Tree Recursion Solutions

Computer Science Mentors 61A

September 19–September 23, 2022

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!

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

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 ______________________________
```

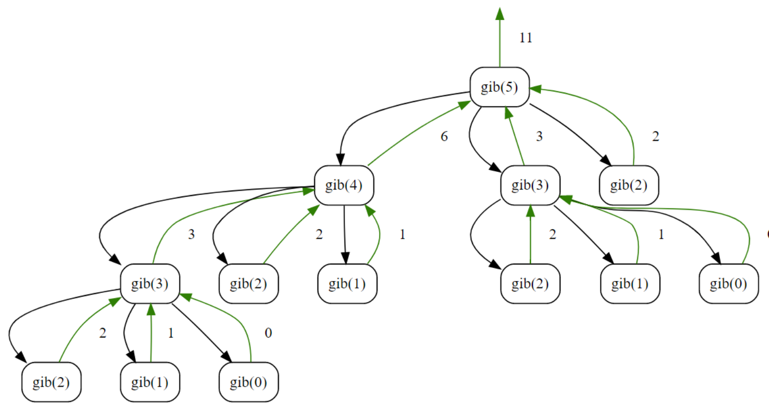
```
def gib(n):
    if n <= 2:
        return n
    return gib(n - 1) + gib(n - 2) + gib(n - 3)
```

- 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.

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`.

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)`.

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:

        ____________________________________________________
```

```
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.

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.

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))
```