Lab Assignment #4

Due 2/21/2020 at 11:59pm on bCourses

The assignment will be graded by an autograder which will check the outputs of your functions. For your functions to be scored properly, it is important that the names of your functions **exactly match** the names specified in the problem statements, and input and output variables to each function are in the **correct order** (i.e. use the given function header exactly). Instructions for submitting your assignment are included at the end of this document.

If any problem specifies a certain method and/or algorithm to be used, you must implement that method and/or algorithm or receive zero (0) point.

Basic Rule of Recursion: A recursive function has a recursion part and a base case (or several). Always make sure that you have both parts in your recursive function!

On this assignment and future assignments, you are required to start checking that the input follows the format we provide in the table. For example, if we tell you that x is a positive integer, your code must evaluate if the x provided is a positive integer. If it is not valid, your program should terminate and you should display (disp) a message to the user why it has terminated.

Program a Turing Machine: Problems 1, 2, and 3

Introduction

The Turing Machine is a critical and fundamental concept for computer science, since it defines what is computable by a computer: any function that can be computed by a Turing Machine is computable by your computer and, more interestingly vice versa. In this problem, you are asked to program a Turing Machine which accepts inputs of the form: $a^n b^n c^n$, $n = 1, 2, 3 \ldots$. If the input is in the format $a^n b^n c^n$, the Turing Machine should accept the input. Otherwise, it will reject the input.

The finite set of allowable symbols includes 'a', 'b', 'c', 'X', 'Y', 'Z' and 'B'(represents blank). **These characters are case sensitive**. The set of input symbols includes 'a', 'b', 'c' and 'B'. The finite set of states include 'q0', 'q1', 'q2', 'q3', 'q4' and 'q5'. Among them, 'q0' is the initial state and 'q5' is the final state. If the Turing Machine halts in the final state, it accepts the input. Otherwise it rejects the input.

For example, if the Turing Machine receives 'aabbccBB', 'abcB' or 'aaabbbcccB' as input, which follows the $a^n b^n c^n$ format and ends with blank ('B'), then the Turing Machine should output the logical value true. Otherwise, it will output the logical value false. Problems 1, 2 and 3 will guide you through programming your own Turing Machine.

1 Turing Machine Part I: Create a Transition Table

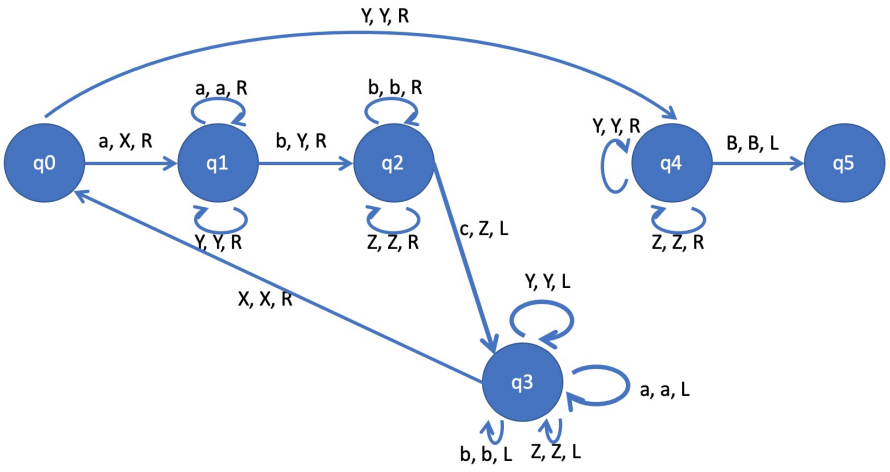
`function anbncn_table = createTable()`		
No Input needed		
Output	Type	Description
`anbncn_table`	`6x8 cell`	The cell array that represents the transition table

Details

For this part, you will translate the transition diagram located below into a transition table. Based on the current state ('qn') and current symbol ('a', 'b', 'c', 'X', 'Y', 'Z', or 'B'), the Turing Machine will get the new state, new symbol, and the movement (left ('L') or right ('R')) from the transition table.

The first column of the transition table will consist of the possible current states 'q0', 'q1', 'q2', 'q3' and 'q4', in that order, except for the first element of the table, located at (1,1), which is empty. The first row of the transition table will be all the possible symbols, including 'a', 'b', 'c', 'X', 'Y', 'Z' and 'B', in that order. The remaining elements will be a 1-by-6 char array which includes the new state, new symbol, and movement, or will remain empty.

For example, based on the transition table, if the current state is 'q0' and the current symbol is 'a', then the corresponding element will be located in (2,2) and be 'q1,X,R'. This tells the Turing Machine to transition to the new state 'q1', change the current symbol (in this case, 'a') to the new symbol 'X', and then move right on the tape.



Tips

- In this problem, you will build a transition table based on the transition diagram. The table will be referenced by your function in problem 2.
- You can preallocate an NxM cell array using the built-in MATLAB function `cell(N,M)`.

- The following chart will give you a sense of what you are going to build in this function. You will need to fill out the ... part with either a char array or leaving it empty.

empty	'a'	'b'	'c'	'X'	'Y'	'Z'	'B'
'q0'	'q1,X,R'	...	...	...	...	...	...
'q1'	...	...	...	...	...	...	...
'q2'	...	...	...	...	...	...	...
'q3'	...	...	...	...	...	...	...
'q4'	...	...	...	...	...	...	...

2 Turing Machine Part II: Look Up the Transition Table

```
function [new_state,new_value,move] = transition(table,now_state,now_value)
```

Input	Type	Description
`table`	`6x8 cell`	The transition table you made in the last problem
`now_state`	`1x2 char`	Current state of the Turing Machine: 'q0', 'q1', 'q2', 'q3' or 'q4'
`now_value`	`1x1 char`	Value read from the tape: 'a', 'b', 'c', 'X', 'Y', 'Z', or 'B'
Output		
`new_state`	`1x2 char`	New state of the Turing Machine: 'q0', 'q1', 'q2', 'q3', 'q4', or 'q5'. See Tips.
`new_value`	`1x1 char`	Value to be written to the tape: 'a', 'b', 'c', 'X', 'Y', 'Z'. See Tips.
`move`	`1x1 char`	The direction of the next movement on tape: 'L', 'R', or '0'. See Tips.

Details

This function figures out the next step the Turing Machine will take based on `now_state` and `now_value`.

Tips

- If the input tape is not in the form $a^n b^n c^n$, the transition table will eventually reference an empty element. In this case, `new_state` should be the current state (`now_state`). `new_value` should be `now_value`. And `move` should be a `1x1 char` equal to '0'
- You may find the built-in MATLAB function isempty() useful
- As you might have already noticed, `now_state` doesn't contain the state 'q5' but `new_state` does. This is because once the Turing machine obtains 'q5', it will halt because it has

determined that the input is in the correct format, and thus won't look up the transition table, move, or write.

- Think about how you extracted a certain element from a char array in Lab02 Problem 6.

3 Turing Machine Part III: Run the Turing Machine

`function output = runTuring(input_tape)`		
Input	Type	Description
`input_tape`	`1xN char`	The char array that represents the input tape, consisting of a's, b's, c's, and B's
Output	Type	Description
`output`	`1x1 logical`	A logical that represents whether the input tape is in the $a^n b^n c^n$ format

Details

Now that everything is prepared, you will program the function `runTuring` that does the following:

1. Call `createTable()` to construct the transition table.
2. Create a variable to represent the current state of the Turing Machine. Since the Turing Machine starts at state 'q0', set this variable initially equal to 'q0'. This variable will take on different states as the Turing Machine runs.
3. Create a variable to represent the position within `input_tape`. We assume the Turing Machine starts at the leftmost element of `input_tape`, you could set this variable initially equal to 1. This variable should change as the Turing Machine moves along `input_tape`.
4. Implement the transitions of the Turing Machine using a while loop. Within the while loop, your function should utilize the `transition` function you wrote in Problem 2 to read instructions from the transition table, write onto the tape, transition to the next state, and move along the tape.
5. You will output a logical value which represents whether the Turing Machine accepts (true) or rejects (false) the input.

Tips

- There are only 2 cases where the Turing Machine will halt:
  - If the current symbol and the current state refer to an empty element in the transition table, `move` = '0' and the Turing Machine halts.
  - If the Turing Machine gets to the final state, the Turing Machine will also halt.

  If Turing Machine halts in the final state, it will accept the input. Otherwise, it will reject the input.
- You might find the built-in MATLAB function strcmp useful.
- Think about how to use each of the inputs and outputs of your `transition` function.
- Think about how the index changes as the Turing Machine moves right or left.

4 Prime Evaluation

`function result = primeCheck(x)`		
Input	Type	Description
`x`	`1x1 double`	A positive integer
Output	Type	Description
`result`	`1xN char`	A character array that is either 'Non-Prime' or 'Prime'

Details

You are required to write a **recursive** program that is able to evaluate if a positive integer x is a prime number. A prime number is defined as a positive integer that is not divisible without remainder by any integer except itself and 1. You are **NOT ALLOWED** to use the built-in MATLAB function `isprime()`. Your program should do the following:

1. Verify that x is a positive integer.
2. Evaluate if the integer x is a prime.
3. Display a message to the user if they used an invalid input using the MATLAB function disp.

Tips

- Remember that 1 is not a prime number, and the only even prime number is 2.
- Because your function has only one input and you are restricted to writing a recursive program, the use of a persistent variable may be helpful to act as a potential divisor of x.
- If you use a persistent variable, clear it when your program concludes. This will help avoid computational mistakes the next time your function is called.

5 Prime Factorization

`function primeFactors = primeFactorization(x,divisor,primeFactors)`		
Input	Type	Description
`x`	`1x1 double`	An integer you wish to perform prime factorization on.
`divisor`	`1x1 double`	A possible positive divisor of x.
`primeFactors`	`1xM double`	An array of the current M prime factors of x. primeFactors is empty the first time your function is called.
Output	Type	Description
`primeFactors`	`1xN double`	An array of all N prime factors of x. See details for an example.

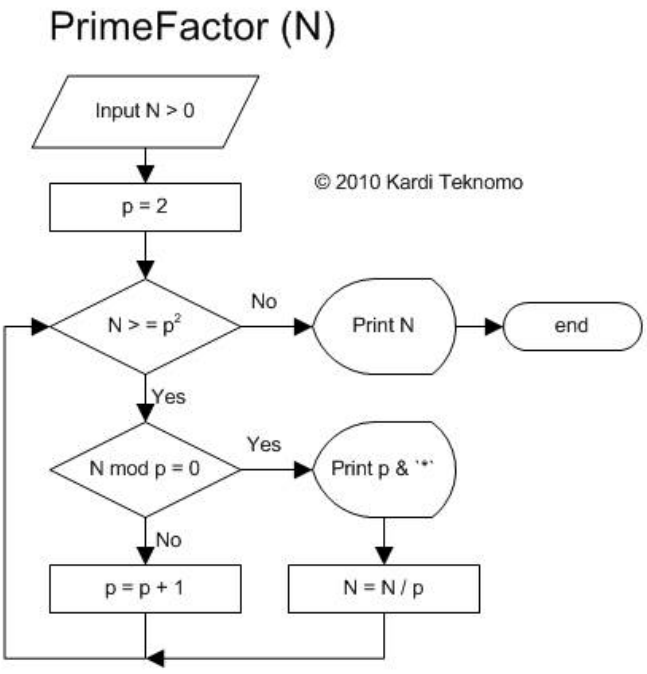
Details

Prime factorization is the process of separating a composite number into its prime factors. You are to write a **recursive** function that accepts a positive integer `x`, a test divisor, and an array of the current prime factors. It should then output the prime factors of the integer `x`. For example, the prime factors of 72 are [2,2,2,3,3]. You are required to do the following:

1. Stop the program if invalid inputs are used.
2. Verify that `x` is a positive integer greater than 1.
3. Verify that `divisor` is a positive integer in the range $(1, \sqrt{x}\,]$
4. Perform the prime factorization of `x`.
5. The elements in your output array should increase from left to right.

Tips

- Remember that 1 is not a prime number, and the only even prime number is 2.
- `divisor` acts similarly to the persistent variable in the primeCheck problem; however, in this case, it is an input of the function.
- If the recursion is implemented correctly, then `primeFactors` should already be sorted.
- You might find the following algorithm and link to be useful.



6 Greatest Common Divisor (GCD)

`function GCD = greatestCommonDivisor(a, b)`		
Input	Type	Description
`a`	`1x1 double`	an integer greater than or equal to 0
`b`	`1x1 double`	an integer greater than or equal to 0
Output	Type	Description
`GCD`	`1x1 double`	the GCD of a and b

Details

Recall that the greatest common divisor (GCD) is the largest positive integer that evenly divides two integers. In this problem, you will write a function that recursively finds the GCD of two input integers. **Do not use the built-in MATLAB function `gcd()`.**

Tips

- The GCD of two integers should not change if the smaller integer is subtracted from the bigger integer.
- Don't forget to program in for cases where `a` or `b` = 0.
- If a and b are both equal to zero, your program should display(disp) a message to remind users that the input is invalid.

Submission Guidelines

When you are ready to test one or more of your function(s), download the `Lab04Tester.m` script from bCourses and run it in the same directory as your function(s). This will run a series of test cases for your function(s) and display the outputs. You should check these results with the example outputs displayed. Note that the series of tests that the script will run is not exhaustive and that it is up to you to make sure that your function fully satisfies the requirements of the problem. If you want to run all the test cases for one function, select that part of the tester script and push the `Run Section` button. If you wish to only check a single test case, you may do so by highlighting the appropriate lines in Matlab and then pressing F9 (on Windows and linux) and `shift + fn + F7` (on a Mac) to run the selected text in the Command Window.

The script will also pack your functions into a zip file for final submission. Your function file names and headers should match the examples given in the problem statement. If they do not match, the tester script will not be able to find them and will issue a warning. If you do not correct this problem, your missing function(s) will not be able to be graded and **you will receive 0 points**, so be sure to check carefully! Make sure to upload the `Lab04.zip` file to bcourses before the deadline. You may resubmit your zip file by re-uploading to bcourses before the deadline, but please be aware that only the most recent zip submission is considered for grading. That said, it is critical that you include your most recent version of **all** of your functions in each new zip submission.

Your final zip file for this assignment should contain the following:

- `createTable.m`
- `transition.m`
- `runTuring.m`
- `primeCheck.m`
- `primeFactorization.m`
- `greatestCommonDivisor.m`