CIS 1100

Recursion

Python

Spring 2025

University of Pennsylvania

Warm Up

```python
def cat(x):
    if x > 10:
        return x
    return dog(x * 2)
```

```python
def dog(y):
    if y > 10:
        return y
    return cat(y * 2)
```

(S7) What does `cat(2)` return? What about `cat(-1)`?

Don't Lose Sight of What You Know!

Today's topic (recursion) is tricky, but it's manageable if you keep in mind everything you know about calling functions.

1. Functions are called by their names with inputs passed in
2. Executing a function call creates a new scope in which only **input variables** and **variables defined within the function** are accessible
3. The only way to have some information "escape" the body of a function is to **return it.**
4. Other than producing a value, `return` also stops the current function execution and brings us back to the place at which the function was called.

Recursive Thinking

The journey of a thousand miles starts with one mile.

And then a journey of 999 miles.

Recursive Thinking

A function is recursive if it invokes itself to do part of its work.

Recursion is a problem-solving approach that can be used to generate simple solutions to certain kinds of problems that are difficult to solve by other means.

Recursion reduces a problem into one or more simpler versions of itself.

Recursion

An alternate to using loops for solving problems

The core of recursion is taking a big task and breaking it up into a series of related small tasks.

- Example: handing out papers for an exam
  - Iterative: have a TA walk down a row of students, giving each person an exam
  - Recursive: A student takes one exam, pass the rest down the aisle
- Example: Which row are you in?

Anatomy of a Recursive Function

Every recursive function needs at least one **base case** and at least one **recursive part**.

The **base case:**

- handles a simple input that can be solved without resorting to a recursive call. Can also be thought of as the case where we "end“ our recursion.

The **recursive part:**

- contains one or more recursive calls to the function.
- In every recursive call, the parameters must be in some sense "closer" to the base case than those of the original call

Practice

In mathematics, the Fibonacci sequence is a sequence in which each number is the sum of the two preceding ones. Numbers that are part of the Fibonacci sequence are known as Fibonacci numbers.

The sequence starts with 0 and 1:

```
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ...]
```

`fib(0)` is 0, `fib(1)` is 1.

We want to write a recursive function to calculate the Nth fibonacci number.

(L11) What are the base case(s) and recursive(s)? (e.g. when do we recurse, when do we not).

Practice

In mathematics, the Fibonacci sequence is a sequence in which each number is the sum of the two preceding ones. Numbers that are part of the Fibonacci sequence are known as Fibonacci numbers.

The sequence starts with 0 and 1:

```
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ...]
```

`fib(0)` is 0, `fib(1)` is 1.

(C12)

Write the function:

```python
def fib(N):
    # TODO
```

Range of a recursive function

An important thing to think about when desigining recursive functions is thinking about:

- Where we start with the problem
- the little bit of work that is done on each step
- the end of the problem

What were each of these things for `def fib(N)`?

What about `def print_stars(N)` (from the videos)?

Practice: (S8-S10)

```
def foo(N):
    if N <= 1:
        return 0
    return 1 + foo(N // 2) # Hint: // is integer division

def fizz(N):
    if N == 0:
        return 0
    return N % 10 + fizz(N // 10)
```

What do these print?

- (S8) `print(foo(16))`
- (S9) `print(fizz(1100))`
- (S19) `print(fizz(8675309))`

Practice:

Consider we want to write the function `remove_vowels(word)` that takes in a string and returns the same string without any vowels in it.

You can assume you have access to the set vowels:

```
vowels = {'A', 'a', 'E', 'e', 'I', 'i', 'O', 'o', 'U', 'u', 'Y', 'y'}
```

So `remove_vowels("Hello")` returns `"Hll"`

Before writing any code **(L13)**

- What is the base case? (Why is it the empty string? `""`)
- What is the recursive case?
- What is the work done on each step?

Practice: (C14)

Finish writing this function:

```
def remove_vowels(word):
    # takes in a string and returns the same string without any vowels in it.
    vowels = {'A', 'a', 'E', 'e', 'I', 'i', 'O', 'o', 'U', 'u', 'Y', 'y'}
    # TODO: What do you put here?
```

Practice:

We want to write a function `sum_numbers` that gets the sum of all integers in a list recursively.

Do this in (C12)

First think:

- What are the base case(s) ?
- What is the little bit of work done on each step?
- What do we tell the function to do recursively?

Hint: You may want to use list slicing

```python
def sum_numbers(nums_list):
    # TODO
    # return a sum of all the numbers in the list
    # your soln doesn't have to be perfect, remember that this is practice
```

Practice: (L11)

We want to write the funcion `ping_pong(N)` which prints "ping" and then "pong" in alternating order for a total of N prints.

`ping_pong(3)` prints: `ping` then `pong` then `ping`

`ping_pong(2)` prings: `ping` then `pong`

`ping_pong(1)` prings: `ping`

(L11) Does this code work? Why or why not:

```
def ping_pong(N):
    if N <= 0:
        return
    if N % 2 == 1:
        print("ping")
    else:
        print("pong")
    ping_pong(N - 1)
```

Helper functions

Sometimes to do recursion we need to remember a bit more information than is provdided to the overall problem.

In this case, what other information do we need for `ping_pong(N)` to get a working recursive solution?
Why can't we just recursively call `ping_pong(N-1)`?

Practice (C12)

We want to write the funcion `ping_pong(N)` which prints "ping" and then "pong" in alternating order for a total of N prints.

`ping_pong(3)` prints: `ping` then `pong` then `ping`

`ping_pong(2)` prings: `ping` then `pong`

`ping_pong(1)` prings: `ping`

(C12) finish writing the fixed version:

```python
def ping_pong_helper(N, extra):
    # TODO: do something here

def ping_pong(N):
    ping_pong_helper(N, ______)  # You probably want to pass in either 0 or a boolean here
```

Practice:

Consider we want to write the function `find_factors(N)` that returns a set containing all positive factors of the input integer `N`.

A number `x` is a *factor* of `N` if and only if `N % x == 0`

NOTE: This is a different problem than one you will see on the homework called `gcd`

Before writing any code **(L13)**

- What is the base case?
- What is the recursive case?
- What is the work done on each step?

Helper functions

Sometimes to do recursion we need to remember a bit more information than is provided to the overall problem.

In this case, what other information do we need for `find_factors(N)` to get a working recursive solution?

Why can't we just recursively call `find_factors(N-1)`?

Practice: (C14)

Finish writing this function:

```python
def find_factors_helper(current, N):
    # TODO: What do you put here?

def find_factors(N):
    return find_factors_helper(0, N)
```

Practice:

Consider we want to write the **recursive** function `binary_search(lst, target)` that returns the position of `target` in the `lst` or `-1` if it's not present.

Before writing any code **(L15)**

- What is the base case(s)?
- What is the recursive case?
- What is the work done on each step?
- What additional information needs to be passed in to do a Binary Search/what's the signature of the helper function?

Helper functions

Binary Search works by repeatedly shrinking the *feasible search area*.

- `lo` is the left-most/lowest index that might still hold the target
- `hi` is the right-most/highest index that might still hold the target
- want to keep searching until we find our target or run out of space to search: when `lo` moves to the right of `hi`

Practice: (C14)

Finish writing this function:

```
def binary_search(lst: list[int], target: int) -> int:
    return binary_search_helper(lst, target, __________)
def binary_search_helper(lst: list[int], target: int, lo: int, hi: int) -> int:
    # TODO!
```