COMPSCI 1XC3 - Assignment 2

Mina Al-Barak (ID: 400513160)

March 10, 2024

Chapter 1

Report Details

1.1 Program Description

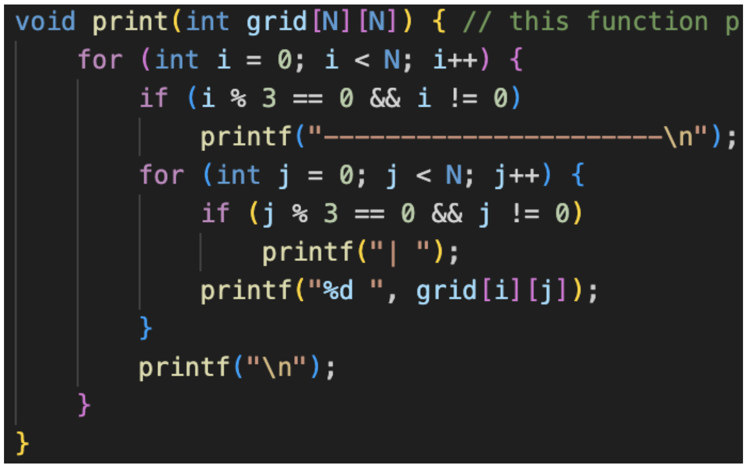
For Assignment 2, I designed a program that solves a hard-coded Sudoku puzzle, displays the solution, and states the number of iterations taken to solve the puzzle. This is achievable by utilizing various C programming methods, such as logical and comparison operators, nested loops, and functions.

1.2 Program Details

1.2.1 void print(int grid[N][N]) function

This function prints the Sudoku grid out. It is void type function because it does not return any values. The function is made of a nested loop, where the outer loop is controlled by variable i and the inner loop is controlled by variable j. We have initialized the loop at i = 0, with a condition for i to be less than N=9 which is the grid size 9x9, and an increment of 1. We say less than 9 because we are working with a zero-index, so we start at 0 and go up to 8 inclusive. I had to put down a horizontal divider between every 3 rows, so I set up a conditional statement if to make sure a divider is placed at every multiple of 3, 0 not inclusive. For that, I used comparison operator == as well as !=, and logical operator &&. This makes sure that i is a multiple of 3 as it has no remainder by division of 3, and i cannot equal to 0, and that both of these conditions are met. Then I want to set a vertical divider; to achieve this, we set an inner loop variable j, and it has to meet the same exact conditions as i. Then I make it print the grid and put placeholders for each cell, so it can later input solutions for each cell into those placeholders.

Figure 1.2.1 shows the first function described above.



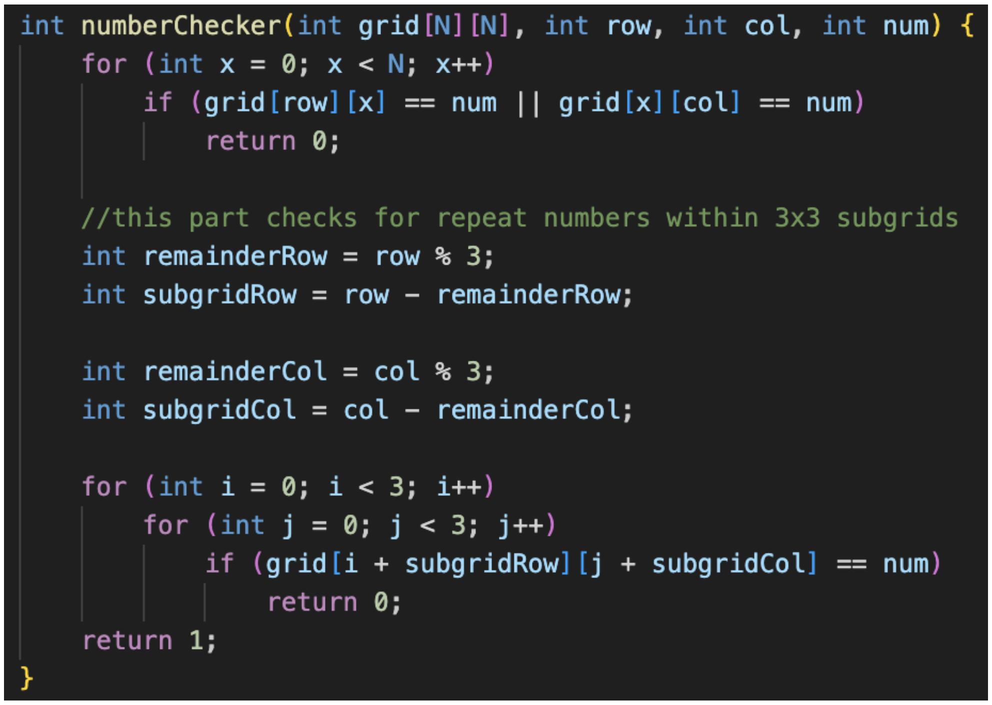
```
void print(int grid[N][N]) { // this function p
    for (int i = 0; i < N; i++) {
        if (i % 3 == 0 && i != 0)
            printf("---------------------\n");
        for (int j = 0; j < N; j++) {
            if (j % 3 == 0 && j != 0)
                printf("| ");
            printf("%d ", grid[i][j]);
        }
        printf("\n");
    }
}
```

1.2.2 int isSafe(int grid[N][N], int row, int col, int num) function

This function checks where a number can be inputted, ie. where an empty cell is to solve the puzzle. So, I call for the parameters which are the row number, column number, and the specific number I'm trying to place in the puzzle. We start this function with a loop, so we reiterate through every number from 1 to 9, through every cell in every row and column. Now to get in depth, we read the function line for line. As stated, the first line is a loop, initialized at 0 with a condition for x to be less than N=9, and an increment of 1. The if conditional statement says that if at any row, x is equal to the number we are trying to put down ie. num, or if at any column, x is equal to num, we cannot put this number down. In Boolean terms, we express this with a 0, so the function returns a 0.

Then, I made a part that scans subgrids, the 3x3 boxes within the puzzle that also can't have repeat numbers. For this, I began by making a calculation that determines the subgrid being read. Then using nested loops, the function reads every cell in the subgrid, such that it returns a 0 if num is found in any cell. This works by deducing the sum of the index and the subgrid row/column number, as to read cell by cell without exceeding 3 cells (since a subgrid is 3x3). Otherwise, return 1 where it is safe to place a number.

Figure 1.2.2 shows the second function described above.

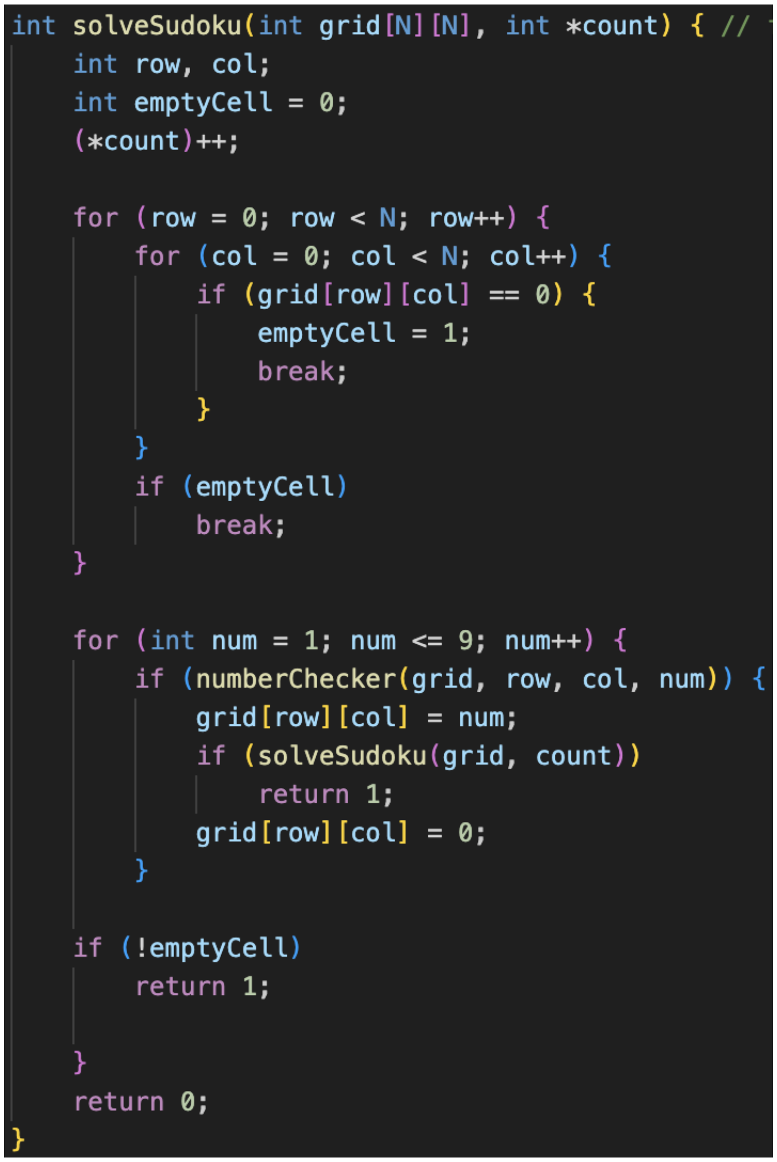


```c
int numberChecker(int grid[N][N], int row, int col, int num) {
    for (int x = 0; x < N; x++)
        if (grid[row][x] == num || grid[x][col] == num)
            return 0;

    //this part checks for repeat numbers within 3x3 subgrids
    int remainderRow = row % 3;
    int subgridRow = row - remainderRow;

    int remainderCol = col % 3;
    int subgridCol = col - remainderCol;

    for (int i = 0; i < 3; i++)
        for (int j = 0; j < 3; j++)
            if (grid[i + subgridRow][j + subgridCol] == num)
                return 0;
    return 1;
}
```

1.2.3 int solveSudoku(int grid[N][N], int *count) function

This function solves the Sudoku puzzle. It utilizes a pointer to count the number of iterations that the function passes through to solve the puzzle. I used a pointer instead of an increment loop because I noticed that the pointer eliminated the issue of passing by reference. Because count needs to retain its value over recursive calls, I need to utilize a pointer so the function can directly modify the original variable in memory and not just a copy of it. Moving on, I declared the variable emptyCell that tracks where empty cells are. The function flags them as it runs. Then, I started a loop that runs through the rows and columns of the puzzle. The following nested loop determines what numbers are placed inside empty cells; it runs values 1 through 9 into each empty cell. If a solution is found, it returns Boolean value 1, otherwise it returns 0 which indicates the function must backtrack and try inputting a different number. In a similar fashion, if the outer loop does not find a solution, it also backtracks, indicating no solution can be found.

Figure 1.2.3 shows the third function described above.

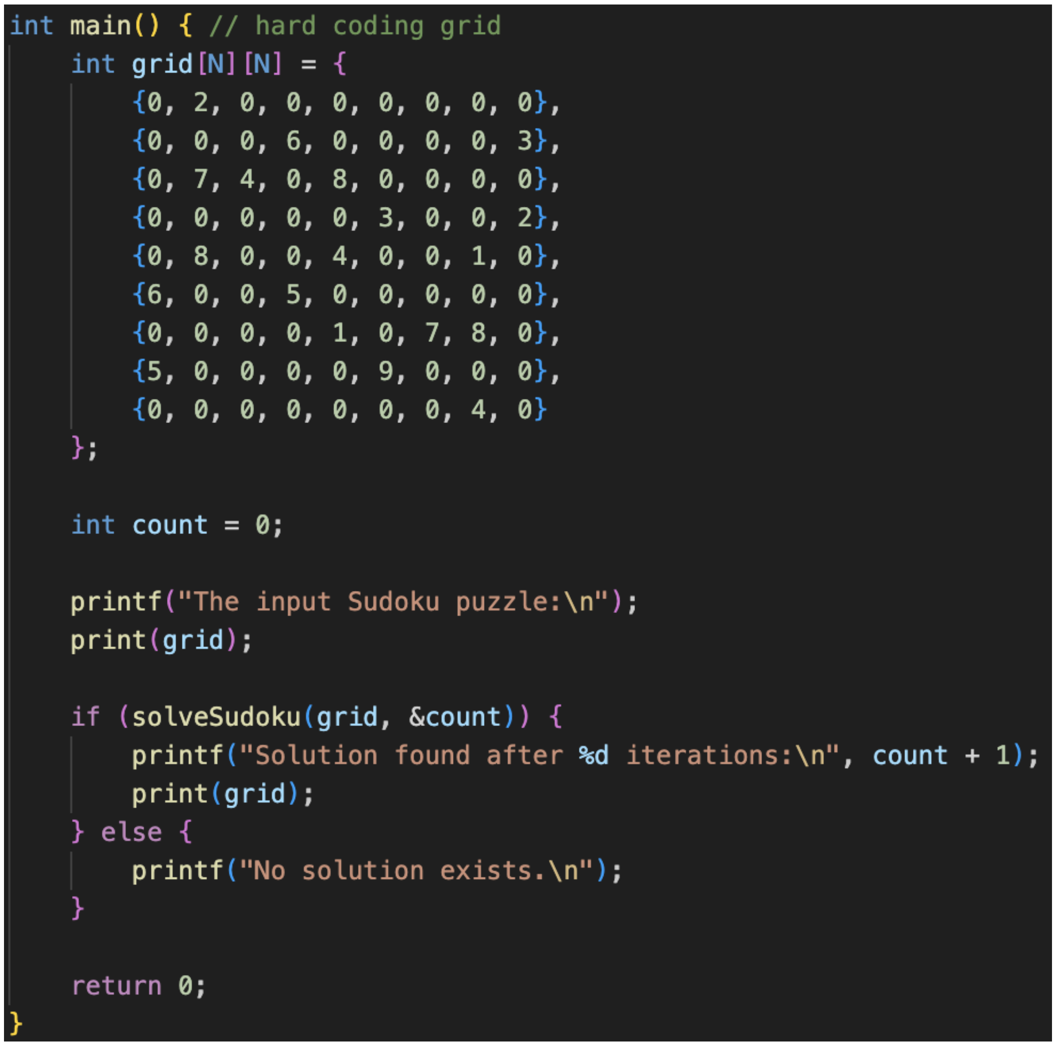


```
int solveSudoku(int grid[N][N], int *count) { //
    int row, col;
    int emptyCell = 0;
    (*count)++;

    for (row = 0; row < N; row++) {
        for (col = 0; col < N; col++) {
            if (grid[row][col] == 0) {
                emptyCell = 1;
                break;
            }
        }
        if (emptyCell)
            break;
    }

    for (int num = 1; num <= 9; num++) {
        if (numberChecker(grid, row, col, num)) {
            grid[row][col] = num;
            if (solveSudoku(grid, count))
                return 1;
            grid[row][col] = 0;
        }

    if (!emptyCell)
        return 1;

    }
    return 0;
}
```

1.2.4 Main function

The main function is made up of the hard-coded puzzle provided by the assignment outline. It initializes the iteration count to 0. Then, it prints the puzzle, as well as the iteration count if a solution was found, or states that no solutions exist if so.

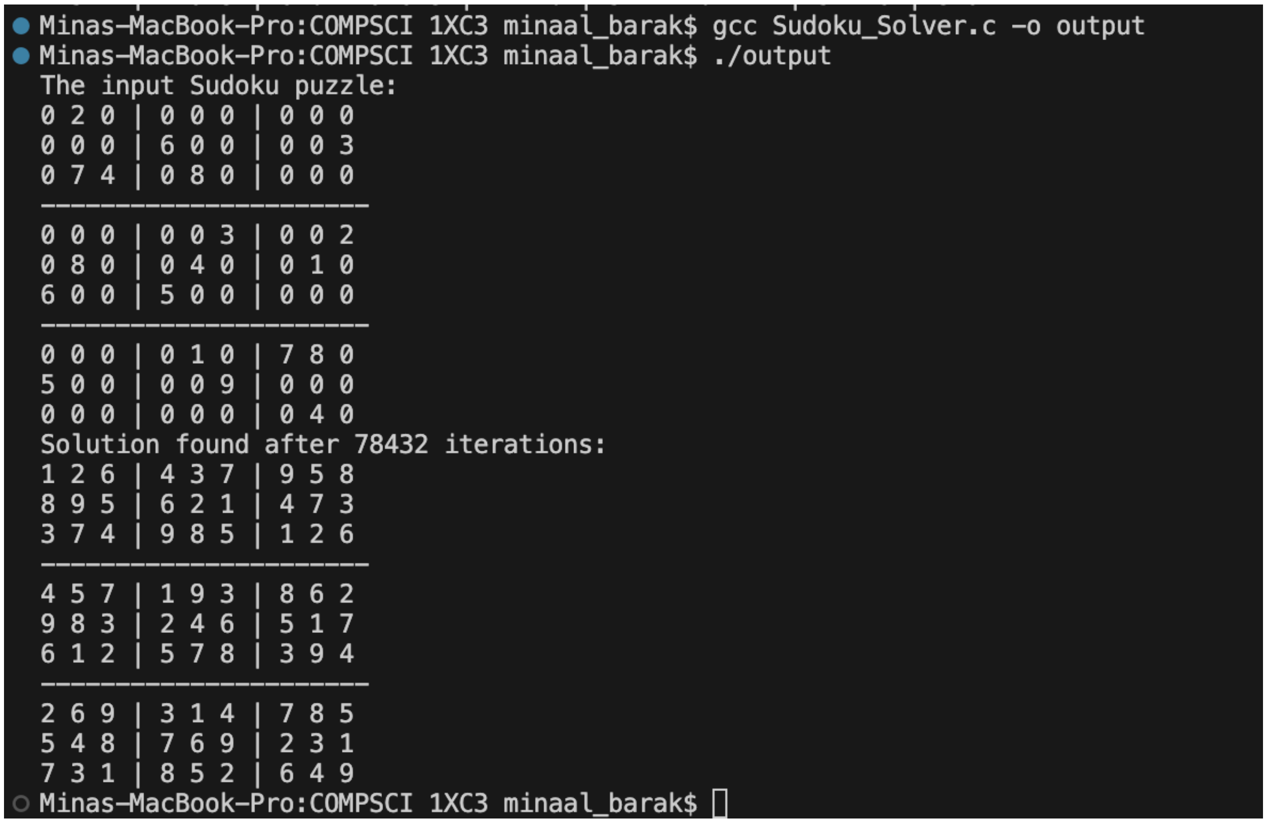
Figure 1.2.4 shows the second function described above.



```
int main() { // hard coding grid
    int grid[N][N] = {
        {0, 2, 0, 0, 0, 0, 0, 0, 0},
        {0, 0, 0, 6, 0, 0, 0, 0, 3},
        {0, 7, 4, 0, 8, 0, 0, 0, 0},
        {0, 0, 0, 0, 0, 3, 0, 0, 2},
        {0, 8, 0, 0, 4, 0, 0, 1, 0},
        {6, 0, 0, 5, 0, 0, 0, 0, 0},
        {0, 0, 0, 0, 1, 0, 7, 8, 0},
        {5, 0, 0, 0, 0, 9, 0, 0, 0},
        {0, 0, 0, 0, 0, 0, 0, 4, 0}
    };

    int count = 0;

    printf("The input Sudoku puzzle:\n");
    print(grid);

    if (solveSudoku(grid, &count)) {
        printf("Solution found after %d iterations:\n", count + 1);
        print(grid);
    } else {
        printf("No solution exists.\n");
    }

    return 0;
}
```

1.2.5 Output

Figure 1.2.5 shows the code output when run on MacOS.



```
● Minas-MacBook-Pro:COMPSCI 1XC3 minaal_barak$ gcc Sudoku_Solver.c -o output
● Minas-MacBook-Pro:COMPSCI 1XC3 minaal_barak$ ./output
  The input Sudoku puzzle:
  0 2 0 | 0 0 0 | 0 0 0
  0 0 0 | 6 0 0 | 0 0 3
  0 7 4 | 0 8 0 | 0 0 0
  ---------------------
  0 0 0 | 0 0 3 | 0 0 2
  0 8 0 | 0 4 0 | 0 1 0
  6 0 0 | 5 0 0 | 0 0 0
  ---------------------
  0 0 0 | 0 1 0 | 7 8 0
  5 0 0 | 0 0 9 | 0 0 0
  0 0 0 | 0 0 0 | 0 4 0
  Solution found after 78432 iterations:
  1 2 6 | 4 3 7 | 9 5 8
  8 9 5 | 6 2 1 | 4 7 3
  3 7 4 | 9 8 5 | 1 2 6
  ---------------------
  4 5 7 | 1 9 3 | 8 6 2
  9 8 3 | 2 4 6 | 5 1 7
  6 1 2 | 5 7 8 | 3 9 4
  ---------------------
  2 6 9 | 3 1 4 | 7 8 5
  5 4 8 | 7 6 9 | 2 3 1
  7 3 1 | 8 5 2 | 6 4 9
○ Minas-MacBook-Pro:COMPSCI 1XC3 minaal_barak$ 
```

1.3 Sources

I used a combination of Prof. Pasandide's lecture notes, sections 2.6 and 3.4, as well as ChatGPT to help me determine how to code how the functions read rows and columns of the grid, as well as interpreting Boolean values.