

UD1.2. Lenguajes de programación y algoritmos

Descargar esta sección en PDF

1. Lenguajes de programación

Una de las herramientas que empleamos para desarrollar aplicaciones informáticas son los **lenguajes de programación**. Estos lenguajes nos permiten expresar de forma precisa y estructurada las **instrucciones** u **órdenes** que el ordenador debe seguir para realizar tareas específicas. Es la herramienta fundamental para la creación del software.

- **Programa:** Un programa es un conjunto de instrucciones escritas en un lenguaje de programación que el ordenador puede interpretar y ejecutar. Estas instrucciones definen cómo se deben manipular los datos, realizar cálculos, interactuar con el usuario y controlar el flujo del programa.
- **Programar:** proceso de escribir, probar y mantener el código fuente de un programa.

Hay que tener en cuenta que existen otros lenguajes que permiten definir otros aspectos de las aplicaciones informáticas:

- **Lenguajes de especificación:** definen el comportamiento esperado de un sistema sin entrar en detalles de implementación. Ejemplo: UML (Unified Modeling Language), VHDL (para diseño de circuitos).
- **Lenguajes de marcado:** definen la estructura y presentación de documentos. Ejemplo: HTML (para páginas web), XML (para intercambio de datos), Markdown (para formateo de texto).
- **Lenguajes de consulta:** permiten extraer información de bases de datos. Ejemplo: SQL (Structured Query Language), XQuery (para XML).
- **Lenguajes de transformación:** transforman datos de un formato a otro. Ejemplo: XSLT (para XML), JSONata (para JSON).
- **Lenguajes de estilo:** definen la presentación visual de documentos. Ejemplo: CSS (Cascading Style Sheets) para páginas web, XSL-FO (para XML)

1.1. Características de los lenguajes de programación

Como cualquier lenguaje, los lenguajes de programación tienen una serie de características que los definen y que son comunes a todos ellos:

- Léxico
- Sintaxis
- Semántica

1.1.1. Léxico

El léxico de un lenguaje de programación define el conjunto de símbolos, palabras clave y operadores que se pueden utilizar. Estos elementos son los "ladrillos" básicos con los que se construyen los programas. Por ejemplo, en Python, palabras como `if`, `for`, `while`, `def` son palabras clave que tienen un significado especial. Concretamente, son parte del léxico de un lenguaje:

- **Palabras reservadas:** Son palabras que tienen un significado especial en el lenguaje y no se pueden usar como identificadores (nombres de variables, funciones, etc.). Por ejemplo, en Python, `if`, `else`, `while`, `for`, `def` son palabras reservadas.
- **Identificadores:** Son nombres que los programadores definen para variables, funciones, clases, etc. Deben seguir ciertas reglas (no pueden empezar con un número, no pueden contener espacios, etc.). Por ejemplo, `mi_variable`, `calcular_suma`, `Usuario`.
- **Literales:** Son valores constantes que se utilizan directamente en el código, como números (`42`, `3.14`), cadenas de texto (`"Hola, mundo!"`), booleanos (`True`, `False`), etc.
- **Operadores:** Son símbolos que representan operaciones matemáticas, lógicas o de comparación. Por ejemplo, `+` (suma), `-` (resta), `*` (multiplicación), `/` (división), `==` (igualdad), `!=` (desigualdad).
- **Delimitadores:** Son caracteres que separan diferentes elementos del código, como paréntesis `()`, corchetes `[]`, llaves `{}`, comillas `"` o `'`, y punto y coma `;`. Incluimos también el tratamiento de los espacios en blanco o tabulaciones, cuyo significado puede variar según el lenguaje. Por ejemplo, en Python, la indentación es crucial para definir bloques de código.
- **Comentarios:** Son anotaciones en el código que no afectan su ejecución, pero ayudan a documentar y explicar el propósito del código. En Python, se utilizan `#` para comentarios de una línea y `'''` o `"""` para comentarios de varias líneas; en Java o C++ se utilizan `//` para comentarios de una línea y `/* ... */` para comentarios de varias líneas.

1.1.2. Sintaxis

La sintaxis de un lenguaje de programación define las reglas gramaticales que deben seguirse para escribir instrucciones válidas. Es similar a la gramática de un idioma natural, pero más estricta. Cada lenguaje tiene su propia sintaxis, y un error en la sintaxis puede hacer que el programa no se ejecute. Incluimos en las reglas sintácticas:

- **Construcción de expresiones:** cómo combinar operadores, literales e identificadores para formar expresiones válidas.
- **Estructura de declaraciones:** cómo definir variables, funciones, clases, estructuras de control (condicionales, bucles, etc.) y otros elementos del programa.
- **Formato del código:** convenciones sobre el uso de espacios, tabulaciones, saltos de línea y comentarios para mejorar la legibilidad del código.
- **Uso de delimitadores:** cómo utilizar paréntesis, corchetes, llaves y comillas para agrupar y delimitar diferentes partes del código.
- **Orden de las declaraciones:** reglas sobre el orden en que deben aparecer las declaraciones en el código, como la declaración de variables antes de su uso.

1.1.3. Semántica

La semántica de un lenguaje de programación define el **significado de las instrucciones y expresiones**. Es decir, qué hacen realmente cuando se ejecutan. Mientras que la sintaxis se ocupa de la forma del código, la semántica se ocupa del comportamiento del programa. Incluye:

- **Significado de las expresiones:** cómo se evalúan las expresiones y qué resultados producen.
- **Comportamiento de las estructuras de control:** cómo se ejecutan las instrucciones condicionales, bucles y otras estructuras de control.
- **Ámbito y visibilidad de las variables:** reglas sobre dónde y cómo se pueden utilizar las variables, funciones y otros identificadores en el código.
- **Manejo de errores y excepciones:** cómo se gestionan los errores en tiempo de ejecución y qué sucede cuando se producen excepciones.
- **Interacción con el entorno:** cómo el programa interactúa con el sistema operativo, la memoria, los dispositivos de entrada/salida y otros recursos.

1.2. Tipos de lenguajes de programación

Los lenguajes de programación tienen diferentes características y, por lo tanto, se pueden clasificar según diferentes criterios:

Según su nivel de abstracción

- **Lenguajes de bajo nivel:** se acercan al hardware y son difíciles de entender para los humanos (ej.: ensamblador). Ejemplo:

- **Lenguajes de nivel medio:** combinan características de bajo y alto nivel, permitiendo un control más directo del hardware (ej.: C). Suelen emplearse para sistemas operativos, controladores y

aplicaciones de alto rendimiento.

- **Lenguajes de alto nivel:** son más abstractos y fáciles de entender, ocultando detalles del hardware (ej.: Python, Java, C#).

Estos son extractos para la misma operación ("hola mundo") en diferentes lenguajes:

```
# Python
print("Hola, mundo!")
```

```
// Java
public class HolaMundo {
    public static void main(String[] args) {
        System.out.println("Hola, mundo!");
    }
}
```

```
// C
#include <stdio.h>
int main() {
    printf("Hola, mundo!\n");
    return 0;
}
```

```
// C++
#include <iostream>
int main() {
    std::cout << "Hola, mundo!" << std::endl;
    return 0;
}
```

```
// JavaScript
console.log("Hola, mundo!");
```

Segun su forma de ejecución

Según cómo se ejecutan los programas escritos en ellos, los lenguajes de programación se pueden clasificar en:

- **Lenguajes compilados:** requieren un compilador que traduce el código fuente a código máquina antes de su ejecución. Esto suele resultar en un mejor rendimiento, pero requiere un paso adicional de compilación. Ejemplos: C, C++, Rust.

- **Lenguajes interpretados:** no requieren un compilador, sino que un intérprete ejecuta el código fuente directamente. Esto permite una mayor flexibilidad y facilidad de depuración, pero puede resultar en un rendimiento inferior. Ejemplos: Python, Ruby, JavaScript.
- **Lenguajes híbridos:** combinan características de compilación e interpretación. Por ejemplo, Java se compila a bytecode que luego es interpretado por la máquina virtual de Java (JVM). Otros lenguajes, como C#, utilizan un enfoque similar con el Common Language Runtime (CLR).

Proceso y fases de compilación


Cuando escribimos un programa en un lenguaje como **C**, **C++** o **Java**, el ordenador no puede ejecutarlo directamente. Ese código fuente necesita pasar por varias transformaciones hasta convertirse en código máquina ejecutable. Este proceso se conoce como **compilación**, y está compuesto por varias fases bien diferenciadas.

Aunque muchas veces todo este proceso parece automático, conocer sus fases es fundamental para entender errores comunes y optimizar el rendimiento del software.

A continuación describimos las fases más importantes:

Análisis léxico (scanner)

El compilador **lee el código fuente** y lo divide en **unidades mínimas significativas** llamadas *tokens*. Los tokens pueden ser palabras clave (`if`, `while`), identificadores (`x`, `nombre`), símbolos (`{`, `}`, `=`, `+`), constantes (`42`, `"Hola"`) o comentarios.


 **Objetivo:** separar y clasificar el texto en piezas que el compilador pueda procesar.

 **Errores detectables:**

- Caracteres no válidos
- Identificadores mal formados


Análisis sintáctico (parser)

Se analiza si los tokens **siguen las reglas gramaticales del lenguaje**. El compilador construye un **árbol de sintaxis abstracta (AST)** que representa la estructura del programa.

 **Objetivo:** validar que las instrucciones están correctamente escritas desde el punto de vista de la sintaxis.


 **Errores detectables:**

- Llaves o paréntesis desbalanceados
- Instrucciones mal estructuradas (`if x = 5` en lugar de `if (x == 5)`)

 **Ejemplo visual:** `if (x > 10) { imprimir(); }` se convierte en un árbol que representa la estructura condicional y su bloque de código.

Análisis semántico

Aquí se comprueba que las construcciones tienen **sentido lógico**. El compilador verifica que las variables estén declaradas, que los tipos de datos coincidan, que las operaciones sean válidas, etc.

 **Objetivo:** detectar errores de significado, no solo de forma.

 **Errores detectables:**

- Usar variables sin declarar
- Sumar una cadena con un número sin conversión
- Llamar funciones con argumentos incorrectos


Generación de código intermedio

Una vez validado el código, el compilador genera una representación más abstracta que aún no es código máquina, pero está más cerca de él. Este paso facilita la optimización y la portabilidad entre plataformas.

- En Java: se genera **bytecode** (`.class`)
- En C: se crea **código objeto** (`.o`)

Optimización

El compilador analiza el código intermedio para **mejorarlo sin alterar su funcionalidad**. Esto puede implicar eliminar instrucciones innecesarias, reorganizar bucles o variables, y aprovechar mejor el hardware.


 **Objetivo:** generar un código más rápido, más compacto o más eficiente.

 Este paso **no es obligatorio**, pero está presente en compiladores avanzados (GCC, LLVM).

Generación de código máquina

En esta fase se transforma el código intermedio en **instrucciones comprensibles para el procesador** (lenguaje máquina). El resultado puede ser:

- Un archivo binario ejecutable (`.exe`, `.out`)
- Un módulo compilado (`.dll`, `.so`) para enlazar con otros programas

 En Java o .NET, esta fase se realiza **en tiempo de ejecución**, por eso se dice que tienen compilación “just in time” (JIT).

Enlazado (linking)

Finalmente, se combinan todas las piezas del programa: funciones, bibliotecas externas, recursos... El enlazador genera el archivo final ejecutable con todas sus dependencias resueltas.

Errores comunes:

- Llamar funciones definidas en bibliotecas no incluidas
- Faltan archivos o símbolos

Ventajas y desventajas de cada forma de ejecución

En general, el ciclo de desarrollo (el tiempo entre el momento en que escribes el código y lo pruebas) es más rápido en un lenguaje interpretado. Eso se debe a que en lenguajes compilados es necesario realizar el proceso de compilación cada vez que cambias el código fuente, aunque con herramientas adicionales se puede automatizar.

Otra desventaja de un lenguaje compilado es que cuando compilas un programa debes crear ejecutables para cada uno de los sistemas operativos en los que lo vayas a utilizar. Un ejecutable creado para Linux no va a servir en Windows por ejemplo. Sin embargo, un lenguaje compilado es mucho más rápido que uno interpretado. Esto se debe a que cuando es ejecutado ya se encuentra en código de máquina y eso también le permite hacer algunas optimizaciones que no son posibles con un lenguaje interpretado.

Además de la velocidad, otra desventaja de un lenguaje interpretado es que, para ser ejecutado, debes tener instalado el interpretador. Esto no es necesario en un lenguaje compilado que es convertido a lenguaje de máquina.

En general, un lenguaje compilado está optimizado para el momento de la ejecución, aunque esto signifique una carga adicional para el programador. Por otro lado, un lenguaje interpretado está optimizado para hacerle la vida más fácil al programador, aunque eso signifique una carga adicional para la máquina.

Segun su paradigma de programación

Entendemos por paradigma de programación un conjunto de conceptos y prácticas que definen cómo se estructura y organiza el código en un lenguaje. Los paradigmas más comunes son:

- **Programación imperativa:** se basa en la secuencia de instrucciones que cambian el estado del programa. Los lenguajes imperativos incluyen C, C++, Java, Python, entre otros.
- **Programación declarativa:** se centra en describir qué se quiere lograr sin especificar cómo hacerlo. Los lenguajes declarativos incluyen SQL (para bases de datos).

Paradigmas de programación imperativa

Dentro de la programación imperativa, que sigue siendo la más común y extendida, existen varios paradigmas que definen diferentes enfoques para estructurar el código:

- **Programación estructurada:** se basa en el uso de estructuras de control como secuencia, selección e iteración para organizar el flujo del programa. Promueve la claridad y la legibilidad del código. Ejemplos: C, Pascal.

- **Programación orientada a objetos (POO):** organiza el código en objetos que encapsulan datos y comportamientos. Facilita la reutilización y la modularidad. Ejemplos: Java, C++, Python.
- **Programación concurrente:** se centra en la ejecución simultánea de múltiples tareas o procesos. Utiliza conceptos como hilos, procesos y sincronización. Ejemplos: Go, Erlang.
- **Programación reactiva:** se basa en la propagación de cambios y la reacción a eventos. Es común en aplicaciones que manejan flujos de datos y eventos asíncronos. Ejemplos: RxJava, ReactiveX.
- **Programación orientada a eventos:** se basa en la respuesta a eventos o acciones del usuario. Es común en aplicaciones gráficas y sistemas interactivos. Ejemplos: JavaScript (en el contexto de desarrollo web), C# (en aplicaciones de escritorio).

Es importante destacar que muchos lenguajes modernos combinan características de varios paradigmas, lo que permite a los desarrolladores elegir el enfoque más adecuado para cada situación. Por ejemplo, Python admite programación estructurada, orientada a objetos y funcional, lo que lo convierte en un lenguaje versátil y popular.

Según su tipado

En todos los lenguajes de programación, los datos se clasifican en diferentes tipos, como enteros, cadenas de texto, booleanos, etc. Según cómo se manejen estos tipos de datos, los lenguajes se pueden clasificar en:

- **Lenguajes de tipado estático:** los tipos de datos se definen en tiempo de compilación y no pueden cambiar durante la ejecución del programa. Esto permite detectar errores de tipo antes de ejecutar el código. Ejemplos: C, C++, Java, Rust. Esto significa, por ejemplo, que si definimos una variable de tipo entero en Java:

```
int numero = 5;
```

Si intentamos asignarle un valor de tipo cadena, como:

```
numero = "HoLa";
```

El compilador generará un error, ya que el tipo de dato no coincide con la definición original.

- **Lenguajes de tipado dinámico:** los tipos de datos se determinan en tiempo de ejecución, lo que permite mayor flexibilidad pero puede llevar a errores que solo se detectan al ejecutar el programa. Ejemplos: Python, JavaScript, Ruby. Esto implica que, por ejemplo, en Python podemos definir una variable sin especificar su tipo:

```
numero = 5
```

Y luego cambiar su valor a una cadena sin problemas:


```
numero = "HoLa"
```

Esto es posible porque Python determina el tipo de dato en tiempo de ejecución.

Según su ámbito de aplicación

Los lenguajes de programación también se pueden clasificar según el ámbito o tipo de aplicaciones para las que están diseñados:

- **Lenguajes de propósito general:** son versátiles y se pueden utilizar para una amplia variedad de aplicaciones. Ejemplos: Python, Java, C++, JavaScript.
- **Lenguajes de propósito específico:** están diseñados para un dominio o tarea particular. Por ejemplo, SQL para bases de datos.
- **Lenguajes de scripting:** se utilizan para automatizar tareas y scripts. Son interpretados y suelen ser más fáciles de aprender. Ejemplos: Python, Bash, Perl.
- **Lenguajes de sistemas:** se utilizan para desarrollar sistemas operativos, controladores y software de bajo nivel. Ejemplos: C, C++.
- **Lenguajes de programación científica:** están diseñados para cálculos científicos y matemáticos. Ejemplos: MATLAB, R, Julia.

Ranking de lenguajes de programación

El interés y uso por los diferentes lenguajes de programación varía con el tiempo y según el contexto. Existen diferentes rankings que miden la popularidad de los lenguajes de programación, como el índice TIOBE, RedMonk o Stack Overflow Developer Survey. Estos rankings se basan en diferentes métricas, como la cantidad de búsquedas en Google, la actividad en GitHub, las preguntas en Stack Overflow, entre otros.

Índice TIOBE

El índice TIOBE es un ranking mensual que mide la popularidad de los lenguajes de programación en función de la cantidad de búsquedas en Google, Bing, Yahoo!, Wikipedia, Amazon, YouTube y Baidu. El índice TIOBE se actualiza mensualmente y proporciona una visión general de las tendencias en el uso de lenguajes de programación. Se puede consultar en [este enlace](#). A julio de 2025, este es el ranking de los 10 primeros lenguajes:

Julio 2025	Julio 2024	Cambio	Lenguaje de programación	Tasa de uso	Cambio
1	1		Python	26.98%	+10.85%
2	2		C++	9.80%	-0.53%
3	3		C	9.65%	+0.16%
4	4		Java	8.76%	+0.17%

Julio 2025	Julio 2024	Cambio	Lenguaje de programación	Tasa de uso	Cambio
5	5		C#	4.87%	-1.85%
6	6		JavaScript	3.36%	-0.43%
7	7		Go	2.04%	-0.14%
8	8		Visual Basic	1.94%	-0.13%
9	24	↑	Ada	1.77%	+0.99%
10	11	↓	Delphi/Object Pascal	1.77%	-0.12%

Es muy interesante también ver la evolución histórica:

Algoritmos

Un algoritmo es una **secuencia precisa de instrucciones** que, ejecutadas adecuadamente, dan lugar al resultado deseado en un número finito de pasos. Ejemplos de algoritmos no informáticos serían una receta de cocina o los planos con las instrucciones para construir una casa. Un algoritmo cumple siempre las siguientes propiedades o características:

- **Finitud:** Tiene un número finito de pasos.
- **Orden:** presentan una secuencia clara y precisa para poder llegar a la solución del problema.
- **Precisión:** debe indicar de forma inequívoca y sin ambigüedades qué se debe hacer en cada paso.
- **Efectividad:** Cada paso debe ser realizable en un tiempo finito y con recursos finitos.
- **Determinismo:** Para una entrada dada, el algoritmo produce siempre la misma salida.

No debemos confundir un **algoritmo** con un . Un algoritmo es una descripción abstracta de un proceso, mientras que un programa es una **implementación concreta** de ese algoritmo en un **lenguaje de programación** específico.

Todo algoritmo está formado por un conjunto de elementos:

- **Datos:** son el conjunto de valores, de diferente naturaleza y tipo, con los que trabaja el algoritmo.

- **Datos de entrada:** son los datos iniciales que se introducen en el algoritmo.
- **Datos de salida:** son los resultados obtenidos al finalizar el algoritmo.
- Los datos pueden ser:
 - **Simples:** cuando no se pueden descomponer en partes más pequeñas (ej.: un número entero, un número decimal, un booleano, un carácter).
 - **Compuestos:** cuando están formados por varios datos simples o por otros compuestos (ej.: una cadena de texto, un vector, una matriz, un registro).
- **Operandos:** son los datos sobre los que se realizan operaciones. Estos pueden ser:
 - **Constantes:** valores fijos que no cambian durante la ejecución del algoritmo.
 - **Literales:** valores constantes a los que nos referimos directamente por su valor (ej.: 5, "Hola", True).
 - **Variables:** son espacios de memoria que pueden contener diferentes valores durante la ejecución del algoritmo. Se definen con un nombre y pueden cambiar su valor a lo largo del tiempo.
- **Operadores:** son acciones que se realizan sobre los operandos. Pueden ser de múltiples tipos:
 - **Aritméticos:** realizan operaciones matemáticas (ej.: suma, resta, multiplicación, división).
 - **Lógicos:** combinan valores booleanos (ej.: AND, OR, NOT).
 - **De comparación o relacionales:** comparan dos valores y devuelven un valor booleano (ej.: igual a, mayor que, menor que).
 - **De asignación:** asignan un valor a una variable (ej.: `=` en muchos lenguajes).
 - **Binarios o bitwise:** operan a nivel de bits (ej.: AND bit a bit, OR bit a bit, XOR bit a bit).
 - **Unarios:** operan sobre un solo operando (ej.: negación, incremento, decremento).

- **Otros:** dependiendo del tipo de dato con el que estemos trabajando, pueden existir operadores específicos (ej.: concatenación de cadenas, acceso a elementos de una lista o matriz).
- **Expresiones:** son combinaciones de operandos y operadores que producen un valor. Pueden ser simples (un solo operando) o complejas (varios operandos y operadores combinados).
- **Instrucciones:** son las acciones que se realizan en el algoritmo. Pueden ser de diferentes tipos:
 - **De declaración:** definen variables, constantes o estructuras de datos.
 - **Primitivas:** realizan operaciones básicas
 - **Asignación:** asigna un valor a una variable.
 - **Entrada/salida:** lee datos de entrada o muestra resultados.
 - **De control:** determina el orden de ejecución de las instrucciones
 - **Secuenciales:** ejecutan un bloque de instrucciones en un orden específico, una tras otra.
 - **Condicionales o selectivas:** permiten tomar decisiones basadas en condiciones (ej.: `if`, `else`, `switch`).
 - **Iterativas o repetitivas:** repiten un bloque de instrucciones mientras se cumpla una condición (ej.: `for`, `while`, `do-while`).
 - **De salto:** alteran el flujo normal de ejecución del algoritmo (ej.: `break`, `continue`, `return`).

El teorema de Böhm-Jacopini establece que cualquier algoritmo puede ser expresado utilizando únicamente **tres estructuras de control: secuencia, selección e iteración**. Esto significa que cualquier algoritmo puede ser implementado en un lenguaje de programación que soporte estas estructuras.

Esto implica que no son necesarias las instrucciones de salto (como `goto`) para expresar cualquier algoritmo, lo que favorece la claridad y mantenibilidad del código. Veremos que los lenguajes de programación modernos suelen evitar el uso de instrucciones de salto y se centran en las estructuras de control mencionadas.

Eficiencia de los algoritmos

Pueden definirse infinidad de algoritmos que resuelvan un mismo problema, pero interesa encontrar el que sea más **eficiente** en cuanto a dos factores fundamentales:

- **Tiempo de ejecución**
- **Recursos necesarios** para implantar el algoritmo.

Existen dos formas de averiguar la eficiencia de un algoritmo:

- **Empírica:** consiste en ejecutar un algoritmo dado con distintas entradas.
- **Teórica:** consiste en determinar matemáticamente la cantidad de recursos que necesita un algoritmo mediante una función. Tiene la ventaja de no depender del ordenador y del lenguaje

de programación que se utilice. En base a esta función, se define un orden de complejidad denominado O mayúscula.

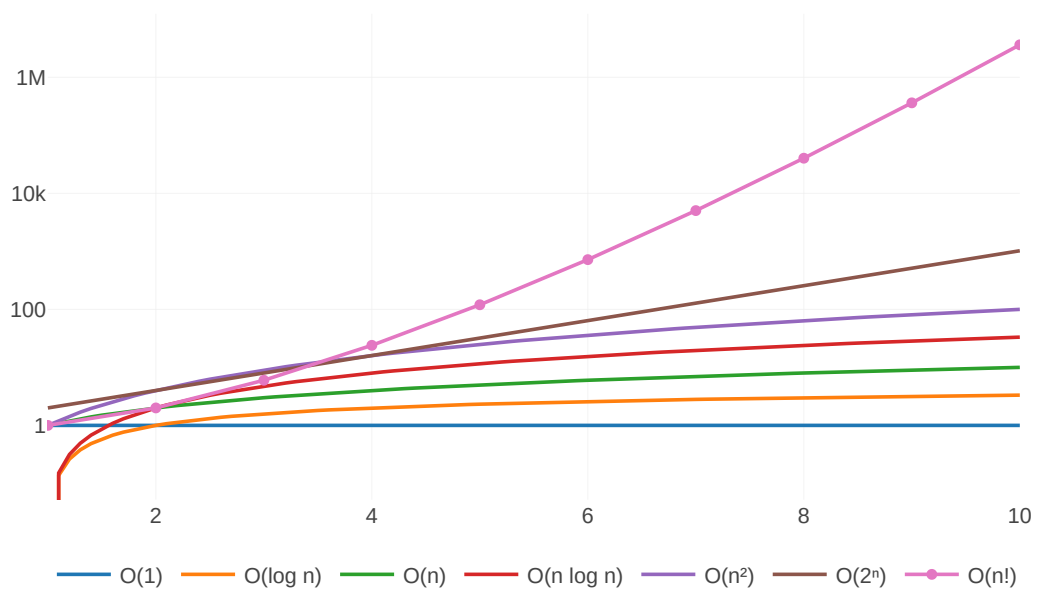
La **complejidad algorítmica** representa la cantidad de recursos (temporales y espaciales) que necesita un algoritmo para resolver un problema y, por tanto, permite determinar su eficiencia. Esta medida del tiempo debe ser independiente de:

- La maquina
- El lenguaje de programación
- El compilador
- Cualquier otro elemento HW/SW que influya en su análisis.

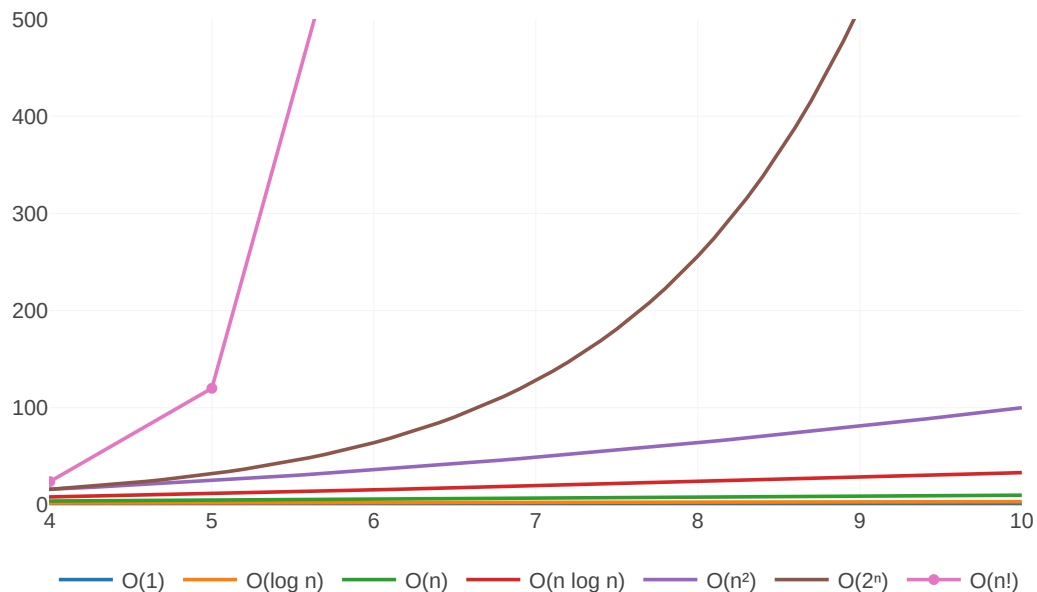
La **complejidad temporal asintótica** es el **comportamiento límite** conforme el tamaño del problema se **incrementa**. Se toma el tamaño de la **entrada** n para medir los requisitos de tiempo de un algoritmo, y el tiempo de ejecución se expresa como una función de la entrada $T(n)$. El orden de un algoritmo indica el grado de la complejidad de un algoritmo. Por ejemplo, un algoritmo cuya función de tiempo sea $T(n) = (n^2 - n)/2$ es de orden $O(n^2)$. Existen diferentes ordenes de algoritmos, expresados con la notación $O(n)$. A continuación, se muestran los órdenes más comunes:

Orden	Descripción
$O(1)$	Tiempo constante: el tiempo de ejecución no depende del tamaño de la entrada.
$O(\log n)$	Tiempo logarítmico: el tiempo de ejecución crece logarítmicamente con el tamaño de la entrada.
$O(n)$	Tiempo lineal: el tiempo de ejecución crece linealmente con el tamaño de la entrada.
$O(n \log n)$	Tiempo lineal-logarítmico: el tiempo de ejecución crece linealmente con el tamaño de la entrada, multiplicado por el logaritmo del tamaño de la entrada.
$O(n^2)$	Tiempo cuadrático: el tiempo de ejecución

Orden	Descripción
	crece cuadráticamente con el tamaño de la entrada.
$O(n^k)$	Tiempo polinómico: el tiempo de ejecución crece polinómicamente con el tamaño de la entrada, donde k es una constante.
$O(2^n)$	Tiempo exponencial: el tiempo de ejecución crece exponencialmente con el tamaño de la entrada.
$O(n!)$	Tiempo factorial: el tiempo de ejecución crece factorialmente con el tamaño de la entrada.



Gráfica de complejidades algorítmicas en escala logarítmica



Gráfica de complejidades algorítmicas en escala lineal

Problemas tipo P y NP

En este contexto de caracterización de la complejidad de un algoritmo, decimos que un problema es de **clase P** si puede ser resuelto por una **máquina determinista en un periodo de tiempo polinomial** en función a los datos de entrada, es decir, si la máquina es capaz de entregar la respuesta correcta para una entrada de longitud n en cn^k con c y n constantes independientes del conjunto de datos. De forma cualitativa, decimos que los problemas de tipo P pueden ser resueltos de manera “razonablemente rápida”.

Por otra parte, se dice que un problema es de tipo NP si, dada una respuesta, se puede verificar en un tiempo polinomial (cn^k con c y n constantes), pero no es posible resolver dicho problema en tiempo polinomial. Cualitativamente, son problemas “muy difícilmente resolubles” por una máquina, ya que tardaría demasiado tiempo en encontrar la solución. Esta clase de problemas son considerados como ejercicios sin solución (indecibles) e irresolubles (intratables). Esta característica se aprovecha en cierto modo, por ejemplo, en los actuales modelos de cifrados empleados en criptografía, de manera que obtener una clave sea “imposible” en un tiempo razonable, sin embargo, verificarla lleve menos de un segundo.

Uno de los problemas del milenio, establecidos por el Clay Mathematics Institute en el año 2000, y premiado con un millón de euros para quien lo resuelva, consiste en decidir si la inclusión entre las clases de complejidad P y NP es estricta. A día de hoy, no ha podido ser demostrado todavía.

Técnicas descriptivas de algoritmos

Para poder describir un algoritmo, es decir, describir los pasos que se deben seguir para resolver un problema, existen diferentes técnicas descriptivas. Las más comunes son:

- **Diagramas de flujo:** son representaciones gráficas que muestran el flujo de control de un algoritmo. Utilizan símbolos estandarizados para representar diferentes tipos de acciones y decisiones.
- **Pseudocódigo:** es una forma de escribir algoritmos utilizando un lenguaje similar al lenguaje de programación, pero sin preocuparse por la sintaxis exacta. Permite expresar la lógica del algoritmo de manera clara y concisa.
- **Lenguajes de programación:** se pueden utilizar lenguajes de programación específicos para describir algoritmos. Esto permite implementar el algoritmo directamente en un lenguaje que pueda ser ejecutado por una máquina.
- **Otros diagramas:** existen otras técnicas gráficas como los diagramas de casos de uso, diagramas de clases, diagramas de secuencia, etc., que pueden ser útiles para describir algoritmos en contextos específicos: diagramas N-S, tablas de decisión, etc.

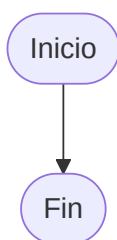
Diagramas de flujo

Los diagramas de flujo permiten describir algoritmos de forma visual, utilizando símbolos estandarizados para representar diferentes tipos de acciones y decisiones. Son especialmente útiles para representar el flujo de control de un algoritmo y para identificar posibles errores o mejoras en su diseño.

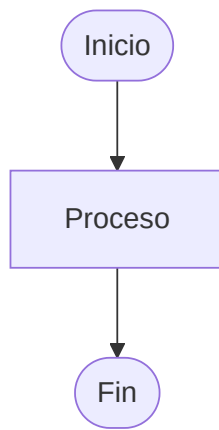
Símbolos básicos

Los diagramas de flujo utilizan una serie de símbolos estandarizados para representar diferentes tipos de acciones y decisiones. Algunos de los símbolos más comunes son:

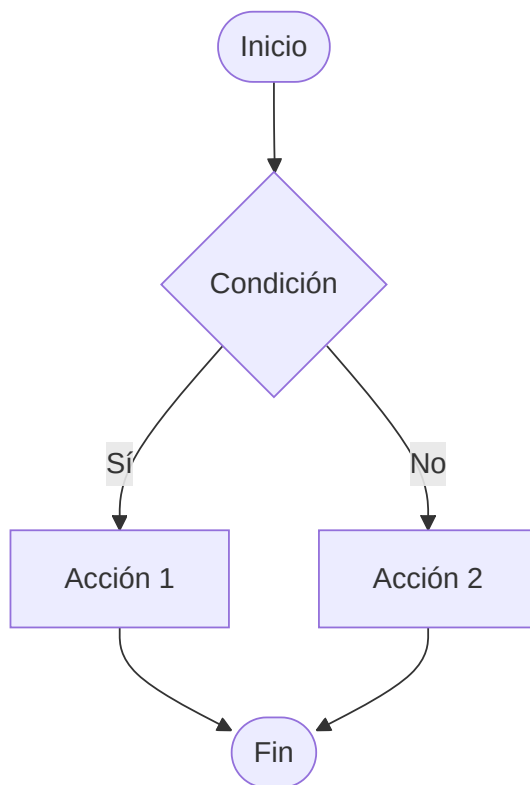
- **Flechas:** Indican la dirección del flujo del algoritmo, conectando los diferentes símbolos entre sí.
- **Inicio/Fin:** Representado por un óvalo, indica el comienzo y el final del algoritmo.



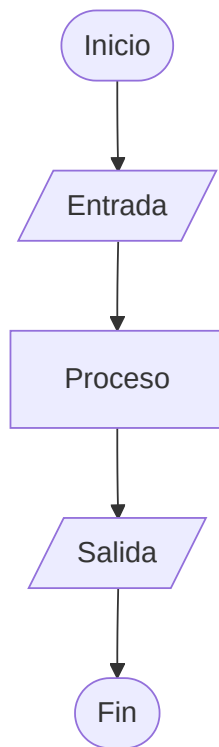
- **Proceso:** Representado por un rectángulo, indica una acción o un conjunto de acciones que se deben realizar.



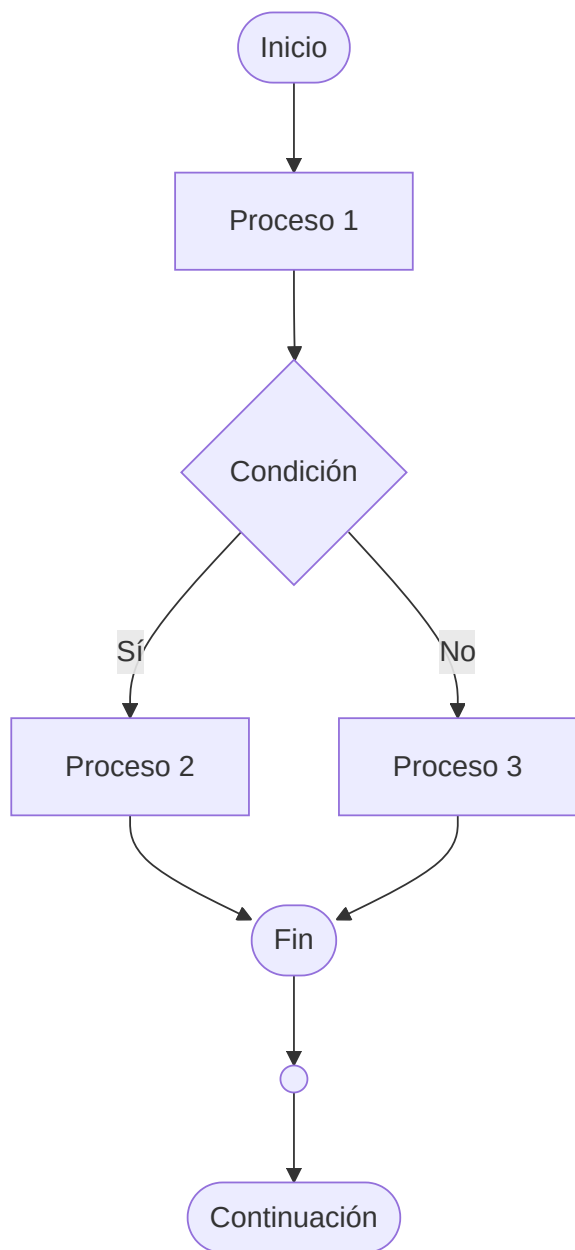
- **Decisión:** Representado por un rombo, indica una pregunta o condición que debe evaluarse. Tiene dos salidas: una para el caso verdadero y otra para el caso falso.



- **Entrada/Salida:** Representado por un paralelogramo, indica una operación de entrada o salida de datos. Por ejemplo, leer un dato del usuario o mostrar un resultado en pantalla.



- **Conector:** Representado por un círculo, se utiliza para conectar diferentes partes del diagrama de flujo, especialmente cuando el diagrama es grande y se necesita dividir en varias páginas.

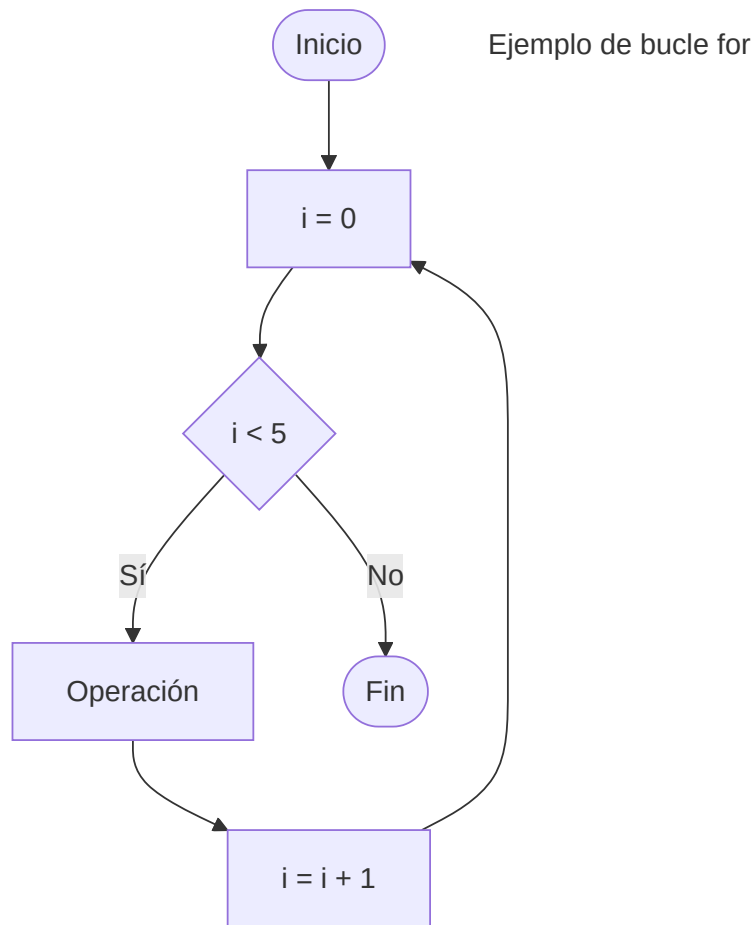


Estructuras de repetición

Las estructuras de repetición o bucles se utilizan para ejecutar un bloque de código varias veces mientras se cumpla una condición. En los diagramas de flujo, en algunos casos se representa mediante un hexágono, describiendo las condiciones de entrada y salida del bucle dentro de él:

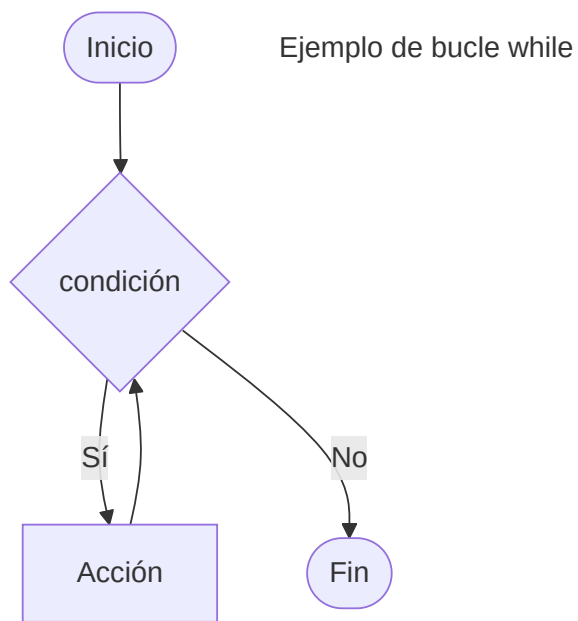
Sin embargo, es más común utilizar los símbolos de decisión, proceso, y flechas con realimentación para representar bucles.

- Bucle `for`: se utiliza para iterar un número determinado de veces. En el diagrama, se muestra una condición de inicio y una condición de finalización.



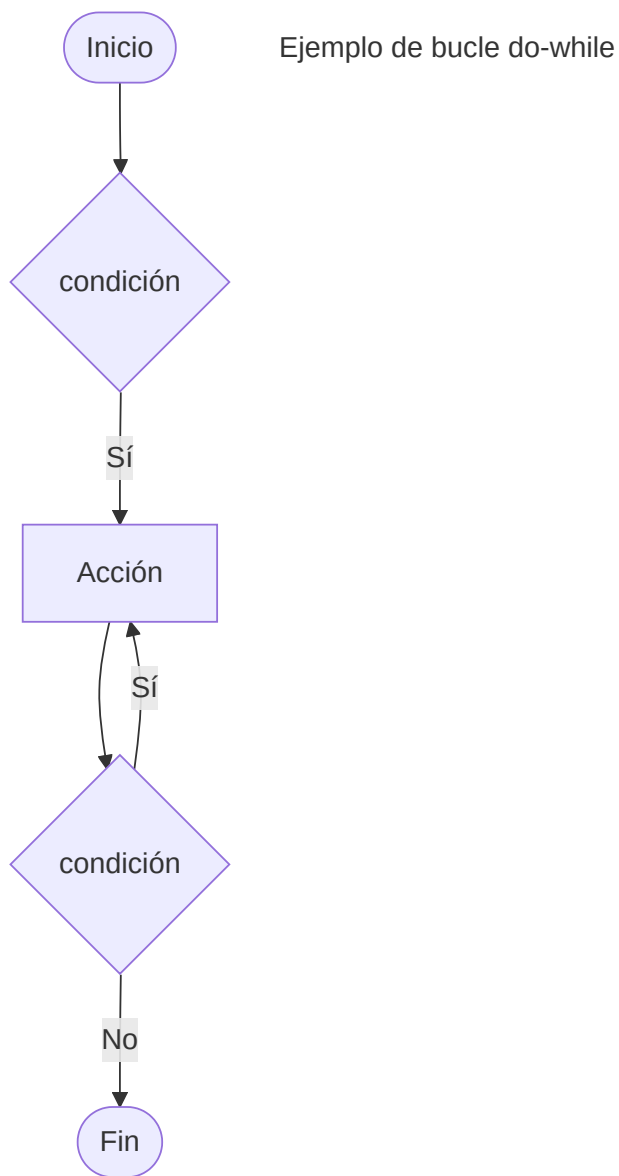
En este ejemplo, el bucle se ejecuta mientras la condición `i < 5` sea verdadera, incrementando `i` en cada iteración.

- Bucle `while`: se utiliza para repetir un bloque de código mientras una condición sea verdadera. En el diagrama, la condición se evalúa antes de ejecutar el bloque de código.



En este caso, el bucle se ejecuta mientras la condición sea verdadera, y se detiene cuando la condición es falsa. La condición se evalúa **antes** de realizar la acción.

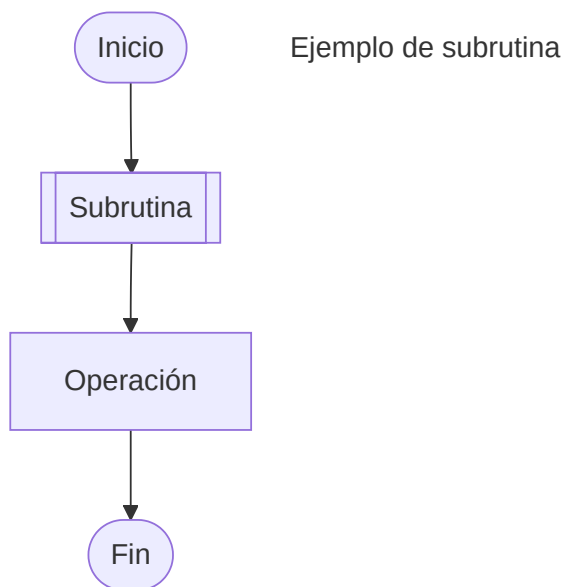
- Bucle `do-while`: similar al bucle `while`, pero la condición se evalúa **después** de ejecutar el bloque de código, garantizando que el bloque se ejecute al menos una vez.



En este caso, el bucle se ejecuta al menos una vez, ya que la condición se evalúa **después** de realizar la acción. Se repite mientras la condición sea verdadera.

Subrutinas

Las Subrutinas o subprogramas son un conjunto de instrucciones, identificadas con un nombre, que se agrupan para realizar una tarea específica y que pueden ser invocadas desde diferentes partes del algoritmo. En los diagramas de flujo, se representan mediante un rectángulo con bordes redondeados.



Pseudocódigo

El pseudocódigo es otra herramienta que nos permite describir un algoritmo de forma más abstracta y cercana al lenguaje natural, sin preocuparnos por la sintaxis específica de un lenguaje de programación. Es una forma de expresar la lógica del algoritmo de manera clara y concisa, utilizando una notación que es fácil de entender.

El pseudocódigo no tiene una sintaxis estricta, pero sigue ciertas convenciones para facilitar su comprensión. Algunas de las convenciones más comunes son:

- Utilizar palabras clave en mayúsculas para las estructuras de control (por ejemplo, `IF`, `WHILE`, `FOR`).
- Utilizar indentación para mostrar la jerarquía y el flujo del algoritmo.
- Utilizar comentarios para explicar partes del pseudocódigo.
- Utilizar nombres descriptivos para las variables y funciones.

En este contexto, lenguajes como PSeInt ayudan a los estudiantes que comienzan a programar a familiarizarse con la lógica de programación sin preocuparse por la sintaxis de un lenguaje específico. PSeInt es un entorno de desarrollo que permite escribir pseudocódigo y ejecutarlo, facilitando el aprendizaje de conceptos básicos de programación.

PSeInt como lenguaje de pseudocódigo

PSeInt es una herramienta para asistir a un estudiante en sus primeros pasos en programación. Mediante un simple e intuitivo pseudolenguaje en español (complementado con un editor de diagramas de flujo), le permite centrar su atención en los conceptos fundamentales de la algoritmia computacional, minimizando las dificultades propias de un lenguaje y proporcionando un entorno de trabajo con numerosas ayudas y recursos didácticos.

Se recomienda la realización de [este tutorial](#) que ayudará a familiarizarse con el uso de PSeInt.

Otros diagramas

Existen otros tipos de diagramas que pueden ser útiles para describir algoritmos en contextos específicos. Algunos de ellos son:

- **Diagramas Nassi-Schneiderman (N-S):** son diagramas de flujo estructurados que representan la lógica de un algoritmo de manera jerárquica. Utilizan bloques anidados para mostrar las estructuras de control y son especialmente útiles para representar algoritmos complejos.
- **Tablas de decisión:** son tablas que representan las decisiones que se deben tomar en un algoritmo. Cada fila de la tabla representa una combinación de condiciones y acciones, lo que permite visualizar de manera clara las decisiones que se deben tomar en cada caso.