Lecture 7: Tree Recursion

CS 61A - Summer 2024

Raymond Tan

Order of Recursive Calls

General Structure of Recursive Functions

- **Base case(s)**: The simplest instance of the problem that can be solved without much work
- **Recursive call**: Making a call to the same function with a smaller input, getting you closer to the base case(s)
- **Recombination**: Using the result of the recursive call to solve the original problem

Cascade

- We want a function that can “cascade” a number
  - Given a number such as **123**, the output would be:

```
123
12
1
12
123
```

Demo: Cascade

Two definitions of cascade

```
def cascade(n):
    if n < 10:
        print(n)
    else:
        print(n)
        cascade(n // 10)
        print(n)
```

```
def cascade(n):
    print(n)
    if n >= 10:
        cascade(n // 10)
        print(n)
```

- If two implementations are equally clear, then shorter is usually better
- In this case, the longer implementation is more clear
- When learning to write recursive functions, put the base cases first
- Both are recursive functions, even though only the first has typical structure

Inverse cascade

Want inverse_cascade(1234)
to return:

```
1
12
123
1234
123
12
1
```

```
def inverse_cascade(n):
    grow(n)
    print(n)
    shrink(n)

def f_then_g(f, g, n):
    if n:
        f(n)
        g(n)
```

Try it out!

```
grow = lambda n : f_then_g(___, ___, ___)
shrink = lambda n : f_then_g(___, ___, ___)
```

Inverse cascade

Want inverse_cascade(1234)
to return:

```
1
12
123
1234
123
12
1
```

```
def inverse_cascade(n):
    grow(n)
    print(n)
    shrink(n)

def f_then_g(f, g, n):
    if n:
        f(n)
        g(n)
```

Try it out!

```
grow = lambda n : f_then_g(grow, print, n // 10)
shrink = lambda n : f_then_g(print, shrink, n // 10)
```

Tree Recursion

What is Tree Recursion?

- A function is defined as **tree recursive** if:
  - It uses recursion (the function calls itself)
  - At the recursive step, **more than one recursive call** is made
- That's it!
- We call it tree recursion because the recursive calls form a tree-like structure when drawn out

Example: fib Revisited

- The fibonacci numbers are by definition, recursive
  - A **fibonacci number** is defined as the sum of the previous two **fibonacci numbers**

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987…

Recursive fib in Python

- **Base case**: What's the simplest input I can give to fib?
  - If n == 0, return 0
  - If n == 1, return 1
- **Recursive case**: What recursive calls should I make?
  - fib(n - 1), fib(n - 2)
- **Recombination**: How do I use the subproblems to solve my current problem?
  - fib(n) = fib(n - 1) + fib(n - 2)

```
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n - 1) + fib(n - 2)
```

Recursive fib in Python

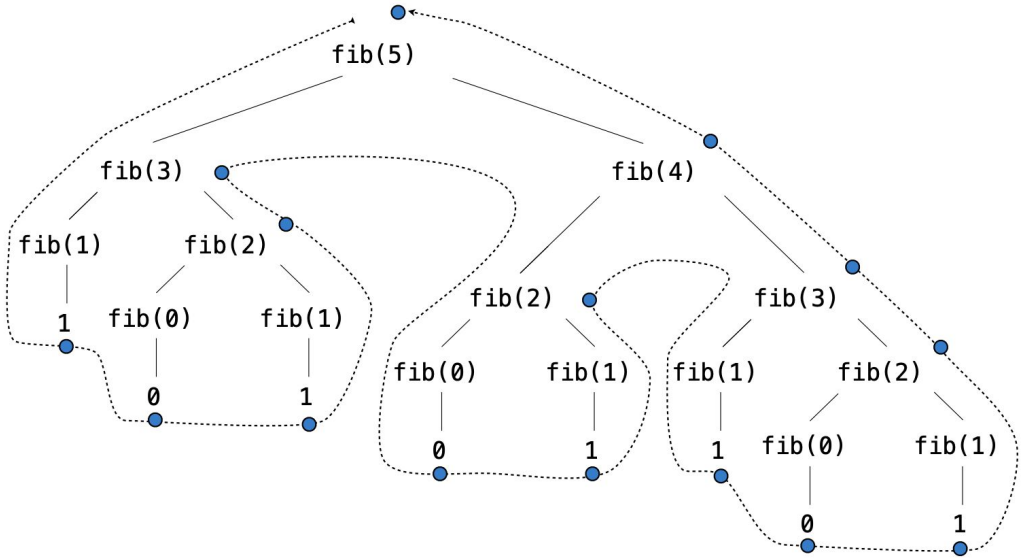
- **Was it necessary to have multiple base cases here?**
  - Yes, since we're subtracting our input by two in one of the recursive calls
- Can reduce this to one if statement:

```
def fib(n):
    if n <= 1:
        return n
    return fib(n - 1) + fib(n - 2)
```

```
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n - 1) + fib(n - 2)
```

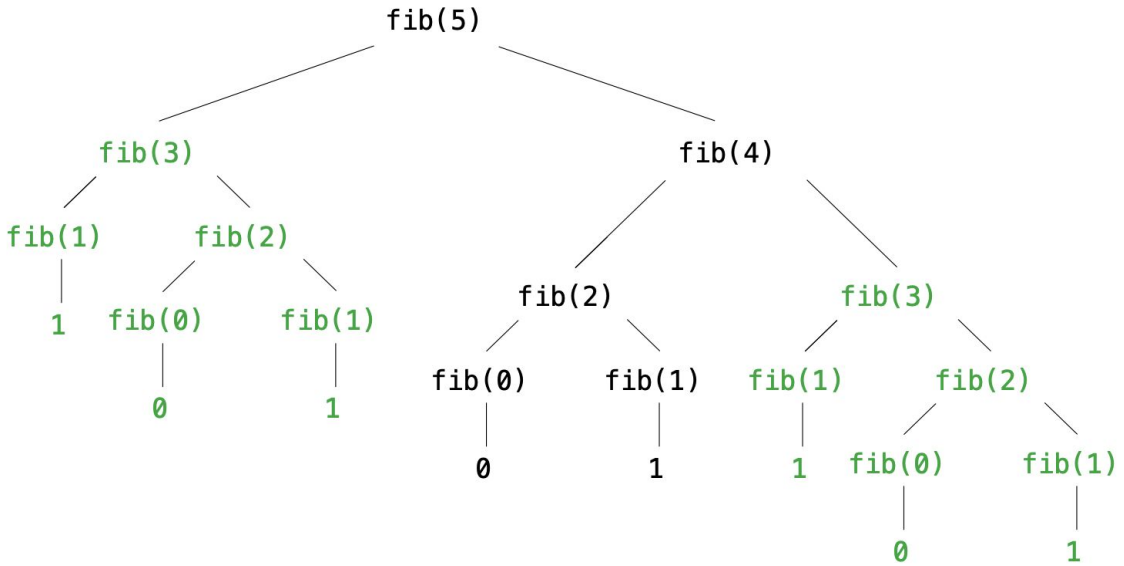
Tree Recursive Structure

The computational process of `fib` evolves into a “tree” structure



Repetition in Tree Recursive Computation

This process is highly repetitive; `fib` is called on the same argument multiple times



We'll see how we can speed this up in the **Efficiency** lecture!

Memoization

- The basic idea of memoization is that each time we execute a recursive computation, we record the result of that computation
- That way, if we ever see exactly the same parameters a second time, we can access the result directly, rather than having to excuse a new series of recursive calls
- We'll have an entire lecture focused on Efficiency in about 2 weeks

Demo: Memoization

Break

Example: Count partitions

Recognizing Tree Recursive Problems

- In the Fibonacci problem, the recursive structure was presented to us in the problem statement—we won't always be so lucky
- The hardest parts of tree recursion problems often are:
    - Figuring out that we need to use tree recursion
    - Figuring out how to represent our problem recursively

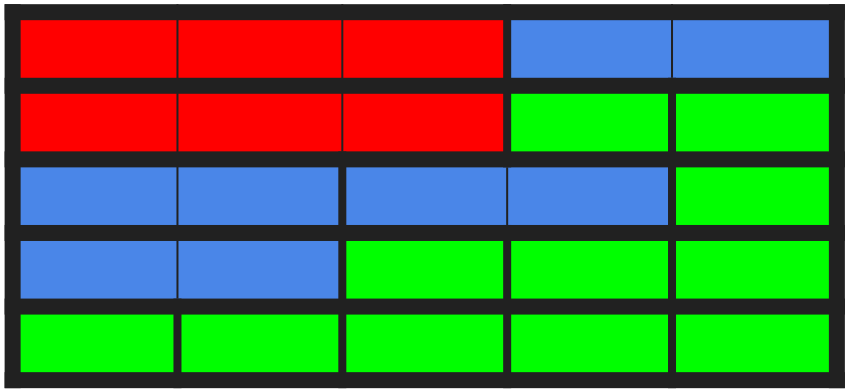
Count partitions

We're going to write a function, `count_partitions`, that takes in two parameters: n and k

`count_partitions` will return the number of different ways that you can "partition" n into a sum of smaller numbers, where the largest partition you're allowed to use is one of size k

For example, if n = 5 and k = 3, the valid partitions are:

```
5 = 3 + 2
5 = 3 + 1 + 1
5 = 2 + 2 + 1
5 = 2 + 1 + 1 + 1
5 = 1 + 1 + 1 + 1 + 1
```



So, `count_partitions(5,3)` should return 5

Solution

```
def count_partitions(n, k):
    if n == 0:
        return 1
    elif k == 0 or n < 0:
        return 0
    return count_partitions(n - k, k)
            + count_partitions(n, k - 1)
```

Base cases

```
if n == 0: return 1
```

I call this the "success case." If we take enough partitions out of n such that we have exactly zero left, we've found a valid partitioning of n.

```
elif k == 0 or n < 0: return 0
```

I call this the "failure case." There are two ways we can fail:

- We make k too small, such that there are no more valid partitions
- We make n too small, presumably by taking out a partition that's larger than n itself

Recursive cases

`count_partitions(n - k, k)`

If we want to count all the ways to do something, we'll just try every possible option and see what works out for us.

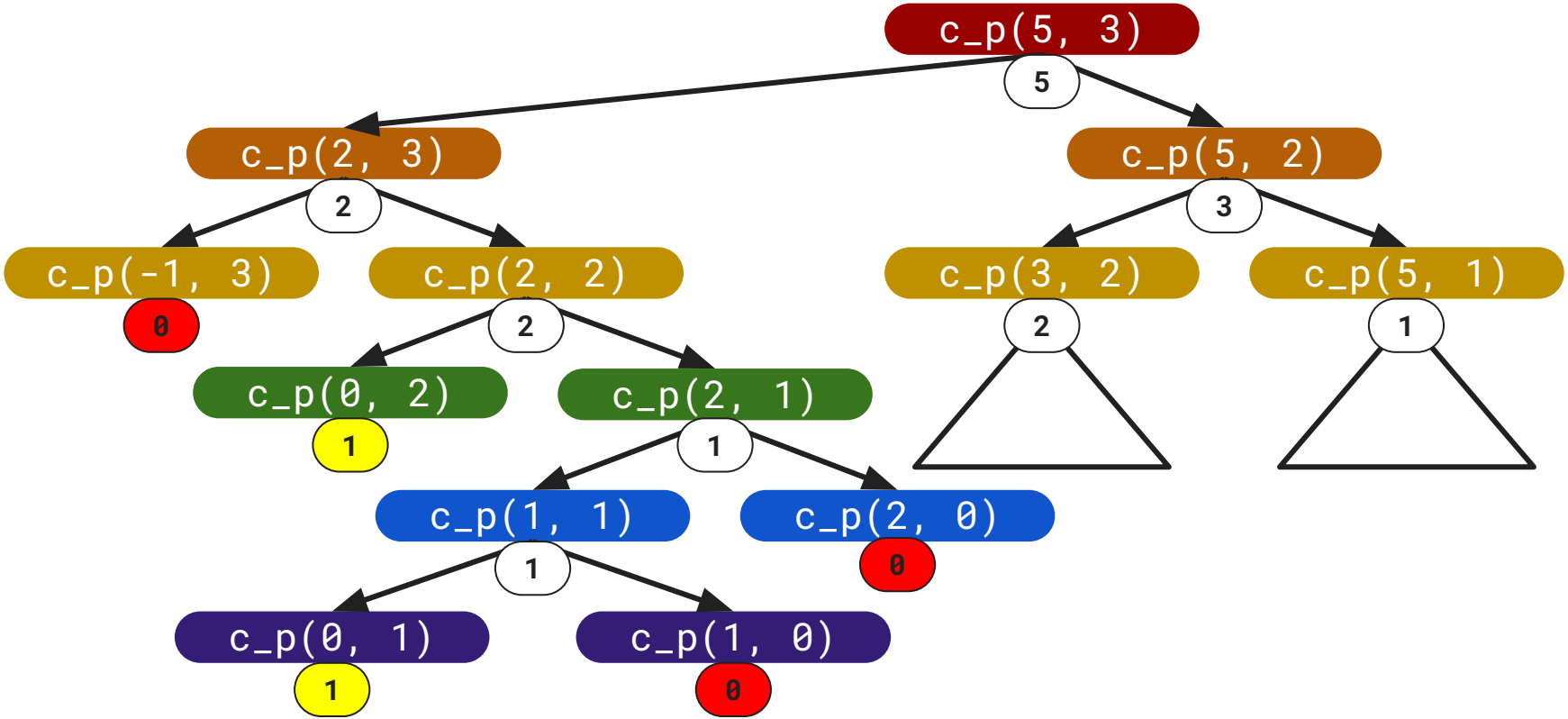
So, one way to try new options is to see what happens if we take out the largest possible partition. Then, we take the recursive leap of faith and let Python do the rest of the work.

`count_partitions(n, k - 1)`

Alternatively, we could decide that we no longer want to attempt to use that largest partition, and see what happens if we further constrain our problem by reducing the partition size. Also takes the recursive leap of faith.

Call diagram

*count_partitions is abbreviated as c_p



Observations about tree recursion

- Tree recursion is especially good for solving problems where we're presented with a decision at each step of the problem
  - We can model each option we're presented with as a recursive call—this allows us to ultimately try all possible options
- It depends on the problem, but sometimes working out your recursive calls is important in helping you figure out your base case
  - One of the trickiest parts can be figuring out how to model "moving forward" in a complex word problem in terms of parameters

Summary

- Execution doesn’t return to a frame until the recursive call returns
  - This is no different than non-recursive function calls in the body of another function
- Tree recursion is when we make multiple recursive calls in the body of a function
  - Examples: fib, count_partitions
- Memoization can make recursive solutions more efficient
- Tree recursion is especially good for solving problems where we're presented with a decision at each step of the problem