Lab Assignment #6
Topics: Time Complexity, Linear Systems, Least Squares Regression

Due 3/20/2020 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 you will receive a zero (0) on that problem.
On this assignment and future assignments, you are required to check 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.

1 Linear System: Solutions

`function message = linearEquationTest(A,y)`		
Input	Type	Description
`A`	`mxn double`	Matrix A in the linear system $Ax = y$
`y`	`mx1 double`	Vector y in the linear system $Ax = y$
Output	Type	Description
`message`	`1xQ char`	A message describing the existence and uniqueness of the solution

Details

Consider the system:

$$Ax = y$$

There exists a solution to the system $Ax = y$ if the rank of the matrix `A` is equal to the rank of `[A y]`. You can read more about the rank of matrices here.

The solution to the system is unique if the number of unknowns in the system is equal to the rank of the matrix `A`.

Your output `message` should be one of the following.

- `'Solution exists and is unique'`
- `'Solution exists but is not unique'`
- `'Solution does not exist'`

2 Big O time complexity

function [answers] = myBigOAnswers()		
Output	Type	Description
`answers`	`1x4 char`	A character array with your answers to the multiple choice questions for 2.1-2.4

Details

In computer science, the time complexity describes the amount of time it takes to run an algorithm. Time complexity is commonly estimated by counting the number of elementary operations performed by the algorithm, supposing that each elementary operation takes a fixed amount of time to perform.

Reporting the exact number of times that each fundemental operation occurs can become quite tedious and often does nothing to help with the interpretation of the results. It is common to report time complexity in big O notation, where all constant coefficients are dropped and only the leading term is reported.

For each of the following, you should analyze the code to determine its time complexity in big O notation. Then, write a function that takes no inputs and outputs the answer you have chosen (`'a'`, `'b'`, `'c'`, `'d'`, or `'e'`) from the multiple choice selection of each problem. Note that no actual coding on your part is necessary for this problem. Your function need only be 1 line with your multiple choice answers. (Your answer is case sensitive – use lower case.)

2.1 `myGoldenRatioIterative.m` from Assignment 3

```
function ratio = myGoldenRatioIterative(n)
F=ones(n+1);
if n >=2
    for i=3:n+1
        F(i)=F(i-1)+F(i-2);
    end
end
ratio=F(n+1)/F(n);
end
```

(a) O(n)

(b) O(n^2)

(c) O($exp(n)$)

(d) O($log(n)$)

(e) O($n * log(n)$)

2.2 `myFarray.m` from Assignment 3

```
function F_n = myFArray(n)
if n == 1 | n == 2
    F_n = 1;
else
    F_n = myFArray(n-2) + myFArray(n-1);
end
```

(a) O(n)

(b) O(n^2)

(c) O($n * log(n)$)

(d) O($exp(n)$)

(e) O($log(n)$)

2.3 `myProduct` function

n is the length of `array`.

```
function [results] = myProduct(array, target)

results=[];

for i = 1:length(array)
    for j = i+1:length(array)
        for k = j+1:length(array)
            Multiplication = array(i)*array(j)*array(k);
            if Multiplication == target
                result_temp = sort([array(i) array(j) array(k)]);
                results=[results;result_temp];
            end
        end
    end
end
results = unique(results,'row');

end
```

(a) $O(n^3)$

(b) $O(exp(n))$

(c) $O(n * log(n))$

(d) $O(n^2)$

(e) $O(n^2 * log(n))$

2.4 Binary search from midterm 1

n is the length of `array`.

```
function index = binarySearch(array,val)
        middle = ceil(length(array)/2);
        if array(middle) == val
            index = mid;
        elseif array(mid) < val
            index = middle + binarySearch(array(middle+1:length(array)),val);
        else
            index = binarySearch(array(1:middle-1),val);
        end
    end
```

(a) $O(n)$

(b) $O(n^2)$

(c) $O(n * log(n))$

(d) $O(exp(n))$

(e) $O(log(n))$

3 A simple linear fit problem for $O(n)$ time complexity

`function [slope, intercept, fig] = myLinearCom(fn, n, choice)`		
Input	Type	Description
`fn`	`function_handle`	A function handle
`n`	`1x1 double`	A positive integer
`choice`	`1x1 logical`	A logical value representing whether the y-intercept is non-zero (`true`) or zero (`false`) in the line of best fit
Output	Type	Description
`slope`	`1x1 double`	The slope of your line of best fit
`intercept`	`1x1 double`	The intercept of your line of best fit
`fig`	`1x1 matlab.ui.Figure`	A figure handle for the plot you produce

Details

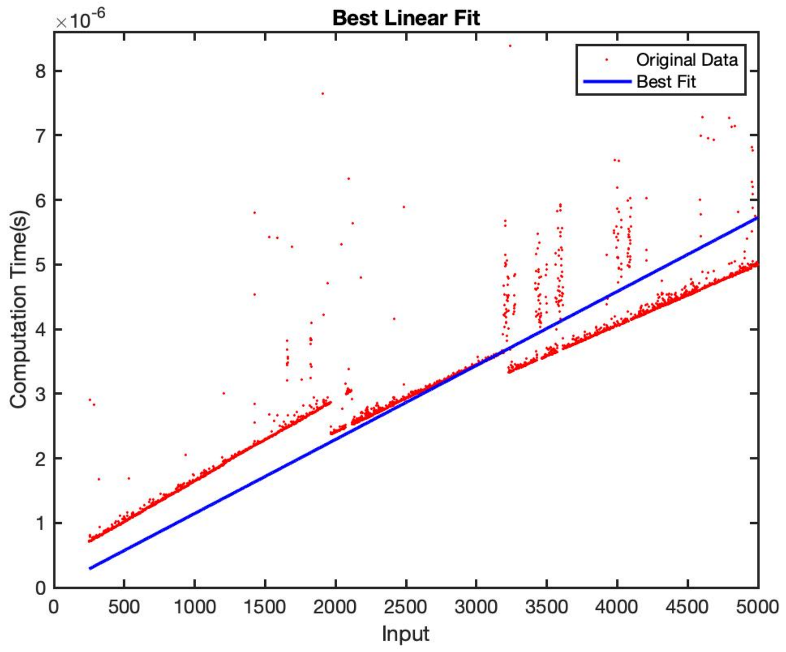
In this problem, you will perform a linear regression over a set of computation times that are generated by a function in $O(n)$ complexity. Your function `myLinearCom.m` will take three inputs: a function handle, a positive integer `n` that indicates how many times `fn` will be called and `choice` representing whether y-intercept is zero.

You will record the computation time `fn` takes to execute using `tic` and `toc` for each element in the array `[1:1:n]`. The function handle `fn` is $O(n)$ time complexity. Thus, you are trying to fit a linear relationship between array `[1:1:n]` and computation time, $time = slope \cdot input + intercept$. You **must** use mldivide, or \, to perform a least-squares regression of the data.

Use `fig=figure;` **before** you do any plotting in order to create the figure handle.

Your code should produce a plot that displays the data along with the line of best fit, as shown. Your code should also output the slope and the y-intercept of the line of best fit.

You can use any plotting style you like, as long as it plots the correct data (x = 1:n vs y = computation times) and includes a title, labels and a legend.



Tips

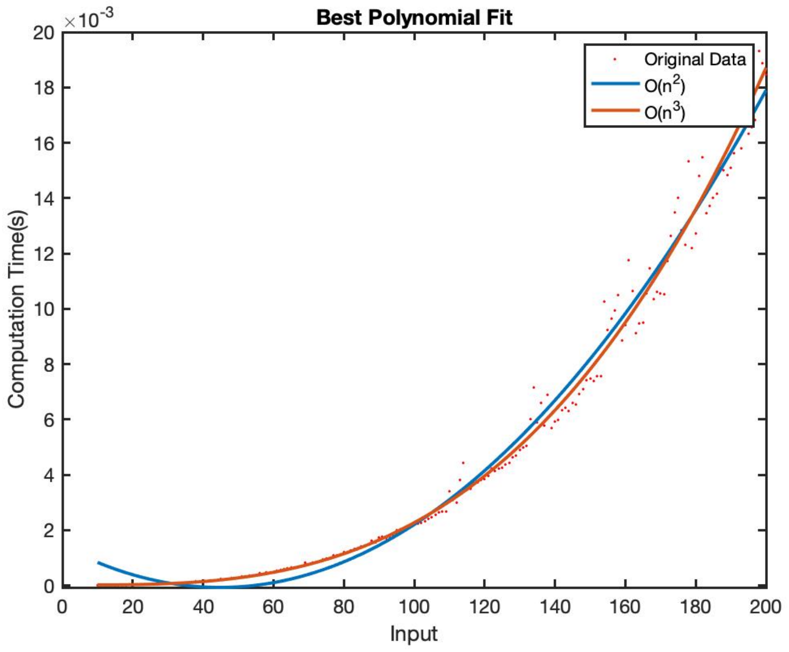
- You can find out how to create function handle in here.
- All plotting commands will be applied to the most recent figure by default.
- In the `Lab06_Tester.m`, you will be given a $O(n)$ time complexity function `linear.m` to test your program with. We recommend using $n \geq 5000$.
- You can improve your fit by removing the first 5% of the data, though this is not required. As you can see from above figure, the first 5% data is dropped out.

4 Polyfit and polyval for polynomial time complexity

`function [coefficient, fig] = myPolyCom(fn, n)`		
Input	Type	Description
`fn`	`function_handle`	A function handle
`n`	`1x1 double`	A positive integer
Output	Type	Description
`coefficient`	`2x4 double`	An array of coefficients for the quadratic $O(n^2)$ and cubic $O(n^3)$ polynomial fit
`fig`	`1x1 matlab.ui.Figure`	A figure handle for the plot you produce

Details

Similar to Problem 3, you will record the computation time `fn` takes to execute for each element in the array `[1:1:n]`. This time, you must produce coefficients of best fit for quadratic and cubic using the MATLAB built-in function polyfit. `fn` and `n` have the same meaning as Problem 3. You must create the curves of best fit using polyval for both the quadratic fit $O(n^2)$, and the cubic fit $O(n^3)$ and plot these along with the data as shown in the figure below.



- Use `fig=figure;` **before** you do any plotting in order to create the figure handle. Use `hold on` or `hold off` to plot multiple times in the same figure.
- The first row of the output `coefficient` should be an array of coefficients from the quadratic ($O(n^2)$) polynomial fit. Since it will not have the cubic coefficient, the first element will be zero. The second row of the output `coefficient` is an array of coefficients from the cubic ($O(n^3)$) polynomial fit.
- You may format the plot however you like, as long as it plots the correct data and includes a title, labels and a legend.

Tips

- You can find out how to create function handle here.
- You can use `ylim` to see the relevant result.
- In the `Lab06_Tester.m`, you will be given two functions `quadratic.m` and `cubic.m` to test. We recommend using $n \geq 500$.
- You can improve your fit by removing the first 5% of the data, though this is not required.

5 Estimating time complexity from function runtimes

`function [complexity] = estimateTimeComplexity(fn, n)`		
Input	Type	Description
`fn`	`function_handle`	A function handle for the function you would like to analyze.
`n`	`1x1 double`	A positive integer
Output	Type	Description
`complexity`	`1xQ char`	A character array detailing the time complexity for the given function

Details

For this problem, you will write a function that will assign a time complexity of `'O(n)'`, `'O(n^2)'`, `'O(n^3)'`, `'O(log(n))'`, or `'O(exp(n))'` based on the computation time of that given function. Similar to Problems 3 and 4, you will have a function handle `fn` and a positive integer `n` as inputs. You will be calling `fn` a total of `n` times and recording the computation time for each function call. A function `arrayBuilder` will be given for generating a `1xn` array that contains the inputs for `fn`. You are to use least-squares regression to determine the best time complexity fit for `fn`. From here, you could analyze this data with different regression methods.

You function should be able to recognize the following time complexities:

1. Polynomial time, which should be designated as `'O(n)'` for linear time complexity, `'O(n^2)'` for quadratic time complexity, and `'O(n^3)'` for cubic time complexity.
2. Logarithmic time, which should be designated as `'O(log(n))'`.
3. Exponential time, which should be designated as `'O(exp(n))'`.

You will be given a template for `estimateTimeComplexity.m` and a function `arrayBuilder.m` for constructing your solution. The template `estimateTimeComplexity.m` contains some code for testing an additional program that you will not be provided. Please make sure that you **DO NOT delete anything from the template**. If you do, it will cause problems with grading.

The best fitting regression should be determined from examining the R^2 value of each fit, which is defined as:

$$R^2 = 1 - \frac{\sum(y_i - f_i)^2}{\sum(y_i - \bar{y})^2}$$

Where y_i is each of the original computation times of your data, $\bar{y}$ is the mean value of the original computation times, and f_i is each of the predicted computation time based on your line of best fit. The model with the closest R^2 value to 1 represents the best fit.

Tips

- To recognize different Polynomial time complexity(`'O(n)'`,`'O(n^2)'` and `'O(n^3)'`), one method is looking at the slope from linear regression in log space: if `n` is large enough, the slope for `'O(n)'` will close to 1, the slope for `'O(n^2)'` will close to 2, the slope for `'O(n^3)'` will close to 3.
- The runtime of a function only scales with its big O time complexity for sufficiently high values of n. You may assume that the values of `n` will be appropriately selected for the easy, correct determination of time complexity for a given function. However, be careful of specifying `n` values that are too high when you are testing out your function, as they may take excessively long to run (especially for exponential time and high order polynomial time).

Submission Guidelines

When you are ready to test one or more of your function(s), download the `Lab06Tester.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 function 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 `Lab06.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:

- linearEquationTest.m
- myBigOAnswers.m
- myLinearCom.m
- myPolyCom.m
- estimateTimeComplexity.m