Sudoku Assignment

Notes:

The following tasks were written using pseudocode based mainly on JavaScript syntax.

Data Structures Used:

```
class Stack {
    data: number[];
    push(element: number) {
        this.data.push(element);
    }
    pop() {
        return this.data.pop();
    }
    read() {
        let lastItem = this.data[this.data.length-1];
        return lastItem;
    }
    constructor(arr?: number[]) {
        arr ? this.data = arr : this.data = [];
    }
}

class Queue {
    data: number[];
    enqueue(element: number) {
        this.data.push(element);
    }
    dequeue() {
        return this.data.shift();
    }
    read() {
        let firstItem = this.data[0];
        return firstItem;
    }
    constructor(arr: number[]) {
        this.data = arr;
    }
}
```

Task 1

```
function MakeVector(row)
    new Vector puzzle(4)
    for 1 ≤ i ≤ 4 do
        puzzle[i].push(...row);
    end for
    return puzzle;
end function
```

Task 2

```
function PermuteVector(row, p)
    if p = 0 return row;
    q <= new Queue(row);
    temp;
    for 1 ≤ i < p do
        if q.read() != undefined then
            temp <= q.dequeue();
            q.enqueue(temp);
        end if
    end for
    return q.data;
end function
```

Task 3

```
function PermuteRows(puzzle, x, y, z)
    row0 <= [...puzzle[0]];
    row1 <= [...puzzle[1]];
    row2 <= [...puzzle[2]];
    row3 <= [...puzzle[3]];

    return [PermuteVector(row0, x), PermuteVector(row1, y),
PermuteVector(row2, z), row3];
end function
```

Task 4

```
function SearchStack(stack, item)
    foundItem <= false;
    leftOverStack <= new Stack()
    while stack.read() do
        if stack.read() = item then
            stack.pop()
            foundItem <= true;
        end if
        leftOverStack.push(stack.pop())
    end while
    if foundItem then
            return leftOverStack
        end if
        else then
            return false
        end else
end function
```

Task 5

```
function CheckColumn(puzzle, j)
    numbers <= new Stack([1,2,3,4]);
    k <= 0;
    while k < 4 do
        if (!SearchStack(numbers, puzzle[k][j]) then
            return false
        end if
        k++
    end while
    return true;
end function
```

Task 6

```
function ColChecks(puzzle)
    for 1 ≤ j ≤ 4 do
        if !CheckColumn(puzzle, j) then
```

```
            return false;
        end if
    end for
    return true;
end function
```

```
function CheckGrids(puzzle)
    grids <= [
        [puzzle[0][0], puzzle[0][1], puzzle[1][0], puzzle[1][1]],
        [puzzle[0][2], puzzle[0][3], puzzle[1][2], puzzle[1][3]],
        [puzzle[2][0], puzzle[2][1], puzzle[3][0], puzzle[3][1]],
        [puzzle[2][2], puzzle[2][3], puzzle[3][2], puzzle[3][3]],
    ];
    k, i <= 0;
    while i < 4 do
        k <= 0;
        while k < 4 do
            numbers <= new Stack([1,2,3,4]);
            if !SearchStack(numbers, grids[k][i]) then
                return false;
            end if
            k++;
        end while
        i++;
    end while
    return true;
end function
```

Task 7

```
class Pointer {
    data: number[];
    constructor(numbers) {
        this.data = numbers;
    }
}

class Vector {
    data: Pointer[];
    constructor(size: number) {
        this.data = [];
        let i = 0;
```

```
        while(i < size) {
            this.data.push(new Pointer([1,2,3,4]));
            i++
        }
    }
}
```

Task 8

```
function MakeSolution(row)
    orderedPuzzle <= MakeVector(row);
    x <= 0;
    y <= 1;
    z <= 2;
    permutedPuzzle <= PermuteRows(orderedPuzzle, x, y, z);
    while true do
        if ColChecks(permutedPuzzle) && CheckGrids(permutedPuzzle) then
            break;
        end if
        x <= Math.round(Math.random()*3);
        y <= Math.round(Math.random()*3);
        z <= Math.round(Math.random()*3);
        permutedPuzzle = PermuteRows(permutedPuzzle, x, y, z);
    end while
end function
```

Task 9

```
function SetBlanks(puzzle, n)
    row <= Math.round(Math.random()*4);
    column <= Math.round(Math.random()*4);
    cleanedCells <= 0;
    while cleanedCells < n do
        if puzzle[row][column] != -1 then
            puzzle[row][column] <= -1;
            cleanedCells++;
        end if
    end while
    return puzzle.map(row => row.map(cell => cell === -1 ? '' : cell));
end function
```

Task 10

As it was pointed out, a mayor flaw in this algorithm is the lack of randomness at the moment of permutating the cells of the puzzle. A better way to do this in order to get the broadest set of possible layouts is to randomly sort the numbers instead of permutating them so we are not limited by the first row inserted.