Recursion wrap-up

CSC148, Introduction to Computer Science

Diane Horton, Sadia Sharmin, & Jonathan Calver

Believing in recursion

- How can we just *assume* the recursive call works? What it if doesn't?
- Let's examine the justification for our confidence.

"Confidence Table" for function sum

depth	example	answer	correct?	

Reasoning about correctness

- Let P(n) =

  “*For any nested list obj of depth n,*
  nested_sum(obj) returns, and
  returns the sum of the numbers in <obj>.”

- We want to know that ∀n ≥ 0, P(n).

Tracing from the smallest case up

- For nested_sum, we traced the function and concluded that:
  - P(0) is true.
  - P(1) is true as long as P(0) is true.
  - P(2) is true as long as P(0) and P(1) are true.

    . . . and we could have continued on to show that . . .
  - P(3) is true as long as P(0), P(1), and P(2) are true.
  - And P(4), P(5), ...
- Crucially, we did not trace the "as long as" parts.
- (We had already convinced ourselves of them.)

Reasoning more formally

- If we show these two things:

  P(0) is true.

  $\forall k \geq 0$,

    P(k+1) is true as long as P(0), … P(k) are all true.

- … we can conclude that:

  $\forall n \geq 0$, P(n).

Base case(s) in the code

- There must be at least one base case.
- There may be more than one.
- Any call to a recursive method must ultimately reach a base case.
  - Otherwise, we have “infinite” recursion.
  - It can’t actually continue indefinitely, because each recursive call needs a stack frame, and we will stop due to running out of memory.
- In order to reach a base case:
  - The problem size must decrease on every recursive call.
  - Recursive calls must ultimately “connect” with a base case.

Examples of infinite recursion

Variations

- Some methods have one recursive call.
- Others have more than one.

- For some methods, the size of the problem is reduced by 1 on each recursive call.
- For others, it is reduced by half.
- Regardless, the problem size must be reduced.

Which option is right depends on the problem.

Example: recursion on a list

Options Include:

Problem size when we recurse	Base case
Depth reduces by 1	Depth 0
Length reduces by 1	Length 1
Length reduces by half	Length either 0 or 1

Structural recursion

- Sometimes our data had a recursive structure.

  A nested list is a list of nested lists
  or just a number.

- Our code's structure mirrored this.

  Return _____ if we have a list of nested lists
  or return _____ if we have just a number.

- We call this "structural recursion".

Recursion isn’t always structural

- Often we recurse over a recursive structure.
- Sometimes we *build* a recursive structure as we recurse.
- But sometimes we do neither.

Example

```
def buyable(n: int) -> bool:
    """Return whether one can buy exactly <n>
    McNuggets.

    McNuggets come in packs of 4, 6, or 25. It is
    considered possible to buy exactly 0
    McNuggets.

    Precondition: n >= 0
    """
```

Communicate via parameters & return

- Each time we call a recursive method:
  - Everything it needs should be sent through parameters
  - Everything it must report back should come through parameters
- Don’t attempt to work around this protocol by using local variables.
  - Each call has its own stack from with its own instance of the local variables.
  - So nothing can accumulate in them across calls.

Helper methods

- Sometimes, a method's interface doesn't "have enough" to support the recursion.
- A helper can have an additional parameter.
  Example: The helper for Tree's __str__ method
  - Adds a parameter for indent
- A helper can have an additional return value.
  Example: The helper for Tree's average method
  - Returns a tuple with total and number

Debugging recursive code

- Write test cases that match the code structure:
  - One test case for each base case
  - One test case for the / for each recursive case
- Run the base case tests first.
- If they work, run the recursive test case(s). Watch for:
  - Not “connecting” to the base cases properly.
  - Incorrect logic in the code that isn’t doing the recursion.
- Can identify and fix the bug without stepping into the recursive calls.

Recursion vs iteration

- Any problem we can solve with iteration can be solved with just recursion.
  - Some languages have *nothing but* recursion!
- Any problem we can solve with recursion can be solved with just iteration.
  - Recursion doesn't add "expressive power"
- But some problems have simple, elegant recursive solutions, and only complex non-recursive solutions.
  - Try writing tree traversal with no recursion, *not even through helpers.*

Tips for writing recursive functions

- Think lazy.
  - What smaller instance(s) of the same problem can I ask someone to solve for me?
  - When the problem is so small that even lazy you can do it, write the code directly.
- Mind your own business.
  - When you make a recursive call, don't concern yourself with how it solves the problem!
  - And don't concern yourself with what *your* caller is going to do with your result.
- Analyze the cases before writing any code.