```
// Ugur Caglar Submission for Coursework Assignment: Sudoku assignment
// Solution is below functional as JS Code demonstrating the concepts of the lect
ure so far
// Comments are added e.g. TASK 1. demonstrating the steps and logic. You may fin
d the completion of all the Tasks below.
// Main starts at the end of the code on line 300
// Code is available on Github https://github.com/ugggur/SudoKu/blob/main/Pseudok
u.js

// TASK 1.1 Creating a vector with numbers 1,2,3,4 randomly sorted.
var vector = [0,0,0,0];
var puzzle = [[0,0,0,0],[0,0,0,0],[0,0,0,0],[0,0,0,0]];
var theSolution = [[0,0,0,0],[0,0,0,0],[0,0,0,0],[0,0,0,0]];
var hiddenSolution = [[0,0,0,0],[0,0,0,0],[0,0,0,0],[0,0,0,0]];


function createVector()
{
    do
    {
        for(var i=0; i<4; i++)
            {
                vector[i] = Math.floor((Math.random()*4+1));
            }
    }
    while((vector[0] == vector[1])
        ||(vector[0] == vector[2])
        ||(vector[0] == vector[3])
        ||(vector[1] == vector[2])
        ||(vector[1] == vector[3])
        ||(vector[2] == vector[3]))
    return vector;
}

// TASK 1.2 Making the puzzle by layering the vector creted in 1.1 in 4 rows.

function createPuzzle()
{
    for(var i=0; i<4 ; i++)
    {
        for(var j=0; j<4 ; j++)
        {
            puzzle[i][j]=vector[j];
        }
```

```
    }
}

// Control TASK for 1.1 & 1.2 Print out the puzzle in console.

function printPuzzle()
{
    createVector();
    createPuzzle();
    for(var i=0;i<4;i++)
    {
        for(var j=0;j<4;j++)
        {
            console.log(puzzle[i][j]);
        }
    }

}


// TASK 2 Permute Vector.

function permuteVector(v,n)
{
    var newVector = [0,0,0,0];

    for(var i=0;i<4;i++)
    {
        newVector[i] = v[(i+n)%4];
    }
    return newVector;

}

// TASK 3 Permute Puzzle.

function permutePuzzle(p,x,y,z)
{
    var newPuzzle = [[0,0,0,0],[0,0,0,0],[0,0,0,0],[0,0,0,0]];

    newPuzzle[0] = permuteVector(vector,x);
    newPuzzle[1] = permuteVector(vector,y);
    newPuzzle[2] = permuteVector(vector,z);
    newPuzzle[3] = vector;
```

```
    return newPuzzle;

}

// TASK 4 & Task 5 Check Columns.

function checkColumn(p,c)
{
    // creating empty column
    var temp = [0,0,0,0]

    // creating control array to store linear search results
    var control = [false,false,false,false];

    // creating column from puzzle and c (number)
    for(var i = 0; i<4;i++)
    {
        temp[i] = p[i][c];
    }

    // applying linear search and recording results
    for(var m=1;m<=4;m++)
    {
        for(var z=0; z<temp.length;z++)
        {
            if(temp[z] == m)
            {
                control[m-1] = true;
            }
        }

    }

    return (control[0] && control[1] && control[2] && control[3]);

}


function checkAllColumns(p)
{
    // creating control array to store Check Column boolean results
    var control = [false,false,false,false];

    for(i=0;i<4;i++)
    {
```

```
        control[i] = checkColumn(p,i);

    }

    return (control[0] && control[1] && control[2] && control[3]);
}

// TASK 6.1 Check a Grid.


function checkGrid(p, row1, col1,row2,col2)
{
    // creating an empty array to hold grid
    var temp = [0,0,0,0]

    // creating control array to store linear search results
    var control = [false,false,false,false];

    // creating temp index z
    var z = 0;

    //creating the grid from puzzle in temp
    for(var i=row1;i<=row1+1;i++)
    {
        for(var j=col1;j<=col1+1;j++)
        {
            temp[z] = p[i][j];
            z++;
        }
    }

    // linear search of temp
    for(var m=1;m<=4;m++)
    {
        for(var z=0; z<temp.length;z++)
        {
            if(temp[z] == m)
            {
                control[m-1] = true;
            }
        }

    }

    return (control[0] && control[1] && control[2] && control[3]);
```

```
}

// TASK 6.2 Check All Grids.

function checkAllGrids(p)
{
    // creating control array to store linear search results
    var control = [false,false,false,false];

    // creating control index z
    var z = 0;

    for(var i=0;i<=1;i++)
    {
        for(var j=0;j<=1;j++)
        {
            control[z] = (checkGrid(p,0+2*i, 0+2*j,1+2*i,1+2*j));
            z++;
        }
    }

    return (control[0] && control[1] && control[2] && control[3]);

}

// Task 7 Bringing all Together

function makeSolution(p)
{
    for(var i = 0;i<4;i++)
    {
        for(var j = 0;j<4;j++)
        {
            for(var m = 0;m<4;m++)
            {
                var potentialSolution = [[0,0,0,0],[0,0,0,0],[0,0,0,0],[0,0,0,0]]
;
                potentialSolution = permutePuzzle(puzzle,i,j,m);
                if(checkAllColumns(potentialSolution) && checkAllGrids(potentialS
olution))
                {
                    theSolution = potentialSolution;
                    return theSolution;
                }
```

```
            }
        }

    }

}

function printSolution(p)
{
    for(i=0;i<4;i++)
    {
        for(k=0;k<4;k++)
        {
            console.log(p[i][k]);
        }
    }
}

// Task 8 Setting Blank Chars as X
// The below function takes m as a parameter and randomly scans Sudoku solution t
o plants in X's
// The function basically iterates one by one and based on a random probability d
ecides whether to replace the solution sudoku number with X or not.
// Controls added to make sure that the algorithm scans the whole array and requi
red # of X's planted.

function hideChars(m)
{
    var counter = m;
    var random;

    //Blanking the numbers in the solution array
    while(counter>0)
    {
        for(i=0;i<4;i++)
        {
            for(j=0;j<4;j++)
            {
                random = Math.random();

                if(random>0.5)
                {
                    if(counter>0)
                    {
```

```
                        if(hiddenSolution[i][j]!='X')
                        {
                            hiddenSolution[i][j]='X';
                            counter--;
                        }
                        else
                        {
                            continue;
                        }
                    }
                    else if( i*j <= 9)
                    {
                        hiddenSolution[i][j] = theSolution[i][j];
                    }
                    else
                    {
                        continue;
                    }
                }
                else
                {
                    hiddenSolution[i][j] = theSolution[i][j];
                }
            }
        }
    }
}


//TASK 9

// Limitations of the Algorithm: This algorithm uses a single vector to construct
 the Pseudoku. Randomness is achieved by cyclic permutation of vectors for each l
ine. However, this is not fully random as each vector is dependent on the other.
To achieve full randomness, numbers need to be assigned fully independently for e
ach grid box. Otherwise, we are limited to a certain number of permutations hence
 a subset of all possible Pseudoku grids.


// Another Algorithm to overcome limitations: Rather than creating vectors and im
plementing cyclic permutation, random numbers can be assigned to each grid box an
d conditions (sub grids, columns, rows) can be checked afterwards. As the vector
already takes care of row condition, in case we fully randomize the grid contents
, we need to add another control (i.e. row control)
```

```
// MAIN PROGRAM - USER GUIDE

// Pseudoku is displayed on the debug console as numbers
// X denote numbers hidden
// Please update hideChars # to change # of numbers hidden
// If you want to see the not hidden version of the solution please comment (cont
inued)
// printSolution(hiddenSolution) and comment out printSolution(theSolution)
// Thank you

createVector();
createPuzzle();
makeSolution();
hideChars(10);
printSolution(hiddenSolution);
//printSolution(theSolution);
```