Mid-term Assessment: Sudoku assignment

Algorithms and Data Structure I

There are 10 tasks in this assignment. The tasks consist, in the main, of functions to be written in pseudocode. For those specific tasks asking for functions should be written in pseudocode, and no explanation of the pseudocode is needed. 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. For example, unlike in programming languages, do not worry about whether functions call by value or call by reference (you can assume either). Having said that, it is highly recommended that you use the pseudocode conventions given in this module.

In tasks 7 and 9, you will be asked to give written answers with the possibility of including diagrams. In these cases written explanations are expected, and not just pseudocode.

There are 50 marks available in total for this assignment

Background: Sudoku and Pseudoku

A Sudoku puzzle consists of a 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 (with each appearing only once)

In this coursework, we won't be generating 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 with some of the squares having integers from 1 to 4. A typical Pseudoku puzzle will look like this:

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

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 (with each appearing only once)

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 this 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. 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 vectors in each element of `puzzle` so that `puzzle` satisfies the Pseudoku conditions
4. Remove values in elements 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 also involve using queue operations and adapting the Linear Search algorithm. A pseudocode function for Step 4 will be presented, which you will analyse.

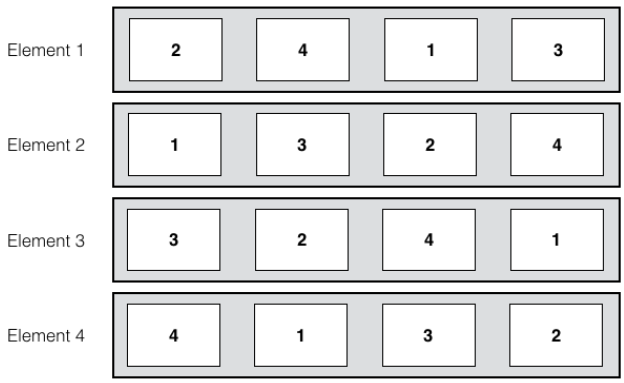
In the following sections, there will be some introductory information to set out the problem that needs to be solved, along with the statement of task.

The puzzle format

As mentioned earlier, we will start with a completed puzzle stored in a four-element vector where every element is itself a four-element vector. If we take the completed puzzle from earlier

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

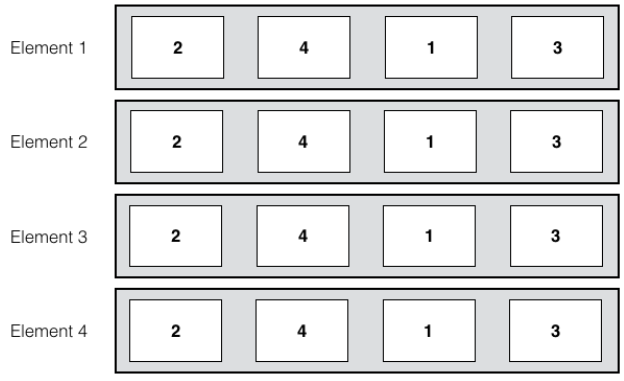
Each row of the puzzle will correspond to an element of a vector, e.g. the first row of the Pseudoku puzzle will be stored as a four-element vector, which itself is an element of a four-element vector. Therefore, this completed Pseudoku puzzle is represented by the following vector:



We can create this vector by first creating a vector of length four, and each element can have a vector assigned to it. Here is a snippet of the pseudocode for this process for the first row of the puzzle:

new Vector *row*(4)
row[1] $\leftarrow$ 2
row[2] $\leftarrow$ 4
row[3] $\leftarrow$ 1
row[4] $\leftarrow$ 3
new Vector *puzzle*(4)
puzzle[1]$\leftarrow$ *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 to write a function in pseudocode that will carry out this process.

Task 1: Complete the following function template:

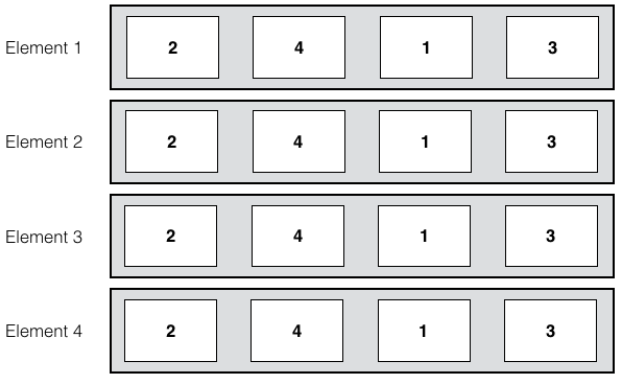
```
function MakeVector(row)
    new Vector puzzle(4)
    ...
end function
```

This function should take a four-element vector called *row* as an input parameter and return a vector of length four where each element stores *row*.

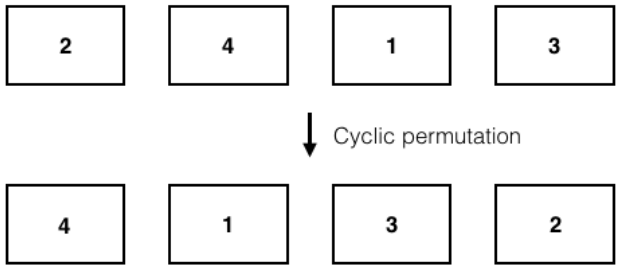
[4 marks]

Cyclic permutation of row vectors

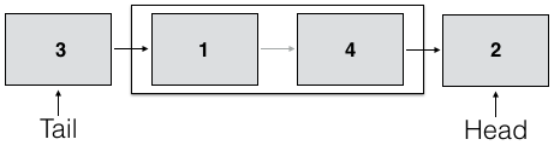
Consider the following vector:



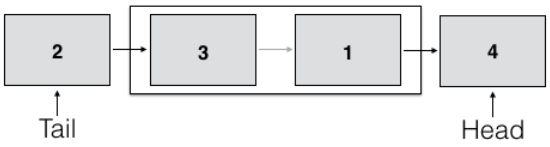
This does not satisfy the Pseudoku conditions since in each column only one number appears. The algorithm for generating Pseudoku puzzles will cyclically permute the elements in the rows until the Pseudoku conditions are satisfied. A cyclic permutation of each row will shift all values of the elements 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 left 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 left. 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 from left to right, e.g. the first row vector in the vector above will give the following queue:



To cyclically permute all values one place to the left we enqueue the value stored at the head, and then dequeue the queue. This process will then give the following queue:



To cyclically permute the values we can just repeat multiple times this process of enqueueing the value at the head and then dequeueing. When we are finished with this process we then just copy the values stored in the queue to our original 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)
    if p = 0 then
        return row
    end if
    new Queue q
    ...
end function
```

This function should return that input vector *row* with its values cyclically permuted by *p* elements to the left: *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* and apply PermuteVector to its bottom three elements. In particular, given vector *puzzle* then elements 2, 3 and 4 of *puzzle* will be cyclically permuted *x*, *y* and *z* places to the left respectively.

Task 3: Complete the following function template:

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

The function should return the input vector *puzzle* but with elements *puzzle[2]*, *puzzle[3]* and *puzzle[4]* cyclically permuted by *x*, *y* and *z* elements respectively to the left: *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. You do *not* need to loop over integers *x*, *y* and *z*.

[4 marks]

Checking the Pseudoku conditions

The next element of the algorithm is 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 two functions to check the column conditions: one function to check the conditions for a single column of *puzzle* and one function to check all columns. To check the sub-grid conditions, we can reuse the functions that check the column conditions, all we need to do

is make convert columns into sub-grids: the third function then to check the Pseudoku conditions will convert a puzzle vector into another vector, but the columns store the values that were stored in the 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
```

The input parameters are a four-element vector called *puzzle* 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 *i*th element *temp*[*i*] will be assigned the *j*th value of the *i*th row *puzzle*[i]. 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]

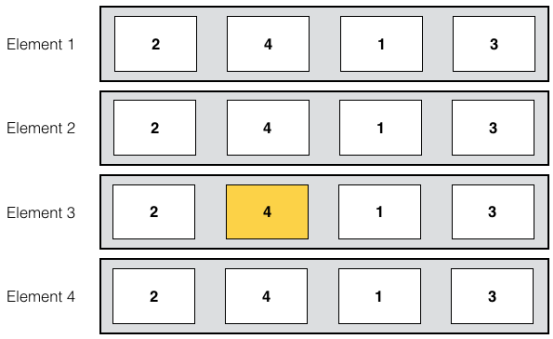
Task 5: Complete the following function template:

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

This 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.

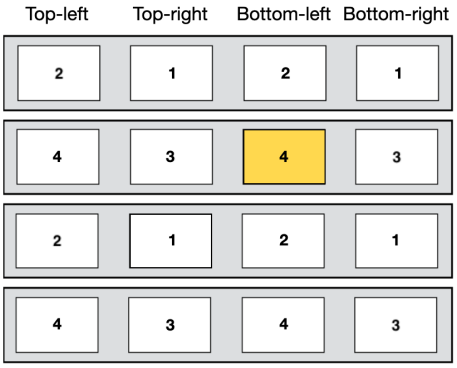
[4 marks]

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* as produced by MakeVector: *row* is the number of the element of *puzzle* that we care about, and *col* is the number 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 make a new vector from the sub-grid elements. This will be done by just specifying the coordinates of the top-left element and the top-right element, which will be specified by coordinates (*row1*,*col1*) and (*row2*,*col2*) respectively. Then to refer to element (*row1*,*col1*) of *puzzle* we use the notation *puzzle*[*row1*,*col1*].

To check the two-by-two sub-grid Pseudoku conditions, all we need is a function that creates a new puzzle vector where the values stored in each sub-grid are now stored in the columns. If we take the example above where each row stores the numbers 2, 4, 1, 3 we can create a new puzzle vector of the form:



As can be seen, each column stores the numbers of in a particular sub-grid, e.g. the top-left sub-grid consisting of elements (1,1), (1,2), (2,1) and (2,2) is stored in the first column. The element highlighted in yellow in the original vector is also highlighted in yellow in this new vector; this is to show the location of the value in this new puzzle vector. Your next task is to devise a method for taking a puzzle vector and making a new puzzle vector where the values in the sub-grids are stored in the columns.

Task 6: Complete the following function template:

```
function ColsFromGrids(puzzle)
    ...
end function
```

This function should return a puzzle vector (of length four) where each column exclusively stores the numbers in one of the sub-grids. It does not matter too much which column stores the numbers from which sub-grid, but a suggestion is the following: the numbers in the top-left and top-right sub-grids are stored in columns 1 and 2, and the numbers in the bottom-left and bottom-right sub-grids are stored in columns 3 and 4.

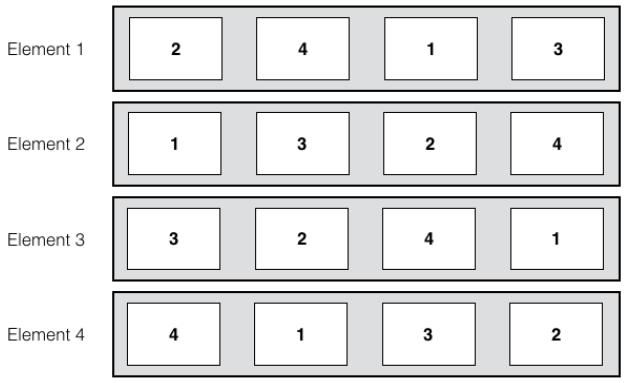
HINT: An integer j where $0 \leq j \leq 3$ can be converted into two numbers (r, c) where r = floor(j/2) and c = i mod 2 so that j = 0 gives (0, 0), j = 1 gives (0, 1), j = 2 gives (1, 0) and j = 3 gives (1, 1). You can use this as the basis to loop over sub-grids.

[5 marks]

Implementing the puzzle vectors

A vector is an abstract data structure. When a vector just stores only a number in each element, we can straightforwardly implement the vector with an array where each element of the array stores a number. However, the vectors considered in this assignment store vectors in their elements. The next task in the assignment is to design a concrete data structure for implementing the puzzle vectors representing Pseudoku puzzles; importantly, each element of the concrete data structure can only store a number or a pointer. Therefore, you could try an implementation based on arrays or linked lists, or a hybrid of both.

Task 7: Consider the following puzzle vector:



Design and explain a concrete data structure that implements this puzzle vector. The data structure must only consist of elements that can store an integer or a pointer to another element or null - elements can be indexed if they are contiguous in memory as with an array. You can draw the data structure and explain how the allowed operations of vectors are implemented on this concrete data structure - additional pointers can be created to traverse lists. One approach could be to use arrays, or linked lists, or another approach completely.

[6 marks]

This seventh task is intentionally vague to allow for all sorts of creative solutions, as long as they are well explained.

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 PermuteRows(*puzzle*, *x*, *y*, *z*) for all combinations of *x*, *y* and *z*) until all the Pseudoku conditions are satisfied.

Task 8: Complete the following function template:

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

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 the returned vector using PermuteRows(*puzzle*, *x*, *y*, *z*) until a set of permutations is found such that all Pseudoku conditions are satisfied; the Pseudoku conditions will be checked using the ColCheck and ColsFromGrids functions. To get full marks you should call the functions MakeVector, PermuteRows, ColsFromGrids and ColCheck.

[5 marks]

All of the methods above will just produce a solved Pseudoku puzzle. In order to produce a proper Pseudoku puzzle, numbers will be deleted from the output of MakeSolution and the elements will store nothing (or 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.

One method for generating blank entries is to generate two random numbers between 1 and 4, called *row* and *col* respectively. Imagine that we have a pseudocode function, called RandomNumber() that, when called, returns a random number between 1 and 4 (inclusive). In the following function we will call this function to generate co-ordinates that will tell us which elements to make blank. Consider the following function that takes the number n as input, which should be the number of blank entries in the final puzzle:

```
function MakeBlanks(puzzle, n)
    for 1≤ i ≤ n do
        row ← RandomNumber()
        col ← RandomNumber()
        puzzle[row][col] ← ""
    end for
    return puzzle
end function
```

Here the value "" represents an element storing a blank entry.

Analysing the algorithm

In the next task, we will start analysing the algorithm in this assignment. First, there is a problem with the function MakeBlanks and your goal in the next task is to describe what the problem is.

Task 9: Explain why function MakeBlanks might not return the vector *puzzle* with n blank entries. Very briefly describe in words how you could fix the function so it works as it should.

[4 marks]

Now assume that MakeBlanks has been fixed and will always return a puzzle with n blank entries. The algorithm to generate Pseudoku puzzles outlined here might not produce all possibly valid Pseudoku puzzles. Have a think why this might be the case and then address the final task.

Task 10: Write a new function in pseudocode that can be used to generate Pseudoku puzzles that cannot be generated by the method outlined in this assignment. Describe how this function would be integrated into the current algorithm of this assignment.

[6 marks]