Coursework Assignment: Sudoku assignment

Algorithms and Data Structure I

The 9 tasks in this assignment make up the Sudoku coursework assignment. The tasks in this assignment consist, in the main, of functions or lines of code to be written in pseudocode. Because your solutions should be written in pseudocode, marks will not be deducted for small syntax errors as long as the pseudocode can be understood by a human. Having said that, it is highly recommended that you use the pseudocode conventions given in this module.

There are 50 marks available in total for this assignment

Background: Sudoku and Pseudoku

A Sudoku puzzle consists of 9-by-9 grid of squares, some of them blank, some of them having integers from 1 to 9. A typical Sudoku puzzle will then look something like this:

		3		5		8	9	7
8				1	2	3		
	9			3		4	2	1
9	3	6			1	7		
		1				5		
		7	2			1	8	6
3	4	2		6			7	
		9	8	2				3
5	6	8		7		2		

To solve this puzzle, all the squares must be filled with numbers from 1 to 9 such that the following are satisfied:

1. every row has all integers from 1 to 9 (with each appearing only once)
2. every column has all integers from 1 to 9 (with each appearing only once)
3. every 3-by-3 sub-grid, or block (with bold outlines around them going from top-left to bottom-right) has all integers from 1 to 9

In this coursework, we won't be generating and solving Sudoku puzzles exactly, but a simplified version of Sudoku puzzles, which I will call Pseudoku puzzles – pronounced the same. In a Pseudoku puzzle, we now have a 4-by-4 grid of squares, some of them blank, some of them having integers from 1 to 4. A typical Pseudoku puzzle will look like this:

	4	**1**	
		2	
3			
	1		**2**

Now to solve this puzzle, all the squares must be filled with numbers from 1 to 4 such that the following are satisfied:

1. every row has all integers from 1 to 4 (with each appearing only once)
2. every column has all integers from 1 to 4 (with each appearing only once)

3. every 2-by-2 sub-grid, or block (with bold outlines around them going from top-left to bottom-right) has all integers from 1 to 4

These three conditions will be called the Pseudoku conditions. For the above Pseudoku puzzle, a solution is:

2	4	1	3
1	3	2	4
3	2	4	1
4	1	3	2

The goal of the whole Sudoku assignment is to produce an algorithm that can generate Pseudoku puzzles. It is important to emphasise that a Pseudoku puzzle is specifically a 4-by-4 puzzle as above, and not 9-by-9, or any other size. So when we refer to Pseudoku puzzles, we are specifically thinking of these 4-by-4 puzzles.

Generating Pseudoku puzzles

You are going to produce an algorithm that generates a Pseudoku puzzle. This algorithm starts with a vector of four elements, with all the integers 1 to 4 in any particular order, e.g. 1,2,3,4 or 4,1,3,2. In addition to this vector, the algorithm also starts with an integer n, which is going to be the number of blank spaces in the generated puzzle. This whole process will be more modular, i.e. the algorithm will combine multiple, smaller algorithms.

The big picture of the algorithm is to construct a solved Pseudoku puzzle by duplicating the input vector mentioned earlier. Then from the solved puzzle, the algorithm will remove numbers and replace them with blank entries to give an unsolved puzzle. These are the main steps in the algorithm:

1. Get the input vector called `row` and number `n`
2. Create a vector of four elements called `puzzle`, where each element of `puzzle` is itself the vector `row`
3. Cyclically permute the elements in each element of `puzzle` so that `puzzle` satisfies the Pseudoku conditions
4. Remove elements in each element of `puzzle` to leave blank spaces, and complete the puzzle

Steps 1 and 2 in this algorithm will involve writing functions in pseudocode and vector operations. Step 3 will, in addition to the tools in Steps 1 and 2, involve using queue operations and adapting the Linear Search algorithm. Step 4 can involve writing a function in pseudocode, or by some other means.

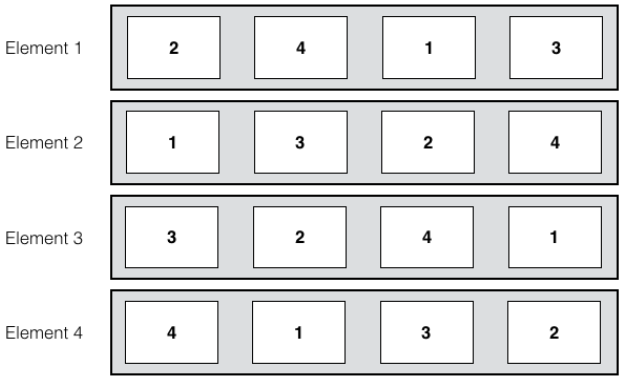
In the following sections, there will be some introductory information to set out the problem that needs to be solved, along with the statement of task.

The puzzle format

As mentioned earlier, we will start with a completed puzzle stored in a four-element vector called `puzzle` where every element is itself a four-element vector, such as

2	4	1	3
1	3	2	4
3	2	4	1
4	1	3	2

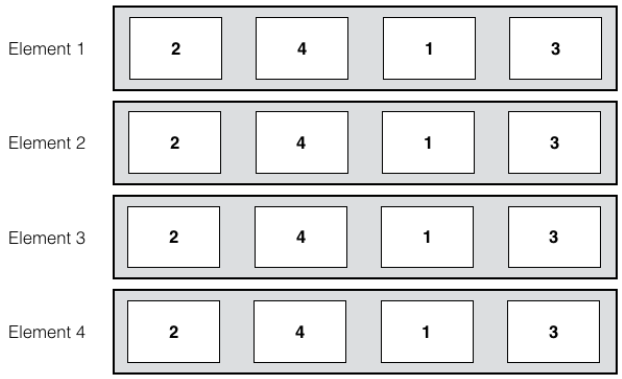
Each row of the puzzle will correspond to an element of a vector, e.g. the first row of the Pseudoku puzzle will be stored as a four-element vector, which itself is an element of a four-element vector. Therefore, this completed Pseudoku puzzle is represented by the following vector:



We could make this vector by initiating a four-element vector, with each element being empty, and then assign a vector to each element. Here is a snippet of the pseudocode for this process but for only the first row of the puzzle:

```
new Vector row(4)
row[1] ← 2
row[2] ← 4
row[3] ← 1
row[4] ← 3
new Vector puzzle(4)
puzzle[1]← row
```

The goal of the algorithm in this coursework is to generate an unsolved Pseudoku puzzle from a row of four numbers. The first step in the process is to make all four elements of a four-element vector to be the same, and this element will be a four-element vector. For example, given a four-element vector with the numbers 2, 4, 1, 3, we produce the following vector:



Your first task is to write a function in pseudocode that will carry out this process.

Task 1: Complete the following function template:

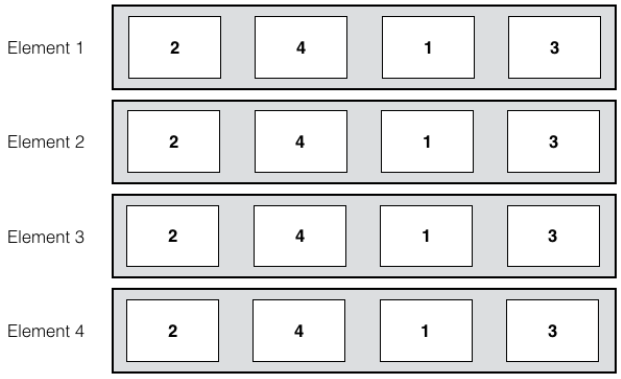
```
function MakeVector(row)
    new Vector puzzle(4)
    ...
end function
```

This function should take a four-element vector called *row* as an input parameter and return a vector of four elements called *puzzle*: each element of *puzzle* should contain the four-element vector *row*. Complete this function.

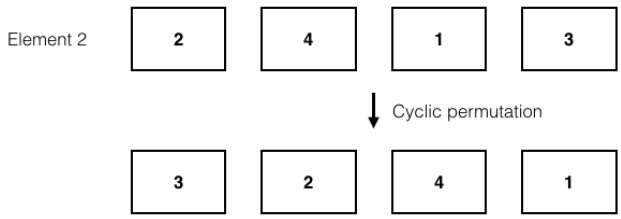
[4 marks]

Cyclic permutation of row vectors

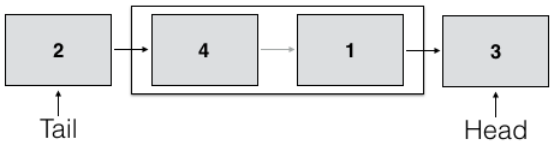
Consider the following vector:



This does not satisfy the Pseudoku conditions since in each column only one number appears. The algorithm for generating Pseudoku puzzles will cyclically permute the elements in each row vector until the numbers in all the rows satisfy the Pseudoku conditions. A cyclic permutation of each row will shift all values of the elements one place to the left (or the right) with the value at the end going to the other end. For example, for the second element in the vector above, if we cyclically permute all elements one place to the right we will have:



Given a four-element vector and an integer p between 0 and 3 (inclusive) we want to write a function to cyclically permute the values in the vector by p elements to the right. An elegant way to do this is to use the queue abstract data structure. All values in a vector will be enqueued to an empty queue, e.g. the first row vector in the vector above will give the following queue:



To cyclically permute all values one place to the right we copy the value stored at the head into a variable called *store*, dequeue the queue, and then enqueue the value stored in *store* to the queue. This process will then give the following queue:

To cyclically permute the values we can just repeat this process of storing the head value, dequeueing and then enqueueing the stored value multiple times. When we are finished this process we then just copy the values stored in the queue to a vector and return this vector. The next task is to formalise this process into a function in pseudocode.

Task 2: Complete the following function template:

```
function PermuteVector(row, p)
    new Queue q
    ...
end function
```

This function should take a four-element vector called *row* as an input parameter and return that vector but with its values cyclically permuted by *p* elements to the right: *p* should be a number between 0 and 3 (inclusive). To be able to get full marks you need to use the queue abstract data structure appropriately as outlined above.

[6 marks]

The function PermuteVector, once completed, will only cyclically permute one vector. The next task is to take a vector *puzzle* of the form returned by the function MakeVector, and apply PermuteVector to each of the elements of *puzzle*. That is, given vector *puzzle* and three numbers *x*, *y* and *z*, elements 1, 2 and 3 of *puzzle* will be cyclically permuted *x*, *y* and *z* places to the right respectively.

Task 3: Complete the following function template:

```
function PermuteRow(puzzle, x, y, z)
    ...
end function
```

This function should take a four-element vector called *puzzle*, which will be of the form of the output of MakeVector as an input parameter, as well as three integers *x*, *y* and *z*. The function will return *puzzle* but with elements *puzzle[1]*, *puzzle[2]* and *puzzle[3]* cyclically permuted by *x*, *y* and *z* elements respectively to the right: *x*, *y* and *z* should all be numbers between 0 and 3 (inclusive). To be able to get full marks you should call the function PermuteVector appropriately. HINT: You do *not* need to loop over integers *x*, *y* and *z*.

[4 marks]

Checking the Pseudoku conditions

The next step in constructing the algorithm is to write methods to decide if the Pseudoku conditions are satisfied. If we start with the output of the function MakeVector(*row*), then all of the row conditions are satisfied as long as *row* is a four-element vector with the numbers 1 to 4 only appearing once. However, the column conditions are not satisfied: only one number appears in each column (four times). The 2-by-2 sub-grid conditions are also not satisfied: in each sub-grid only two numbers appear (twice). In this part of the coursework we will write four functions: one function to check the conditions for a single column of *puzzle*; one function to check all columns; and one function to check the conditions for all 2-by-2 sub-grids.

In order to test whether all numbers from 1 to 4 appear in a column, we will use the Linear Search Algorithm repeatedly. That is, first we construct a vector out of the four values in a column, and then we check if all the numbers from 1 to 4 appear in that vector. The relevant Linear Search Algorithm for an input vector and the value *item* is written as:

```
function LinearSearch(vector, item)
    for 1≤ i≤LENGTH[vector] do
        if vector[i]=item then
            return TRUE
        end if
```

```
    end for
    return FALSE
end function
```

Task 4: Complete the following function template:

```
function CheckColumn(puzzle, j)
    new Vector temp(4)
    ...
end function
```

This function should take a four-element vector called *puzzle*, which will be of the form of the output of MakeVector as an input parameter, as well an integer *j* that will be a number between 1 and 4 (inclusive). This function should construct a four-element vector called *temp*: each element *temp*[*j*] will be the *j*th element value of each row vector in *puzzle*. Once the vector is constructed, apply LinearSearch(*temp*, *k*) for each integer $1 \leq k \leq 4$. If all numbers *k* are found in *temp* then return TRUE, otherwise return FALSE. To be able to get full marks you should call the function LinearSearch appropriately.

[6 marks]

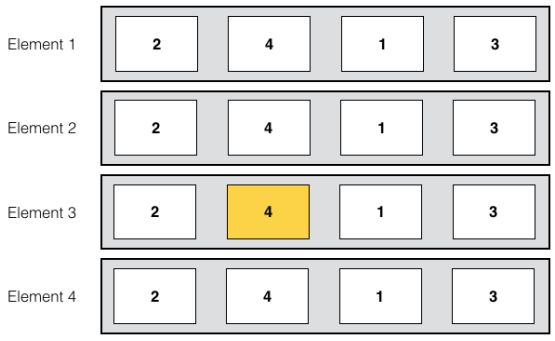
Task 5: Complete the following function template:

```
function ColCheck(puzzle)
    ...
end function
```

This function should take a four-element vector called *puzzle*, which will be of the form of the output of MakeVector as an input parameter. This should return TRUE if and only if CheckColumn(*puzzle*, *j*) returns TRUE for all *j*. To be able to get full marks you should call the function CheckColumn appropriately.

[4 marks]

The next set of conditions to check is to see if all integers from 1 to 4 appear in the 2-by-2 sub-grids. We need a convenient way to refer to the sub-grids. We will use a coordinate system of (*row*,*col*) for the four-element vector *puzzle*: *row* is the index of the element of *puzzle* that we care about, and *col* is the index of the element in *puzzle*[*row*] that we care about. Consider the following vector:



The coordinates of the element in yellow are (3,2), for example. Using this coordinate system to refer to the 2-by-2 sub-grids, the top-left sub-grid will consist of the elements (1,1), (1,2), (2,1) and (2,2): the top-left element is at (1,1) and the bottom-right is at (2,2). We can now use this system to describe the 2-by-2 sub-grids. This can be done by just specifying the coordinates of the top-left element and the bottom-right elements of the 2-by-2 sub-grid, which will be specified by coordinates (*row1*,*col1*) and (*row2*,*col2*) respectively.

Consider the following pseudocode:

```
function CheckGrids(puzzle)
    for 0 ≤ i ≤ 1 do
        for 0 ≤ j ≤ 1 do
            if CheckGrid(puzzle, 1 + 2i, 1 + 2j, 2 + 2i, 2 + 2j) = FALSE then
                return FALSE
            end if
        end for
    end for
    return TRUE
end function
```

This pseudocode will check if 2-by-2 sub-grids defined by the top-left and bottom-right coordinates (*row1*,*col1*) and (*row2*,*col2*) respectively return FALSE if we call the function CheckGrid. This function should check whether all four elements of a 2-by-2 sub-grid defined by (*row1*,*col1*) and (*row2*,*col2*) contain all integers from 1 to 4: it should return TRUE if it so, and FALSE otherwise. In the next task, the goal is write the function called CheckGrid, which will take the row and column coordinates of the top-left and bottom-right elements of a 2-by-2 sub-grid and return a Boolean.

Task 6: Complete the following function template:

```
function CheckGrid(puzzle, row1, col1, row2, col2)
    ...
end function
```

This function will take the four-element vector *puzzle* and numbers *puzzle*, *row1*, *col1*, *row2*, *col2* as arguments. The numbers *row1*, *col1* and *row2*, *col2* define the top-left and bottom-right coordinates of a 2-by-2 subgrid respectively. The function, when completed, should return TRUE if all numbers from 1 to 4 are stored in this sub-grid, and FALSE otherwise. You can assume all the functions in the previous tasks are completed and defined, and you may call these functions.

[6 marks]

Putting everything together

We now have all the ingredients to generate a solved puzzle given a row vector called *row*. The next task will involve generating the initial four-element vector called *puzzle* from *row* using MakeVector(*row*), trying all cyclic permutations (using PermuteRow(*puzzle*, *x*, *y*, *z*) for all combinations of *x*, *y* and *z*) to see if the returned vector returns TRUE for both CheckGrids and ColCheck.

Task 7: Complete the following function template:

```
function MakeSolution(row)
    ...
end function
```

This function will take the four-element vector *row* as input, which is the same input for the function MakeVector. The function should return a solved Pseudoku puzzle such that all column and sub-grid Pseudoku conditions are satisfied. The function will generate a vector using MakeVector(*row*), then try cyclic permutations on this vector using PermuteRow(*puzzle*, *x*, *y*, *z*) until a set of permutations is found such that all Pseudoku conditions are satisfied (checked using CheckGrids and ColCheck). To be able to get full marks you should call the

functions MakeVector, PermuteRow, CheckGrids and ColCheck.

[6 marks]

All of the methods above will just produce a solved Pseudoku puzzle. In order to produce a proper Pseudoku puzzle, numbers will need to be removed from the output of MakeSolution and replaced with a blank character. To complete the algorithm for generating Pseudoku puzzles, in addition to the input vector *row*, we have the integer *n*, which will stipulate the number of blank entries in the final puzzle.

Task 8: Describe a method for setting values to be blank characters in the elements of the output of MakeSolution. You can describe the method in words, use pseudocode, or use a flowchart. The method should take the number *n* as an input parameter and set *n* values to be blank characters. You do not need to go into great detail as long as the method makes sense. Maximum word count for the whole task (excluding diagrams): 200 words.

[4 marks]

Analysing the algorithm

In the next task, the goal is to analyse the algorithm in this assignment. The algorithm to generate Pseudoku puzzles outlined here might not produce all possibly valid Pseudoku puzzles. Remember that, generally speaking, the algorithm works by cyclically permuting several rows of a vector until the Pseudoku conditions are satisfied. In the next task you should aim to identify all of the weaknesses you can think of in this algorithm and propose how you could amend the current algorithm or design a new one to overcome these weaknesses.

Task 9: Describe and very briefly explain the limitations of the algorithm in this assignment [4 marks]. Then you should outline another algorithm, which could be a modification of the existing one, or something completely new, that overcomes the limitations of the algorithm in this assignment [6 marks]. Maximum word count for the whole task: 600 words.

[10 marks]