Function Name: `recursiveSum`

Inputs:

1. (*double*) a vector of length *n*

Outputs:

1. (*double*) the sum of the vector's elements

Banned Functions:

`sum()`

Background:

You've now trained to be master at MATLAB. To show off your new skills, you decide to write your own sum function in place of MATLAB's built-in one.

Function Description:

Given a vector, recursively determine the sum of the vector. For example, if the vector was [5, 7, 9, 3], the sum of it would be 24.

Notes:

- You MUST use recursion to solve this problem.

Function Name: `cheerfulDigits`

Inputs:

1. (*double*) A number to determine whether its digits are happy

Outputs:

1. *(char*) A character vector describing whether or not the input is happy

Background:

A number is a "happy number" if the sum of the squares of its digits adds up to one. If the sum of the squares of the digits of a number result in another number, and the sum of that second number's squares adds up to one, then the original number was happy. And so on!

Function Description:

Write a function to determine whether or not a number is happy. A number is happy if the sum of the squares of its digits is happy. The number `1` is happy. When numbers are not happy, trying to find the sum of squares of the digits causes an infinite loop. If the sum of the squares of the digits is ever the number `4`, then you have reached one of these infinite loops and know that the number is not happy.

If the number is happy, your output the following:

`'Wow! <num>'s digits sure are cheerful.'`

If the number is unhappy, output this instead:

`'Sadly, <num>'s digits aren't very cheerful.'`

Example:

```
cheerfulDigits(1) => 'Wow! 1's digits sure are cheerful.'
```

$1^2 \rightarrow 1$

```
cheerfulDigits(20) => 'Sadly, 20's digits aren't very cheerful.'
```

$2^2 + 0^2 \rightarrow 4$
$4^2 \rightarrow 16$
$1^2 + 6^2 \rightarrow 37$
$3^2 + 7^2 \rightarrow 58$ ...

...
$5^2 + 8^2 \rightarrow 89$
$8^2 + 9^2 \rightarrow 145$
$1^2 + 4^2 + 5^2 \rightarrow 42$
$4^2 + 2^2 \rightarrow 20$...

Notice that you end up at 20 again, meaning that this loop would be infinite if we kept trying to find the sum of the squares of digits of 20. Instead, we know that if the sum is ever `4`, we are about to enter an infinite loop and can decide that the number is unhappy.

Notes:

- Your function must be recursive.

Hints:

- Wondering how to do math on a number to get a specific digit? Throwback to the `sepDigits()` problem from homework 2!
- Consider writing a wrapper function and a helper function, with the helper function being the recursive one.

Function Name: `r_nFib`

Inputs:

1. (*double*) A non-negative number to begin the sequence
2. (*double*) A non-negative integer (n) denoting the number of terms to return

Outputs:

1. (*double*) A 1xN vector of the resulting Fibonacci sequence

Function Description:

The Fibonacci sequence is very important in mathematics, physics, nature, life, etc. Each number in the sequence is the sum of the previous two values, where the first two numbers in the sequence are always 0 and 1, respectively.

Write a MATLAB function that puts a twist on the classic Fibonacci sequence. This function will input a number to begin the sequence and the number of terms of the Fibonacci sequence to evaluate, and it will output a vector of the corresponding sequence. If the initial term is a 0 or 1, the second term will be a 1; if the initial term is any other number, the second term will be that initial number, repeated.

Therefore, one can find the sequence for 6 terms, beginning at the number 2, with the output 1xN vector of the entire sequence to be:

```
out =  [2 2 4 6 10 16]
```

Notes:

- You will not have any negative input values.
- You **MUST** use recursion to receive credit for this problem.

Hints:

- You may find a helper function useful.

Function Name: collatz

Inputs:

1. (*double*) any positive integer

Outputs:

1. (*double*) the number resulting from the algorithm
2. *(double)* the number of recursive steps required by the algorithm

Function Description:

The Collatz conjecture (also known as the 3n+1 conjecture) was proposed by Lothar Collatz in 1937. 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
- repeat (i.e. recursively call the function)

Your job is to write a recursive MATLAB function that implements the Collatz conjecture. Your function should output the final number that the algorithm reaches as well as the number of recursive calls it took to get there.

Notes:

- If you try running this function with large numbers (like 40-digit numbers), MATLAB will crash as it will reach its maximum recursion limit.

Hints:

- To keep track of the number of recursive calls, consider making a helper function whose inputs are the current number and the number of recursive calls that have been made.

Function Name: `determinant`

Inputs:

1. (*double*) An MxM matrix

Outputs:

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

Banned Functions:

`det(), eig()`

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 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:

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

Here is an example of computing the determinant of a 3x3 matrix using cofactor expansion along the first column:

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

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

Continued...

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

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

Similarly, when computing the determinant of a 4x4 or larger matrix, you must recursively perform cofactor expansion until you have only 2x2 submatrices and can use the formula for a 2x2 matrix.

For a 4x4 matrix, you would have to 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 (each multiplied by the corresponding cofactor) to compute the overall determinant of the 4x4 matrix. Since this problem can be solved recursively, you only have to compute the 2x2 sub-matrix as a base case and approach this case while iterating across the cofactors.

Notes:

- You must use recursion to solve this problem!
- The input will always be a square matrix.
- This is a very brute-force determinant algorithm, so it is best not to try it on large matrices or it will take a very long time to run.

Extra Credit

Function Name: `luigisMansion`

Inputs:

1. (*struct*) MxN structure array representing a haunted mansion
2. (*double*) 1x2 vector of your starting position given as [row, column]

Outputs:

2. *(double*) number of rooms you passed through to escape

Background:

You and your friends are super excited because the newest version of Luigi's Mansion has come out for the Nintendo Switch! You open the package, pop the game in and start playing! Everything is going great until you realize that the goal of the game is not to capture ghosts, but instead, escape the mansion with your life! Luckily you know MATLAB and can use your skills to write a function to help you beat the game.

Function Description:

The goal of this function is to find a path through the mansion to reach the exit. You are given an MxN structure array where each individual structure represents a room in the mansion. Each structure has the fields **`'West'`**, **`'North'`**, **`'East'`**, and **`'South'`**, representing the different sides of the room that you can attempt to move through. Each field will have **one** of the following character vectors as its value: **`'start'`**, **`'door'`**, **`'back'`**, **`'wall'`**, or **`'exit'`**. The values of these fields will dictate how you move through the mansion.

Your function should output the total number of rooms you pass through (including your starting room) to reach the exit. You should recursively move through the mansion as follows:

1. Always check the sides of the room for a door to go through in the order: **West, North, East, South** regardless of which field contains **`'start'`**.
2. If you find a **`'door'`**, you should go through it (recursively) to the next room and repeat the process of searching for the exit.
3. If you encounter a **`'wall'`** or the strings **`'start'`** or **`'back'`**, you should check the next side of the room, since you cannot make forward progress by going through any of those.
4. Finding the **`'exit'`** means you have escaped and whatever path you took to reach it, is your final path.
5. If you encounter a room with no doors, then you have reached a dead end and must return to the previous room and try the next side of the room.

Continued...

Example:

Given the following structure array `mansion =`

start wall door door	wall back door door	wall back wall door
back wall wall exit	back wall wall wall	back wall wall wall

and `startingPosition = [1 1]`, find the number of rooms you pass through.

You function should output **`out = 2`**, since you have to pass through two rooms to reach the exit.

Notes:

- There will only be one path leading to the exit.
- The strings **`'start'`** and **`'back'`** are for helping you keep track of how you have moved through the mansion thus far by signifying that you are in the starting room or that a door you have just passed through is on that side of the room.

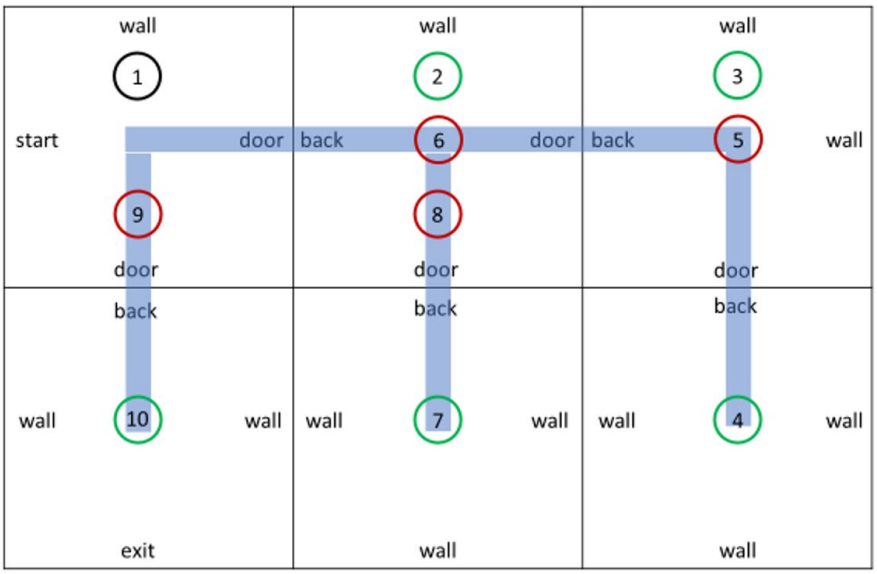
Hints:

- Think of your path as going up and down the stack, where the number of rooms you went through is the total number of “tiers” on the stack that ends with the exit.
- You should put iteration in your recursive call.
- Try using a wrapper function to help keep track of the number of rooms you have passed through.

Continued...

Example:

A solution to a this problem will trace through the example problem as follows:



During the first call to your function, you should start at the location (1,1). You would loop through each side of the room (loop through the fields) until you came to the first door. Your second call would move through the door, to number 2, at the location (1,2). You would again loop through the sides of the rooms until you come to the first door. Your next calls would loop through successive rooms until you reach position 4. There are no doors, so you would have to go back to the previous room (up the stack) and check the rest of the sides of that room for an open door. Since there are none, you again move back a room (6) and check for more doors. Since there's another door, move through and check for more doors (7). Since there are no doors, move back to the previous rooms (8). Finish checking for doors in that room. Since there are no more doors, and you have not reached the exit, go back to the previous room (9). Continue to check for doors or the exit. Proceed through the last door (10) to the final room. Here, as you loop through the sides of the room, you find **`'exit'`** (10), so your call to that room should immediately end and you should count up the number of rooms from the start to your exit.

Extra Credit

Function Name: `mobyDick`

Inputs:

1. *(char*) Filename of the first shelf to look in, including a file extension

Outputs:

1. (*char*) The filename of the shelf where *Moby-Dick* was found

Background:

At last, Thanksgiving break is coming up! What better way to enjoy it than to visit your local library? You arrive and realize, to your horror, that the library is a complete mess! In fact, one could say the entire house is in dis***array***. Books are all over the ***floor()*** and there's hardly any (***lin***)space to move around. There doesn't appear to be any ***logical*** ordering to the books anymore. You can't ***find()*** anything!

Unfortunately, it seems that you'll be spending your break cleaning up the library. Books are out of alphabetical order, some are on the wrong shelves, and, worst of all, your favorite ~~maritime~~ ~~adventure~~ ~~novel~~ whaling encyclopedia, *Moby-Dick*, is nowhere to be found! Luckily, you are a MATLAB guru, so you decide to employ MATLAB to help you find this book in the disorganized library.

Function Description:

The books in your library are organized alphabetically (by title) into shelves, where each shelf is represented by a plain text file named `<letter>.txt`. Each shelf used to contain only books whose titles began with the corresponding letter, but now they are disorganized so that some shelves contain books belonging to other shelves.

Your task is to search through the shelves for Herman Melville's *Moby-Dick*, starting with the shelf specified by the input. If the first shelf doesn't contain *Moby-Dick*, then look in that shelf for books that don't belong. For every book that doesn't belong on that shelf, look on its corresponding correct shelf to search for *Moby-Dick* there. Repeat the process until you either find *Moby-Dick* or until you've searched all of the possible shelves and didn't find *Moby-Dick* in any of them.

Continued...

Example:

Suppose the first shelf to look in is `m.txt`, and that file contains the following:

```
Macbeth - William Shakespeare
Madame Bovary - Gustave Flaubert
Magician's Nephew, The - C.S. Lewis
Man Who Mistook His Wife for a Hat, The - Oliver Sacks
Fear and Loathing in Las Vegas - Hunter S. Thompson
```

The last book's title (highlighted in green) starts with an `F` and not an `M`, so it's on the wrong shelf. Since *Moby-Dick* isn't here, but there is a book from the `f.txt` shelf, you should now look at the `f.txt` shelf. Suppose that shelf contains the following:

```
Fahrenheit 451 - Ray Bradbury
Fall of Hyperion, The - Dan Simmons
Fantastic Beasts and Where to Find Them - Newt Scamander
Moby-Dick - Herman Melville
Fast Food Nation - Eric Schlosser
Perry's Chemical Engineers' Handbook - R.H. Perry and D.W. Green
```

This shelf contains *Moby-Dick*, so you are done looking; your function should then return `'f.txt'`.

Notes:

- Shelves are always named by a single lowercase letter.
- If you don't find *Moby-Dick* in any of the shelves, output an empty string (`''`) for the shelf.
- Make sure not to attempt to search shelves more than once.
- Don't forget the hyphen in the title of *Moby-Dick*!
- There might not be a shelf for every letter of the alphabet, but it is guaranteed that there will not be a book with no corresponding shelf.
- The number of shelf files you potentially have to search through is not guaranteed, besides the limit of 26 letters in the alphabet.

Hints:

- This is on the recursion homework for a reason!

Extra Credit

Function Name: `towersOfHaynoi`

Inputs:

1. *(double)* The number of hay bales to move
2. (*char*) The name of the starting platform
3. (*char*) The name of the destination platform
4. (*char*) The name of the helper platform

Outputs:

(none)

File Outputs:

1. A text file with the instructions to move the bales of hay from the starting platform to the destination platform

Function Description:

While visiting the local farm for the pumpkin patch, corn maze, and other fun Fall activities, the farmer asks for your help! He needs to move a stack of hay bales from one platform to another and doesn't know the best way to do it. The hay bales are all different sizes, and the starting state is the stack of hay on the starting platform with the smallest bale on top and the largest on the bottom and they are all in order by size (think like a tiered cake). The goal state is the same order of hay bales but all on the destination platform. There are, however, some restrictions on the task. First, the farmer can only lift one bale at a time, so he can only move one at a time. Second, a larger bale cannot be stacked on top of a smaller hay bale, or it might be unbalanced. Third, there is a third platform besides the starting and destination platforms that the farmer will leverage to make this task doable.

You decide to use your MATLAB skills to help the farmer out! You will write a function called `towersOfHaynoi` to give the farmer instructions. The function will take in the number of bales to move and the names of the three platforms and will write a text file with instructions on how to move the hay bales. The name of the text file will be

```
Instructions_<Name of start platform>to<Name of destination platform>.txt
```

The contents of the file will be formatted as follows: The first line will be of the form:

```
Instructions for moving <number> bales of hay from <start platform> to
                        <destination platform>:
```

The second line will be blank, and all subsequent lines will be the instructions, formatted as:

```
Move top bale from <start platform> to <destination platform>.
```

Note that the start and destination platforms in the first line are from the inputs (the overall start and destination platforms), and the start and destination platforms in the instructions indicate the other movements. For example, if the function is called as `towersOfHaynoi(2, 'A', 'B', 'C')`, your function should produce a file called `Instructions_AtoB.txt` with the following content:

```
Instructions for moving 2 bales of hay from A to B:

Move top bale from A to C.
Move top bale from A to B.
Move top bale from C to B.
```

There is a blank line in between the first line and the rest, but no blank line at the end.

Notes:

- Your solution must produce the optimal number of moves to move the stack of hay
- If you are not sure if you are getting the optimal number, your solution should have $2^n - 1$ moves, where n is the number of bales.
- Because of the exponential nature of how the number of instructions increases as the number of bales increases, know that if you try to test your function with large n, it may take a very long time--if you have 20 bales it will take over 1 million moves. If you have 50 bales it will take over 10^{15} moves. If your computer could calculate one move every microsecond, this would take about 35 years to compute! You will not be tested with any inputs nearly that large, and though you are free to try to test your code however you please, this is your warning.

Hints:

- To move a large stack of hay, remember that you cannot move a larger stack on top of a smaller one, which means that the biggest stack can only be moved to an empty platform.
- You will need a helper function for this problem.
- Try to avoid thinking about how you would construct a strategy for moving the bales and instead try to think of a recursive pattern.