This homework uses commands such as `class`, `isa`, `nargin`, `varargin`, `nargout`, `varargout`, `try`, `catch`, `inputname`, `assignin`, `strmatch`, `evalin`, and `feval`.

1. Write a function that returns a structure (`struct`) with 7 fields, `DoubleCount`, `NDDoublesCount` (ndims>2), `StructCount`, `CellCount`, `CharCount`, `OthersCount`, and `TypesOtherCount`. The values should be nonnegative integers, which are numbers of the types of variables that are in the calling workspace. For example, if the workspace contains a 3-by-4 `double`, a 2-by-6 `double`, a 3-by-4-by-6 `double`, a 2-by-2 `tf`, a 3-by-1 `tf`, a 4-by-2 `tf`, a 5-by-6 `struct`, a 2-by-5-by-6 `struct`, a 1-by-7 `struct`, a 3-by-4 `cell`, and a 3-by-4-by-5 `ss`, then the results should be `DoubleCount = 3`, `NDDoublesCount=1`, `StructCount = 3`, `CellCount = 1`, `CharCount = 0`, `OthersCount = 4`, and `TypesOtherCount = 2`.

2. Matlab allows you to “grow” an array simply by assigning beyond its boundaries. For example (just for example)

```matlab
F = []; for i=1:10000  
    F(i) = i;  
end
```

will result in a 1-by-10000 array F. Behind the scenes, this involves constantly allocation of new memory, copying contents of memory, and freeing memory associated with the old version of F. On my computer (kind of slow), this takes about 0.18 seconds. Changing the loop to 20000 results in a 1.5 second run, and to 100000 results in 105 second run. Note that the time grows much worse than linearly with dimension. So, normally, you preallocate matrices to be of the correct size, and then fill in as you compute. For example

```matlab
F = zeros(1,100000); \% preallocate
for i=1:100000  
    F(i) = i;  
end
```

only takes about 0.19 seconds. Moreover, it roughly grows linearly with dimension. What if you don’t know the eventual size of an array (eg., storing some data being computed in a `while` loop, with no apriori idea of how many loops will run)? How would you do this, in order to avoid the problem of constantly assigning beyond the current dimension? Illustrate this with an actual example, like the one above. What are the tradeoffs with your approach (eg., extra memory that is never used, etc.)?
3. Write a function which calls itself recursively (at least one recursive call), and use `whos` to display the contents of the FunctionInstance workspace, clearly showing that each instance of a function has a different workspace.

4. Write a function called `inc` that has one input argument, and no output arguments. If the input argument is a scalar, `double` variable, then the function should increment its value back in the calling workspace. Using `tic` and `toc`, estimate the overhead (timewise) in calling your function versus just doing `a = a+1;`. Finally, if the input argument is a different class and/or size, the function can issue an appropriate error.

5. Suppose that `matop` is a function that takes as input several 2-d (i.e., `ndims==2`) `DOUBLE` arrays, and returns several 2-d `DOUBLE` arrays as output. Assume that the dimensions of the output variables only depend on the dimensions of the input matrices, not the actual data within the input matrices. We want to write a function `ndapply`, which works for 3-d, 4-d, N-d `DOUBLE` arrays, returning similarly dimensioned output arguments. The calculation performed by `matop` should simply be done on every 2-d “slice” of the input arguments.

   For example, the command `EIG` computes the eigenvalues of a square matrix. If `A` is N-by-N, then `B = EIG(A)` will be an N-by-1 vector containing the eigenvalues of `A`.

   Now, if `A` is N-by-N-by-N1-by-N2-...-by-Nk, then we want `B = ndapply(@eig,A)` to return a N-by-1-by-N1-by-N2-...-by-Nk matrix whose entries are the corresponding eigenvalues of the square 2-d “slices” of `A`. Note that `B = ndapply('eig',A)` should also work.

   Test cases will be at the class website soon.