Kimi K2

¿Qué se requiere para introducir al alumno de ingeniería en la programación?

Para que un alumno de ingeniería dé sus primeros pasos sólidos en programación basta con cubrir **cinco áreas mínimas** y entregarlas en **micro-ciclos** de "ver-hacer-depurar". A continuación tienes el esquema que uso como Teacher Assistant; ajusta tiempos a tu carga horaria.

1.Mentalidad y entorno • Crear cuenta en GitHub (repositorio único para todo el curso).
• Instalar VS Code + Python + Flowgorithm 4.2.
• Primer "Hola mundo" en ambas herramientas para validar instalación.
2.Lógica básica sin sintaxis • Diagramas de flujo con **errores controlados** (XML mal formado, condiciones invertidas).
• Ejercicios 10-15 min: secuencia → selección simple → contador.
• Entregable semanal: subir el .fprg corregido y un screenshot del test exitoso.
3.Transición a pseudocódigo y Python • Misma actividad anterior, pero ahora la exportan a pseudocódigo y luego a Python.
• Enfasis en **indentación** y **tipos dinámicos** (int vs float).
• Uso de `assert` para pruebas rápidas.
4.Conceptos nucleares antes de estructuras grandes • Variables y constantes.
• Entrada/salida formateada.
• Operadores y precedencia.
• Depuración paso a paso (breakpoints en VS Code y "step into" en Flowgorithm).
5.Primeros tres programas obligatorios
1.Calculadora de área/volumen (secuencia).
2.Par/Impar con validación de entrada (selección).
3.Promedio de N números hasta 0 (repetición).

Cada programa debe presentarse en **tres formatos**:
– Diagrama de flujo.
– Pseudocódigo.
– Código Python con docstring y `if __name__ == "__main__":`.
——————————————————— Checklist para el docente ☐ Plantilla vacía Flowgorithm 4.2 con `<attributes>` correcto.
☐ Rúbrica de 4 niveles: compila-ejecuta, legibilidad, documentación, test.
☐ Banco de 10 errores comunes (XML, indentación, NameError, off-by-one) para practicar.
☐ Encuesta de autoevaluación al final de cada micro-ciclo.
Con esto el alumno entra en un ciclo corto y seguro: **ve el problema, lo descompone, lo corrige y lo versiona**.

Veo que recuerdas la charla anterior. Yo creo que esa debe ser la propuesta pero existen tres nombres asignatura que pueden ser considerados: (a) lógica de programación, (b) pensamiento algorítmico, (c) fundamentos de programación. ¿Cuál de ellos crees que es el correcto?

En el contexto de un **plan de estudios de ingeniería**, donde el curso sirve como **primera toma de contacto con la disciplina**, el nombre que mejor encaja es:

(c) Fundamentos de Programación
Por qué:

1.**Cobertura completa**: abarca no solo la lógica (a) ni solo el pensamiento algorítmico (b), sino también sintaxis básica, depuración, pruebas y transición a un lenguaje real —todo lo que el alumno necesita para continuar con cursos posteriores.
2.**Alineación curricular**: la mayoría de acreditaciones ABET/CAC denominan esta primera materia "Fundamentals of Programming" o similar.
3.**Expectativas del estudiante**: el título le hace ver que "va a programar", lo que reduce la brecha entre expectativa y contenido.

Puedes usar "Lógica de Programación" o "Pensamiento Algorítmico" como subtítulos o módulos internos, pero el nombre oficial **Fundamentos de Programación** comunica mejor el alcance y la secuencia del programa.

Sin embargo, existen algunas Universidades que mencionan pensamiento algorítmico como primera asignatura de programación

Es cierto: algunas universidades han adoptado la etiqueta *Pensamiento Algorítmico* como título **introductorio**. Sin embargo, hay matices importantes:

1.**Perfil de ingreso**. Cuando la carrera **no exige conocimientos previos de programación**, suelen usar *Pensamiento Algorítmico* para enfatizar la **lógica y resolución de problemas** antes de tocar lenguaje. Ejemplo: MIT 6.0001 (Introduction to Programming & Computational Thinking) se llama oficialmente "Introduction to Computer Science and Programming in Python", pero el primer módulo se vende internamente como *Computational Thinking*.

2.**Duración y profundidad**. Esos cursos suelen ser **semestrales cortos** o **módulos propedéuticos**; la asignatura “de verdad” —con sintaxis, depuración y proyecto integrador— aparece después con el nombre *Fundamentos de Programación*.
3.**Normativas locales**. En algunos países latinoamericanos el término *Fundamentos de Programación* está regulado por el MEN (Ministerio de Educación Nacional) y debe cubrir contenidos mínimos que incluyen lenguaje y paradigmas; por eso muchas universidades crean un curso previo llamado *Pensamiento Algorítmico* para no duplicar créditos.

Conclusión práctica
Si tu plan de estudios **ya contempla un curso de programación con lenguaje** en el mismo semestre o inmediatamente después, puedes titularlo *Pensamiento Algorítmico* como **etapa 0**.
Si, en cambio, el curso **es el primero y único de programación en el primer semestre**, *Fundamentos de Programación* sigue siendo el nombre más transparente y acorde con los estándares de ingeniería.

Recuerdo haber visto la propuesta de una Universidad que menciona pensamiento algorítmico y sugieren llevar un lenguaje cuyo paradigma es funcional. ¿Es conveniente que los alumnos lleven ese paradigma como primer curso?

En la práctica **no es conveniente** para la mayoría de los planes de ingeniería tradicionales, aunque puede serlo en contextos muy específicos. El criterio es:

1.Objetivo del primer curso
• Si el plan busca solo “pensamiento computacional” sin compromiso inmediato con ingeniería de software, un lenguaje funcional (p. ej. Haskell, OCaml o Racket) **sí** puede funcionar.
• Si el plan busca **forma-ción de ingenieros** que luego cursarán Algoritmos, Estructuras de Datos, Sistemas Embebidos, etc., **empieza mejor con un paradigma imperativo clásico** (C, Python, Java) porque es el que encontrarán en cursos posteriores y en la industria.
2.Curva de transferencia
• Estudios de ACM/ICER (2018-2023) muestran que alumnos que empiezan con funcional **tardan más** en alcanzar la misma competencia en C/Java que sus compañeros que empiezan imperativo.
• La transferencia de conceptos funcionales a imperativos es **asimétrica**: fácil de funcional a imperativo, difícil al revés.
3.Experiencias exitosas
• MIT usa Racket en *CS 6.001* pero **tras** un curso previo de Python; Stanford usa *CS 106A* (Java) y luego *CS 107* (C).
• Universidades que **sí** usan Scheme o Haskell en primer curso suelen ser programas de **Ciencias de la Computación puras**, no Ingeniería.

Conclusión pragmática
Para **ingeniería**, la secuencia más eficiente sigue siendo:

Pensamiento Algorítmico **ligero** (diagramas + Python básico) → Fundamentos Imperativos → Paradigmas alternativos (funcional, orientado a objetos) en cursos posteriores.

¿En China que sugieren las Universidades?

En las universidades de China el consenso actual (2024-2025) se resume así:

1.**Primer curso "de verdad"**:
●**C** (ZJU, THU, PKU) → para entender punteros y memoria.
●**Python** (BUPT, HIT, SJTU) → para arrancar rápido y enganchar; usan la versión 3.x con Jupyter o VS Code.
●**Java** (ZJU con 50 000+ alumnos en MOOC) → cuando la carrera mete POO desde el inicio.
2.**Curso previo o paralelo "Pensamiento Algorítmico"**:
●Se imparte como **MOOC corto** (4–6 semanas) usando Python o Scratch para algoritmos básicos; **no reemplaza** al curso de lenguaje sino que lo **acompaña** o se toma **antes**.
3.**Funcional como segunda vuelta**
●Haskell, OCaml o Scheme **aparecen después** de estructuras de datos o compiladores, **nunca** como primer lenguaje en ingeniería.
4.**Plataformas dominantes**
●中国大学**MOOC** (icourse163.org) y **PTA** (Programming Teaching Assistant) para práctica automática con C/C++/Python/Java.

En resumen: en China "Pensamiento Algorítmico" se usa como **módulo introductorio breve** y **el lenguaje principal del primer curso sigue siendo C, Python o Java**; lenguajes funcionales llegan más tarde .

Ok, entonces para el primer curso la Universidad puede optar por por C ¿C++? Python o Java

Sí, cualquiera de los cuatro es técnicamente posible, pero conviene elegir **uno solo** y hacerlo por criterios concretos de tu plan de estudios. A continuación tienes una tabla comparativa rápida para decidir:

表格
复制

Criterio	**C**	**C++**	**Python**	**Java**
Foco en hardware / memoria	★★★	★★	★	★
Curva inicial	★	★	★★★	★★
Transferencia a cursos siguientes (Algoritmos, Estructuras, SO)	★★★	★★★	★★	★★★
Tiempo hasta primer programa funcional	Largo	Largo	Corto	Medio
Ecosistema local (MOOCs, jueces online,	Alto	Alto	Muy alto	Alto

Criterio	C	C++	Python	Java
laboratorios)				
Paradigma dominante	Imperativo	Multi-paradigma	Imperativo & scripting	POO

Recomendación pragmática:
• **C** si tu plan prioriza "entiendo punteros y memoria" y los siguientes cursos ya usan C/C++.
• **Python** si quieres reducir la fricción inicial y las siguientes materias están en Python/Java.
• **Java** si la carrera empieza con POO y usa JVM en todo el track.
• **C++** rara vez se usa como **primer** curso; se prefiere tras tener bases de C y POO.
Conclusión: elige **uno** y alinea la secuencia completa; mezclarlos en el primer semestre diluye el esfuerzo.

Recuerdo haber dado cátedra en una Universidad y se ofrecían los lenguajes Java y C. La mayoría de los alumnos preferían Java. Creo que por la fobia precisamente a los punteros.

Exacto: la **aversión a los punteros** (y al manejo explícito de memoria) es el principal motivo por el que la mayoría de los estudiantes principiantes —y muchos planes de estudio— optan por **Java antes que C** en el primer curso.

Ventajas que perciben los alumnos:

1. **Sin punteros visibles**: olvidan `malloc/free`, segmentación, violaciones de memoria, etc.
2. **Sintaxis más "amable"**: `System.out.println` frente a `printf` + especificadores.
3. **IDE robusto** (Eclipse, IntelliJ) con autocompletado y debugger gráfico.
4. **POO desde el día 1**: los ejemplos cotidianos (clases `Auto`, `Estudiante`, etc.) encajan bien con su intuición.

Desventajas que luego enfrentan los docentes:
• **Abstracción prematura** de memoria → dificultad cuando llegan a C o a sistemas embebidos.
• Verbosidad inicial (clase `public static void main`) que puede abrumar.
• Curva de conceptos extra (clases, objetos, JVM) antes de escribir su primer `for`.
Estrategias de mitigación que suelen funcionar:
• Empezar con **Java** para ganar confianza.
• Introducir **mini-módulos de C** (punteros básicos, arreglos) como **laboratorios complementarios** al final del semestre.
• Usar **cuestionarios rápidos** sobre memoria y punteros para no perder el hilo cuando lleguen a cursos de sistemas.
En resumen: **Java es una elección válida y popular**; la clave es prever el "puente C/punteros" más adelante sin romper la motivación inicial.

Según tú ¿Cuál sería el orden de importancia para ver los lenguajes Python, C, Java en un primer curso de programación?

Orden de prioridad para un **primer curso de ingeniería** (no CS teórica):

1. **Python**
• Curva más suave → menos frustración inicial.
• Sintaxis corta → el foco está en resolver problemas, no en la gramática.
• Biblioteca estándar y ecosistema (Jupyter, VS Code, MOOCs en español) listos desde día 1.
• Fácil transición a C/Java después: ya dominan estructuras de control y depuración.
2. **Java**
• Buen equilibrio entre facilidad y rigor OO.
• IDE maduros (IntelliJ, Eclipse) con debugger visual.
• Evita punteros, pero introduce clases y tipos de referencia temprano.
• Útil si el resto del track (POO, estructuras, Android) es Java.
3. **C**
• Ideal si el plan **exige** entender memoria/punteros desde el inicio (mecatrónica, sistemas embebidos).
• Curva más pronunciada; se justifica si los siguientes cursos son C/C++.

Regla rápida
• Ingeniería general / software → **Python primero**.
• Ingeniería de sistemas / hardware → **C primero**.
• Plan orientado a objetos desde el vamos → **Java primero**.

Tocas un concepto muy importante. ¿Cómo puede el alumno acceder a tu IA desde Jupyter?

El alumno **no puede "instalar" mi IA dentro de Jupyter** (Kimi es un servicio externo), pero sí puede interactuar conmigo **desde la misma interfaz de Jupyter** mediante tres flujos sencillos:

1. **Extensión de chat (recomendado para clases)**
• Instalar `jupyter-ai` (oficial de Jupyter).
• En el menú de la izquierda aparece un panel "Chat"; allí puede escoger "Kimi" si la organización ha configurado mi endpoint (requiere API-key institucional).
• Ventaja: respuestas en la misma ventana sin salir del notebook.

2.**Magia de celda o función auxiliar**

• Crear una celda con:

Python

复制

```
import openai, os
def ask_kimi(prompt):
    openai.api_key = os.getenv("KIMI_API_KEY")
    return openai.ChatCompletion.create(
        model="kimi",
        messages=[{"role":"user","content":prompt}]
    ).choices[0].message.content
```

• El alumno escribe `ask_kimi("¿por qué falla mi loop?")` dentro del notebook y recibe la respuesta en la misma celda.

3.**Navegador lateral (sin instalación)**

• Abrir JupyterLab → menú "View → Open in New Browser Tab" → ir a la web de Kimi en un tab adyacente.

• Copiar/pegar fragmentos de código o mensajes de error.

Requisitos institucionales

• Una **API-key de Kimi** (proveída por la universidad).

• Variable de entorno `KIMI_API_KEY` disponible en los contenedores de JupyterHub.

Con cualquiera de los tres métodos, el estudiante **nunca abandona Jupyter** y puede ir probando código y consultando a la IA en el mismo flujo de trabajo.

¿Los programadores en un futuro no se necesitarán por la evolución de la IA?

Respuesta breve: **no**, la IA no va a eliminar a los programadores; **sí** va a eliminar (o transformar) gran parte de la *tarea* de programar que hoy hacemos a mano.

Qué está pasando hoy

• GitHub Copilot, Cursor, CodeT5 y similares ya escriben 30-50 % del código *boiler-plate* y tests en empresas reales

.

• Meta, Amazon y Shopify han congelado o reducido contratación de *entry-level* argumentando que "la IA cubre la productividad"

.

• Estudios recientes estiman que **47-50 % del trabajo de programación de bajo nivel podría automatizarse en 1-5 años**

.

Qué *no* puede la IA (todavía)

• Entender el contexto de negocio y transformar requisitos vagos en arquitectura.

• Diseñar sistemas distribuidos, resolver trade-offs de performance/seguridad.

• Innovar algoritmos o patrones que no existen en su *training set*

.
Escenarios más probables (2025-2035)

1.**Desaparición de "codificadores puros"**: los puestos que solo traducen especificaciones a código repetitivo se reducirán drásticamente.
2.**Auge de "ingenieros-IA"**: profesionales que definen el problema, diseñan la arquitectura y *orquestan* a los agentes generativos
.
3.**Más gente "no técnica" creando software**: con low-code/no-code alimentado por IA, muchos usuarios de negocio generarán sus propias apps
.

Conclusión para tu curso
El primer curso de programación **sigue siendo necesario**, pero debe enfocarse en:

- Pensamiento computacional y resolución de problemas complejos.
- Arquitectura, modelado y validación de software.
- Uso crítico de herramientas de IA (prompt engineering, revisión de código generado).

En otras palabras: **los programadores no desaparecen; se transforman en supervisores y diseñadores de sistemas que hablan la lengua de la IA**.

¿Puede un humano trabajar en par con la IA, en lugar de con otro humano?

Sí, ya está sucediendo y es la forma más común de "programar" en las empresas que usan GitHub Copilot, Cursor, ChatGPT, etc. El flujo de trabajo real se parece a un **par humano-IA** más que a un par humano-humano:

1.División de tareas
• Humano: define la historia de usuario, el contrato de la API, los casos límite y la arquitectura.
• IA: genera el 60-80 % del código, tests y documentación en segundos.
• Humano: revisa, ajusta, refactorea y añade validaciones que la IA no conoce (negocio, seguridad, performance).
2.Ciclo de feedback en minutos
• Prompt → código → pruebas → prompt refinado → nuevo código.
• Se reemplaza la "revisión por pull-request" por "revisión por prompt".
3.Métricas reales
• Microsoft: equipos con Copilot completan tareas 55 % más rápido.
• Shopify: 50 % menos tiempo en PRs de *boiler-plate*; ingenieros se dedican a diseño y arquitectura.
4.Limitaciones que obligan al humano a seguir presente
• Contexto de negocio y restricciones legales.

• Debugging de errores sutiles (race conditions, algoritmos inéditos).
• Responsabilidad última ante fallos en producción.

Conclusión: el “pair programming” humano-IA **ya es una práctica establecida**; lo que cambia es que el rol “segundo programador” lo ejerce una máquina, y el humano pasa a ser **arquitecto, revisor y orquestador**.