CURSO DE LÓGICA DE PROGRAMACIÓN

INTRODUCCIÓN

Hoy en día, **la programación constituye una competencia fundamental en la sociedad digital contemporánea**. Su influencia se extiende mucho más allá del campo de la informática: está presente en disciplinas como la biología (procesamiento de datos genómicos), la economía (modelos predictivos), la ingeniería (automatización de procesos industriales), las humanidades digitales (análisis de corpus textuales), e incluso en el arte y la música (creación generativa).

Cuando utilizamos aplicaciones como **Instagram, TikTok o Spotify**, interactuamos con **sistemas programados** que gestionan cantidades masivas de información en fracciones de segundo. Los **vehículos autónomos** procesan datos de sensores en tiempo real gracias a algoritmos complejos, mientras que los **electrodomésticos inteligentes** emplean programación embebida para optimizar consumos energéticos y asistir en las tareas cotidianas, como avisar cuándo un alimento está próximo a caducar.

Sin embargo, **saber un lenguaje de programación (Python, Java, C++, etc.) no basta**. Lo esencial es **aprender a pensar como un programador**, es decir, a estructurar problemas en pasos lógicos, claros y ordenados. Este enfoque se denomina **pensamiento algorítmico**, y constituye la base de toda solución computacional.

Objetivos del curso

Este curso tiene como propósito desarrollar en el estudiante la capacidad de análisis, abstracción y estructuración lógica de problemas para su resolución computacional. Más allá de la sintaxis de un lenguaje concreto, lo que se busca es:

1. Comprender cómo **traducir un problema real a un modelo formal**.
2. Desarrollar la habilidad de **diseñar algoritmos correctos y eficientes**.
3. Reconocer la importancia de la **optimización temporal y espacial** en los programas.
4. Fortalecer la capacidad de **razonamiento lógico**, aplicable tanto en informática como en cualquier disciplina científica.

Estructura y módulos del curso

El curso está organizado en **6 módulos progresivos**, concebidos bajo una lógica universitaria que combina teoría, práctica y pensamiento crítico.

Módulo 1: Fundamentos del pensamiento computacional

- Historia y evolución de la programación.
- Concepto de algoritmo y sus propiedades (finitud, claridad, efectividad).
- Tipos de problemas computables.

- Introducción a la abstracción: dividir un problema en subproblemas.

Módulo 2: Representación de algoritmos

- Diagramas de flujo y pseudocódigo.
- Estándares internacionales de representación gráfica.
- Ventajas y limitaciones de cada método.
- Ejercicios prácticos: diseñar algoritmos sin escribir código.

Módulo 3: Estructuras de control

- Secuencias, decisiones (condicionales) e iteraciones (bucles).
- Resolución de problemas mediante combinaciones de estructuras de control.
- Análisis de casos de estudio (ejemplo: búsqueda y ordenamiento simple).

Módulo 4: Estructuración modular

- Subprogramas: funciones y procedimientos.
- Ventajas de la modularidad: legibilidad, reutilización y depuración.
- Principios de la programación estructurada.
- Ejercicios: descomposición modular de problemas complejos.

Módulo 5: Complejidad algorítmica

- Introducción al análisis temporal y espacial.
- Conceptos básicos de notación Big-O.
- Comparación entre soluciones ingenuas y soluciones optimizadas.
- Aplicación en problemas reales: búsquedas, ordenamientos y recorridos de estructuras.

Módulo 6: Resolución de problemas integrados

- Proyecto final: plantear un problema real, diseñar su algoritmo, representarlo y analizar su eficiencia.
- Opcional: implementación en Python con acceso al repositorio oficial del curso en GitHub.
- Discusión crítica sobre las soluciones de los estudiantes.

Metodología

El curso se desarrollará con un enfoque **teórico-práctico**:

- **Clases magistrales**: exposición de los conceptos teóricos con ejemplos interdisciplinares.
- **Ejercicios guiados**: aplicación paso a paso para consolidar la comprensión.
- **Retos individuales y grupales**: fomentar el trabajo colaborativo y la creatividad.
- **Prácticas opcionales en Python**: implementación de los algoritmos para estudiantes que deseen experimentar con código.
- **Repositorio GitHub**: con soluciones, recursos complementarios y ejercicios de refuerzo.

Filosofía del aprendizaje

La programación, al igual que un idioma o un deporte, requiere **práctica constante**. La inactividad provoca el "óxido cognitivo" y la pérdida de fluidez en el razonamiento. Por ello:

- Cada módulo incluye **ejercicios de dificultad progresiva**.
- Los estudiantes serán animados a **reflexionar críticamente** sobre sus propios algoritmos: ¿son correctos?, ¿eficientes?, ¿claros para otra persona?
- Se fomentará una visión de la programación como **herramienta transversal**, no como fin en sí mismo.

Resultados esperados

Al finalizar el curso, el estudiante será capaz de:

- Formular problemas de manera lógica y estructurada.
- Representar soluciones mediante diagramas de flujo y pseudocódigo.
- Diseñar algoritmos correctos, eficientes y modulares.
- Aplicar nociones básicas de complejidad algorítmica en sus soluciones.
- Trasladar estas competencias a cualquier lenguaje de programación que decida aprender posteriormente.

MÓDULO 1: PENSAMIENTO ALGORÍTMICO

¿Qué es la lógica de programación?

En el ámbito de la informática, cuando hablamos de **lógica de programación** nos referimos a la capacidad de **estructurar instrucciones de forma coherente, clara y no ambigua** para que un ordenador pueda ejecutarlas.

Un ordenador, a diferencia de los seres humanos, carece de sentido común y de la capacidad de inferencia por contexto. Mientras que una persona puede comprender la orden *"abre la puerta"* aunque no se le indique explícitamente que debe caminar hacia ella, girar el pomo y empujar, un ordenador necesita que cada acción esté **descompuesta en pasos concretos y precisos**.

De ahí que la lógica de programación pueda entenderse como un **lenguaje puente entre el razonamiento humano y la interpretación estricta de la máquina**.

Un ejemplo simple ilustra esta necesidad: si quisiéramos determinar si un número es **par o impar**, sería absurdo intentar evaluarlo sin antes conocer cuál es el número. La secuencia lógica que seguiría un ordenador sería:

1. Solicitar al usuario un número.
2. Calcular el resto al dividirlo entre 2.
3. Si el resto es 0 → el número es par.
4. Si el resto es distinto de 0 → el número es impar.

Este proceso trivial revela una verdad profunda: **la programación exige un orden lógico impecable**.

```python
valor = int(input("Ingrese un valor: "))
if valor % 2 == 0:
    print(f"El valor {valor} es par")
else:
    print(f"El valor {valor} es impar")
```

Concepto de algoritmo

Un **algoritmo** es un conjunto finito de instrucciones ordenadas que permiten resolver un problema o realizar una tarea. Aunque el término suene abstracto, los algoritmos forman parte de nuestra vida diaria. Por ejemplo:

- **En el supermercado**: si no encontramos el pan habitual, debemos decidir entre las opciones disponibles. El proceso de elección (comparar precio, saludabilidad y decidir) constituye un algoritmo.
- **Al atarse los zapatos**: si son de cordones, aplicamos una técnica; si son de velcro, otra distinta. El "algoritmo" depende de las condiciones iniciales.

En programación, los algoritmos son el **esqueleto lógico** que da sentido a cualquier programa. Un ordenador no "entiende" el objetivo global, pero sí puede ejecutar secuencias bien definidas.

Un ejemplo clásico es la **sucesión de Fibonacci**, donde cada término se define como la suma de los dos anteriores. Un algoritmo sencillo para obtener el enésimo término de la sucesión sería:

1. Definir los dos primeros valores: 0 y 1.
2. Iterar desde el tercer término hasta el n-ésimo.
3. En cada paso, calcular la suma de los dos anteriores.
4. Devolver el resultado.

Aunque pueda implementarse en distintos lenguajes, el núcleo lógico es siempre el mismo.

```python
def fibonacci(n):
    a, b = 0, 1
    for _ in range(n):
        a, b = b, a + b
    return a

num = int(input("Ingrese un número: "))
print(f"El número Fibonacci de {num} es: {fibonacci(num)}")
#El 0 está en la posición 0, el 1 en la posición 1, etc.
```

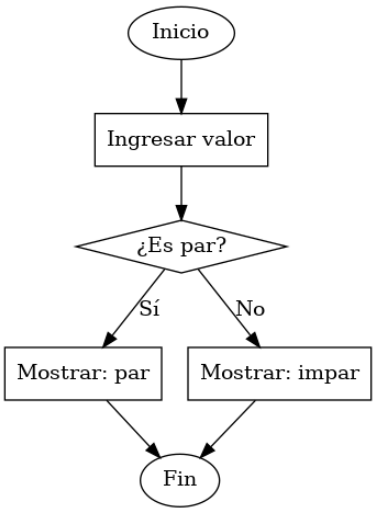
En el ejemplo anterior, se ha definido la secuencia de Fibonacci por separado del resto del código. No te preocupes por entenderlo a fondo ahora: más adelante veremos cómo empaquetar estas instrucciones en algo llamado función, que nos permitirá reutilizar el mismo algoritmo sin tener que escribirlo cada vez. En este ejemplo de Fibonacci hemos escrito las instrucciones paso a paso, pero pronto aprenderás una forma más elegante y reutilizable de hacerlo.

Introducción a diagramas de flujo

Antes de escribir código, resulta altamente recomendable **visualizar el algoritmo mediante un diagrama de flujo**. Esta herramienta permite:

- **Anticipar problemas lógicos**: al representar gráficamente las decisiones y bifurcaciones, los errores se detectan antes de programar.
- **Facilitar la comunicación**: es más sencillo explicar un diagrama que leer directamente líneas de código.
- **Reducir ambigüedad**: cada símbolo representa una acción concreta (inicio/fin, proceso, entrada/salida, decisión).

Un diagrama de flujo del problema "número par o impar" tendría los siguientes pasos:



La fuerza de este método radica en su **carácter visual** y en que obliga al programador a **estructurar su razonamiento paso a paso**.

Ejercicios: Diseñar algoritmos para calcular el factorial

El **factorial** de un número entero positivo *n* (denotado como *n!*) se define como el producto de todos los enteros desde 1 hasta *n*:

$$n! = 1 \times 2 \times 3 \times \ldots \times n$$

Por ejemplo:

$$5! = 1 \times 2 \times 3 \times 4 \times 5 = 120$$

Objetivo del ejercicio

Diseñar un **diagrama de flujo** que represente el cálculo del factorial de un número dado por el usuario.

Pasos lógicos del algoritmo

1. Inicio.

2. Solicitar al usuario el número *n*.
3. Inicializar el resultado en 1.
4. Iterar desde 1 hasta *n*.
5. Multiplicar el resultado por cada valor del bucle.
6. Mostrar el resultado.
7. Fin.

Este ejemplo demuestra cómo un problema aparentemente complejo puede resolverse al **descomponerlo en pasos simples**.

Reto adicional
Implementar el algoritmo en código (por ejemplo, en Python). Esto permite comprobar la **correspondencia entre la representación gráfica y la implementación real**.

Reflexión final del módulo

Este primer módulo busca asentar la idea de que **programar no es aprender un lenguaje, sino aprender a pensar en términos algorítmicos**.

- Un buen programador no es aquel que memoriza la sintaxis de Python, C o Java, sino quien es capaz de **traducir un problema a una secuencia lógica y clara**.
- El aprendizaje de la lógica algorítmica es transversal: sirve para matemáticas, ciencias, economía, filosofía e incluso en la vida cotidiana.
- Dominar la lógica garantiza que, independientemente del lenguaje o la tecnología que se use, el estudiante podrá **adaptarse, aprender y resolver problemas reales**.

MÓDULO 2: ESTRUCTURAS DE CONTROL

MÓDULO 3: PENSAMIENTO RECURSIVO Y MODULARIDAD

MÓDULO 4: ESTRUCTURAS DE DATOS LÓGICOS

MÓDULO 5: ALGORITMOS CLÁSICOS Y ANÁLISIS CLÁSICOS

MÓDULO 6: PENSAMIENTO COMPUTACIONAL Y PROBLEMAS ABIERTOS