

Programación y algoritmia básica

**Breve descripción:**

El componente formativo en algoritmia enseña la definición y aplicación de algoritmos para resolver problemas de manera eficaz. Incluye el pensamiento algorítmico, fases de diseño, ejemplos prácticos, y herramientas como pseudocódigo y diagramas de flujo, proporcionando una base sólida para la programación y la optimización de procesos computacionales.

**Julio 2024**

Tabla de contenido

[Introducción 6](#_Toc173246550)

[1. Introducción a la algoritmia 8](#_Toc173246551)

[Definición de algoritmo 8](#_Toc173246552)

[1.1. Pensamiento algorítmico 9](#_Toc173246553)

[1.2. Solución de problemas y programación 11](#_Toc173246554)

[1.3. Análisis del problema (entenderlo) 14](#_Toc173246555)

[Formular claramente el problema 14](#_Toc173246556)

[Precisar elementos de entrada 15](#_Toc173246557)

[Precisar resultados esperados 16](#_Toc173246558)

[1.4. Desarrollo de la creatividad: elementos, modelos, fases y sus objetivos 16](#_Toc173246559)

[2. Lógica matemática 18](#_Toc173246560)

[2.1. Componentes de la lógica matemática 18](#_Toc173246561)

[Proposiciones 18](#_Toc173246562)

[Conectores lógicos y proposiciones compuestas 19](#_Toc173246563)

[Tablas de verdad 20](#_Toc173246564)

[2.2. Ejercicios de lógica proposicional 27](#_Toc173246565)

[3. Metodología de algoritmos 31](#_Toc173246566)

[3.1. Análisis 32](#_Toc173246567)

[3.2. Diseño 33](#_Toc173246568)

[3.3. Pruebas de escritorio o trazas 36](#_Toc173246569)

[Solución 36](#_Toc173246570)

[4. Diagramas de flujo 41](#_Toc173246571)

[Elaborar diagramas de flujo 44](#_Toc173246572)

[Herramientas digitales para creación y prueba de algoritmos 44](#_Toc173246573)

[5. Uso de identificadores y palabras reservadas 46](#_Toc173246574)

[6. Operadores y jerarquía en los operadores 50](#_Toc173246575)

[Jerarquía en los operadores 51](#_Toc173246576)

[7. Estructuras de control secuencial 54](#_Toc173246577)

[Ejemplo estructura de control secuencial 55](#_Toc173246578)

[7.1. Estructura condicional 57](#_Toc173246579)

[7.2. Estructuras de iteración o repetitivas 60](#_Toc173246580)

[Estructura PARA (FOR) 60](#_Toc173246581)

[Estructura MIENTRAS (WHILE) 65](#_Toc173246582)

[8. Estructura de datos 69](#_Toc173246583)

[Matrices 74](#_Toc173246584)

[9. Conceptos básicos de programación 79](#_Toc173246585)

[10. Sintaxis del lenguaje JavaScript 82](#_Toc173246586)

[10.1. Tipos de datos, operadores y orden de evaluación 85](#_Toc173246587)

[a. Tipos de datos 86](#_Toc173246588)

[b. Operadores 87](#_Toc173246589)

[c. Orden de evaluación de operadores 91](#_Toc173246590)

[10.2. Expresiones y comentarios 92](#_Toc173246591)

[10.3. Estructuras de selección 93](#_Toc173246592)

[10.4. Estructuras de repetición 97](#_Toc173246593)

[a. Estructura de repetición FOR 97](#_Toc173246594)

[b. Estructura de repetición WHILE 101](#_Toc173246595)

[10.5. Estructuras de datos 103](#_Toc173246596)

[a. Vectores 103](#_Toc173246597)

[b. Matrices 106](#_Toc173246598)

[c. Registros 107](#_Toc173246599)

[10.6. Estructuras de salto 111](#_Toc173246600)

[10.7. Métodos de ordenamiento y búsqueda 115](#_Toc173246601)

[a. Ordenamiento de cadenas de texto 115](#_Toc173246602)

[b. Ordenamiento de datos numéricos 118](#_Toc173246603)

[c. Ordenamiento de registros 121](#_Toc173246604)

[d. Buscar un elemento 122](#_Toc173246605)

[11. Depuración y fallas de sintaxis 124](#_Toc173246606)

[12. Manejo de errores y excepciones 126](#_Toc173246607)

[Síntesis 129](#_Toc173246608)

[Material complementario 130](#_Toc173246609)

[Glosario 133](#_Toc173246610)

[Referencias bibliográficas 134](#_Toc173246611)

[Créditos 137](#_Toc173246612)

Introducción

La algoritmia es una disciplina fundamental en la programación de computadoras, ya que permite la resolución precisa y eficiente de problemas mediante la aplicación de una serie de pasos detallados y ordenados. La correcta comprensión y aplicación de algoritmos es esencial para el desarrollo de software, pues un algoritmo bien diseñado no solo garantiza una solución adecuada a un problema, sino que también optimiza el rendimiento y la eficacia del programa resultante. La base de esta práctica radica en la capacidad de descomponer problemas complejos en operaciones más simples y manejables, facilitando así su resolución.

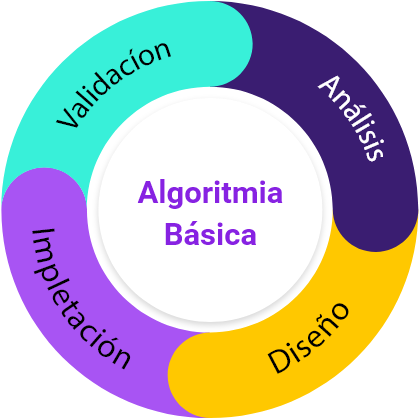
El pensamiento algorítmico es una habilidad crucial que permite abordar problemas desde diferentes enfoques lógicos, llevando siempre a una solución eficiente. A través de la algoritmia, los aprendices desarrollan la capacidad de estructurar secuencias de pasos que, aunque puedan variar en su ejecución, convergen en una solución común al problema planteado. Esta habilidad es especialmente relevante en contextos donde se requiere procesar grandes volúmenes de datos o realizar múltiples operaciones en un corto período, tareas en las que las computadoras sobresalen debido a su capacidad de realizar operaciones matemáticas básicas de manera extremadamente rápida.

En este componente formativo, se explorarán los conceptos básicos de algoritmos, su definición, y las fases para su diseño y desarrollo. Se analizarán ejemplos prácticos que ilustran cómo los algoritmos pueden aplicarse en situaciones cotidianas, así como en problemas más complejos. Además, se abordarán las herramientas y metodologías utilizadas en la elaboración de algoritmos, incluyendo el uso de pseudocódigo y diagramas de flujo.

# Introducción a la algoritmia

La correcta aplicación de una serie de pasos detallados puede garantizar una solución precisa para un problema. Es precisamente en esta práctica donde se basa la programación de computadoras y la construcción de software.

1. Pasos en la algoritmia básica



### Definición de algoritmo

Según la Real Academia Española (2020), un algoritmo está definido como un “conjunto ordenado y finito de operaciones que permite hallar la solución de un problema”.

Es importante comprender que computacionalmente, las máquinas están en capacidad de realizar unas pocas operaciones matemáticas muy básicas y sencillas, pero también se considera que computan enormes cantidades de estas operaciones en unidades de tiempo muy corto.

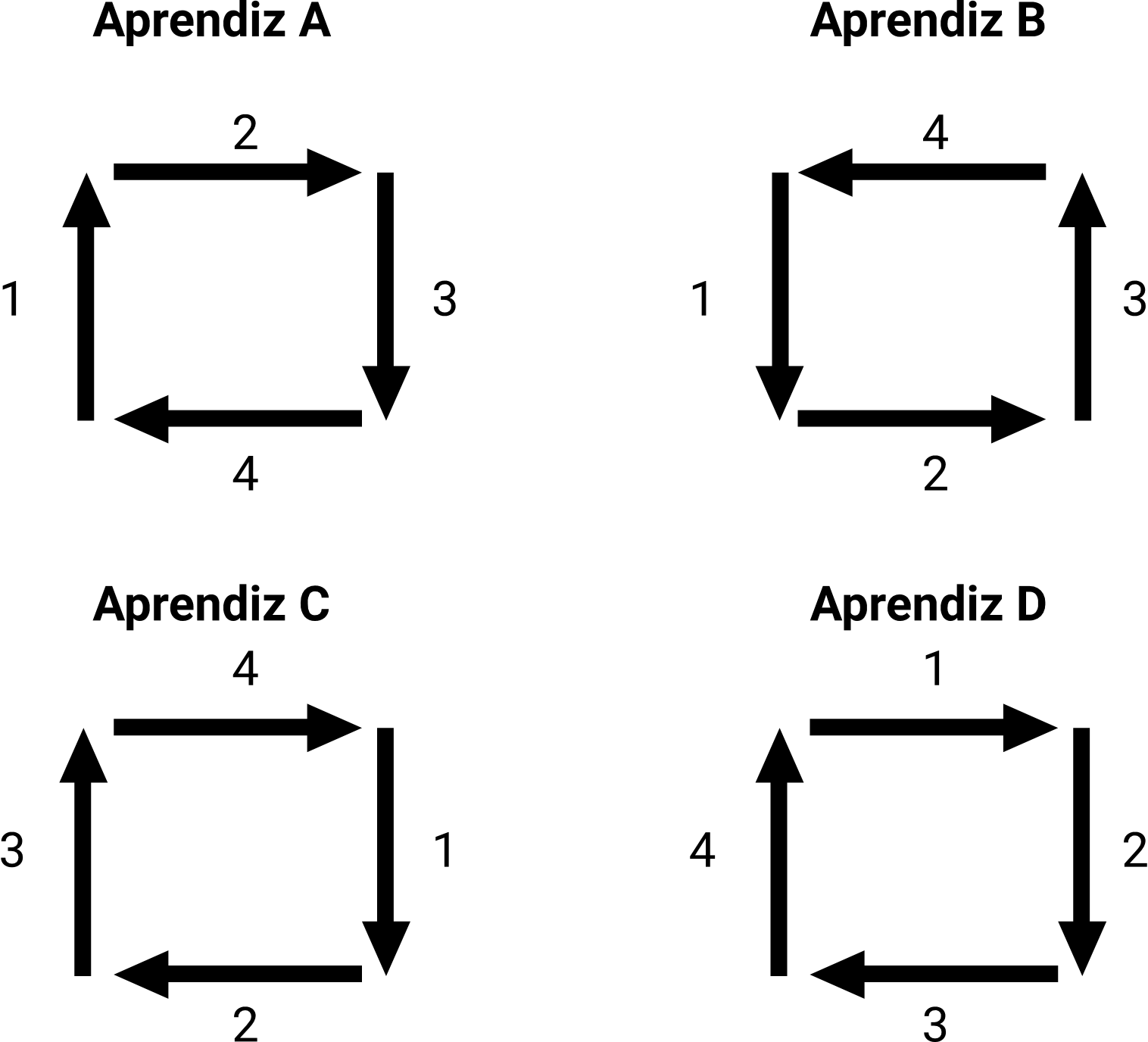
Entonces, la solución de un problema utilizando computadoras fácilmente se puede convertir en una enorme cantidad de pequeñas operaciones matemáticas, tal vez abrumadoras desde el punto de vista humano. Es por eso que se requiere la especialización de conjuntos de operaciones, los cuales se agrupan en pequeños procesos que componen la solución de un problema.

## Pensamiento algorítmico

Así como un problema puede ser abordado de varias maneras y aportar la misma solución, lo que siempre tendrán en común es que para resolverlo se utilizó una secuencia de pasos ordenados.

Por ejemplo, si se entrega una hoja de papel cuadriculado a los aprendices de una clase y se les pide que dibujen un cuadrado sin levantar el lápiz, habrá quienes lo hagan de manera diferente y obtengan el mismo resultado, como se presenta en la siguiente figura:

1. Pensamiento lógico y procedimental



En la figura, el aprendiz A comenzó realizando una línea vertical hacia arriba, mientras que el aprendiz B trazó una vertical hacia abajo, y el aprendiz D dibujó una horizontal hacia la derecha. Todos ellos llevaron a cabo un proceso que les permitió cumplir con la solución al problema planteado. En términos globales, prácticamente existe una gran cantidad de soluciones al mismo problema.

Al imaginar que el problema es mucho más complejo que el descrito en el ejemplo anterior, se concluye que existen tantas soluciones como el enfoque o lógica empleada por cada aprendiz o persona. Sin embargo, lo que sí es común en todos los casos es que se realizaron unas fases mentales secuencialmente ordenadas, las cuales son:

1. Entender el problema enunciado por el instructor (entender el problema).
2. Escoger una ruta a seguir del rectángulo (hacer un plan).
3. Plasmar en el papel cuadriculado ese plan (ejecutar el plan).
4. Revisar el resultado y ver que se alcanzó el objetivo (revisar).

Cuando el problema es muy complejo de resolver, muchas veces toca volver y analizar el problema, trazar un nuevo plan o plasmar ese nuevo plan una y otra vez, revisando los resultados y volviendo a analizar hasta encontrar la solución del problema.

## Solución de problemas y programación

Los algoritmos son una herramienta que permite describir claramente un conjunto finito de instrucciones.

“En el ámbito de la computación, los algoritmos son una herramienta que permite describir claramente un conjunto finito de instrucciones ordenadas secuencialmente y libres de ambigüedad, que debe llevar a cabo un computador para lograr un resultado previsible. Vale la pena recordar que un programa de computador consiste en una serie de instrucciones muy precisas y escritas en un lenguaje de programación que el computador entiende”. (López, 2019, p.7).

Ejemplo 1: se propone diseñar un algoritmo para apagar una computadora.

**Algoritmo: apagar computadora**

INICIO

Dar clic en el icono INICIO.

Dar clic en el icono que dice APAGAR.

Esperar que se apague completamente.

Desconectar los cables del enchufe.

FIN

Ejemplo 2: se propone diseñar un algoritmo para apagar una computadora.

**Algoritmo: cargar un celular**

INICIO

Buscar el cargador.

Agarrar el cargador.

Agarrar el celular.

Poner el enchufe en el celular.

Conectarlo al tomacorriente.

Esperar a que se ponga a cargar.

Desconectarlo del tomacorriente.

Ponerlo en un lugar seguro.

FIN

Con base en los ejemplos anteriormente planteados. Se puede concluir que un algoritmo debe ser:

* **Realizable**

El proceso algorítmico debe terminar después de una cantidad finita de pasos. Se dice que un algoritmo es inaplicable cuando se ejecuta con un conjunto de datos iniciales y el proceso resulta infinito o durante la ejecución se encuentra con un obstáculo insuperable sin arrojar un resultado.

* **Comprensible**

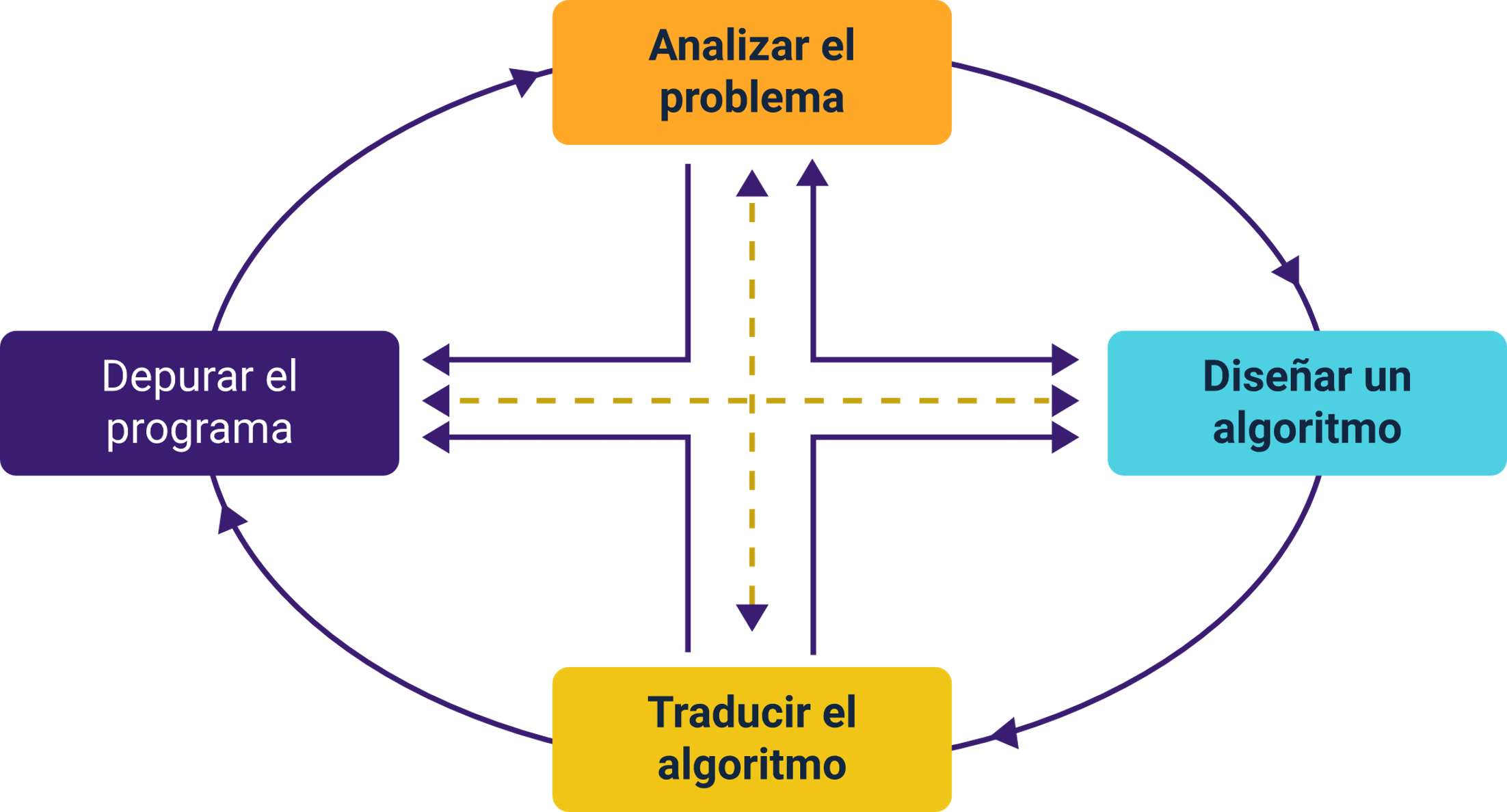
Debe ser claro lo que hace, de forma que quien ejecute los pasos (ser humano o máquina) sepa qué, cómo y cuándo hacerlo. Debe existir un procedimiento que determine el proceso de ejecución.

* **Preciso**

El orden de ejecución de las instrucciones debe estar perfectamente indicado. Cuando se ejecuta varias veces, con los mismos datos iniciales, el resultado debe ser el mismo siempre. La precisión implica determinismo. (López, 2009, p. 22)

La figura presenta las fases y sus relaciones en el proceso de diseño de algoritmos de programas informáticos, que están basados en un algoritmo que posteriormente es escrito en un lenguaje de programación.

1. Fases para elaborar un programa de computador



## Análisis del problema (entenderlo)

Como los programas de computador tienen por objetivo resolver problemas muy puntuales, lo primero que se debe realizar para resolverlos es lograr una mejor comprensión posible de estos.

Para realizar esta actividad se debe:

### Formular claramente el problema

A continuación, algunas preguntas previas a la formulación del problema:

* ¿Es posible definir de forma más clara el problema?
* ¿Qué palabras no son conocidas o se desconoce su significado?
* ¿Se ha resuelto antes algún problema similar?
* ¿Qué información es importante para resolver el problema?
* ¿Qué información no es importante y se puede omitir?

#### Ejemplo:

Don Juan necesita decidir cómo comprar un teléfono celular que cuesta $870.000 de contado o $980.000 a crédito; él tiene $630.000 pesos en efectivo.

Luego de revisar las preguntas previas se puede concluir:

Como el efectivo que tiene don Juan no le alcanza para comprar el teléfono celular de contado, entonces él tiene dos opciones:

Comprarlo completamente a crédito o también pagar una parte de contado (como cuota inicial) y el resto diferirlo a crédito.

Para poder resolver este problema se debe conocer el número de cuotas en el que difiere, si desea pagarlo totalmente a crédito o conocer el número de cuotas y el valor total del celular si se da una cuota inicial de $630.000 pesos.

### Precisar elementos de entrada

Otro punto importante en la fase de análisis del problema es determinar cuál es la información disponible, por lo tanto:

* ¿Qué información es importante y necesaria para resolver el problema?
* ¿Qué información no es importante y se puede prescindir?
* ¿Cuáles son los datos de entrada conocidos?
* ¿Cuál es la incógnita o qué se debe calcular?
* ¿Los datos se pueden agrupar en categorías?
* ¿Qué información adicional hace falta para resolver el problema?

Otro elemento a resaltar es el nivel de conocimiento y las limitaciones que se tienen en el ámbito o el contexto del problema que está tratando de resolver. Por ejemplo, si el problema requiere conceptos financieros para entender la naturaleza de los datos, o si es necesario adquirir nueva información y/o poder consultarla, para estar seguros de que es posible tratarla conforme el contexto lo requiera.

### Precisar resultados esperados

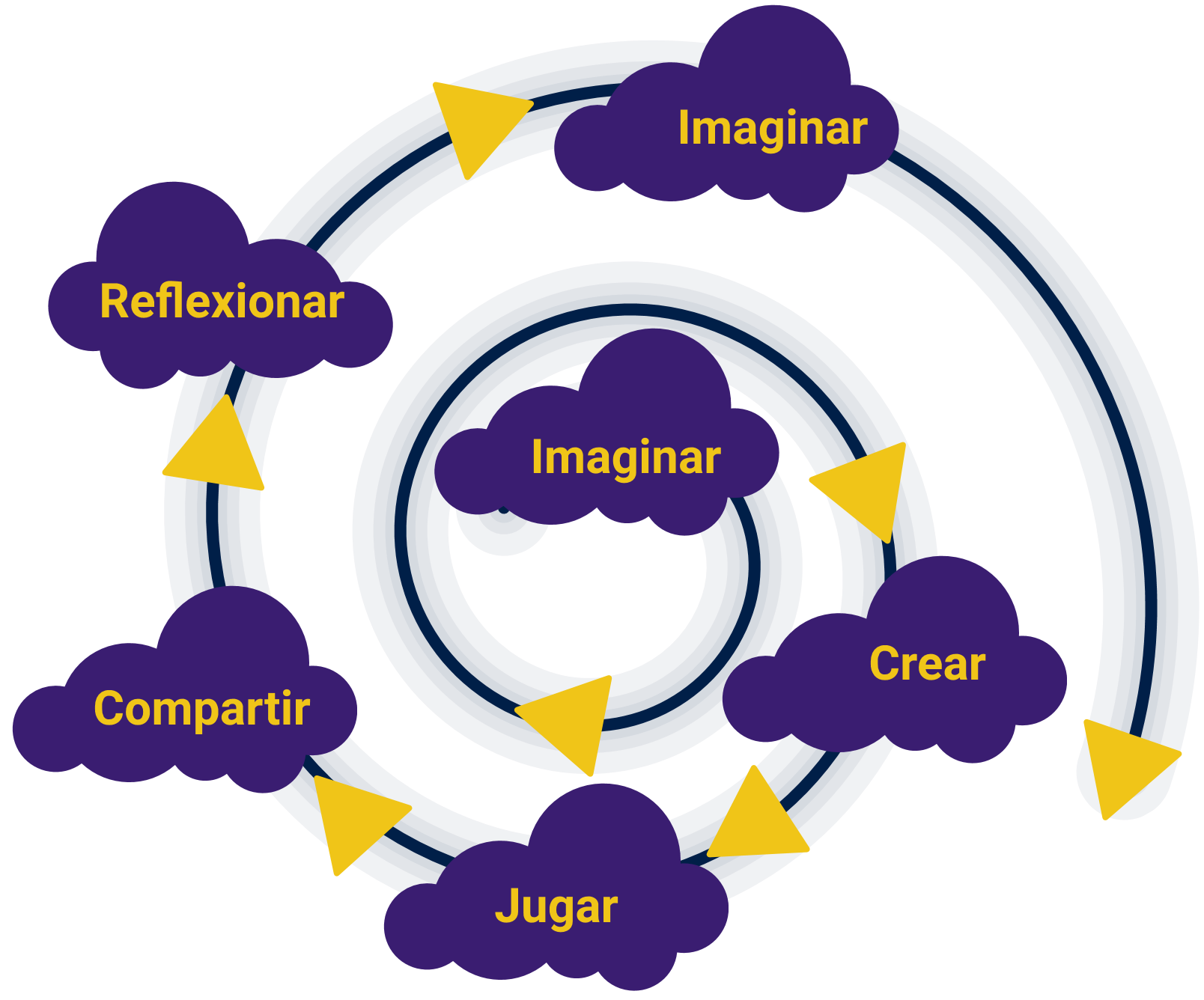
Para determinar con claridad cuál es el resultado final esperado (o producto) que se quiere obtener es necesario determinar o definir qué resultados se solicitan y, en qué tipo o formato deben estar (mostrados en pantalla, diagramación, orden, entre otros.). Para ello, es importante tener en cuenta:

* ¿Qué información están solicitando?
* ¿En qué formato se debe entregar esta información?

## Desarrollo de la creatividad: elementos, modelos, fases y sus objetivos

Resolver algoritmos es una actividad que requiere, en la mayoría de los casos, creatividad por parte del programador. Esta es una habilidad que se va fortaleciendo en la medida en que se desempeña más y más en procesos de desarrollo de software.

1. Espiral del Pensamiento Creativo diseñada por el Dr. Mitchel Resnick



“Para estimular el pensamiento creativo, propuesto por Resnick (2007), el programador debe imaginar qué quiere hacer; crear un proyecto basado en sus ideas; jugar y compartir sus ideas y creaciones con otros para reflexionar sobre sus experiencias y permitirse imaginar nuevas ideas y nuevos proyectos. La espiral genera un proceso indefinido de mejoramiento continuo”. (López, 2009, p. 19).

# Lógica matemática

La lógica matemática es la disciplina que trata de métodos de razonamiento. En un nivel elemental, la lógica proporciona reglas y técnicas para determinar si un argumento dado es válido o no. El razonamiento lógico se emplea en matemáticas para demostrar teoremas; en ciencias de la computación para verificar si los programas son correctos; en las ciencias físicas y naturales para sacar conclusiones de experimentos; y en las ciencias sociales y en la vida cotidiana para resolver una multitud de problemas. Ciertamente, se usa de forma constante el razonamiento lógico para realizar cualquier actividad (EcuRed, s.f.).

## Componentes de la lógica matemática

Todo razonamiento lógico está compuesto por una serie de elementos que se interrelacionan entre sí. La correcta interpretación de estos componentes y de sus relaciones corresponde al análisis deductivo. Estos componentes son las proposiciones y los conectores lógicos, que se describen a continuación:

### Proposiciones

Una proposición o enunciado es una oración que puede ser falsa o verdadera, pero no ambas a la vez. La proposición es un elemento fundamental de la lógica matemática.

Las proposiciones pueden ser simples o compuestas. Las primeras están formadas por una sola proposición; por el contrario, las proposiciones compuestas están conformadas por dos o más proposiciones unidas por un conector.

**p**: la tierra es plana.

**q**: -17 + 38 = 21

**r**: hola ¿cómo estás?

**S**: lava el carro por favor.

**p** y **q** son proposiciones simples válidas porque pueden tomar un valor verdadero o falso; sin embargo, los enunciados r y s son proposiciones no válidas, porque no pueden tomar un valor de falso o verdadero. Uno de ellos es un saludo y el otro es una orden.

### Conectores lógicos y proposiciones compuestas

Existen las proposiciones compuestas que son las que se componen de varias proposiciones, las cuales utilizan para su conexión operadores o conectores lógicos que son:

1. **Operador AND (y) conjunción**

Conecta dos proposiciones que se deben unir para que se pueda obtener un resultado verdadero, esta se representa por el siguiente símbolo ∧.

**Ejemplo**

* Nuestra bandera es de color amarillo, azul y rojo.
* El número 9 es mayor que 7 y menor que 12.
* El carro enciende cuando tiene corriente eléctrica y gasolina en el tanque.

1. **Operador OR (o) disyunción**

Conecta dos proposiciones y se obtiene una proposición compuesta verdadera cuando por lo menos unas de las proposiciones son verdaderas.

**Ejemplo**

* Puedo ir a Bogotá en avión o en bus.
* Puedo entrar al estadio si compro mi boleta o si tengo un pase de cortesía.
* Voy a desayunar con huevos revueltos o huevos pericos.

1. **Operador NOT (no) negación**

Su propósito es negar una proposición, es decir, que si tenemos una proposición verdadera al aplicarle este tipo de operador se obtiene la negación, es decir, es falso.

**Ejemplo**

* No es cierto que Gabriel García Márquez escribió la Ilíada.
* No es cierto que el América de Cali es de la ciudad de Bogotá.
* Es falso que el agua es un hidrocarburo.

### Tablas de verdad

Una estrategia de la lógica que tiene como objetivo determinar la validez de varias propuestas en alguna situación es la tabla de verdad. Esta estrategia determina las condiciones necesarias en las que un enunciado es verdadero, representando dichas condiciones en una tabla que establece el valor de verdad de todas las proposiciones que componen la situación. Así como lo explica el principio de bivalencia: una proposición puede ser verdadera o falsa únicamente, nunca ambas.

Algunas consideraciones:

1. El valor de una proposición verdadera se representa por “V” o un “1”.
2. El valor de una proposición falsa se representa por “F” o un “0”.
3. Existe una tabla de verdad para cada uno de los conectores lógicos, lo que se tiene que evaluar para responder si una proposición compuesta es verdadera o falsa es el estatus actual de las proposiciones simples.

A continuación, se presentan las tablas de verdad.

1. Tabla de verdad conjunción (Y)

| P | Q | P ∧ Q |
| --- | --- | --- |
| V | V | V |
| V | V | V |
| F | V | F |
| F | F | F |

O puede

1. Tabla de verdad conjunción (Y)

| P | Q | P ∧ Q |
| --- | --- | --- |
| 1 | 1 | 1 |
| 1 | 0 | 0 |
| 0 | 1 | 0 |
| 0 | 0 | 0 |

1. Tabla de verdad disyunción (O)

| P | Q | P ∨ Q |
| --- | --- | --- |
| V | V | V |
| V | F | V |
| F | V | V |
| F | F | F |

O puede

1. Tabla de verdad disyunción (O)

| P | Q | P ∨ Q |
| --- | --- | --- |
| 1 | 1 | 1 |
| 1 | 0 | 1 |
| 0 | 1 | 1 |
| 0 | 0 | 0 |

1. Tabla de verdad negación (no)

| P | ¬P |
| --- | --- |
| V | F |
| F | V |

O puede

1. Tabla de verdad negación (no)

| P | ¬P |
| --- | --- |
| 1 | 0 |
| 0 | 1 |

Existe otro tipo de conectores que se llaman proposiciones condicionales, aquellas que están formadas por dos proposiciones simples o compuesta por p y q, la cual se representa y se lee de la siguiente manera:

**p → q se lee “Si p entonces q”**

* **Ejemplo 1**

**Las palomas vuelan.**

**p**: todas las palomas vuelan.

**q**: todas las aves tienen plumas.

**r**: luego todas las palomas son aves.

Estas proposiciones se pueden escribir utilizando los operadores lógicos de la siguiente manera: (p ∧ q) → r.

**Para el anterior ejercicio se construye la** **tabla de verdad:**

1. Tabla de verdad ejemplo 1

| P | Q | R | P ∧ Q | ¬R |
| --- | --- | --- | --- | --- |
| V | V | V | V | F |
| V | V | F | V | V |
| V | F | V | F | F |
| V | F | F | F | V |
| F | V | V | F | F |
| F | V | F | F | V |
| F | F | V | F | F |
| F | F | F | F | V |

* **Ejemplo 2**

**Si no llueve se arruina la cosecha.**

Este tipo de proposición se conoce como la condición “para que”, de manera que para que no se arruine la cosecha se requiere que llueva.

**p**: si no llueve.

**q**: se arruina la cosecha.

**Al aplicar el operador p → q su tabla de verdad quedaría de la siguiente manera:**

1. Tabla de verdad ejemplo 2

| P | Q | P → Q |
| --- | --- | --- |
| V | V | V |
| V | F | F |
| F | V | V |
| F | F | V |

O puede

1. Tabla de verdad ejemplo 2

| P | Q | P → Q |
| --- | --- | --- |
| 1 | 1 | 1 |
| 1 | 0 | 0 |
| 0 | 1 | 1 |
| 0 | 0 | 1 |

De igual manera, existe una proposición bicondicional que indica que una proposición es verdadera si y solo si las proposiciones simples o compuestas que forman la proposición son verdaderas o son falsas si y solo si ambas proposiciones son falsas, esta se representa y se lee de la siguiente manera: p ↔ q se lee "p si solo si q"

* **Ejemplo 3**

**Es buen estudiante.**

Es buen estudiante, si y solo si tiene promedio de cinco, donde se puede concluir lo siguiente:

**p**: es un buen estudiante.

**q**: tiene promedio de 5.

**al aplicar el operador p ↔ q su tabla de verdad quedaría de la siguiente manera:**

1. Tabla de verdad ejemplo 3

| P | Q | P ↔ Q |
| --- | --- | --- |
| V | V | V |
| V | F | F |
| F | V | V |
| F | F | V |

O puede

1. Tabla de verdad ejemplo 3

| P | Q | P ↔ Q |
| --- | --- | --- |
| 1 | 1 | 1 |
| 1 | 0 | 0 |
| 0 | 1 | 1 |
| 0 | 0 | 1 |

#### Cuando se aplican tablas de verdad se puede llegar a tres tipos de resultado que son:

* **Tautología**: es una expresión lógica que es verdadera para todos los posibles valores de verdad de sus componentes atómicos. Las proposiciones dan el resultado positivo.
* **Contradicción**: son aquellas fórmulas que son falsas para cualquier valoración de los símbolos proposicionales que contiene. Las proposiciones dan el resultado negativo.
* **Contingencia**: son aquellas fórmulas cuyo valor de verdad o falsedad depende de la valoración de los símbolos proposicionales que contiene. Las proposiciones dan un resultado negativo y positivo. (TOMi.digital, 2021).

## Ejercicios de lógica proposicional

A continuación, se plantean a manera de ejemplo algunos ejercicios con su respectiva solución que permiten identificar un problema, reconocer los datos de entrada y los procesos a ser aplicados como estrategia a seguir con la utilización de la lógica proposicional en la identificación y solución de problemas. Para dar solución al enunciado anterior se realiza las siguientes actividades.

**Ejercicio 1**: se necesita personal femenino que sepa Microsoft Office o Gsuite.

Pasos a seguir:

1. Identificar el problema.
2. Descomponer el enunciado en proposiciones atómicas.
3. Evaluar los posibles casos derivados del enunciado anterior.
4. Determinar los casos donde se cumpla la condición y hacer la selección de las personas de acuerdo con el requerimiento.

**Solución**

* **Paso 1**

**Identificar el problema**

Se trata de buscar o contratar personal femenino que sepa utilizar Microsoft Office o Google Suite.

* **Paso 2**

**Descomponer el enunciado**

Descomponer en proposiciones atómicas del enunciado (cuando están separadas del contexto del problema se llaman proposiciones atómicas).

**p**: es personal femenino.

**q**: sabe utilizar el Microsoft Office.

**r**: sabe utilizar la suite de Google.

* **Paso 3**

**Determinar los casos**

Determinar las relaciones lógicas de acuerdo con lo planteado en el enunciado, utilizando las proposiciones atómicas definidas en el punto anterior.

p ∧ (q V r).

**Ejercicio 2:** Juan sabe utilizar Microsoft Office y Gsuite, Martha sabe utilizar Gsuite, Natalia sabe utilizar Gsuite y Microsoft Office.

Crear la tabla de verdad y aplicar la lógica proposicional para elegir las personas que cumplan con los requisitos exigidos.

Para la construcción de la tabla de verdad se evalúa cada una de las preposiciones atómicas y, se van validando para verificar si se cumple lo que la haría verdadera o si no se cumple, lo cual haría que la proposición sea falsa.

Se analiza cada posible caso por lo que se evalúa a Juan, para la primera proposición sería un valor falso, ya que p requiere que sea personal femenino, el cual se pondrá en la columna 2, luego se evalúa si Juan sabe utilizar Microsoft Office o la Suite de Google, el valor va en la columna tres y por último, se evalúa la conjunción para determinar la solución.

1. Tabla de verdad ejemplo 2

| Nombre | P | (q ∨ r) | p ∧ (q ∨ r) |
| --- | --- | --- | --- |
| Juan | F | V | F |
| Martha | V | V | V |
| Nathalia | V | V | V |

**Ejercicio 3**. Sandra dice adelgazo o no hago dieta.

Para este enunciado además se propone una serie de alternativas que podrían ser válidas o no, dependiendo de la solución propuesta utilizando la lógica matemática. Las siguientes son las posibles respuestas de las cuales se deberá elegir la que se considera es real al problema planteado.

* Si adelgazo, hago dieta.
* Si no adelgazo, hago dieta.
* Si hago dieta, no adelgazo.
* Si hago dieta adelgazo.
* Hago dieta o no adelgazo.

**Solución**

* **Paso 1**

**Problema**

Identificar la respuesta correcta al problema presentado por Sandra.

* **Paso 2**

**Definir**

Definir las proposiciones atómicas:

**p**: adelgazo.

**q**: hago dieta.

* **Paso 3**

**Determinar**

Determinar la relación lógica.

**p V ¬ q.**

* **Paso 4**

**Crear**

Crear la tabla de verdad.

1. Tabla de verdad ejercicio 3

| P | Q | PQP∨¬Q |
| --- | --- | --- |
| V | V | V |
| V | F | V |
| F | V | F |
| F | F | V |

# Metodología de algoritmos

De manera implícita, en las distintas áreas productivas de la sociedad se usan algoritmos. 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 continúa 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 estas.

**En general, existen tres fases secuenciales para la elaboración de un algoritmo, de las cuales se abordarán las dos primeras:**

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.
* Implementación.

1. **Codificación de algoritmo**

* Prueba y depuración.
* Documentación.
* Mantenimiento.

## Análisis

Como se explicó en componentes anteriores, la fase de análisis trata de responder a la pregunta: ¿Qué es lo que se debe hacer? 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 entre 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 obligatoria a tener en cuenta:

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

## Diseño

En esta etapa se responde, básicamente, a la pregunta: ¿Cómo se va a hacer?, refiriéndose a la manera en 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, especialmente si se está comenzando en el diseño de soluciones basadas en algoritmos y no se cuenta con mucha experiencia:

1. Actividades diseño algoritmo



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. Inicialmente, se empleará 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 un lenguaje de programación que las computadoras puedan interpretar. Sin embargo, se puede expresar ese mismo contenido en el lenguaje natural, 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:**

* **Análisis**

El programa debe solicitar al usuario que ingrese un número. Luego, recibirá ese número desde el teclado y lo almacenará en una variable. Para ello, se debe definir una variable antes de empezar el algoritmo, reservando un espacio para guardar un número entero.

Una vez definida la variable, se leerá el número ingresado por el usuario y se almacenará en dicha variable.Finalmente, el programa imprimirá el contenido de la variable, concluyendo así el proceso.

* **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.

ALGORITMO Mostrar número digitado;

VAR

ENTERO entrada;

INICIO

ESCRIBIR ("Por favor ingrese un número");

LEER (entrada);

ESCRIBIR (entrada);

FIN

En un algoritmo, se indica el nombre del mismo y luego se declaran las variables que serán usadas (con VAR), especificando 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. Usamos ESCRIBIR() para mostrar 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 simulaciones de ejecución de un algoritmo que permiten determinar la validez o efectividad del mismo.

#### Pruebas de escritorio o trazas

Consisten en escribir en una tabla con tantas columnas como variables tiene el algoritmo y seguir las instrucciones, registrando los valores correspondientes.

Lo invitamos a consultar el PDF “**Pruebas de escritorio o trazas**”, el cual se encuentra en la carpeta Anexos.

#### 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.

### Solución

* **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 de texto que indique si es el resultado del cuadrado o del cubo.

* **Diseño**

El siguiente código tiene la representación del resultado al análisis anterior.

ALGORITMO Calcular el cuadrado y el cubo;

VAR

ENTERO entrada;

ENTERO cuadrado;

ENTERO cubo;

INICIO

ESCRIBIR ("Por favor ingrese un número");

LEER ( entrada );

cuadrado <= entrada \* entrada;

cubo <= entrada \* entrada \* entrada;

ESCRIBIR( "El Cuadrado " cuadrado );

ESCRIBIR( "El Cubo " cubo);

FIN

Las pruebas de escritorio permiten verificar que el pseudocódigo realice lo que se espera al resolver el problema. Es importante que a todos los pseudocódigos se les aplique, al menos, una 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 debe recorrer línea a línea el pseudocódigo, identificando tanto las variables como las condiciones e ir anotándolas en la tabla de la prueba de escritorio.

Es importante que en la tabla no aparezcan variables o condiciones repetidas. En la última columna se pueden anotar los datos de salida, identificándolos como "Salida".

**Ejecutando línea a línea el algoritmo del ejemplo anterior, se va llenando la tabla:**

* **Paso 1**

ESCRIBIR ("Por favor ingrese un número");

1. Tabla paso 1

| Número | Cuadrado | Cubo | Salida |
| --- | --- | --- | --- |
|  |  |  | Por favor ingrese un número |

* **Paso 2**

LEER(entrada);

1. Tabla paso 2

| Número | Cuadrado | Cubo | Salida |
| --- | --- | --- | --- |
| 3 |  |  | Por favor ingrese un número 3 |

* **Paso 3**

cuadrado <= entrada \* entrada;

1. Tabla paso 3

| Número | Cuadrado | Cubo | Salida |
| --- | --- | --- | --- |
| 3 | 9 |  | Por favor ingrese un número 3 |

* **Paso 4**

cubo <= entrada \* entrada \* entrada;

1. Tabla paso 4

| Número | Cuadrado | Cubo | Salida |
| --- | --- | --- | --- |
| 3 | 9 | 27 | Por favor ingrese un número 3 |

* **Paso 5**

ESCRIBIR (cuadrado);

1. Tabla paso 5

| Número | Cuadrado | Cubo | Salida |
| --- | --- | --- | --- |
| 3 | 9 | 27 | Por favor ingrese un número 3  El cuadrado 9 |

* **Paso 6**

ESCRIBIR (cuadrado);

1. Tabla paso 6

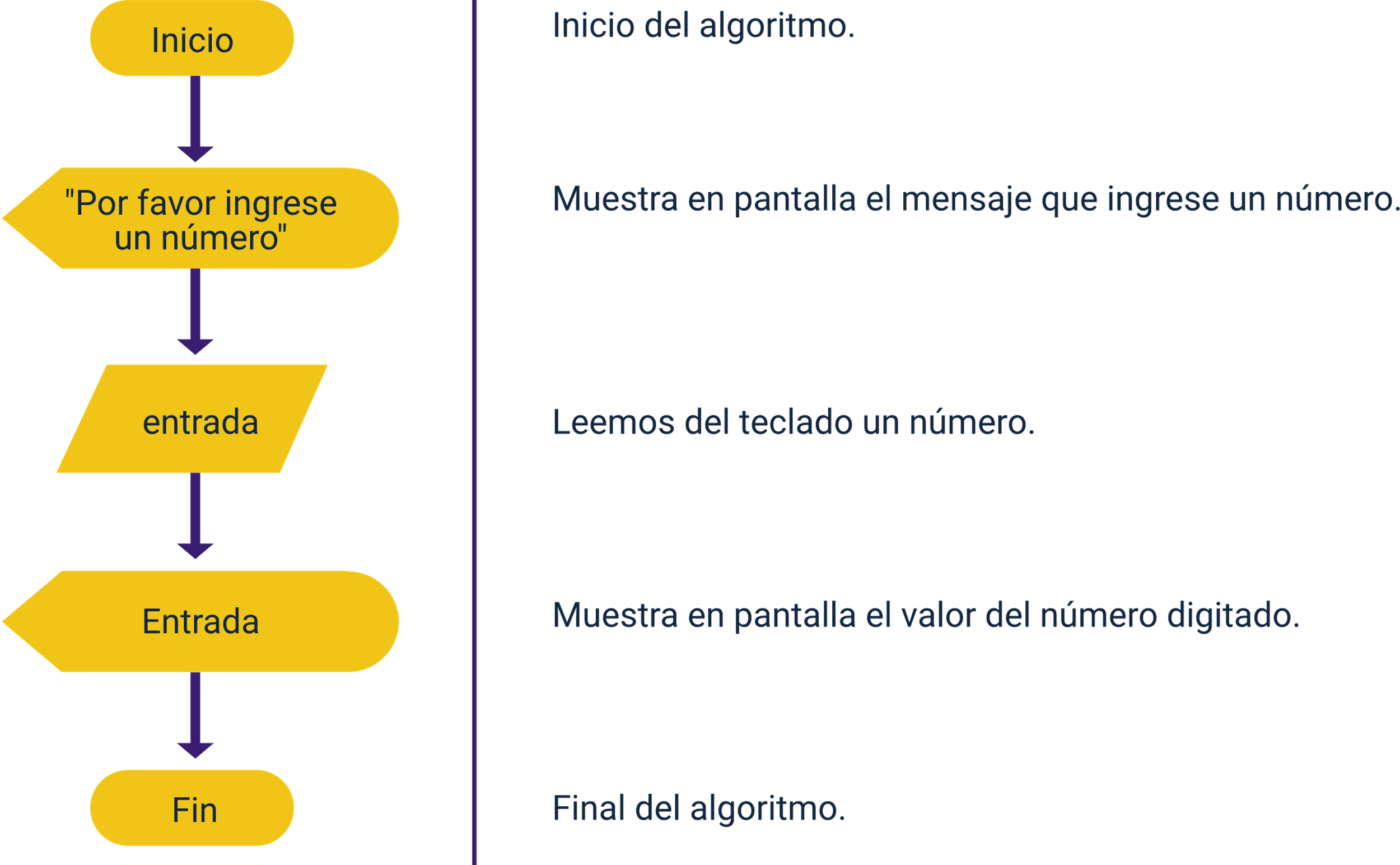
| Número | Cuadrado | Cubo | Salida |
| --- | --- | --- | --- |
| 3 | 9 | 27 | Por favor ingrese un número 3  El cuadrado 9  El cubo 27 |

También es posible realizar pruebas de escritorio a otras representaciones de algoritmos, como los diagramas de flujo, expuestos a continuación. Se recomienda realizar la prueba de escritorio correspondiente a cada ejemplo presentado.

# Diagramas de flujo

Una forma común de presentar los algoritmos es mediante diagramas de flujo, que consisten en una representación gráfica. Mientras que en pseudocódigo las sentencias se representaban con proposiciones, en los diagramas de flujo las proposiciones son reemplazadas por figuras.

1. Diagrama de flujo imprimir un entero



#### Ventajas de los diagramas de flujo

Es ideal revisar la siguiente infografía sobre las ventajas de la representación gráfica de un algoritmo a través del diagrama de flujo.

Lo invitamos a consultar el PDF “**Ventajas de los diagramas de flujo**”, el cual se encuentra en la carpeta Anexos.

La representación del algoritmo en la figura (algoritmo que calcula el cuadrado y el cubo de un número introducido por teclado y muestra los resultados):

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



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.

* **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.

* **Entrada General**

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

* **Iteración**

Indica que una instrucción o grupo de instrucciones deben ejecutarse varias veces.

* **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.

* **Salida Impresa**

Indica la presentación de uno o varios resultados en forma impresa.

* **Llamada a subrutina**

Indica la llamada a una subrutina o procedimiento determinado.

* **Salida en Pantalla**

Instrucción de presentación de mensajes o resultados en pantalla.

* **Acción/Proceso General**

Indica una acción o instrucción general que debe realizar el computador (cambios de valores de variables, asignaciones, operaciones aritméticas).

* **Conector**

Indica el enlace de dos partes de un diagrama dentro de la misma página.

* **Flujo**

Indica el seguimiento lógico del diagrama. También indica el sentido de ejecución de las operaciones.

* **Conector**

Indica el enlace de dos partes de un diagrama en páginas diferentes.

### 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).

#### Diagramas de flujo

A continuación, se puede conocer cómo se estructura el diagrama de flujo.

Lo invitamos a consultar el PDF “**Diagramas de flujo**”, el cual 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 esta herramienta, 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 presentan en la siguiente figura.

1. Seleccionar formas



# 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:

1. Pueden estar formados por una combinación de letras y números (saldoMes, salario, fecha2, etc.).
2. El primer carácter de un nombre debe ser una letra.
3. La mayoría de los lenguajes de programación diferencian las mayúsculas de las minúsculas.
4. Los nombres deben ser nemotécnicos; con solo leerlos se debe entender lo que contienen.
5. Deben ser muy descriptivos; no utilizar abreviaturas, a menos que se justifique.
6. Es conveniente utilizar una sola palabra para nombrar páginas, controles, variables, etc.
7. No utilizar caracteres reservados (%, +, /, >, etc.) ni letras acentuadas (á, é, í, ó, ú).
8. 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, se usa el identificador "entrada" para representar el número digitado. En el algoritmo de imprimir el cuadrado y el cubo de un número entero, se emplean los siguientes identificadores:

1. "entrada" para representar el número digitado.
2. "cuadrado" para representar el valor del cuadrado del número de entrada.
3. "cubo" para almacenar el cubo del número de entrada.

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

* **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 último valor o dato asignado a esta.

* **Constantes**

Las constantes se crean de la misma forma que las variables, a diferencia de que se procura su definición o creación al 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.

ALGORITMO Contador;

VAR

ENTERO contador;

INICIO

contador = 0;

contador <= contador + 1;

contador <= contador + 1;

contador <= contador + 1;

ESCRIBIR ( contador ) ;

FIN

* **Acumuladores**

Los acumuladores son una versión ampliada de contadores. Tienen las mismas características que un contador, excepto que el valor que se va incrementando no es una constante, sino 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:

* **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 (=).

* **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.

* **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 (>=).

* **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. El siguiente es el orden (jerarquía) para ejecutar operadores:

1. Paréntesis (se ejecutan primero los más internos).
2. Signo (e.g., -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. Relacionales (=, <, >).
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, en particular los más interiores. De esta forma, el computador se interesará en resolver primero el contenido de (7 + 8), cuyo resultado es 15.  De esta forma, la expresión queda reducida a:  (2 + 15 \* 16 + 3 ^ 2) / 4 + 5 |
| 2 | 3 ^ 2 | 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 (3 ^ 2) 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 + 3^2) / 4 + 5.

**Solución:**

Primero que todo, lo que más valor tiene en la jerarquía son los paréntesis, 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.

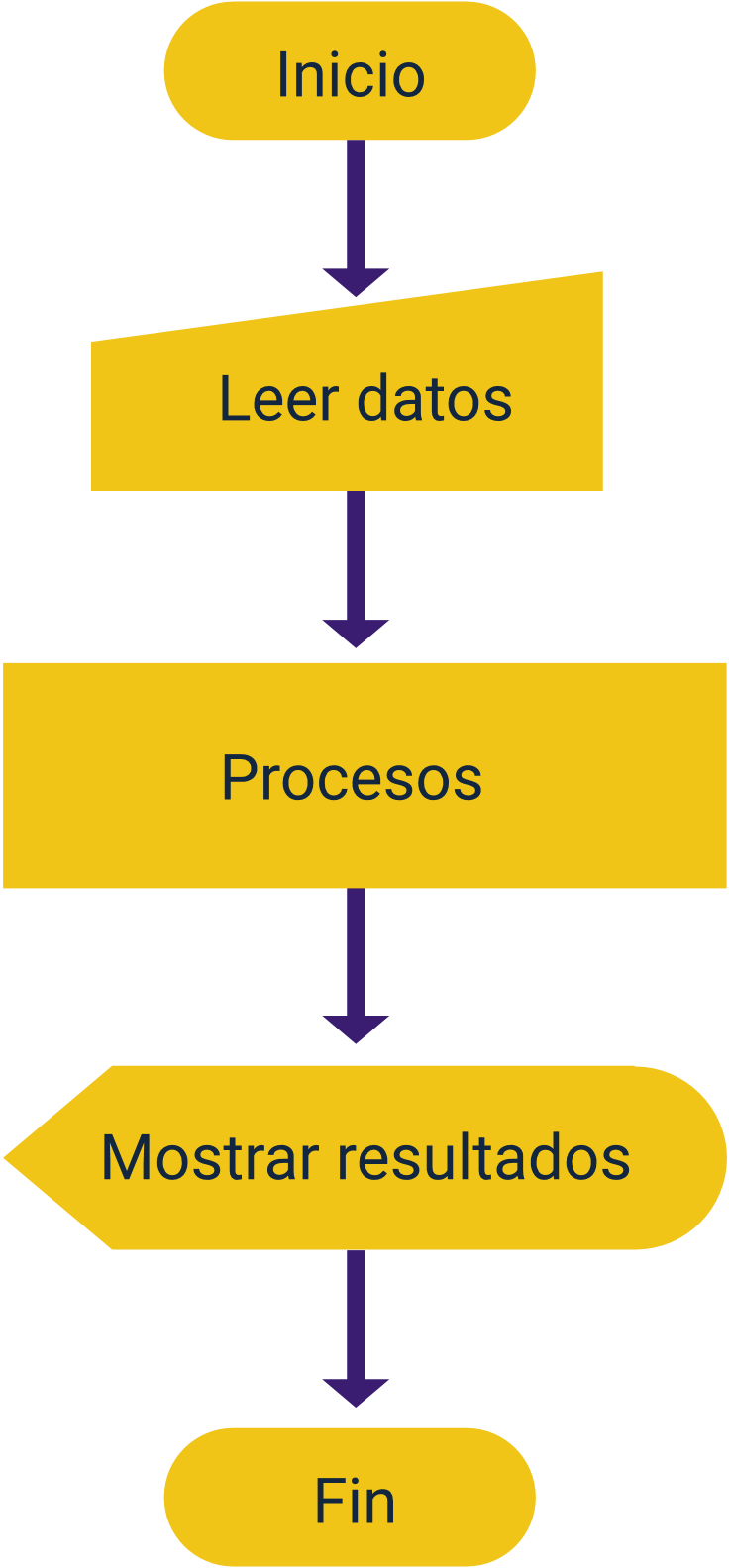
# Estructuras de control secuencial

Una estructura se define 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 de 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 utilizan otras estructuras iterativas (repetir varias veces un conjunto de instrucciones) y condicionales (López, 2009).

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

1. Modelo de estructura secuencial



### 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.

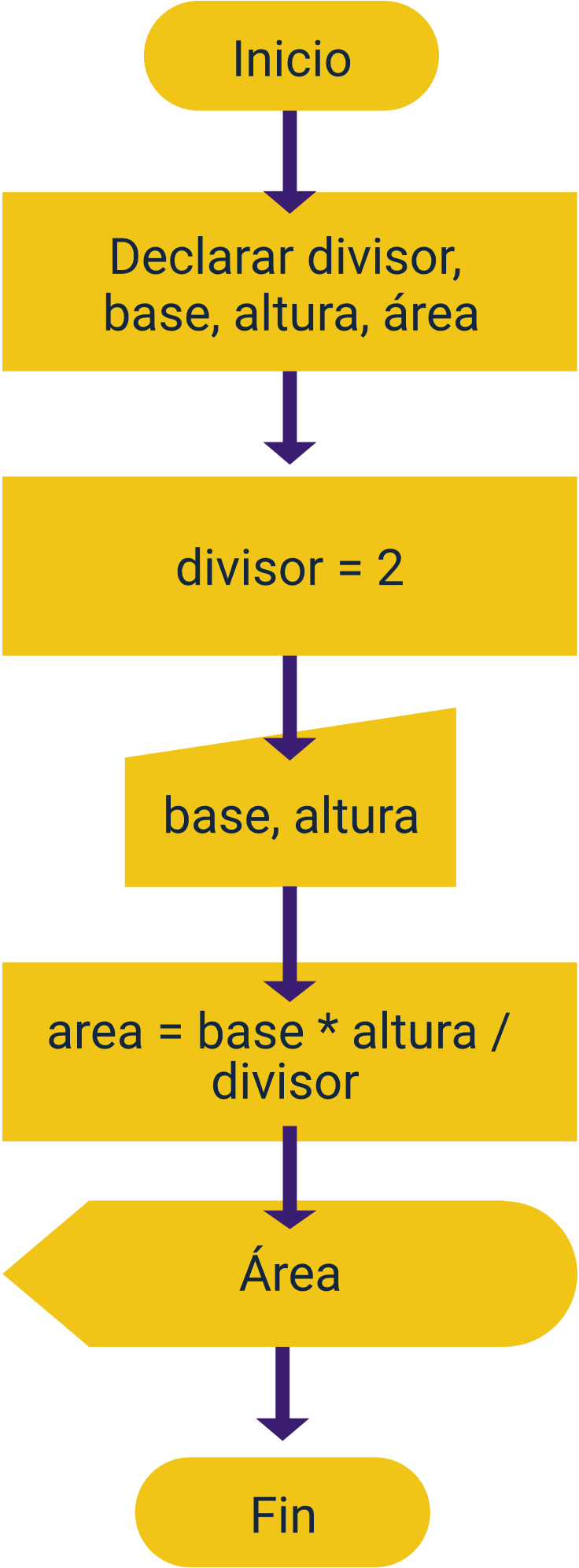
#### Análisis

* **Formular el problema**: está claramente planteado y consiste en calcular el área de un triángulo 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.

#### Diseño

El algoritmo resultado del análisis, en la que están representados tanto el diagrama de flujo como en pseudocódigo. Mientras se está aprendiendo a diseñarlos, es importante hacer los algoritmos en diagrama de flujo, y a medida que se va adquiriendo experticia con la representación en pseudocódigo, es más conveniente hacer esta representación debido a la velocidad a la hora de diseñarlo.

1. Diagrama al área de un triángulo



ALGORITMO área de un triángulo;

VAR

ENTERO divisor;

REAL area;

REAL base;

REAL altura;

INICIO

divisor = 2;

ESCRIBIR( "Escriba la base ");

LEER ( base )

ESCRIBIR( "Escriba la altura ");

LEER( altura ) ;

area = base \* altura / divisor;

ESCRIBIR( area ) ;

FIN

Del diagrama de flujo y el pseudocódigo se puede notar lo siguiente: el identificador "area" 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 valor entero, sino también cifras decimales.

## Estructura condicional

Es importante prestar atención a las estructuras que se 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 correspondiente.

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.

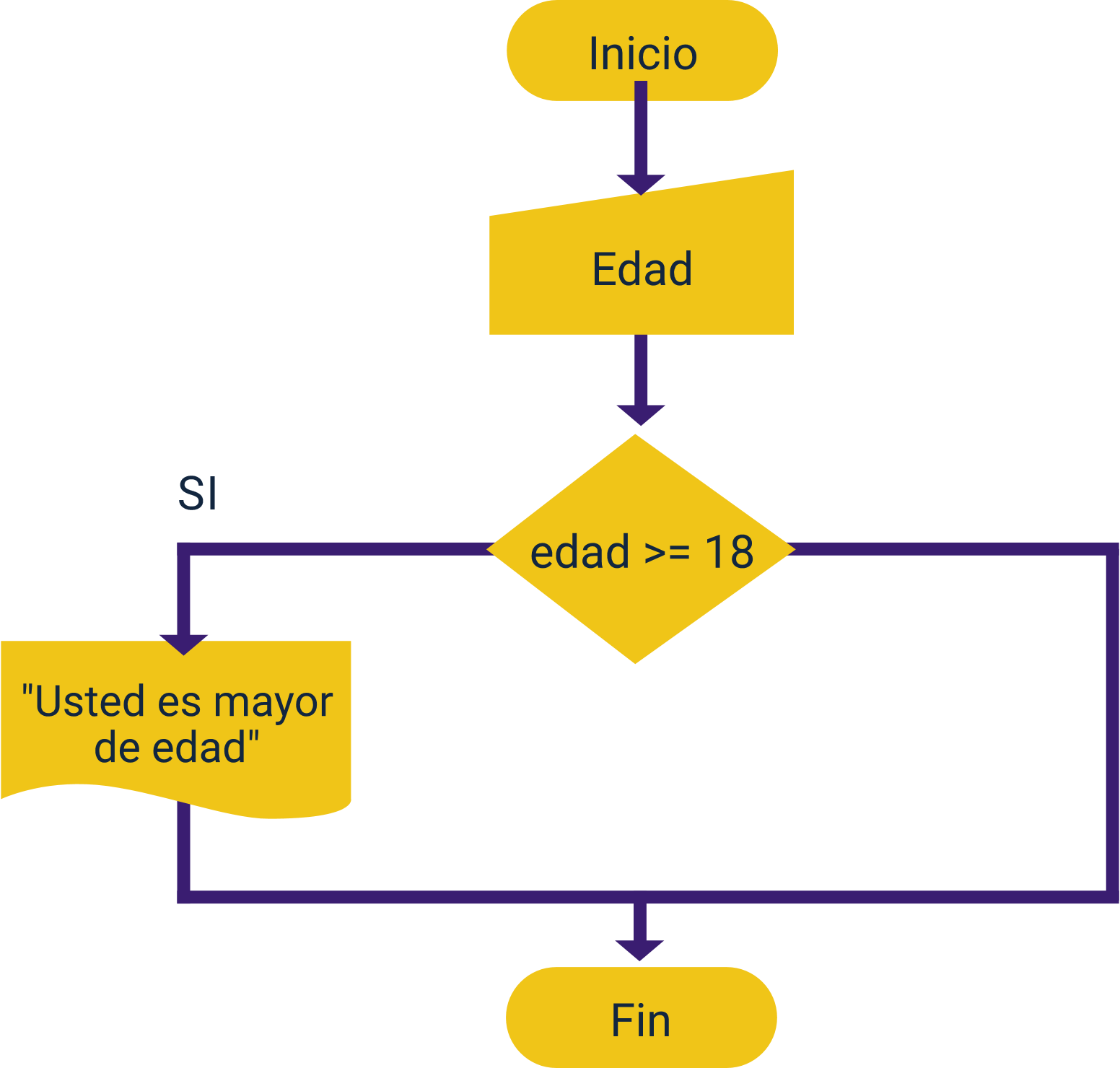
#### Análisis

* **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**: necesarios: hacer que se imprima en pantalla el mensaje 7 veces.

#### Diseño

Como se presenta en la figura, 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.

1. Estructura condicional simple



#### ALGORITMO Calificación del usuario.

ALGORITMO Edad del usuario;

VAR

REAL edad;

INICIO

ESCRIBIR( "Escriba la calificación");

LEER ( edad )

SI ( edad >= 18)

ESCRIBIR ( "Usted es mayor de edad" );

FINSI

FIN

## 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.

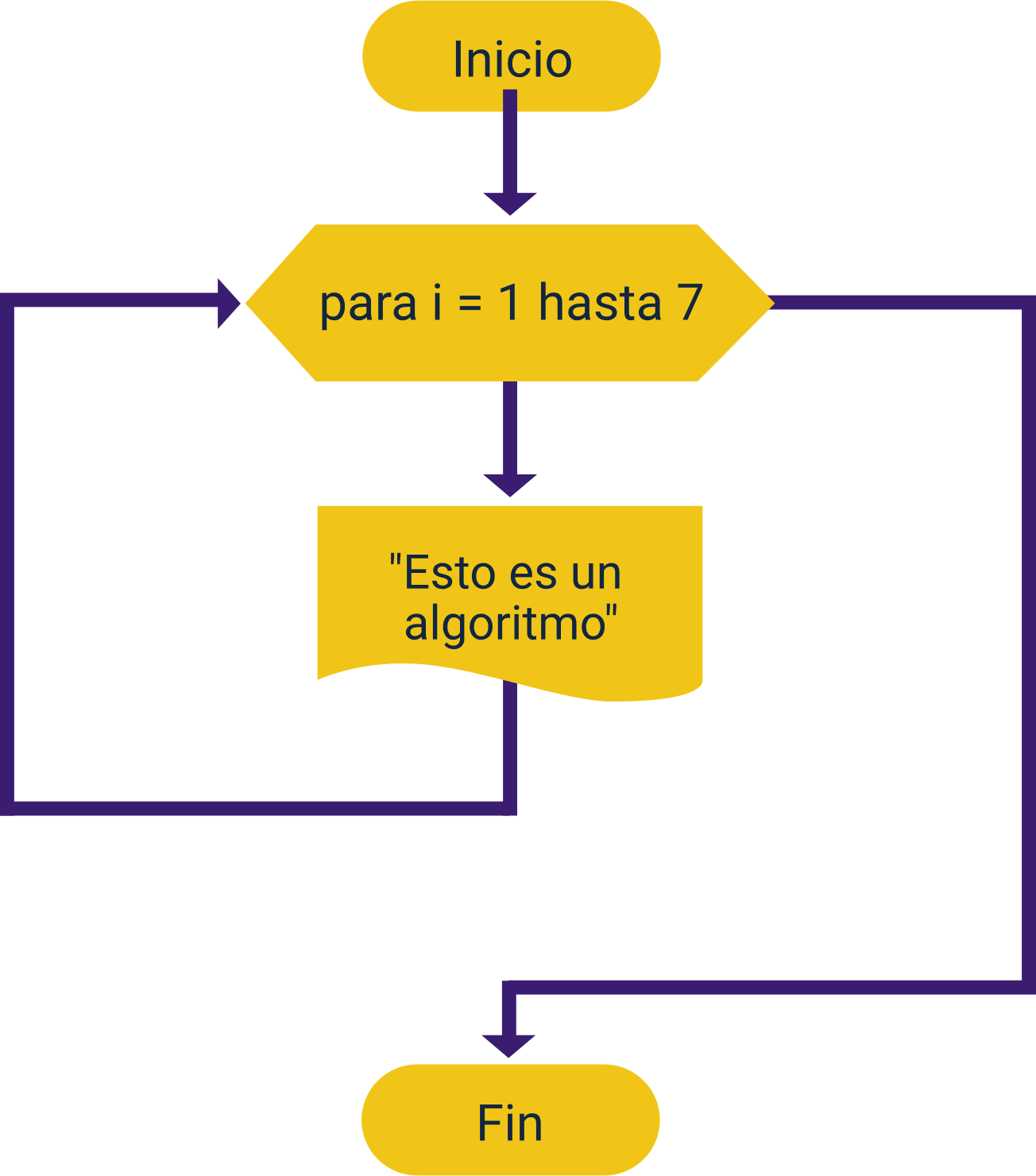
#### Análisis

* **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.

#### Diseño

Como se presenta en el algoritmo de la siguiente figura, existe una variable i tendrá el valor 1, 2, 3 ... hasta 7, esta variable, se puede usar dentro del el bloque PARA hasta FINPARA.

1. Diagrama al área de un triángulo



ALGORITMO Ciclo para;

INICIO

PARA i DESDE 1 HASTA 7

ESCRIBIR( "Esto es un algoritmo" );

FINPARA

FIN

**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

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.

#### 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.

#### Diseño

La siguiente figura presenta el algoritmo en sus dos representaciones más comunes, se propone en secciones posteriores hacer una prueba de escritorio a este problema.

1. Estructura de repetición FOR acumulador suma



ALGORITMO sumas sucesivas;

VAR

ENTERO número;

ENTERO suma;

INICIO

suma = 0;

ESCRIBIR( "Escriba el número" ) ;

LEER ( número

PARA i DESDE 1 HASTA numero

suma = suma + i;

FINPARA

ESCRIBIR ( suma ) ;

FIN

### 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.

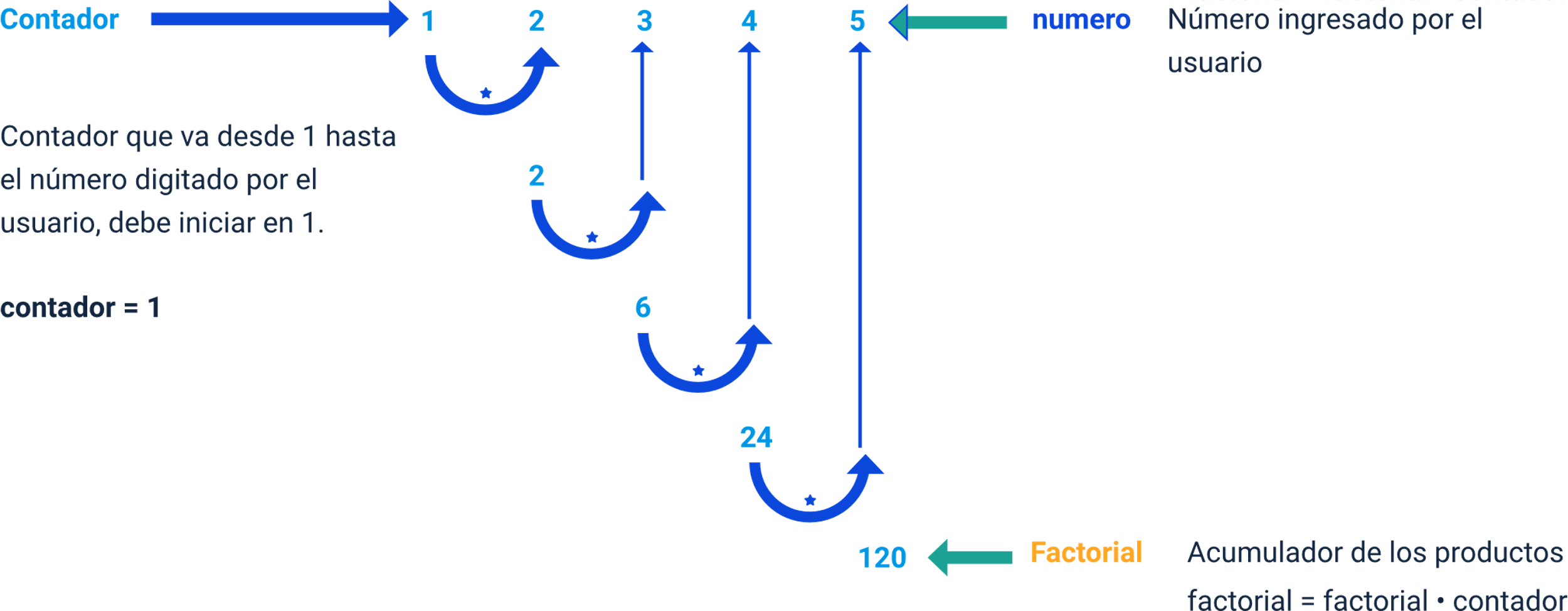
#### Ejemplo:

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 de factorial



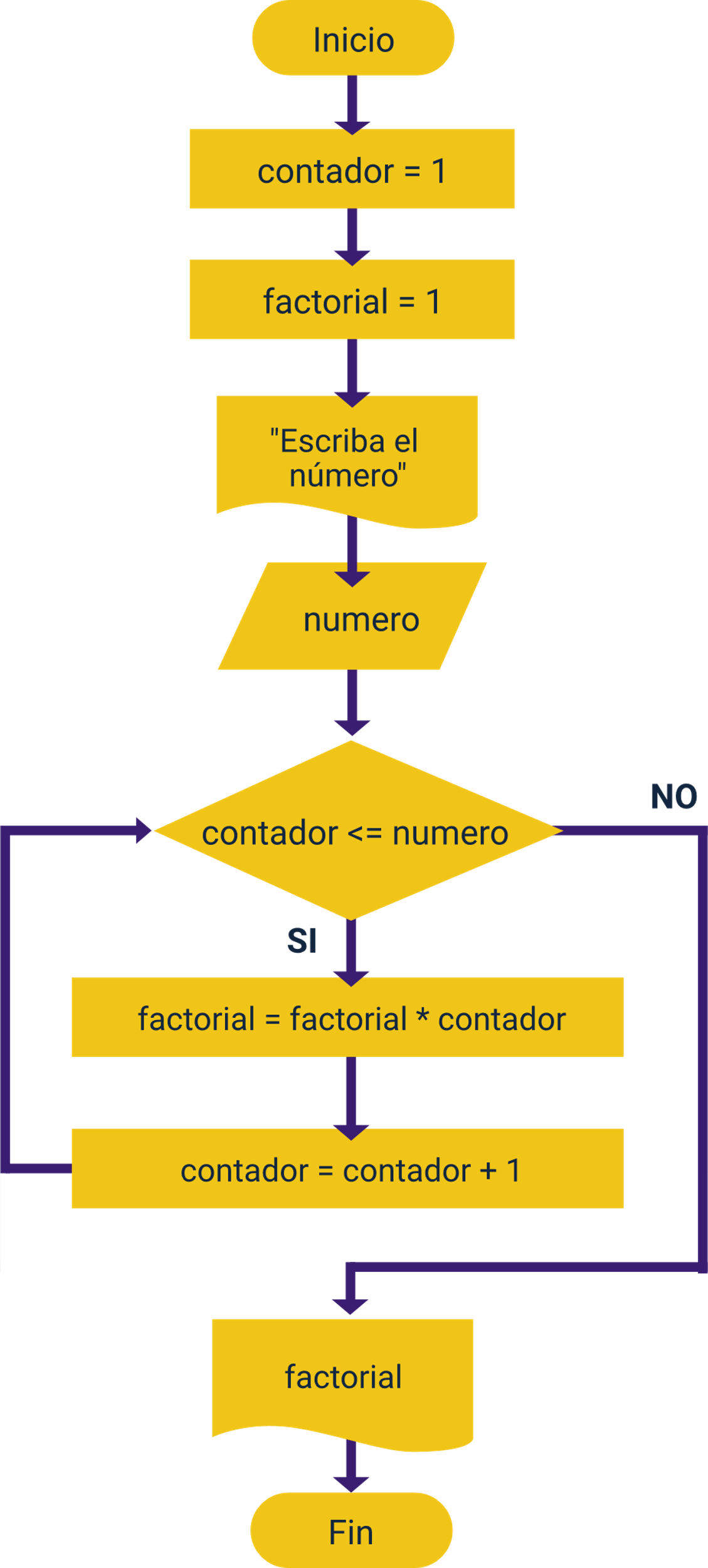
#### Análisis

* **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.

#### Diseño

El diseño es el siguiente:

1. Algoritmo factorial ciclo mientras



ALGORITMO factorial;

VAR

ENTERO contador;

ENTERO factorial;

INICIO

contador = 1;

factorial = 1;

ESCRIBIR( "Escriba el número" ) ;

LEER ( numero ) ;

MIENTRAS

factorial = factorial \* contador;

contador = contador + 1;

FINMIENTRAS

ESCRIBIR( factorial );

FIN

# Estructura de datos

Es importante conocer el concepto de arreglos e identificar cuándo usarlos en la aplicación de los algoritmos. Asimismo, es crucial ser capaz de resolver problemas básicos mediante diagramas de flujo y pseudocódigo. Existen varios tipos de arreglos, siendo los más comunes los vectores y las matrices.

#### Vectores

Los arreglos o arrays (en inglés), también conocidos como matrices, arreglos o vectores, se definen como un conjunto finito y ordenado de elementos del mismo tipo. La propiedad "ordenado" implica que el primer elemento, 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:

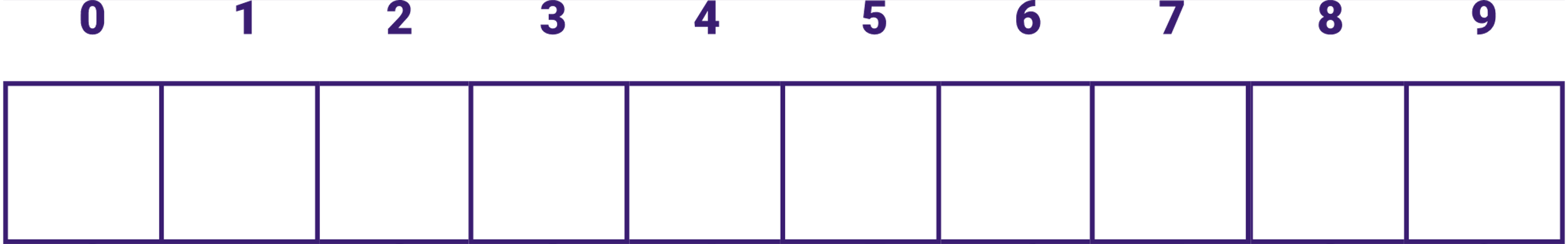
* **Unidimensionales:**

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, lo que 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 empiezan a contarse desde 0, como se presenta en la figura anterior.

Para referirse a 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 de la figura 17, se debe hacer así: lista[4]. Es importante notar que la quinta posición se accede con el índice 4, porque se empieza 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.

#### A continuación, se revisan los ejemplos:

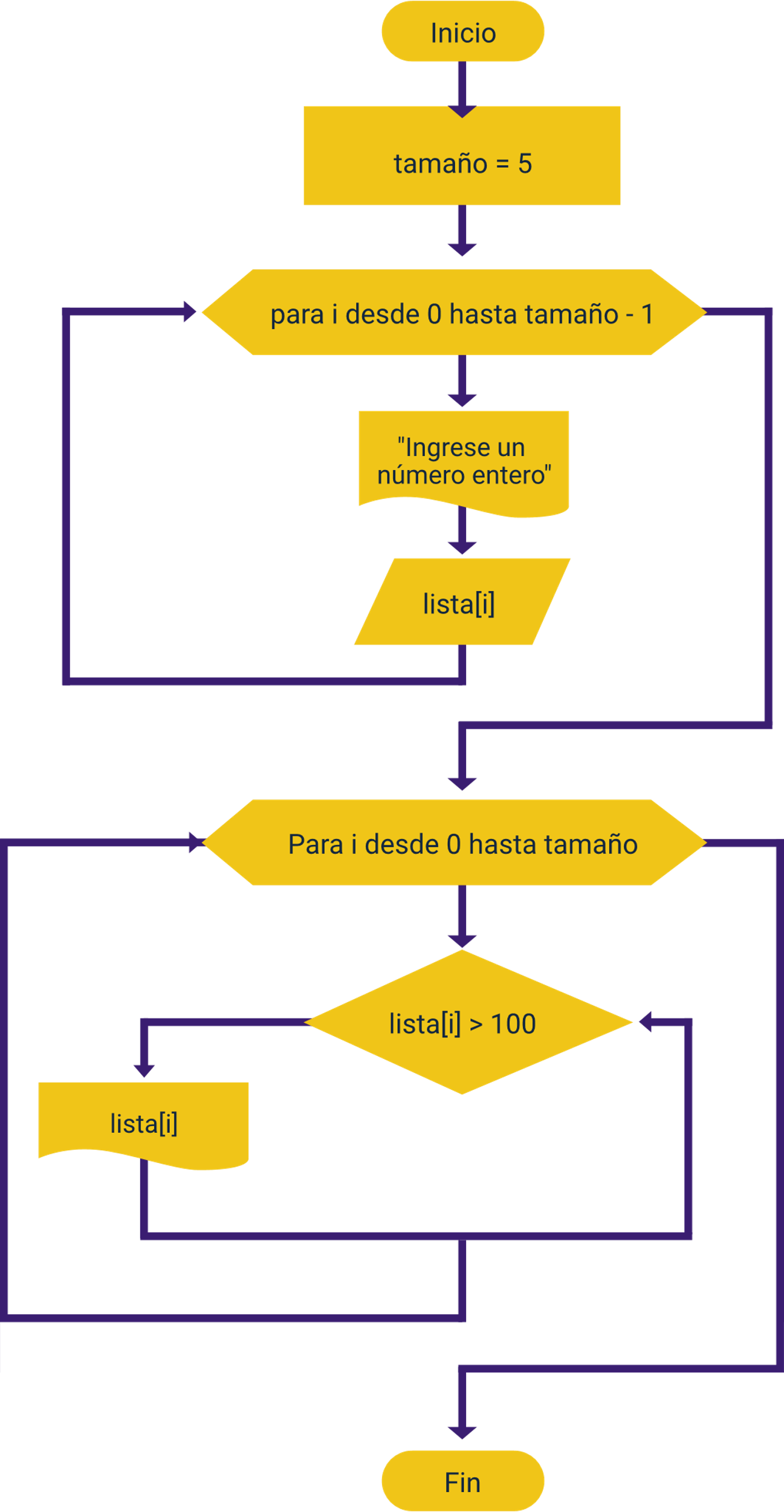
#### Análisis

* **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.

#### Diseño

El diseño es el siguiente:

1. Vector de 5 elementos



ALGORITMO Vector de 5 enteros;

VAR

ENTERO tamano;

ENTERO lista[tamano] ;

INICIO

tamano = 5;

PARA i DESDE 0 HASTA tamano -1

ESCRIBIR( "Escriba el número" );

LEER ( lista[ i ] );

FINPARA

PARA i DESDE 0 HASTA tamano -1

SI ( lista[ i ] > 100 )

ESCRIBIR (lista[ i ] );

FINSI

FINPARA

FIN

### Matrices

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

1. Matriz de 4 x 5

| Columna 1 | Columna 2 | Columna 3 | Columna 4 |
| --- | --- | --- | --- |
| 0,0 | 0,1 | 0,2 | 0,3 |
| 1,0 | 1,1 | 1,2 | 1,3 |
| 2,0 | 2,1 | 2,2 | 2,3 |
| 3,0 | 3,1 | 3,2 | 3,3 |
| 4,0 | 4,1 | 4,2 | 4,3 |

Lo anterior se aplica cuando se quiere acceder a un elemento de la matriz. El identificador usa corchetes dobles. Por ejemplo, para acceder al último elemento de la tabla, se utilizaría: matriz[4][3].

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

#### Programación modular

Existe un área denominada 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 basada 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. Esto 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 almacenándolos para su posterior reutilización cuantas veces sea necesario.

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:

* **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.

* **Según su identificación**

Cada módulo tiene un nombre de identificación que permite ejecutarlo y diferenciarlo de otros. Es recomendable que al elegir este nombre sea representativo de la función que realiza, de esa forma resulta intuitivo en el momento de usarlo dentro de un algoritmo.

* **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.

* **Parámetro de salida**

También se pueden 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.

* **Pruebas de escritorios**

Como se mencionó antes, las pruebas de 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 mayor 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.

#### 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.

#### Diseño:

El diseño es el siguiente:

1. Estructura Algoritmo el mayor de dos números



ALGORITMO El mayor de dos números;

VAR

ENTERO numero1;

ENTERO numero2;

INICIO

ESCRIBIR ( "Escriba un número" ) ;

LEER ( numero1 );

ESCRIBIR( "Escriba un número" ) ;

LEER ( numero2 ) ;

numerol > numero2

ESCRIBIR( numero1 );

SINO

ESCRIBIR( numero2 );

FINSI

FIN

# 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 le 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:

* El código creado debe traducirse de manera que el procesador del computador o smartphone pueda comprenderlo, a este proceso se le llama compilación.
* El lenguaje compilado necesita que antes de ser ejecutado, se convierta el código fuente escrito a un lenguaje de máquina.

#### Las características son:

* **Primero**

Para un lenguaje compilado, luego de compilar un programa se deben crear ejecutables para cada uno de los sistemas operativos en los que se vaya a utilizar. Es decir, un programa compilado en Linux no servirá en Windows o en Mac OS.

* **Segundo**

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.

* **Tercero**

Los lenguajes compilados están optimizados para el momento de la ejecución, pero esto significa más trabajo para el programador (compilarlo).

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

En adelante, JavaScript será el lenguaje de programación abordado. 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 tanto, 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, adicionalmente, se puede instalar el intérprete para el sistema operativo de preferencia. (Framebits, 2020)

Dado lo anterior, existen herramientas, manuales y videos que se emplean para el desarrollo de programas en JavaScript.

# 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 tenga a disposición la documentación técnica oficial del lenguaje y el material complementario propuesto.

En JavaScript se pueden utilizar muchas funciones matemáticas. Por ejemplo, un algoritmo que toma dos números (8 y 6) y les aplica las 4 operaciones básicas puede servir para revisar las características de la sintaxis de JavaScript y conocer las operaciones matemáticas del lenguaje.

Para ello, ingresar en la ruta del navegador a <https://playcode.io/new> y escribir el código de la línea 1 a la 12 en la pestaña script.js, como se presenta a continuación:

let numero1 = 8;

let numero2 = 6;

var suma = numero1 + numero2;

var resta = numero1 - numero2;

var producto = numero1 \* numero2;

var division = numero1 / numero2;

console.log("la suma es: " + suma );

console.log("la resta es: " + resta );

console.log("el producto es: " + producto );

console.log("la división es: " + division );

Consola

la suma es: 14

la resta es: 2

el producto es: 48

la división es: 1.3333333333333333

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 palabras reservadas numero1 y numero2 se les asigna el valor correspondiente. Es de conocer que se definen y se inicializa la variable en una misma línea, esto es una característica del lenguaje.

Luego en las líneas 3 al 6 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 el operador +, usado anteriormente para sumar dos números. Sin embargo, en este caso, no hay dos números, sino una cadena de texto y un número. Aquí, el operador + concatena o junta el texto "la suma es:" y el valor de la variable suma. Esta característica de los operadores se llama sobrecarga de operadores, porque según el contexto, el operador + puede sumar o concatenar.

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

#### Se debe crear un algoritmo que muestre el seno del valor PI (3.1416) multiplicado por 3 y que también eleve 6 a la tercera potencia, mostrando su resultado:

var angulo = Math.PI \* 3;

console.log('seno(3PI) =' + Math.sin(angulo));

var potencia = Math.pow(6, 3);

console.log('6^3 =' + potencia);

Consola

seno(3PI) = -3.6739403974420594e-16

6^3 = 216

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

#### Funciones clase Math

A continuación, puede revisar el listado de funciones de la clase Math.

Lo invitamos a consultar el PDF “**Funciones clase Math**”, el cual se encuentra en la carpeta Anexos.

Todas las funciones matemáticas se pueden combinar con otras operaciones aritméticas o emplear operadores de evaluación de condición. 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:

### a. 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 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:

* **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.

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.

### b. Operadores

Asignar un valor a una variable también se considera una expresión, aunque de un tipo distinto. Para todas estas operaciones se emplean los denominados "operadores". A continuación, se listan 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:

1. Operadores de asignación compuestos

| 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 otros tipos de datos.

1. Operadores de comparación

| Nombre | Operador abreviado | Significado |
| --- | --- | --- |
| 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 devuelven 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.

1. Operadores de aritméticos

| Nombre | Operador abreviado | Significado |
| --- | --- | --- |
| 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 utilizando los operadores lógicos como AND (&&), OR (||), y NOT (!). Estos operadores permiten evaluar múltiples expresiones booleanas en una sola sentencia.

1. Operadores lógicos

| Nombre | Operador abreviado | Significado |
| --- | --- | --- |
| 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. |

### c. 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:

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

## Expresiones y comentarios

JavaScript permite poner comentarios en el código fuente, como la mayoría de los lenguajes de programación. Existen dos tipos de comentarios:

/\*

Comentario que puede tener más de una línea

de texto

\*/

var var1 = 3;

var var2 = 4;

console.log( var2 > var1 ); // Este es un comentario de línea

console.log( var2 < var1 ); // Este es otro comentario de línea

console.log( var1 !== var2 );

Consola

true

false

true

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 que 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

calificacion = 4;

SI ( calificación >= 3)

ESCRIBIR( "Aprobo" ) ;

SINO

ESCRIBIR( "Reprobo" );

FINSI

FIN

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

/\*

Algoritmo calificaciones

\*/

const calificacion = 4;

if (calificacion >= 3) {

console.log('Aprobó');

} else {

console.log('Reprobó');

}

Consola

Aprobó

Como la línea 1 empieza con /\*, todo lo que sigue será un comentario hasta que aparezcan los caracteres \*/ (línea 3). En este comentario se debe escribir qué hace el algoritmo y otra información relevante que se verá más adelante.

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

**Para recordar**: 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:

1. Se usa la palabra reservada if y, entre paréntesis, la condición que se quiere comparar.
2. Se emplea el símbolo de corchete abierto {, que se llama abrir corchete.
3. En la línea 7 se envía la instrucción de imprimir el resultado de que aprobó.
4. En la línea 8 se cierra un bloque de código con el símbolo de corchete cerrado }, lo que indica 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.
5. 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. Enseguida se abre el contexto con el carácter de corchete abierto {.
6. Luego, en la línea 9, se ha puesto la operación de escribir en la consola que ha reprobado.
7. Para finalizar el contexto, se cierra con el carácter de 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:

/\*

Algoritmo saber si es o no mayor de edad

\*/

const edad = 41;

if (edad >= 18) {

console.log('Es mayor de edad');

}

Consola

Es mayor de edad

De acuerdo con el código anterior, no existe un contexto si no se cumple la condición de mayoría de edad; a esta estructura se le 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 un número determinado o indeterminado de veces. A continuación, se ejemplifican las más comunes (FOR, WHILE), que se pueden codificar también en la sintaxis de JavaScript.

### a. 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.

/\*

Algoritmo imprimir 7 veces un mensaje

\*/

for (let i = 0; i < 7; i++) {

console.log('esto es un algoritmo');

}

Consola

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 identificar en la línea 4, se ha usado la palabra reservada for y dentro del paréntesis aparecen tres instrucciones separadas por punto y coma.

* **Primero**

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 indicando que la variable i solo podrá ser accedida dentro del contexto del for.

* **Segundo**

Luego se ve una condición i < 7;. Esta condición significa que, mientras 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).

* **Tercero**

Una vez ha terminado de ejecutar el contexto, se ejecutará la tercera sentencia i++, que es equivalente a i = i + 1. Esto incrementa 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 del contexto (lo que está dentro de los corchetes).

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

#### Para recordar:

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.

/\*

sumar los números que van desde 1 hasta el valor dado

\*/

const cantidad = 5;

var suma = 0;

for (let i = 1; i <= cantidad; i++) {

suma += i; // esto es igual a suma = suma + i;

}

console.log(suma);

Consola

15

Como se puede ver, se define una constante con valor 5 y una variable que será un acumulador donde se almacenarán las sumas. También, note que los valores de i van desde 1 hasta exactamente el valor que tiene la constante (es decir, 5).

### b. 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:

ALGORITMO cálculo de factorial;

VAR

ENTERO contador;

ENTERO factorial;

INICIO

contador = 1;

factorial = 1;

ESCRIBIR ( "Escriba el número" ) ;

LEER (numero ) ;

MIENTRAS contador <= numero HACER

factorial = factorial \* contador;

contador = contador + 1;

FINMIENTRAS

ESCRIBIR ( factorial );

FIN

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

/\*

Comentario que puede tener más de una línea de texto

\*/

var var1 = 3;

var var2 = 4;

console.log(var2 > var1); // Este es un comentario de línea

console.log(var2 < var1); // este es otro comentario de línea

console.log(var1 !== var2);

Consola

true

false

true

Es ideal transcribir el código usando <https://playcode.io/>. Para ello, se debe escribir la condición del while, es decir, contador <= numero, al final, para que el sistema PlayCode no entre en 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 importante recordar que las estructuras de repetición básicas y su sintaxis son comúnmente utilizadas para recorrer estructuras de datos.

## Estructuras de datos

Las estructuras de datos, también conocidas como arreglos, son importantes en el desarrollo de algunos algoritmos y 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.

### a. Vectores

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

1. let frutas = ["Manzana", "Banana", "Pera"];

2.

3. console.log(frutas);

4. console.log(frutas.length);

Console

[ "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 llama al atributo frutas.length, que contiene el tamaño del vector (es decir, 3).

A continuación, se presenta cómo acceder a los elementos de un vector:

1. let arreglo = [5, 7, 23, 12, 43];

2.

3. console.log(arreglo[0]);

4. console.log(arreglo[2]);

5. console.log(arreglo[arreglo.length - 1]);

6. console.log(arreglo[arreglo.length]);

Consola

5

23

43

undefined

Los índices de los vectores 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:

let resultados = [];

for (let i = 1; i <= 10; i++) {

resultados.push(i \* 5);

}

for (let i = 1; i <= 10; i++) {

console.log('5 x ' + i + ' = ' + resultados[i - 1]);

}

Consola

5 x 1 = 5

5 x 2 = 10

5 x 3 = 15

5 x 4 = 20

5 x 5 = 25

5 x 6 = 30

5 x 7 = 35

5 x 8 = 40

5 x 9 = 45

5 x 10 = 50

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. |

### b. 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 se presenta a continuación:

"Manzana" "Pera" "Papaya" "Piña"

"Papa" "Tomate" "Yuca" "Ajo"

"Arroz" "Frijol" "Cebada" "Garbanzo"

let datos = [

["Manzana", "Pera", "Papaya", "Piña"],

["Papa", "Tomate", "Yuca", "Ajo"],

["Arroz", "Frijol", "Cebada", "Garbanzo"]

];

console.log(datos);

### c. Registros

Hasta ahora se ha revisado 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 |
| Maria | maria@sindato.com | 16 | 123 |

Exceptuando el encabezado de la tabla 8, cada fila representa un registro y cada dato de un registro puede tener un tipo de dato diferente. Si se está interesado en codificar la tabla 8, en una estructura con JavaScript el código sería como se tiene en:

let registros = [

["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],

["Maria", "maria@sindato.com", 16, 123]

];

Para recorrer todos los registros se precisan dos (2) ciclos for, uno dentro de otro (anidado) como se presenta a continuación:

let registros = [

["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],

["Maria", "maria@sindato.com", 16, 123]

];

var total = 0;

for (let i = 0; i < registros.length; i++) {

console.log(`Registro #${i + 1}`);

for (let j = 0; j < registros[i].length; j++) {

console.log(" " + registros[i][j]);

}

}

Consola

Registro #1

Juan

juan@sindato.com

19

36.234

Registro #2

Luis

luis@sindato.com

18

23.234

Registro #3

Andrea

andrea@sindato.com

22

0

Registro #4

Pedro

pedro@sindato.com

12

65.234

Registro #5

Maria

maria@sindato.com

16

123

Como se puede identificar, se requieren dos ciclos for anidados, y cada ciclo tiene una variable de índice de nombre diferente i y j y la manera de indexar la matriz vista anteriormente en el ejemplo de registros [ i ][ j ].

## 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 siguientes palabras reservadas: continue, break y return.

* **La 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:

var i = 0;

while (i < 5) {

i++;

console.log(i);

}

Consola

1

2

3

4

5

var i = 0;

while (i < 5) {

i++;

if (i == 3) {

continue;

}

console.log(i);

}

Consola

1

2

4

5

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.

* **Sentencia break**

La sentencia break, detiene la ejecución del ciclo independientemente de cuántas veces este configurando el ciclo; preste especial atención en la salida de consola.

for (let i = 1; i <= 5; i++) {

console.log(i);

}

Consola

1

2

3

4

5

for (let i = 1; i <= 5; i++) {

if (i == 3) {

break;

}

console.log(i);

}

Consola

1

2

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.

* **Sentencia return**

El uso de la sentencia return sirve para terminar la ejecución de un bloque de instrucciones y es muy común en la programación modular cuando se quiere retornar un resultado. A continuación, se presenta un ejemplo para ilustrar su uso:

Sin return

var divideByFour = function (num) {

num / 4;

};

var divided = divideByFour(27);

console.log(divided); // undefined

Con return

var divideByFour = function (num) {

return num / 4;

};

var divided = divideByFour(27);

console.log(divided); // 6.75

## Métodos de ordenamiento y búsqueda

Existen varias técnicas de ordenamiento de vectores que son importantes para realizar búsquedas. Al igual que en la vida real, es más fácil buscar en un lugar que esté ordenado que en uno que no lo esté. JavaScript está diseñado para pequeños algoritmos que resuelven problemas específicos y convertirlos en servicios. El uso de estas técnicas puede no tener mucha aplicabilidad en volúmenes de datos pequeños, ya que la diferencia en tiempos computacionales de ejecución es considerable solo cuando se manejan grandes volúmenes de datos. Para volúmenes de datos pequeños, los tiempos computacionales no presentan diferencias significativas.

### a. Ordenamiento de cadenas de texto

Para ordenar un vector de mayor a menor valor y teniendo en cuenta que el arreglo se modifica al ordenarlo, es importante hacer una copia del arreglo si se desea mantener el vector original inmutable. A continuación, se presenta un ejemplo que ilustra cómo realizar este proceso en JavaScript:

* **Ordenamiento de números como cadenas de texto**

Por defecto el método Array.sort() ordena los elementos del array como si fueran cadenas, inclusive si los datos son de tipo entero, como en este ejemplo:

var arg = [10, 53, 12, 56, 120];

console.log(arg.join(', '));

arg.sort();

console.log(arg.join(', '));

Consola

10, 53, 12, 56, 120

10, 12, 120, 53, 56

* **Ordenamiento de cadenas de texto**

El arreglo a ordenar en la línea 1, en la línea 2, se presenta 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 (línea 2). Luego se aplica la función sort() en la tercera línea de código y, posteriormente, se presenta el vector ya ordenado. Como se presenta, el resultado en consola es un ordenamiento como si los elementos fueran cadenas de texto.

1. var data = ["Montería", "Sincelejo", "Cartagena", "Bogotá"];

2. data.sort();

3. console.log(data);

Consola

["Bogotá", "Cartagena", "Montería", "Sincelejo"]

* **Invertir el orden de los elementos**

En el ejemplo de un ordenamiento de cadenas de texto. También se puede ordenar el arreglo de forma descendente con el método Array.reverse().

var data = ["Montería", "Sincelejo", "Cartagena", "Bogotá"];

data.sort().reverse();

console.log(data);

Consola

["Sincelejo", "Montería", "Cartagena", "Bogotá"]

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 última 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

Como se explica, el ordenamiento con la función Array.sort() se realiza considerando los elementos del arreglo como cadenas de texto. Antes de describir cómo se ordenan con datos numéricos, se explicará qué es una función en JavaScript.

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).

#### Ejemplo

Se quiere construir una función que reciba dos (2) datos numéricos, hay que multiplicar estos dos números, al resultado se le suma el valor de 100 y, por último, retorna la mitad de este resultado.

function operacion(a, b) {

return (a \* b + 100) / 2;

}

var dato1 = operacion(3, 8);

var dato2 = operacion(2, 4);

console.log(dato1);

console.log(dato2);

Consola

62

54

En la línea 1, se usa la palabra reservada function y luego se elige un identificador para la función, que en 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 corchetes) para indicar cuáles son las instrucciones que ejecutará la función. En la línea 2, se indica que la función retorna la multiplicación de a \* b sumando 100 y a este resultado dividido entre 2. En la línea 4, se ve cómo se invoca la función operacion y se le pasan los parámetros 3 y 8 para las variables a y b respectivamente, el resultado es almacenado en la variable dato1. Y la función puede invocarse más de una vez (las que se necesiten).

var arg = [10, 53, 12, 56, 120];

arg.sort(function(a, b) {

return a - b;

});

console.log(arg.join(', '));

Consola

10, 12, 53, 56, 120

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 otro, es muy común usar la resta porque, al restar un número, solo existen tres tipos de respuestas. Cuando a un número a se le resta 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; y si el resultado es cero, es porque a y b son iguales.

Como se puede apreciar, se pasa una función (sin identificador) que recibe dos datos del arreglo (a y b) y los compara con la diferencia (resta) de ellos. A las funciones que no se les pone identificador se les llaman funciones anónimas. También existe otra forma de sintaxis para funciones anónimas, denominada funciones flecha. A continuación, se presenta cómo ordenar un arreglo numérico de manera inversa utilizando la sintaxis de función:

var arg = [10, 53, 12, 56, 120];

arg.sort((a, b) => b - a);

console.log(arg.join(', '));

Consola

120, 56, 53, 12, 10

### c. Ordenamiento de registros

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

1. Registros

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

En esta oportunidad en las líneas 6 y 8, que se puede usar contexto (corchetes) a pesar de que dentro solo hay una sentencia (línea 7), el resultado a retornar es la diferencia entre b.edad y a.edad para que el ordenamiento sea de mayor a menor.

1 var data = [

2 {usuario: 'Mariela', edad: 31, rol: 'SAC'},

3 {usuario: 'Eduardo', edad: 30, rol: 'CEO'},

4 {usuario: 'Andrés', edad: 34, rol: 'Project Manager'}

5 ];

6 data.sort((a, b) => {

7 return b.edad - a.edad;

8 });

9 console.log(data);

Consola

[

{usuario: "Andrés", edad: 34, rol: "Project Manager"},

{usuario: "Mariela", edad: 31, rol: "SAC"},

{usuario: "Eduardo", edad: 30, rol: "CEO"}

]

### d. 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).

const array1 = [5, 12, 8, 130, 44];

const found = array1.find(element => element > 10);

console.log(found);

Consola

12

Se utiliza el identificador elemento para referenciar la variable a la que se aplicará el criterio de búsqueda, que en este ejemplo es encontrar el primer elemento mayor que 10. Muchas veces es importante que, en lugar de retornar un solo valor, el resultado sea más de uno. Por ejemplo, si en el arreglo se retornan los valores mayores que 10, se debe retornar otro vector con los valores resultantes. Para esto, existe la función Array.filter(), que crea un nuevo arreglo con los datos que cumplan con la condición.

const array1 = [5, 12, 8, 130, 44];

const found = array1.filter(element => element > 10);

console.log(found);

Consola

[ 12, 130, 44]

Es importante notar que la consola solo devuelve los datos que cumplen la condición de que el elemento es mayor que 10. A medida que se avanza en el conocimiento de la sintaxis del lenguaje, el código fuente se enriquece con más elementos y, por tanto, aumenta en complejidad. Esto hace necesario conocer herramientas para la depuración de fallos y la corrección de errores en la redacción del código.

# Depuración y fallas de sintaxis

La principal funcionalidad que se requiere en la depuración del código es la de mostrar salidas parciales de datos para verificar el estado de las variables, o tal vez la necesidad de indicar 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 utilizado la consola para verificar datos de depuración mediante el uso de console.log(), que es un método usado para verificar mensajes de confirmación de operaciones o notificaciones de éxito en las operaciones del algoritmo. Sin embargo, también existen otros métodos:

* 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.

const array1 = [5, 12, 8, 130, 44];

console.dir(array1);

console.warn("Mensaje de alerta " + array1[3]);

console.error("Mensaje de alerta " + array1[4]);

Consola

[5, 12, 8, 130, 44]

Mensaje de alerta 130

error: Mensaje de alerta 44

# 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 para probar 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 es enviar un “Error” como se indica a continuación:

throw new Error("No se puede dividir por cero");

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 sección 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.

let numerador = 4;

let denominador = 0;

var resultado = null;

try {

if (denominador === 0) {

throw new Error("No se puede dividir por cero");

} else {

resultado = numerador / denominador;

}

} catch (e) {

console.error(e);

resultado = null;

}

Consola

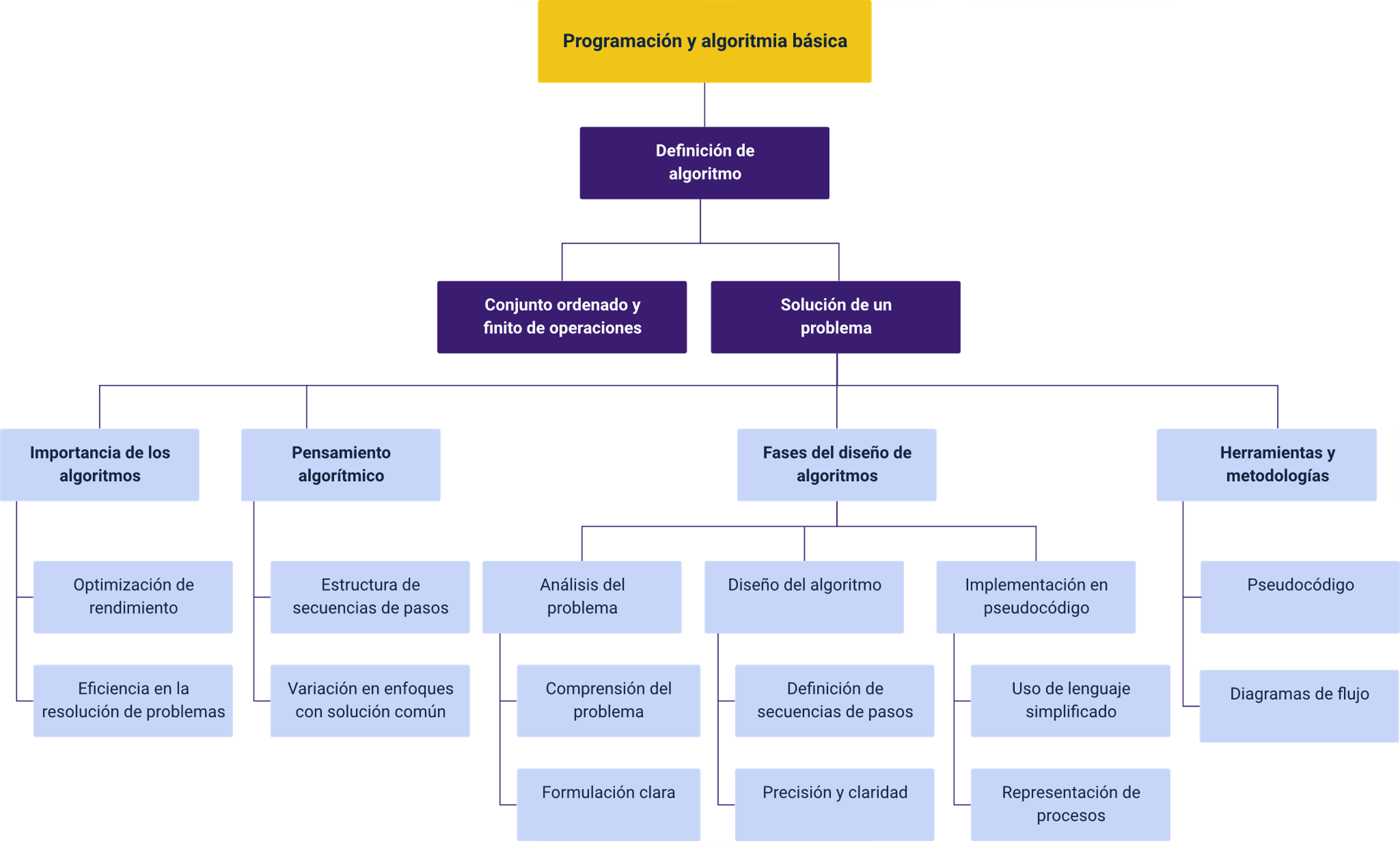
error: Error: No se puede dividir por cero

En el código se describe un algoritmo que divide dos números y, como se sabe, en este proceso se debe validar que no exista una división por 0, ya que no está definida aritméticamente.

Para eso, existe el contexto try que va desde la línea 4 a la línea 10. Dentro de él se verifica el caso de 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, enviando un mensaje de error y asignando a la variable resultado el valor nulo. Así, la excepción es lanzada y capturada, y el programa no se interrumpe abruptamente, sino que es controlado a pesar de los problemas que se puedan presentar.

Síntesis

A continuación, se presenta una síntesis de la temática estudiada en el componente formativo:



Material complementario

| Tema | Referencia | Tipo de material | Enlace del recurso |
| --- | --- | --- | --- |
| Introducción a la algoritmia | Magic Markers. (2015). ¿Qué es un algoritmo? [Video]. YouTube. | Video | <https://www.youtube.com/watch?v=U3CGMyjzlvM> |
| Introducción a la algoritmia | Calderón, J. P. (2015). Fundamentos de lógica matemática. [Video]. YouTube. | Video | <https://www.youtube.com/watch?v=9zYDaZhS7Ac> |
| Diagramas de flujo | Material\_Audiovisual\_100319633. (6 de enero de 2021). App Diagrams.net (básico para diagramas de flujo). [Video]. YouTube. | Video | <https://youtu.be/jdnqmFpqTvo> |
| Diagramas de flujo | Enrique Olivares. (23 de febrero de 2016). Introducción a utilizar el programa DIA para diagramas de flujo a nivel bachillerato. [Video]. YouTube. | Video | <https://youtu.be/DnRx9D7QWg8> |
| Diagramas de flujo | DiscoDurodeRoer. (27 de abril de 2015). Ejercicios PseInt - Básicos #1 - Empezando por lo básico. [Video]. YouTube. | Video | <https://youtu.be/DHIi4dcaMEc> |
| Diagramas de flujo | Insituto de Informática Uach. (08 de septiembre de 2020). Cómo descargar Draw.io 2 para dibujar diagramas de flujo. [Video]. YouTube. | Video | <https://www.youtube.com/watch?v=encz3h8TLWA> |
| Entornos de codificación e instalación JavaScript | Framebits. (19 de enero de 2020). Descargar e instalar Node Js en Windows 10. | Video | <https://youtu.be/v0x1Ku5Tgac> |
| Entornos de codificación e instalación JavaScript | Code Compadre. (30 de junio de 2020). How to Download and Install Visual Studio Code for Windows 10. | Video | <https://youtu.be/KpzkPlh_HsU> |
| Entornos de codificación e instalación JavaScript | VideoMarketingViral. (11 de enero de 2020). Descargar Google Chrome para PC 2021 (WINDOWS 7/8/10). | Video | <https://youtu.be/V-M2rMBhgKc> |
| Entornos de codificación e instalación JavaScript | MDN. (2021). Array - JavaScript. | Documentación técnica | <https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Global_Objects/Math> |
| Entornos de codificación e instalación JavaScript | MDN. (2021). Expresiones y operadores - JavaScript | Documentación técnica | <https://developer.mozilla.org/es/docs/Web/JavaScript/Guide/Expressions_and_Operators> |
| Entornos de codificación e instalación JavaScript | MDN. (2021). Funciones - JavaScript. | Documentación técnica | <https://developer.mozilla.org/es/docs/Web/JavaScript/Guide/Functions> |
| Entornos de codificación e instalación JavaScript | MDN. (2021). Array.prototype.find() - JavaScript. | Documentación técnica | <https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Global_Objects/Array/find> |
| Entornos de codificación e instalación JavaScript | Choque, R. C. (10 de mayo de 2020). Cómo usar la consola de Google Chrome para JavaScript. [Video] YouTube. | Video | <https://youtu.be/Hf3n-p3VYx4> |
| Entornos de codificación e instalación JavaScript | Autodidacta, F. (11 de mayo de 2017). Depurar o hacer debug a JavaScript con Google Chrome. [Video] YouTube. | Video | <https://youtu.be/I388w3wDkjc> |
| Entornos de codificación e instalación JavaScript | Framebits. (19 de enero de 2020). Descargar e instalar Node Js en Windows 10. | Video | <https://youtu.be/v0x1Ku5Tgac> |

Glosario

**Algoritmos**: conjunto ordenado y finito de operaciones que permite hallar la solución de un problema.

**Compilación**: proceso de traducción del código fuente de un programa a un lenguaje de bajo nivel ejecutable.

**Depuración**: proceso de identificar y corregir errores o fallos en un programa informático.

**Diseño**: proceso de planificar la solución a un problema a través de un algoritmo.

**Flujo**: secuencia de pasos que determina el orden en que se ejecutan las instrucciones de un programa.

**Modularidad**: propiedad de un sistema que permite dividirlo en partes más pequeñas y manejables.

**Optimización**: mejora del rendimiento o eficiencia de un programa mediante la reducción de recursos utilizados o el aumento de la velocidad de ejecución.

**Programación**: proceso de crear software escribiendo, probando, depurando y manteniendo el código fuente en un lenguaje de programación.

**Pseudocódigo**: forma simplificada de representar un algoritmo utilizando expresiones del lenguaje natural.

**Variables**: elementos que almacenan datos que pueden cambiar durante la ejecución de un programa.

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>

Eduteka.org. (2008). Sembrando las semillas para una sociedad más creativa. Dr. Mitchel Resnick. [Traducción]. Universidad ICESI. <http://www.eduteka.org/ScratchResnickCreatividad.php>

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

López, J. (2009). Algoritmos y programación (guía para docentes). <https://libros.metabiblioteca.org/bitstream/001/169/8/AlgoritmosProgramacion.pdf>

López, J. (2009). Algoritmos y programación (guía para docentes).

López, J., C. (2009). Educación básica algoritmos y programación. Guía para docentes. Segunda edición. Fundación Gabriel Piedrahita Uribe. <https://libros.metabiblioteca.org/bitstream/001/169/8/AlgoritmosProgramacion.pdf>

MDN. (2021). Array - JavaScript. [https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Global\_Objects/Array#](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. (2021c). Array.prototype.find() - JavaScript. <https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Global_Objects/Array/find>

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/>

Medina, H., M. (2018). Estrategias metodológicas para el desarrollo del pensamiento lógico-matemático. Didasc@lia: Didáctica y Educación, 9(1), p. 125-132. <https://dialnet.unirioja.es/descarga/articulo/6595073.pdf>

Real Academia Española (2020). Algoritmo | Diccionario de la lengua española. RAE. <https://dle.rae.es/algoritmo>

Real Academia Española (2020). Programa | Diccionario de la lengua española. RAE. <https://dle.rae.es/programa>

Resnick, M. (2007). Sembrando semillas para una sociedad más creativa.

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

Rojas, A., V. y Ñacato C., J. (1980). Técnica de flujogramas I. Editora Andina. <https://biblioteca.epn.edu.ec/cgi-bin/koha/opac-detail.pl?biblionumber=27333>

Suppes, P., y Hill, S. (2021). Introducción a la lógica matemática. Editorial Reverté. <https://www.reverte.com/libro/introduccion-a-la-logica-matematica_91496/>

Tapia, L. (1995). Cómo desarrollar el razonamiento lógico matemático. Editorial Universitaria. <https://www.worldcat.org/title/como-desarrollar-el-razonamiento-logico-y-matematico/oclc/50414752>

TOMi.digital. (s.f.). Valores de las tablas de verdad. Colegio Nuestra Señora de las Victorias. <https://tomi.digital/es/59552/valores-de-las-tablas-de-verdad?utm_source=google&utm_medium=seo>

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 | Centro de Formación y Regional |
| --- | --- | --- |
| Milady Tatiana Villamil Castellanos | Responsable del Ecosistema | Dirección General |
| Olga Constanza Bermúdez Jaimes | Responsable de Línea de Producción | Centro de Servicios de Salud - Regional Antioquia |
| Henry Eduardo Bastidas Paruma | Experto Temático | Centro de Teleinformática y Producción Industrial - Regional Cauca |
| Peter Emerson Pinchao | Experto Temático | Centro de Teleinformática y Producción Industrial - Regional Cauca |
| Paola Alexandra Moya Peralta | Evaluadora Instruccional | Centro de Servicios de Salud - Regional Antioquia |
| Yerson Fabián Zárate Saavedra | Diseñador de Contenidos Digitales | Centro de Servicios de Salud - Regional Antioquia |
| Jhon Jairo Urueta Álvarez | Desarrollador Fullstack | Centro de Servicios de Salud - Regional Antioquia |
| Edgar Mauricio Cortés García | Actividad Didáctica | Centro de Servicios de Salud - Regional Antioquia |
| Luis Gabriel Urueta Álvarez | Validador de Recursos Educativos Digitales | Centro de Servicios de Salud - Regional Antioquia |
| Margarita Marcela Medrano Gómez | Evaluador para Contenidos Inclusivos y Accesibles | Centro de Servicios de Salud - Regional Antioquia |
| Daniel Ricardo Mutis Gómez | Evaluador para Contenidos Inclusivos y Accesibles | Centro de Servicios de Salud - Regional Antioquia |