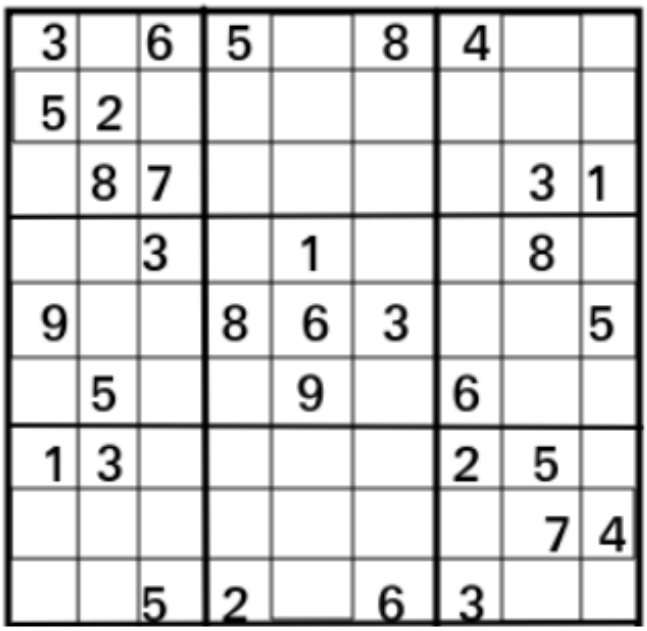
Advanced Algorithm Mini Project Report

Name : Vaibhav Vijay

SRN : PES1UG20CS479

Date : 24-11-2022

Project Title : Sudoku Solver



Given a partially filled grid of size 9×9, goal is to assign numbers from 1 to 9 to the empty cells such that every row, every column and every sub grid of size 3×3 contains exactly one instance of numbers from 1 to 9

This game has become popular now in large number of countries and many developers have tried to generate more complicated and interesting puzzles

Solving Sudoku helps to improve concentration, reduce anxiety and stress, promotes a healthy competition and improves problem-solving skills

There are 3 methods of implementing Sudoku solver –

1. Brute Force

This approach is used to generate all possible configurations of numbers from 1 to 9 to fill the empty cells. Try every configuration one by one until the correct configuration is found. After filling all the unassigned positions check if the matrix is safe or not. If safe print else recurs for other cases

bruteforce.c

```c
#include<stdio.h>
int isSafe(int N,int grid[N][N],int row,int col,int num)
{
    for(int x = 0;x <= 8;x++)
    {
        if(grid[row][x] == num)
            return 0;
    }
    for (int x = 0;x <= 8;x++)
    {
        if (grid[x][col] == num)
            return 0;
    }
    int startRow;
    startRow = row - row % 3;
    int startCol;
    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 0;
        }
    }
    return 1;
}
```

```c
int solveSudoku(int N,int grid[N][N],int row,int col)
{
    if(row == N - 1 && col == N)
        return 1;
    if(col == N)
    {
        row++;
        col = 0;
    }
    if(grid[row][col] > 0)
        return solveSudoku(N,grid,row,col + 1);
    for(int num = 1;num <= N;num++)
    {
        if(isSafe(N,grid,row,col,num))
        {
            grid[row][col] = num;
            if(solveSudoku(N,grid,row,col + 1))
                return 1;
        }
        grid[row][col] = 0;
    }
    return 0;
}
void printGrid(int N,int arr[N][N])
{
    for(int i = 0;i < N;i++)
    {
        for(int j = 0;j < N;j++)
            printf("%d\t",arr[i][j]);
        printf("\n");
    }
}
```

```c
int main()
{
    int N = 9;
    int grid[9][9] = {{ 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(N,grid,0,0))
        printGrid(N,grid);
    else
        printf("No solution exists");
    return 0;
}
```

2. Backtracking

Before assigning a number check whether it is safe to assign. After checking for safety assign that number and recursively check whether this assignment leads to a solution or not. If the assignment does not lead to a solution, try next number for the current empty cell. And if none of the numbers from 1 to 9 leads to a solution, then print no solution exists

backtracking.c

```c
#include <stdio.h>
int findUnassignedLocation(int N,int grid[N][N],int *row,int *col)
{
    int i;
    int j;
    for(i = 0;i < N;i++)
    {
        for(j = 0;j < N;j++)
        {
            if(grid[i][j] == 0)
            {
                *row = i;
                *col = j;
                return 1;
            }
        }
    }
    *row = i;
    *col = j;
    return 0;
}
int usedInRow(int N,int grid[N][N],int row,int num)
{
    for(int col = 0;col < N;col++)
    {
        if (grid[row][col] == num)
            return 1;
    }
    return 0;
}
```

```c
int usedInCol(int N,int grid[N][N],int col,int num)
{
    for(int row = 0;row < N;row++)
    {
        if(grid[row][col] == num)
            return 1;
    }
    return 0;
}
int usedInBox(int N,int grid[N][N],int boxStartRow,int boxStartCol,int num)
{
    for(int row = 0; row < 3; row++)
    {
        for(int col = 0; col < 3; col++)
        {
            if(grid[row + boxStartRow][col + boxStartCol] == num)
                return 1;
        }
    }
    return 0;
}
int isSafe(int N,int grid[N][N],int row,int col,int num)
{
        return !usedInRow(N,grid,row,num) && !usedInCol(N,grid,col,num) && !usedInBox(N,grid,row - row % 3,col - col % 3,num) && grid[row][col] == 0;
}
```

```c
int solveSudoku(int N,int grid[N][N])
{
    int row;
    int col;
    if (!findUnassignedLocation(N,grid,&row,&col))
        return 1;
    for(int num = 1;num <= N;num++)
    {
        if(isSafe(N,grid,row,col,num))
        {
            grid[row][col] = num;
            if(solveSudoku(N,grid))
                return 1;
            grid[row][col] = 0;
        }
    }
    return 0;
}
void printGrid(int N,int grid[N][N])
{
    for(int row = 0;row < N;row++)
    {
        for(int col = 0;col < N;col++)
            printf("%d\t",grid[row][col]);
        printf("\n");
    }
}
```

```c
int main()
{
    int N = 9;
    int grid[9][9] = {{ 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(N,grid))
        printGrid(N,grid);
    else
        printf("No solution exists");
    return 0;
}
```

3. Bit Masks

This method is a slight optimization to the above 2 methods. For each row or column or box create a bitmask and for each element in the grid set the bit at position ‘value’ to 1 in the corresponding bitmasks

bitmasks.c

```c
#include<stdio.h>
#define N 9
int row[N];
int col[N];
int box[N];
int getBox(int i,int j)
{
    return i/3*3 + j/3;
}
int isSafe(int i,int j,int number)
{
    return !((row[i] >> number) & 1) && !((col[j] >> number) & 1) && !((box[getBox(i, j)] >> number) & 1);
}
void setInitialValues(int grid[N][N])
{
    for(int i = 0;i < N;i++)
    {
        for(int j = 0;j < N;j++)
        {
            row[i] |= 1 << grid[i][j];
            col[j] |= 1 << grid[i][j];
            box[getBox(i, j)] |= 1 << grid[i][j];
        }
    }
}
```

```
int solveSudoku(int grid[N][N],int i,int j,int *seted)
{
    if(!*seted)
    {
        *seted = 1;
        setInitialValues(grid);
    }
    if(i == N-1 && j == N)
        return 1;
    if(j == N)
    {
        j = 0;
        i++;
    }
    if(grid[i][j])
        return solveSudoku(grid,i,j + 1,seted);
    for(int nr = 1;nr <= N;nr++)
    {
        if(isSafe(i,j,nr))
        {
            grid[i][j] = nr;
            row[i] |= 1 << nr;
            col[j] |= 1 << nr;
            box[getBox(i, j)] |= 1 << nr;
            if(solveSudoku(grid,i,j + 1,seted))
                return 1;
            row[i] &= ~(1 << nr);
            col[j] &= ~(1 << nr);
            box[getBox(i, j)] &= ~(1 << nr);
        }
        grid[i][j] = 0;
    }
    return 0;
}
```

```
void printGrid(int arr[N][N])
{
    for(int i = 0;i < N;i++)
    {
        for(int j = 0;j < N;j++)
            printf("%d\t",arr[i][j]);
        printf("\n");
    }
}
int main()
{
    int grid[9][9] = {{ 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 }};
    int seted = 0;
    if(solveSudoku(grid,0,0,&seted))
        printGrid(grid);
    else
        printf("No solution exists");
    return 0;
}
```

Brute Force Technique of Solving Sudoku

- create a function that checks if the given matrix is a valid sudoku or not. Keep hashmap for the row, column and boxes. If any number has frequency greater than 1 in the hashmap return false else return true

- create a recursive function that takes grid, current row and column index

- check for some base cases

    - if index is at i = N - 1 and j = N, then check if the grid is safe or not. If safe print grid and return true else return false

    - if j = N, then do i++ and j = 0

- if the current index is not assigned then fill elements from 1 to 9 and recur for all 9 cases with index of next element. If the recursive call returns true then break the loop and return true

- if the current index is assigned then call the recursive function with the index of the next element

- since for every unassigned index there are 9 possible options, so time complexity is O(9^(N^2))

- since for storing the output array a matrix is needed, so space complexity is O(N^2)

```
C:\Users\dell\Desktop\advanced algorithms>gcc bruteforce.c

C:\Users\dell\Desktop\advanced algorithms>a
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
```

Backtracking Technique of Solving Sudoku

- create a function that checks after assigning the current index if grid becomes unsafe or not. Keep hashmap for row, column and boxes. If any number has frequency greater than 1 in the hashmap return false else return true. hashmap can be avoided by using loops

- create a recursive function that takes a grid

- check for any unassigned location

    - if present then assign a number from 1 to 9

    - check if assigning the number to current index makes the grid unsafe or not

    - if safe then recursively call the function for all safe cases from 0 to 9

    - if any recursive call returns true end the loop and return true. If no recursive call returns true then return false

- if there is no unassigned location then return true

- since for every unassigned index there are 9 possible options, so time complexity is O(9^(N^2)). The time complexity remains same but there will be some early pruning so time taken will be much less than the brute approach, but upper bound time complexity remains same

- since for storing the output array a matrix is needed, so space complexity is O(N^2)

```
C:\Users\dell\Desktop\advanced algorithms>gcc backtracking.c

C:\Users\dell\Desktop\advanced algorithms>a
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
```

Bit Masks Technique of Solving Sudoku

- create 3 arrays each one for rows, columns and boxes of size N

- boxes are indexed from 0 to 8. In order to find the box index use this formula: row / 3 * 3 + column / 3

- map the initial values of grid

- each time we add an element to the grid or remove an element from the grid set the bit to 1 or 0 accordingly to the corresponding bitmasks

- since for every unassigned index there are 9 possible options, so time complexity is O(9^(N^2)). Time complexity remains the same but checking if a number is safe to use is much faster as it takes O(1)

- since for storing the output array a matrix is needed and 3 extra arrays of size N are required for bitmasks, so space complexity is O(N^2)

```
C:\Users\dell\Desktop\advanced algorithms>gcc bitmasks.c

C:\Users\dell\Desktop\advanced algorithms>a
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
```

Sudoku game developed using brute force approach

sudoku_server.c

```c
#include<stdio.h>
#include<stdlib.h>
#include "sudoku_header.h"
int **createPuzzle()
{
    int **puzzle;
    int arrayPuzzle[9][9] = {{ 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 }};
    puzzle = (int**)malloc(sizeof(int*) * 9);
    for(int i = 0;i < 9;i++)
    {
        puzzle[i] = (int*)malloc(sizeof(int) * 9);
        for(int j = 0;j < 9;j++)
            puzzle[i][j] = arrayPuzzle[i][j];
    }
    return puzzle;
}
```

```c
void printPuzzle(int **puzzle)
{
    printf("\n");
    printf("  0 | 1  2  3 | 4  5  6 | 7  8  9 | X\n");
    printf(" ---------------------------------\n");
    for(int i = 0,a = 1;i < 9;i++,a++)
    {
        for(int j = 0;j < 9;j++)
        {
            if(j == 0)
                printf(" %d |", a);
            else if(j%3 == 0)
                printf("|");
            printf(" %d ", puzzle[i][j]);
            if(j == 8)
                printf("|");
        }
        printf("\n");
        if((i + 1) % 3 == 0)
            printf(" ---------------------------------\n");
    }
    printf(" Y\n");
}
```

```c
int checkAvailable(int **puzzle,int *row,int *column)
{
    for(int i = 0;i < 9;i++)
    {
        for(int j = 0;j < 9;j++)
        {
            if(puzzle[i][j] == 0)
            {
                *row = i;
                *column  = j;
                return 1;
            }
        }
    }
    return 0;
}
```

```c
int checkBox(int **puzzle,int row,int column,int val)
{
    for(int i = 0;i < 9;i++)
    {
        if(puzzle[i][column] == val)
            return 0;
    }
    for(int j = 0;j < 9;j++)
    {
        if(puzzle[row][j] == val)
            return 0;
    }
    int squareRow;
    int squareColumn;
    squareRow = row - row % 3;
    squareColumn = column - column % 3;
    for(int i = 0;i < 3;i++)
    {
        for(int j = 0;j < 3;j++)
        {
            if(puzzle[squareRow + i][squareColumn + j] == val)
                return 0;
        }
    }
    return 1;
}
```

```c
int solvePuzzle(int **puzzle)
{
    int i;
    int j;
    if(!checkAvailable(puzzle,&i,&j))
        return 1;
    for(int val = 1;val < 10;val++)
    {
        if(checkBox(puzzle,i,j,val))
        {
            puzzle[i][j] = val;
            if(solvePuzzle(puzzle))
                return 1;
            else
                puzzle[i][j] = 0;
        }
    }
    return 0;
}
int **copyPuzzle(int **puzzle)
{
    int **newPuzzle;
    newPuzzle = (int**)malloc(sizeof(int*) * 9);
    for (int i = 0;i < 9;i++)
    {
        newPuzzle[i] = (int*)malloc(sizeof(int) * 9);
        for(int j = 0;j < 9;j++)
            newPuzzle[i][j] = puzzle[i][j];
    }
    return newPuzzle;
}
```

```c
void userChoice(int **userPuzzle,int **tempPuzzle)
{
    int positionX;
    int positionY;
    int userVal;
    char c;
    int i;
    int j;
    while(1)
    {
        if(!checkAvailable(userPuzzle,&i,&j))
        {
            printf("\nsuccessfully solved the puzzle!\n");
            return;
        }
        while(1)
        {
            printf("\npress enter to continue or press q to quit\n");
            c = getchar();
            if((c == 'q') || (c == 'Q'))
            {
                getchar();
                solvePuzzle(userPuzzle);
                printf("\nsolved puzzle:\n");
                printPuzzle(userPuzzle);
                return;
            }
            else if((c != '\n') && (c != 'q'))
                getchar();
            else
                break;
        }
        printf("\nenter coordinate for the square to insert value as X Y:\n");
        scanf("%d %d",&positionX,&positionY);
        while(1)
```

```
        while(1)
        {
            if((positionX > 9) || (positionX < 1) || (positionY > 9) || (positionY < 1) || (userPuzzle[positionY - 1][positionX - 1] != 0))
            {
                printf("\ncan't insert value to this coordinate, so enter new coordinate\n");
                scanf("%d %d",&positionX,&positionY);
            }
            else
            {
                positionX -= 1;
                positionY -= 1;
                break;
            }
        }
        printf("\nenter value from 1 to 9\n");
        scanf("%d",&userVal);
        while(1)
        {
            if((userVal > 9) || (userVal < 1))
            {
                printf("\nenter valid value:\n");
                scanf("%d",&userVal);
            }
            else
                break;
        }
        if(checkBox(userPuzzle,positionY,positionX,userVal))
            userPuzzle[positionY][positionX] = userVal;
        else
            printf("\nvalue entered is already present in that row or column or box, so try again\n",positionX + 1,positionY + 1);
        for(int i = 0;i < 9;i++)
```

```
         for(int i = 0;i < 9;i++)
         {
             for(int j = 0;j < 9;j++)
                 tempPuzzle[i][j] = userPuzzle[i][j];
         }
         if(!solvePuzzle(tempPuzzle))
         {
             printf("\nvalue entered is not leading to the solution, so try again\n",positionX + 1,positionY + 1);
             userPuzzle[positionY][positionX] = 0;
         }
         getchar();
         printPuzzle(userPuzzle);
    }
    return;
}
```

sudoku_client.c

```c
#include<stdio.h>
#include<stdlib.h>
#include"sudoku_header.h"
int main()
{
    int **puzzle = createPuzzle();
    int **userPuzzle = copyPuzzle(puzzle);
    int **tempPuzzle = copyPuzzle(puzzle);
    printf("Rules-\n\n");
    printf("The objective of sudoku is to fill a 9x9 grid made of squares such that each row,
    each column and each full 3x3 squares use numbers from 1 to 9 only one time\n");
    printf("Insert numbers in the squares having value 0\n");
    printf("To check solved puzzle press q key\n\n");
    printf("Let's start the game!\n");
    printPuzzle(userPuzzle);
    userChoice(userPuzzle,tempPuzzle);
    free(puzzle);
    free(userPuzzle);
    free(tempPuzzle);
}
```

sudoku_header.h

```c
int **createPuzzle();
void printPuzzle(int **puzzle);
int checkAvailable(int **puzzle,int *row,int *column);
int checkBox(int **puzzle,int row,int column,int val);
int solvePuzzle(int **puzzle);
int **copyPuzzle(int **puzzle);
void userChoice(int **userPuzzle,int **tempPuzzle);
```

```
C:\Users\dell\Desktop\advanced algorithms>gcc sudoku_server.c sudoku_client.c

C:\Users\dell\Desktop\advanced algorithms>a
Rules-

The objective of sudoku is to fill a 9x9 grid made of squares such that each row, each column and each full 3x3 squares use numbers from 1 to 9 only one time
Insert numbers in the squares having value 0
To check solved puzzle press q key

Let's start the game!

 0 | 1  2  3 | 4  5  6 | 7  8  9 | X
 --------------------------------
 1 | 3  0  6 | 5  0  8 | 4  0  0 |
 2 | 5  2  0 | 0  0  0 | 0  0  0 |
 3 | 0  8  7 | 0  0  0 | 0  3  1 |
 --------------------------------
 4 | 0  0  3 | 0  1  0 | 0  8  0 |
 5 | 9  0  0 | 8  6  3 | 0  0  5 |
 6 | 0  5  0 | 0  9  0 | 6  0  0 |
 --------------------------------
 7 | 1  3  0 | 0  0  0 | 2  5  0 |
 8 | 0  0  0 | 0  0  0 | 0  7  4 |
 9 | 0  0  5 | 2  0  6 | 3  0  0 |
 --------------------------------
 Y

press enter to continue or press q to quit
q

solved puzzle:

 0 | 1  2  3 | 4  5  6 | 7  8  9 | X
 --------------------------------
 1 | 3  1  6 | 5  7  8 | 4  9  2 |
 2 | 5  2  9 | 1  3  4 | 7  6  8 |
 3 | 4  8  7 | 6  2  9 | 5  3  1 |
 --------------------------------
 4 | 2  6  3 | 4  1  5 | 9  8  7 |
 5 | 9  7  4 | 8  6  3 | 1  2  5 |
 6 | 8  5  1 | 7  9  2 | 6  4  3 |
 --------------------------------
 7 | 1  3  8 | 9  4  7 | 2  5  6 |
 8 | 6  9  2 | 3  5  1 | 8  7  4 |
 9 | 7  4  5 | 2  8  6 | 3  1  9 |
 --------------------------------
 Y
```

```
press enter to continue or press q to quit


enter coordinate for the square to insert value as X Y:
0 3

can't insert value to this coordinate, so enter new coordinate
7 10

can't insert value to this coordinate, so enter new coordinate
1 1

can't insert value to this coordinate, so enter new coordinate
2 1

enter value from 1 to 9
4

value entered is already present in that row or column or box, so try again

 0 | 1  2  3 | 4  5  6 | 7  8  9 | X
 --------------------------------
 1 | 3  0  6 | 5  0  8 | 4  0  0 |
 2 | 5  2  0 | 0  0  0 | 0  0  0 |
 3 | 0  8  7 | 0  0  0 | 0  3  1 |
 --------------------------------
 4 | 0  0  3 | 0  1  0 | 0  8  0 |
 5 | 9  0  0 | 8  6  3 | 0  0  5 |
 6 | 0  5  0 | 0  9  0 | 6  0  0 |
 --------------------------------
 7 | 1  3  0 | 0  0  0 | 2  5  0 |
 8 | 0  0  0 | 0  0  0 | 0  7  4 |
 9 | 0  0  5 | 2  0  6 | 3  0  0 |
 --------------------------------
 Y
```

```
press enter to continue or press q to quit


enter coordinate for the square to insert value as X Y:
2 1

enter value from 1 to 9
8

value entered is already present in that row or column or box, so try again

 0 | 1  2  3 | 4  5  6 | 7  8  9 | X
 --------------------------------
 1 | 3  0  6 | 5  0  8 | 4  0  0 |
 2 | 5  2  0 | 0  0  0 | 0  0  0 |
 3 | 0  8  7 | 0  0  0 | 0  3  1 |
 --------------------------------
 4 | 0  0  3 | 0  1  0 | 0  8  0 |
 5 | 9  0  0 | 8  6  3 | 0  0  5 |
 6 | 0  5  0 | 0  9  0 | 6  0  0 |
 --------------------------------
 7 | 1  3  0 | 0  0  0 | 2  5  0 |
 8 | 0  0  0 | 0  0  0 | 0  7  4 |
 9 | 0  0  5 | 2  0  6 | 3  0  0 |
 --------------------------------
 Y
```

```
press enter to continue or press q to quit


enter coordinate for the square to insert value as X Y:
2 1

enter value from 1 to 9
1

 0 | 1  2  3 | 4  5  6 | 7  8  9 | X
 --------------------------------
 1 | 3  1  6 | 5  0  8 | 4  0  0 |
 2 | 5  2  0 | 0  0  0 | 0  0  0 |
 3 | 0  8  7 | 0  0  0 | 0  3  1 |
 --------------------------------
 4 | 0  0  3 | 0  1  0 | 0  8  0 |
 5 | 9  0  0 | 8  6  3 | 0  0  5 |
 6 | 0  5  0 | 0  9  0 | 6  0  0 |
 --------------------------------
 7 | 1  3  0 | 0  0  0 | 2  5  0 |
 8 | 0  0  0 | 0  0  0 | 0  7  4 |
 9 | 0  0  5 | 2  0  6 | 3  0  0 |
 --------------------------------
 Y
```

```
press enter to continue or press q to quit


enter coordinate for the square to insert value as X Y:
5 1

enter value from 1 to 9
2

value entered is not leading to the solution, so try again

 0 | 1  2  3 | 4  5  6 | 7  8  9 | X
 --------------------------------
 1 | 3  1  6 | 5  0  8 | 4  0  0 |
 2 | 5  2  0 | 0  0  0 | 0  0  0 |
 3 | 0  8  7 | 0  0  0 | 0  3  1 |
 --------------------------------
 4 | 0  0  3 | 0  1  0 | 0  8  0 |
 5 | 9  0  0 | 8  6  3 | 0  0  5 |
 6 | 0  5  0 | 0  9  0 | 6  0  0 |
 --------------------------------
 7 | 1  3  0 | 0  0  0 | 2  5  0 |
 8 | 0  0  0 | 0  0  0 | 0  7  4 |
 9 | 0  0  5 | 2  0  6 | 3  0  0 |
 --------------------------------
 Y
```

```
press enter to continue or press q to quit


enter coordinate for the square to insert value as X Y:
5 1

enter value from 1 to 9
7

 0 | 1  2  3 | 4  5  6 | 7  8  9 | X
 --------------------------------
 1 | 3  1  6 | 5  7  8 | 4  0  0 |
 2 | 5  2  0 | 0  0  0 | 0  0  0 |
 3 | 0  8  7 | 0  0  0 | 0  3  1 |
 --------------------------------
 4 | 0  0  3 | 0  1  0 | 0  8  0 |
 5 | 9  0  0 | 8  6  3 | 0  0  5 |
 6 | 0  5  0 | 0  9  0 | 6  0  0 |
 --------------------------------
 7 | 1  3  0 | 0  0  0 | 2  5  0 |
 8 | 0  0  0 | 0  0  0 | 0  7  4 |
 9 | 0  0  5 | 2  0  6 | 3  0  0 |
 --------------------------------
 Y
```