200312082

Task 1:

```
function MakeVector(row)
        new Vector puzzle(4)
        for     1 ≤ j ≤ 4       do
                puzzle[j] ← row
        end for

        return puzzle

end function
```

Task 2:

```
function PermuteVector(row, p)
        if p = 0 then
                return row
        end if

        new Queue q
        for     1 ≤ i ≤ 4       do              (Add all element in row to q)
                ENQUEUE(row[i], q)
        end for

        for     1 ≤ j ≤ p       do              (Permute it)
                ENQUEUE[HEAD[q], q]
                DEQUEUE[q]
        end for

        new Vector newRow(4)
        for     1 ≤ k ≤ 4       do              (Take all items in queue to the newRow vector)
                newRow[k] ← DEQUEUE[q]
        end for

        return newRow

end function
```

Task 3:

```
function PermuteRows(puzzle, x, y, z)
        puzzle[1] ←  PermuteVector(puzzle[1], x)
        puzzle[2] ←  PermuteVector(puzzle[2], y)
        puzzle[3] ←  PermuteVector(puzzle[3], z)
        return puzzle

end function
```

Task 4:

```
function SearchStack(stack, item)
    new Stack s
    itemStored ← 0        (This is used to track the number of times when item was found in the stack)

    while EMPTY[stack] = FALSE        do
        if (TOP(stack) ≠ item) then
            PUSH[POP[stack], s]
        else
            POP[stack]
            itemStored ← itemStored + 1              (item is found increase itemStored)
        end if
    end while

    new Stack orderedStack      (This is used to rearrange the stack in the right order)
    while EMPTY(s) = FALSE do
        PUSH[POP[s], orderedStack]
    end while
```

(itemStored is either zero or greater than one. if it's zero then the item is not stored in the stack hence return FALSE, if it's greater than one duplicate item was found in the stack hence return FALSE)

```
    if (itemStored = 1) then
        return orderedStack
    else
        return FALSE
    end if

end function
```

Task 5:

```
function CheckColumn(puzzle, j)
    new Stack number
    sn ← 4
            (Generate a stack in the order: 1 2 3 4)
    while   sn > 0  do
        PUSH[sn, number]
        sn ←  sn - 1
    end while

    for     1 ≤ k ≤ 4       do
        row ← puzzle[k]
        value ← row[j]
        numbers ← SearchStack(numbers, value)
        if numbers = FALSE
            return FALSE
        end if
    end for
    return TRUE
end function
```

Task 6:

function CheckGrids(puzzle)

(My approach to solving this task is by creating a function: “SearchAGrid” which specifically searches a single grid to ensure that the grid follows the rule. if it does, it return TRUE and FALSE otherwise)

```
    function SearchAGrid(puzzle, startRow, firstColumn, secondColumn)
        currentRow ← startRow
        endRow ← currentRow + 1

        new Stack number
        sn ← 4
                (Generate a stack in the order: 1 2 3 4)
        while   sn > 0  do
            PUSH[sn, number]
            sn ←  sn - 1
        end while

        while currentRow ≤ endRow do
            currentColumn ← firstColumn
            endColumn ← secondColumn + 1

            while currentColumn < endColumn
                value ←  puzzle[currentRow][currentColumn]
                numbers ← SearchStack(numbers, value)
                if numbers = FALSE
                    return FALSE
                end if
                currentColumn ← currentColumn + 1
            end while

            currentRow ← currentRow + 1
        end while
        return TRUE
    end function

    rowIndex  ← 1
    while   rowIndex  ≤ 4    do
        j ← 1
        while    j ≤ 4     do
            checkAGrid ← SearchAGrid(puzzle, rowIndex, j, j + 1)
            if checkAGrid = FALSE then
                return FALSE
            end if
            j ← j + 2
        end while

        rowIndex  ← rowIndex + 2
    end while

    return TRUE

end function
```

Task 7

ArrLin Data Structure

Array: $\boxed{8}$ $\boxed{1}$ head $\rightarrow \boxed{2} \xrightarrow{2} \boxed{4} \xrightarrow{3} \boxed{1} \xrightarrow{4} \boxed{3} \rightarrow$ null

$\boxed{2}$ head $\rightarrow \boxed{1} \xrightarrow{2} \boxed{3} \xrightarrow{3} \boxed{2} \xrightarrow{4} \boxed{4} \rightarrow$ null

$\boxed{3}$ head $\rightarrow \boxed{3} \xrightarrow{2} \boxed{2} \xrightarrow{3} \boxed{4} \xrightarrow{4} \boxed{1} \rightarrow$ null

$\boxed{4}$ head $\rightarrow \boxed{4} \xrightarrow{2} \boxed{1} \xrightarrow{3} \boxed{3} \xrightarrow{4} \boxed{2} \rightarrow$ null

I have approached this task by designing a concrete data structure called ArrLin. ArrLin is a combination of array and linked lists.The whole data is encapsulated in a dynamic array of length, 2 * row of puzzle vector i.e for this puzzle vector it's row is 4 therefore the length of ArrLin is 8. The structure of ArrLin is that row number comes first and linked list follows i.e in this puzzle vector, 1 is the first element and pointer to linked list of all the elements in row 1 follows, same as row 2 and pointer to linked list of all row 2 elements … upto the 4th row as shown above in the diagram.
Each linked list has the following pointers: head, 2, 3, 4. Head points to the first element in a row, 2 to second element, 3 to third element and 4 to the fourth element in a row when the pointers are dereferenced.

ArrLin has some internal method that converts columns to have the internal workings pointer described above.

The concrete data structure allows only odd numbers to be indexed on it's interface.
For instance, ArrLin[3] is possible but ArrLin[4] is not possible.
This is because "even" indexes are the value to "odd" index (as in row), as a result of this the concrete data structure has a mechanism for determing it's element: row and column.

Each indexed(odd) value of ArrLin returns a linked list with pointers from 1 to 4. The ArrLin data structure has some internal workings that map the pointers to head, 2, 3, 4.

read[ArrLin[row], column] returns a value at the specified row and column

Length[ArrLin] returns the length of ArrLin, by internal workings of Array[0] (Dynamic Array)

Store[row, column, value] of ArrLin is equivalent to this internal working on the linked list:
write[value, read[ArrLin[row], column]]

The pseudocode below can be use to traverse ArrLin.

```
row ← 1
while   row < Length[ArrLin]   do
        for 1 ≤ column ≤ 4 do
                (access puzzle vector element with row and column variables)
        end for
        row ← 1 + 2
end while
```

```
function ColChecks(puzzle)
        for 1 ≤ j ≤ 4 do
                if CheckColumn(puzzle, j) = FALSE then
                        return FALSE
                end if
        end for
        return TRUE
end function
```

Task 8:

```
function MakeSolution(row)
          puzzle ← MakeVector(row)
          solution ← puzzle

          (The three loops below generate all possible values of x y z for  PermuteRows function)

          for       1 ≤ i < 4          do

                    for       1 ≤ j < 4          do

                              for       1 ≤ k < 4          do

                                        solution ← PermuteRows(puzzle, i, j, k)
                                        gridCheck ← CheckGrids(solution)
                                        checkCol   ← ColChecks(solution)

                                        if gridCheck = TRUE ^ checkCol = TRUE
                                                  return solution
                                        end if

                              end for

                    end for

          end for

          return solution

end function
```

Task 9:

A method that can be used for setting values to be blank characters in the elements of the output of MakeSolution is by using a function that can generate a random number given two integers say 1 and 4, the function generates a number from 1 to 4 (both inclusive).
The number will be generated twice, one will be used for row and the other for column to get a particular value out of the puzzle, the value at the position of generated number will be set to “X” which indicates blank. The process will be repeated for up to **n** times input parameters of blank characters desired. Attached below is the pseudocode diagram:

```
function SetBlanks(puzzle, n)
        maximumBlank ← LENGTH[puzzle] * LENGTH[puzzle] (Determines highest possible number of blanks for the puzzle)
        if   n > maximumBlank    then
                n ← maximumBlank (Restrict value of n from being higher the highest possible number of blanks)
        end if

        blanks ← 0                                   (Track number of blank in the puzzle, initially 0)
        function SetBlank()                          (The function sets a single blank in to the puzzle)
                randomRow ← genRandom(1, 4)          (genRandom(1, 4) returns number from 1 to 4 - both inclusive)
                randomColumn ← genRandom(1, 4)

        (If randomly selected value is already blank, the function calls itself recursively to re-generate the number)

                if puzzle[randomRow][randomColumn] = X
                        SetBlank()
                else
                        puzzle[randomRow][randomColumn] ←X
                        blanks ← blanks + 1
                end if

        end function

        while blanks < n
                SetBlank()
        end while

        return puzzle

end function
```

Task 10:

The algorithm is limited in it's specific function because it depends on the input value to MakeSolution for uniqueness because when permuting the rows of the vector only the first to the third row are cyclically permuted, when input that are not unique is passed to MakeSolution there is a probability that the algorithm generates a wrong solved pseudoku puzzle.

Likewise, the algorithm only generates one solved pseudoku puzzle when the conditions are satisfied (Grid and Column) and not all possible solutions.

Also, the time complexity of the algorithm is more than big O of n square as aresult of nested looping in most of the functions implementation.

In addition, the algorithm is not in any way extensible to solve other pseudoku puzzle with length greater than 4 because of hoe the functions are designed.

For instance, the implementation of CheckGrids is tricky and is not re-usable for other puzzles with length greater than 4 despite the different loops in the function; the same thing applies to ColChecks.

The algorithm also makes use of static implementation whereas dynamic implementation can be done e.g ColChecks function (image below) stactically iterate j upto 4 instead of LENGTH[puzzle]

```
function ColChecks(puzzle)
        for 1 ≤ j ≤ 4 do
                if CheckColumn(puzzle, j) = FALSE then
                        return FALSE
                end if
        end for
        return TRUE
end function
```