

Quiz #1 Lenguajes de Programación

Miguel Ángel Celis López

Universidad Sergio Arboleda

Lenguajes de Programación

Joaquin Fernando Sanchez

Bogotá, Colombia

2026

1. Objetivo:

Formular, interpretar y diseñar tres ejercicios prácticos orientados a reforzar los conocimientos sobre estructuras de datos y programación funcional, como introducción a la asignatura de Lenguajes de Programación. En este taller, se resaltan funciones de un navegador web y de un sistema de recomendaciones orientado a los productos y a las necesidades del usuario.

1.1 Objetivos Específicos:

- Diseñar e implementar la solución de los tres problemas propuestos utilizando de forma adecuada las estructuras de datos estudiadas a lo largo de la carrera.
- Codificar los algoritmos de cada problema en un lenguaje de programación específico (por ejemplo, Python) y representar su funcionamiento mediante diagramas que faciliten su comprensión y explicación.

2. Desarrollo:

Problema Número 1:

Sistema de navegación de páginas web

Diseñe un sistema que permita a los usuarios navegar por las páginas web, incluida la capacidad de volver a la página anterior y avanzar a la página siguiente, de forma similar a como funcionan los navegadores web modernos.

Cree un diagrama de flujo o diagrama que ilustre cómo funcionará la navegación.

Defina los métodos que necesitará, como 'goBack()', 'goForward()' y 'loadPage(url)'.

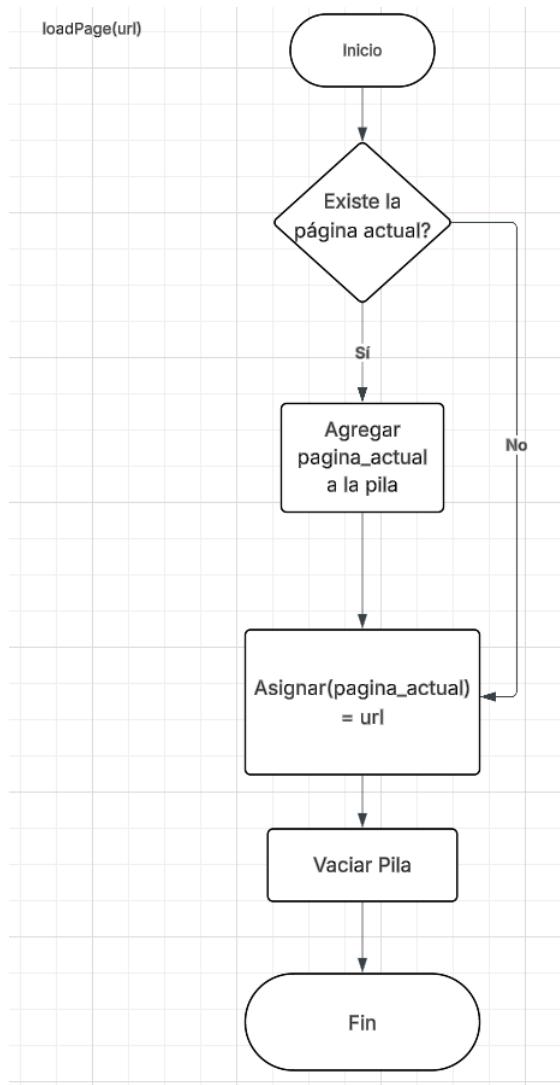
Implementar la solución usando Python:

Escribe el código de tu sistema de navegación.

Asegúrese de manejar casos extremos, como intentar retroceder cuando no hay páginas anteriores.

Diagrama de flujo de la navegación:

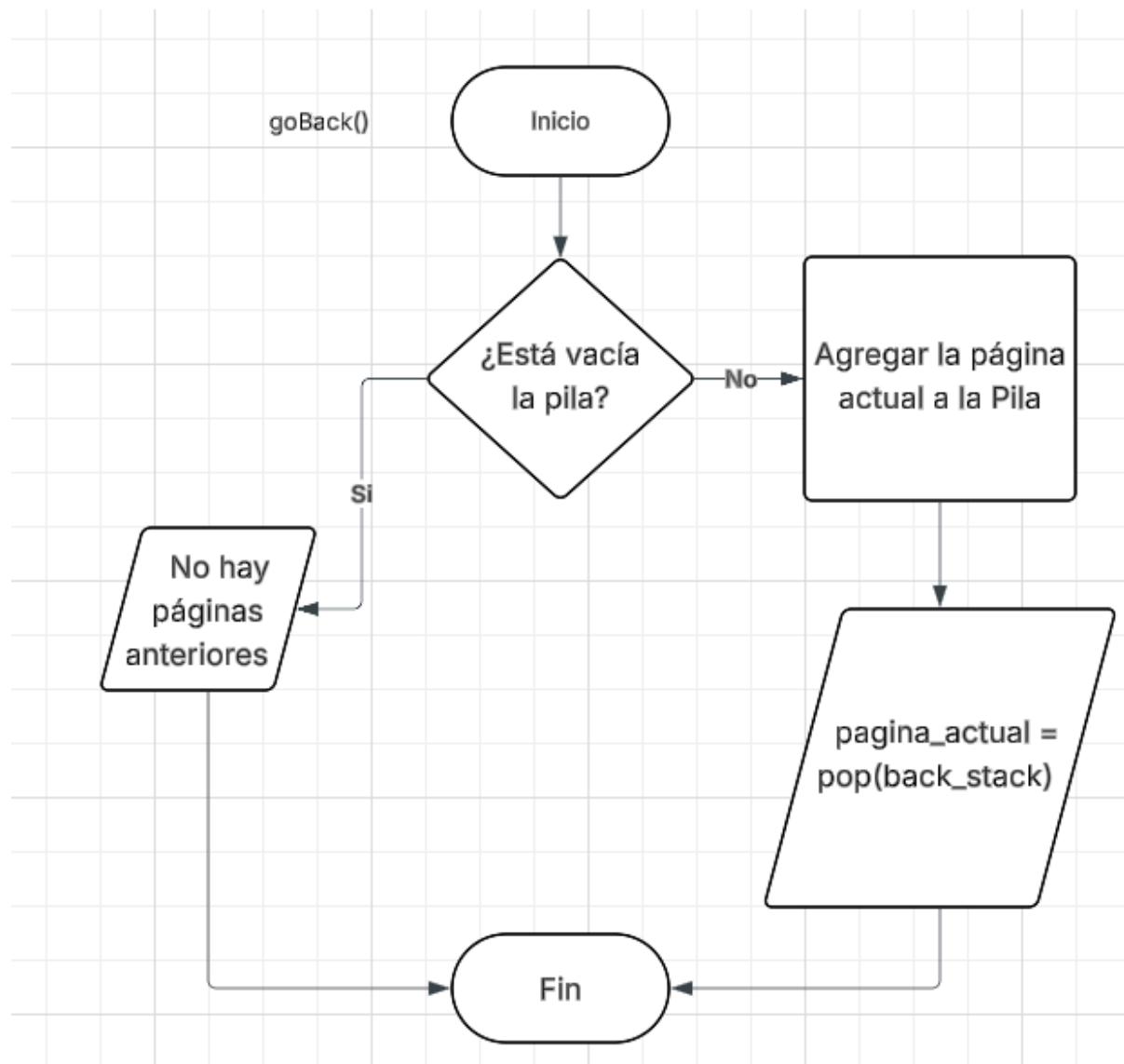
Diagrama del funcionamiento de loadPage(url):



Este diagrama de flujo muestra la lógica encargada al momento de que el sistema, cuando el usuario solicita navegar a una nueva dirección web. Se explicará de manera detallada:

Lo primero que se muestra, es un rombo de decisión, el cual indica si ya existe una página abierta, en el caso de que si, la asigna a la pila y se le asigna una URL a la página actual, para que el usuario pueda visualizarla, en el caso de que no exista ninguna página abierta, se omite el guardado de la página y se inicializa desde cero el navegador.

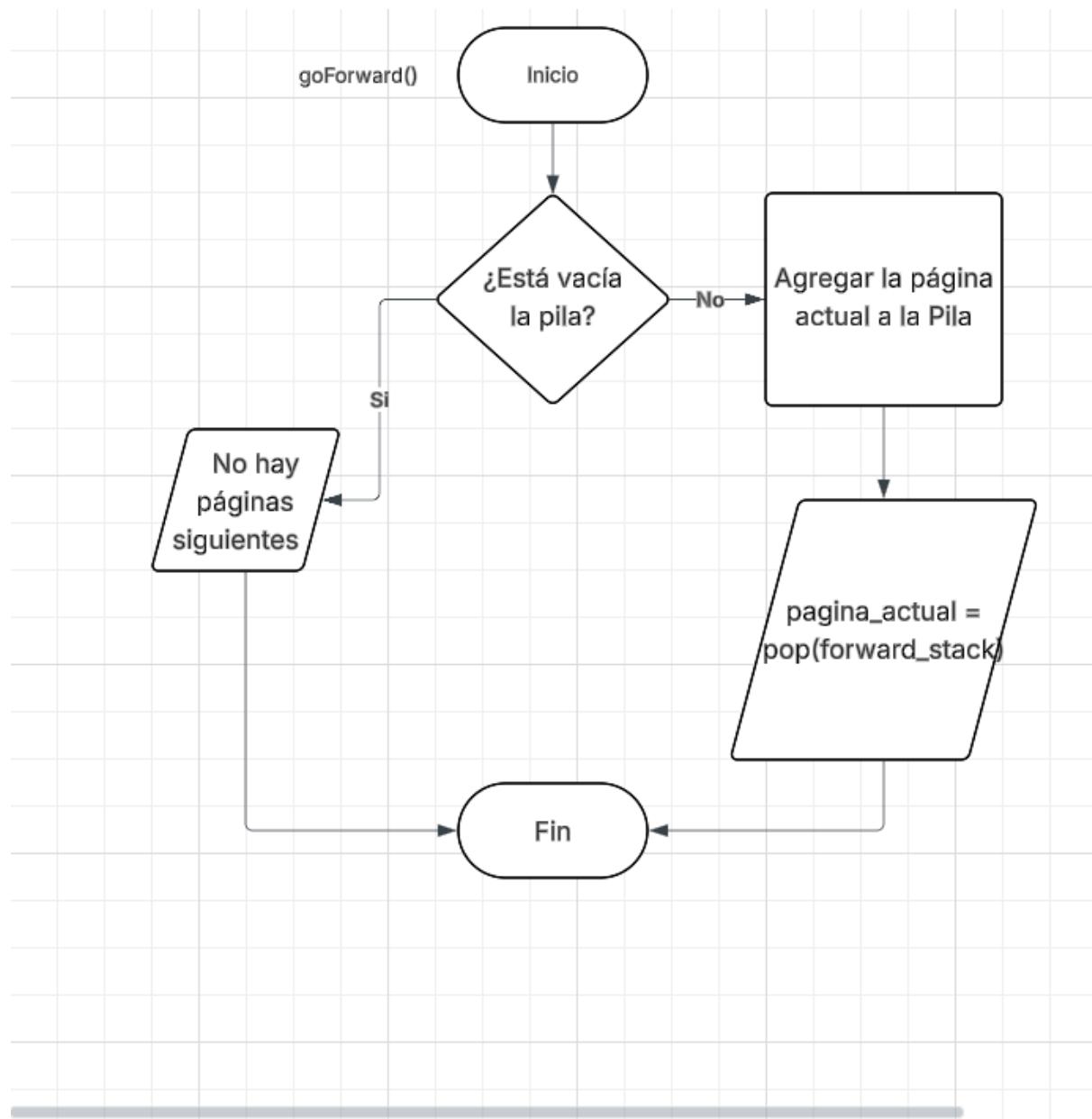
Diagrama del funcionamiento de goBack():



Este diagrama de flujo muestra la lógica encargada al momento de que el sistema recibe la orden de retroceder a una página visitada anteriormente. Se explicará de manera detallada:

Lo primero que se muestra, es un rombo de decisión, el cual verifica si la pila de historial está vacía; en el caso de que sí (esté vacía), el sistema lanza un aviso indicando que no hay páginas anteriores y termina el proceso. En el caso de que no (es decir, sí hay historial), se toma la página actual y se agrega a la pila de "siguentes" para no perderla, y finalmente se recupera la última página guardada en el historial para que el usuario pueda visualizarla nuevamente.

Diagrama del funcionamiento de goForward():



Este diagrama de flujo muestra la lógica encargada al momento de que el sistema recibe la orden de avanzar hacia una página siguiente (una que fue visitada previamente antes de retroceder). Se explicará de manera detallada:

Lo primero que se muestra, es un rombo de decisión, el cual verifica si la pila de "siguientes" está vacía; en el caso de que sí (esté vacía), el sistema lanza un aviso indicando que "No hay páginas siguientes" y termina el proceso, ya que no existe un camino hacia adelante. En el caso de que no (es

decir, sí hay páginas guardadas en el futuro), se toma la página actual y se agrega a la pila de "anteriores" (historial) para no perderla, y finalmente se recupera la página que estaba en la espera en la pila de "siguientes" para asignarla como actual, permitiendo que el usuario pueda visualizarla.

Código Fuente:

```
class NavegadorWeb:  
    def __init__(self):  
        # Se inicializa back_stack y forward_stack como listas vacías  
        self.paginasAnteriores = []  
        self.paginasSiguientes = []  
  
        #Se inicializa como 'None' dado que no hay una página actual  
        self.paginaActual = None  
  
    #Carga una nueva página  
    def loadPage(self, url):  
        # Si no es la primera página, se agrega a la pila de páginas anteriores  
        if self.paginaActual is not None:  
            #Se agrega la página actual a la pila de páginas anteriores  
            self.paginasAnteriores.append(self.paginaActual)  
  
        #Se actualiza la página actual  
        self.paginaActual = url  
        self.paginasSiguientes.clear()  
        print(f"Navegando a.....{self.paginaActual}")  
  
    #Regresa a la página anterior  
    def goBack(self):  
        # Si no hay páginas anteriores, se muestra un mensaje  
        if not self.paginasAnteriores:  
            print("No hay paginas anteriores")  
            return  
  
        self.paginasSiguientes.append(self.paginaActual)
```

```
self.paginaActual = self.paginasAnteriores.pop()

print(f'Regresando a.....{self.paginaActual}')

def goForward(self):

    if not self.paginasSiguientes:
        print("No hay paginas siguientes")
        return

    self.paginasAnteriores.append(self.paginaActual)
    self.paginaActual = self.paginasSiguientes.pop()

    print(f'Avanzando a.....{self.paginaActual}')

if __name__ == "__main__":
    firefox = NavegadorWeb()

    print("BIENVENIDO A FIREFOX")
    print("-----")

    firefox.loadPage("google.com")
    firefox.loadPage("youtube.com")
    firefox.loadPage("facebook.com")
    firefox.loadPage("instagram.com")
    firefox.loadPage("x.com")

    print("Regresando a la pagina anterior")
    firefox.goBack()
    firefox.goBack()

    print("Avanzando a la pagina siguiente")
    firefox.goForward()
    firefox.goForward()
```

```
print("Probando Nueva Ruta")
firefox.loadPage("discord.com")

print("Avanzado Fallido")
firefox.goForward()
```

Explicación del Código:

Este código simula el funcionamiento interno de un navegador básico (como Firefox) utilizando dos listas en forma de pilas para organizar el historial. La lógica se basa en tres acciones principales: loadPage, que abre un sitio nuevo y borra automáticamente cualquier historial futuro porque has cambiado de rumbo; goBack, que te permite retroceder guardando tu página actual en una lista de espera (por si decides volver a avanzar); y goForward, que recupera esas páginas de la lista de espera. Básicamente, el programa mueve las páginas de una pila a otra para asegurar que los botones de "Atrás" y "Adelante" siempre te lleven al lugar correcto en el orden en que se navega.

Problema Número 2:

Problema 2: Función de autocompletar

Cree una función de autocompletar que, dado un prefijo, devuelva todas las palabras posibles que comienzan con ese prefijo, similar a un motor de búsqueda.

Determine cómo la función recibirá la entrada y devolverá los resultados.

Piense en la eficiencia de la búsqueda de palabras.

Cree un diagrama que ilustre cómo se estructurará el trie.

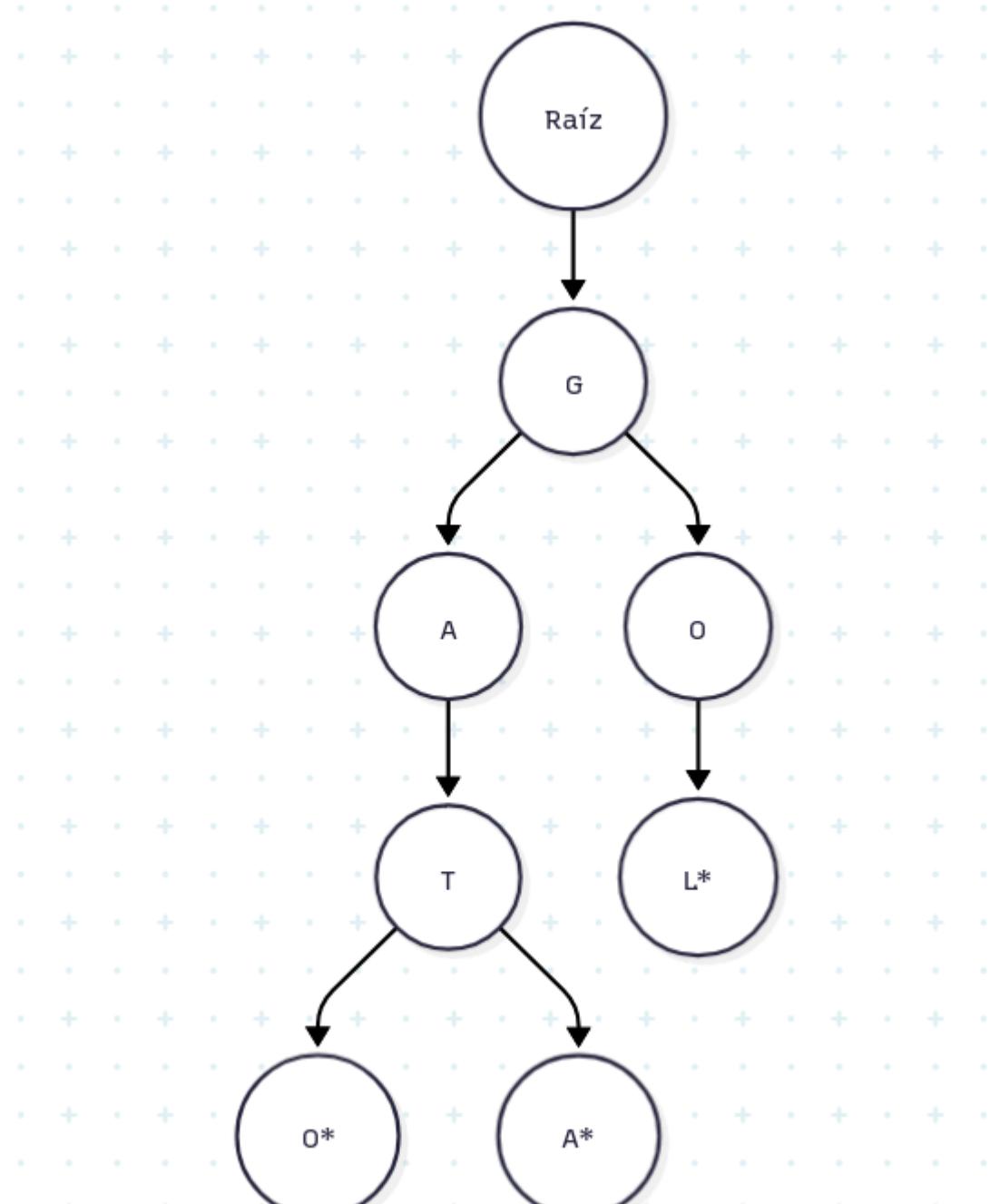
Defina los métodos que necesitará, como 'insert(word)' y 'autocomplete (prefix)'.

Implementar la solución en python:

Escribe el código para tu función de autocompletar.

Asegúrese de que pueda manejar varios prefijos y devolver las palabras correctas.

Diagrama del problema:



Este diagrama muestra cómo funciona por dentro un Trie (Árbol de Prefijos). El diagrama inicia desde un punto inicial "Raíz". En donde las palabras forman un camino para encontrar palabras: por ejemplo, "GATO" y "GATA" usan las mismas letras al principio ("G" → "A" → "T") y solo se separan al final. En cambio, "GOL" toma un camino diferente después de la primera letra. Por último, los círculos que tienen un asterisco (*) son marcas especiales que le avisan al programa que ahí termina una palabra completa.

Código Fuente:

```
class PTreeNode:  
    def __init__(self):  
        self.hijo = {}  
        self.fin_de_palabra = False  
  
class AutocompletarSistema:  
    def __init__(self):  
        self.root = PTreeNode()  
  
    def insertar(self, palabra):  
        nodo = self.root  
        for letra in palabra:  
            if letra not in nodo.hijo:  
                nodo.hijo[letra] = PTreeNode()  
            nodo = nodo.hijo[letra]  
        nodo.fin_de_palabra = True  
  
    def Autocompletar(self, prefijo):  
        nodo = self.root  
  
        for caracter in prefijo:  
            if caracter not in nodo.hijo:  
                return [] #  
            nodo = nodo.hijo[caracter]  
  
        resultados = []  
        # 2. Llamamos a la función recursiva que faltaba
```

```
self._recorrer(nodo, prefijo, resultados)
return resultados

def _recorrer(self, nodo, palabra_actual, resultados):
    if nodo.fin_de_palabra:
        resultados.append(palabra_actual)

    for letra, nodo_hijo in nodo.hijo.items():
        self._recorrer(nodo_hijo, palabra_actual + letra, resultados)

if __name__ == "__main__":
    print("BIENVENIDO AL SISTEMA DE AUTOCOMPLETAR")
    print("-----")

motor = AutocompletarSistema()

palabras = ["auto", "automatico", "automovil", "autobus", "arbol", "avion", "gato"]
print("Las palabras insertadas son: ", palabras)
print("-----")

for palabra in palabras:
    motor.insertar(palabra)

consulta = "auto"
print(f"Buscando prefijo: '{consulta}'")
print("Resultados:", motor.Autocompletar(consulta))

consulta2 = "a"
print(f"\nBuscando prefijo: '{consulta2}'")
print("Resultados:", motor.Autocompletar(consulta2))

consulta3 = "mp"
print(f"\nBuscando prefijo: '{consulta3}'")
print("Resultados:", motor.Autocompletar(consulta3))

print("\nFIN DEL PROGRAMA")
```

Explicación del Código:

El código implementado desarrolla un sistema de autocompletado eficiente utilizando la estructura de datos Trie (Árbol de Prefijos). La solución se estructura en dos clases principales: PTreeNode, que representa cada nodo del árbol conteniendo un diccionario de hijos y un indicador booleano (fin_de_palabra) para marcar el término de una palabra válida; y AutocompletarSistema, que gestiona la lógica del árbol. El algoritmo funciona en dos formas: primero, el método insertar construye la estructura anidada carácter por carácter. Luego, para la búsqueda, el método Autocompletar navega hasta el nodo que representa el final del prefijo ingresado y asigna la tarea al método auxiliar _recorrer. Este último utiliza recursividad para explorar todas las ramificaciones descendentes desde ese punto, recolectando y devolviendo todas las palabras posibles que completan el prefijo solicitado.

Problema Número 3: Sistema de recomendación de productos

Implemente un sistema que recomiende productos en función de las relaciones entre los usuarios y los productos, como "los usuarios que compraron X también compraron Y".

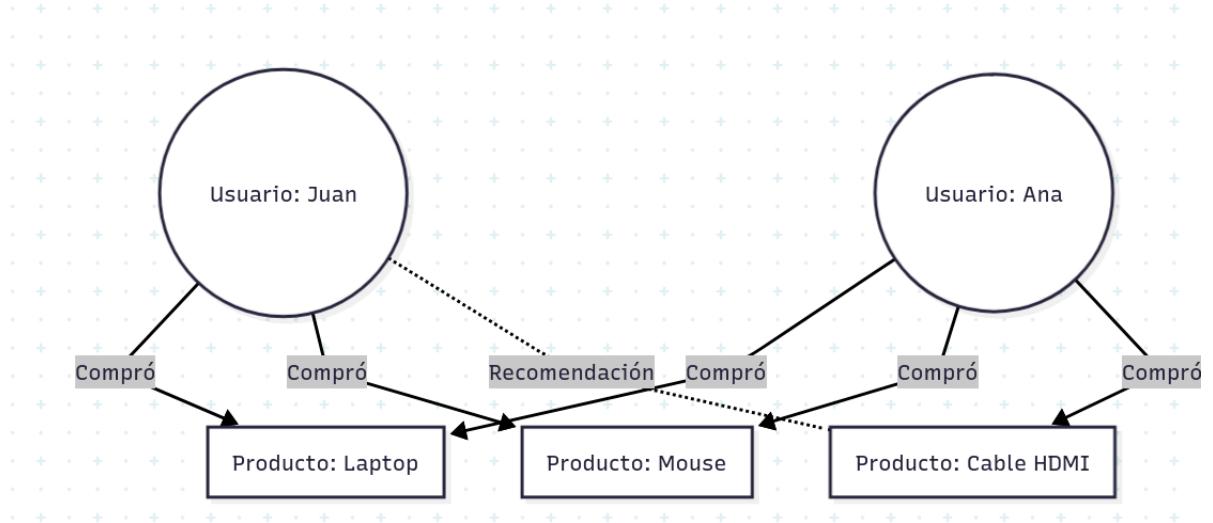
Cree un diagrama que ilustre cómo se conectan los usuarios y los productos.

Defina los métodos que necesitará, como 'addPurchase(usuario, producto)' y 'getRecommendations(usuario)'.

Escriba el código para su sistema de recomendaciones.

Asegúrese de que pueda manejar varias relaciones usuario-producto.

Diagrama (Grafo):



Explicación del Grafo:

El diagrama utiliza un grafo de relaciones entre usuarios y productos. Las líneas (“Compró”) representan el historial real de transacciones, evidenciando que tanto Juan como Ana comparten los mismos intereses al haber adquirido ambos una Laptop y un Mouse. Al detectar esta fuerte similitud en sus patrones de compra, el sistema identifica un producto que Ana tiene pero Juan no: el Cable HDMI. La línea punteada (“Recomendación”) ilustra la inferencia lógica del algoritmo, que sugiere este producto a Juan bajo la premisa de que usuarios con gustos parecidos tienden a querer los mismos artículos.

Código Fuente:

```

class SistemaRecomendacion:
    def __init__(self):
        self.compras = {}
        self.compradores = {}

    def addPurchase(self, usuario, producto):
        if usuario not in self.compras:
            self.compras[usuario] = set()
        self.compras[usuario].add(producto)

        if producto not in self.compradores:
            self.compradores[producto] = set()

    def recommend(self, usuario, producto):
        compras_usuario = self.compras.get(usuario, set())
        compradores_producto = self.compradores.get(producto, set())
        recomendados = compras_usuario & compradores_producto
        return list(recomendados)

```

```
self.compradores[producto].add(usuario)

print(f'Compra registrada: {usuario} compró {producto}')

def getRecommendations(self, usuario):
    if usuario not in self.compras:
        return []

    mis_prod = self.compras[usuario]
    recomendaciones = set()

    for producto in mis_prod:
        otros = self.compradores.get(producto, set())
        for otro_usuario in otros:
            if otro_usuario != usuario:
                prod_otro = self.compras[otro_usuario]
                recomendaciones.update(prod_otro)

    finales = recomendaciones - mis_prod

    return list(finales)

if __name__ == "__main__":
    print("--- SISTEMA DE RECOMENDACIONES ---")
    unilago = SistemaRecomendacion()

    unilago.addPurchase("Juan", "Laptop")
    unilago.addPurchase("Juan", "Mouse")

    unilago.addPurchase("Ana", "Laptop")
    unilago.addPurchase("Ana", "Mouse")
    unilago.addPurchase("Ana", "Cable HDMI")

    unilago.addPurchase("Pedro", "Monitor")

    print("\n--- GENERANDO RECOMENDACIONES ---")
```

```
recos_juan = unilago.getRecommendations("Juan")
print(f'Recomendaciones para Juan: {recos_juan}')

recos_ana = unilago.getRecommendations("Ana")
print(f'Recomendaciones para Ana: {recos_ana}')

recos_pedro = unilago.getRecommendations("Pedro")
print(f'Recomendaciones para Pedro: {recos_pedro}')
```

Explicación del Código Fuente:

El código fuente implementa un sistema de recomendación. Para optimizar las búsquedas, se utilizan dos diccionarios que almacenan conjuntos (set) de datos: compras, registra el historial de adquisiciones por usuario, y compradores, actúa como un índice invertido relacionando cada producto con los usuarios que lo adquirieron. El algoritmo reside en el método getRecommendations, el cual identifica primero a usuarios con patrones de consumo similares basándose en productos compartidos; posteriormente, agrupa el catálogo completo de estos vecinos y aplica una operación para filtrar los artículos que el usuario objetivo ya posee, devolviendo así una lista con sugerencias novedosas y relevantes.

3. Análisis

El desarrollo de estos tres problemas demuestra que la elección de la estructura de datos es el que permite que el programa funcione de la mejor forma. Habían distintas formas de resolver los problemas, pero las elegidas fueron una opción óptima y resultaron ser buenas.

- **Problema 1 - Navegador:** Se evidenció que el comportamiento de un navegador web es intrínsecamente "LIFO" (Last In, First Out). El uso de Pilas permitió manejar la temporalidad del historial de navegación de forma natural. Una lista convencional habría requerido índices complejos para saber en qué punto de la historia se encuentra el usuario, mientras que las pilas simplifican el proceso de "avanzar" y "retroceder" mediante operaciones push y pop de costo constante.

- **Problema 2 - Autocompletado:** Para el autocompletado, el análisis indica que una búsqueda lineal en una lista de palabras es ineficiente a medida que crece el vocabulario. La implementación de un arbol de prefijos o Trie optimizó el rendimiento, permitiendo que el tiempo de búsqueda dependa únicamente de la longitud del prefijo ingresado y no de la cantidad total de palabras almacenadas.
- **Problema 3 - Recomendaciones:** El sistema de recomendaciones mostró que los datos no siempre son lineales, sino relacionales. Al modelar el problema como un Grafo Bipartito (Usuarios-Productos) y utilizar operaciones de Conjuntos (Sets), se logró identificar relaciones indirectas entre usuarios (similitud de gustos) sin necesidad de recorrer bucles. El uso de diccionarios permitió accesos para filtrar y cruzar datos de compras como en el diagrama

4. Conclusión

La realización de estos tres problemas demuestra que la eficiencia y funcionalidad de cualquier software dependen de la elección correcta de la estructura de datos (en este caso). Al implementar soluciones para situaciones cotidianas como el historial de un navegador (Pilas), el autocompletado de búsquedas (Tries) y las recomendaciones de productos (Grafos) se concluye que no todo depende de tener un buen manejo de la programación, sino de saber elegir con que bases trabajar y mediante que cosas basarse, además de tener un gran apoyo antes de trabajar, en este caso los diagramas de cada problema. De la misma forma se puede conocer las distintas formas de cómo funcionan las cosas que usamos día a día, de una forma computacional.