ASSIGNMENT 3 –

DIVIJA L

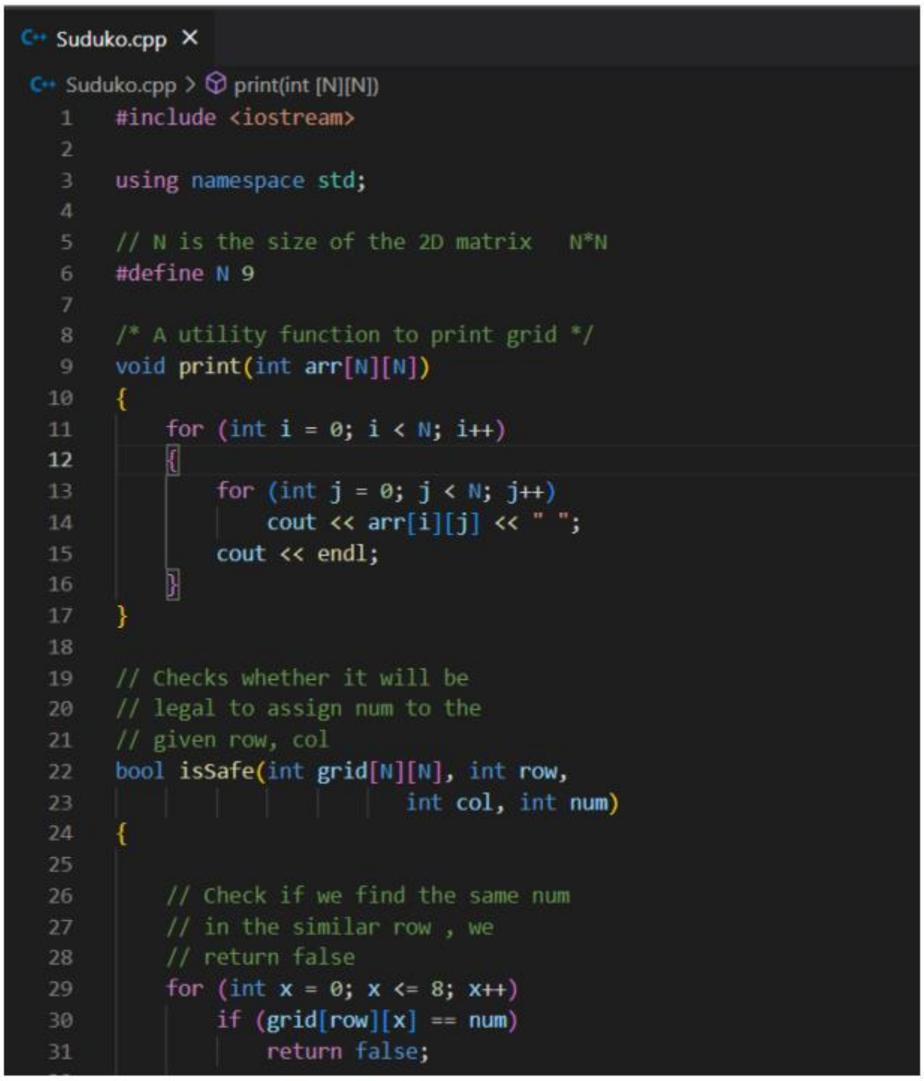
PES1UG20CS134

SEN 5 - SEC C

A. Find the solution to 9X9 sudoku problem(with/without initial hints)

Backtracking

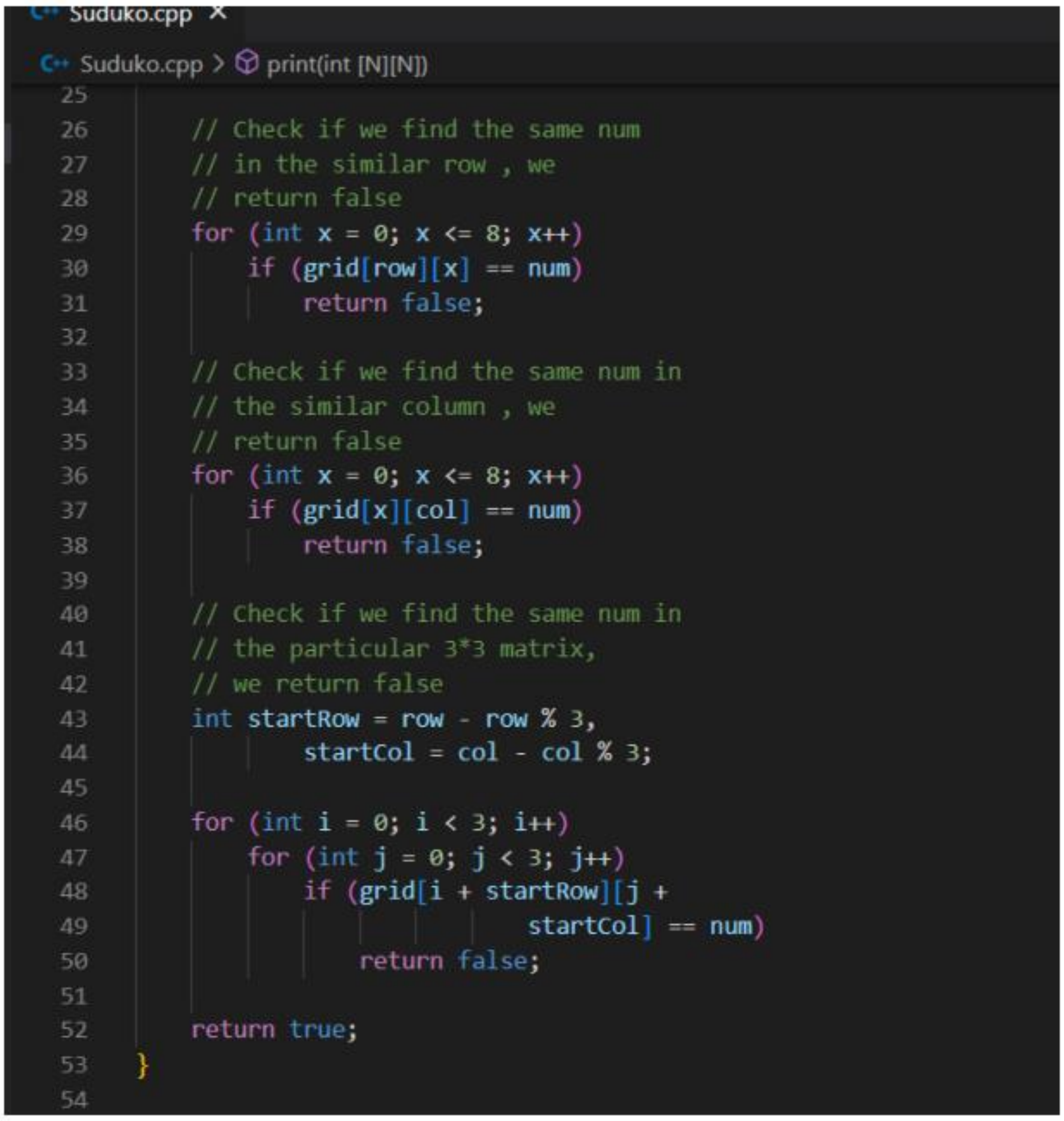
Backtracking is a kind of brute-force approach which comes into picture when solving a problem requires considering multiple choices as we don't know which choice is correct and we try to solve the problem using trail and error method considering one choice at a time until required answer is obtained.



Suduko.cpp ×

Suduko.cpp > print(int [N][N])

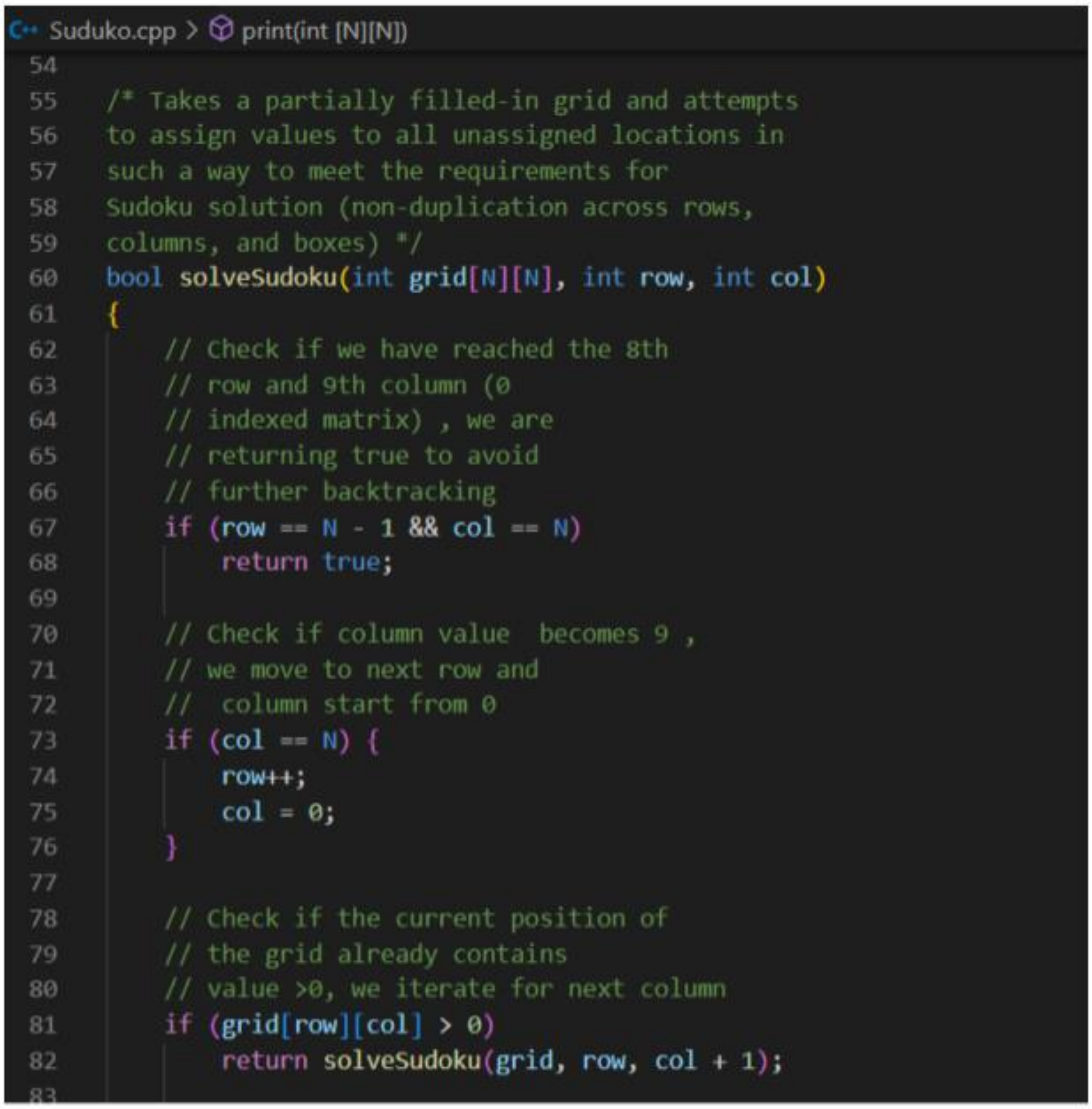
```cpp
#include <iostream>

using namespace std;

// N is the size of the 2D matrix   N*N
#define N 9

/* A utility function to print grid */
void print(int arr[N][N])
{
    for (int i = 0; i < N; i++)
    {
        for (int j = 0; j < N; j++)
            cout << arr[i][j] << " ";
        cout << endl;
    }
}

// Checks whether it will be
// legal to assign num to the
// given row, col
bool isSafe(int grid[N][N], int row,
                        int col, int num)
{

    // Check if we find the same num
    // in the similar row , we
    // return false
    for (int x = 0; x <= 8; x++)
        if (grid[row][x] == num)
            return false;
```



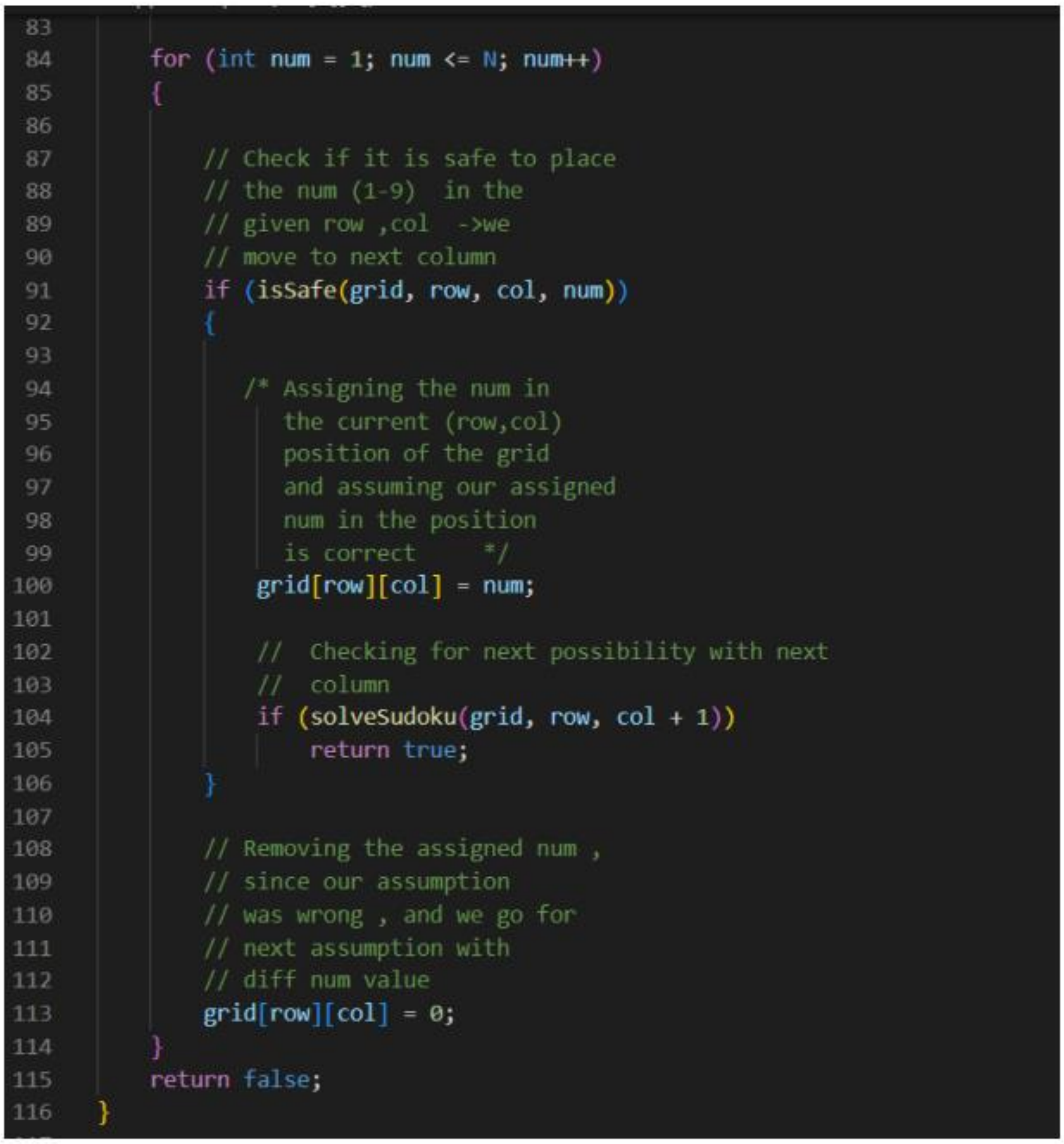
Suduko.cpp ×

Suduko.cpp > print(int [N][N])

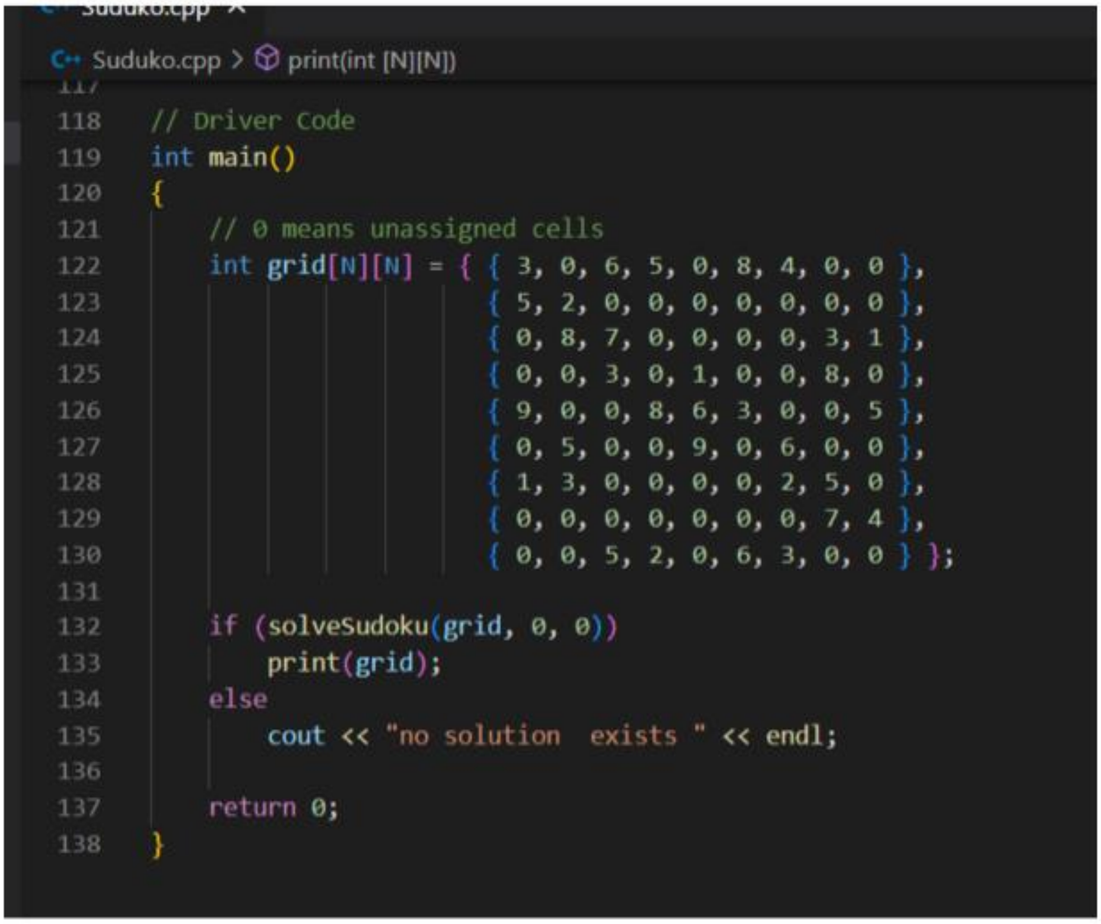
```cpp

    // Check if we find the same num
    // in the similar row , we
    // return false
    for (int x = 0; x <= 8; x++)
        if (grid[row][x] == num)
            return false;

    // Check if we find the same num in
    // the similar column , we
    // return false
    for (int x = 0; x <= 8; x++)
        if (grid[x][col] == num)
            return false;

    // Check if we find the same num in
    // the particular 3*3 matrix,
    // we return false
    int startRow = row - row % 3,
            startCol = col - col % 3;

    for (int i = 0; i < 3; i++)
        for (int j = 0; j < 3; j++)
            if (grid[i + startRow][j +
                            startCol] == num)
                return false;

    return true;
}

```


C++ Suduko.cpp > print(int [N][N])

```

/* Takes a partially filled-in grid and attempts
to assign values to all unassigned locations in
such a way to meet the requirements for
Sudoku solution (non-duplication across rows,
columns, and boxes) */
bool solveSudoku(int grid[N][N], int row, int col)
{
    // Check if we have reached the 8th
    // row and 9th column (0
    // indexed matrix) , we are
    // returning true to avoid
    // further backtracking
    if (row == N - 1 && col == N)
        return true;

    // Check if column value  becomes 9 ,
    // we move to next row and
    //  column start from 0
    if (col == N) {
        row++;
        col = 0;
    }

    // Check if the current position of
    // the grid already contains
    // value >0, we iterate for next column
    if (grid[row][col] > 0)
        return solveSudoku(grid, row, col + 1);

```

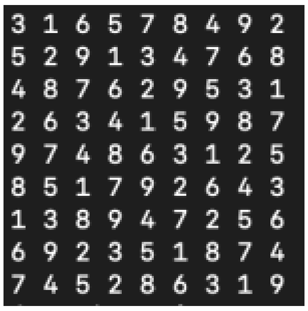


```
    for (int num = 1; num <= N; num++)
    {

        // Check if it is safe to place
        // the num (1-9)  in the
        // given row ,col  ->we
        // move to next column
        if (isSafe(grid, row, col, num))
        {

          /* Assigning the num in
             the current (row,col)
             position of the grid
             and assuming our assigned
             num in the position
             is correct     */
           grid[row][col] = num;

           //  Checking for next possibility with next
           //  column
           if (solveSudoku(grid, row, col + 1))
               return true;
        }

        // Removing the assigned num ,
        // since our assumption
        // was wrong , and we go for
        // next assumption with
        // diff num value
        grid[row][col] = 0;
    }
    return false;
}
```



```cpp
// Driver Code
int main()
{
    // 0 means unassigned cells
    int grid[N][N] = { { 3, 0, 6, 5, 0, 8, 4, 0, 0 },
                       { 5, 2, 0, 0, 0, 0, 0, 0, 0 },
                       { 0, 8, 7, 0, 0, 0, 0, 3, 1 },
                       { 0, 0, 3, 0, 1, 0, 0, 8, 0 },
                       { 9, 0, 0, 8, 6, 3, 0, 0, 5 },
                       { 0, 5, 0, 0, 9, 0, 6, 0, 0 },
                       { 1, 3, 0, 0, 0, 0, 2, 5, 0 },
                       { 0, 0, 0, 0, 0, 0, 0, 7, 4 },
                       { 0, 0, 5, 2, 0, 6, 3, 0, 0 } };

    if (solveSudoku(grid, 0, 0))
        print(grid);
    else
        cout << "no solution  exists " << endl;

    return 0;
}
```

Output:



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