TREE RECURSION Meta

COMPUTER SCIENCE MENTORS 61A

September 19–September 23, 2022

Recommended Timeline

- Mini-lecture and Gibonacci (Q1 & Q2): 15 min
- Donut (Q3): 10 min
- Mario (Q4): 15 min (Would recommend going over this question)
- Discussions (Q5): 10 min
- Make Change (Q6): 10 min

The recommended timeline does not sum to 50 minutes because we do not expect any mentors to finish all questions within a section.

In most recursive problems we've seen so far, the solution function contains only one call to itself. However, some problems will require multiple recursive calls—we call this type of recursion "tree recursion" because the propagation of function frames reminds us of the branches of a tree. Despite the fancy name, these problems are still solved the same way as those requiring a single function call: we define a base case, use a recursive call to solve a smaller subproblem, and then solve the original, larger problem with the solution to our subproblem. The difference? Instead of just using the solution to one subproblem, we may need to use multiple subproblems' solutions to solve our original problem.

Tree recursion will often be needed when solving counting problems (how many ways are there of doing something?) and optimization problems (what is the maximum or minimum number of ways of doing something?), but remember that there are all sorts of problems that may need multiple recursive calls!

Teaching Tips

- Stress the power of tree recursion: it lets us find a single solution among many futures.
- Try dividing tree recursion questions into three parts: base cases, recursive calls, and combining recursive calls.
  1. What are the simplest possible arguments for the function?
     - There may be hints for base cases in doc tests. Run through simple examples!
  2. What options should be recursively explored?
     - Drawing tree diagrams can help a lot for this section.
  3. How should the answers of subproblems be combined?
     - Trust recursive calls to return the correct values (recursive leap of faith!) and combine them with mathematical or logical operators.

Vibha Tantry and Jordan Schwartz, with
Gabe Classon, Aditya Balasubramanian, Oliver Yu, Hanze Tan, Gene Pan, Carolyn Wang, Raman Varma, Noor Haq, Angela Zhang, Abigail Brooks, Ted Kim, Aiden Joshua Vehemente, Marie Chorpita, Ashley Chiu, Bill Hu, Omar Yu, Angel Alberto Aldaco, Andrew Park, Chase Clements, Grace Yi, Preshtha Garg, Laksith Venkatesh Prabu, Aditya Murali, Irene Yang, Andy Chen, and Hailey Park

- It is worth mentioning that perhaps one of the strongest skills one can develop in 61a is the ability to simplify doctests. Oftentimes running through a tree recursive problem's doctests that have larger inputs involves drawing a very complicated and lengthy call tree diagram, which is simply not feasible at times. Breaking down the problem in the aforementioned steps will yield the base case, but creating additional simpler doctests to understand how the function works at a smaller level will be optimal to finding the solution. I would like to note that in developing this skill, if a students creates a doctest that is wrong, it would be detrimental to their understanding of the problem, but if they understand how the function works, creating simpler doctests can show them how the function behaves with edge cases and different sized inputs.

We tend to throw around the term "recursive leap of faith" a lot, and I think that it confuses students. The "recursive leap of faith" is not synonymous with "the recursive call is correct". Rather it's a specific assumption we make *while writing* a recursive function that the recursive calls we make produce the correct output, even though the function in its current state would not. That is, the function we're writing works even though we're not done writing it. The fact that recursive calls return the correct value in the *completed* function is a mathematical fact that does not require any faith, so you should not conflate the two. Recursion is not magic; it is math.

Here's the way I think about it: the recursive leap of faith is essentially the scaffolding we need to help us build the recursive function. We pour the concrete for the base case and then layer our recursive logic on top of that until we have a sturdy structure, using the leap of faith's scaffolding to help us, as fallible human builders, to figure out the right way for the pieces to fit together. Once we're done, we can remove the scaffolding, but our tower still stands strong and sturdy.

1. The *Gibonacci sequence* is a recursively defined sequence of integers; we denote the nth Gibonacci number g_n. The first three terms of the sequence are $g_0 = 0, g_1 = 1, g_2 = 2$. For $n \geq 3$, g_n is defined as the sum of the previous three terms in the sequence.

   Complete the function `gib`, which takes in an integer `n` and returns the nth Gibonacci number, g_n. Also, identify the three parts of recursive function design as they are used in your solution.

```
def gib(n):
    """
    >>> gib(0)
    0
    >>> gib(1)
    1
    >>> gib(2)
    2
    >>> gib(3) # gib(2) + gib(1) + gib(0) = 3
    3
    >>> gib(4) # gib(3) + gib(2) + gib(1) = 6
    6
    """
    if ______________________________:

        return ______________________________

    return ______________________________
```

```
def gib(n):
    if n <= 2:
        return n
    return gib(n - 1) + gib(n - 2) + gib(n - 3)
```

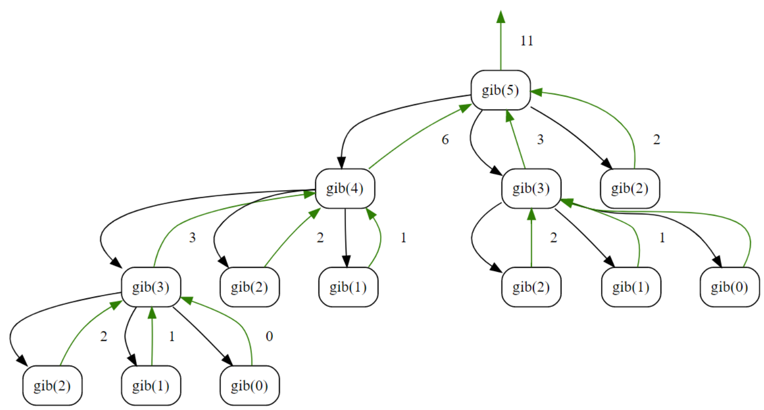
- Base case: `if n <= 2: return n`
- Recursive calls: `gib(n - 1),gib(n - 2),gib(n - 3)`
- Solving the larger problem: adding the results of the three recursive calls.

- This problem is very similar to Fibonacci, which students should be familiar with from lecture.
- Ensure that students understand how the minor alteration to the sequence slightly modifies the recursive calls and the base case

2. Including the original call, how many calls are made to `gib` when you evaluate `gib(5)`?

13.

There are a few ways you could determine this. First, you could draw out a call graph that shows all of the calls that are made. `https://tinyurl.com/gibsol`



You could also build things from the ground up using recursive intuition.

- `gib(0)` takes one call because it's a base case.
- `gib(1)` takes one call because it's a base case.
- `gib(2)` takes one call because it's a base case.
- `gib(3)` takes four calls: one for the original call, one for `gib(0)`, one for `gib(1)`, and one for `gib(2)`.
- `gib(4)` takes seven calls: one for the original call, four for `gib(3)`, one for `gib(2)`, and one for `gib(1)`.
- `gib(5)` takes 13 calls: one for the original call, seven for `gib(4)`, four for `gib(3)`, and one for `gib(2)`.

This second method is much faster, especially for higher values of `n`.

In general, I believe that drawing out call graphs can be confusing for students, especially for more difficult recursive problems. The whole idea behind recursion, after all, is that you don't have to worry about the dense tree of calls that result from a single call to a recursive function—you can just trust that the recursive calls you do make return the correct result and don't have to worry about how they specifically got that result. However, I believe that it is useful to do this sort of thing once because it demonstrates how recursion actually works in practice and why the recursion is valid and useful.

When teaching this problem, I recommend first going over the call tree. I would emphasize a few ideas. First, our base cases are at the bottom of the call graph, and every call eventually terminates at one of the base cases. They literally serve as the foundation of the tree. They are the "truth" upon which the function's validity for higher inputs rests. I would also point out that every call that's not a base case has three calls underneath it because each of these calls makes three recursive calls. Hopefully this makes the idea of recursion more concrete for the students and qualms perennial concerns that the logic behind recursion is circular.

I would then going over the "smarter" way of doing the problem. I would point out how doing the problem in this way allows us to avoid the repetitive, confusing, and error prone work of considering every call in the function. I would note that if we tried to do the same thing with `gib(1000)`, the graph would be untenable. The lesson I want them to walk away with is: abstracting away the huge number of calls is a valuable tool when working with recursion, and that they should try to avoid thinking about these calls if at all possible, as it only gets more confusing for more complicated functions.

3. Gabe's Donut shop has an unlimited supply of f different flavors of donuts. Adit wants to buy a box containing d donuts. Complete the skeleton for the function `donut`, which determines the number of possible ways there are for Adit to select his d donuts from the f flavors. You may assume that `d` and `f` are non-negative integers.

   Hint: Does order matter?

```
def donut(d, f):
    """
    >>> donut(12, 1)
    1
    >>> donut(12, 2)
    13
    >>> donut(12, 12)
    1352078
    >>> donut(0, 0)
    1
    """
    if ____________________:

        return ____________________

    if ____________________:

        return ____________________

    return ________________________________________
```

```
def donut(d, f):
    if d == 0:
        return 1
    if f == 0:
        return 0
    return donut(d - 1, f) + donut(d, f - 1)
```

   Suppose the flavors are numbered 1 through f. Because order does not matter, we're going to say that Adit fills up his donut box starting with flavor number f and working down to flavor number 1. Under this scheme, Adit has a choice to make: does he want a donut of flavor number f, or does he not want one? If he does initially take a donut of flavor number f, then the number of ways to select the remaining $d - 1$ slots of the box is `donut(d - 1, f)`. If he does not want a donut of flavor number f, then he has d slots left in the box and $f - 1$ flavors to fill them with, making the number of ways to make the selection `donut(d, f - 1)`. Since he must make the choice one way or the other, the total number of ways to fill up the box is `donut(d - 1, f)+ donut(d, f - 1)`.

If your students are confused on why the doctest `donut(0, 0)` returns 1, you can think about it this way:

if you want 0 donuts and they have no flavors, then there is one action you can take, do nothing and don't purchase donuts

A big concept within tree recursion counting problems is this idea that we can use some sort of "choice" to break our problem down into smaller subproblems. I would emphasize this heavily to my students. Here, the choice is whether or not Adit takes a certain flavor.

In this case, we have deliberately neglected to provide several base cases in the doc tests. While a good exam problem would include these doctests, they are often not provided, and students should be able to deduce the bases cases by either 1) mathematical maturity or 2) deduction from test inputs. In the latter case, they could for example note that `donut(1,1)== donut(0, 1)+ donut(1, 0)` and use this to help figure out what `donut` should return at these base cases.

4. Mario needs to get from one end of a level to the other, but there are deadly Piranha plants in his way! Mario only moves forward and can either *step* (move forward one space) or *jump* (move forward two spaces) from each position. A level is represented as a series of ones and zeros, with zeros denoting the location of Piranha plants. Mario can step on ones but not on zeros. How many different ways can Mario traverse a level without stepping or jumping into a Piranha plant? Assume that every level begins with a 1 (where Mario starts) and ends with a 1 (where Mario must end up).

   Hint: Does it matter whether Mario goes from left to right or right to left? Which one is easier to check?

```
def mario_number(level):
    """
    >>> mario_number(10101)
    1
    >>> mario_number(11101)
    2
    >>> mario_number(100101)
    0
    """
    if _______________________:

        ______________________

    elif _____________________:

        ______________________

    else:

        ____________________________________________________
```

```python
def mario_number(level):
    if level == 1:
        return 1
    elif level % 10 == 0:
        return 0
    else:
        return mario_number(level // 10) + mario_number((level // 10) //
            10)
```

You can think about this tree recursion problem as testing out all of the possible ways Mario can traverse the level, and adding 1 every time you find a possible traversal.

Here it doesn't matter whether Mario goes left to right or right to left; either way we'll end up with the same number of ways to traverse the `level`. In that case, we can simply choose for Mario to start from the right, and then we can process the level like we process other numbers in digit-parsing related questions by using floor division (`//`) and modulo (`%`)

At each point in time, Mario can either step or jump. We use a single floor division (`//`) of `level` by 10 to represent taking one step (if we took a step, then the entire `level` would be left except for the last number), while two floor divisions by 10 (or equivalently one floor division by 100) corresponds to a jump at this point in the `level` (if we took a jump, then the entire `level` would be left except for the last two numbers).

To think of the base cases, you can consider the two ways that Mario ends his journey. The first, corresponding to `level == 1`, means that Mario has successfully reached the end of the level. You can `return 1` here, because this means you've found one additional path to the end. The second, corresponding to `level % 10 == 0`, means that Mario has landed on a Piranha plant. This returns 0 because it's a failed traversal of the `level`, so you don't want to add anything to your result.

In tree recursion, you need to find a way to combine separate recursive calls. In this case, because `mario_number` returns an integer and the base cases are integers and you're trying to count the total number of ways of traversal, it makes sense to add your recursive calls.

Teaching Tips

- Some leading questions:
  - What are our base cases? (When do we know we've reached the end of the level? When do we know that we've failed?)
  - Is there any difference between going left to right or right to left in terms of the number of ways to traverse the level?
  - What are our two options at each step?
  - What do those look like in a recursive call?
  - How should we combine our recursive calls? (and, or, addition, etc.)
- Try leaning into the narrative of the question! It's fun and can help rephrase recursive calls "in plain english" I also like drawing the problem out along with the doctests to visualize the different steps Mario can take :D !
- It's very useful to draw a tree diagram! Each function call has one branch for stepping once and another branch for stepping twice. Each branch then has 2 branches of their own (until a base case is reached).

- Teach students that recursive calls can be treated as numbers using the recursive leap of faith, so combining the two recursive call branches with addition is really just adding two numbers.

5. In an alternate universe, 61A software is not that good (inconceivable!). Tyler is in charge of assigning students to discussion sections, but sections.cs61a.org only knows how to list sections with either `m` or `n` number of students (the two most popular sizes). Given a `total` number of students, can Tyler create sections and not have any leftover students? Return `True` if he can and `False` otherwise.

```
def has_sum(total, n, m):
    """
    >>> has_sum(1, 3, 5)
    False
    >>> has_sum(5, 3, 5) # 0 * 3 + 1 * 5 = 5
    True
    >>> has_sum(11, 3, 5) # 2 * 3 + 1 * 5 = 11
    True
    >>> has_sum(61, 11, 15) # can't express 61 as a * 11 + b * 15
    False
    """
    if ______________________________________________:

        return True

    elif ____________________________________________________:

        return False

    return ____________________________________________
```

```
def has_sum(total, n, m):
    if total == 0:
        return True
    elif total < 0: # you could also put total < min(m, n)
        return False
    return has_sum(total - n, n, m) or has_sum(total - m, n, m)
```

An alternate solution you could write that may be slightly faster in certain cases:

```
def has_sum(total, n, m):
    if total == 0 or total % n == 0 or total % m == 0:
        return True
    elif total < 0: # you could also put total < min(m, n)
        return False
    return has_sum(total - n, n, m) or has_sum(total - m, n, m)
```

(Solution continues on the next page)

When thinking about the recursive calls, we need to think about how each step of the problem works. Tree recursion allows us to explore the two options we have: either create a new `m`-person discussion at this step or create a new `n`-person discussion at this step and can combine the results after exploring both options. Inside the recursive call for `has_sum(total - n, n, m)`, which represents accommodating `n` students, we again consider adding either `n` or `m` students to the next section.

Once we have these recursive calls we need to think about how to put them together. We know the return should be a boolean so we want to use either **`and`** or **`or`** to combine the values for a final result. Given that we only need one of the calls to work, we can use **`or`** to reach our final answer.

In the base cases we also need to make sure we return the correct data type. Given that the final return should be a boolean we want to return booleans in the base cases.

Another alternate base case would be: `total == 0 or total % n == 0 or total % m == 0`. This solution would also work! You would just be stopping the recursion early, since the total can be a multiple of `n` or `m` in order to trigger the base case - it doesn't have to be 0 anymore. Just be sure to still include the `total == 0` check, just in case someone inputs 0 as the total into the function.

Teaching Tips

- Some leading questions:
  - What are the base cases (when to return True/False)?
  - How can we reduce this problem into smaller subproblems (recursive step)?
  - What does the value of a recursive call tell us?
  - How can we put recursive calls together to get a final answer?
- Ask students about the simplest possible cases to identify base cases; make sure they realize `(total == n)` or `(total == m)` is incorrect because `(total == 0)` is a simpler True case.
- Point out the fact that tree recursion problems usually have you consider multiple "options" or "possibilities", and they should all be explored when you are writing your recursive cases.
- Out of all possible combinations of `n` and `m`, we only need 1 way for `n` and `m` to sum to the total for the function to return True, which implies **`or`** is an appropriate way to aggregate our recursive calls.

6. Implement the function `make_change`, which takes in a non-negative integer amount in cents `n` and returns the minimum number of coins needed to make change for `n` using 1-cent, 3-cent, and 4-cent coins.

```
def make_change(n):
    """
    >>> make_change(5) # 5 = 4 + 1 (not 3 + 1 + 1)
    2
    >>> make_change(6) # 6 = 3 + 3 (not 4 + 1 + 1)
    2
    """

    if _____________________:
        return 0

    elif ___________________:

        _____________________________________

    elif ___________________:

        _____________________________________
    else:

        _____________________________________
```

```
def make_change(n):
    if n == 0:
        return 0
    elif n < 3:
        return 1 + make_change(n - 1)
    elif n < 4:
        return 1 + min(make_change(n - 1), make_change(n - 3))
    else:
        return 1 + min(make_change(n - 1), make_change(n - 3),
           make_change(n - 4))
```

Teaching Tips

- Be careful of the wording here - we're counting the **minimum number of coins** used to make change, not the number of ways to make change.
- Go over the doctests to show that choosing the largest coin at the beginning isn't always the most efficient; we have to explore all possibilities.
- Some questions you can ask your students:
  - What are the different options available to us for making change at each step?
  - What are some cases where not all coins are available to us?
  - In the case that there are multiple options available, which one do we pick?
    - Is there a Python function that we can use to choose the right combination?
- Note that this skeleton is directed towards leading students to the answer (by building up from only using one coin, to using two, to using three). It is not the most efficient one (for example, you can just return n if n $<$ 3).
- If your students come up with a more compact solution, use that as an opportunity to solicit ideas from your entire section about how to reduce the complexity! You don't always have to present the exact solution.