



Reporte de Divide y Vencerás

TECNICAS ALGORITMICAS

JOSE ALEXANDER HERNANDEZ HERNANDEZ
230300992

11/26/2024

Emmanuel Morales Saavedra

```
# revisa fila
for i in range(9):
    if board[row][i] == num:
        return False
```

Recorre la fila en la posición row (posición de la fila) del tablero. Tiene como fin comprobar si el número num ya está presente en la fila. Si lo encuentra, retorna False (no es válido colocar el número en esa celda).

```
# revisa columna
for i in range(9):
    if board[i][col] == num:
        return False
```

Recorre la columna en la posición col (posición de la columna) del tablero. De igual manera comprueba si el número num ya está presente en la columna. Si hay, retorna False.

```
start_row, start_col = 3 * (row // 3), 3 * (col // 3)
for i in range(3):
    for j in range(3):
        if board[start_row + i][start_col + j] == num:
            return False
```

Calcula la esquina superior izquierda del subcuadro 3x3 que contiene la celda (row, col):

start_row = 3 * (row // 3): Encuentra la fila inicial del subcuadro.

start_col = 3 * (col // 3): Encuentra la columna inicial del subcuadro.

Recorre las 9 celdas del subcuadro 3x3 usando dos bucles anidados (i y j).

Verifica si el número num ya está en alguna de esas celdas.

Propósito: Garantiza que num no se repita dentro del subcuadro 3x3 al que pertenece la celda (row, col). Si lo encuentra, retorna False.

```
def is_valid(board, row, col, num):    Trailing whitespace
    """Verifica si un número es válido en la posición dada."""
    # revisa fila
    for i in range(9):
        if board[row][i] == num:
            return False
    # revisa columna
    for i in range(9):
        if board[i][col] == num:
            return False
    # revisa el subcuadro 3x3
    start_row, start_col = 3 * (row // 3), 3 * (col // 3)
    for i in range(3):
        for j in range(3):
            if board[start_row + i][start_col + j] == num:
                return False
    return True
```

```
def solve_sudoku(board, row=0, col=0):
    #ROW(POSICION DE LA FILA)
    #COL(COLUMNA DE LA FILA)
    """
    Resolicon del Sudoku usando la tecnica de Divide y Vencerás.
    row, col: Indican la celda actual a resolver.
    """
```

Se define la función para resolver el Sudoku.

board: Es el tablero de Sudoku, una lista de listas de 9x9.

```
# ultima celda, el tablero está completo
if row == 9: Trailing whitespace
    return True

# Continuar a la siguiente fila si llega al final de una columna
if col == 9: Trailing whitespace
    return solve_sudoku(board, row + 1, 0)

# Si ya está completa la celda, sigue a la otra celda que continúa
if board[row][col] != 0: Trailing whitespace
    return solve_sudoku(board, row, col + 1)
```

1. Verifica si se alcanzó el final del tablero. El índice row indica la fila actual, y el tablero tiene 9 filas (índices de 0 a 8).
2. Sí row == 9, significa que el algoritmo ya revisó todas las celdas del tablero, por lo que devuelve True, indicando que el Sudoku está completo y resuelto. En este caso:
 - Incrementa row en 1 para pasar a la siguiente fila.
 - Reinicia col a 0 para comenzar en la primera columna de la nueva fila.
 - Llama recursivamente a solve_sudoku para continuar resolviendo.
 - Saltar celdas ya llenas. Verifica si la celda actual (board[row][col]) ya contiene un número (es decir, no es 0).
 - Si la celda no está vacía:
 - No intenta resolverla.
 - Avanza a la siguiente celda de la misma fila incrementando col en 1.
 - Llama recursivamente a solve_sudoku para resolver desde la nueva posición.

```

# Prueba con números del 1 al 9
for num in range(1, 10):
    if is_valid(board, row, col, num):
        board[row][col] = num # Coloca el número provisionalmente
        print(f"Colocando {num} en la posición ({row}, {col}):") Trailing whitespace
        print_board(board)

        # Resuelve la celda siguiente
        if solve_sudoku(board, row, col + 1):
            return True

        # Retrocede si no es válido
        board[row][col] = 0
        print(f"Retrocediendo en la posición ({row}, {col}):") Trailing whitespace
        print_board(board)

return False # No se pudo resolver (no hay solución)

```



1. El ciclo recorre los números del 1 al 9 para intentar colocar un número válido en la celda actual.
2. Si el número es válido, se coloca en la celda y se intenta resolver la siguiente celda.
3. Si una celda no lleva a una solución, se retrocede eliminando el número y probando con otro.
4. Si todas las celdas se completan correctamente, la función devuelve True y el Sudoku está resuelto.
5. Si no se puede completar el tablero, el algoritmo devuelve False y retrocede hasta encontrar una solución alternativa.

```

def print_board(board):
    """Imprime el tablero de Sudoku."""
    for i in range(9):
        if i % 3 == 0 and i != 0:
            print("-" * 21)
        for j in range(9):
            if j % 3 == 0 and j != 0:
                print("| ", end="")
            print(board[i][j] if board[i][j] != 0 else ".", end=" ")
        print()

```

1. La función recorre las filas (i) y las columnas (j) del tablero de Sudoku.
2. Cada vez que se llega a una posición de columna que es múltiplo de 3, se imprime una barra (|) para separar los subcuadros de 3x3.
3. Imprime el número de cada celda o un punto (".") si la celda está vacía (es decir, tiene un valor de 0).
4. Después de cada fila, se imprime una línea separadora después de cada 3 filas, para mejorar la visualización del tablero.
5. El resultado final es un tablero de Sudoku bien formateado que se muestra de forma legible.

EJEMPLO DE SALIDA

```
# Tablero (0 = celdas vacías)
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]
]
```

De esta manera quedaría la tabla al ejecutarlo

5	3	.		.	7
6	.	.		1	9	5		.	.	.
.	9	8		6	.

8	.	.		8
4	.	.		4	1	9		.	.	5
7	3	.

.	6		2	8	.
.	.	.		8	.	.		.	7	9
.	.	.		.	6

```
print("Tablero inicial:")
print_board(sudoku_board)

if solve_sudoku(sudoku_board):
    print("\n¡Sudoku resuelto!")
    print_board(sudoku_board)
else:
    print("\nNo se encontró solución para el Sudoku.")
```

1. Se imprime el tablero de Sudoku inicial.
2. Se intenta resolver el tablero llamando a solve_sudoku.
 - Si el Sudoku se resuelve correctamente, se muestra el mensaje "¡Sudoku resuelto!" y el tablero resuelto.
 - Si no se puede resolver, se muestra el mensaje "No se encontró solución para el Sudoku."

EL SUDOKU RESUELTO

TIEMPO 7:11 SEGUNDOS

```
def is_valid(board, row, col, num):
    for i in range(3):
        for j in range(3):
            if board[start_row + i][start_col + j] == num:
                return False
    return True

def solve_sudoku(board, row=0, col=0):
    """
    Resolución del Sudoku usando la técnica de Divide y Vencés.
    row, col: Indican la celda actual a resolver.
    """
    # Última celda, el tablero está completo
    if row == 9:
        return True

    for num in range(1, 10):
        if is_valid(board, row, col, num):
            board[row][col] = num
            if solve_sudoku(board, row, col + 1):
                return True
            board[row][col] = 0

    return False

# Ejemplo de tablero de Sudoku
board = [
    [2, 8, 7, 4, 1, 9, 6, 3, 5],
    [3, 4, 5, 2, 8, 6, 1, 7, 9],
    [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]
]

# Resolución del Sudoku
if solve_sudoku(board):
    print("Sudoku resuelto!")
    for row in board:
        print(row)
else:
    print("No se encontró solución para el Sudoku.")
```

2 8 7 | 4 1 9 | 6 3 5
3 4 5 | 2 8 6 | 1 7 9
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

[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 tomado: 7.11 segundos
PS C:\Users\herna\OneDrive\Desktop\SUDOKU>

SIN SOLUCIÓN

```
def print_board(board):
    print()

# Tablero (0 = celdas vacías)
sudoku_board = [
    [0, 2, 0, 0, 7, 0, 0, 0, 0],
    [6, 0, 0, 1, 0, 5, 0, 0, 0],
    [0, 9, 8, 0, 0, 0, 0, 0, 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, 0, 0, 0],
    [0, 0, 4, 1, 0, 0, 0, 5],
    [0, 0, 0, 0, 8, 0, 0, 7, 9]
]

# Resolución del Sudoku
if solve_sudoku(sudoku_board):
    print("Sudoku resuelto!")
    for row in sudoku_board:
        print(row)
else:
    print("No se encontró solución para el Sudoku.")
```

. 6 . | . . . | 2 8 .
. . . | 4 1 9 | . 5
. . . | 8 . 1 | 7 9
Retrocediendo en la posición (0, 3):
4 2 5 | . 7 . | . . .
6 . . | 1 9 5 | . . .
. 9 8 | . . . | . 6 .

8 . . | . 6 . | . . 3
4 . . | 8 . 3 | . . 1
7 . . | . 2 . | . . 6

. 6 . | . . . | 2 8 .
. . . | 4 1 9 | . 5
. . . | 8 . 1 | 7 9
Retrocediendo en la posición (0, 2):
4 2 . | . 7 . | . . .
6 . . | 1 9 5 | . . .
. 9 8 | . . . | . 6 .

8 . . | . 6 . | . . 3
4 . . | 8 . 3 | . . 1
7 . . | . 2 . | . . 6

. 6 . | . . . | 2 8 .
. . . | 4 1 9 | . 5
. . . | 8 . 1 | 7 9
No se encontró solución para el Sudoku.
PS C:\Users\herna\OneDrive\Desktop\SUDOKU>

CÓDIGO DE DIVIDE Y VENCERÁS

```
def is_valid(board, row, col, num):  
  
    """Verifica si un número es válido en la posición dada."""  
  
    # revisa fila  
  
    for i in range(9):  
  
        if board[row][i] == num:  
  
            return False  
  
    # revisa columna  
  
    for i in range(9):  
  
        if board[i][col] == num:  
  
            return False  
  
    # revisa el subcuadro 3x3  
  
    start_row, start_col = 3 * (row // 3), 3 * (col // 3)  
  
    for i in range(3):  
  
        for j in range(3):  
  
            if board[start_row + i][start_col + j] == num:  
  
                return False  
  
    return True  
  
def solve_sudoku(board, row=0, col=0):  
  
    #ROW(POSICION DE LA FILA)  
  
    #COL(COLUMNA DE LA FILA)  
  
    """  
  
    Resolicon del Sudoku usando la tecnica de Divide y Vencerás.  
  
    row, col: Indican la celda actual a resolver.  
  
    """  
  
    # ultima celda, el tablero está completo
```



```

if row == 9:

    return True

# Continuar a la siguiente fila si llega al final de una columna

if col == 9:

    return solve_sudoku(board, row + 1, 0)

# Si ya está completa la celda, sigue a la otra celda que continúa

if board[row][col] != 0:

    return solve_sudoku(board, row, col + 1)

# Prueba con números del 1 al 9

for num in range(1, 10):

    if is_valid(board, row, col, num):

        board[row][col] = num # Coloca el número provisionalmente

        print(f"Colocando {num} en la posición ({row}, {col}):")

        print_board(board)

        # Resuelve la celda siguiente

        if solve_sudoku(board, row, col + 1):

            return True

        # Retrocede si no es válido

        board[row][col] = 0

        print(f"Retrocediendo en la posición ({row}, {col}):")

        print_board(board)

```

```

        return False # No se pudo resolver (no hay solución)

def print_board(board):

    """Imprime el tablero de Sudoku."""

    for i in range(9):

        if i % 3 == 0 and i != 0:

            print("-" * 21)

        for j in range(9):

            if j % 3 == 0 and j != 0:

                print("| ", end="")

            print(board[i][j] if board[i][j] != 0 else ".", end=" ")

        print()

# Tablero (0 = celdas vacías)

sudoku_board = [

    [4, 2, 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]

]

print("Tablero inicial:")

```

```
print_board(sudoku_board)

if solve_sudoku(sudoku_board):
    print("\n¡Sudoku resuelto!")
    print_board(sudoku_board)
else:
    print("\nNo se encontró solución para el Sudoku.")
```

PROGRAMACIÓN DINÁMICA

```
def is_valid(board, row, col, num):  
    """Verifica si 'num' puede colocarse en la celda dada."""  
    # Verifica la fila y la columna  
    for i in range(9):  
        if board[row][i] == num or board[i][col] == num:  
            return False
```

verifica si num ya está presente en el subcuadro de 3x3 que incluye la celda (row, col). Para localizar el subcuadro correspondiente, calcula las coordenadas de la esquina superior izquierda (start_row y start_col):

```
start_row, start_col = 3 * (row // 3), 3 * (col // 3)  
for i in range(3):  
    for j in range(3):  
        if board[start_row + i][start_col + j] == num:  
            return False
```

Si ninguna de estas condiciones se cumple, el número es válido en esa posición.

```
def find_empty_cell(board):  
    """Encuentra una celda vacía en el tablero."""  
    for row in range(9):  
        for col in range(9):  
            if board[row][col] == 0:  
                return row, col  
    return None
```

Encuentra una celda vacía (marcada con 0) en el tablero. Devuelve las coordenadas de la primera celda vacía encontrada o None si no quedan celdas vacías.

```
empty_cell = find_empty_cell(board)  
if not empty_cell:  
    return True # Tablero resuelto  
row, col = empty_cell
```

Busca una celda vacía utilizando find_empty_cell. Si no hay celdas vacías, el tablero está resuelto.

```
for num in range(1, 10):  
    if is_valid(board, row, col, num):  
        board[row][col] = num # Asigna un número provisional  
  
        # Llama recursivamente para resolver el resto del tablero  
        if solve_sudoku(board):  
            return True  
  
        # Retrocede si no funciona  
        board[row][col] = 0  
return False
```

Prueba los números del 1 al 9 en la celda vacía. Por cada número:

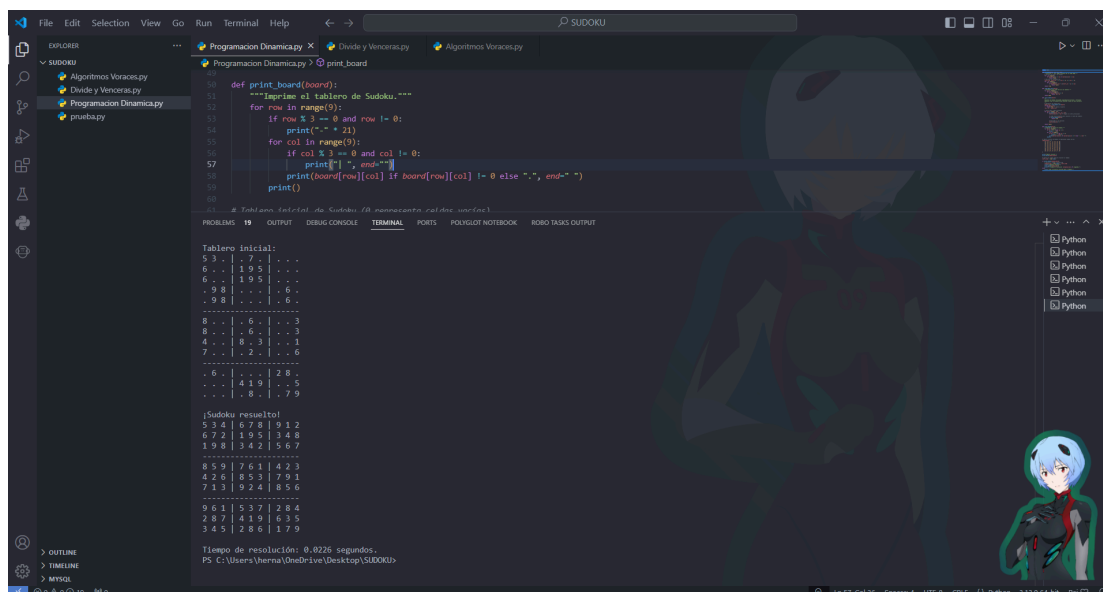
- Verifica si es válido con `is_valid`.
- Si es válido, lo coloca provisionalmente en la celda.
- Llama recursivamente a `solve_sudoku` para resolver el resto del tablero.
- Si la solución recursiva falla, elimina el número (retroceso) y prueba el siguiente.

Si ningún número es válido, retorna `False` para retroceder al paso anterior.

```
def print_board(board):
    """Imprime el tablero de Sudoku."""
    for row in range(9):
        if row % 3 == 0 and row != 0:
            print("-" * 21)
        for col in range(9):
            if col % 3 == 0 and col != 0:
                print("|", end="")
            print(board[row][col] if board[row][col] != 0 else ".", end=" ")
        print()
```

Imprime el tablero de forma legible, con líneas horizontales y verticales que delimitan los subcuadros 3x3

TIEMPO DE SOLUCIÓN: 0.0226 SEGUNDOS



```
File Edit Selection View Go Run Terminal Help
SUDOKU
Programacion Dinamica.py X Divide y Venceras.py Algoritmos Voraces.py
def print_board(board):
    """Imprime el tablero de Sudoku."""
    for row in range(9):
        if row % 3 == 0 and row != 0:
            print("-" * 21)
        for col in range(9):
            if col % 3 == 0 and col != 0:
                print("|", end="")
            print(board[row][col] if board[row][col] != 0 else ".", end=" ")
        print()

Tablero inicial:
5 3 . | . 7 . | . . .
6 . . | 1 9 5 | . . .
6 . . | 1 9 5 | . . .
. 9 8 | . . . | . 6 .
. 9 8 | . . . | . 6 .
-----
8 . . | . 6 . | . . 3
8 . . | . 6 . | . . 3
4 . . | 8 . 3 | . . 1
7 . . | . 2 . | . . 6
-----
. 6 . | . . . | 2 8 .
. . . | 4 1 9 | . . 5
. . . | . 8 . | . . 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 5 5
3 4 5 | 2 8 6 | 1 7 9

Tiempo de resolución: 0.0226 segundos.
PS C:\Users\hernan\OneDrive\Desktop\SUDOKU>
```

ALGORITMOS VORACES

Primero, verifica si el número ya existe en la misma fila o columna:

```
def is_valid(board, row, col, num):    Missing module docstring
    """Verifica si un número puede colocarse en la posición dada."""
    # Verifica la fila
    for i in range(9):
        if board[row][i] == num:
            return False
    # Verifica la columna
    for i in range(9):
        if board[i][col] == num:
            return False
```

Después, verifica el subcuadro 3x3 correspondiente a la celda. Usa `row // 3` y `col // 3` para localizar la esquina superior izquierda del subcuadro:

```
start_row, start_col = 3 * (row // 3), 3 * (col // 3)
for i in range(3):
    for j in range(3):
        if board[start_row + i][start_col + j] == num:
            return False
return True
```

Si el número no está presente en la fila, columna o subcuadro, regresa `True`.

Comienza con todos los números del 1 al 9:

```
candidates = set(range(1, 10))
```

Elimina los números que ya aparecen en la fila y la columna:

```
def find_candidates(board, row, col):  
    """Encuentra los posibles números válidos para una celda."""  
    candidates = set(range(1, 10))  
    for i in range(9):  
        if board[row][i] in candidates:  
            candidates.remove(board[row][i])  
        if board[i][col] in candidates:  
            candidates.remove(board[i][col])
```

También elimina los números que ya están en el subcuadro 3x3 correspondiente:

```
    start_row, start_col = 3 * (row // 3), 3 * (col // 3)  
    for i in range(3):  
        for j in range(3):  
            if board[start_row + i][start_col + j] in candidates:  
                candidates.remove(board[start_row + i][start_col + j])  
    return list(candidates)
```

Retorna los números válidos como una lista.

Recorre todas las celdas del tablero. Para cada celda vacía (con valor 0), calcula sus candidatos con `find_candidates`:

```
def find_least_candidates_cell(board):  
    """Encuentra la celda vacía con el menor número de candidatos."""  
    min_candidates = float('inf')  
    best_cell = None  
    for row in range(9):  
        for col in range(9):  
            if board[row][col] == 0:  
                candidates = find_candidates(board, row, col)
```

Si encuentra una celda con menos candidatos que las evaluadas anteriormente, la guarda como la mejor opción:

```
if len(candidates) < min_candidates:  
    min_candidates = len(candidates)  
    best_cell = (row, col)
```

Devuelve la celda con menos candidatos o `None` si no quedan celdas vacías.


```
def solve_sudoku_greedy(board):
    """
    Resuelve el Sudoku usando un enfoque voraz.
    Encuentra la celda con la menor cantidad de candidatos y asigna el primero válido.
    """
    cell = find_least_candidates_cell(board)
    if not cell:
        return True # Tablero resuelto
    row, col = cell

    candidates = find_candidates(board, row, col)
    for num in candidates:
        if is_valid(board, row, col, num):
            board[row][col] = num # Asigna provisionalmente
            if solve_sudoku_greedy(board):
                return True
            board[row][col] = 0 # Retrocede si es necesario

    return False # No se encontró solución
```

1. Encuentra la celda más restrictiva usando `find_least_candidates_cell`. Si no hay celdas vacías, el tablero está resuelto
2. Obtiene los candidatos válidos para esa celda con `find_candidates` y prueba cada número
3. Si el número es válido (`is_valid`), lo coloca provisionalmente en la celda y llama recursivamente a `solve_sudoku_greedy`
4. Si el intento falla (retroceso), vacía la celda (`board[row][col] = 0`) y prueba el siguiente candidato.
5. Si ninguno de los candidatos funciona, retorna `False` para retroceder.

```
def print_board(board):
    """Imprime el tablero de Sudoku."""
    for i in range(9):
        if i % 3 == 0 and i != 0:
            print("-" * 21)
        for j in range(9):
            if j % 3 == 0 and j != 0:
                print("| ", end="")
            print(board[i][j] if board[i][j] != 0 else ".", end=" ")
        print()

# Tablero inicial (0 representa celdas vacías)
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]
]

print("Tablero inicial:")
print_board(sudoku_board)

if solve_sudoku_greedy(sudoku_board):
    print("\n¡Sudoku resuelto!")
    print_board(sudoku_board)
else:
    print("\nNo se encontró solución para el Sudoku.")
```

IMPRIME EL TABLERO JUNTO CON LOS MENSAJES DE SUDOKU RESUELTO Y NO SE ENCONTRO SOLUCION PARA EL SUDOKU.

TIEMPO DE SOLUCIÓN: 6:98 SEGUNDOS

```
def solve_sudoku(board, row=0, col=0):
    if col == 9:
        return solve_sudoku(board, row + 1, 0)
    trailing_whitespace = len(board[row].rstrip())
    # Si ya está completa la celda, sigue a la otra celda que continúa
    if board[row][col] != 0:
        return solve_sudoku(board, row, col + 1)
    trailing_whitespace = len(board[row].rstrip())
    # Prueba con números del 1 al 9
    for num in range(1, 10):
        if is_valid(board, row, col, num):
            board[row][col] = num # Coloca el número provisionalmente
            print(f"Colocando {num} en la posición ({row}, {col}):")
            print_board(board)
            trailing_whitespace = len(board[row].rstrip())
            # Si ya está completa la celda, sigue a la otra celda que continúa
            if col == 9:
                return solve_sudoku(board, row + 1, 0)
            return solve_sudoku(board, row, col + 1)
    # Si no se encuentra solución, devuelve None
    return None

def is_valid(board, row, col, num):
    # Verificar fila
    for i in range(9):
        if i != row and board[i][col] == num:
            return False
    # Verificar columna
    for i in range(9):
        if i != col and board[row][i] == num:
            return False
    # Verificar subcuadrante 3x3
    row_start = (row // 3) * 3
    col_start = (col // 3) * 3
    for i in range(3):
        for j in range(3):
            if (i + row_start != row and j + col_start != col and board[i + row_start][j + col_start] == num):
                return False
    return True

def print_board(board):
    """Imprime el tablero de Sudoku."""
    for i in range(9):
        if i % 3 == 0 and i != 0:
            print("-" * 21)
        for j in range(9):
            if j % 3 == 0 and j != 0:
                print("| ", end="")
            print(board[i][j] if board[i][j] != 0 else ".", end=" ")
        print()

# Tablero inicial (0 representa celdas vacías)
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]
]

print("Tablero inicial:")
print_board(sudoku_board)

if solve_sudoku_greedy(sudoku_board):
    print("\n¡Sudoku resuelto!")
    print_board(sudoku_board)
else:
    print("\nNo se encontró solución para el Sudoku.")
```

2 8 7 | 4 1 9 | 6 3 5
3 4 5 | 2 8 6 | . 7 9
Colocando 1 en la posición (8, 6):
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

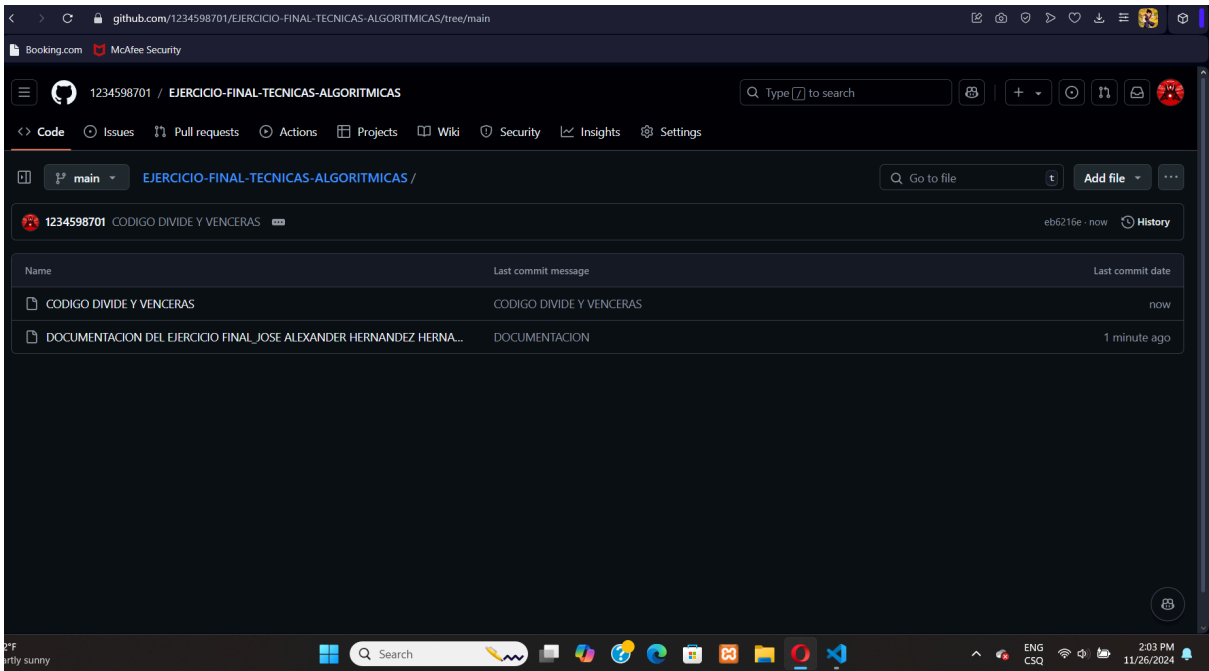
¡Sudoku resuelto!
5 3 4 | 6 7 8 | 9 1 2
6 7 2 | 1 9 5 | 3 4 8
1 9 8 | 3 4 2 | 5 6 7

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 tomado: 6.98 segundos
PS C:\Users\herma\OneDrive\Desktop\SUDOKU>

GITHUB EVIDENCIA



La solución del Sudoku utilizando el algoritmo divide y vencerás es generalmente más eficiente y mejor estructurada en comparación con los otros dos enfoques (búsqueda voraz y retroceso sin optimización) por varias razones clave:

1. Uso de Heurísticas Inteligentes

En el algoritmo de divide y vencerás, se priorizan las celdas con la menor cantidad de candidatos posibles (heurística "Minimum Remaining Values" o MRV). Esto significa que el algoritmo:

- Reduce rápidamente las posibilidades al enfocarse en las celdas más restringidas.
- Minimiza el número de intentos y retrocesos necesarios, pues las celdas más difíciles se resuelven primero.
- Por qué es mejor: Los otros dos enfoques intentan resolver el Sudoku siguiendo un orden arbitrario (por ejemplo, recorriendo las celdas fila por fila o columna por columna). Esto puede llevar a asignaciones que resultan inválidas más adelante, aumentando los retrocesos.

2. Reducción del Espacio de Búsqueda

El algoritmo de divide y vencerás descompone el problema en subproblemas más pequeños y maneja cada subproblema de forma independiente, enfocándose en celdas específicas en lugar de intentar resolver todo el tablero de una vez.

Por qué es mejor: Este enfoque divide la complejidad del tablero completo en pasos manejables, lo que optimiza el tiempo de ejecución al evitar exploraciones innecesarias.

3. Estrategia Basada en Candidatos

El algoritmo encuentra candidatos válidos para cada celda antes de hacer las asignaciones. De esta manera:

- Solo prueba números que tienen posibilidades reales de ser válidos.
- Reduce significativamente el número de intentos necesarios.
- Por qué es mejor: Los otros enfoques, especialmente el retroceso básico, prueban todos los números posibles (del 1 al 9) en una celda sin considerar cuán restringido es el espacio de búsqueda. Esto lleva a un mayor número de asignaciones y verificaciones inútiles.

4. Orden de Exploración Óptimo

El enfoque divide y vencerás se basa en encontrar las celdas con las restricciones más estrictas (menos candidatos). Esto:

- Maximiza la eficiencia del algoritmo al garantizar que las decisiones más importantes se tomen primero.
- Evita asignaciones erróneas que podrían invalidar grandes porciones del tablero.
- Por qué es mejor: En comparación, los enfoques lineales procesan celdas en un orden fijo, lo que puede resultar en decisiones tempranas que necesitan ser revertidas más adelante.

5. Retroceso Optimizado

Aunque los tres algoritmos usan retroceso como técnica base, en el enfoque divide y vencerás:

- El retroceso ocurre con menos frecuencia porque el espacio de búsqueda ya está reducido mediante heurísticas inteligentes.
- Las decisiones incorrectas se detectan más rápido.
- Por qué es mejor: En el retroceso básico, se desperdicia mucho tiempo verificando combinaciones que no tienen sentido debido a la falta de optimización inicial.

Con esta pequeña explicación de porque el algoritmo de divide y vencerás es más eficiente en resumen porque:

- Hace un uso inteligente de heurísticas.
- Divide el problema de manera estructurada.
- Reduce el número de combinaciones probadas.
- Optimiza la búsqueda mediante una estrategia de candidatos válidos.

En cambio, los otros dos algoritmos, aunque funcionales, son menos eficientes porque:

- No utilizan heurísticas avanzadas para reducir el espacio de búsqueda.
- Pueden realizar muchas operaciones redundantes debido al orden arbitrario de exploración.
- El enfoque divide y vencerás no solo encuentra soluciones más rápido, sino que también es más robusto para tableros más grandes o con configuraciones iniciales más complejas.