Segundo ejercicio entregable: Ejercicios de descubrimiento

En estos ejercicios se exploran temas que van un poco más allá de lo que hemos visto en el temario pero que puedes investigar de formar sencilla con el conocimiento que sí has adquirido. Elige **sólo uno** y desarróllalo.

Si hay otro tema relacionado que te interese, propónselo al profesor.

Algoritmo de Backtracking para Resolver un Sudoku

Objetivo: Explorar el concepto de **backtracking**, una técnica algorítmica que se utiliza para resolver problemas que requieren la búsqueda de soluciones válidas a partir de una serie de decisiones. En este ejercicio, los estudiantes aprenderán cómo funciona el backtracking aplicándolo al juego clásico del Sudoku.

Descripción del Problema: El **Sudoku** es un rompecabezas de lógica que consiste en una cuadrícula de 9x9 celdas, dividida en subcuadrículas de 3x3. Cada fila, columna y subcuadrícula debe contener los números del 1 al 9 sin repetir.

El objetivo es implementar un algoritmo que pueda resolver un tablero de Sudoku incompleto utilizando **backtracking**.

Pasos para Resolver el Ejercicio:

1. **Introducción al Sudoku**:
    a. Mostrar un ejemplo de un tablero de Sudoku incompleto.
    b. Explicar las reglas básicas del juego:
        i. Cada fila debe tener números del 1 al 9 sin repetir.
        ii. Cada columna debe tener números del 1 al 9 sin repetir.
        iii. Cada subcuadrícula de 3x3 debe tener números del 1 al 9 sin repetir.
2. **Planteamiento del Problema**:
    a. Dado un tablero de Sudoku incompleto, encontrar una solución válida.
    b. Representar el tablero utilizando una **matriz bidimensional** (una lista de listas en Python).
3. **Descripción del Algoritmo de Backtracking**:
    a. **Algoritmo de Backtracking**:
        i. El backtracking es una técnica de prueba y error en la que se toman decisiones secuenciales para encontrar una solución válida. Si una decisión no lleva a una solución, se "retrocede" para probar otras opciones.
    b. **Pasos del Algoritmo**:
        i. Encuentra una celda vacía en el tablero.
        ii. Intenta rellenar la celda con un número del 1 al 9.
        iii. Si el número no viola las reglas del Sudoku, procede a rellenar la siguiente celda vacía.
        iv. Si no se puede rellenar una celda de forma válida, retrocede y prueba con otro número.
4. **Implementación en Pseudocódigo**:
    a. **Pseudocódigo del Algoritmo**:

```
resolver_sudoku(tablero):
    si no hay más celdas vacías:
        devolver True (el Sudoku está resuelto)

    para cada número del 1 al 9:
        si el número es válido en la celda actual:
            poner el número en la celda
            si resolver_sudoku(tablero):
                devolver True
            quitar el número (retroceder)

    devolver False (si no hay solución)
```

5. **Implementación en Python (opcional)**:
    a. Implementar el algoritmo de backtracking en Python para resolver un tablero de Sudoku dado.
    b. Crear una función que tome un tablero incompleto y lo resuelva de manera recursiva.
6. **Actividad Participativa**:
    a. **Pregunta para el grupo**: ¿En qué otros problemas creen que el backtracking podría ser útil?
    b. **Discusión en equipos**: Discutir cómo podríamos hacer el algoritmo más eficiente (por ejemplo, probando los números en un orden específico o utilizando técnicas para reducir las opciones).
7. **Extensión del Ejercicio**:
    a. Introducir el concepto de **heurísticas** para mejorar la eficiencia del backtracking.
    b. Mostrar cómo el **algoritmo de fuerza bruta** se diferencia del backtracking y cuál es más adecuado para el Sudoku.

Criterios de Evaluación:

- Implementación correcta del algoritmo de backtracking.
- Comprensión del proceso de retroceso y cómo se aplica en problemas de búsqueda.
- Explica posibles mejoras en la eficiencia del algoritmo.

Algoritmos Genéticos para Optimización

Objetivo: Explorar el concepto de **algoritmos genéticos**, que son técnicas de optimización basadas en el proceso de selección natural. En este ejercicio, los estudiantes aprenderán cómo funciona la selección, el cruce y la mutación para resolver un problema complejo.

Descripción del Problema: El objetivo es optimizar la asignación de recursos para la **extracción de especia** en el planeta Arrakis (inspirado en *Dune*). Los recursos son limitados y deben ser distribuidos de manera que se maximice la cantidad de especia recolectada, teniendo en cuenta restricciones de energía y número de trabajadores.

Pasos para Resolver el Ejercicio:

1. **Introducción a los Algoritmos Genéticos**:
    a. **Inspiración en la Naturaleza**: Explicar cómo los algoritmos genéticos se inspiran en la evolución y selección natural.
    b. **Conceptos Básicos**:
        i. **Población**: Conjunto de posibles soluciones al problema.
        ii. **Genes y Cromosomas**: Cada solución está compuesta por "genes", que son los parámetros a optimizar.
        iii. **Selección, Cruce y Mutación**: Mecanismos para evolucionar las soluciones.
2. **Planteamiento del Problema**:
    a. **Asignación de Recursos**: Tienes n zonas de extracción, `m` trabajadores, y e unidades de energía.
    b. **Objetivo**: Maximizar la cantidad de especia recolectada con las restricciones dadas.
    c. **Representación**:
        i. Cada solución se representa como una lista donde cada elemento indica la cantidad de trabajadores y energía asignados a una zona.
3. **Construcción de Algoritmo Genético**:
    a. **Inicializar la Población**: Crear una población inicial de posibles asignaciones de recursos.
    b. **Función de Evaluación (Fitness)**: Evaluar cada solución según la cantidad de especia recolectada.
    c. **Selección**: Seleccionar las mejores soluciones para la reproducción.
    d. **Cruce (Crossover)**: Combinar dos soluciones para producir una nueva solución.
    e. **Mutación**: Realizar pequeños cambios aleatorios en una solución para explorar nuevas posibilidades.
4. **Implementación en Pseudocódigo**:

```
inicializar_poblacion(tamaño):
    crear una lista de soluciones aleatorias

evaluar_fitness(poblacion):
    para cada solucion en poblacion:
        calcular especia recolectada y guardar valor

seleccion(poblacion):
    seleccionar las mejores soluciones basadas en su fitness

cruce(solucion1, solucion2):
    combinar las dos soluciones para producir una nueva

mutacion(solucion):
    cambiar aleatoriamente un parámetro de la solución

algoritmo_genetico():
    poblacion = inicializar_poblacion(tamaño)
    mientras no se alcance el criterio de finalización:
        evaluar_fitness(poblacion)
        nueva_poblacion = []
        mientras nueva_poblacion no esté completa:
            padres = seleccion(poblacion)
            hijo = cruce(padres[0], padres[1])
            si aleatorio() < probabilidad_mutacion:
                hijo = mutacion(hijo)
            agregar hijo a nueva_poblacion
        poblacion = nueva_poblacion
    devolver la mejor solución encontrada
```

5. **Implementación en Python (opcional)**:
    a. Implementar un programa simple que resuelva la optimización de recursos utilizando el pseudocódigo dado.
    b. Crear una función de evaluación que mida el rendimiento de cada solución.
6. **Actividad Participativa**:
    a. **Discusión en Equipos**: ¿Qué significa una buena "solución" en este contexto? ¿Qué implicaciones tiene para la eficiencia de la extracción de especia?
    b. **Pregunta para el Grupo**: ¿En qué otros problemas podríamos aplicar algoritmos genéticos?

7. **Extensión del Ejercicio**:
    a. Introducir el concepto de **elitismo** en los algoritmos genéticos, asegurando que las mejores soluciones pasen directamente a la siguiente generación sin cambios.
    b. Discutir las ventajas y limitaciones de los algoritmos genéticos frente a otros métodos de optimización.

Criterios de Evaluación:

- Implementación correcta del algoritmo genético.
- Comprensión de los conceptos de selección, cruce y mutación.
- Explica cómo mejorar el proceso de evolución.

Árboles de Decisión para Clasificación

Objetivo: Explorar el concepto de **Árboles de Decisión**, que es una técnica de **aprendizaje automático** utilizada para tareas de clasificación y regresión. En este ejercicio, los estudiantes descubrirán cómo se construyen los árboles de decisión y cómo se utilizan para tomar decisiones en problemas complejos.

Descripción del Problema: Imaginen que tienen un conjunto de datos de distintos tipos de plantas en Arrakis (inspirado en *Dune*), y necesitan clasificar si una planta puede sobrevivir en un clima específico basándose en características como:

- **Resistencia al calor** (alta, media, baja)
- **Necesidad de agua** (alta, media, baja)
- **Profundidad de la raíz** (profunda, media, superficial)

El objetivo es implementar un árbol de decisión que, basándose en las características de cada planta, determine si puede sobrevivir en un clima seco y caliente.

Pasos para Resolver el Ejercicio:

1. **Definir los Atributos y Resultados**:
    a. Cada planta tiene tres características principales:
        i. Resistencia al calor (`alta`, `media`, `baja`)
        ii. Necesidad de agua (`alta`, `media`, `baja`)
        iii. Profundidad de la raíz (`profunda`, `media`, `superficial`)
    b. El objetivo es determinar si la planta puede sobrevivir (`Sí` o `No`).
2. **Construcción del Árbol de Decisión**:
    a. Crear una representación sencilla de un **árbol de decisión**, donde cada nodo representa una pregunta sobre una característica de la planta (por ejemplo: "¿Resistencia al calor es alta?").
    b. Utilizar las respuestas (`sí` o `no`) para ir bajando por las ramas del árbol hasta llegar a una **hoja**, que es la decisión final (`Sí` o `No`).
3. **Implementación en Pseudocódigo**:
    a. Representar el árbol como un conjunto de condiciones anidadas.
    b. **Ejemplo**:

```
Si Resistencia al Calor = alta:
    Si Necesidad de Agua = baja:
        Entonces -> Sobrevivirá (Sí)
    Si Necesidad de Agua = media o alta:
        Entonces -> No sobrevivirá (No)
Si Resistencia al Calor = media:
    Si Profundidad de la Raíz = profunda:
```

```
        Entonces -> Sobrevivirá (Sí)
    De lo contrario -> No sobrevivirá (No)
Si Resistencia al Calor = baja:
    -> No sobrevivirá (No)
```

4. **Implementación en Python (opcional)**:
    a. Crear un pequeño programa que implemente este árbol de decisión utilizando funciones o estructuras condicionales (`if-elif-else`).
    b. Preguntar al usuario las características de una planta y determinar si la planta puede sobrevivir.
5. **Discusión y Descubrimiento**:
    a. **Pregunta para los estudiantes**: ¿Qué ventajas y desventajas tiene el uso de árboles de decisión frente a otros métodos?
    b. **Actividad de Descubrimiento**: Intentar modificar el árbol de decisión para incluir más características, y observar cómo se complica la estructura.
6. **Extensión del Ejercicio**:
    a. Introducir el concepto de **entropía** e **impureza de Gini** (sin entrar en detalles matemáticos complejos), para que los estudiantes tengan una idea de cómo los algoritmos construyen estos árboles de manera automática en aprendizaje automático.

Criterios de Evaluación:

- Correctitud del árbol de decisión construido.
- Capacidad de interpretar las decisiones tomadas por el árbol.
- Explica cómo mejorar el árbol y qué problemas podría tener.

Optimización del Entrenamiento de Jugadores de Baloncesto

Objetivo: Explorar el uso de **algoritmos de optimización y planificación** para gestionar los entrenamientos de un equipo de baloncesto. Los estudiantes aprenderán a asignar sesiones de entrenamiento específicas a los jugadores, basándose en sus fortalezas y debilidades, y a optimizar el tiempo y el uso de las instalaciones disponibles.

Descripción del Problema: El entrenador de un equipo de baloncesto tiene varios jugadores con distintas habilidades y niveles de condición física. Hay que asignar sesiones de entrenamiento específicas a cada jugador para que se preparen para un próximo partido importante. Además, el entrenamiento debe ajustarse a las limitaciones de tiempo y disponibilidad de la cancha.

Pasos para Resolver el Ejercicio:

1. **Introducción a la Optimización en el Entrenamiento**:
    a. **Problema de Asignación**: Se trata de un problema de asignación en el que los recursos (cancha, entrenadores) y las restricciones (disponibilidad de jugadores, tiempo de entrenamiento) deben gestionarse de forma eficiente.
    b. **Objetivo**: Maximizar la eficacia del entrenamiento asignando las sesiones más adecuadas a cada jugador según sus necesidades.
2. **Planteamiento del Problema**:
    a. **Entrenamientos Disponibles**:
        i. **Tiros de Media Distancia**
        ii. **Defensa en Equipo**
        iii. **Condición Física**
        iv. **Dribling y Control del Balón**
    b. **Jugadores**:
        i. Cada jugador tiene una puntuación de habilidad en cada una de estas áreas y un área en la que necesita mejorar.
    c. **Restricciones**:
        i. Hay solo una cancha disponible y el entrenamiento se realiza en una sesión de 3 horas.
        ii. Cada jugador puede tener un máximo de 30 minutos en cancha para entrenamientos específicos.
3. **Construcción del Algoritmo de Planificación**:
    a. **Representación**:
        i. Utilizar una **lista** de jugadores y una **lista** de entrenamientos disponibles.

        ii. Cada jugador tiene un diccionario con sus habilidades (`tiros`, `defensa`, `condición física`, `dribling`).
    b. **Criterios de Optimización**:
        i. Cada jugador debe recibir al menos una sesión para mejorar su área más débil.
        ii. La cancha debe estar ocupada en todo momento para maximizar el uso del tiempo disponible.
4. **Implementación en Pseudocódigo**:

```
planificar_entrenamiento(jugadores, entrenamientos, tiempo_total):
    crear una lista vacía para el plan de entrenamiento
    tiempo_por_jugador = tiempo_total / número de jugadores

    para cada jugador en jugadores:
        encontrar el área de menor habilidad del jugador
        asignar una sesión de entrenamiento en esa área
        agregar al plan de entrenamiento

    si hay tiempo adicional disponible:
        distribuir sesiones de refuerzo en habilidades adicionales

    devolver plan de entrenamiento
```

5. **Implementación en Python (opcional)**:
    a. Crear un programa que implemente el algoritmo de planificación.
    b. Crear una función que optimice las sesiones de entrenamiento y asigne a cada jugador su entrenamiento específico.
6. **Actividad Participativa**:
    a. **Pregunta para el Grupo**: ¿Cómo podríamos decidir qué jugador necesita más tiempo en cada entrenamiento?
    b. **Discusión en Equipos**: Dividir a los estudiantes en pequeños equipos y hacer que propongan su propio plan de entrenamiento para mejorar una habilidad específica del equipo (por ejemplo, mejorar la defensa en equipo).
7. **Extensión del Ejercicio**:
    a. **Asignación Dinámica**: Introducir el concepto de **programación dinámica** para planificar las sesiones de entrenamiento de manera óptima, minimizando los conflictos de tiempo entre los jugadores.
    b. **Análisis de Eficiencia**: Analizar cómo la disponibilidad de tiempo afecta la calidad del entrenamiento y cómo el uso de un algoritmo de planificación puede mejorar la gestión del tiempo.

Criterios de Evaluación:

- Correctitud de la asignación de las sesiones de entrenamiento.
- Comprensión del problema de asignación y de cómo optimizar los recursos disponibles.
- Cómo mejorarías el plan de entrenamiento.

Gestión de Estadísticas de Jugadores de Baloncesto con Listas y Diccionarios

Objetivo: Utilizar estructuras de datos como **listas** y **diccionarios** para gestionar las estadísticas de un equipo de baloncesto. Los estudiantes aprenderán a almacenar, organizar y buscar información sobre jugadores y sus rendimientos durante los partidos.

Descripción del Problema: El entrenador de un equipo de baloncesto necesita gestionar las estadísticas de sus jugadores, como puntos anotados, rebotes y asistencias, para tener una mejor visión del rendimiento del equipo durante los partidos.

Pasos para Resolver el Ejercicio:

1. **Introducción a la Gestión de Estadísticas**:
    a. Cada jugador tiene varias estadísticas clave: **puntos anotados**, **rebotes**, **asistencias**, **robos** y **bloqueos**.
    b. Se necesita una forma de **almacenar** y **consultar** estos datos de forma eficiente, usando listas y diccionarios.
2. **Representación de los Datos**:
    a. Utilizar un **diccionario** para representar a cada jugador, donde las claves son las estadísticas (`"puntos"`, `"rebotes"`, `"asistencias"`, etc.).
    b. Utilizar una **lista** para almacenar todos los jugadores del equipo.
    c. **Ejemplo**:

```
python
Copiar código
jugadores = [
    {"nombre": "Juan", "puntos": 15, "rebotes": 7, "asistencias": 4,
"robos": 2, "bloqueos": 1},
    {"nombre": "Pedro", "puntos": 10, "rebotes": 5, "asistencias":
7, "robos": 1, "bloqueos": 0},
    {"nombre": "Luis", "puntos": 20, "rebotes": 10, "asistencias":
2, "robos": 3, "bloqueos": 2}
]
```

3. **Ejercicio Práctico**:
    a. **Objetivo**: Implementar funciones que permitan gestionar y consultar las estadísticas de los jugadores.
    b. **Tareas a Realizar**:

i. **Mostrar las Estadísticas de un Jugador**: Crear una función `mostrar_estadisticas(nombre)` que busque un jugador por su nombre y muestre sus estadísticas.
ii. **Actualizar las Estadísticas**: Crear una función `actualizar_estadisticas(nombre, estadistica, valor)` que permita actualizar una estadística específica de un jugador (por ejemplo, incrementar los puntos anotados).
iii. **Calcular el Mejor Jugador**: Crear una función `mejor_jugador(estadistica)` que devuelva el jugador con el mejor valor en una estadística específica (por ejemplo, más rebotes).

4. **Implementación en Pseudocódigo**:

```
mostrar_estadisticas(nombre):
    para cada jugador en lista de jugadores:
        si jugador["nombre"] es igual a nombre:
            imprimir las estadísticas del jugador

actualizar_estadisticas(nombre, estadistica, valor):
    para cada jugador en lista de jugadores:
        si jugador["nombre"] es igual a nombre:
            jugador[estadistica] = valor

mejor_jugador(estadistica):
    inicializar mejor_valor como -1
    inicializar mejor_jugador como None
    para cada jugador en lista de jugadores:
        si jugador[estadistica] > mejor_valor:
            mejor_valor = jugador[estadistica]
            mejor_jugador = jugador["nombre"]
    devolver mejor_jugador
```

5. **Implementación en Python (opcional)**:
    a. Implementar las funciones utilizando listas y diccionarios.
    b. Hacer que los estudiantes prueben el código con diferentes valores para ver cómo cambian las estadísticas y cómo se determina el mejor jugador en una categoría.
6. **Actividad Participativa**:
    a. **Pregunta para el Grupo**: ¿Qué otra información creen que sería útil añadir a las estadísticas del jugador?

    b. **Discusión en Parejas**: Cada pareja debe idear una nueva estadística que podría ser interesante y explicar por qué. Luego deben agregarla a su estructura de datos.
7. **Extensión del Ejercicio**:
    a. **Ordenar los Jugadores**: Crear una función que ordene a los jugadores según una estadística dada (por ejemplo, puntos anotados) y muestre la lista ordenada.
    b. **Analizar Rendimiento**: Implementar una función que calcule el promedio de puntos anotados por todos los jugadores.

Criterios de Evaluación:

- Correctitud de las funciones implementadas.
- Uso adecuado de listas y diccionarios para organizar la información.
- Qué estadísticas serían útiles y cómo representarlas. Por ejemplo: rebotes de otros jugadores con un jugador en pista.

Gestión de Partidas de Ajedrez con Tableros y Movimientos

Objetivo: Utilizar **matrices** y **diccionarios** para representar un tablero de ajedrez y gestionar los movimientos de las piezas. Los estudiantes aprenderán a modelar el tablero, realizar movimientos legales y validar las reglas básicas del ajedrez.

Descripción del Problema: Un tablero de ajedrez tiene 8x8 casillas y está compuesto por diferentes tipos de piezas. El objetivo es implementar una representación del tablero y simular algunos movimientos de las piezas.

Pasos para Resolver el Ejercicio:

1. **Introducción a la Representación del Tablero de Ajedrez**:
    a. Un tablero de ajedrez se puede representar utilizando una **matriz bidimensional** (lista de listas en Python) de 8x8, donde cada celda puede contener una pieza o estar vacía.
    b. Cada pieza puede ser representada por una abreviatura (`P` para peón, `R` para torre, `N` para caballo, `B` para alfil, `Q` para reina, `K` para rey).
2. **Planteamiento del Problema**:
    a. **Objetivo**: Crear un tablero de ajedrez inicial y permitir que los jugadores realicen movimientos.
    b. **Movimientos Permitidos**:
        i. Empezar por movimientos simples de peones y torres.
    c. **Restricciones**:
        i. Solo se permiten movimientos legales para cada pieza.
        ii. Validar si una casilla está ocupada por otra pieza.
3. **Construcción del Algoritmo de Movimiento**:
    a. **Representación del Tablero**:
        i. Utilizar una **lista de listas** para representar las 8 filas y 8 columnas del tablero.
        ii. Cada celda puede contener un string que representa la pieza (`'P'`, `'R'`, etc.) o estar vacía (`None`).
    b. **Movimientos de Piezas**:
        i. Implementar funciones para mover un **peón** (un solo paso hacia adelante) y una **torre** (cualquier número de casillas en línea recta).
4. **Implementación en Pseudocódigo**:

```
inicializar_tablero():
    crear una matriz 8x8 con None
    colocar piezas iniciales para peones y torres en sus posiciones
```

```
mover_pieza(tablero, posicion_inicial, posicion_final):
    si posicion_inicial contiene una pieza:
        si el movimiento es válido para la pieza:
            mover la pieza a la posicion_final
            vaciar la posicion_inicial
        de lo contrario:
            imprimir "Movimiento no permitido"
    de lo contrario:
        imprimir "No hay ninguna pieza en la posición inicial"

es_movimiento_valido(pieza, posicion_inicial, posicion_final,
tablero):
    si la pieza es un peón:
        validar que el movimiento sea un paso hacia adelante
    si la pieza es una torre:
        validar que el movimiento sea en línea recta y que no haya
piezas bloqueando
    devolver True si el movimiento es válido, False de lo contrario
```

5. **Implementación en Python (opcional)**:
    a. Crear una función `inicializar_tablero()` que inicialice un tablero de ajedrez con peones y torres en sus posiciones iniciales.
    b. Implementar una función `mover_pieza()` que permita al usuario mover una pieza de una posición a otra y verifique si el movimiento es legal.
6. **Actividad Participativa**:
    a. **Pregunta para el Grupo**: ¿Cómo podríamos representar el resto de las piezas de ajedrez? ¿Qué reglas tendríamos que añadir para cada pieza?
    b. **Discusión en Equipos**: Dividir a los estudiantes en equipos y hacer que cada equipo elija una pieza (caballo, alfil, reina o rey) e intente escribir las reglas de movimiento para esa pieza.
7. **Extensión del Ejercicio**:
    a. **Captura de Piezas**: Implementar la lógica para que las piezas puedan capturar otras piezas si se encuentran en la posición final del movimiento.
    b. **Comprobación de Jaque**: Introducir el concepto de **jaque al rey** y validar si un movimiento deja al rey en jaque.

Criterios de Evaluación:

- Implementación correcta del tablero y los movimientos de las piezas.

- Comprensión de las reglas del ajedrez y cómo representarlas utilizando estructuras de datos.
- Explorar distintas formas para la representación de otras piezas y reglas adicionales.