Drill Problem: #1

Function Name: choCollatz

Inputs:

1. (*double*) An integer N greater than or equal to 0, representing the number of pieces of chocolate you have

Outputs:

1. (*double*) The number of pieces of chocolate you will have left
2. (*double*) The number of children you can give candy out to

Function Description:

Oh no! It's Halloween night and you've forgotten to buy the most important item: candy! Even worse, the kids in your neighborhood will surely cry if you don't have something for them. Luckily, after rummaging through your pantry, you've found a single chocolate bar, broken into N pieces. You reason that if you split it well, at least some number of kids will get candy.

You know each child will demand half of any amount of chocolate you have, and no child will want less than two pieces. Knowing this, you decide to implement a variation of the Collatz conjecture (also known as the 3n+1 conjecture). The conjecture says that any positive integer N can be recursively manipulated **to be a number less than 2** by the following algorithm:

- if the number is even, divide it by 2
- if the number is odd, multiply it by 3 and add 1, then divide it by 2
- repeat (i.e. recursively call the function)

So, if you have an even number of pieces N, you will give half of them (N/2) to the next kid. However, if you have an odd number of pieces M, you will break up your entire chocolate bar into 3M+1 pieces, and then give the next kid half of those pieces. Once the number of pieces you have is less than 2, you will no longer have enough chocolate to give out.

Your job is to write a recursive MATLAB function called `choCollatz()` that implements this Collatz conjecture, given your input is the number of pieces of chocolate you start off with. The first output of your function should be the number of leftover pieces of chocolate (i.e. the final number that the algorithm reaches). Your second output should be the number of kids you can give candy out to (i.e. the number of times you can recursively run through the algorithm).

Notes:

- If you try running this function with very large numbers (as in 40-digit numbers), MATLAB will crash as it will reach its maximum recursion limit.
- For the sake of this problem, the algorithm for an odd number of pieces should include both multiplying by 3 and adding 1, as well as dividing 2. The even and the odd step should both only be one recursive call.

Hints:

- You may find it useful to make a helper function to keep track of the number of recursive calls that have been made.

Drill Problem: #2

Function Name: jinkies

Inputs:
1. (*struct*) A 1x1 structure

Outputs:
1. (*char*) The name of the culprit
2. (*double*) The number of masks worn to hide the culprit's identity

Function Description:

Jinkies! Shaggy and Scooby-Doo have won six VIP passes to the All-You-Can-Eat Hall of Fame, and they've decided to bring you and the rest of the gang along with them. You—of course—accept their invitation, and after a road trip in the Mystery Machine, you arrive. However, things are not as they seem. The All-You-Can-Eat Hall of Fame has been having mysterious happenings, and many of its patrons are saying it's—gulp—haunted! Of course, Mystery Incorporated gets straight to work catching the bad guy, and you help them with your handy MATLAB knowledge! When the gang catches the villain, the villain will be wearing a series of masks; it is your job to help peel them all back to uncover who the bad guy really is.

Write a function in MATLAB that takes a 1x1 structure as its input with an unknown number of fields. One of the fields will contain the name of the mask being worn—which will be the culprit's name after all the masks have been uncovered—and another field will contain a nested structure in the same format as the current structure—this is the new identity under the current mask. You are **not** guaranteed any specific field names or a number of fields. However, you ***are*** guaranteed that the first field will contain the name of the person or disguise, and the last field name will be the "identity" of the suspect under the current mask. Your job is to recursively "unmask" the current suspect **until there is no longer a nested structure in the final field**. When this is the case, a string of the suspect's name will be in villain name field. You are to output this final string, as well as the number of masks the culprit wore that had to be cycled through to find the villain's true identity.

When there is no longer a mask (structure in the last field) to uncover, the function will output the culprit name (first field) from that structure and how many masks were worn by the villain.

For example, If the input structure's first field were below on the left, you would recursively grab the nested structure, contained in the last fieldname. The middle structure is what an intermediate mask may be. This recursive call would continue until you had no nested structure, like the structure on the right. Scrappy-Doo would be the villain here with 2 masks.

`Name: 'Monster'`
`Disguise: [1x1 struct]`
	`Name: 'The Hulk'`
	`Disguise: [1x1 struct]`
		`Name: 'Scrappy-Doo'`
		`Disguise: []`

Hints:
- You may find the `fieldnames()` function useful.
- It may be helpful to look at a test case to fully visualize what this function is doing.

Drill Problem: #3

Function Name: speedStack

Inputs:
1. (*double*) The length of the base of the pyramid
2. (*char*) A pyramid character

Outputs:
1. (*char*) The created pyramid of characters

Function Description:

Speed Stack is a competition in which contestants stack cups in a pyramid conformation with the goal being to do so in the shortest amount of time. Since not everyone in the world has been blessed with great manual dexterity, your local Speed Stack organizers have decided to hold an alternate competition in which a pyramid with a size of their choosing must be created in the shortest amount of time possible by any means the contestants' desire. Recognizing that using a computer is an almost surefire way to win, you decide to enter the competition with MATLAB.

Write a recursive function that takes in the length of the base of the pyramid as well as any single character with which to build a pyramid (pyramid character) and outputs an array of class `'char'` that contains the pyramid. The dimensions of your array should be <length of base> by <(2*length of base)-1>. Starting in row 1 of the array, each subsequent row should have 1 more pyramid character than the row above it and a single space (ASCII value 32) should separate each pyramid character within a row. Any other index within the array that does not contain a pyramid character should be filled with a space (ASCII value 32).

For example, for a pyramid of base length 10 and constructed from the character `'/'`, your function should output a 10x19 character array as shown below (in order to better visualize the actual construction of the array, the ASCII values for this array have been printed below with the pyramid character's value highlighted)

```
         /
        / /
       / / /
      / / / /
     / / / / /
    / / / / / /
   / / / / / / /
  / / / / / / / /
 / / / / / / / / /
/ / / / / / / / / /
```

Drill Problem: #3

32	32	32	32	32	32	32	32	32	47	32	32	32	32	32	32	32	32	32
32	32	32	32	32	32	32	32	47	32	47	32	32	32	32	32	32	32	32
32	32	32	32	32	32	32	47	32	47	32	47	32	32	32	32	32	32	32
32	32	32	32	32	32	47	32	47	32	47	32	47	32	32	32	32	32	32
32	32	32	32	32	47	32	47	32	47	32	47	32	47	32	32	32	32	32
32	32	32	32	47	32	47	32	47	32	47	32	47	32	47	32	32	32	32
32	32	32	47	32	47	32	47	32	47	32	47	32	47	32	47	32	32	32
32	32	47	32	47	32	47	32	47	32	47	32	47	32	47	32	47	32	32
32	47	32	47	32	47	32	47	32	47	32	47	32	47	32	47	32	47	32
47	32	47	32	47	32	47	32	47	32	47	32	47	32	47	32	47	32	47

Notes:

- You must use recursion to solve this problem. Any function not using recursion will result in a 0 for this problem.
- Your function should work for any base length that is <498, since this is MATLAB’s maximum recursion limit, allowing for helper functions.
- The pyramid character is guaranteed to be of length 1, and the size of the pyramid base is guaranteed to be at least 1.

Drill Problem: #4

Function Name: determinant

Inputs:

1. (*double*) An MxM matrix

Outputs:

1. (*double*) The determinant of the input matrix

Function Description:

In linear algebra, the determinant of a square matrix is used for a variety of computations. The determinant of a 2x2 matrix is computed by:

$$A = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \quad \rightarrow \quad \det(A) = ad - bc$$

For example: $$A = \begin{bmatrix} 2 & 7 \\ 1 & 4 \end{bmatrix} \quad \rightarrow \quad \det(A) = ad - bc = (2) \times (4) - (7) \times (1) = 1$$

However, matrices larger than 2x2 must be “broken down” into smaller 2x2 sub-matrices (called cofactor expansion). To compute the determinant of a 3x3 matrix, you must recursively compute individual determinants of 2x2 sub-matrices using the following method.

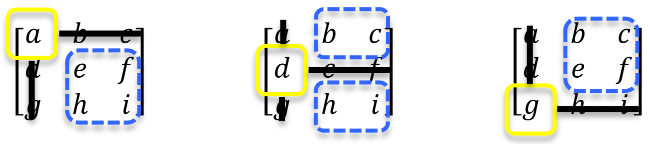
A single column of the matrix is chosen. Each element of that column is called a “cofactor”. There is a 2x2 sub-matrix corresponding to each cofactor. This sub-matrix is defined as all elements of the array not in the row or column of its cofactor. Note that this is always a square array. The determinants of each of these 2x2 sub-matrices are multiplied by their corresponding cofactors, and then added together to compute the overall determinant. When adding, the sign (+/-) of each cofactor is determined by the formula:

$$(-1)^{(cofactor\ row\ num + 1)}$$

Computing the determinant of a 3x3 matrix:

$$A = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} \quad \rightarrow \quad \det(A) = a \times \det\left(\begin{bmatrix} e & f \\ h & i \end{bmatrix}\right) - d \times \det\left(\begin{bmatrix} b & c \\ h & i \end{bmatrix}\right) + g \times \det\left(\begin{bmatrix} b & c \\ e & f \end{bmatrix}\right)$$

$$A = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} \rightarrow break\ down \rightarrow$$



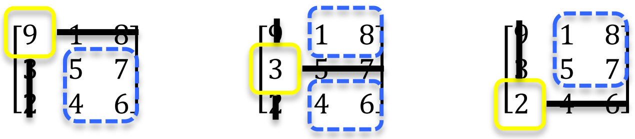
Here, ***a***, ***d***, and ***g*** are the cofactors for each 2x2 matrix

For example:

$$A = \begin{bmatrix} 9 & 1 & 8 \\ 3 & 5 & 7 \\ 2 & 4 & 6 \end{bmatrix} \quad \rightarrow \quad \det(A) = 9 \times \det\left(\begin{bmatrix} 5 & 7 \\ 4 & 6 \end{bmatrix}\right) - 3 \times \det\left(\begin{bmatrix} 1 & 8 \\ 4 & 6 \end{bmatrix}\right) + 2 \times \det\left(\begin{bmatrix} 1 & 8 \\ 5 & 7 \end{bmatrix}\right)$$

Drill Problem: #4

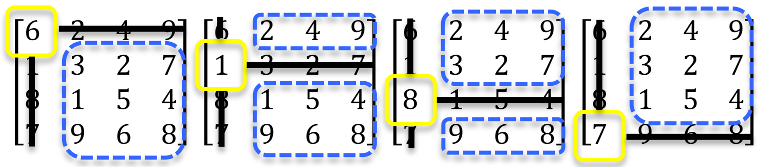
$$A = \begin{bmatrix} 9 & 1 & 8 \\ 3 & 5 & 7 \\ 2 & 4 & 6 \end{bmatrix} \rightarrow \textit{break down} \rightarrow$$



Similarly, when computing the determinant of a 4x4 or larger matrix, you must **recursively** perform cofactor expansion until you have only 2x2 sub-matrices.

For example, a 4x4:

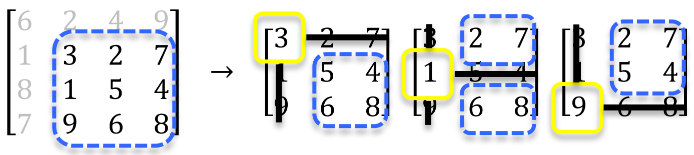
$$A = \begin{bmatrix} 6 & 2 & 4 & 9 \\ 1 & 3 & 2 & 7 \\ 8 & 1 & 5 & 4 \\ 7 & 9 & 6 & 8 \end{bmatrix} \rightarrow \textit{break down} \rightarrow$$



$$\downarrow$$

$$\textit{break down each 3x3 into smaller 2x2s}$$

$$\downarrow$$



Here, **6, 1, 8,** and **7** are the cofactors for each 3x3 sub-matrix. **3, 1,** and **9** are the cofactors for each 2x2 sub-matrix **within** the **first** 3x3 sub-matrix.

$$\det(A) = 6 \times det\left(\begin{bmatrix} 3 & 2 & 7 \\ 1 & 5 & 4 \\ 9 & 6 & 8 \end{bmatrix}\right) - 1 \times det\left(\begin{bmatrix} 2 & 4 & 9 \\ 1 & 5 & 4 \\ 9 & 6 & 8 \end{bmatrix}\right) + 8 \times det\left(\begin{bmatrix} 2 & 4 & 9 \\ 3 & 2 & 7 \\ 9 & 6 & 8 \end{bmatrix}\right) - 7 \times det\left(\begin{bmatrix} 2 & 4 & 9 \\ 3 & 2 & 7 \\ 1 & 5 & 4 \end{bmatrix}\right)$$

Compute the determinants of each 2x2 sub-matrix within each 3x3 sub-matrix, and use the method for a 3x3 determinant to compute the determinants of each 3x3 sub-matrix. Then add together the determinants of each 3x3 sub-matrix (each multiplied by the corresponding cofactor) to compute the overall determinant of the 4x4 matrix.

If all of this sounds horribly confusing, have no fear! Because recursion is awesome you only have to code the 2x2 case (base case) and then make sure you approach this case when you iterate across the cofactors. All of this is done **recursively**!

Notes:

- You must use recursion to solve this problem!
- You may **not** use the built-in `det()` or `eig()` functions.
- The input will always be a square matrix (same number of rows and columns).
- This is a very brute-force determinant algorithm so it is best not to try it on large matrices.
- Input matrices will be 2x2 or larger.

Drill Problem: #5

Function Name: fountainOfYouth

Inputs:

1. (*char*) A character array of `'T'`s, `' '`s (spaces), and, possibly, a single `'X'`

Outputs:

1. (*logical*) Whether or not you can reach the `'X'`

Function Description:

The search for the Fountain of Youth contributed to the beginnings of overseas exploration and still remains a mystery to this day. The promise of infinite youth was coveted among explorers and is still questioned by today's most esteemed CS1371 professors.

Your job is to write a MATLAB function called `fountainOfYouth()` that will determine whether you can find the hypothetical Fountain of Youth from a given array of characters. This array of characters consists of `'T'` and `' '` (space, ASCII 32) characters, as well as a single `'X'` character that may be found near the center of the array. Your job is to check each outer layer of the array, and if you can find an opening (a `' '` character), then you can keep searching closer to the center of the array and, hopefully, reach the `'X'` that marks the spot of the desired Fountain of Youth!

A couple of sample arrays are displayed below with the openings highlighted in yellow:

7X7							8X8							
'T'	'T'	'T'	' '	'T'	'T'	'T'	'T'	'T'	'T'	'T'	'T'	'T'	'T'	'T'
'T'	' '	' '	' '	' '	' '	'T'	'T'	' '	' '	' '	' '	' '	' '	'T'
'T'	' '	'T'	'T'	'T'	' '	'T'	'T'	' '	'T'	'T'	' '	'T'	' '	'T'
'T'	' '	'T'	'X'	'T'	' '	'T'	'T'	' '	'T'	' '	' '	'T'	' '	'T'
'T'	' '	'T'	'T'	'T'	' '	'T'	'T'	' '	'T'	'X'	' '	'T'	' '	'T'
'T'	' '	' '	' '	' '	' '	'T'	'T'	' '	'T'	'T'	'T'	'T'	' '	'T'
'T'	'T'	'T'	'T'	'T'	'T'	'T'	' '	' '	' '	' '	' '	' '	' '	'T'
							'T'	'T'	'T'	'T'	'T'	'T'	'T'	'T'
output -> false							output -> true							

Note that every other interior layer is guaranteed to be completely filled with spaces. If an opening is found, then you must recursively check the next interior layer of `'T'`s for an additional opening in order to keep moving towards the center of the given array. Once you have reduced the array to either a 2x2 or a 1x1, check if an `'X'` is contained in the array. If an `'X'` character is found, your output for the function should be `true`. If you cannot reach the inner part of the array, or you cannot find an `'X'` once you get there, then your output is `false`.

Drill Problem: #5

Notes:

- You must use recursion to solve this problem.
- Every other interior layer is guaranteed to be completely filled with `' '` characters.
- You are guaranteed to be given a square array with `'T'` characters along the outside.
- The `'X'` character (if it exists) will never be outside of the inner 2x2 sub array.
- The character `'T'` was chosen to represent the passing of time, if you are curious.

Drill Problem: EXTRA CREDIT

Function Name: flood

Inputs:

1. (*double*) A vector of integers representing building heights

Outputs:

1. (*double*) The amount of water that can be held between the buildings

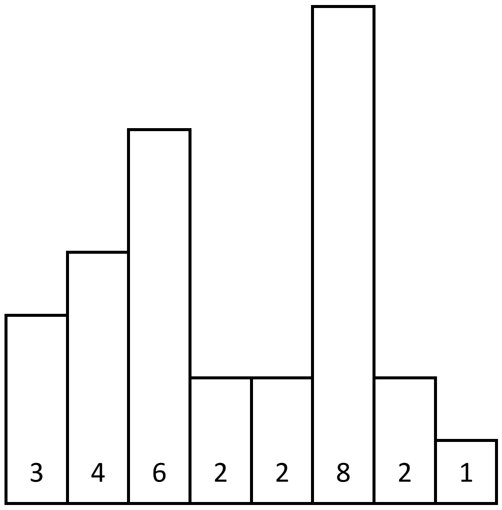
Function Description:

Apocalyptic movies are all the rage these days. Massive earthquakes, alien invasions, zombies; it has all been in theaters at one point or another. You, however, have an idea for a completely original movie where an entire city gets flooded. No one has ever done that before! Unfortunately, your budget has run out so you are going to have to cut back on special effects. More specifically, you are going to have to do your special effects in 2D. You still need to know how much water it's going to take to flood your CGI city, though, so you write a MATLAB function that takes in the heights of the buildings of your 2D city and outputs how much water can be held between them (see illustration). All of the buildings are 1 unit wide and the input to the function specifies their height (in units). All of the buildings are water-tight and there are no gaps between buildings (however, gaps could be represented by a "building" with a height of 0).

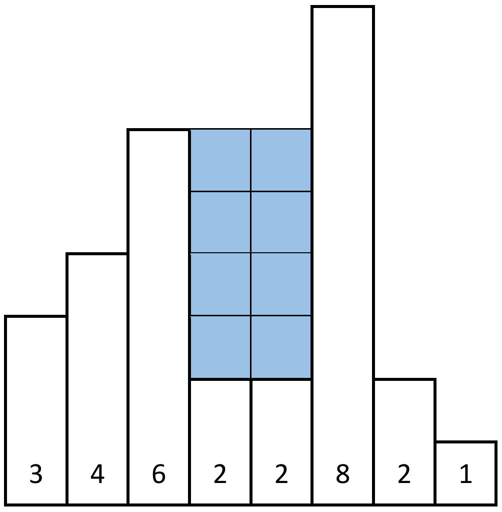
Example Illustrations:

```
input = [3, 4, 6, 2, 2, 8, 2, 1]
```

represents a city that looks like:



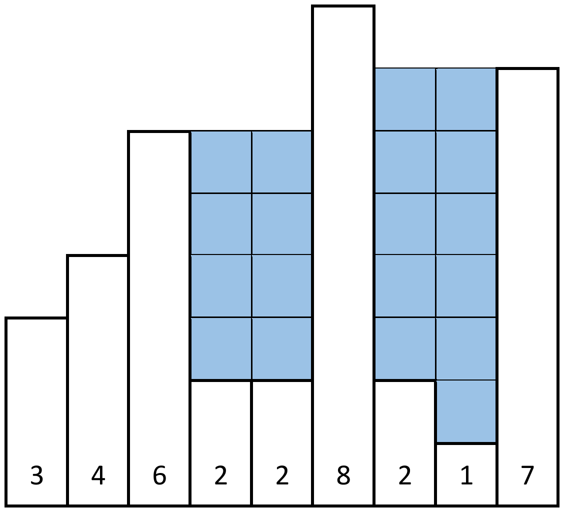
And can hold 8 units of water:



So, the output would be 8.

However, if a building of height 7 is added to the end, then the output becomes 19 because water can be trapped between buildings 8 and 7:

```
Input = [3, 4, 6, 2, 2, 8, 2, 1, 7] → Output = 19
```



The input building heights can contain any number of separate pools of water, or none at all (in which case the output would be 0).

Notes:

- Buildings can have a height of any integer value greater than or equal to zero.
- Test your function with lots of edge cases!

Hints:

- A carefully created base case for this problem can make the recursive call surprisingly simple.
- Think about what sort of height pattern is guaranteed to hold water regardless of other buildings.

Drill Problem: EXTRA CREDIT

Function Name: sixDegreesOfWaldo

Inputs:

1. (*struct*) 1x1 Structure representing a person.

Outputs:

1. (*logical*) Whether Waldo can be found.
2. (*double*) The degrees of separation from the starting person to Waldo.

Function Description:

After spending your childhood years desperately scanning page after page for the Hide-And-Seek World Champion (also known as the infamous Waldo), you decide enough is enough. You are going to use MATLAB to help you decide once and for all if Waldo is hiding among us.

Write a function called `sixDegreesOfWaldo()` that takes in a 1x1 structure representing a single person. The input structure is guaranteed to have the following fields:

1. `Name`: (String representing the name)
2. `Age`: (double representing the age)
3. `Friends`: (1xN Structure array representing all of this person's friends)

Note that for each friend, they are guaranteed to have the above 3 fields, but may have any number of extra fields in addition to the 3 above.

Your job is to **recursively** search through all of the people available by checking all of the first person's friends, then each of their friends, and so on.

The first output of the function will be a single logical indicating if Waldo is a friend of anyone in the group (e.g. if Waldo is present). Unfortunately, Waldo is sneaky and doesn't always put `'Waldo'` in the Name field, so **the first output of the function should be true if ANY field of ANY of the people in the group contains the string** '`Waldo`'**.** Note that the spelling and case must appear exactly as shown.

The second output of the function will be a double representing the degrees of separation from the first person to Waldo. **If Waldo does not exist in the group, this output should be 0. The degrees of separation is the number of levels of friends you have to move through (from the beginning person) to find Waldo.** If the first person is Waldo, then the degrees of separation will be 0, if (instead) a friend of theirs is Waldo, the degrees of separation will be 1, if (instead) a friend of a friend of the first person is Waldo, the degrees of separation will be 2, and so on.

Notes:

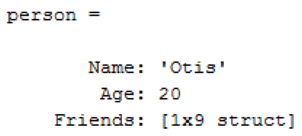
- There will never be more than 1 Waldo in the group.
- Do not hardcode the field names for the friends: there is no guarantee how many there will be, nor what they will be called.
- You **must use recursion** to solve the problem**.**

Hints:

- Think about the best way to keep track of the results of each friend as you traverse the group.

Visual Example:

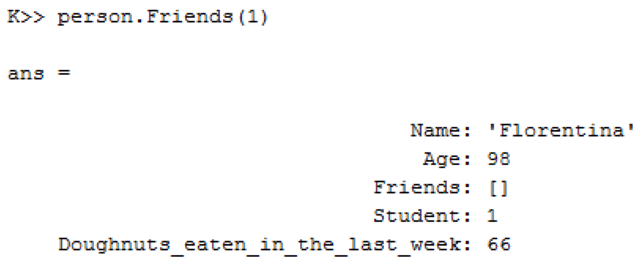
The following is a possible input to the function:



```
person =

       Name: 'Otis'
        Age: 20
    Friends: [1x9 struct]
```

This person has 9 total friends. The first of these friends looks like this:

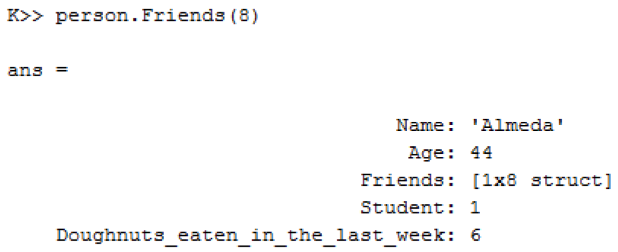


```
K>> person.Friends(1)

ans =

                                Name: 'Florentina'
                                 Age: 98
                             Friends: []
                             Student: 1
    Doughnuts_eaten_in_the_last_week: 66
```

The above person has no friends in the list. Since they are not Waldo, we are done searching for this friend. The 8th friend of our original person looks like this:

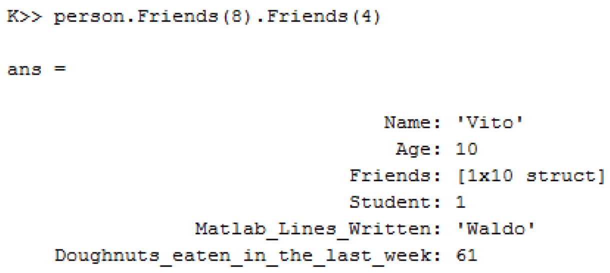


```
K>> person.Friends(8)

ans =

                                Name: 'Almeda'
                                 Age: 44
                             Friends: [1x8 struct]
                             Student: 1
    Doughnuts_eaten_in_the_last_week: 6
```

The above person has 8 friends, each of which need to be searched if they are (or have any friends who are) Waldo. Now let's look at Almeda's 4th friend (Almeda is the person above):



```
K>> person.Friends(8).Friends(4)

ans =

                                Name: 'Vito'
                                 Age: 10
                             Friends: [1x10 struct]
                             Student: 1
                Matlab_Lines_Written: 'Waldo'
    Doughnuts_eaten_in_the_last_week: 61
```

The `'Matlab_Lines_Written'` field has a value of `'Waldo'` which means this person is Waldo in disguise. In this case, the *first output of your function would be* `true`.

The *second output of your function would be* `2` since we had to go through two layers of friends to reach Waldo (The first layer was Almeda, the second layer was Waldo himself – since he is a friend of Almeda).

Drill Problem: EXTRA CREDIT

Extra Help Provided:

Function Name: generatePeople (already written)

Inputs:
None

Outputs:
1. (struct) Structure representing a person

In order to help you test your function (`sixDegreesOfWaldo`), we have provided the following code for you as a .p file. This function generates a possible input for `sixDegreesOfWaldo`. Note that the generation is completely random – it is only made to help you test your code. *It does not guarantee any kind of score after grading*.

To use it, simply run the function. There are no inputs, and only one output – a sample input to the sixDegreesOfWaldo function. You can use the output of this function as an input to both the solution file and your own code to help you test your `sixDegreesOfWaldo` code.

These are in supplement to the test cases already provided in the hw11.m file.

Note:

- **Do not** include the `generatePeople` function in your `sixDegreesOfWaldo` function. ***It will result in a score of 0 for your entire homework 11 assignment***.