Ejercicio Final

Descripción del Problema: Desarrolla un algoritmo eficiente para resolver un Sudoku utilizando una de las técnicas algorítmicas avanzadas: programación dinámica, divide y vencerás, o algoritmos voraces. Elige la técnica que consideres más adecuada y justifica tu selección en base a la estructura del problema y la complejidad computacional esperada.

1. Elección de Técnica: En esta sección del documento explicaremos de qué trata cada técnica y cómo se adecua a las reglas del sudoku, lo que permitirá evaluar cuál es más adecuado para resolver el sudoku. Evaluaran las tecnicas de Divide y Vencerás, Programación Dinámica, Algoritmos Voraces y una adicional que es Backtracking que es una técnica que usa un enfoque similar al divide y vencerás pero que se adecua más a las reglas de resolución del sudoku.

1. Divide y Vencerás

Esta técnica divide un problema en subproblemas independientes, resuelve cada subproblema de manera recursiva y combina sus soluciones para obtener el resultado final.

- **Complejidad Computacional**:
  - Depende del número de divisiones y combinaciones necesarias, pero generalmente es $O(n \log n)$ en problemas clásicos como el merge-sort.
- **Facilidad de Implementación**:
  - Requiere identificar subproblemas bien definidos y diseñar cómo combinar cada solución.
- **Adaptación al Sudoku**:
  - Por lo descrito anteriormente es poco adecuada. El Sudoku tiene restricciones globales (filas y columnas) que conectan todos los subcuadrantes, lo que hace difícil dividir el tablero en subproblemas independientes. Resolver bloques 3x3 no garantiza que el tablero completo sea válido.

2. Programación Dinámica

Este enfoque almacena soluciones parciales para reutilizarlas y evitar cálculos redundantes. Es útil cuando un problema tiene subproblemas superpuestos (como en el caso del Fibonacci). En un Sudoku, puedes calcular todos los números válidos para una celda y guardar estas opciones, entonces cuando resuelvas otra celda, actualizas las opciones almacenadas.

- **Complejidad Computacional**:
  - Depende del número de estados y de las transiciones entre ellos. Para Sudoku, $O\,(n \times m)$, donde n es el número de celdas vacías y m es el promedio de valores posibles por celda.
  - En un Sudoku difícil con 54 celdas vacías y un promedio de 4 números válidos por celda: $O\,(54 \times 4) \;=\; O(216)$.

    - Pero existe la posibilidad que no se pueda resolver el tablero de sudoku con esta técnica, por lo que el tiempo puede crecer debido a la falta de retroceso.
- **Facilidad de Implementación**:
    - Es compleja ya que requiere diseñar estructuras adicionales para almacenar los posibles valores de cada celda y actualizar dinámicamente las opciones.
- **Adaptación al Sudoku**:
    - Por lo descrito anteriormente es parcialmente adecuada, ya que puede resolver tableros simples eliminando opciones no válidas, pero al no tener posibilidad de retroceso, impide corregir errores en tableros complejos.

3. Algoritmos Voraces

Este enfoque toma decisiones inmediatas basadas en lo que parece ser la mejor opción local, sin considerar el impacto en el futuro. En Sudoku, llenar cada celda vacía con el primer número válido disponible.

- **Complejidad Computacional**:
    - Teóricamente es muy eficiente,$O(n)$, donde *n* es el número de celdas vacías.
    - Por ejemplo si tienes 54 celdas vacías, el algoritmo revisará cada una solo una vez: $O(54) = O(54)$.
- **Facilidad de Implementación**:
    - Es realmente sencillo ya que consiste en iterar por las celdas vacías y llenar con el primer número válido.
- **Adaptación al Sudoku**:
    - Es la técnica menos adecuada, ya que las decisiones locales no consideran restricciones globales, lo que hace que este enfoque falle en la mayoría de los Sudokus y más si son complejos. Así que si una decisión anterior invalida una opción futura, el algoritmo falla ya que no existe la posibilidad de retroceder y corregir.

4. Backtracking

Este enfoque explora todas las combinaciones posibles, además si una combinación no cumple con las restricciones, retrocede y prueba otra opción. En Sudoku, si colocas un número en una celda y eso bloquea el progreso, el algoritmo puede corregir esa decisión y probar con otro número. El **backtracking** comparte un enfoque similar al **divide y vencerás** en cuanto a que divide el problema en subproblemas más pequeños (resolver cada celda). Sin embargo, agrega la capacidad de retroceder y corregir decisiones, lo que lo hace ideal para problemas con restricciones dinámicas como el Sudoku.

- **Complejidad Computacional**:
    - En teoría este algoritmo prueba todas las combinaciones posibles: $O\,(\,9^{n}\,)$, donde *n* es el número de celdas vacías.
    - En la práctica, es mucho más eficiente debido a las restricciones que descartan combinaciones no válidas.

    - Por ejemplo si hay 54 celdas vacías sería $O(9^{54})$, este número es enorme, pero realmente no se revisarán todas esas posibilidades, ya que las combinaciones se descartan rápidamente debido a las restricciones.
- **Facilidad de Implementación**:
    - Tiene una dificultad moderada. ya que requiere implementar recursión y validaciones dinámicas
    - Solo se necesita verificar si un número cumple con las reglas de filas y columnas y subcuadrantes.
- **Adaptación al Sudoku**:
    - Es la más adecuada de todas las técnicas revisadas, ya que maneja restricciones globales en cada decisión, esto garantiza que encontrará la solución en caso de que exista y es capaz de corregir errores mediante retroceso.

Justificación

En resumen, el **backtracking** es la técnica más adecuada para resolver el Sudoku debido a su capacidad de manejar restricciones globales (filas, columnas y subcuadrantes) de manera integral. A diferencia de otras técnicas, como el divide y vencerás o la programación dinámica, el backtracking puede retroceder cuando una decisión incorrecta bloquea el progreso, garantizando una solución válida.

Aunque su complejidad teórica es alta $O\ (\ 9^{n}\ )$, en la práctica, el uso de restricciones reduce significativamente el espacio de búsqueda, haciéndolo eficiente incluso para Sudokus complejos. Además, su implementación es más sencilla en comparación con la programación dinámica, y a diferencia del enfoque voraz, siempre encuentra una solución válida. Entonces podemos concluir que, el backtracking es la mejor opción para resolver el Sudoku en prácticamente todos los casos.

2. Implementación:

Se implementara el algoritmo seleccionado: **Backtracking,** además de realizar diferentes pruebas para comprobar porque es la mejor opción para resolver diferentes tableros de Sudoku:

https://colab.research.google.com/drive/1-Od1FCjM0A7jBUGxJkDIyAJG3HGKP-ra?usp=sharing

Captura del tablero de Sudoku resuelto con la técnica de Backtracking y el tiempo de ejecución.

```
[11] # Tablero de Sudoku  (con celdas vacías representadas por 0)
     sudoku_board = [
         [5, 3, 0, 0, 7, 0, 0, 0, 0],
         [6, 0, 0, 1, 9, 5, 0, 0, 0],
         [0, 9, 8, 0, 0, 0, 0, 6, 0],
         [8, 0, 0, 0, 6, 0, 0, 0, 3],
         [4, 0, 0, 8, 0, 3, 0, 0, 1],
         [7, 0, 0, 0, 2, 0, 0, 0, 6],
         [0, 6, 0, 0, 0, 0, 2, 8, 0],
         [0, 0, 0, 4, 1, 9, 0, 0, 5],
         [0, 0, 0, 0, 8, 0, 0, 7, 9]
     ]
```

```
[4]  # Resolver y medir el tiempo
     start_time = time.time()
     if solve_sudoku(sudoku_board):
         print("Sudoku Resuelto:")
         print_board(sudoku_board)
     else:
         print("No tiene solución.")
     end_time = time.time()

     print(f"Tiempo de ejecución: {end_time - start_time:.4f} segundos")
```

```
Sudoku Resuelto:
5 3 4 6 7 8 9 1 2
6 7 2 1 9 5 3 4 8
1 9 8 3 4 2 5 6 7
8 5 9 7 6 1 4 2 3
4 2 6 8 5 3 7 9 1
7 1 3 9 2 4 8 5 6
9 6 1 5 3 7 2 8 4
2 8 7 4 1 9 6 3 5
3 4 5 2 8 6 1 7 9

Tiempo de ejecución: 0.1132 segundos
```

Se complementará lo desarrollado en el Colab, explicando el funcionamiento teórico de cada técnica con un ejemplo:

Evaluación de cada técnica con un ejemplo de sudoku 4x4

	0	1	2	3
0	5	3		
1			3	
2				4
3		2		

1. Algoritmo Voraz

El algoritmo voraz rellena las celdas vacías con el **primer número válido** que encuentra, sin retroceder para corregir decisiones.

Pasos:

1. En la celda (0, 2), coloca el primer número válido: 1.
2. En la celda (0, 3), coloca el siguiente número válido: 4.
3. Continua a la siguiente celda vacía (1, 0) y coloca 1.
4. Al intentar completar la celda (1, 1), no encuentra un número válido porque las decisiones previas ya bloquean todas las opciones posibles.

Resultado: El algoritmo voraz no puede avanzar porque no corrige errores pasados ni intenta probar otras combinaciones. **Falla en resolver el Sudoku.**

2. Programación Dinámica

Este enfoque almacena posibles valores para cada celda basándose en las restricciones actuales y actualiza estas posibilidades conforme se rellenan las celdas.

Pasos:

1. Calcula los posibles valores para cada celda vacía.
    - Por ejemplo, para (0, 2), los posibles valores son 1 , 4.
2. Intenta rellenar las celdas con el número que tiene el menor impacto en las opciones de las demás celdas.
    - Para (0, 2), elige 1.
3. Actualiza las posibilidades de las demás celdas.
4. Al llegar a una celda sin valores posibles (por ejemplo, (1, 1)), el algoritmo falla porque no tiene retroceso para ajustar las decisiones previas.

Resultado: Aunque más inteligente que el voraz, la programación dinámica no puede resolver tableros donde las decisiones anteriores bloquean futuras posibilidades. Falla en resolver el Sudoku.

3. Divide y Vencerás

El algoritmo divide el tablero en bloques 2x2 y los resuelve independientemente, combinando luego las soluciones. Sin embargo, las restricciones globales generan conflictos.

Pasos:

1. **Bloque 1 (0,0) - (1,1)**:
    - En la celda (1,0), coloca 4.
    - En la celda (1,1), coloca 1.
2. **Bloque 2 (0,2) - (1,3)**:
    - En la celda (0,2), coloca 4.

- En la celda (0,3), coloca 1.
- En la celda (1,3), no encuentra un número válido debido a restricciones globales.

3. **Combinación de Bloques**:
   - Al combinar, se detectan números repetidos en la columna 3 (1 y 4).
   - El tablero no puede ser resuelto debido a restricciones en filas y columnas.

Resultado: El algoritmo falla porque no puede manejar las restricciones globales.

4. Backtracking

El backtracking prueba cada número válido para una celda, avanzando recursivamente. Si no encuentra una solución, retrocede y prueba otra opción.

Pasos:

1. En la celda (0, 2), coloca el primer número válido: 1.
2. En la celda (0, 3), coloca el siguiente número válido: 4.
3. En la celda (1, 0), coloca el primer número válido: 1.
4. En la celda (1, 1), no hay números válidos debido a las decisiones anteriores.
5. **Retrocede** a la celda (1, 0) y prueba el siguiente número: 2.
6. Actualiza las celdas restantes con números válidos.
7. Continúa el proceso hasta completar el tablero.

Resultado: El backtracking ajusta sus decisiones conforme avanza, resolviendo completamente el Sudoku.

Sudoku resuelto:

	0	1	2	3
0	5	3	1	4
1	4	2	3	1
2	1	4	2	4
3	3	2	4	5

El backtracking sobresale porque evalúa todas las combinaciones posibles y corrige decisiones incorrectas retrocediendo.

3. Evaluación y Análisis de Complejidad

1. Complejidad Temporal

El algoritmo de **backtracking** explora todas las combinaciones posibles para llenar el tablero. En el peor caso, intenta colocar hasta 9 números diferentes en cada celda vacía. Si hay *n* celdas vacías, la complejidad temporal se expresa como: $O(9^n)$.

- **Peor caso**:
  - Cuando no hay pistas iniciales, el algoritmo realiza una búsqueda exhaustiva, probando todas las combinaciones posibles.
- **Casos prácticos**:
  - Debido a las restricciones del Sudoku (filas, columnas y subcuadrantes), muchas combinaciones se descartan rápidamente. Esto reduce significativamente el tiempo real de ejecución en tableros con pistas iniciales bien distribuidas.

2. Complejidad Espacial

El backtracking utiliza memoria adicional para:

- Almacenar el estado actual del tablero.
- Manejar las llamadas recursivas.

La profundidad de la recursión está limitada por el número de celdas vacías (n), lo que resulta en una complejidad espacial de: $O(n)$.

- En tableros típicos con $n \leq 81$ (Sudoku estándar), la complejidad espacial del algoritmo de backtracking es manejable. Además, en la práctica, los tableros siempre tienen pistas iniciales que imponen restricciones, lo que reduce significativamente el espacio de búsqueda. Esto evita que se alcance el peor caso teórico, en el cual todas las 81 celdas estarían vacías y requerirían un análisis exhaustivo.

Resultado del algoritmo:

```
[11] # Tablero de Sudoku  (con celdas vacías representadas por 0)
     sudoku_board = [
         [5, 3, 0, 0, 7, 0, 0, 0, 0],
         [6, 0, 0, 1, 9, 5, 0, 0, 0],
         [0, 9, 8, 0, 0, 0, 0, 6, 0],
         [8, 0, 0, 0, 6, 0, 0, 0, 3],
         [4, 0, 0, 8, 0, 3, 0, 0, 1],
         [7, 0, 0, 0, 2, 0, 0, 0, 6],
         [0, 6, 0, 0, 0, 0, 2, 8, 0],
         [0, 0, 0, 4, 1, 9, 0, 0, 5],
         [0, 0, 0, 0, 8, 0, 0, 7, 9]
     ]
```

```
Sudoku Resuelto:
5 3 4 6 7 8 9 1 2
6 7 2 1 9 5 3 4 8
1 9 8 3 4 2 5 6 7
8 5 9 7 6 1 4 2 3
4 2 6 8 5 3 7 9 1
7 1 3 9 2 4 8 5 6
9 6 1 5 3 7 2 8 4
2 8 7 4 1 9 6 3 5
3 4 5 2 8 6 1 7 9

Tiempo de ejecución: 0.1132 segundos
```

Sudoku Resuelto:

El tablero resuelto cumple con todas las restricciones del Sudoku:

- Cada fila contiene los números del 1 al 9 sin repetirse.
- Cada columna contiene los números del 1 al 9 sin repetirse.
- Cada subcuadrante 3x3 contiene los números del 1 al 9 sin repetirse.

Tiempo de Ejecución:

- El tiempo de ejecución de **0.1132 segundos** es consistente con el rendimiento esperado del backtracking para un tablero estándar.
- Factores como la cantidad de pistas iniciales y la distribución de estas pueden influir en el tiempo final.

Comparación con Complejidad Teórica:

- Aunque el algoritmo tiene una complejidad teórica de $O(9^n)$, en la práctica, el tiempo observado es mucho menor debido a que muchas combinaciones se descartan rápidamente gracias a las restricciones.

Como conclusión, podemos afirmar que, aunque el algoritmo de **backtracking** tiene una complejidad teórica elevada, en la práctica resulta bastante eficiente gracias a su capacidad para aprovechar las propias reglas del Sudoku. Las celdas previamente llenadas actúan como restricciones que reducen significativamente el espacio de búsqueda, lo que permite un desempeño óptimo en la mayoría de los casos.

Como hemos observado a lo largo del reporte, esto convierte al backtracking en una opción confiable para resolver tableros de diferentes niveles de dificultad, destacando su efectividad en tableros de dificultad baja y media, donde las restricciones iniciales son suficientes para evitar un espacio de búsqueda excesivamente amplio.