**KATHMANDU UNIVERSITY**

SCHOOL OF SCIENCE

**DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING **

**[COMP 314]**

**Mini-Project Report**

**on**

**“Sudoku Solver”**

**Submitted by**

Allen Maharjan

**Submitted to**

**Rajani Chulyadyo**

**Department of Computer Science and Engineering**

**Submission Date: 03 Dec 2020**

**Introduction**

An **algorithm** is a procedure or formula for solving a problem, based on conducting a sequence of specified actions. A computer program can be viewed as an elaborate algorithm. In mathematics and computer science, an algorithm usually means a small procedure that solves a recurrent problem. There are different algorithm strategies that are present today and backtracking is one of them.

**Backtracking** is an algorithmic-technique for solving problems recursively by trying to build a solution incrementally, one piece at a time, removing those solutions that fail to satisfy the constraints of the problem at any point of time. This algorithm strategy can be used to solve many problems and Sudoku solving is one of them.

**Sudoku:** The basic premise of solving a sudoku puzzle is to place the numbers one through nine exactly once in each row, column and 3x3 box based on the starting numbers that are already filled in and any subsequent numbers you fill in. This should be done by logic rather than guessing.

**Backtracking and Sudoku**

Backtracking Algorithm, in regards with Sudoku, is a simple method of filling one square and then moving to the other square, filling it with one number after counting 0-9, satisfying the situation. Once we are left with no choice to fill then roll back to the previous square and fill it will another number and then following the same.

In this project I am going to solve a Sudoku board with the help of backtracking algorithm.

**Algorithm for solving Sudoku using backtracking**

1. Find an empty spot (in this case 0) and check whether the number inserted is valid or not.
   1. If the inserted number is valid:
      1. Find another empty spot and check again.
      2. Repeat 2(a) until there is no empty spot and return True.
   2. If the number inserted is not valid, return False change the number in step 1 and repeat the process again.
2. Repeat step 1 until we reach an optimal solution.

**Pseudo code:**

solve():

if Find\_Empty() <== Null

return True

else

a,b <== Find\_Empty()

for i in len(board)

if check()== True

board[a][b] <== i

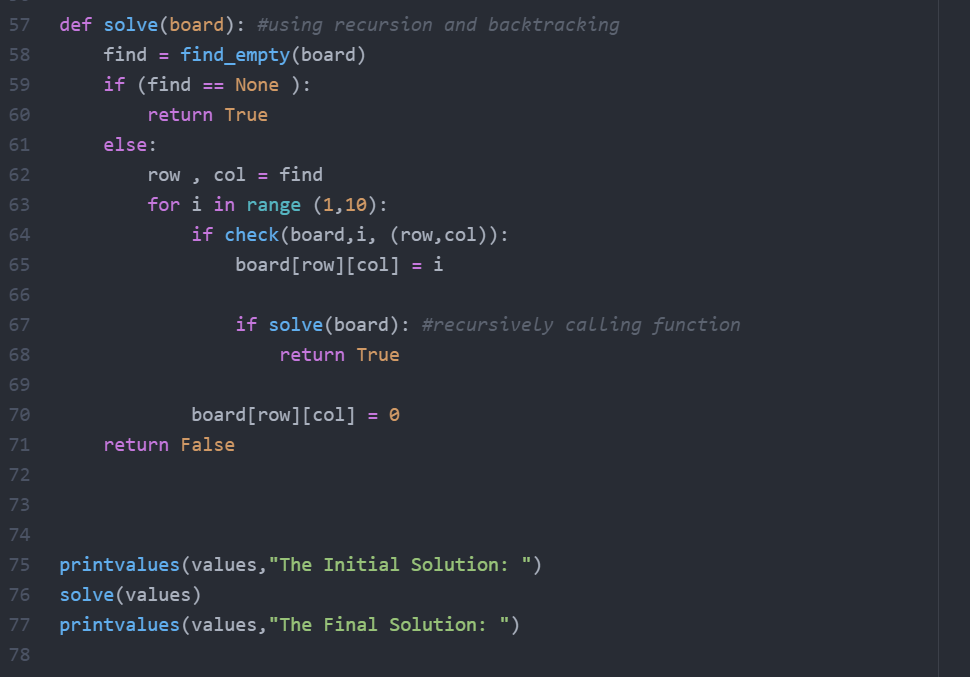
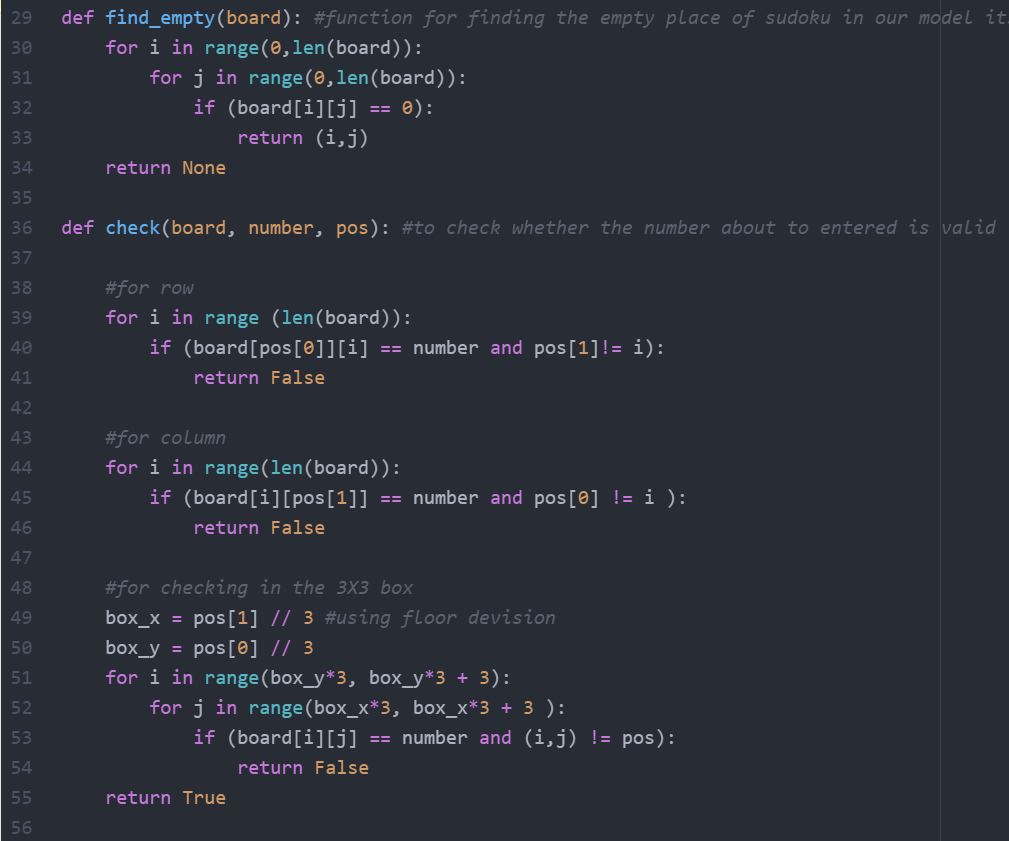
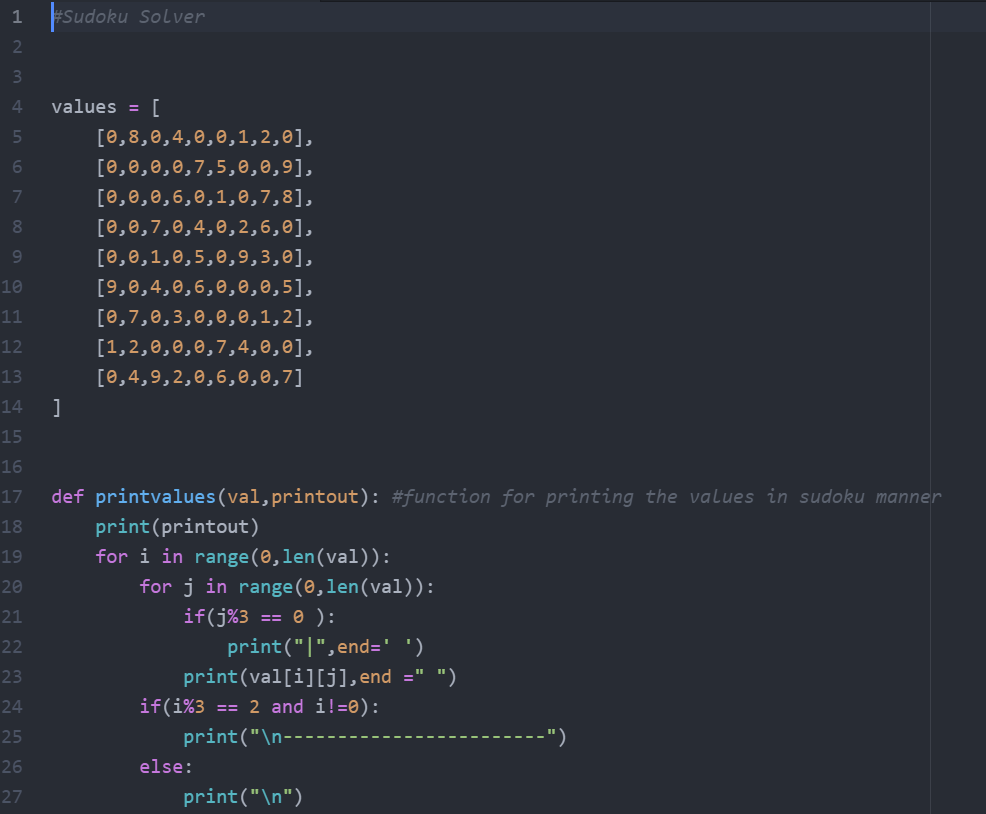
if solve():

return True

board[a][b] == 0

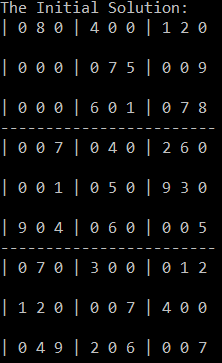
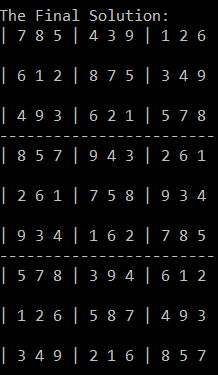
return False

**Source Code:**



**Chapter 3 : Conclusion**

Sudoku can be solved by different ways backtracking and brute force are some of them. In backtracking for every unassigned index, there are 9 possible options so the time complexity is O (9^ (n\*n)). The time complexity remains the same but there will be some early pruning so the time taken will be much less than the naive algorithm but the upper bound time complexity remains the same. Space Complexity: O (n\*n). During research I have also found that programmers reported that such an algorithm may typically require as few as 15,000 cycles, or as many as 900,000 cycles to solve a Sudoku, each cycle being the change in position of a "pointer" as it moves through the cells of a Sudoku. A Sudoku can be constructed to work against backtracking.

****