CS 120 (Fall 21): Introduction to Computer Programming II

Long Project #8 - Annoying Recursion (Long)

due at 5pm, Tue 19 Oct 2021

1 Overview

In this project, we'll be continuing with a few more "annoying" problems - submit the annoying problems in a file named `annoying_recursion_part2.py` . In addition, you will be writing a few functions that recurse over linked lists; submit these in a file named `linked_list_recursion_part2.py` .

Follow the same rules as the Short problem: "annoying" problems **must** obey the annoying rules; the linked list problems do not. But of course, the linked list problems **must** be recursive! Likewise, just like in the Short problem, ~~loops and helper functions (including default arguments) are banned.~~

1.1 Loops Banned

In every function in this project - whether "annoying" or not - loops are banned! (You are allowed to use string/list multiplication, however.)

1.2 Helper Functions Banned

In every function in this project - whether "annoying" or not - helper functions are banned. Every one of these functions can be completed with only a single, recursive function. (And you have to use exactly the parameters I require, or you won't pass the testcases.)

Some of you may know about default arguments in Python (if not, that's OK) - these are banned as well! Default arguments are a cool feature - but they are basically just a way to write a helper function, so they aren't allowed in this project, either.

2 `annoying_triangleNumbers(n)`

In `annoying_recursion_part2.py` , write a function, `annoying_triangleNumbers(n)`, which returns the sum $1 + 2 + 3... + n$.

Triangle numbers (`https://en.wikipedia.org/wiki/Triangle_numbers`) are calculated by summing up all of the values from 1 to n. Back in Algebra class, you probably learned that you can calculate that sum, quite easily, with the formula $\frac{n(n+1)}{2}$. That's nice - but we're not going to use it. Instead, you're going to write a function that calculates it recursively.

This function must obey the Annoying Requirements.

3 annoying_fibonacci_sequence(n)

In `annoying_recursion_part2.py` , write a function, `annoying_fibonacci_sequence(n)`, which returns the first n numbers in the Fibonacci sequence.

In the previous project, you wrote a recursive function which returned a single integer, which was the n-th Fibonacci number. This time, return an array of length n, which contains that many of the Fibonacci numbers. Thus, if $n = 0$, return an empty array; if $n = 3$ return `[0,1,1]`; if $n = 6$, return `[0,1,1,2,3,5]`.

This function must obey the Annoying Requirements. (Remebmer, the Annoying Requirements don't allow you to use helper functions - so it's not legal to call a function to generate each Fibonaacci number! But remember the definition of Fibonacci numbers - if you had an array with the first $n-1$ Fibonacci numbers, what simple trick could you use to calculate the n-th one?)

4 annoying_valley(n)

In `annoying_recursion_part2.py` , write a function `annoying_valley(n)`, which prints out the shape of a "valley": that is, a bunch of slashes, in the shape of a V pointing right. Do not return anything.

This is the proper output for $n = 5$:

```
..../
.../
../
./
*
.\
..\
...\
....\
```

This is the proper output for $n = 2$:

```
./
*
.\
```

This is the proper output for $n = 1$:

```
*
```

For $n = 0$, print out nothing at all.

This function must obey the Annoying Requirements.

4.1 Hints

Did you notice that **loops are banned** in this project? How, then are you going to print out a whole bunch of periods in the various lines of your valley?

Remember: **string multiplication is allowed.** Use it to create long, repeated sequences! Maybe you don't remember string multiplication? Try out this snippet of code, and see what it does:

```
print("abc"*4)
```

5 array_to_list_recursive(data)

Include this function in the file `linked_list_recursion_part2.py` .

Repeat the problem from the previous project - but this time, you **must** do it recursively. Remember to handle the corner cases!

This function must be recursive, but is **not** required to obey the Annoying Requirements.

5.1 Hints

If you ever find yourself thinking, "How do I put a node on the tail of the list?" you're probably going about this backwards. To recursively build a list, we will usually (if possible) build a smaller list, and then add one more element **at the head.** How would you recurse into the array, in order for that to be possible?

6 accordion_recursive(head)

Include this function in the file `linked_list_recursion_part2.py` .

`accordion_recursive(head)` takes a linked list (which might be empty) as its only parameter, and returns a new linked list, where every other node has been removed. The new list must **use the same nodes** as the original list (don't create any new ones).

When removing nodes from the list, you must remove the old head, the node 2 after the old head, and the node 2 after that, out to the end of the list. For instance, if the input list is like this:

```
10 -> 20 -> -1 -> "asdf" -> 13 -> "qwerty" -> 1024
```

then the list that you return must look like this:

```
20 -> "asdf" -> "qwerty"
```

If the old list contained only a single node, or was empty, return an empty list.

This function must be recursive, but is **not** required to obey the Annoying Requirements.

6.1 Hints

You may be tempted to recurse into `head.next` at every step. But this is not going to work - because then, sometimes your code will want to **keep** the head, and other times, your code will want to **discard** it. Since we don't allow helpers, how are you going to do this?

Remember: while your function has to be recursive, it doesn't have to always recurse to the **very next** node. Where should you recurse, really, if you want to make accordion work?

7 `pair_recursive(head1, head2)`

This function takes **two** lists, which may be of different lengths (either or both of them might be empty). It builds new `ListNode` objects, where the value of each one is a tuple, consisting of one value from the first list, and one value from the second list, in the same order as they existed in those input lists.

If the lists are different lengths, then the length of the returned list will be equal to the length of the shorter one.

This function must be recursive, but is **not** required to obey the Annoying Requirements.

7.1 EXAMPLE

Suppose that the input lists are as follows:

```
head1:  123  ->  456  ->  789
head2: "abc" -> "def" -> "ghi" -> "jkl"
```

then the list that you return must look like this:

```
(123,"abc") -> (456,"def") -> (789,"ghi")
```

7.2 Recursing on Two Things

So far, all of the recursive functions you've done have been over a single thing: either an integer, or a linked list. But there are some problems - like this one - where you recurse over two things in parallel.

While this seems weird at first, if you look at it carefully, you'll realize it's not a lot more complex than what you've already done; in this problem you will pair the first value from list 1 with the first value from list 2, and then the second value from list 1 with the second value from list 2, an so on. So really, the only trick here is that when you recurse, you must recurse into the `next` field of **both** lists, at the same time.

8 Turning in Your Solution

You must turn in your code using GradeScope.