NAME: NICHOLAUS ISACK THOMAS

ID: 20215035

Artificial Intelligence: Homework 2

I implemented solutions for the two problems using the Backtracking algorithm, which is the recursive solution for solving problems by trying to build a solution incrementally, decarding the solutions that fail to satisfy the constraints of the problem at any point in time (time elapsed till reaching any level of the search tree).

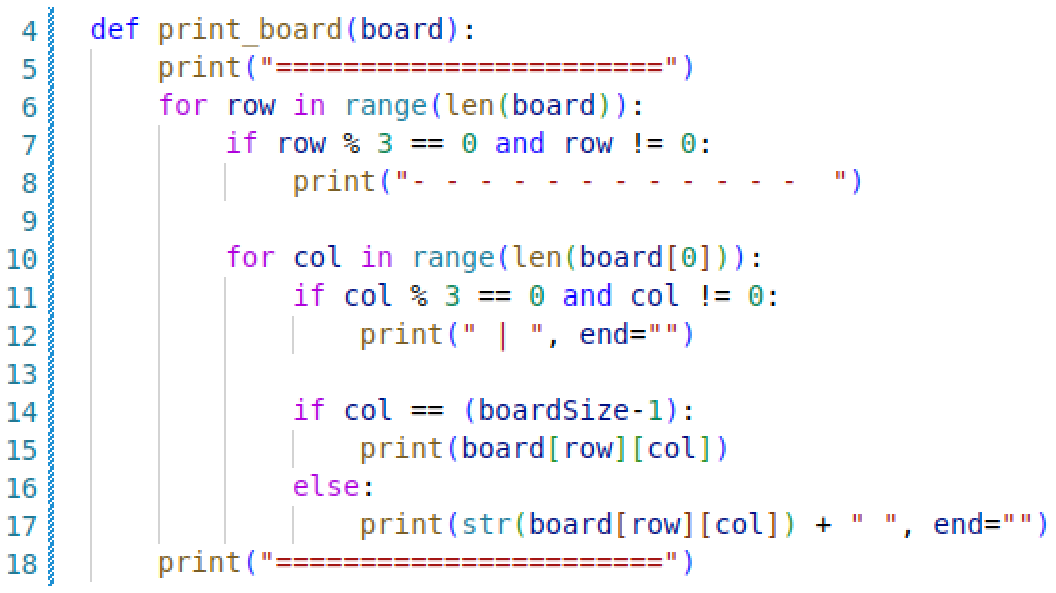
Compared with a naive approach, which relies on the generation of all possible combinations of the solution and the try every combination, Backtracking is far more efficient because wherever it finds a solution that cannot lead to a solution, it discards that solution (backtracks) and tries the next solution instantly.

1. Solving Sudoku using backtracking:
    a) Constraints/rules to solve Standard [9x9] Sudoku
        - No digit repetition on the current row, column, or 3x3 subgrid.

    b) Searching for solution
        i) Check if it is safe to assign a certain digit to a particular empty cell, that is, if there is no same digit in the current row, column, or 3x3 subgrid.
        ii) Recursively check whether the chosen digit and cell lead to a valid solution or not.
        iii) If it doesn't, discard it and try the next digit for that empty cell.
        iv) Return false if none of the digits (1 to 9) leads to a valid solution; else, return true, print the solution, and ex.

Steps followed and implementation steps:

The following function prints out the solution of the solved Sudoku.

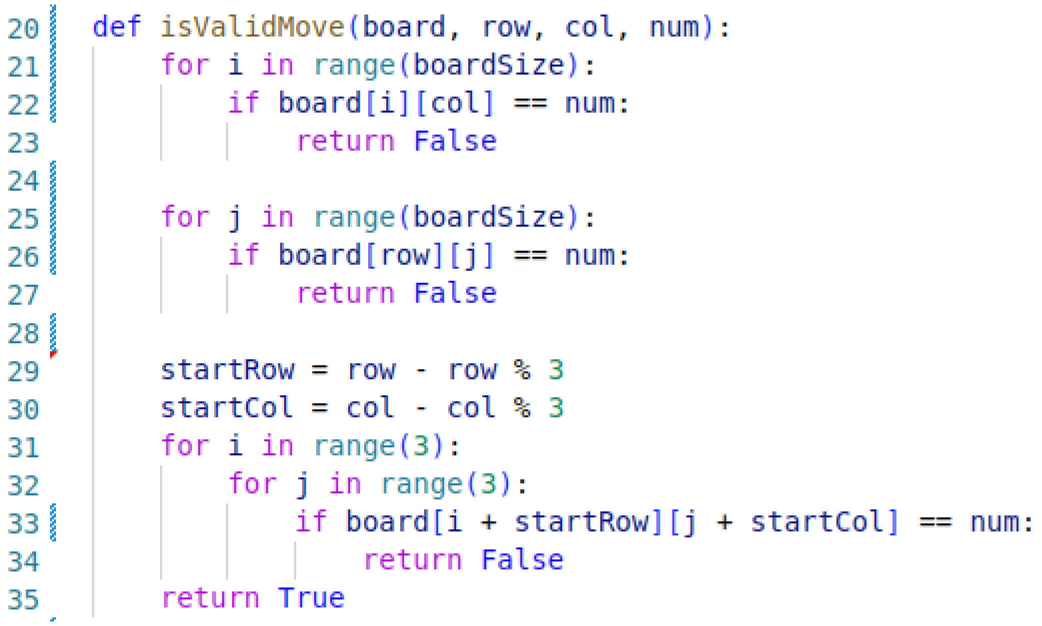


```python
def print_board(board):
    print("=======================")
    for row in range(len(board)):
        if row % 3 == 0 and row != 0:
            print("- - - - - - - - - - - -   ")

        for col in range(len(board[0])):
            if col % 3 == 0 and col != 0:
                print(" | ", end="")

            if col == (boardSize-1):
                print(board[row][col])
            else:
                print(str(board[row][col]) + " ", end="")
    print("=======================")
```

The following function inspects and returns True if the digit assignment is valid first in the current column, then the current row, and the subgrid in which the current cell is located.

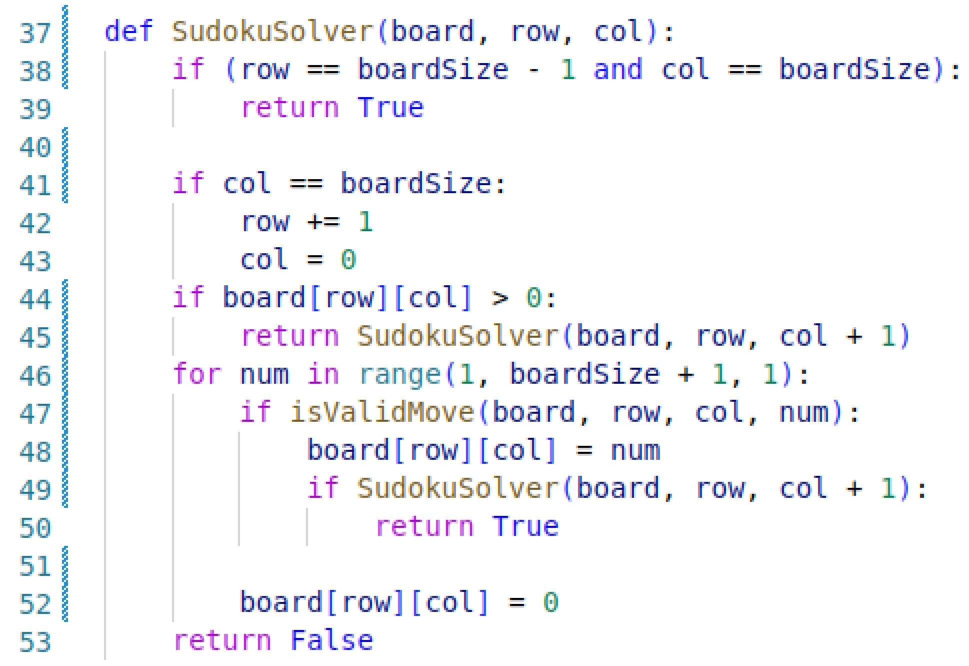


```python
def isValidMove(board, row, col, num):
    for i in range(boardSize):
        if board[i][col] == num:
            return False

    for j in range(boardSize):
        if board[row][j] == num:
            return False

    startRow = row - row % 3
    startCol = col - col % 3
    for i in range(3):
        for j in range(3):
            if board[i + startRow][j + startCol] == num:
                return False
    return True
```

The following function is the backbone of our solver. It implements the backtracking algorithm. It starts by receiving a board in a particular (incomplete) state and attempting to assign a digit to an empty cell. It returns true if we hit the end of the board and prevent unnecessary backtracking. Then it moves to the next row and iteratively checks if the cell is empty or not. If it's empty, it checks if it is a legal move and assigns a digit, then recursively performs a digit assignment, checks validity, and discards the move if it doesn't satisfy the constraints.



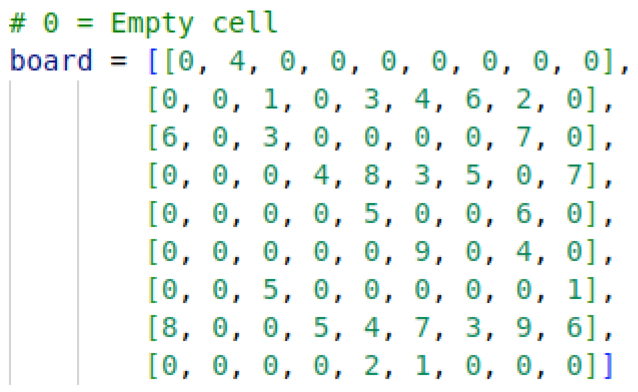
```python
def SudokuSolver(board, row, col):
    if (row == boardSize - 1 and col == boardSize):
        return True

    if col == boardSize:
        row += 1
        col = 0
    if board[row][col] > 0:
        return SudokuSolver(board, row, col + 1)
    for num in range(1, boardSize + 1, 1):
        if isValidMove(board, row, col, num):
            board[row][col] = num
            if SudokuSolver(board, row, col + 1):
                return True

        board[row][col] = 0
    return False
```

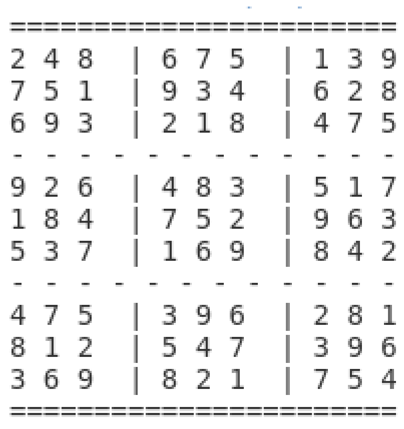
Testing the implementation:

Sample Input



```python
# 0 = Empty cell
board = [[0, 4, 0, 0, 0, 0, 0, 0, 0],
         [0, 0, 1, 0, 3, 4, 6, 2, 0],
         [6, 0, 3, 0, 0, 0, 0, 7, 0],
         [0, 0, 0, 4, 8, 3, 5, 0, 7],
         [0, 0, 0, 0, 5, 0, 0, 6, 0],
         [0, 0, 0, 0, 0, 9, 0, 4, 0],
         [0, 0, 5, 0, 0, 0, 0, 0, 1],
         [8, 0, 0, 5, 4, 7, 3, 9, 6],
         [0, 0, 0, 0, 2, 1, 0, 0, 0]]
```

Output solution

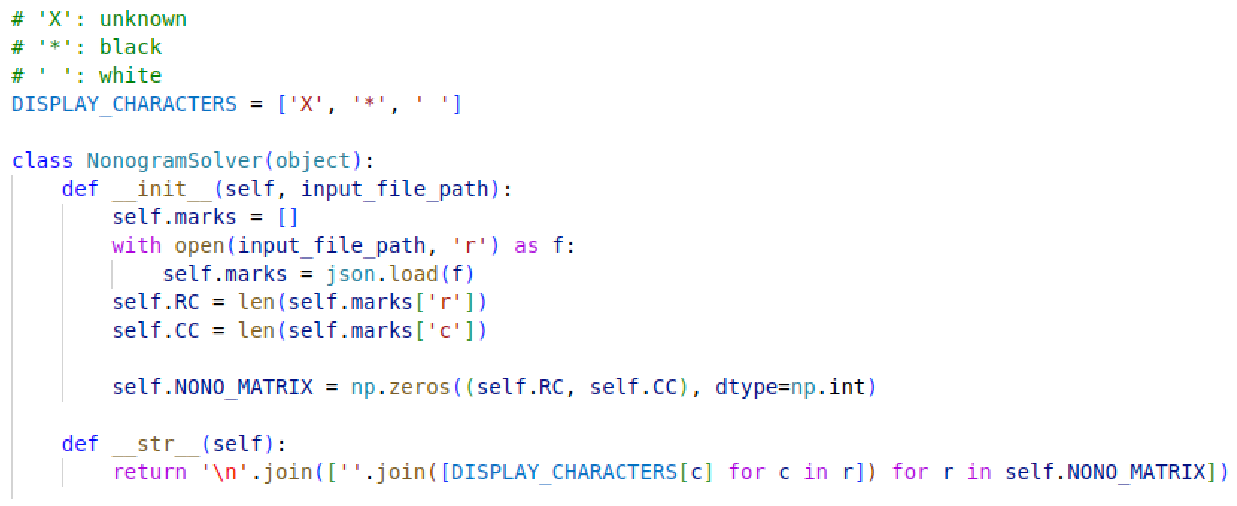


```
=======================
2 4 8  | 6 7 5  | 1 3 9
7 5 1  | 9 3 4  | 6 2 8
6 9 3  | 2 1 8  | 4 7 5
- - - - - - - - - - - -
9 2 6  | 4 8 3  | 5 1 7
1 8 4  | 7 5 2  | 9 6 3
5 3 7  | 1 6 9  | 8 4 2
- - - - - - - - - - - -
4 7 5  | 3 9 6  | 2 8 1
8 1 2  | 5 4 7  | 3 9 6
3 6 9  | 8 2 1  | 7 5 4
=======================
```

2. Nonogram Problem:
    a) Constraints/rules to solve any Nanogram problem.
        - All the rules and tricks are obtained from a blog found here: https://gambiter.com/puzzle/Nonogram.html

Steps followed and implementation steps:

I implemented the solution using a python class and several helper functions. The class initializes the marks, loads the user input from the JSON file, gets the number of rows and columns, and initializes the puzzle matrix with zeroes.



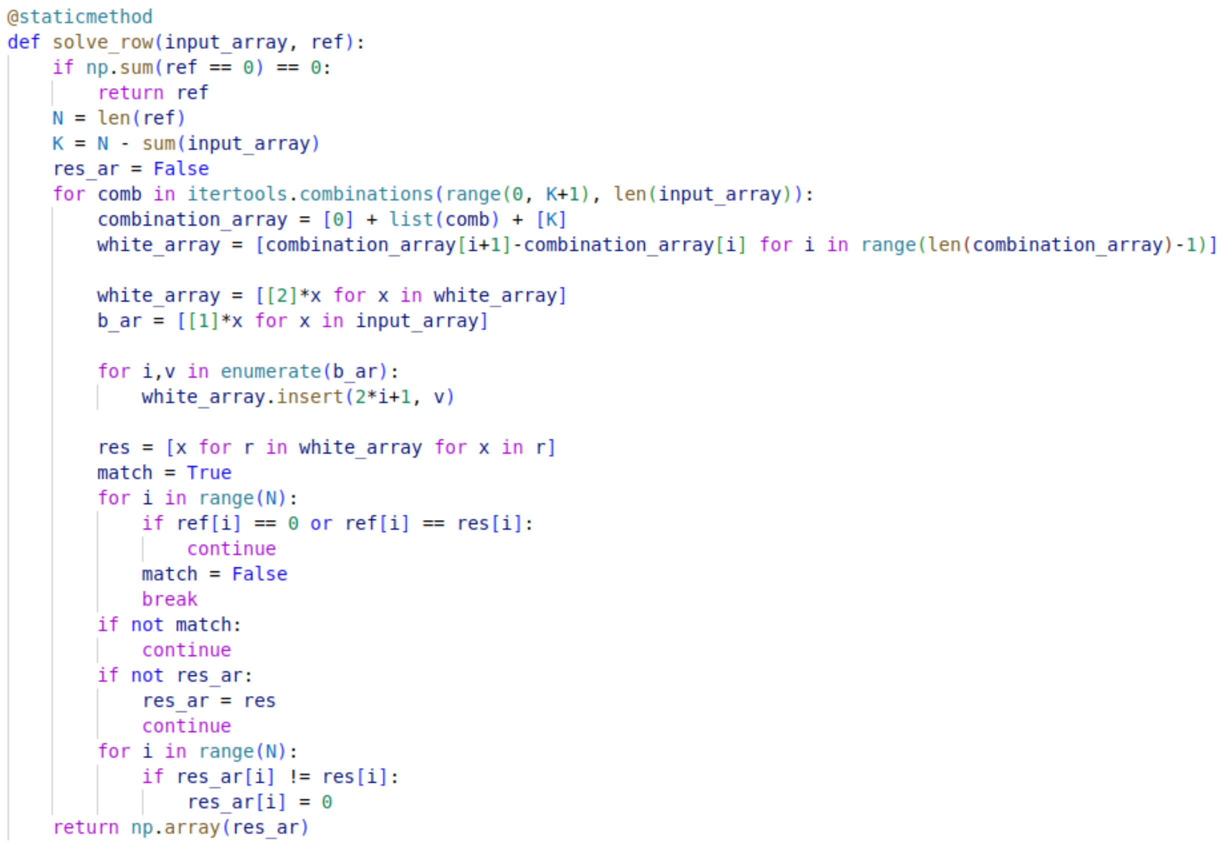
```python
# 'X': unknown
# '*': black
# ' ': white
DISPLAY_CHARACTERS = ['X', '*', ' ']

class NonogramSolver(object):
    def __init__(self, input_file_path):
        self.marks = []
        with open(input_file_path, 'r') as f:
            self.marks = json.load(f)
        self.RC = len(self.marks['r'])
        self.CC = len(self.marks['c'])

        self.NONO_MATRIX = np.zeros((self.RC, self.CC), dtype=np.int)

    def __str__(self):
        return '\n'.join([''.join([DISPLAY_CHARACTERS[c] for c in r]) for r in self.NONO_MATRIX])
```

I used a row-solving algorithm to generate all combinations that don't violate previously found cells for each possible combination mark common cells. It returns the result of common cells as a new solution row. An array of the count of consecutive black marks and returns yielded a possible combination of marks starting and ending with num of empty.



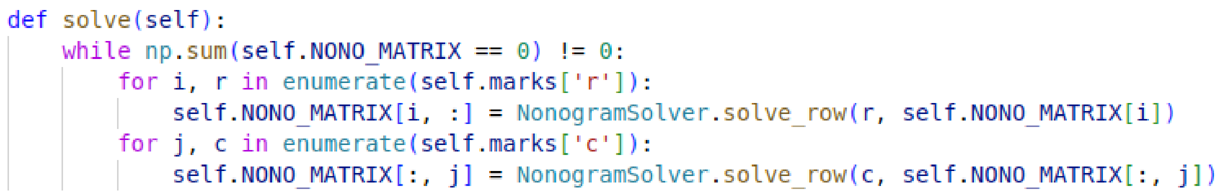
```
@staticmethod
def solve_row(input_array, ref):
    if np.sum(ref == 0) == 0:
        return ref
    N = len(ref)
    K = N - sum(input_array)
    res_ar = False
    for comb in itertools.combinations(range(0, K+1), len(input_array)):
        combination_array = [0] + list(comb) + [K]
        white_array = [combination_array[i+1]-combination_array[i] for i in range(len(combination_array)-1)]

        white_array = [[2]*x for x in white_array]
        b_ar = [[1]*x for x in input_array]

        for i,v in enumerate(b_ar):
            white_array.insert(2*i+1, v)

        res = [x for r in white_array for x in r]
        match = True
        for i in range(N):
            if ref[i] == 0 or ref[i] == res[i]:
                continue
            match = False
            break
        if not match:
            continue
        if not res_ar:
            res_ar = res
            continue
        for i in range(N):
            if res_ar[i] != res[i]:
                res_ar[i] = 0
    return np.array(res_ar)
```

I implemented a Backtracking algorithm along with a simple algorithm that turns every column and row into rows, then solves for generated row repeat, updates the result matrix, then repeats the process until all cells are solved.



```
def solve(self):
    while np.sum(self.NONO_MATRIX == 0) != 0:
        for i, r in enumerate(self.marks['r']):
            self.NONO_MATRIX[i, :] = NonogramSolver.solve_row(r, self.NONO_MATRIX[i])
        for j, c in enumerate(self.marks['c']):
            self.NONO_MATRIX[:, j] = NonogramSolver.solve_row(c, self.NONO_MATRIX[:, j])
```

The following part of the code initiates the program's execution but accepts the JSON file path as input.

```
parser = argparse.ArgumentParser()
parser.add_argument('-f', '--file', required=True, help='Nonogram user input file in json format')
args = parser.parse_args()

nono_solution = NonogramSolver(args.file)
nono_solution.solve()
print(nono_solution)
```

The input file must contain two sets of elements (Key value pairs): ' r' for a row that indicates how many colored blocks are for a particular row and 'c' for how many blocks for a specific column.

The following is the sample input file:

```
{
    "r": [
        [1, 1],
        [2, 2],
        [5],
        [7]
    ],
    "c": [
        [1],
        [6],
        [2, 6],
        [8]
    ]
}
```

Testing:

Input

```
{
    "r": [
        [1, 1],
        [2, 2],
        [5],
        [1, 1, 1],
        [7],
        [5, 2],
        [3, 1],
        [3, 1],
        [4, 2],
        [7]
    ],
    "c": [
        [1],
        [6],
        [2, 6],
        [8],
        [2, 6],
        [6, 2],
        [1, 1],
        [1],
        [1, 2],
        [4]
    ]
}
```

Output:

```
 *   *
 ** **
 *****
 * * *
*******
 *****  **
  ***    *
  ***    *
  ****  **
  *******
```