Week 5 - More with Recursion

Updated Oct 8, 2:30pm

The Recursive Guarantee

Every recursive function you write will have two components:

- One or more **base cases**, the smallest problem sizes, which you can solve directly, without a recursive function call.
- The **recursive thinking** you can use to break down a problem into smaller parts with the same structure.

Note that the "smaller parts" really means closer to the base cases - for lists, this is certainly true by considering the *rest* of a list.

The recursive guarantee says that if:

1. you've correctly identified all the base cases, and are handling them correctly, and
2. you've done the correct recursive thinking and have translated that thinking into code, then

your function is correct, no call stack required.

This guarantee tell you how to approach solving problems recursively: identify your base cases and handle them, identify the recursive structure problem and write code to reflect it, and you're done! (Note that you might find during this process that you need to add more base cases if your recursive structure doesn't work all the time, and that's okay!)

The converse guarantee

The Recursive Guarantee has a converse: if your code is not working, it can *only* be because you've missed a base case, you're handling a base case incorrectly, or your recursive thinking/implementation is incorrect.

Trying to find missing/incorrect base cases is usually straight-forward, as you can analyse the code and problem normally. To identify flaws in your recursive thinking, however, it is critical that you take to heart the message from last lecture: **when you get to a recursive call in your code, assume it works according to its docstring.**

Your recursive thinking acts as a transformation from the result of the recursive call(s) to the final output you want. If something is going wrong, it's not because the recursive call(s) are incorrect - it's because **you are not using the output in the right way**. That is, your ingredients are not the problem; the problem is how you are combining them to bake your cake.

Recursive Linked List Mutation (Deletion)

As with the node-based linked lists, things get a little trickier when mutation is involved.

However, the basic idea - changing the links to achieve both insertion and deletion - remains the same.

```
def remove(self, index):
        """ (LinkedListRec, int) -> NoneType

        Remove item at position index from the list.
        Raise an IndexError if index is out of bounds.
        """
```

Applying our recursive thinking, we notice that there are really two possible base cases: when the list is empty, and when `index` is 0. Note that we've divided this up into two base cases because the function really should behave differently for the two cases: in the former, an `IndexError` should be raised, and in the latter, the first item in the list should be removed.

What about the recursive step? As with `__getitem__`, if we're asked to delete (say) the third item in a list, that's the same as deleting the second item in the *rest* of the list. This now gives us our overall structure:

```
def remove(self, index):
    if self.is_empty():
        raise IndexError
    elif index == 0:
        self.remove_first()
    else:
        self.rest.remove(index - 1)
```

Now the only remaining thing to do is implement `remove_first`, which we do below. Note that this is the *only* part that actually changes the list, and it does so by basically removing `self.first`.

```
def remove_first(self):
    if self.is_empty():
        raise IndexError
    else:
        self.first = self.rest.first
        self.rest = self.rest.rest
```

Insertion

We didn't cover this in lecture, but I'm leaving this up for you to use as a reference for Lab 5.

Before tackling the overall insertion method, let's handle the special case of insertion at the front. The basic idea is we want to update `self.rest` to be the old list of `self`, and update `self.first` to be the new element. You might be surprised to find that the following doesn't work:

```
def insert_first(self, item):
    self.rest = self
    self.first = item
```

The problem is that by setting `self.rest = self`, we actually lose the only reference we

had to the rest of the list beyond the first item! See a nice visualization of this here; at this point, a new `LinkedListRec` object has been made, and `insert_first` has just been called. Press "Forward" to see what happens next!

```
def insert_first(self, item):
    temp = LinkedListRec([])
    temp.first = self.first
    temp.rest = self.rest
    self.first = item
    self.rest = temp
```

Question: does this work when `self` is the empty list, i.e., if its `first` attribute is `EmptyValue`, and `rest` attribute is `None`? What should be created as a result?

In the lab, you'll finish up insertion by using `insert_first` to implement `insert`, much in the same way as `delete`. Note that I'm leaving it to you because this is where the recursive thinking actually comes in!

A Non-List Example: Binary Strings

A binary string is a string containing only 0's and 1's. Let's try to print out a list of all binary strings of length `n`.

```
def all_binary_strings(n):
    """ (int) -> list of str

    Return a list of all binary strings of length n.
    >>> all_binary_strings(0)
    ['']
    >>> all_binary_strings(2)
    ['00', '01', '10', '11']
    """
```

Before writing any code, we need to understand the recursive structure of the problem. Often students begin writing before they understand this; this leads to really complicated code, and goes hand in hand with then having to trace through this code to figure out what it's doing, because it isn't working.

When trying to identify the recursive structure, it usually helps to look at small cases - but not too small, because if the cases are too small then the pattern can be obscured. Let's list all of the binary strings of length 3:

000, 001, 010, 011, 100, 101, 110, 111

Remember that we're looking to break down the structure into a smaller instance with the same structure: in this case, we might notice that if we remove the first character of these binary strings, the result is a bunch of binary strings of length 2:

~~0~~**00**, ~~0~~**01**, ~~0~~**10**, ~~0~~**11**, ~~1~~**00**, ~~1~~**01**, ~~1~~**10**, ~~1~~**11**

What do you notice about this? It looks like *every binary string of length 2* is in the list, and *each one appears exactly twice*. That's interesting! Can you come up with an explanation?

What we've stumbled upon is a relationship between "all binary strings of length 3" and "all binary strings of length 2". And hey, that's exactly what we're looking for in "recursive

structure"!

We can summarize the recursive structure of the problem in an algorithmic way as follows. To compute all binary strings of length n,

1. Compute all binary strings of length n - 1
2. Take each of those strings, and add a 0 to the front.
3. Take each of those strings, and add a 1 to the front.
4. Return all of the strings from steps 2 and 3.

That's it for the recursive structure: what about the base case? Well, when `n = 0` there's only one binary string of length 0, the empty string. With this, we can fully implement the function.

(Before looking at the code below, try doing this yourself! It's a much more valuable experience, trust me!)

```
def all_binary_strings(n):
    if n == 0:
        return ['']
    else:
        short_strings = all_binary_strings(n - 1)

        strings = []
        for s in short_strings:
            strings.append('0' + s)
        for s in short_strings:
            strings.append('1' + s)

        return strings
```

Computer Science
UNIVERSITY OF TORONTO

David Liu (liudavid at cs dot toronto dot edu)
Come find me in BA4260

Site design by Deyu Wang (dyw999 at gmail dot com)