Sample

Final Exam

COMPUTER PROGRAMMING FOR ENGINEERING AND SCIENCE
Held on 13th of August, 2015 (CSCE 155N, SUMMER 2015)

Name :

Save : **YES** **NO**

Course No : **CSCE155N Matlab**

Instructions:

1. This is open book, open note, but not open computer or open neighbor. Please do not use email, texting, Morse code, sign language, spy cameras, etc. during the exam.
2. If you have a question about the meaning of an exercise, ask! Getting things wrong because of misunderstandings can be aggravating for me as well as you.
3. If you wish to have the exam saved so that you can retrieve it later, please indicate this. Exams so marked will be saved until at least the second week of classes in the fall.

1. (10 points) Consider the following snippet of code. How will Matlab respond to each line? Explain the significance of *eval*, of the *input* functions inside the assignments, and the pair of single quotes scattered throughout the assignment for *exuberant*.

```
happy = 'glad';
glad = 'elated';
elated = 'exuberant';

eval('happy')

eval(happy)

eval(eval(happy))

eval(eval(eval('happy')))

fprintf('I, %s, passed with an %s.\n', input('Enter your name: '), ...
input('Enter grade: '))

exuberant = 'fprintf(''I, %s, passed with an %s.\n'',
input(''Enter your name: '',''s''), input(''Enter grade: '',''s'')) '
      note: sorry for the overflow line above - imagine as one long line

eval(exuberant)
```

2. (10 points) Consider the following functions. What would be the response to the command line **x = nifty(2,3) * nifty(1,2)**?

```
function a = nifty(b,c)
t = c + neato(bummer(c), bummer(b));
a = t - bummer(neato(t,c));
end

function x = neato(y,z)
x = y + z;
end

function x = bummer(y)
x = y * 2;
end
```

3. (10 points) The recursive function for calculating a Fibonacci number as shown in class was horribly inefficient because of repeated calls for the same value. Take a look at the following. What does it display if n is 7? **Note that each f = ... does not end with semicolon, so something will be printed!** Does it solve the inefficiency problem of the earlier version? Explain.

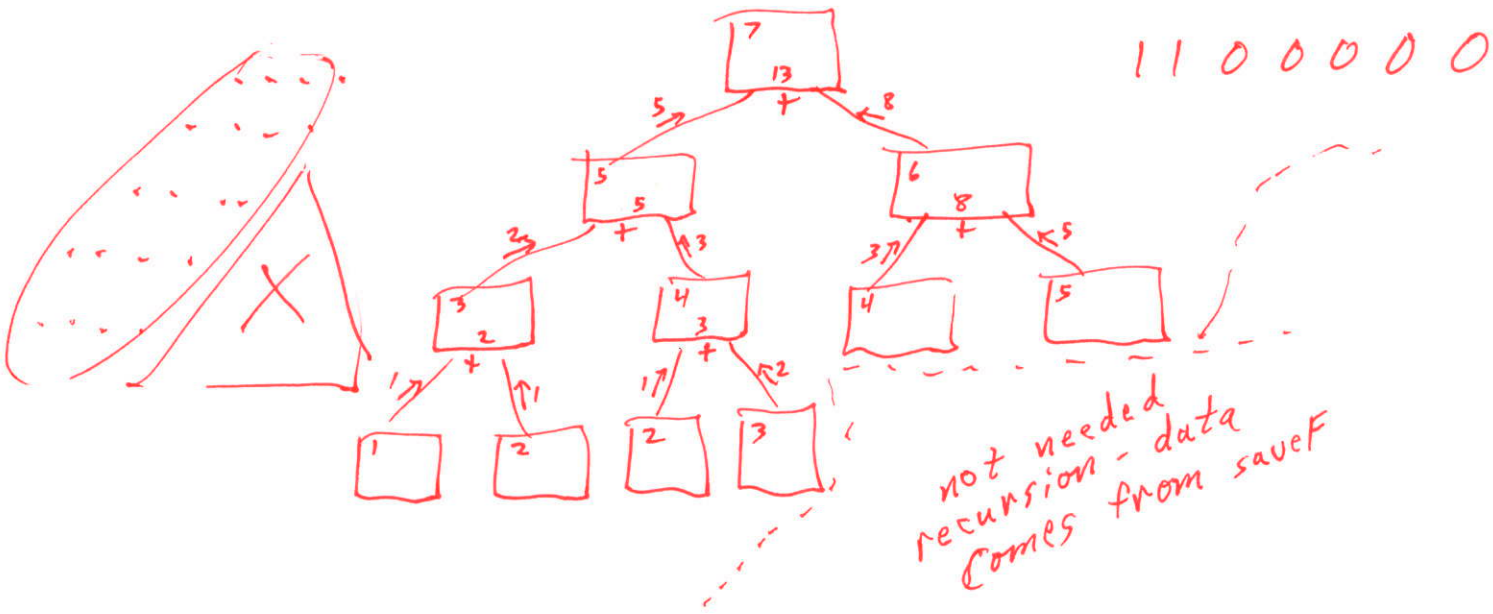
yes!

```
function f = fibonacci(n)
saveF = zeros(1,n);
saveF(1:2) = [1 1];
f = fibo(n)
   function f = fibo(n)      % Note: Nested function so saveF is seen everywhere
   if n < 3
      f = 1                  % Might display here
   elseif saveF(n) > 0
      f = saveF(n)           % Might display here
   else
      f = fibo(n-2) + fibo(n-1)  % Might display here
      saveF(n) = f;
   end
   end
end
```

f= 2
3
5
8
13
13
ans = 13

Here is the original inefficient version:

```
function x = fibo(n)
if n <= 2
   x = 1;
else
   x = fibo(n-2) + fibo(n-1);
end
```



4. (10 points) Consider the following functions. What would be the response to the call **foo(5)**? Note that there is both a returned value and side effects.

```
function x = foo(n)
disp(n)
if n >= 1
   x = n + 2 * bar(n-1);
else
   x = 1;
end

function x = bar(n)
disp(n)
if n >= 1
   x = n - 2 * foo(n-1);
else
   x = -1;
end
```

(side effects)

5
4
3
2
1
0

ans = -31

foo(5) ↓ ↑ x = 5 + 2·(-18) = -31
bar(4) ↓ ↑ x = 4 - 2·11 = -18
foo(3) ↓ ↑ x = 3 + 2·4 = 11
bar(2) ↓ ↑ x = 2 - 2·(-1) = 4
foo(1) ↓ ↑ x = 1 + 2·(-1) = -1
bar(0) ↓ ↑ x = -1

5. (10 points) Consider the following function.

   (a) What would be the response to the call **fun** with no arguments?

   (b) Rewrite it (the function and default x) so that it evaluates infix notation expressions that are formed as nested cell arrays.

```
function t = fun(x)
if nargin == 0
   x = {{5 2 '+'}{{{7 1 '*'} 6 '*'}{3 4 '*'}'+'}'+'};
end

if iscell(x)
   switch x{2}
      case '+'
         t = fun(x{1}) + fun(x{3});
      case '*'
         t = fun(x{1}) * fun(x{3});
   end
else
   t = x;
end
```

7
7
12
42
54

61

{5 + 2} + {{{7 * 1} * 6} + {3 * 4}}

6. (10 points) Consider the following code that checks to see if an array is sorted. Create both an anonymous function and a regular function for each of the following cases that responds with **true** if the first argument is less than the second. Finally show how to invoke **isSorted** to answer the requests that follow:

   (a) Compare two temperatures according to distance from zero degrees C. The array of temperatures to verify is called *temperatures*.

   cmpTemp = @(t1, t2) abs(t1) < abs(t2)

   (b) Compare two dates according to their occurrence in the year. A date is a struct with two numeric fields *month* and *day*. The array of dates to verify is called *dates*.

   cmpDate = @(d1, d2) d1.month < d2.month || d1.month == d2.month && d1.day < d2.day

```
function s = isSorted(a, cFun)
   s = true;
   for ii = 2:length(a)
      if ~cFun(a(ii-1), a(ii))
         s = false;
      end
   end
```

precede with @ for regular function

isSorted(temperatures, cmpTemp)

isSorted(dates, cmpDate)

function tless = cmpTemp(t1, t2)
tless = abs(t1) < abs(t2)
end

function dless = cmpDate(d1, d2)
dless = d1.month < d2.month || d1.month == d2.month && d1.day < d2.day
end

7. (10 points) Consider the following code. Trace through how it works. Effectively, what is its practical purpose?

8 9 9 9 ←

```
x = [3 4 8 0 0 0 0];
m = length(x);
ii = m;
while ii > 0 && x(ii) == 0
   x(ii) = 9;
   ii = ii - 1;
end
if ii > 0
   x(ii) = x(ii) - 1;
end
```

decrement (subtract 1)

from right to left changes 0's into 9's until reaches rightmost non-zero which it decrements

8. (10 points) Consider the following code. What do the functions **enqueue** and **dequeue** really do? What is the result of the code that follows the function definitions?

```
function q = enqueue(v, q)
   q = [v q];
end

function [v q] = dequeue(q)
   v = q(end);
   q = q(1:end-1);
end

r = [];
q = [];
q = enqueue(1,q)
q = enqueue(2,q)
q = enqueue(3,q)
[v q] = dequeue(q)
r = enqueue(v,r)
q = enqueue(4,q)
[v q] = dequeue(q)
r = enqueue(v,r)
[v r] = dequeue(r)
q = enqueue(v,q)
```

add v to front of q

remove v from back of q

q r

[1] [2 1] [3 2 1] [3 2] [1] [4 3 2] [4 3] [2 1] [2] [1 4 3]

9. (10 points)

(a) Describe or sketch the structure of *cella* after the following lines are run.

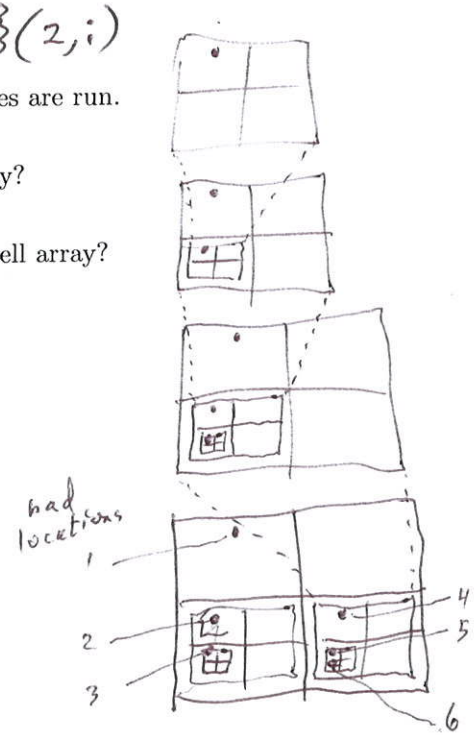
(b) How could one access each of the *bad* messages in the cell array?

(c) How could one access the most deeply nested em [3 4] in the cell array?

```
cella = {{'bad' 'good'}, [2 3; 3 4]; pi, true};
cella{2,1} = cella;
cella{2,1} = cella;
cella{2,2} = cella;
```

cella{2,2}{2,1}{2,1}{1,2}(2,:)

1 cella{1,1}{1}
2 cella{2,1}{1,1}{1}
3 cella{2,1}{2,1}{1,1}{1}
4 cella{2,2}{1,1}{1}
5 cella{2,2}{2,1}{1,1}{1}
6 cella{2,2}{2,1}{2,1}{1,1}{1}



10. (10 points) Please describe how you expect to be using programming in future classes and your career. (If you don't have any idea, I would like to know that also!)

free

Question	Points	Score
1	10	
2	10	
3	10	
4	10	
5	10	
6	10	
7	10	
8	10	
9	10	
10	10	10
Total:	100	