Herramientas de IA para la programación

M1: Orientación al perfil y metodología del curso

AE4: Utilizar herramientas de inteligencia artificial para el aumento de la productividad en la programación según las buenas prácticas de la industria.

alkemy

Introducción 🧑‍💻

En la actualidad, la programación ya no se limita únicamente a escribir código de forma manual: las herramientas inteligentes han llegado para **complementar y potenciar el trabajo del desarrollador**. Este manual tiene como objetivo principal introducir al estudiante en dos dimensiones fundamentales: el diseño de algoritmos mediante **pseudocódigo y diagramas de flujo**, y el aprovechamiento de **herramientas de inteligencia artificial** —como ChatGPT— para mejorar la productividad durante el proceso de programación.

El enfoque se divide en dos grandes bloques. En la primera parte, se exploran los fundamentos del pensamiento algorítmico: qué es un algoritmo, cómo se estructura, y cómo se puede representar utilizando pseudocódigo y diagramas de flujo. Se trabajará con la herramienta **PSeInt** para practicar el diseño y la ejecución paso a paso de algoritmos básicos.

En la segunda parte, se abordarán conceptos clave sobre inteligencia artificial aplicada al desarrollo de software. Aprenderás cómo utilizar herramientas como ChatGPT para corregir, generar y optimizar código mediante prompts efectivos, y también se presentarán buenas prácticas para usar IA de manera responsable y eficiente.

Aprendizaje esperado

Cuando finalices la lección serás capaz de:

- Comprender qué es un algoritmo y aplicar pensamiento algorítmico para resolver problemas básicos.
- Utilizar pseudocódigo y diagramas de flujo para representar algoritmos de manera estructurada.
- Emplear la herramienta PSeInt para diseñar, ejecutar y visualizar algoritmos.
- Entender los conceptos fundamentales de la inteligencia artificial y su aplicación en el desarrollo de software.
- Usar herramientas de IA como ChatGPT para apoyar procesos de corrección, generación y optimización de código.

Algoritmos y pseudocódigo

¿Qué es un algoritmo?

Un **algoritmo** es un conjunto finito de **pasos ordenados y definidos** que permiten resolver un problema o realizar una tarea específica. Cada paso debe ser claro y preciso, sin ambigüedades, de manera que pueda ser ejecutado tanto por una persona como por una computadora.

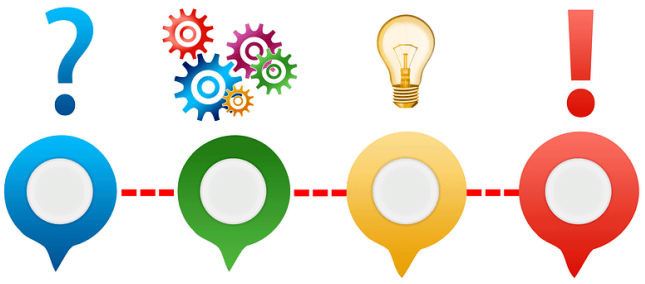
Un algoritmo debe tener las siguientes propiedades:

- → **Entrada:** uno o más valores de datos iniciales.
- → **Proceso:** instrucciones claras y secuenciales para manipular los datos.
- → **Salida:** uno o más resultados como solución al problema.
- → **Determinismo:** cada paso del algoritmo debe producir un resultado bien definido.
- → **Finitud:** el algoritmo debe finalizar en un número finito de pasos.

Ejemplo sencillo de algoritmo: Calcular el promedio de tres números.

1. Leer tres números (A, B, C)
2. Sumar A + B + C
3. Dividir el resultado entre 3
4. Mostrar el promedio

¿Qué es el pensamiento algorítmico?



El **pensamiento algorítmico** es una habilidad lógica que permite resolver problemas mediante la creación de algoritmos. Implica descomponer un problema complejo en partes más pequeñas y manejables, diseñar soluciones paso a paso y prever sus resultados antes de implementarlas.

Este tipo de pensamiento es fundamental no solo en programación, sino en muchas actividades cotidianas donde se toman decisiones secuenciales o se resuelven problemas mediante pasos estructurados.

Características del pensamiento algorítmico

- ★ **Secuencialidad:** comprender la importancia del orden de ejecución.
- ★ **Repetición:** detectar patrones que pueden automatizarse con bucles.
- ★ **Condicionalidad:** aplicar decisiones según condiciones específicas.
- ★ **Abstracción:** ignorar los detalles irrelevantes para centrarse en la lógica del problema.
- ★ **Modularidad:** dividir problemas grandes en subproblemas.

Ejemplo aplicado

Imagina que necesitas hacer una receta de cocina. Para hacer una torta, no empezamos batiendo los huevos antes de romperlos: seguís un orden. Ese mismo enfoque paso a paso es lo que ocurre en el diseño de algoritmos.

- **Problema**: Determinar si una persona es mayor de edad.
    - Algoritmo:
        - ↪ Leer edad.
        - ↪ Si tiene edad ≥ 18, entonces mostrará "Mayor de edad".
        - ↪ Sino, mostrar "Menor de edad".

Importancia para la programación

En la programación, el pensamiento algorítmico permite:

- ✓ Escribir código limpio y funcional.
- ✓ Anticipar errores o excepciones.
- ✓ Aumentar la eficiencia y legibilidad del software.
- ✓ Reutilizar soluciones en diferentes contextos (patrones).

Componentes de un algoritmo: *Variables y estructuras de control*

Componentes básicos de un algoritmo

Para que un algoritmo sea funcional, necesita ciertos elementos estructurales fundamentales. Los principales son:

→ **Datos de entrada:** información que el algoritmo necesita para funcionar.
→ **Procesamiento:** conjunto de pasos, reglas o instrucciones lógicas.
→ **Datos de salida:** resultados generados por el procesamiento.

Pero en la práctica, todo esto se logra mediante herramientas internas del algoritmo como **variables** y **estructuras de control**, que explicaremos a continuación.

📝 Variables

Una **variable** es un **espacio de almacenamiento con nombre** que contiene un valor que puede cambiar durante la ejecución del algoritmo. Es como una "caja" que guarda información.

- **Ejemplo en pseudocódigo:**

```
Leer A
Leer B
Suma ← A + B
Mostrar Suma
```

En este ejemplo **A, B** y **Suma** son variables.

Las variables pueden ser de distintos tipos (según el lenguaje que uses):

- Números enteros (int)
- Decimales (float)
- Cadenas de texto (string)
- Booleanos (true o false)

🔁 Estructuras de control

Las estructuras de control son instrucciones que alteran el flujo normal de ejecución de un algoritmo. Hay tres tipos principales:

1. Secuencia
   - ↪ La más simple: las instrucciones se ejecutan en el orden en que aparecen.

```
Leer nombre
Mostrar "Hola " + nombre
```

2. Condicionales (decisiones)
   - ↪ Permiten ejecutar diferentes acciones según una condición.

```
Si edad >= 18 Entonces
    Mostrar "Mayor de edad"
Sino
    Mostrar "Menor de edad"
```

3. Repetitivas (bucles)
   - ↪ Permiten repetir un bloque de instrucciones varias veces.

```
Para i Desde 1 Hasta 5 Hacer
    Mostrar "Hola mundo"
Fin Para
```

   También se puede usar `Mientras` o `Repetir-Hasta`.

Ejemplo completo en pseudocódigo

```
Leer nota
Si nota >= 6 Entonces
    Mostrar "Aprobado"
Sino
    Mostrar "Desaprobado"
```

Aquí usamos:

→ **Una variable**: `nota`
→ **Una estructura condicional**: `Si...Entonces...Sino`

¿Por qué son importantes?

El dominio de las variables y estructuras de control te permite:

- ✓ Representar cualquier proceso lógico.
- ✓ Adaptar algoritmos a situaciones dinámicas.
- ✓ Generar soluciones eficientes y reutilizables.
- ✓ Facilitar el paso del pseudocódigo a un lenguaje real de programación.

Representación de algoritmos

¿Qué es el pseudocódigo?

El **pseudocódigo** es una forma intermedia entre el lenguaje natural y la programación real. Se utiliza para describir de manera lógica los pasos de un algoritmo sin depender de la sintaxis de un lenguaje específico (como Java o Python).

- No se ejecuta en una computadora.
- Está pensado para ser fácilmente entendido por personas.
- Ayuda a diseñar y planificar programas antes de codificarlos.

Propósito del pseudocódigo

- ✓ Representar ideas de forma estructurada y comprensible.
- ✓ Describir la lógica de un problema sin errores de sintaxis.
- ✓ Ser usado como puente entre el pensamiento algorítmico y el código real.
- ✓ Facilitar el trabajo colaborativo entre programadores, docentes o analistas.

Estructura básica del pseudocódigo

Algunas convenciones comunes en pseudocódigo incluyen:

- → **Inicio** y **Fin** para delimitar el algoritmo.
- → Uso de palabras clave como Leer, Mostrar, Si, Entonces, Mientras, Para, etc.
- → Asignaciones con ← (flecha) o =.
- → Sangrías para reflejar bloques de código.

Ejemplo:

```
Inicio
    Leer edad
    Si edad >= 18 Entonces
        Mostrar "Es mayor de edad"
    Sino
        Mostrar "Es menor de edad"
Fin
```

Reglas generales del pseudocódigo

- Ser claro y preciso.
- Evitar ambigüedades.
- Usar nombres de variables descriptivos.
- Mantener coherencia en la estructura.

Ventajas del pseudocódigo

★ **Facilita la comprensión:** Es más fácil de leer que un lenguaje de programación.
★ **Lenguaje neutral:** No depende de ningún software o lenguaje.
★ **Ideal para enseñanza:** Útil para introducir conceptos sin preocuparse por errores de compilación.
★ **Herramienta de planificación:** Ayuda a validar la lógica de un algoritmo antes de implementarlo.

Ejemplo de pseudocódigo completo

↪ **Problema:** Leer dos números y mostrar el mayor.

```
Inicio
    Leer A
    Leer B
    Si A > B Entonces
        Mostrar "A es mayor"
    Sino
        Mostrar "B es mayor o igual"
Fin
```

Dominar el pseudocódigo es clave para organizar tus ideas antes de codificar. No importa si usas Java, Python o C++: si tu lógica está bien estructurada en pseudocódigo, convertirla en un programa funcional será mucho más fácil.

Utilización de PSeInt para la elaboración de algoritmos

¿Qué es PSeInt?

PSeInt (PSeudo Intérprete) es una herramienta educativa diseñada para ayudar a los estudiantes a desarrollar la lógica algorítmica utilizando pseudocódigo estructurado. Su interfaz amigable y sus funcionalidades permiten escribir, ejecutar y depurar algoritmos sin necesidad de aprender un lenguaje de programación formal.

Principales características de PSeInt

- ★ Permite escribir pseudocódigo en español o inglés.
- ★ Ejecuta el algoritmo paso a paso.
- ★ Muestra el estado de las variables en cada paso.
- ★ Incluye editor de diagramas de flujo.
- ★ Ofrece plantillas, ayudas y sugerencias.
- ★ Tiene un entorno visual simple e intuitivo.

📥 Instalación de PSeInt

1. Ir al sitio oficial: https://pseint.sourceforge.net
2. Descargar el instalador correspondiente a tu sistema operativo.
3. Seguir los pasos del asistente de instalación.
4. Abrir la aplicación y comenzar un nuevo algoritmo.

¿Cómo crear un algoritmo en PSeInt?

1. **Nuevo archivo:** Desde el menú, seleccionar "Nuevo".
2. **Escribir pseudocódigo:** Ingresar instrucciones como Leer, Escribir, Si, Mientras, etc.
3. **Guardar:** Guardar con extensión .psc.
4. **Ejecutar:** Hacer clic en "Ejecutar" o "Paso a paso"

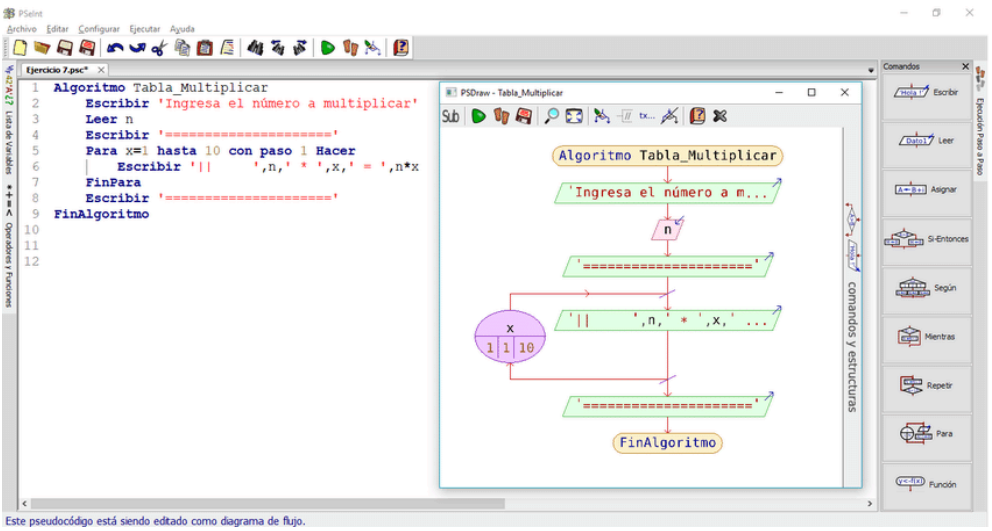
Ejemplo:

```
Proceso MayorDeDosNumeros
    Definir A, B Como Entero
    Escribir "Ingrese el primer número:"
    Leer A
    Escribir "Ingrese el segundo número:"
    Leer B
    Si A > B Entonces
        Escribir "El mayor es ", A
    Sino
        Escribir "El mayor es ", B
    FinSi
FinProceso
```

PSeInt permite ejecutar el algoritmo paso a paso. Esta funcionalidad es clave para:

- → **Visualizar la lógica en acción.**
- → **Observar el valor de cada variable a lo largo del algoritmo.**
- → **Detectar errores lógicos.**

🎢 Diagramas de flujo en PSeInt



Fuente: Researchgate

Además del pseudocódigo, PSeInt permite visualizar y editar el algoritmo como un **diagrama de flujo** automáticamente, lo que ayuda a reforzar la comprensión visual del flujo de control. Acá te compartimos una muestra en Youtube.

Ejecución de un algoritmo: *paso a paso y vista de las variables*

La **ejecución paso a paso** de un algoritmo es una de las estrategias más efectivas para comprender el flujo lógico de un programa y detectar errores. En el contexto educativo, permite observar el comportamiento del algoritmo en tiempo real y analizar cómo cambian las variables en cada instrucción.

¿Qué es la ejecución paso a paso?

Se trata de **simular el comportamiento del algoritmo línea por línea**, mostrando cómo se evalúan las condiciones, cómo se ejecutan los bucles y cómo se modifican las variables. Este enfoque permite identificar:

→ Errores de lógica (por ejemplo, una condición mal escrita).
→ Errores de inicialización o actualización de variables.
→ El orden exacto de ejecución de instrucciones.

¿Cómo se hace en PSeInt?

1. Escribe tu algoritmo normalmente.
2. Haz clic en el botón **"Paso a paso"** del menú de ejecución.
3. Aparecerá una ventana que muestra:
    - La instrucción actual a ejecutar.
    - El estado de las variables (valores actuales).
    - La consola de salida (mensajes del programa).

Esto te permite seguir el flujo de ejecución, verificar cada entrada/salida y analizar cómo evoluciona la información internamente.

Ejemplo práctico

Supongamos el siguiente pseudocódigo:

```
Proceso SumaAcumulada
    Definir i, suma Como Entero
    suma <- 0
    Para i <- 1 Hasta 5 Con Paso 1 Hacer
        suma <- suma + i
        Escribir "i=", i, " suma=", suma
    FinPara
FinProceso
```

Instrucción actual	Valor de i	Valor de suma
i <- 1	1	0
suma <- suma + i	1	1
i <- 2	2	1
suma <- suma + i	2	3
...	...	...

¿Por qué es tan útil esta función?

- Permite **desarrollar intuición algorítmica**: el estudiante comprende el "por qué" detrás de cada instrucción.
- Ayuda a detectar **errores invisibles** como bucles infinitos o condiciones mal estructuradas.
- Mejora la habilidad de **traducción entre pseudocódigo y código real**.

📌Recomendaciones

➔ Usa esta función especialmente cuando estés aprendiendo nuevas estructuras como `Si`, `Mientras`, `Repetir`, `Para`, etc.

➔ Crea algoritmos cortos para probar ideas específicas y observar cómo interactúan las variables.

Actividad recomendada

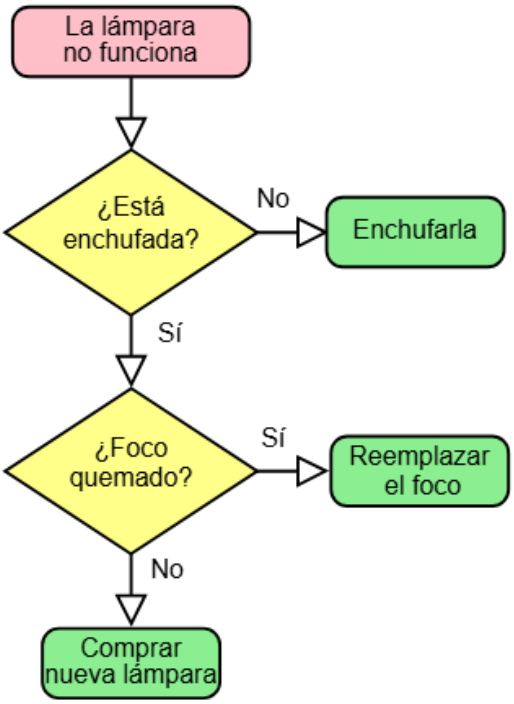
Implementa un algoritmo que calcule el **factorial de un número** y ejecútalo paso a paso en PSeInt. Observa cómo cambia la variable acumuladora en cada iteración.

Diagramas de flujo para representar un algoritmo

Los **diagramas de flujo** son representaciones gráficas de un algoritmo que muestran de manera visual y ordenada las operaciones, decisiones y caminos que sigue un programa. Son ampliamente usados en programación, ingeniería de software y procesos empresariales por su claridad y capacidad de comunicar ideas complejas de forma sencilla.

¿Qué es un diagrama de flujo?

Es un esquema visual que describe la **secuencia lógica de pasos** necesarios para resolver un problema. Cada paso se representa con un **símbolo estandarizado** (como óvalos, rectángulos o rombos), unidos entre sí por flechas que indican el flujo del control.



Principales símbolos

Símbolo	Significado
⬭ Ovalo	Inicio o fin del algoritmo
▭ Rectángulo	Acción o proceso (asignaciones, cálculos)
🔷 Rombo	Decisión (condición lógica que genera dos caminos)
➡ Flechas	Dirección del flujo
⧉ Paralelogramo	Entrada/salida (leer/escribir datos)

Ejemplo práctico:

Para el siguiente algoritmo:

```
Proceso ParOImpar
    Definir num Como Entero
    Escribir "Ingrese un número:"
    Leer num
    Si num MOD 2 = 0 Entonces
        Escribir "Es par"
    Sino
        Escribir "Es impar"
    FinSi
FinProceso
```

El diagrama de flujo tendría estos pasos:

1. Inicio →
2. Entrada de num →
   Decisión: num % 2 = 0?
   - **Si**: Mostrar "Es par"
   - **No**: Mostrar "Es impar" →
3. Fin

Herramientas para crear diagramas de flujo

→ **draw.io** (diagrams.net) – gratuita, online, fácil de usar.

→ **Lucidchart** – colaborativa y con plantillas.

→ **Pencil Project o yEd** – herramientas de escritorio.

→ **Miro** → lienzo en el que se puede diagramar diversa información.

→ **El propio PSeInt** permite ver el diagrama de flujo del algoritmo escrito.

El uso de algoritmos y flujogramas resulta clave porque fomenta el pensamiento lógico y secuencial, facilitando la comprensión estructurada de los procesos. Además, permiten documentar de forma clara los pasos de un algoritmo, lo cual es útil tanto para quienes desarrollan como para otros programadores que necesiten entender o reutilizar ese código.

También son una herramienta eficaz para identificar errores o caminos sin salida en procesos complejos, optimizando así el diseño de soluciones. Por último, resultan especialmente valiosos en contextos educativos, ya que permiten enseñar lógica algorítmica incluso a personas sin formación técnica previa.

Herramientas de IA para la productividad en la programación

Introducción al uso de IA en programación

El desarrollo de software ha evolucionado más allá del código escrito manualmente. Las herramientas de Inteligencia Artificial están revolucionando la productividad del programador al asistir en la escritura, corrección, generación, documentación y optimización del código. Estas herramientas no solo reducen el tiempo de desarrollo, sino que también aumentan la calidad del software entregado.

Con el auge de modelos de lenguaje como **ChatGPT** o asistentes como **GitHub Copilot**, el enfoque del programador moderno pasa a ser más estratégico: se trata de saber **qué pedir**, **cómo evaluar** las respuestas de la IA, y **cómo integrarlas** con criterio en su flujo de trabajo.

La IA en el desarrollo de software se basa en modelos de lenguaje extenso (LLM), entrenados con grandes volúmenes de código. Estos modelos no "entienden" la lógica humana, sino que predicen la secuencia de texto más probable para completar o generar código en función de un *prompt* (instrucción).

Conceptos básicos de inteligencia artificial

- **Inteligencia Artificial (IA):** Es la capacidad de una máquina o sistema informático para imitar funciones cognitivas humanas, como aprender, razonar, resolver problemas y tomar decisiones.
- **Aprendizaje automático (Machine Learning)**: Es una rama de la IA que permite a los sistemas aprender y mejorar automáticamente a partir de la experiencia, sin ser programados explícitamente para cada tarea.

- **Redes neuronales artificiales:** Son modelos inspirados en el cerebro humano que se utilizan en machine learning para identificar patrones complejos en datos. Están formadas por capas de nodos (“neuronas”) interconectadas.
- **Aprendizaje profundo (Deep Learning):** Subcategoría del aprendizaje automático que utiliza redes neuronales profundas para analizar grandes volúmenes de datos. Se usa en tareas como reconocimiento de voz, imagen y lenguaje natural.
- **Procesamiento del lenguaje natural (NLP):** Es el campo que permite a las computadoras entender, interpretar y generar lenguaje humano. Se utiliza, por ejemplo, en asistentes virtuales o traductores automáticos.
- **Entrenamiento y datos:** Para que un modelo de IA funcione, necesita ser entrenado con grandes volúmenes de datos. Cuanto más representativos y variados sean esos datos, mejor será su desempeño.
- **Modelos predictivos:** Son sistemas entrenados con datos que permiten anticipar resultados o comportamientos futuros. Se usan en áreas como marketing, salud o finanzas.
- **IA generativa:** Es un tipo de IA capaz de crear contenido nuevo como texto, imágenes, código, audio o video, a partir de datos aprendidos. ChatGPT o DALL·E son ejemplos de IA generativa.
- **Sesgo algorítmico:** Ocurre cuando los datos o los algoritmos utilizados en un sistema de IA reflejan prejuicios humanos, afectando la equidad o precisión de los resultados. Es un concepto clave para entender el uso responsable de la IA.
- **Prompt:** Es la instrucción o mensaje que una persona le da a un modelo de IA para obtener una respuesta. En herramientas de IA generativa, como ChatGPT, los prompts pueden ser preguntas, tareas, pedidos de código, imágenes, entre otros. 👉 *La calidad del prompt influye directamente en la calidad de la respuesta.* Por eso, aprender a escribir buenos prompts —lo que se conoce como **prompt engineering**— se ha vuelto una habilidad clave.

Aplicaciones comunes de la IA en el desarrollo

A continuación, se detallan algunos casos concretos de aplicación de la IA en el desarrollo de software:

- ➔ **Asistencia en tiempo real**: mientras escribes código, la IA predice posibles líneas de continuación o funciones completas.
- ➔ **Autocompletado inteligente**: con sugerencias contextualizadas que entienden el código existente.
- ➔ **Explicación de código**: ideal para aprender lenguajes nuevos o comprender código legado.
- ➔ **Detección y corrección de errores**: análisis semántico y sintáctico del código con sugerencias claras.
- ➔ **Generación de código boilerplate**: por ejemplo, plantillas de clases o controladores en frameworks web.
- ➔ **Conversión entre lenguajes**: útil en migraciones o aprendizaje entre entornos.
- ➔ **Creación de pruebas unitarias**: generación automática de tests a partir del código fuente.

ChatGPT como asistente de programación

ChatGPT es uno de los asistentes conversacionales más potentes actualmente, con capacidad para interactuar con múltiples lenguajes de programación y ofrecer resultados útiles para tareas como:

- → Consultas de sintaxis.
- → Corrección de errores de compilación o lógica.
- → Sugerencias de estructuras algorítmicas.
- → Revisión de seguridad en código.
- → Generación de documentación técnica (README, javadoc, etc.).

Ejemplo:

- **Prompt:** “Explícame qué hace este fragmento de código en [lenguaje de programación] y cómo puedo optimizarlo.”
  - ↪ **Resultado:** Descripción detallada del propósito, análisis de eficiencia y sugerencias.

Casos reales en Python usando IA 🐍

Aunque el foco de este módulo es el pseudocódigo, es importante ver cómo la IA se integra en casos reales de Python.

1. **Caso:** *Estructuras de datos y funciones*

    - **Problema**: Dada una lista de productos (nombre, precio), generar una función que devuelva solo los productos cuyo precio sea mayor a un umbral.

    - **Prompt a la IA:**

    "Genera en Python una función `filtrar_productos_caros(productos, minimo)` donde `productos` es una lista de diccionarios con nombre y precio. Devuelve solo los productos cuyo precio sea mayor o igual a `minimo`. Usa nombres de variables descriptivos y sigue PEP 8. Al final, genera 3 pruebas unitarias con pytest."

    - **Qué se logra:**

        → Código con listas, diccionarios y funciones.
        → Mejores nombres y estilo correcto.
        → Pruebas mínimas generadas automáticamente.

2. **Caso 2:** *Llamada a una API simple*

    - **Problema**: Consumir una API pública (por ejemplo, de clima) y mostrar la temperatura actual de una ciudad.

    - **Prompt a la IA**:

    "Genera un script en Python que consulte una API pública de clima (por ejemplo, OpenWeatherMap) para una ciudad pasada por parámetro. El script debe manejar errores básicos (ciudad no encontrada, error de red) e imprimir la temperatura actual. Sigue PEP 8 y agrega comentarios breves. No incluyas la clave real de API, solo un placeholder."

    - **Qué se observa**:

        → Uso de biblioteca `requests` (u otra).
        → Manejo básico de errores.
        → Comentarios y estilo alineado a buenas prácticas.

Esto ayuda a entender cómo pedir a la IA soluciones más completas, y luego revisarlas críticamente.

Uso de prompts efectivos

Dominar el uso de prompts es clave. Un buen prompt es claro, completo y específico.

El *prompt* debe ser claro, conciso y especificar el formato de salida deseado.

Tarea	Prompt (Instrucción)
Generación de Pseudocódigo	"Genera el pseudocódigo para un algoritmo que calcule el factorial de un número entero positivo ingresado por el usuario, usando ciclos."
Generación de PSeInt	"Genera el código en formato PSeInt para un algoritmo que pida 10 números y muestre el mayor de ellos."
Generación de Código Real	"Escribe una función en Python que implemente la búsqueda binaria en una lista ordenada, y añade comentarios explicativos."

¿Qué hay que considerar al momento de diseñar un prompt? 🤔

1. **Definición del Objetivo:** Establece claramente lo que esperas lograr con el prompt. Define el problema, la tarea o la información que buscas obtener.
    - Ejemplo: “Este código lanza un error de tipo `TypeError`. ¿Puedes ayudarme a identificar el problema y corregirlo?” (y se adjunta el código correspondiente al lenguaje que se está usando)
2. **Contextualización:** Proporciona detalles relevantes que el modelo necesita para realizar la tarea.
    - Incluye:
        - Contexto del problema.
        - Herramientas o lenguajes específicos (por ejemplo, Python o Selenium).
        - Restricciones o limitaciones.
3. **Construcción del Prompt Inicial:** Escribe un prompt inicial que cubra lo básico. No te preocupes por hacerlo perfecto en el primer intento.
    - Ejemplo: *"Genera un script para probar el endpoint* `/users` *de una API REST."*

4. **Prueba del Prompt:** Ejecuta el prompt y revisa si la respuesta cumple tus expectativas.
    - Preguntas clave:
        - ¿El resultado es relevante?
        - ¿Está completo o falta información?
        - ¿Es ambiguo?
5. **Análisis del resultado:** Identifica problemas como:
    - Respuestas incompletas o incorrectas.
    - Falta de contexto en el resultado.
    - Generación de contenido irrelevante.
6. **Iteración y Refinamiento:** Mejora el prompt, añade más contexto, especifica el formato esperado o ajusta el lenguaje para que sea más claro.
    - Ejemplo refinado:
      "Genera un script en Python para probar el endpoint `/users` *de una API REST. Asegúrate de validar que el estado HTTP sea 200 y que los datos incluyan las claves* `name` *y* `email`*."*
7. **Incorporación de estrategias avanzadas:** Aplica técnicas como:
    - Definir roles: *"Actúa como un ingeniero de QA especializado en pruebas de APIs."*
    - Dar ejemplos específicos: *"Este es un ejemplo de respuesta deseada..."*
    - Establecer restricciones: *"Usa únicamente Python con la biblioteca* `requests`*."*
8. **Validación:** Si el modelo responde correctamente y consistentemente, el prompt está listo. Si no, vuelve al paso de refinamiento.
9. **Documentación:** Registra el prompt final, su propósito, y los ajustes realizados. Esto es clave para reutilización y aprendizaje.
10. **Implementación:** Usa el prompt en el flujo de trabajo real o automatización correspondiente.

Uso de Prompts para la Corrección de Errores en el Código

En lugar de solo preguntar "¿Qué está mal?", la **IA** es más útil si se le pide:

1. **El Código:** El fragmento defectuoso.

2. **El Resultado Esperado:** "Debería devolver 15, pero devuelve 0".
3. **El Error Actual:** El mensaje de error o el comportamiento incorrecto.

Uso de Prompts Avanzados Aplicados a Problemas Reales

Los *prompts* avanzados son detallados y guían a la **IA** con contexto y restricciones.

Tarea	Prompt Avanzado
Refactorización y Optimización	"Analiza el siguiente pseudocódigo. Identifica si hay ineficiencias en el uso de ciclos anidados y propón una versión optimizada que minimice la complejidad algorítmica a *O(n)*. [Incluir código a optimizar]"
Corrección de Errores	"Estoy intentando que este algoritmo en *PSeInt* me calcule la suma total, pero la variable 'Total' siempre devuelve 0. Revisa el código, corrige la inicialización de la variable y explícame el error conceptual. [Incluir código defectuoso]"
Explicación y Documentación	"Actúa como un profesor de programación. Explica en 300 palabras la diferencia entre un ciclo *Mientras* y un ciclo *Para*, usando el contexto del pseudocódigo que genera el número *Fibonacci*."

Algunos ejemplos prácticos:

- **Corrección de errores:**
  - ↪ *"Este código en Python lanza un IndexError. ¿Cómo lo corrijo?"*
  - ↪ *"Sugiere una versión más clara del mismo código (sin cambiar qué hace)."*
- **Optimización:**
  - ↪ *"¿Cómo puedo hacer que este algoritmo tenga menor complejidad?"*
  - ↪ *"¿Hay una versión más eficiente de este código para recorrer una lista?"*
- **Generación:**
  - ↪ *"Genera una clase [nombre de la clase] que represente una factura con métodos para calcular impuestos."*

- ↪ *"Dame un algoritmo en pseudocódigo para ordenar una lista usando el método de burbuja."*

- **Documentación:**
  - ↪ *Crea una descripción corta para el README de [proyecto/módulo].*

- **Traducción:**
  - ↪ *"Convierte este código de [Lenguaje 1] a [Lenguaje 2] manteniendo el mismo resultado."*

Refactorización avanzada con IA: patrones y complejidad ⚙️

La IA también puede ayudar a **mejorar** código existente, no solo a generarlo:

Tareas típicas de refactorización con IA:

- ↪ Renombrar variables y funciones para hacer el código más legible.
- ↪ Extraer funciones más pequeñas a partir de bloques grandes.
- ↪ Reemplazar bucles ineficientes por estructuras más adecuadas (por ejemplo, usar `set` o comprensiones de listas en Python).
- ↪ Sugerir patrones de diseño básicos (por ejemplo, funciones puras, separación de responsabilidades).

Ejemplo de prompt avanzado:

```
"Analiza este código en Python que calcula estadísticas sobre una lista de ventas. Identifica duplicación de lógica y sugiere una refactorización aplicando el principio DRY (Don't Repeat Yourself). Explica cómo cambió la complejidad del algoritmo si corresponde."
```

Recomendación:

- ↪ Siempre probar el código refactorizado con pruebas automatizadas (unit tests) para asegurar que no se cambió el comportamiento esperado.

Integración con buenas prácticas de la industria (PEP 8, testing) 🧪

La IA se puede usar para reforzar estándares profesionales, no solamente para "escribir algo que funcione".

Ejemplos de uso:

- **Estilo PEP 8 (Python)**
    - Prompt:
      “Reescribe esta función en Python para que cumpla con PEP 8: nombres de variables, sangría, longitud de línea y uso de docstring. Explica qué cambios hiciste.”
- **Testing**
    - Prompt:

      “Genera pruebas unitarias con pytest para estas funciones en Python. Cubre al menos un caso feliz, un caso límite y un caso de error esperado.”
- **Revisión de calidad**
    - Prompt:
      “Actúa como un revisor de código senior. Analiza este módulo en Python y señala problemas de legibilidad, acoplamiento, duplicación y posibles errores.”

Objetivo: que el código resultante no solo funcione, sino que sea **mantenible** y alineado con estándares de la industria.

Trabajo colaborativo con IA: pair programming + IA 🤝

La IA puede formar parte de un flujo similar al pair programming (programación en pareja):

Dinámica sugerida en clase o equipos:

- **Rol 1**: persona “driver” que escribe el pseudocódigo o código.
- **Rol 2**: persona “navigator” que revisa, propone mejoras y formula prompts para la IA.
- **IA**: tercera “voz” que sugiere alternativas, explica errores o genera ejemplos.

Ejemplo de actividad colaborativa:

1. El equipo recibe un problema algorítmico simple (por ejemplo, calcular el promedio de n números, o validar una contraseña con ciertas reglas).

2. En conjunto, diseñan el pseudocódigo.
3. La persona “navigator” formula un prompt a la IA para:
    - Generar el algoritmo en PSeInt.
    - Proponer una implementación en Python.
4. El equipo revisa el resultado, corrige y mejora tanto la solución como el prompt.

Esto simula un flujo profesional donde se combinan: lógica humana, colaboración en equipo y apoyo de IA.

Seguridad al usar IA en código 🛡️

Es fundamental utilizar estas herramientas con criterios de seguridad claros.

Riesgos frecuentes:

- **Dependencias y ejemplos inseguros**
    - La IA puede sugerir versiones desactualizadas de librerías, o patrones vulnerables (por ejemplo, manejo inseguro de contraseñas, SQL sin parámetros, etc.).
- **Exposición de datos sensibles**
    - Nunca se debe pegar en la IA: claves de API, credenciales, datos personales reales, código propietario confidencial.
- **Malas prácticas normalizadas**
    - Ejemplos sin manejo de errores, sin validación de entradas, sin logging, etc.

Buenas prácticas de seguridad:

- Revisar siempre las dependencias sugeridas (versiones, issues reportados).
- Validar el código con herramientas adicionales (linters, analizadores de seguridad).
- Usar datos ficticios o anonimizados en los prompts.
- Preguntar explícitamente a la IA sobre riesgos:

"¿Ves algún posible problema de seguridad o de validación de entradas en este código? ¿Cómo lo mejorarías?"

Recordatorio clave:

La responsabilidad final del código en producción es siempre humana, no de la herramienta de IA.

Actividad práctica guiada: *IA en PSeInt y Python*

Este ejercicio demuestra cómo usar la **IA** para obtener soluciones algorítmicas y luego trasladarlas a un entorno de ejecución real.

Problema: Desarrollar un algoritmo que determine si un número ingresado por el usuario es primo.

1. **Paso 1:** Generación del Pseudocódigo con IA

    - **Prompt a la IA:** `"Necesito el pseudocódigo para el algoritmo 'Es Primo'. Debe recibir un número entero positivo, verificar si tiene solo dos divisores (1 y sí mismo) y devolver un mensaje claro. Genera el código en formato compatible con **PSeInt**."`

    - **Resultado Esperado de la IA (Ejemplo de PSeInt):**

```
Algoritmo EsPrimo
  Definir num Como Entero
  Definir i, contador Como Entero
  contador <- 0

  Escribir "Ingrese un número entero positivo:"
  Leer num

  Si num <= 1 Entonces
    Escribir num, " no es primo."
  SiNo
    Para i <- 1 Hasta num Con Paso 1 Hacer
      Si num MOD i = 0 Entonces
        contador <- contador + 1
      FinSi
    FinPara

    Si contador = 2 Entonces
      Escribir num, " es un número primo."
    SiNo
      Escribir num, " no es un número primo."
    FinSi
  FinSi
FinAlgoritmo
```

1. **Paso 2:** Ejecución y Depuración en PSeInt
    - Copia el código generado por la IA en PSeInt.
    - Ejecuta el algoritmo **Paso a Paso** (función de depuración).
    - **Observa:** Detente en la línea contador <- contador + 1 y verifica que el contador solo llegue a 2 si el número es primo.
2. **Paso 3:** Traducción a un Lenguaje Real con IA
    - **Prompt a la IA:** `"Ahora, traduce el pseudocódigo anterior del algoritmo 'Es Primo' a una función simple en Python. La función debe aceptar un argumento y devolver True si es primo, False en caso contrario."`

```python
def es_primo(num):
    # La IA traduce la lógica a Python
    if num <= 1:
        return False

    # Solo necesitamos verificar divisores hasta la raíz cuadrada del número
    # (Optimización que la IA debería sugerir)
    for i in range(2, int(num**0.5) + 1):
        if num % i == 0:
            return False

    return True

# Pruebas
print(f"El 17 es primo: {es_primo(17)}") # Salida: True
print(f"El 15 es primo: {es_primo(15)}") # Salida: False
```

Este proceso demuestra cómo la IA sirve tanto para generar la lógica en pseudocódigo como para implementarla eficientemente en un lenguaje real.

Comparativa de herramientas de IA populares

Herramienta	Descripción	Nivel de integración	Lenguajes soportados
ChatGPT	IA conversacional generalista que entiende código	Web/app/API	Multilenguaje
GitHub Copilot	Asistente de codificación de GitHub	Integrado en IDEs (VSCode, JetBrains)	Principalmente Python, JavaScript, Java

Tabnine	Autocompletado potenciado por IA entrenado en tu código	IDEs	Multilenguaje

Actividad Colaborativa: Refinando Prompts

La calidad de la respuesta de la IA depende directamente de la calidad del *prompt*. Esta actividad busca mejorar el criterio profesional mediante la práctica en grupo.

Tarea: Refinar un *prompt* inicial hasta obtener un resultado óptimo.

Paso	Objetivo	Prompt Inicial (Malo)	Prompt Refinado (Bueno)
1. Definir Tarea	Generar una función para invertir una cadena de texto.	"Haz un código para invertir un texto."	**"Genera una función en Python que tome una cadena de texto y devuelva la misma cadena invertida. No uses funciones integradas de Python, hazlo con un ciclo for."**
2. Definir Formato	Obtener un diagrama de flujo para un proceso.	"Quiero el diagrama de flujo de un cajero automático."	**"Genera el código en DOT (Graphviz) para un diagrama de flujo que represente el proceso de ingreso de PIN y selección de retiro en un cajero automático."**
3. Depuración	Corregir un error de lógica.	"Mi código no funciona."	**"Este es mi pseudocódigo. [Insertar código]. El problema es que solo calcula bien el primer IVA. Necesito que lo calcule correctamente en un ciclo para 5 productos. ¿Cómo lo corrijo?"**

Dinámica Colaborativa:

- Los estudiantes deben trabajar en parejas para comparar y criticar sus *prompts*.
- El foco debe estar en la **especificación del lenguaje/formato** y la **inclusión de restricciones** (Ej. "no usar funciones pre-construidas" o "formato PSeInt").

- Comparar la respuesta de la IA al *prompt* malo vs. el *prompt* refinado para ilustrar la ganancia en productividad.

Consideraciones y cuidados al utilizar herramientas de inteligencia artificial

Buenas prácticas en el uso de IA

1. **Verificación constante del código generado:**
   Una de las prácticas fundamentales al trabajar con herramientas de IA generativa, como ChatGPT o Copilot, es revisar cuidadosamente cada fragmento de código generado. Si bien estas herramientas ofrecen respuestas rápidas y precisas en muchos casos, pueden producir errores sutiles, problemas de seguridad o soluciones ineficientes. La validación manual es indispensable para garantizar que el código cumpla con los requisitos funcionales y de calidad del proyecto.

2. **Evitar la dependencia excesiva:**
   El uso de herramientas de IA debe complementarse con el juicio y el conocimiento del programador. Es importante evitar que los desarrolladores se acostumbren a delegar completamente el diseño lógico y la resolución de problemas a la IA. En su lugar, se recomienda emplearla como una extensión de la capacidad humana, que acelera tareas repetitivas, sugiere soluciones o explica fragmentos de código, sin reemplazar el pensamiento crítico.

3. **Resguardo de la privacidad y datos sensibles:**
   Jamás se debe introducir en una herramienta de IA información confidencial, credenciales, claves de API o datos personales de usuarios. Muchas de estas herramientas utilizan la información ingresada para mejorar sus modelos, lo que podría derivar en filtraciones involuntarias. Es preferible trabajar con datos sintéticos o anónimos durante las interacciones con la IA.

4. **Actualización constante del conocimiento técnico:**
   El ecosistema tecnológico avanza rápidamente: nuevas versiones de lenguajes, frameworks o librerías pueden dejar obsoletos los ejemplos de código sugeridos por la IA. Por tanto, el desarrollador debe mantenerse actualizado, validando la vigencia del código y adaptándolo según las mejores prácticas actuales. Además, las propias herramientas de IA se

actualizan y mejoran continuamente, por lo que conocer sus nuevas capacidades es clave.

Consideraciones éticas

- ➔ **Citación y respeto por la autoría:** Si una herramienta de IA genera código basado en ejemplos populares o repositorios públicos, se debe procurar citar adecuadamente las fuentes si el código será reutilizado, especialmente en contextos académicos, proyectos de código abierto o comerciales. Esto refuerza una cultura de respeto hacia el trabajo de otros desarrolladores.
- ➔ **Evitar automatizaciones maliciosas o usos fraudulentos:** La IA puede facilitar tareas como el scraping masivo de datos, el envío automatizado de solicitudes, el desarrollo de bots que infringen normas o incluso la ingeniería inversa. Estos usos deben evitarse totalmente, tanto por razones legales como éticas. Un uso responsable implica apegarse a las políticas de uso aceptable y priorizar la construcción de soluciones seguras y legítimas.
- ➔ **Reconocer las limitaciones de la IA**: La IA no es infalible. Puede generar código ineficiente, vulnerable o simplemente incorrecto. Incluso cuando el resultado parece coherente o funcional, no debe considerarse una verdad absoluta. Es fundamental que el desarrollador mantenga una actitud crítica y use herramientas de testing, análisis estático y revisión de código como respaldo.

Beneficios concretos

- **Incremento de la productividad del desarrollador:** El uso inteligente de herramientas de IA permite acelerar significativamente el desarrollo de software. Desde la generación de funciones repetitivas, la documentación de APIs, hasta la automatización de pruebas unitarias, la IA libera tiempo que puede emplearse en tareas de mayor valor.
- **Mejora en la calidad del código:** Muchos modelos de IA están entrenados con buenas prácticas de programación, lo que puede ayudar a los desarrolladores a adoptar patrones correctos de diseño, nombres de variables descriptivos y estructuras lógicas limpias. Esto contribuye a escribir código más legible, mantenible y menos propenso a errores.
- **Aceleración del aprendizaje técnico:** Las herramientas de IA pueden actuar como tutores interactivos. Explican errores, sugieren soluciones alternativas y permiten a los principiantes comprender conceptos

complejos en un lenguaje natural. Así, los desarrolladores en formación pueden aprender mientras construyen y depuran código real.

- **Documentación automatizada:** La generación de documentación técnica es una tarea esencial pero a menudo relegada. Las IA pueden generar descripciones automáticas de funciones, estructuras de datos o flujos de trabajo, lo que garantiza mayor coherencia en los proyectos y facilita la incorporación de nuevos miembros al equipo.
- **Apoyo en debugging y testing:** Al identificar errores en el código, muchos asistentes de IA pueden sugerir posibles causas o soluciones. También pueden ayudar a construir pruebas unitarias o de integración a partir del código existente, lo que mejora la cobertura de pruebas y reduce el tiempo dedicado a tareas manuales.

Mala Práctica	Buena Práctica (Criterio Profesional)
Aceptar Ciegamente	**Validar y Verificar:** Siempre probar el código de la **IA**. La **IA** puede generar *errores sutiles* (alucinaciones).
Pedir la Solución Completa	**Descomponer el Problema:** Usar la **IA** para partes del código, manteniendo la lógica central en el pensamiento humano.
Revelar Datos Sensibles	**Privacidad:** Nunca ingresar código que contenga credenciales, claves **API** o información confidencial.
Dependencia Total	**Comprender antes de Copiar:** El objetivo es aumentar la productividad, no reemplazar el entendimiento. Usar la **IA** para entender, no solo para copiar la respuesta.

Cierre ✍️

Las herramientas de inteligencia artificial están transformando la forma en que programamos, aprendemos y resolvemos problemas. A lo largo de este manual, exploramos desde los fundamentos del pensamiento algorítmico y el uso de pseudocódigo, hasta la implementación práctica con herramientas como PSeInt y el aprovechamiento de asistentes de IA como ChatGPT para optimizar el desarrollo.

Sin embargo, la clave no está solo en conocer estas herramientas, sino en aplicarlas con criterio, responsabilidad y espíritu crítico. La IA puede ser un aliado poderoso, pero es esencial mantener la autonomía del pensamiento humano, evaluar la calidad del código y comprender los alcances éticos y técnicos de su uso.

Te animamos a que sigas explorando, probando y aprendiendo con estas tecnologías. En un mundo donde la automatización avanza a pasos agigantados, tu habilidad para combinar conocimientos técnicos con herramientas inteligentes marcará la diferencia en tu carrera profesional. ¡Seguí programando, pero también aprendiendo a pensar con algoritmos y ética!

Referencias📚

- OpenAI. (2024). *ChatGPT para desarrolladores*. https://platform.openai.com/docs
- GitHub Copilot. (2024). *Documentation & Guides*. https://docs.github.com/en/copilot
- JetBrains. (2024). *AI Assistant*. https://www.jetbrains.com/ai/
- Visual Paradigm. *What is Algorithm Flowchart?*. https://www.visual-paradigm.com/guide/programming/what-is-algorithm-flowchart/
- PSeInt. *Manual de usuario y ejemplos prácticos*. https://pseint.sourceforge.io/
- Mozilla Developer Network (MDN). *JavaScript Algorithms and Data Structures*. https://developer.mozilla.org/
- Towards Data Science. (2023). *How to Use ChatGPT to Write Better Code*. https://towardsdatascience.com
- DeepLearning.AI. *Generative AI for Developers Course*. https://www.deeplearning.ai/
- Microsoft Learn. *Prompt Engineering Guide*. https://learn.microsoft.com/en-us/azure/ai-services/openai/concepts/prompt-engineering
- FreeCodeCamp. *Best Practices for Prompting and Debugging with AI*. https://www.freecodecamp.org/news/tag/ai/

¡Muchas gracias!

Nos vemos en la próxima lección

alkemy