**Mid-term Assessment: Sudoku assignment**

**Algorithms and Data Structure I**

**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.**

1. function MakeVector(row)

new Vector puzzle(4)

puzzle[1] <- row

puzzle[2] <- row

puzzle[3] <- row

puzzle[4] <- row

return puzzle

end function

**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.**

1. function PermuteVector(row, p)

if p = 0 then

return row

end if

new Queue q

x <- 1

while x <= length(row) do

enque(row[x], q)

x <- x + 1

end while

for 0 ≤ i < p do

enqueue(head[q], q)

dequeue(q)

end for

for 1 ≤ i ≤ length(q) do

row[i] <- head[q]

dequeue(q)

end for

return row

end function

**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.**

1. function PermuteRows(puzzle, x, y, z)

puzzle[2] <- PermuteVector(puzzle[2], x)

puzzle[3] <- PermuteVector(puzzle[3], y)

puzzle[4] <- PermuteVector(puzzle[4], z)

return puzzle

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 ≤ k ≤ 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.**

1. function CheckColumn(puzzle, j)

new Vector temp(4)

for 1 ≤ i ≤ 4 do

o <- puzzle[i]

temp[i] <- o[j]

end for

k <- 1

c <- 0

while k < 5 do

if LinearSearch(temp, k) = true then

c <- c + 1

end if

k <- k + 1

end while

if c = 4 then

return true

end if

return false

end function

**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.**

1. function ColCheck(puzzle)

c <- 0

for 1 ≤ i ≤ 4 do

if CheckColumn[puzzle, i] = true then

c <- c + 1

end if

end for

if c = 4 then

return true

end if

return false

end function

**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.**

1. function ColsFromGrids(puzzle)

new Vector (4) tempArr

new Vector (4) tempRow

for 1 ≤ i ≤ 4 do

for 1 ≤ j ≤ 4 do

a <- 1 + floor(i/3) + 2 \* floor(j/3)

b <- 1 + (i+1) mod 2 + 2 \* ((j+1) mod 2)

c <- puzzle[a]

tempRow[j] <- c[b]

end for

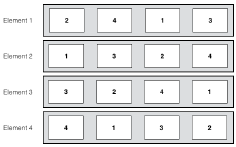
tempArr[i] <- tempRow

end for

return tempArr

end function

**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.**

1. The easiest way to implement our puzzle vector in a concrete data structure is to use the array that stores other arrays as its elements. For example, if we used a simple linear array our puzzle vector has looked something like this: [row1, row2, row3, row4]. We can access any element of this array using its index like ‘puzzle[i]’ (this is analog of the vector operation ‘select[i]’, but we should take into consideration that arrays indexes start from 0). But now, we change row1 from integers to arrays (another element), that contain our integers. For example, row1 = [2, 4, 1, 3], row2 = [1, 3, 2, 4] and so on. Now, we can access any number in two steps: the first step is to access the row of the element, and the second step is to access the required element in this row by its index. So if we need the last element of the second row we can write it like row2[3], while row2 can be accessed as puzzle[1], so we can shortcut select command as ‘puzzle[1][3]’. This type of concrete data structure is called the two-dimensional array. All other vector operations can also be executed on a two-dimensional array. length[array] – returns the length of the element, while length[array[i]] – returns the length of the inner element with an index i. To add the value to a two-dimensional array: ‘puzzle[i][j] <- x’ assign the value ‘x’ to the element with the index ‘j’ in the row ‘I’. So, in the end, our puzzle vector can be created like this:

‘puzzle = [[2, 4, 1, 3], [1, 3, 2, 4], [3, 2, 4, 1], [4, 1, 3, 2]]’

**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.**

1. function MakeSolution(row)

puzzle <- MakeVector(row)

puzzleGrid <- ColsFromGrids(puzzle)

for 1 ≤ i ≤ 3 do

for 1 ≤ j ≤ 3 do

for 1 ≤ k ≤ 3 do

if ColCheck(puzzle) = true AND ColCheck(puzzleGrid) = true then

return puzzle

end if

PermuteRows(puzzle, 0, 0, 1)

end for

PermuteRows(puzzle, 0, 1, 0)

end for

PermuteRows(puzzle, 1, 0, 0)

end for

end function

**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.**

1. The ‘MakeBlanks’ function using the ‘RandomNumber’ function that returns the random numbers between 1 and 4, that itself used to access a random element of the puzzle vector (one for a row, one for a column) and change the value of this element to a blank. This algorithm is executed ‘n’ times, but if the ‘RandomNumber’ function will return the same pair of numbers more than once then the same element will be changed, and the function will return the puzzle vector with ‘(n – number of the matching pairs)’ blank entries. To fix this we should check if the value of the puzzle[row][col] element is not blank before we change it. If the value is already blank, then we can generate another pair of random numbers and try to check again. This way we will guarantee that there will be ‘n’ blank entries in the resulting puzzle vector.

**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.**

1. This algorithm will generate Pseudoku with the order of the numbers repeating that of the initial argument. For example, if we MakeSolution([1,2,3,4]), the solution will also have number ‘2’ between numbers ‘1’ and ‘3’ in all rows, number ‘3’ between ‘2’ and ‘4’ and so on. We can make a function that will rearrange numbers in the puzzle, so we can have different variants of the puzzle. We will use the following functions:

function SwitchNumbers(puzzle, num1, num2)

temp <- 0;

ind1 <- 0;

ind2 <- 0;

for 1 ≤ i ≤ length[puzzle] do

for 1 ≤ j ≤ length[puzzle[i]] do

if puzzle[i][j] = num1 then

ind1 = j

end if

if puzzle[i][j] = num1 then

ind2 = j

end if

end for

temp <- puzzle[i][ind1]

puzzle[i][ind1] <- puzzle[i][ind2]

puzzle[i][ind2] <- temp

end for

return puzzle

end function

function Shuffle(puzzle, iterations)

for 1 ≤ i ≤ iterations do

SwitchNumbers(puzzle, RandomNumber(), RandomNumber())

end for

return puzzle

end function

And now, to integrate this function and get our result we can execute it something like this:

MakeBlanks(Shuffle(MakeSolution([1, 2, 3, 4]), 5), 6)

Executing this code, we will get different Pseudoku’s every time, even if we use the same Row argument because the “Shuffle” function will switch numbers in the random order.