Function Name: `unnest`

Inputs:

1. (*cell*) 1xN cell array

Outputs:

1. (*cell*) 1xM cell array, with no nested cells

Function Description:

Given a cell array containing nested elements, use recursion to go through the levels of the cells and bring all of the non-cell values to the top layer (wrapped in only the outer cell array). Any cell can contain a 1xP vector of values (of any data type).

Example:

```
ca = { {3, {true}}, {{'ab', {[1, 2, 3, 4]}}} }  (1x2 cell array)
out = unnest(ca)
out => { 3, true, 'a', 'b', 1, 2, 3, 4 }         (1x7 cell array)
```

Notes:

- For a cell containing a vector of non-cell values, each scalar value of the vector should have its own cell in the output cell array.
- You must use recursion to solve this problem.

Hints:

- The `num2cell()` function works on arrays of doubles, chars, and logicals.

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.

For example, the first 6 terms of the modified Fibonacci sequence beginning at the number 2 are the following:

```
out = r_nFib(2, 6)
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: `recursiveGetPi`

Inputs:

1. *(double)* Number of terms to use in approximation

Outputs:

1. *(double)* Result of the approximation

Banned:

`pi()`

Background:

You are a mathlete taking the International Math Olympiad in 1988. You have arrived at problem 6, which asks you to prove that for $a, b \in \mathbb{N}$ such that $ab + 1$ divides $a^2 + b^2$, the number $(a^2 + b^2)/(ab + 1)$ is a perfect square. After some thinking, you come up with an inventive way to use Vieta's formulas to solve the problem, which you decide to call Vieta Jumping. While writing down your proof, you decide to think about some of Vieta's other contributions to mathematics, including one of the first ways of calculating π...

Function Description:

There are many ways to calculate the mathematical constant π. One of the ways to do it is based on a product of nested radicals. This fact will allow you to calculate π:

$$\frac{2}{\pi} = \frac{\sqrt{2}}{2} \cdot \frac{\sqrt{2+\sqrt{2}}}{2} \cdot \frac{\sqrt{2+\sqrt{2+\sqrt{2}}}}{2} \cdots$$

Each term is the square root of the quantity 2 plus the numerator of the previous term all divided by 2. The first term is always $\frac{\sqrt{2}}{2}$. Write a function that takes in the number of terms to multiply together and approximates π using the product of the terms. The function must be recursive.

Example:

```
>> recursiveGetPi(3) => 3.1214
```

$$\frac{2}{\pi} = \frac{\sqrt{2}}{2} \cdot \frac{\sqrt{2+\sqrt{2}}}{2} \cdot \frac{\sqrt{2+\sqrt{2+\sqrt{2}}}}{2} = 0.6407 \implies \pi = 3.1214$$

Notes:

- The input will always be a positive integer.
- Please round to 10 decimal places.
- You must use recursion to solve this problem

...Continued

Hints:

- A recursive helper function may be useful to calculate each nested radical
- Look back at your original `getPumpkinPi` function from Week 3. Try using the same input in both functions and see which one is closer to π.

Function Name: `determinant`

Inputs:

1. (*double*) An MxM matrix

Outputs:

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

Banned Functions:

`det(), eig()`

Function Description:

The determinant of a square matrix greater than 1x1 can be determined via cofactor expansion. For a 1x1 matrix, the determinant is the single element. For example, in a 3x3 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:

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

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

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

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

You can repeat this process each of the 2x2 matrices, performing cofactor expansion until you have only 1x1 matrices, the determinants of which are just the single value they contain.

Similarly, when computing the determinant of a 4x4 or larger matrix, you must recursively perform cofactor expansion until you have only 1x1 submatrices.

Continued...

For a 4x4 matrix, you would have to compute the determinants of each 1x1 sub-matrix within each 2x2 sub-matrix within each 3x3 sub-matrix.

Notes:

- You must use recursion to solve this problem!
- Calculating cofactor expansion down any column or across any row of a square matrix will always result in the same answer.
- 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.

Function Name: `mondrian`

Inputs:
1. *(char)* 2^N x 2^N character array

Outputs:
1. *(char)* 1x1 character
2. *(double)* Path taken through the array

Background:

While at the Museum of Modern Art (MoMA) in New York City, you come across the painting *Broadway Boogie Woogie*, a painting by Piet Mondrian. Mondrian was famous as a pioneer of abstract art as part of the De Stijl art movement in the early 20th century. His paintings are known for their use of geometric shapes painted in solid colors. Inspired by his work, you decide to create your own abstract ASCII art, but the meaning of it eludes you. Stumped, you turn to MATLAB to help you analyze your ASCII art...

Function Description:

You will be given a 2^N x 2^N character array, which can be separated into four evenly sized quadrants. Three of the quadrants will contain all the same characters. The fourth will contain the same four-quadrant setup, with three-quarters of that quadrant using the same character while one quadrant is different. Recursively go through the array until you are left with one character (which should not appear anywhere else in the array), and output that character as the first output. As the second output, output a double vector of the quadrants that you traced through while searching the array.

- Top left is quadrant 1
- Bottom left is quadrant 2
- Top right is quadrant 3
- Bottom right is quadrant 4

Example:

```
art1 =>
    'yyyyyyyy'
    'yyyyyyyy'
    'yyyyyyyy'
    'yyyyyyyy'
    'ccccyyyy'
    'ccccyyyy'
    'ccI2yyyy'
    'cc22yyyy'
```

...Continued

```
[abstract1,path1] = mondrian(art1) => abstract1 = 'I', path1 = [2,4,1]
```

In the original 8x8 image, quadrants 1,3, and 4 all contain the letter `'y'`, while quadrant 2 does not. Now consider quadrant 2. Within quadrant 2, the quadrants 1, 2, and 3 all contain the character `'c'`, while quadrant 4 does not. Now consider that quadrant 4. Within that quadrant 4, the quadrants 2, 3, and 4 all contain the character `'2'`, but quadrant 1 does not. Now consider that quadrant 1. Since it is a single character `'I'`, that is the first output. The second output is the order of the quadrants: `[2,4,1]`.

Notes:

- No character will be used on two different levels within the array.
- You must use recursion to solve this problem.

Hints:

- Try checking the middle four characters of the array to easily determine which quadrant contains the nested quadrant setup.

Extra Credit

Function Name: `anagram`

Inputs:

1. (*char*) String representing the anagram
2. (*char*) Name of text file to write to

Outputs:

none

File Outputs:

1. A text file containing all the permutations of the anagram

Banned Functions:

`perms(), permute(), ipermute(), nchoosek(), combnk()`

Background:

You guys know what do. You are professional MATLAB now, you got this!

Function Description:

Given a string, recursively determine all the unique ways to rearrange the letters. (i.e all the unique permutations). For example, if the string was `'abc'`, the list of permutations would be `{'abc','acb','bac','bca','cab','cba'}.`

Write each permutation as a separate line in a text file, in alphabetical order. Case should be ignored and all letters should be lowercase in the output text file. Non-letter characters may occur in the string and should not be included. Characters may be repeated multiple times. There should be no newline character at the end of your file.

Notes:

- Do not attempt to run this function with char vecs longer than about 10. If you do, you may be sitting around waiting a while.
- The fourth test case will likely take a while, but all grading test cases will be shorter.

Hints:

- The `unique()` function will be useful.
- You may need a wrapper function.