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. 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 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, 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 (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 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 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, in addition to the tools in Steps 1 and 2, involve using queue operations and adapting the Linear Search algorithm. A pseudocode function for Step 4 will be presented, which you will analyse.

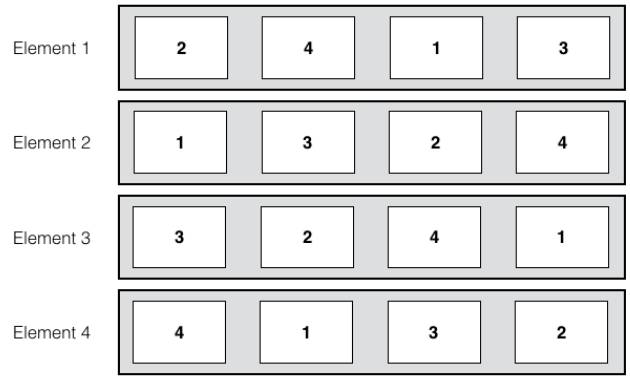
In the following sections, there will be some introductory information to set out the problem that needs to be solved, along with the statement of task.

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

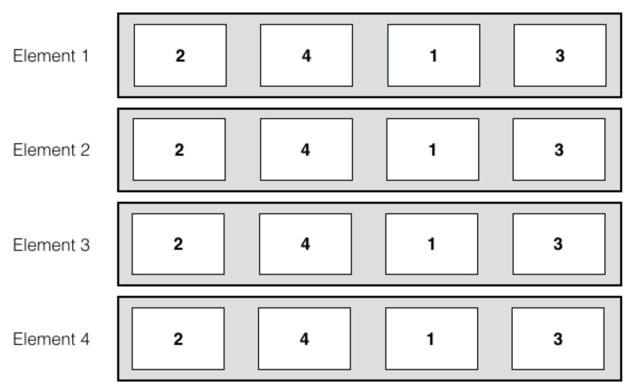
Each row of the puzzle will correspond to an element of a vector, e.g. the first row of the Pseudoku puzzle will be stored as a four-element vector, which itself is an element of a four-element vector. Therefore, this completed Pseudoku puzzle is represented by the following vector:



We 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 to write a function in pseudocode that will carry out this process.

<u>Task 1</u>: Complete the following function template:

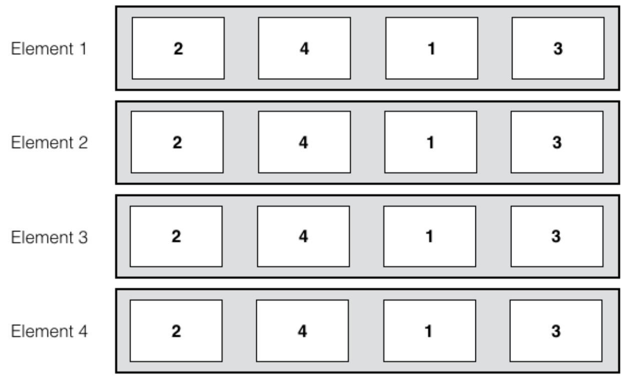
function MAKEVECTOR(*row*)
    new Vector *puzzle*(4)
    ...
end function

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.

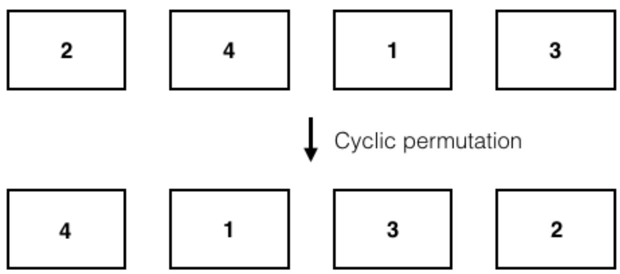
[4 marks]

Cyclic permutation of row vectors

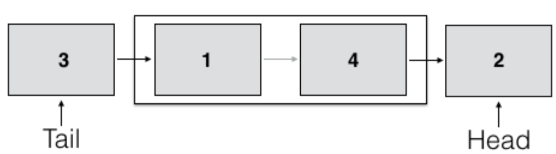
Consider the following vector:



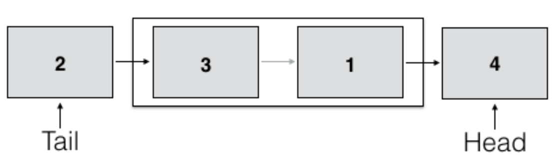
This does not satisfy the Pseudoku conditions since in each column only one number appears. The algorithm for generating Pseudoku puzzles will cyclically permute the elements in each row vector until the numbers in all the rows satisfy the Pseudoku conditions. A cyclic permutation of each row will shift 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 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 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 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* 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 2, 3 and 4 of *puzzle* will be cyclically permuted *x*, *y* and *z* places to the right respectively.

Task 3: Complete the following function template:

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

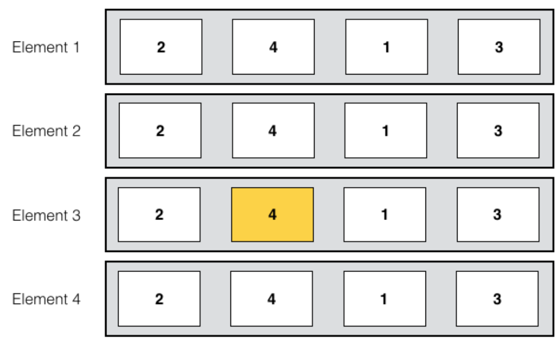
This function should take a four-element vector called *puzzle*, which will be of the form of the output of MakeVector as an input parameter, as well as three integers *x*, *y* and *z*. The function will return *puzzle* but with elements *puzzle[2]*, *puzzle[3]* and *puzzle[4]* cyclically permuted by *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. 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).

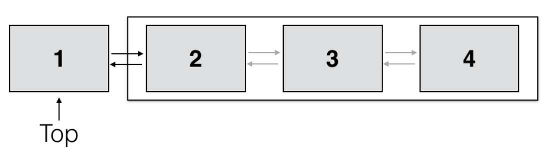
We need a convenient way to refer to elements of the two-dimensional puzzle. We will use a coordinate system of (*row*,*col*) for the four-element vector *puzzle* as produced by MakeVector: *row* is the number of the element of *puzzle* that we care about, and *col* is the number of the element in *puzzle*[*row*] that we care about. Consider the following vector:



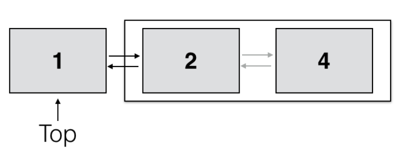
The coordinates of the element in yellow are (3,2), for example. So to select the value stored there we use the syntax *puzzle*[3][2], since we are looking at the second element in *puzzle*[3]. Using this coordinate system to refer to the 2-by-2 sub-grids, for example, the top-left sub-grid will consist of the elements (1,1), (1,2), (2,1) and (2,2).

You will check the Pseudoku conditions using a stack. In particular, you will start with a stack containing all numbers from 1 to 4, and then inspect each element in each column (or sub-grid) to see if the number in that element is stored in the stack: if it is stored in the stack, pop that value from the stack and return the stack; otherwise, return false. This process will be repeated for every element in a column or sub-grid. If, at the end of checking every element, the stack is empty then all numbers appear in the the column or sub-grid; if the stack is not empty, then not all numbers from 1 to 4 appear.

We will complete a function called SearchStack(stack, item) that will search a stack for the value item: if item is in one of the elements of the stack, we remove that element storing item, otherwise FALSE is returned. At the end of the function, aside from missing the value item (if it were there before the function call), the stack should be as before. Let's demonstrate this with diagram. We have the following stack:



We want to look for the value 3 in the stack and remove it if it is there, otherwise if it is not there, return FALSE. The value 3 is stored in the stack so we then need to remove it so the stack looks like this:



The next task will be to complete a function that does this process of searching a stack and possibly removing an element.

Task 4: Complete the following function:

```
function SearchStack(stack, item)
    new Stack second
    bool ← TRUE
    while TOP[stack] ≠ item ∧ EMPTY[stack] = FALSE  do
        ...
    end while
    if EMPTY[stack] = TRUE then
        bool ← FALSE
    else
        POP[stack]
    end if
    while EMPTY[second] ≠ TRUE do
        ...
    end while
    if bool = FALSE then
        ...
    end if
    return stack
end function
```

This function will take a stack and a value (called item) as input parameters, and return FALSE if item is not stored in the stack, otherwise return the stack without the element storing item. Before returning FALSE in the former case, the stack should be put back to its initial condition. Make sure you include the existing lines of the pseudocode (excluding the ellipses) in your function.

[4 marks]

To check whether a column contains all numbers from 1 to 4 in a puzzle vector, we will do the following:

1. Create a stack called numbers, which contains all numbers from 1 to 4
2. Initialise a variable k to be 1
3. For the element k in a column, store the number in that element to a variable called value and call SearchStack(numbers, value)
4. If the function returns FALSE, then we should return FALSE as a number appears twice or not at all in the stack
5. If the function returns the stack, increase the value of k by one and go to step 3
6. If after checking all elements in the column, SearchStack has not returned FALSE, we return TRUE

In the next task you will need to complete a function that carries out this algorithm for column j of the input puzzle.

Task 5: Complete the following function:

```
function CheckColumn(puzzle, j)
    ...
end function
```

This function will take the vector puzzle (as produced by MAKEVECTOR) as an input parameter and check that column j contains all numbers from 1 to 4: if it does contain all numbers from 1 to 4, it should return TRUE, otherwise it should return FALSE. The procedure you should use is the one outlined above. To get full marks you need to call SEARCHSTACK(stack, item).

[6 marks]

Once we have a method for checking one column, we can use the following function to check all columns:

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

This will be useful later on.

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. In the next task, the goal is to replicate the approach of the function CHECKCOLUMN but for these sub-grids. In the function you should repeatedly call SEARCHSTACK for each element in a sub-grid, and then do this for all four sub-grids.

Task 6: Complete the following function:

```
function CheckGrids(puzzle)
    ...
end function
```

This function will take the vector puzzle (as produced by MAKEVECTOR) as an input parameter and check that all sub-grids contain all numbers from 1 to 4: if every sub-grid does contain all numbers from 1 to 4, it should return TRUE, otherwise it should return FALSE. For each sub-grid you should create a stack with numbers from 1 to 4, and then repeatedly search the stack to see if the values in the sub-grid are stored there. To get full marks you need to call SEARCHSTACK(stack, item).

[6 marks]

The solution methods outlined are based on searching a stack. There are other methods for checking the Pseudoku conditions.

Task 7: Describe an alternative method for checking that a column in a puzzle vector contain all numbers from 1 to 4. You can describe the method in words.

[4 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 PERMUTEROWS(*puzzle*, *x*, *y*, *z*) for all combinations of *x*, *y* and *z*) to see if the returned vector returns TRUE for both CHECKGRIDS and COLCHECKS.

<u>Task 8</u>: 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 PERMUTEROWS(*puzzle*, *x*, *y*, *z*) until a set of permutations is found such that all Pseudoku conditions are satisfied (checked using CHECKGRIDS and COLCHECKS). To be able to get full marks you should call the functions MAKEVECTOR, PERMUTEROWS, CHECKGRIDS and COLCHECKS.

[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.

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.

<u>Task 9</u>: 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]