: Tamaño = 5
* Iteración: Para i desde 0 hasta tamaño - 1
* Salida impresa: “Ingrese un número entero”
* Entrada general: Lista [i]
* Iteración: Para i desde 0 hasta tamaño
* Decisión: Lista [i] mayor > 100
* Acción: Lista [i]
* Punto final

## Matrices

Las matrices son arreglos de más de una dimensión, en la siguiente figura se presenta un ejemplo de una matriz de 2 dimensiones que son 4x5 (correspondiente a 4 columnas y 5 filas).

1. Matriz de 4 X 5



Esta matriz, por ejemplo, se configura con cuatro columnas y cinco filas que contienen arreglos de números desde 0,0 hasta 0,3 en su primera fila, 1,0 hasta 1,3 en la segunda, 2,0 hasta 2,3 en la siguiente, 3,0 hasta 3,3 en la cuarta y 4,0 hasta 4,3 en su última fila.

Para recorrer estas estructuras de datos se requieren ciclos anidados; estas técnicas se abordarán más adelante.

# Programación modular

Existe un área que se denomina ingeniería del “software”, que se interesa en el proceso de creación o producción de “software” y concentra sus esfuerzos en aportar herramientas, procedimientos y técnicas para su construcción.

La ingeniería del “software” propone que la construcción de “software” esté pensada en la descomposición de un problema en un conjunto de subproblemas independientes entre sí, más sencillos de analizar y resolver; estos subproblemas pueden ser tratados separadamente unos de otros. Lo anterior consiste, básicamente, en la modularidad del “software”.

Debido a la modularidad, se pueden probar los subprogramas o módulos de manera independiente, depurando errores antes de su uso en el programa principal y almacenándose para su posterior reutilización cuantas veces se necesite.

El enfoque básico de modularidad en un algoritmo puede estar compuesto de otros algoritmos que se especializan en una función determinada. Para saber cómo descomponer un algoritmo en pequeños algoritmos, se listan los aspectos que pueden ayudar a definir la división de algoritmos:

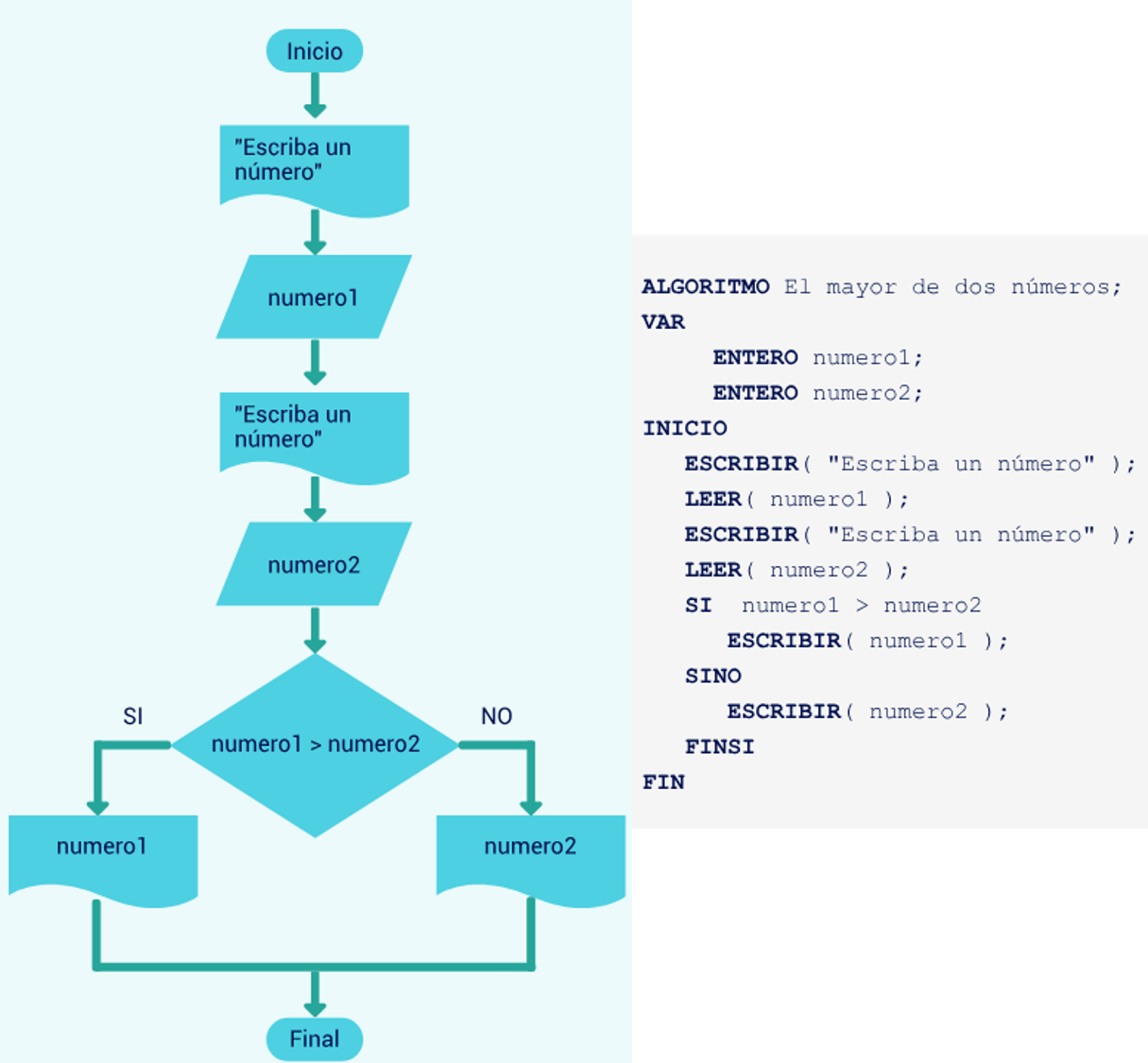
1. **Según la funcionalidad**: un algoritmo debe realizar una función específica y no ir más allá de sus responsabilidades. Por ejemplo, si se tiene un módulo que calcula la multiplicación de dos números y devuelve el resultado, no sería adecuado que, además, imprima mensajes en pantalla o haga otra cosa diferente de su propósito.
2. **Según su identificación**: cada módulo tiene un nombre de identificación que permite ejecutarlos y diferenciarlos de otros, es recomendable que al elegir este nombre que sea representativo de la función que realiza, de esa forma resulta intuitivo en el momento de usarlo dentro de un algoritmo.
3. **Según sus datos de entrada**: los módulos comúnmente reciben datos de entrada necesarios para realizar las operaciones requeridas. Se pueden clasificar los módulos de acuerdo con los parámetros que se requieren para obtener los resultados solicitados. Al momento de usar el módulo se tienen que agregar todos los parámetros necesarios, en el orden establecido y deben ser del tipo correcto.
4. **Parámetro de salida**: también se puede emplear los resultados a la salida del algoritmo para clasificar los módulos, ya que en el momento de definir un algoritmo se debe indicar el tipo de dato que devuelve al momento de la ejecución del algoritmo.
5. **Pruebas de escritorios**: como se mencionó antes, las pruebas escritorio sirven para depurar y probar que un algoritmo solucione el problema, en esta sesión se mencionarán los objetivos principales y se ejecutará una prueba a un algoritmo de mejor complejidad para su comprensión.

**Ejemplo**: realizar un algoritmo que solicite dos números e imprima cuál es el mayor de ambos, si son iguales, mostrar cualquiera.

1. **Análisis**.

* Formular el problema: es un algoritmo que solita y compara dos números para mostrar el mayor.
* Resultados esperados: el número que es mayor de ambos.
* Datos disponibles: 2 números ingresados por el usuario.
* Datos adicionales: ninguno.
* Restricciones: validar cuál número es mayor para saber cuál mostrar.
* Procesos necesarios: se requiere definir los datos a recibir, pedirle que al usuario que digite los números, comparar para saber cuál es el mayor y mostrarlo.

1. **Diseño**.



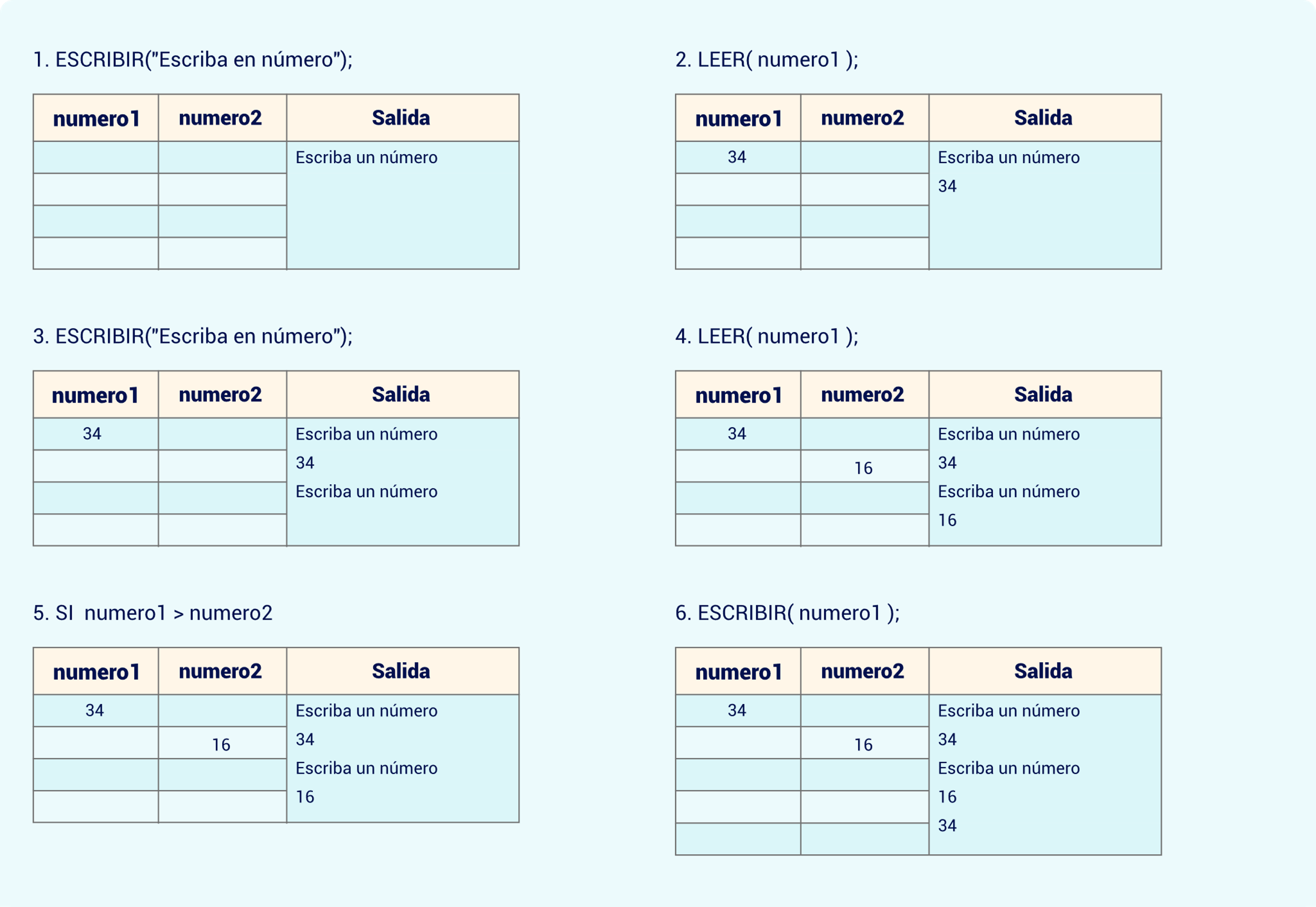
Para el ejemplo donde se realiza un algoritmo que solicita dos números e imprime el mayor de ellos, la estructura se configura así:

* Punto de inicio
* Salida impresa: “Escriba un número”
* Entrada general: Número 1
* Salida impresa: “Escriba un número”
* Entrada general: Número 2
* Decisión: Número 1 > Número 2
* Salida impresa: Número 1 (si cumple la condición)
* Salida impresa: Número 2 (si no cumple la condición)
* Punto final

**Diseño de prueba 1**

* Selección de datos: se usarán para la prueba los números 34 y 16.
* Salida esperada: debe mostrar el número 34.

1. Prueba de escritorio de algoritmo el mayor de dos números



Este diseño de prueba se estructura así:

* ESCRIBIR(“Escriba en número”);
* LEER( número1 );
* ESCRIBIR(“Escriba en número “);
* LEER( número1 );
* SI número1 > número2
* ESCRIBIR( número1 );

En la selección de datos se usarán para la prueba los números 34 y 16. La salida esperada muestra el número 34.

Una prueba de escritorio no es determinante en este ejercicio, es necesario hacerla para los dos casos faltantes, en caso que el segundo número es mayor que el primero y el caso en que ambos números ingresados son iguales.

# Conceptos básicos de programación

Programar es el proceso de crear “software” escribiendo, probando, depurando y dando mantenimiento a las instrucciones del computador en un lenguaje de programación. A este conjunto de instrucciones se les denomina código fuente del “software” creado.

Existen centenares de lenguajes de programación, muchos de ellos fueron creados para dar respuesta a problemas particulares o máquinas específicas. No es de interés ahora discutir todos los tipos de lenguajes existentes, sino, más bien, conocer aquellos que pueden ser interpretados por computadores, “smartphones” o que pueden ser empleados para proveer servicios informáticos a través de internet y su clasificación más general.

Un lenguaje de programación es diferente al lenguaje de códigos que puede entender la máquina (lenguaje de máquina). Los lenguajes de programación, pueden dividirse en dos categorías: lenguajes interpretados o lenguajes compilados, a saber:

1. Lenguajes de programación - **Definición**

| Compilado | Interpretado |
| --- | --- |
| El código creado debe traducirse de manera que el procesador del computador o “smartphone” pueda comprenderlo, a este proceso se le llama compilación. | Un programa escrito en un lenguaje interpretado, necesita de otro programa auxiliar (el intérprete), que traduce los códigos fuentes de un programa y los interpreta para la máquina. |
| El lenguaje compilado necesita que antes de ser ejecutada una compilación se convierta el código fuente escrito a un lenguaje de máquina. | Un lenguaje interpretado es convertido a lenguaje de máquina cada vez que es ejecutado (para ello requiere del intérprete), que traduce las instrucciones. |

1. Lenguajes de programación – **Características**

| Compilado | Interpretado |
| --- | --- |
| Para un lenguaje compilado luego de compilar un programa se debe crear ejecutables para cada uno de los sistemas operativos en los que lo vaya a utilizar. Es decir, un programa compilado en Linux no servirá en Windows o en Mac OS. | Por lo general, en el ciclo de desarrollo de “software” (tiempo entre el momento en que escribe el código y lo prueba) es más corto en un lenguaje interpretado dado que no se necesita realizar el proceso de compilación, cada vez que cambia el código fuente. |
| Cuando se está ejecutando un programa escrito en un lenguaje compilado es mucho más rápido que uno interpretado, dado que se encuentra en código de máquina. | Un lenguaje interpretado para ser ejecutado, debe tener instalado el interpretador. |
| Los lenguajes compilados están optimizados para el momento de la ejecución, pero esto significa más trabajo para el programador (compilarlo). | El código fuente en un lenguaje interpretado está optimizado para hacerle la vida más fácil al programador, aunque esto signifique una carga adicional de procesamiento para la máquina. |

JavaScript, al igual que cualquier otro lenguaje de programación, tiene algunas características especiales: sintaxis, modelo de datos, etc. JavaScript es un lenguaje esencial para cualquier principiante de programación y también para quienes ya saben programar, debido a que las diferencias con otros lenguajes de programación son numerosas siendo un lenguaje necesario para el desarrollo de aplicaciones web y móviles hoy en día.

En adelante, JavaScript es el lenguaje de programación abordado y es un lenguaje interpretado que inicialmente fue creado para que los navegadores lo interpreten y hoy en día muchos dispositivos tienen el intérprete que permite correr programas escritos en él por lo que será más sencillo el trabajo tanto para la máquina como para el programador.

# Entornos de codificación e instalación JavaScript

En su mayoría los navegadores (Chrome, Mozilla, Edge) tienen intérprete de JavaScript y adicional se puede instalar el intérprete para el sistema operativo de preferencia (Framebits, 2020).

Dado lo anterior, existen herramientas, manuales o videos que se emplean para el desarrollo de programas en JavaScript y a continuación se comparten recursos de acceso y material audiovisual de apoyo de cada herramienta, tal como se indica en la siguiente tabla:

1. Entornos de codificación

| Herramienta | Recurso web | Uso y/o instalación |
| --- | --- | --- |
| Editor y compilador en línea PlayCode | <https://playcode.io/> | Harlley Oliveira (2017). PLAYCODE MAKERS: O QUE É JAVASCRIPT? <https://www.youtube.com/watch?v=1mjvPBHiYnY> |
| Intérprete de JavaScript: Node js | <https://nodejs.org/es/> | Framebits. (19 de enero. 2020). Descargar e instalar Node Js en Windows 10. <https://youtu.be/v0x1Ku5Tgac> |
| Editor de código Visual Studio Code | <https://code.visualstudio.com/> | Code Compadre. (30 de junio. 2020). How to Download and Install Visual Studio Code for Windows 10. <https://youtu.be/KpzkPlh_HsU> |
| El navegador: Google Chrome | <https://www.google.com/chrome/> | Choque, R. C. (2020, May 10). Cómo usar la consola de google chrome para Javascript. Youtube. Retrieved from <https://www.youtube.com/watch?v=Hf3n-p3VYx4> |

# Sintaxis del lenguaje JavaScript

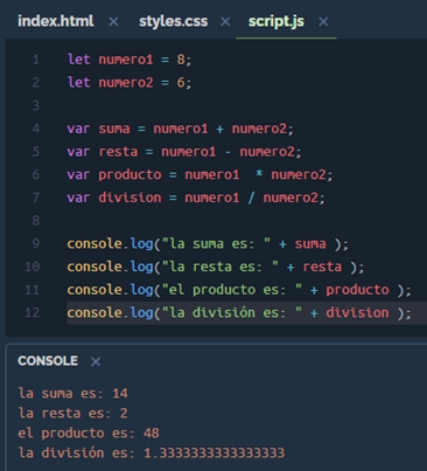
En adelante, se presentan las características básicas de la sintaxis de JavaScript, codificando algoritmos básicos, para ejemplificar la estructura propia del lenguaje de programación, sin embargo, se recomienda que siempre que se esté codificando algoritmos, se pueda tener a disposición la documentación técnica oficial del lenguaje y material complementario propuesto.

## Funciones y operaciones matemáticas

En JavaScript se pueden utilizar muchas funciones matemáticas; por ejemplo, un algoritmo que toma dos números (8 y 6) y le aplica las 4 operaciones básicas y en el editor de código “online” PlayCode (disponible en <https://playcode.io/>) se revisan las características de la sintaxis del JavaScript y a la vez se conocen las operaciones matemáticas del lenguaje.

Ingresar en la ruta del navegador a <https://playcode.io/new>, escribir el código de la línea 1 a la 12 en la pestaña script.js como se muestra en la siguiente figura:

1. Operaciones aritméticas con Javascript



El código de la línea uno a la doce es:

* 1 let numero1 = 8;
* 1 let numero2 = 6;
* 3
* 4 var suma = numero1 + numero2;
* 5 var resta = numero1 – numero2;
* 6 var producto = numero1 \* numero2;
* 7 var división = numero1 / numero2;
* 8
* 9 console.log(“la suma es: “ + suma );
* 10 console.log(“la resta es: “ + resta );
* 11 console.log(“el producto es: “ + producto );
* 12 console.log(“la division es: “ + division );

CONSOLE X

* la suma es: 14
* la resta es: 2
* el producto es: 48
* la división es: 1.3333333333333333

**Explicación**:

* En las líneas 1 y 2 aparece la palabra reservada let que sirve para indicar que se define una variable accesible en el contexto donde se crea (cuando se vea programación modular quedará más claro esto del contexto).
* Las variables numero1 y numero2 se les asigna el valor correspondiente. Es de observar que se definen y se inicializa la variable en una misma línea, esto es una característica del lenguaje.
* Luego en las líneas 4 al 7 se crean o se definen unas variables con la palabra reservada var, a diferencia de la palabra let estas variables se pueden ver desde cualquier parte o módulo del script.
* Una vez se crean las variables con identificadores suma, resta, producto, división, también son inicializados sus valores empleando las operaciones matemáticas respectivas.
* Una de las formas que JavaScript permite mostrar los resultados es con la función log del objeto console y se accede console.log() (con el operador punto .) y entre paréntesis lo que se desea mostrar en la pantalla de la consola.

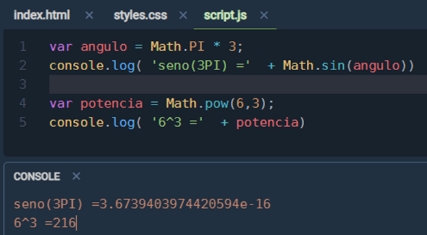
**Validación**:

* Dentro de los paréntesis “la suma es: "+suma, aparece la operación más + usado para sumar dos números antes, solo que acá no hay dos números sino una cadena de texto y un número, pero en este caso lo que hace el operador + es concatenar o juntar el texto “la suma es:” y el valor de la variable suma. A esta característica de los operadores se les llama sobrecarga de operadores, porque según el tipo de datos suma y según otra concatena.

Cuando las operaciones matemáticas son más complejas como la raíz cuadrada, la función exponencial, las funciones trigonométricas, existe un objeto que agrupa estas funciones se llama Math, puede ver la documentación técnica de la clase completa en Math (MDN, 2021b), a continuación, un ejemplo de su uso.

Se debe crear un algoritmo que muestre el seno del valor PI (3,1416) multiplicado por 3 la función, y también que nos eleve 6 a la tercera potencia mostrando su resultado:

1. Uso del objeto Math



La imagen muestra la inscripción del código así:

* 1 var angulo = Math.PI \* 3;
* 2 console.log( ´seno(3PI) “´ + Math.sin(angulo))
* 3
* 4 var potencia = Math.pow(6,3) ;
* 5 console.log( ´6>3 =´ + potencia)

CONSOLE X

* seno(3PI) =3.6739403974420594e-16

6>3 =216

Se debe escribir el código fuente y obtener los mismos resultados en la consola, de esta forma se familiariza con la herramienta de trabajo.

Conozca el listado de funciones de la clase Math, descargando el contenido del PDF denominado **Métodos del objeto Math**, que se encuentra en la carpeta Anexos.

Todas las funciones matemáticas se pueden combinar con otras operaciones aritméticas o emplear operadores de evaluación de condición y previo a ello es preciso conocer los diferentes tipos de datos que existen.

## Tipos de datos, operadores y orden de evaluación

Existen muchos tipos de operadores para JavaScript, cuya función es realizar una operación entre dos o más valores contenidos en variables, constantes o acumuladores:

1. **Tipos de datos**.

JavaScript tiene los siguientes tipos de operadores (se describirán los principales para aprender a programar).

Según sea el tipo que las variables o constantes que almacenan, se pueden clasificar en seis (6) tipos de datos primitivos; la referencia del lenguaje dice que son:

* **Undefined**: indeterminado o indefinido.
* **Boolean**: tipo booleano los valores posibles son true o false.
* **Number**: números enteros, o decimales.
* **String**: cadenas de texto.
* **BigInt**: números enteros grandes.
* **Symbol**: referencia a otros datos.

Otros tipos de datos más complejos o abstractos de datos son:

* **Null**: tipo primitivo especial que tiene un uso adicional para su valor: si el objeto no se hereda, se muestra null.
* **Object**: tipo estructural especial que no es de datos, pero para cualquier instancia de objeto construido que también se utiliza como estructuras de datos (“new Object”, “new Array”, “new Map”, “new Set”, “new WeakMap”, “new WeakSet”, “new Date” y casi todo lo hecho con la palabra clave “new”).
* **Function**: una estructura sin datos, aunque también responde al operador **t**.

Cuando se declaran variables se debe considerar los siguientes tipos, ya se han usado dos (2) de ellas:

* **var**: declara una variable, opcionalmente la inicia a un valor.
* **let**: declara una variable local con ámbito de bloque, opcionalmente la inicia a un valor.
* **const**: declara un nombre de constante de solo lectura y ámbito de bloque.

Amplíe sus conocimientos y la información sobre los tipos de datos y estructuras en JavaScript, explorando el contenido del siguiente enlace: <https://developer.mozilla.org/es/docs/Web/JavaScript/Guide/Grammar_and_Types#tipos_de_datos>

Una variable en JavaScript puede tener cualquier tipo de dato, entero, real, cadena de texto etc. JavaScript no es un lenguaje fuertemente tipado de manera que queda en el programador la responsabilidad de saber qué tipo de dato está almacenando en cada variable.

1. **Operadores**.

Se considera una expresión, aunque de un tipo distinto, el asignar un valor a una variable. Para todas estas operaciones se emplean los denominados “operadores”. Se listan a continuación algunos de los operadores que se utilizan con mucha frecuencia:

* **Operadores de asignación**. Un operador de asignación asigna un valor a la variable a la izquierda basándose en el valor de su operando derecho. El operador de asignación más simple es igual (=), que asigna el valor de su operando derecho a su operando izquierdo. Es decir, x = y asigna el valor de y a x. A continuación, se presenta una lista de operadores más usados.

También hay operadores de asignación compuestos que son una abreviatura de las operaciones enumeradas en la siguiente tabla:

| Nombre | Operador abreviado | Significado |
| --- | --- | --- |
| Asignación | x = y | x = y |
| Asignación de adición | x += y | x = x + y |
| Asignación de resta | x -= y | x = x - y |
| Asignación de multiplicación | x \*= y | x = x \* y |
| Asignación de división | x /= y | x = x / y |
| Asignación de residuo | x %= y | x = x % y |
| Asignación de exponenciación | x \*\*= y | x = x \*\* y |

* **Operadores de comparación**. Son operadores cuya función es comparar dos expresiones y como resultado de la comparación devuelven un valor falso o verdadero (booleano), que representa la relación de sus valores comparados. Existen operadores para comparar valores numéricos, pero también operadores para comparar cadenas y operadores para comparar otros tipos de datos:

| Operador | Descripción | Ejemplos que devuelven true |
| --- | --- | --- |
| Igual (==) | Devuelve true si los operandos son iguales. | 3 == var1  "3" == var1  3 == '3' |
| No es igual (!=) | Devuelve true si los operandos no son iguales. | var1 != 4  var2 != "3" |
| Estrictamente igual (===) | Devuelve true si los operandos son iguales y del mismo tipo. Consulta también Object.is y similitud en JS. | 3 === var1 |
| Desigualdad estricta (!==) | Devuelve true si los operandos son del mismo tipo, pero no iguales, o son de diferente tipo. | var1 !== "3"  3 !== '3' |
| Mayor que (>) | Devuelve true si el operando izquierdo es mayor que el operando derecho. | var2 > var1  "12" > 2 |
| Mayor o igual que (>=) | Devuelve true si el operando izquierdo es mayor o igual que el operando derecho. | var2 >= var1  var1 >= 3 |
| Menor que (<) | Devuelve true si el operando izquierdo es menor que el operando derecho. | var1 < var2  "2" < 12 |
| Menor o igual (<=) | Devuelve true si el operando izquierdo es menor o igual que el operando derecho. | var1 <= var2  var2 <= 5 |

* **Operadores de aritméticos**. Los operadores aritméticos toman valores numéricos (ya sean literales o variables) como sus operandos y devuelve un solo valor numérico. Los operadores aritméticos básicos son suma (+), resta (-), multiplicación (\*) y división (/). Estos operadores funcionan como en la mayoría de los otros lenguajes de programación cuando se usan con números de punto flotante, teniendo en cuenta que la división por cero produce un error:

| Operador | Descripción | Ejemplo |
| --- | --- | --- |
| Residuo (%) | Operador binario. Devuelve el resto entero de dividir los dos operandos. | 12 % 5 devuelve 2 |
| Incremento (++) | Operador unario. Agrega uno a su operando. Si se usa como operador prefijo (++x), devuelve el valor de su operando después de agregar uno; si se usa como operador sufijo (x++), devuelve el valor de su operando antes de agregar uno. | Si x es 3, ++x establece x en 4 y devuelve 4, mientras que x++ devuelve 3 y, solo entonces, establece x en 4. |
| Decremento (--) | Operador unario. Resta uno de su operando. El valor de retorno es análogo al del operador de incremento. | Si x es 3, entonces --x establece x en 2 y devuelve 2, mientras que x-- devuelve 3 y, solo entonces, establece x en 2. |
| Negación unaria (-) | Operador unario. Devuelve la negación de su operando. | Si x es 3, entonces -x devuelve -3. |
| Positivo unario (+) | Operador unario. Intenta convertir el operando en un número, si aún no lo es. | +"3" devuelve 3. +true devuelve 1. |
| Operador de exponenciación (\*\*) | Calcula la base a la potencia de exponente, es decir, base exponente. | 2 \*\* 3 retorna 8.  10 \*\* -1 retorna 0.1. |

* **Operadores lógicos**. Con los operadores lógicos, se pueden crear condiciones compuestas, por ejemplo, cuando se deben cumplir dos o más condiciones para elegir las operaciones a ejecutar; además, se pueden describir estas combinaciones de condiciones:

| Operador | Uso | Descripción |
| --- | --- | --- |
| AND Lógico (&&) | expr1 && expr2 | Devuelve expr1 si se puede convertir a false; de lo contrario, devuelve expr2. Por lo tanto, cuando se usa con valores booleanos, && devuelve true si ambos operandos son true; de lo contrario, devuelve false. |
| OR lógico (||) | expr1 || expr2 | Devuelve expr1 si se puede convertir a true; de lo contrario, devuelve expr2. Por lo tanto, cuando se usa con valores booleanos, || devuelve true si alguno de los operandos es true; si ambos son falsos, devuelve false. |
| NOT lógico (!) | !expr | Devuelve false si su único operando se puede convertir a true; de lo contrario, devuelve true. |

1. **Orden de evaluación de operadores**.

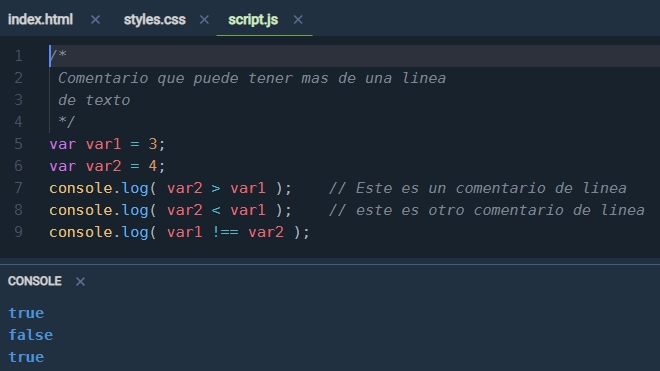
Es importante recordar que los computadores ejecutan los operadores en un orden establecido. El siguiente es el orden (jerarquía), de acuerdo con los expuestos anteriormente, y deben conservar el orden de precedencia de operadores:

* Paréntesis (se ejecutan primero los más internos).
* Signo (-2), si un valor es positivo o negativo.
* Potencias (^) y raíces y otras funciones (Math.sqrt()); productos y divisiones (\* y /), en este mismo orden.
* Sumas y Restas (+ y -).
* Concatenación (+).
* Relacionales (=, <, >).
* Negación (no).
* Conjunción (y).
* Disyunción (o).

## Expresiones y comentarios

JavaScript permite poner comentarios en el código fuente, como la mayoría de los lenguajes de programación y existen dos tipos de comentarios en línea que comienzan con una doble barra: //, y los comentarios multilínea, que comienzan con /\* y terminan con \*/ tal como se observa en la siguiente figura:

1. Tipos de comentarios



Los comentarios son importantes porque le dan mantenibilidad al programa, es decir que otro programador, o el mismo, tiempo después puede revisar el código y apoyarse en los comentarios para saber qué hace el algoritmo y como lo hace.

En adelante se procura que exista al menos un comentario en cada código fuente empleado para ejemplificar los elementos constitutivos del lenguaje de programación JavaScript y puede hacer lo mismo durante la creación de los diferentes códigos.

## Estructuras de selección

La estructura de selección se necesita cuando el código de tu programa ejecuta uno de varios resultados posibles, basado en el valor de una condición.

**Ejemplo**: un aprendiz aprueba un examen cuando la calificación de este es mayor o igual a 3. Elaborar un programa en JavaScript donde, dada una calificación, aplique el criterio de aprobación e imprima “Aprobado” o “Reprobado”, según sea el caso.

Como resultado del análisis a este problema tenemos el diseño del siguiente algoritmo:

* Algoritmo: calificación del usuario
* VAR
* REAL Calificacion
* INICIO
* Calificación – 4;
* SI (calificación >= 3)
  + - ESCRIBIR (“Aprobo” ) ;
* SINO
  + - ESCRIBIR ( “Reprobo” ) ;
* FINSI
* FIN

El mismo algoritmo se puede visualizar, pero ya en lenguaje de programación:

1. Algoritmo calificaciones en Java Script



La imagen muestra la inscripción del código así:

* 1 /\*\*
* 2 | \* Algoritmo calificaciones
* 3 | \*/
* 4 const calificación = 4;
* 5
* 6 if (calificación >= 3 ({
* 7 | console.log(´Aprobó´);
* 8 }else{
* 9 | console.Llog(´Reprobó´);
* 10 }

CONSOLE X

* Aprobó

Como la línea 1 empieza con /\* quiere decir que toda continuación será un comentario hasta que aparezca los caracteres \*/ (línea 3), en él se debe escribir que hace el algoritmo y otra información que se verá más adelante.

En la línea 4 se define un valor constante (es decir no se modificará más en todo el algoritmo donde definimos) se ha puesto por identificador la palabra calificacion.

**Nota importante**: una palabra reservada es una palabra que no se puede utilizar como identificador de algún dato.

* En la línea 6 ocurren varias cosas que se deben tener en cuenta:
* Se usa la palabra reservada if y entre paréntesis la condición que la se quiere comparar.
* Se emplea el símbolo corchete abierto { se llama abrir corchete.
* En la línea 7 se envía la instrucción de que imprima el resultado de que aprobó.
* En la línea 8 se cierra un bloque de código con el símbolo corchete cerrado **}**, esto quiere decir que hasta este punto termina el bloque. A todo lo que esté entre estos corchetes se le llamará CONTEXTO, en el primer contexto están agrupadas las operaciones que se ejecutarán si la condición calificacion >= 3 es verdadera.
* También en la línea 8 se observa la palabra reservada else, que sirve para indicarle al intérprete de JavaScript que se creará un bloque o contexto para cuando la condición evaluada por el **if** no se cumpla, y enseguida se abre el contexto con el carácter corchete abierto **{**.
* Luego, en la línea 9 se ha puesto la operación de escribir en la consola que ha reprobado.
* Para finalizar el contexto, se cierra con el carácter corchete cerrado **}**.

Ahora se debe escribir el programa y cambiar los valores de la constante para validar que el algoritmo está bien diseñado.

De la misma forma, se puede realizar el algoritmo que, dada una edad en años, evalúe e imprima si es mayor de edad:

1. Escribir en pantalla si es o no mayor de edad



La imagen muestra la inscripción del código así:

* 1 /\*\*
* 2 | \* Algoritmo saber si es o no
* 3 | \* mayor de edad
* 4 | \*/
* 5 const edad = 41;
* 6
* 7 if (edad >= 18 ){
* 8 | console.log(´Es mayor de edad´);
* 9 }

CONSOLE X

* Es mayor de edad

Si no se cumple la condición de mayoría de edad, a esta estructura se denomina CONDICIONAL SIMPLE.

El flujo de ejecución del algoritmo anterior es básico y puede haber otros mucho más complejos como las estructuras de repetición o iterativas que también tienen una sintaxis particular.

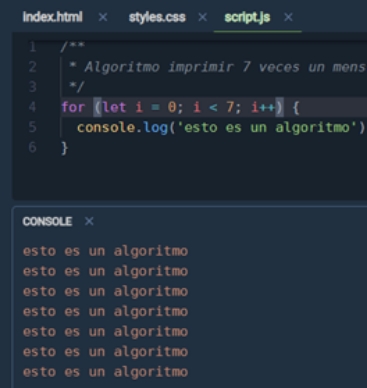
## Estructuras de repetición

Las estructuras de repetición permiten repetir un bloque de instrucciones, determinado o no, un número de veces. A continuación, se ejemplifican los más comunes (FOR, WHILE) que se pueden codificar también en la sintaxis de JavaScript.

### Estructura de repetición FOR

**Ejemplo 1**: escribir un procedimiento que muestre siete (7) veces en pantalla la frase “Esto es un algoritmo”. Mostrar en pantalla la salida del ejercicio.

1. Imprimir 7 veces un mensaje



La imagen muestra el código así:

* 1 /\*\*
* 2 | \* Algoritmo imprimir 7 veces un mesaje
* 3 | \*/
* 4 for (let i = 0; i < 7; i++) {
* 5 | console.log(´esto es un algoritmo´);
* 6 }

CONSOLE X

* esto es un algoritmo

esto es un algoritmo

esto es un algoritmo

esto es un algoritmo

esto es un algoritmo

esto es un algoritmo

esto es un algoritmo

Como se puede observar en la línea 4, se ha usado la palabra reservada **for** y dentro del paréntesis aparecen tres (3) instrucciones (separadas por punto y coma).

La primera instrucción **let i = 0**; es la creación e inicialización de una variable con identificador **i**, como se usó la palabra reservada let se está indicado que la variable i solo podrá ser accedida dentro del contexto del **for**.

Luego se ve una condición **i < 7**; esta condición significa que mientras que **i** tenga valores de 0 a 6 debe ejecutar el bloque o contexto definido en el **for** (desde el corchete abierto hasta el corchete cerrado).

Una vez ha terminado de ejecutar el contexto (línea 6) se ejecutará la tercera sentencia de la línea 7 o sea **i++**, esta línea es equivalente a **i = i + 1**; incrementar en 1 la variable **i**, luego de incrementarla procede a evaluar nuevamente la condición y si se cumple volverá a ejecutar las instrucciones que están dentro el contexto (lo que está dentro de paréntesis).

Es de notar que los valores toman la variable **i** son 0, 1, 2, 3, 4, 5, 6. En total 7 valores el número de veces que se necesita que se repita el bucle.

**Nota importante**: el carácter punto y coma se usa para decirle al computador, o intérprete, que ha finalizado una instrucción. La estructura de repetición PARA (FOR) se usa cuando se sabe cuántas veces se debe repetir un grupo de instrucciones.

**Ejemplo 2**: se necesita elaborar un algoritmo en JavaScript que, dado un número entero, sume todos los números naturales que hay hasta ese número. Por ejemplo, si el usuario digita 3, el programa debe sumar: 1 + 2 + 3, si el usuario digita 5 el programa debe sumar 1+2+3+4+5. Al finalizar, debe imprimir el resultado.

1. Sumar los números desde 1 hasta un número



* 1 /\*\*
* 2 | \* sumar los números que de de 1 hasta el
* 3 | \* valor dado
* 4 | \*/
* 5 |
* 6 const cantidad = 5;
* 7 var suma = 0;
* 8
* 9 for (let i = 1; i <= cantidad; i++) {
* 10 | suma += i; // esto es igual suma = suma +i;
* 11 }
* 12 console.log(suma);

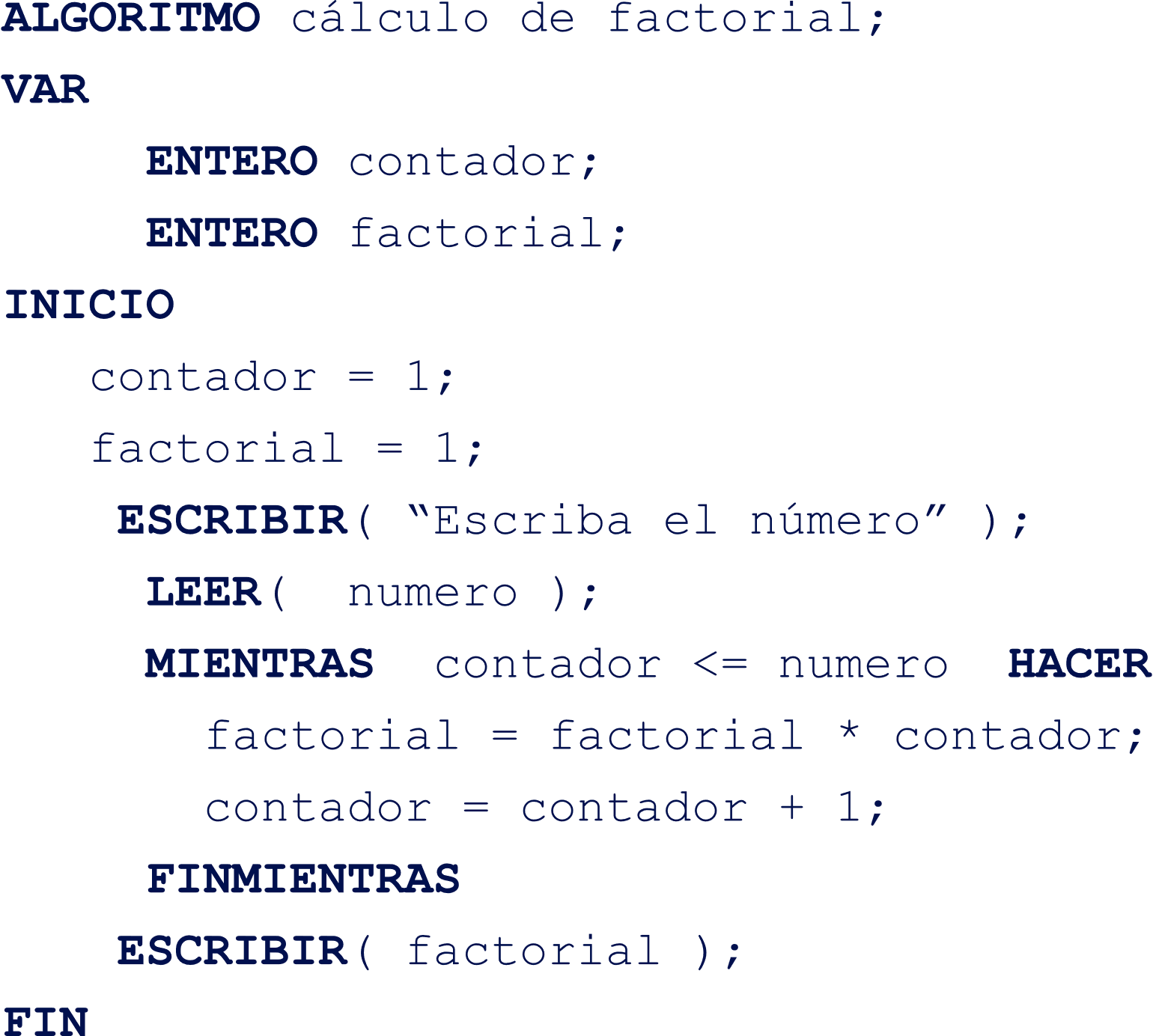
CONSOLE X

* 15

Se decidió una constante con valor 5 y una variable que será para el algoritmo un acumulador, donde se almacenarán las sumas. También, los valores de **i** van desde 1 hasta exactamente el valor que tiene la constante (es decir 5).

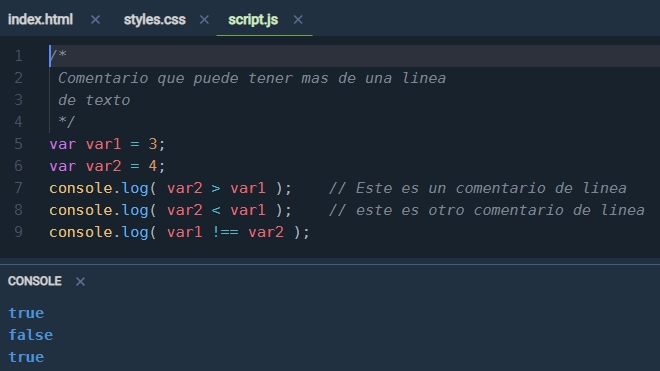
### Estructura de repetición WHILE

A través del ejemplo del cálculo de factorial se explicará el uso y comportamiento de la sentencia while. Recordando el algoritmo diseñado:



De acuerdo con el anterior algoritmo, el resultante para el cálculo de factorial es:

1. Seudocódigo algoritmo cálculo de factorial



* 1 /\*
* 2 | comentario que puede tener más de una linea
* 3 | de texto
* 4 | \*/
* 5 var var1 = 3;
* 6 var var2 = 4;
* 7 console.log( var 2 > var1 ); // este es un comentario de linea
* 8 console.log( var 2 > var1 ); // este es otro comentario de linea
* 9 console.log( var1 ! == var2 );

CONSOLE X

* True
* False
* True

Es ideal poder transcribir el código usando <https://playcode.io>, para lo cual se debe escribir la condición del while, es decir, **contador <= numero** de último, para que el sistema **PlayCode** no entre a un ciclo que nunca termina. Una vez funcione correctamente, se propone el código que está entre comentarios (//) para practicar las dos formas de sintaxis.

Es de recordar que las estructuras de repetición básicas y su sintaxis pueden pasar su aplicación más común para recorrer la estructura de datos.

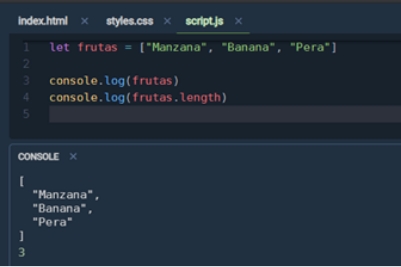
## Estructura de datos

Las estructuras de datos también conocidas como arreglos, son importantes en el desarrollo de algunos algoritmos son obligatorias a la hora de hacer aplicaciones web o móviles; por ello, se explica su sintaxis en JavaScript para los vectores, matrices y registros.

### Vectores

Los vectores se definen en JavaScript como un tipo de dato con métodos, y atributos que lo definen. Es ideal analizar lo siguiente:

1. Definición de un vector



* 1 let frutas = [“manzana”, “banana”, “pera”]
* 2
* 3 console.log(frutas)
* 4 console.log(frutas.length)
* 5
* CONSOLE X
* [
* “Manzana” ,
* “Banana” ,
* “Pera”
* ]
* 3

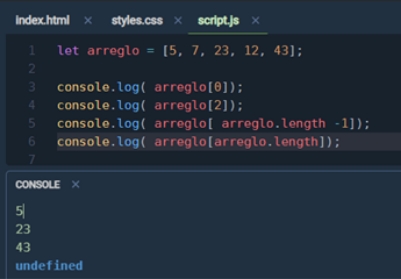
En la línea 1 se ve cómo se define un vector con identificador frutas, de tres (3) elementos, cada uno de ellos es una cadena de texto con una fruta.

En la línea 3 se manda imprimir en la consola todo el vector con su contenido, como se ve en la consola.

En la línea 4 se le llama al atributo **frutas.length** que contiene el tamaño del vector (es decir 3).

### Acceder a los elementos de un vector

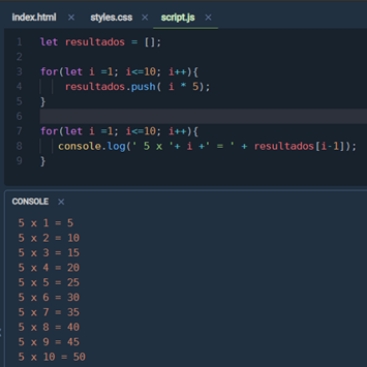
1. Indexar un vector



Los índices de los vectores se comienzan en cero, en otras palabras, el índice del primer elemento de un vector es 0, y el del último elemento es igual al valor de la propiedad length del “array” restándole 1. Si se utiliza un número de índice no válido, se obtendrá: **undefined**.

Ahora, se debe crear un programa que guarde los resultados de la tabla de 5 en un vector, con estos resultados recorrer el vector e imprimir la tabla del 5:

1. Crear la tabla del 5



Es ideal revisar la lista de los métodos más comunes en los arreglos:

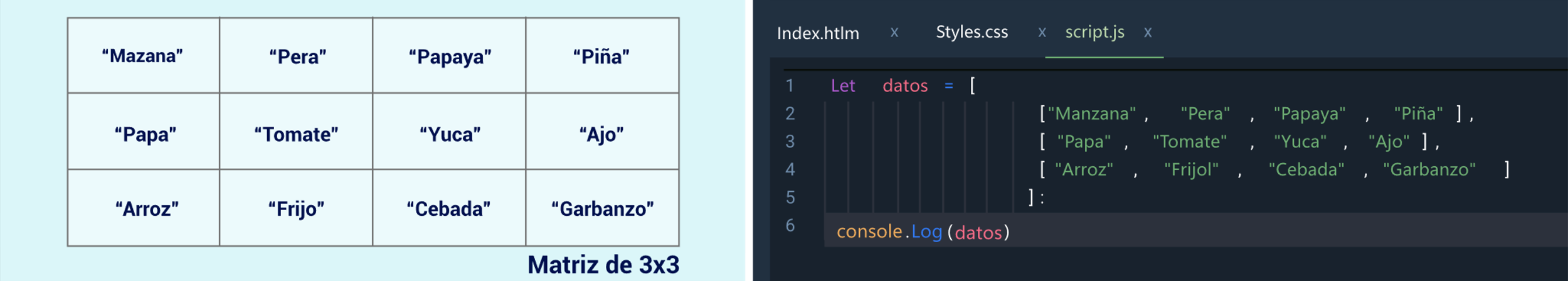
1. Funciones comunes de un vector

| Función | Operador abreviado |
| --- | --- |
| indexOf() | Devuelve el índice del primer elemento del “array” que sea igual a elemento Buscado, o -1 si no existe. |
| join() | Concatena en un string todos los elementos de un “array”. |
| push() | Añade uno o más elementos al final de un “array” y devuelve el nuevo valor de su propiedad length. |
| pop() | Elimina el último elemento de un “array”, y devuelve dicho elemento. |
| sort() | Ordena los elementos de un “array”, modificando este, y devuelve el “array” ordenado. |
| shift() | Elimina el primer elemento de un “array”, y devuelve dicho elemento. |

### Matrices

Las matrices se pueden ver lógicamente como un vector y cada uno de sus elementos es otro vector, así la representación de la matriz queda codificada como muestra la figura siguiente:

1. Matriz de 3X4



La imagen de la matriz codificada se muestra como un vector, donde cada uno de sus elementos es otro vector:

* Manzana, pera, papaya, piña
* Papa, tomate, yuca, ajo
* Arroz, frijol, Cebada, Garbanzo

Cada fila representa un registro y cada dato de un registro puede tener un tipo de dato diferente.

### Registros

Hasta ahora se ha visto cómo un arreglo es la colección de datos del mismo tipo, pero un registro es una colección de datos de diferente tipo que se relacionan entre sí:

1. Registros

| Nombre | Correo | Edad | Saldo |
| --- | --- | --- | --- |
| Juan | juan@sindato.com | 19 | 36.234 |
| Luis | luis@sindato.com | 18 | 23.234 |
| Andrea | andrea@sindato.com | 22 | 0 |
| Pedro | pedro@sindato.com | 12 | 65.234 |
| María | maria@sindato.com | 16 | 123 |
|  |  |  |  |

## Estructuras de salto

Estas estructuras son instrucciones que permiten romper la ejecución natural o secuencial de los programas, permitiendo que se salte a otro punto de la ejecución del programa, estas instrucciones tienen las palabras reservadas: continue, break y return:

1. **Sentencia continue**: la sentencia continue se salta a la siguiente iteración del ciclo for o while como se presenta en el siguiente ejemplo y teniendo en cuenta los resultados.



Cuando la variable i tiene el valor 3, se ejecuta la sentencia continue se salta la sentencia donde debe imprimir el valor de 3, por ese motivo no aparece en la consola.

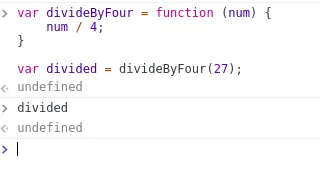
1. **Sentencia break**: por otro lado, la sentencia break, detiene la ejecución del ciclo independientemente de cuántas veces este configurando el ciclo.

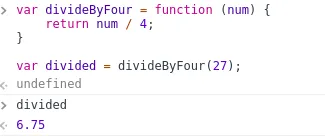


En la anterior imagen, cuando i tiene el valor de 3 ya no se sigue ejecutando más el ciclo, aunque esté programado para ejecutarse 5 veces, solo ejecuta 3 iteraciones y, ya que en el tercero luego de evaluar la condición i == 3 ejecuta la sentencia break haciendo que termine todo.

1. **Sentencia return**: sirve para terminar la ejecución de un bloque de instrucciones, se usa mucho en programación modular cuando se quiere retornar un resultado, se usará más adelante.

Por ejemplo, se tiene una función que divide un número entre 4 y se ha creado para obtener resultados. En el primer caso se le pasa 27 para que lo divida entre 4 pero como no hay return dentro, devuelve undefined. No accede al número porque el número solo vive dentro de la función y no hay nada que dé el valor. Si se le agrega return, efectivamente se obtiene el resultado de la división.



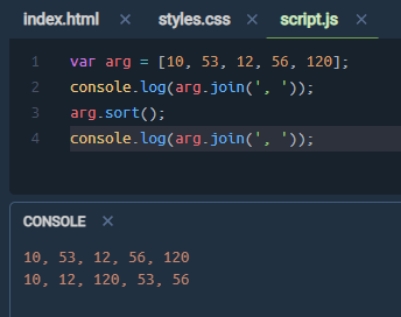


## Métodos de ordenamiento y búsqueda

Existen varias técnicas de ordenamiento de vectores las cuales son importantes para realizar búsquedas y, al igual que en la vida real, es más fácil buscar en un lugar que esté ordenado que en uno que no lo esté, como JavaScript está diseñado para pequeños algoritmos que resuelven pequeños problemas y convertirlos en servicios, el uso de esas técnicas no tiene mucha aplicabilidad, pues la diferencia en tiempos computacionales de ejecución es considerable solo cuando se manejan volúmenes de datos grandes, mientras que para volúmenes de datos cortos o pequeños los tiempos computacionales no tienen diferencias comparables. Es por eso que, si en JavaScript el volumen de datos en los arreglos no será grande, no es necesario la aplicación de esas técnicas.

### A. Ordenamiento de cadenas de texto

1. Ordenamiento de números como cadenas de texto

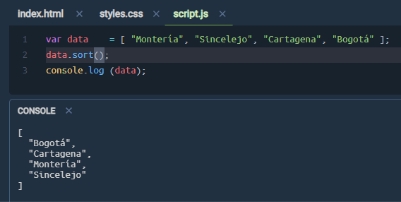


Se ordenará un vector de mayor a menor valor y de acuerdo con los datos que tiene, se debe tener en cuenta que el arreglo se modifica al ordenarlo, devolviendo la misma matriz ordenada, pero no una nueva. Esto es importante, si se quiere mantener inmutable el vector y obtener otro ordenado, lo que se tendría que hacer es una copia del arreglo antes de ordenarlo.

Por defecto el método **Array.sort()** ordena los elementos del “array” como si fueran cadenas inclusive si los datos son de tipo entero.

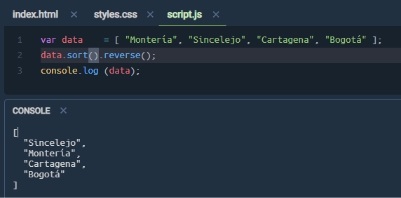
La imagen muestra un arreglo de datos numéricos a ordenar en la línea 1, en la línea 2 se muestra en consola el vector, se ha usado la función **join(‘, ’)** que sirve para concatenar los datos del vector o arreglo poniendo entre ellos una coma (, ) y espacio. Luego se aplica la función **sort()** en la tercera línea de código y, posteriormente, se muestra el vector ya ordenado. Como se observa, el resultado en consola es un ordenamiento como si los elementos fueran cadenas de texto.

1. Ordenamiento de cadenas de texto



El ejemplo de la figura anterior muestra un ordenamiento de cadenas de texto.

1. Invertir el orden de los elementos



También es posible ordenar el arreglo de forma descendente con el método Array.reverse().

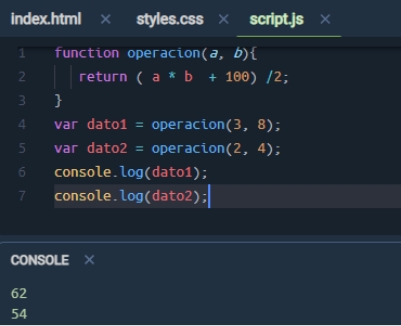
La función Array.reverse() no ordena los elementos, simplemente toma los elementos y les invierte el orden, es decir, el primero pasa a ser el último, el segundo, el penúltimo y así hasta que el último pasa al primer puesto, es por esto que primero se ordena en orden ascendente y luego se invierte para lograr el orden descendente.

### B. Ordenamiento de datos numéricos

El ordenamiento con la función Array.sort() se hace considerando los elementos del arreglo como cadenas de texto. Antes de ver cómo se ordenan con datos numéricos se explicará que es una función en JavaScript.

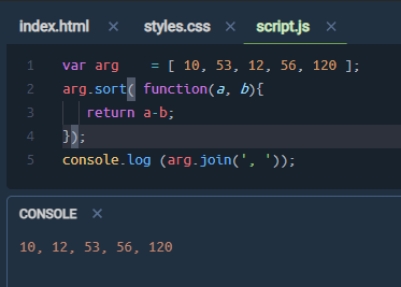
**¿Qué es una función?**

Las funciones son uno de los bloques de construcción fundamentales en JavaScript.Una función en JavaScript es similar a un procedimiento o “un conjunto de instrucciones que realiza una tarea o calcula un valor, pero para que un procedimiento califique como función, debe tomar alguna entrada y devolver una salida donde hay alguna relación obvia entre la entrada y la salida”. Para usar una función, se debe definir en algún lugar del ámbito en el que se desea llamarla (MDN, 2021d).



Como se puede observar en la anterior imagen, se usa la palabra reservada “function” y luego se escoge un identificador para la función que para este caso es **operacion**, luego entre paréntesis se seleccionan nombres para las variables de entrada (en el ejemplo a, y b), a continuación, se define un contexto (espacio de código entre llaves {}) para indicar cuáles son las instrucciones que ejecutará la función.

Para ordenar un vector con datos numéricos se debe pasar como argumento una función al método Array.sort(), esta función indica qué operación debe hacerse sobre los datos a comparar entre dos datos del arreglo. Para comparar si un número es mayor que otros, es muy común usar la resta porque si se resta un número solo existen tres tipos de respuestas.



Cuando a un número **a** le restamos un número **b** (a - b):

* si el resultado es negativo es porque b es mayor que a.
* si el resultado es positivo es porque a es mayor que b.
* si el resultado es cero es porque a y b son iguales.

### Ordenamiento de registros

Suponiendo que se tienen los registros de la siguiente tabla, se desea ordenarlos de mayor edad a menor edad.

1. Ordenamiento de registros

| Usuario | Edad | Rol |
| --- | --- | --- |
| Mariela | 31 | SAC |
| Eduardo | 30 | SEO |
| Andrés | 34 | “Project Manager” |

### Buscar un elemento

El método **find()** devuelve el valor del primer elemento del “array” que cumple la función de prueba proporcionada Array.prototype.find() - JavaScript, MDN, 2021.

Se usa el identificar elemento para referenciar a la variable que se va a aplicar el criterio que en el ejemplo es encontrar el primer elemento que sea mayor que 10.

En ocasiones, es clave que, en lugar de retornar un solo valor, el resultado sea más de uno, por ejemplo, si en el arreglo se retornan los valores que son mayores que 10, debe retornar otro vector con los valores resultados. Para esto existe la función Array.filter(), que crea un nuevo arreglo con los datos que cumplan con la condición.

Es de interés notar que la consola solo retornó los datos que cumplen la condición que el elemento es mayor que 10.

En la medida que se va avanzando en el conocimiento de la sintaxis del lenguaje, el código fuente se va enriqueciendo de elementos y por tanto de complejidad, esto hace que sea necesario conocer herramientas para la depuración de fallas y corregir los fallos en la redacción del código.

# Depuración y fallas de sintaxis

La principal funcionalidad que se requiere en la depuración del código, es mostrar salidas parciales de datos de depuración del estado de las variables, o tal vez se tenga la necesidad de mostrar que se está ejecutando determinada línea de código. Por lo general, la salida de JavaScript está dirigida hacia un navegador o hacia un flujo de datos de un servidor web, dejando la consola para realizar estas depuraciones.

Hasta ahora se ha usado la consola para mostrar datos de depuración cuando se emplea console.log(), que es un método usado para mostrar mensajes de confirmación de operaciones o mitificaciones de éxito en las operaciones del algoritmo. Pero también existen más:

* console.log() para enviar y registrar mensajes generales.
* console.dir() para registrar un objeto y visualizar sus propiedades.
* console.warn() para registrar mensajes de alerta.
* console.error() para registrar mensajes de error.

Puede revisar el procedimiento de despliegue y uso de consola para el navegador Google Chrome en el siguiente enlace: <https://www.youtube.com/watch?v=Hf3n-p3VYx4&feature=youtu.be>

# Fallas de lógica

Para realizar las depuraciones y correcciones en la lógica, es preciso realizar la ejecución paso a paso de algunas líneas de código, para poder apreciar los cambios que las variables tienen en el transcurso de la ejecución. A continuación, se indican algunas instrucciones para realizar una revisión paso a paso, cuando la depuración se hace desde un navegador web siguiente los siguientes pasos.

### A. Chrome DevTools

Se puede ir a las herramientas de desarrollo (DevTools) pulsando "**Command+Option+I**" en Mac o "**Control+Shift+I**" en Windows y Linux para ver el panel de inspección en la parte izquierda de la ventana del navegador. Al inspeccionar el elemento ofrece diferentes opciones para trabajar.

Las opciones presentes para depuración en Chrome son:

* “Console”: la consola se utiliza para imprimir registros, muestra errores y advertencias.
* “Elements”: se pueden seleccionar los elementos para inspeccionar los elementos HTML y hacer estilos en ellos.
* “Sources”: permite navegar los archivos que están presentes en la página cargada y se puede elegir el archivo que se desea consultar.
* “Network”: empleado para inspeccionar todas las llamadas de red que realizan los clientes al servidor, como las cabeceras HTTP, el contenido y el tamaño.

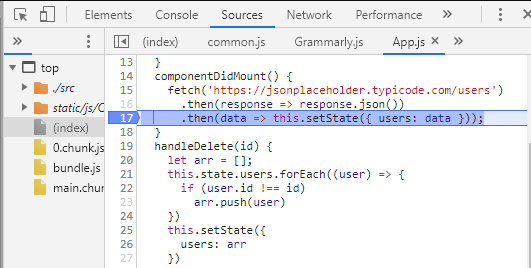
### B. Seleccionar fuente (“sources”)

Una vez se tiene que ir a la sección del archivo con el código fuente a depurar, se puede navegar a través de él e identificar la porción de código en la que se sospecha ocurre un error en la lógica.



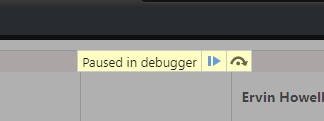
### C. Marcar punto de interrupción

Un punto de interrupción es una línea de código donde el programa se detendrá. Ya seleccionado el archivo a ser depurado, se puede ver que hay números de línea escritos por cada línea de código. Se debe hacer clic en este número de línea y esta línea actuará como un punto de interrupción y la ejecución se detendrá una vez que el control llegue a esta línea.



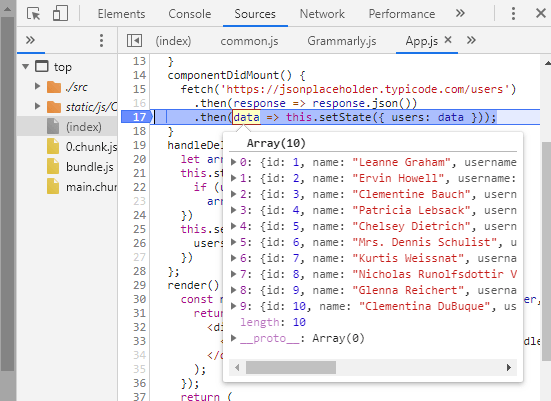
### D. Avanzar línea a línea

Cuando la ejecución se detiene en el punto de interrupción, se puede reanudar la ejecución haciendo clic en el botón azul que se muestra en la figura, así la ejecución avanza línea por línea. Aquí se puede inspeccionar cada línea de código junto con los valores contenidos en las variables.



### E. Comprobar el estado de las variables

El punto más importante es conocer los valores de las variables, esto se hace pasando el ratón por encima de la variable. La principal ventaja de usar puntos de ruptura es que se pueden conocer los valores de más de una variable al mismo tiempo, lo que no es posible en el caso de las consolas de registro, donde se debe escribir **console.log()** para cada variable.



Hay un ejemplo disponible de comprobación del estado de variables en el siguiente enlace:

<https://www.youtube.com/watch?v=I388w3wDkjc&feature=youtu.be>

# Manejo de errores y excepciones

Las excepciones son errores imprevistos que ocurren en la ejecución de un programa; son anormalidades que impiden o perturban el comportamiento o el flujo normal de las instrucciones de un programa.

El objetivo principal en el manejo de errores es separar el código de la lógica del programa, del código de manejo de excepciones: de esta forma, al producirse la anomalía, si ocurre un error y existe un manejador de excepciones, él toma el control de la ejecución del programa.

En JavaScript, el control de errores resulta muy necesario tal vez más que en otros lenguajes debido a la dificultad natural de este para testear aplicaciones o los cambios de versión de intérprete en múltiples navegadores.

La forma más sencilla de disparar un error es con la palabra reservada “throw”, este comando permite enviar al intérprete de JavaScript el evento de que ha ocurrido un error, generalmente permite enviar cualquier tipo de dato, pero lo más común que envíe un “Error” como se muestra enseguida:



Existe un tipo de estructura de control en JavaScript que está pendiente de revisar en el flujo de ejecución de un programa para detectar comportamientos inesperados, la estructura es **try … catch** que consta de un bloque que intenta la ejecución de una sesión de código (contexto), a la espera de que pueda ocurrir una posible excepción. Finalizado el contexto existe otro denominado catch que tiene un argumento o variable de contexto, donde se captura el elemento disparado.

1. Bloque de captura de excepción

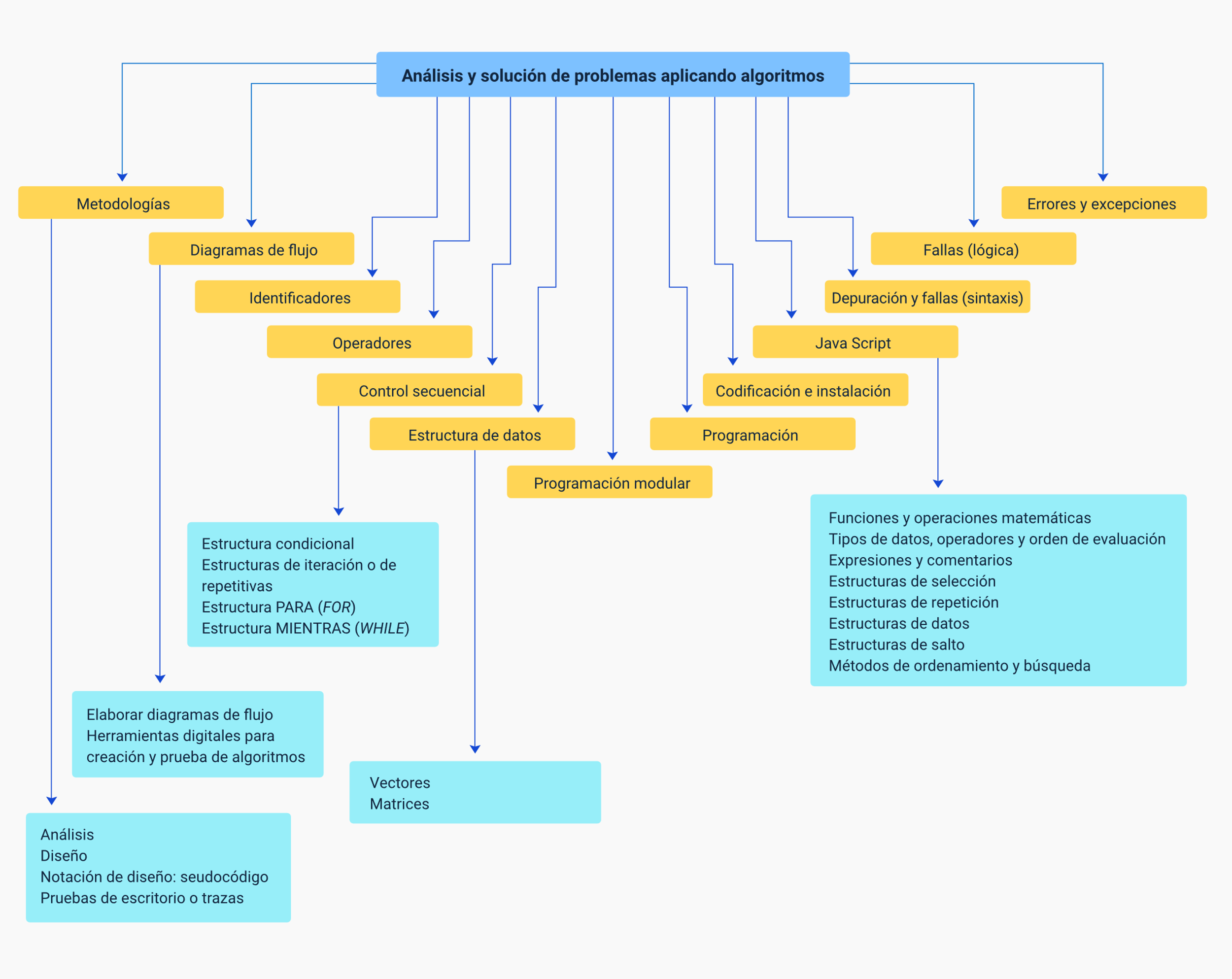


En la figura se muestra un algoritmo que divide dos números y, como se sabe, en este proceso se debe validar que no exista una división por 0 al no estar definida aritméticamente.

Para eso existe el contexto “try” que va desde la línea 4 a la línea 10, y dentro de él se verifica el caso que el divisor si es igual a 0 debe disparar una excepción. Ella es capturada en la línea 10 por el contexto catch en la variable e, de esta forma, el contexto de catch de la línea 10 a la 13 maneja la excepción, mostrando un mensaje de error y volviendo la variable resultada a nulo. Así, la excepción es lanzada y capturada y el programa no es interrumpido abruptamente, sino que es controlado a pesar de los problemas que se puedan presentar.

Síntesis

Aquí finaliza el estudio de los temas de este componente formativo. En este punto, analice el esquema que se muestra enseguida y realice su propia síntesis de lo estudiado. ¡**Adelante**!



Este esquema conceptual del componente formativo, expone que el avance tecnológico lleva implícito la frontera del conocimiento. Esto, a su vez, genera nuevos conceptos, terminologías, notaciones, nomenclaturas y su representación a través de diagramas, planos o artefactos que terminan convirtiéndose en un lenguaje natural de la tecnología en cuestión. Este componente presentó los artefactos, diagramas y nomenclaturas con los que se describen los algoritmos.

Material complementario

|  |  |  |  |
| --- | --- | --- | --- |
| Tema | Referencia | Tipo de material | Enlace del recurso |
| 2. Diagramas de flujo | Material\_Audiovisual\_100319633. (6 de enero de 2021). App Diagrams.net (básico para diagramas de flujo). | Video | <https://www.youtube.com/watch?v=jdnqmFpqTvo&feature=youtu.be> |
| 2. Diagramas de flujo | Enrique Olivares. (23 de febrero de 2016). Introducción a utilizar el programa DIA para diagramas de flujo a nivel bachillerato. | Video | <https://www.youtube.com/watch?v=DnRx9D7QWg8&feature=youtu.be> |
| 2. Diagramas de flujo | DiscoDurodeRoer. (27 de abril de 2015). Ejercicios PseInt - Básicos #1 - Empezando por lo básico. | Video | <https://www.youtube.com/watch?v=DHIi4dcaMEc&feature=youtu.be> |
| 2. Diagramas de flujo | Insituto de Informática Uach. (08 de septiembre de 2020). Cómo descargar Draw.io 2 para dibujar diagramas de flujo. | Video | <https://www.youtube.com/watch?v=encz3h8TLWA> |
| 9. Entornos de codificación e instalación | Framebits. (19 de enero de 2020). Descargar e instalar Node Js en Windows 10. [Video]. | Video | <https://www.youtube.com/watch?v=v0x1Ku5Tgac&feature=youtu.be> |
| 9. Entornos de codificación e instalación | Code Compadre. (30 de junio de 2020). How to Download and Install Visual Studio Code for Windows 10. | Video | <https://www.youtube.com/watch?v=KpzkPlh_HsU&feature=youtu.be> |
| 9. Entornos de codificación e instalación | VideoMarketingViral. (11 de enero de 2020). Descargar Google Chrome para PC 2021 (WINDOWS 7/8/10). [Video] YouTube. | Video | <https://www.youtube.com/watch?v=V-M2rMBhgKc&feature=youtu.be> |
| 10. Sintaxis del lenguaje JavaScript | MDN. (2021). Array – JavaScript. | Documentación técnica | <https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Global_Objects/Math> |
| 10. Sintaxis del lenguaje JavaScript | MDN. (2021). Expresiones y operadores – JavaScript. | Documentación técnica | <https://developer.mozilla.org/es/docs/Web/JavaScript/Guide/Expressions_and_Operators> |
| 10. Sintaxis del lenguaje JavaScript | MDN. (2021). Funciones – JavaScript. | Documentación técnica | <https://developer.mozilla.org/es/docs/Web/JavaScript/Guide/Functions> |
| 10. Sintaxis del lenguaje JavaScript | MDN. (2021). Array.prototype.find() – JavaScript. | Documentación técnica | <https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Global_Objects/Array/find> |
| 10. Sintaxis del lenguaje JavaScript | Choque, R. C. (10 de mayo de 2020). Cómo usar la consola de Google Chrome para JavaScript. [Video] YouTube | Video | <https://www.youtube.com/watch?v=Hf3n-p3VYx4&feature=youtu.be> |
| 10. Sintaxis del lenguaje JavaScript | Autodidacta, F. (11 de mayo de 2017). Depurar o hacer debug a JavaScript con Google Chrome. [Video] YouTube. | Video | <https://www.youtube.com/watch?v=I388w3wDkjc&feature=youtu.be> |

Glosario

**Código fuente**: compuesto por la línea de texto de un programa en determinado lenguaje de programación.

**Código máquina**: código de instrucciones que directamente interpreta el circuito microprogramable microprocesador de una computadora.

**Compilador**: programa de computador encargado de traducir el código fuente de otro programa escrito en un lenguaje de programación a otro lenguaje de programación.

**Concatenar**: unir o enlazar dos o más cosas (RAE, 2021).

**Iteración**: es repetir sucesivas veces un proceso con la finalidad de lograr una meta resultado u objetivo. A cada repetición del proceso también se le denomina "iteración", generalmente, y los resultados obtenidos en una iteración se emplean como punto de partida hacia la siguiente iteración.

**Modularidad**: consiste en dividir un programa en módulos, cada uno de ellos debe poder ejecutarse por separado, pero están diseñados para trabajar con otros módulos.

Referencias bibliográficas

Autodidacta, F. (11 de mayo de 2017). Depurar o hacer debug a JavaScript con Google Chrome. [Video] <https://www.youtube.com/watch?v=I388w3wDkjc&t=192s>

Choque, R. C. (10 de mayo de 2020). Cómo usar la consola de Google Chrome para JavaScript. [Video] <https://www.youtube.com/watch?v=Hf3n-p3VYx4>

Framebits. (19 de enero de 2020). Descargar e instalar Node Js en Windows 10. [Video]. <https://youtu.be/v0x1Ku5Tgac>

https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Global\_Objects/Math <https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Global_Objects/Array/find>

López, J. (2009). Algoritmos y programación (guía para docentes). <https://eduteka.icesi.edu.co>

MDN. (2021). Array - JavaScript. <https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Global_Objects/Array>

MDN. (2021a). Expresiones y operadores - JavaScript. <https://developer.mozilla.org/es/docs/Web/JavaScript/Guide/Expressions_and_Operators>

MDN. (2021b). Math - JavaScript. <https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Global_Objects/Math>

MDN. (2021d). Funciones – JavaScript <https://developer.mozilla.org/es/docs/Web/JavaScript/Guide/Functions>

MDN. (2021e). Tipos de datos y estructuras en JavaScript - JavaScript. <https://developer.mozilla.org/es/docs/Web/>

Resnick, M. (2007). Sembrando semillas para una sociedad más creativa. <https://eduteka.icesi.edu.co/articulos/ScratchResnickCreatividad>

Rojas A., V. y Ñacato C., J. (1980). Técnica de flujogramas I. Editora Andina.

Vázquez, J. (2012). Análisis y diseño de algoritmos.

Vedia, M. (2018). Cuándo usar Return en JavaScript. <https://medium.com/@mvtercero85/cuando-usar-return-en-javascript-7b80e025eb7f>

Créditos

| Nombre | Cargo | Regional y Centro de Formación |
| --- | --- | --- |
| Claudia Patricia Aristizábal | Líder del Ecosistema | Dirección General |
| Rafael Neftalí Lizcano Reyes | Responsable de Línea de Producción | Centro Industrial del Diseño y la Manufactura - Regional Santander |
| Henry Eduardo Bastidas Paruma | Experto temático | Centro de teleinformática y producción industrial - Regional Cauca |
| Jonathan Guerrero Astaiza | Experto temático | Centro de teleinformática y producción industrial - Regional Cauca |
| Peter Emerson Pinchao Solís | Experto temático | Centro de teleinformática y producción industrial - Regional Cauca |
| Zulema Yidney León Escobar | Experta temática | Centro de teleinformática y producción industrial - Regional Cauca |
| Oscar Absalón Guevara | Diseñador instruccional | Centro de gestión industrial - Regional Bogotá |
| Alix Cecilia Chinchilla Rueda | Diseñadora instruccional | Centro de gestión industrial - Regional Bogotá |
| José Gabriel Ortiz Abella | Evaluador instruccional | Centro para la industria y la comunicación gráfica - Regional Distrito Capital |
| Julia Isabel Roberto | Evaluadora instruccional | Centro para la industria y la comunicación gráfica - Regional Distrito Capital |
| Ana Catalina Córdoba Sus | Metodólogo para formación virtual | Centro Industrial del Diseño y la Manufactura - Regional Santander |
| Fabián Leonardo Correa Díaz | Diseñador Instruccional | Centro Industrial del Diseño y la Manufactura - Regional Santander |
| Carmen Alicia Martínez Torres | Animador y Productor Multimedia | Centro Industrial del Diseño y la Manufactura - Regional Santander |
| Wilson Andrés Arenales Cáceres | Storyboard e ilustración | Centro Industrial del Diseño y la Manufactura - Regional Santander |
| Camilo Andrés Bolaño Rey | Locución | Centro Industrial del Diseño y la Manufactura - Regional Santander |
| Yerson Fabian Zarate Saavedra | Diseñador de Contenidos Digitales | Centro Industrial del Diseño y la Manufactura - Regional Santander |
| Francisco José Lizcano Reyes | Desarrollador “Full-stack” | Centro Industrial del Diseño y la Manufactura - Regional Santander |
| Daniel Ricardo Mutis Gómez | Evaluador para Contenidos Inclusivos y Accesibles | Centro Industrial del Diseño y la Manufactura - Regional Santander |
| Zuleidy María Ruíz Torres | Validador de Recursos Educativos Digitales | Centro Industrial del Diseño y la Manufactura - Regional Santander |
| Luis Gabriel Urueta Álvarez | Validador de Recursos Educativos Digitales | Centro Industrial del Diseño y la Manufactura - Regional Santander |