Some Recursion Exercises

A recursive function definition is one that includes a call to the function being defined. (It looks like circular reasoning, something that shouldn't make sense or be allowed.) These exercises are intended to help you understand how to use it and how it works.

1. Run the first function countdown in the posted code by typing

   ```
countdown(10)
   ```

   in the Shell. Note that although this function does something repeatedly, which you might ordinarily encode using a for or a while statement, the code contains no such looping structure. Instead it is based on the following recursive definition of counting down from n:

   To count down from n:
   If \( n > 0 \), write \( n \) and count down from \( n - 1 \)
   Otherwise, do nothing

2. A common error is to forget to specify what happens when the recursion 'bottoms out'—in this example, it is the 'do nothing' in the above description. What do you think will happen if you eliminate the if \( n > 0 \) line in the code? Test your guess by commenting out this line and running the experiment again. Be prepared to kill the program!

3. (The 'stack', part 1.) Restore the commented-out line, but change the function call inside the code from `countdown(n-1)` to `countdown(n)`.

   What do you think will happen? Test your conclusion by running the experiment again, but this time comment out the print(\( n \)) line, which will make things go faster. The function does not really 'run forever'---what actually does happen is quite instructive.

4. (The 'stack', part 2.) Do the following experiment, to keep track of what is happening while the function is executing: First, restore the function to its original form. Then insert the line

   ```
   print('Calling countdown with n=',n)
   ```

   before the initial if, and add the line

   ```
   print('Returning from countdown with n=',n)
   ```
at the very end, outside the body of the \texttt{if} statement. This will tell us each time the function is called, and each time execution of a call of the function is completed. Before you run the experiment, try to guess what you will see when you do.

5. The second function in the posted code is based on a modified recursive definition of counting down from \textit{n}.

\textit{To count down from \textit{n}:}
\begin{itemize}
  \item \texttt{If} \textit{n}>0, write \textit{n} and count down from \textit{n-1}
  \item Otherwise, print \texttt{"Lift off"}
\end{itemize}

Run this. What will happen if you call \texttt{countdown2(-1)}?

Then try to guess what would happen if you modify the function, replacing

\begin{verbatim}
    else:
        print("Lift off!")
\end{verbatim}

by

\begin{verbatim}
    print("Lift off!")
\end{verbatim}

That is, get rid of the \texttt{else}, but change the indentation so that the \texttt{print} statement is outside the body of the \texttt{if}.

6. Return to the original function. Make a new function called \texttt{countdown3} which, instead of printing out the countdown, returns a list of the values to be printed. So that, for example, if you type

\begin{verbatim}
    print(countdown3(10))
\end{verbatim}

in the Shell, you will see:

\begin{verbatim}
[10,9,8,7,6,5,4,3,2,1]
\end{verbatim}

Remember, the idea is to do this using recursion, so don't just throw out the code and start over with a loop! Think about how to proceed when the recursion 'bottoms out': What should the function return when it is called with 0 as the argument?

7. The last function in the posted code is a (non-recursive) function that returns the smallest factor of an integer > 1. For example, the smallest factor of 15 is 3. Use this to write a recursive function \texttt{prime_factorization(n)} that returns the factorization of an integer \textit{n} into its prime factors. For example, if the argument is 60, then the function should return the list \([2,2,3,5]\).
The core idea is the following recursive definition:

*The prime factorization of \( n \) consists of \( p \) followed by the prime factorization of \( n/p \), where \( p \) is the smallest prime factor of \( n \).*

In other words, the prime factorization of 60 consists of 2 followed by the prime factorization of 30, which is in turn 2 followed by the prime factorization of 15, ...
The problem is that this doesn't tell you how to proceed when the recursion bottoms out and you find yourself trying to compute the prime factorization of 1. The solution is similar to what you will do to solve Problem 6.