Recursion, Tree Recursion Meta

Computer Science Mentors 61A

September 23 – September 27, 2024

Example Timeline

- Tree Recursion Mini Lecture [8 Mins]

    1. Consider creating a simple example to demonstrate recursion. Example: A recursive accumulation function (ie: next # = $\sum$ previous #s)

- Problems: Recursion [20 Mins]
    - Q1 - FizzBuzz
    - Q2 - Sum Prime digits
- Problems: Tree Recursion [20 Mins]
    - Q4 - Copy Machine
    - Q5 - Mario Number
    - Q6 - [Conceptual & Optional] Fast Modular Exponentiation

Created by Jim Fang, Bryan Adolfo Ventura Benitez. Worksheet feedback: links.jimfang.dev/feedback

1 Recursion

1. Implement a recursive version of `fizzbuzz`.

```
def fizzbuzz(n):
    """Prints the numbers from 1 to n. If the number is divisible by 3, it
    instead prints 'fizz'. If the number is divisible by 5, it instead
       prints
    'buzz'. If the number is divisible by both, it prints 'fizzbuzz'. You
       must do this recursively!

    >>> fizzbuzz(15)
    1
    2
    fizz
    4
    buzz
    fizz
    7
    8
    fizz
    buzz
    11
    fizz
    13
    14
    fizzbuzz
    """

    if n == 1:
        print(n)
    else:
        fizzbuzz(n - 1)
        if n % 3 == 0 and n % 5 == 0:
            print('fizzbuzz')
        elif n % 3 == 0:
            print('fizz')
        elif n % 5 == 0:
            print('buzz')
        else:
            print(n)
```

It may be beneficial to reiterate the recursive leap of faith! Hypothetically, if fizzbuzz works for n, what would fizzbuzz(n - 1) output?

This is an intro example of recursion - Feel free to skip if your students are clear on the concept of recursion.
⇒ Suggested Time: 3 min; Difficulty (Official): Medium; Difficulty (Adjusted): Easy/Intro

2. Complete the definition for `sum_prime_digits`, which returns the sum of all the prime digits of `n`. Recall that 1 is not prime. Assume you have access to a function `is_prime`; `is_prime(n)` returns `True` if `n` is prime, and `False` otherwise.

```
def sum_prime_digits(n):
    """
    >>> sum_prime_digits(12345)
    10 # 2 + 3 + 5
    >>> sum_prime_digits(4681029)
    2 # 2 is the only prime number
    """
    if ____________________________________________:

        return ____________________________________

    if ________________________________________:

        return ________________________________

    return ____________________________________
```

```
    if n == 0:
        return 0
    if is_prime(n % 10):
        return n % 10 + sum_prime_digits(n // 10)
    return sum_prime_digits(n // 10)
```

- Again, this problem is likely skippable if you have advanced students.
- Make sure your students understand why you are calling `sum_prime_digits` twice.

Suggested Time: 5 min; Difficulty (Official): Medium; Difficulty (Adjusted): Easy → Medium Mezzanine

2 Tree Recursion

Tree Recursion vs Recursion

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 colloquially call this type of recursion "tree recursion," since the propagation of function frames reminds us of the branches of a tree. "Tree recursive" or not, these problems are still solved the same way as those requiring a single function call: a base case, the recursive leap of faith on a subproblem, and solving the original problem with the solution to our subproblems. The difference? We simply may need to use multiple subproblems 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 there are all sorts of problems that may need multiple recursive calls! Always come back to the recursive leap of faith.

Let's work through an example of tree recursion in action! The **counting partitions** problem is a common utilization of tree recursion. Say we want to define a function that will count all the ways we can fill up a space with a certain amount of blocks. For the purposes of working out this problem, we'll usea space of size 4. If we use brute force to solve this problem in terms of adding blocks of sizes 1, 2, 3, and 4, here's what we come up with:

- 4 = 4 // filling up the space with a block that perfectly fits it
- 4 = 3 + 1 // filling up the space with a block of size 3 and a block of size 1
- 4 = 2 + 2
- 4 = 2 + 1 + 1
- 4 = 1 + 1 + 1 + 1

This doesn't seem too bad, but with bigger spaces, this is exponentially harder to brute force! With this in mind, however, we can recognize a pattern with the number of possibilities present with different "partition" sizes.

Recall that within recursive questions, arguments are used as ways to keep track of certain possibilities. We want to specify our arguments within this problem to be what's changing on iteration. Therefore, within our recursive arguments, we want to keep track of the maximum block size on each iteration, and how much more space we have left after adding that partition.

Now we know what arguments we want, we can construct the body of our code!

```
def count_partitions(space_left, block_size):
```

We know that a recursive function will have base cases and recursive cases. While everybody's way of working through these is different, for the sake of explanation, we'll start with our base cases. When working with recursion questions, we generally want to start with the biggest problem and work our way down to the most atomic ways of solving our problem, similar to the way in which our doctests are structured.

We will consider a way of counting partitions valid if the partitions we add perfectly fit the space provided. When a space is perfectly fit, the `space_left` will be zero! Therefore, we will count a case when it's `space_left` is equal to 0.

A doctest demonstrating this would be the most atomic iteration of `count_partitions`, `count_partitions(0, 0)`. There is only one way to count to zero with the maximum number to add being zero!

Now we consider invalid ways of counting partitions – a way of counting would be invalid if we *overfill* or *underfill* the space provided. Therefore, if our `block_size` is greater than the space left, we would overfill the space! Similarly if the `block_size` we're adding is 0, that means we've run out of partitions to add, resulting in an *underfilled* space.

With this in mind, here is our current working of `count_partitions` with the base cases in hand.

```
def count_partitions(space_left, block_size):
    if (space_left == 0):
        return 1 // valid! we've fit the space perfectly
    elif (space_left < block_size):
        return 0 // not valid! overfilled
    elif (block_size == 0):
        return 0 // not valid! space is underfilled
    else:
        // recursive cases :D
```

As with any recursive case, we want to consider how we can break our problem into possibilities that allow us to get closer to our base cases.

On each frame of `count_partitions`, we want to have both the `space_left` and `block_size` go closer to zero such that they can reach the base case.

Consider the first case for the 4 example we used earlier. We want to see how many different potential ways we can fill up a space of 4 blocks with maximum block size of 4. Upon our first call to this function we would want to include the way of filling up the space with the partition of size 4 in our recursive call. With this in mind, to include this way of counting, we would have to subtract the `block_size` of size 4 from the `space_left` of size 4 such that we can get to our base case! This would result in a recursive call of `count_partitions(space_left - block_size, block_size)` (We keep m the same as we would want to see if we can add more blocks of that same size). However, having just that case would end our counting prematurely. Thus we need to include a separate branch where we *don't* use that block and instead use smaller block sizes.

Similar to the doctest, we want to see every possibility using blocks of every possible size. The next block size down from 4 would be 3, thus resulting in a recursive call of `count_partitions(space_left, block_size - 1)`. From here, there are no other recursive cases we can call upon, thus completing our code. We add the recursive cases together as we want to sum all possibilities of counting such partitions, resulting in the following:

```
def count_partitions(space_left, block_size):
    if (space_left == 0):
        return 1
    elif (space_left < block_size):
        return 0
    elif (block_size == 0):
        return 0
    else:
        return count_partitions(space_left - block_size, block_size) +
            count_partitions(space_left, block_size - 1)
```

Teaching Tips

- Stress the power of tree recursion: it lets us find a single solution among 14,000,605 futures.
- Before going into the counting partitions example within the text overview, teach tree recursion with Fibonacci and *then* go into counting partitions. Counting partitions is pretty abstract for most people the first time around!
- If you choose to explain the counting partitions example
- 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.

1. James wants to print this week's discussion handouts for all the students in CS 61A. However, both printers are broken! The first printer only prints multiples of `n` pages, and the second printer only prints multiples of `m` pages. Help James figure out whether or not it's possible to print exactly `total` number of handouts!

```
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
    """
    if ________________________________________________:

        return ______________________________

    elif ______________________________________________:

        return ______________________________

    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 while printing: either print `m` papers at this step or print `n` papers 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 printing `n` papers, we again consider printing either `n` or `m` papers.

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.

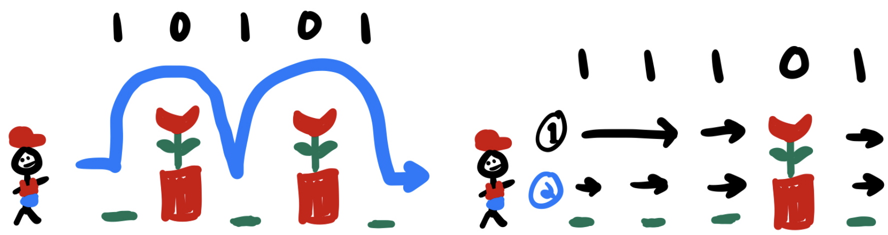
Another alternate base case would be: `total == 0 or total % n == 0 or total % m == 0`. This solution would also work! You would just be stopping the recursion early, since the total can be a multiple of `n` or `m` in order to trigger the base case - it doesn't have to be 0 anymore. Just be sure to still 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 recrusive 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.
- This is a good mini-lecture problem to use as a demo since this is probably the closest thing the students have seen to what they demo-ed in lecture.

⇒ Suggested Time: 6 min; Difficulty (Official): Medium; Difficulty (Adjusted): Easy

2. 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.

⇒ Suggested Time: 10 min; Difficulty (Official): Medium; Difficulty (Adjusted): Easy → Medium (Mezzanine)

The classic mario-number problem from CSM. A worksheet wouldn't be complete without it!

3. **Fast Modular Exponentiation:** In many computing applications, we need to quickly compute $n^x \mod z$ where $n > 0$, and x and z are arbitary whole numbers. Computing $n^x \mod z$ for large numbers can get extremely slow if we repeatedly multiply n for x times. We can implement the following recursive algorithm to help us speed up the exponentiation operation.

$$x^n \mod z = \begin{cases} x * (x^2)^{(n-1)/2} \,\%\, z & \text{if } n \text{ is odd} \\ (x^2)^{(n/2)} \,\%\, z & \text{if } n \text{ is even} \end{cases}$$

This is an example of a "divide & conquer" algorithm and follows the same train of thought as tree-recursion problems (you are dividing some complex problem into smaller parts and performing both options).

```
def modular_exponentiation(base, exponent, modulus):
    """
    >>> modular_exponentiation(2, 2, 2)
    0
    >>> modular_exponentiation(4, 2, 3)
    1
    """
    if _____________________:

        return ___________________________________

    if _____________________:

        half_power = ___________________________________
        # Hint: Which math formula above has exponent *just* divided by
            half?

        return ___________________________________ % modulus

    else:

        half_power = ___________________________________

        return _________________________________________ % modulus
```

Note: The algorithm you just implemented is a key part of modern day cryptography techniques such as RSA and Diffie-Hellman key exchange. In some cases, the exact operations you just implemented is used in modern day, state of the art, programs (if you are curious, Google "Right-to-left binary method"). You will learn more about RSA in CS70. If you want to learn more about computer security, consider taking CS161 after CS61C.

```python
def modular_exponentiation(base, exponent, modulus):
# Base case: exponent is 0
if exponent == 0:
    return 1

# Recursive case
if exponent % 2 == 0: # If exponent is even
    half_power = modular_exponentiation(base, exponent // 2,modulus)
    return (half_power * half_power) % modulus
else: # If exponent is odd
    half_power = modular_exponentiation(base, (exponent - 1) // 2,
        modulus)
    return (base * half_power * half_power) % modulus
```

- This problem doesn't exactly follow the scheme of tree recursion discussed in lecture, but it does follow the general philosophy behind tree recursion. You probably want to reiterate this.

  I think it is important for students to relate ”tree recursion” which is a SICP concept to something in the real world (whether that'd be a type of ~~leetcode~~ algorithms problem or a real world use). Here real world use made more sense as a lot of algorithms problems with tree recursion involved binary trees (and trees in general), which hasn't been introduced in lecture yet.

- This problem is not meant to be tricky or as much of a ”drill” problem as the other ones. This is more of a ”here is what tree recursion can do” type – so feel free to walk through this problem if you want.

Suggested Time: 10 min; Difficulty (Official): Medium; Difficulty (Adjusted): Medium