RECURSION Meta

COMPUTER SCIENCE MENTORS 61A

September 12–September 16, 2022

Recommended Timeline

- Recursion Mini Lecture and Wrong Factorial: 15 minutes
- Selective Sum: 10 minutes
- is_sorted and/or collapse: 10–20 minutes
- Mandelbrot: 20+ minutes, only do if your students are very comfortable with recursion

As a reminder, there is no expectation that you get through all problems in a section. Since is_sorted and collapse are so similar, it is probably not necessary to get through both.

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

There are three steps to writing a recursive function:

1. Create base case(s)
2. Reduce your problem to a smaller subproblem and call your function recursively to solve the smaller subproblem
3. Figure out how to use the smaller subproblem's solution in the larger problem's solution

Real World Analogy for Recursion

Imagine that you're in line for boba, but the line is really long, so you want to know what position you're in. You decide to ask the person in front of you how many people are in front of them. That way, you can take their response and add 1 to it. Now, the person in front of you is faced with the same problem that you were trying to solve, with one less person in front of them than you. They decide to take the same approach that you did by asking the person in front of them. This continues until the very first person in line is asked. At this point, the person at the front knows that there are 0 people in front of them, so they can tell the person behind them that there are 0 people in front. Now, the second person can figure out that there is 1 person in front of them, and can relay that back to the person behind them, and so on, until the answer reaches you.

Looking at this example, we see that we have broken down the problem of "how many people are there in front of me?" to 1 + "how many people are there in front of the person in front of me"? This problem will terminate with the person at the front of the line (with 0 people in front of them). Putting this into more formal terms, we are breaking down the problem into a **recurrence relationship**, and the termination case is called the **base case**.

Teaching Tips

1. Base Case - What is the simplest case? Or in what case do you want your recursion to stop?
2. Break the problem down into smaller problems
    - What do you need to do to reach your base case?
    - For example: in factorial (usually seen in lecture), we have to subtract by one each time we do a recursive call
3. Solve the smaller problem recursively
    - How would you use the solution to the smaller problem to write a solution to the original problem?
    - "Recursive Leap of Faith"—When writing the recursive statement, assume the function works as intended for the smaller problems.

- If you don't know what the recursive call needs to be, you can take an educated guess and see what happens.

1. What is wrong with the following function? How can we fix it?

```
def factorial(n):
    return n * factorial(n)
```

There is no base case and the recursive call is made on the same `n`.

```
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n - 1)
```

Great time to remind your students that the return type of the base case is the the same type as the function (i.e. if the function returns an int, the base case will have to return an int)

2. Write a function `selective_sum`, which takes in an integer `n` and a predicate function `cond`. `selective_sum` returns the sum of all positive integers up to `n` for which `cond(n)` is true.

```
def selective_sum(n, cond):
    """
    >>> is_odd = lambda x: x % 2 == 1
    >>> selective_sum(5, is_odd) # 5 + 3 + 1 = 9
    9
    >>> bigger_than_10 = lambda x: x > 10
    >>> selective_sum(13, bigger_than_10) # 13 + 12 + 11 = 36
    36
    >>> selective_sum(-1, is_odd) # no positive integers <= 1
    0
    """
    if ________________________________________________:

        return ________________________________________________

    if ________________________________________________:

        return ________________________________________________

    return ________________________________________________
```

```
def selective_sum(n, cond):
    if n <= 0:
        return 0
    if cond(n):
        return n + selective_sum(n - 1, cond)
    return selective_sum(n - 1, cond)
```

Teaching Tips

- Make sure to explain what predicate functions are (functions that return either True or False).
- Start with the base case! If there are no numbers that satisfy `cond` what do we return?
- Then move to the recursive step. How do we keep only the numbers that satisfy `cond`?

3. Write a function `is_sorted` that takes in an integer `n` and returns true if the digits of that number are nondecreasing from right to left.

```
def is_sorted(n):
    """
    >>> is_sorted(2)
    True
    >>> is_sorted(22222)
    True
    >>> is_sorted(9876543210)
    True
    >>> is_sorted(9087654321)
    False
    """

    right_digit = n % 10
    rest =  n // 10
    if rest == 0:
        return True
    elif right_digit > rest % 10:
        return False
    else:
        return is_sorted(rest)
```

First, let's look into the base case. At what point will you know a number is sorted/not sorted immediately?

1. If `n` only has 1 digit or is 0, we know it is definitely sorted with itself. This corresponds to the first if condition, `rest == 0`.
2. If the 2nd-to-last and last digits are not in sorted order, we know the number is not sorted. To do this, we need at least 2 digits in `n` to compare, which is why we check this in **`elif`** after ensuring `n` is not 0.

Next, let's go into the recursive step. We build off of the base cases: if the base cases fail, then we can now work off of the assumption that `n` has at least 2 digits and the last 2 digits of n are in sorted order. Next, notice that after chopping off the last digit, to check that the rest of `n` is sorted, we can use our function `is_sorted` on the number `rest`. So finally, we make the recursive call with `rest` as the argument.

Teaching Tips

- If your students are quiet at the beginning, it might be good to start by going through each of the doctests and asking them why they're true of false.

- Start with the base case! At what point will you know a number is sorted?
- Then move to the recursive step. How do we get the last digit (`n % 10`) and the rest (`n // 10`)?

4. Fill in `collapse`, which takes in a non-negative integer `n` and returns the number resulting from removing all digits that are equal to an adjacent digit, i.e. the number has no adjacent digits that are the same.

```
def collapse(n):
    """
    >>> collapse(12234441)
    12341
    >>> collapse(11200000013333)
    12013
    """
    rest, last = n // 10, n % 10

    if ____________________________________:

        ____________________________________

    elif __________________________________:

        ____________________________________
    else:

        ____________________________________
```

```
def collapse(n):
    rest, last = n // 10, n % 10
    if rest == 0:
        return last
    elif last == rest % 10:
        return collapse(rest)
    else:
        return collapse(rest) * 10 + last
```

Teaching Tips

- Demonstrating with the doc tests is very important- the problem description can be confusing.
- How are we going to traverse the number?
  - As always, we traverse right to left, since traversing left to right will only produce the same results for more work.
  - Knowing this, we can figure `rest` stands for the remaining number and `last` stands for the last digit
- What are the base cases?
  - The simplest possible case is if `n` is one digit, at which there is nothing to compare it to.
- What are the recursive cases?
  - Either we remove a digit or we do not.
  - How do we structure the recursive call when we **don't** want to get rid of `last`? We need to add it back on **after** the recursive call.
- Why does this work?
  - Remind students that part of the recursive leap of faith is to trust that calling `collapse(left)` will remove identical left adjacent numbers.
  - All they need to care about in the moment is whether or not `last` should be removed or not.

5. The *Mandelbrot sequence starting at* (a, b) is a sequence of points in the plane recursively defined by the following:

   - The first term of the sequence is (a, b).
   - If a term in the sequence is (x, y), then the following term is $(x^2 - y^2 + a, 2xy + b)$.

   For example, the first three terms of the Mandelbrot sequence starting at $(1, -1)$ are as follows:

$$(1, -1)$$
$$(1^2 - (-1)^2 + 1, 2(1)(-1) + -1) = (1, -3)$$
$$(1^2 - (-3)^2 + 1, 2(1)(-3) - 1) = (-7, -7)$$

   Write a higher order function `mandelbrot_seq` that accepts two numbers, `start_x` and `start_y`. `mandelbrot_seq` returns a function that takes two numbers `x` and `y` and returns the next term after `(x, y)` in the Mandelbrot sequence starting at `(start_x, start_y)`.

```
def mandelbrot_seq(start_x, start_y):
    """
    >>> seq = mandelbrot_seq(1, -1)
    >>> seq(1, -1)
    (1, -3)
    >>> seq(1, -3)
    (-7, -7)
    """
    def mandelbrot_next(x, y):

        return ____________________, _______________________

    return _____________________________
```

```
def mandelbrot_seq(start_x, start_y):
    def mandelbrot_next(x, y):
        return x ** 2 - y ** 2 + start_x, 2 * x * y + start_y
    return mandelbrot_next
```

The main concept in this problem that will throw students off is that we return a tuple, so ensure to go over tuples if they aren't familiar with them when you walk through the doctests.

6. Write a function `in_or_out`, which returns `False` if any of the first `limit` terms of the Mandelbrot sequence starting at `(start_x, start_y)` is a distance of more than 2 away from the point $(0, 0)$, and `True` otherwise.

```
def in_or_out(start_x, start_y, limit):
    """
    >>> in_or_out(1, -1, 1) # (1, -1) dist is sqrt(2) < 2
    True
    >>> in_or_out(1, -1, 3) # (1, -3) dist is sqrt(10) > 2
    False
    >>> in_or_out(100, 100, 0) # no terms to consider
    True
    """
    next_term = ______________________________________________
    def helper(x, y, limit):

        if ______________________________________________:
            return True

        elif ______________________________________________:
            return False
        else:
            next_x, next_y = next_term(x, y)

            return ______________________________________________

    return ______________________________________________
```

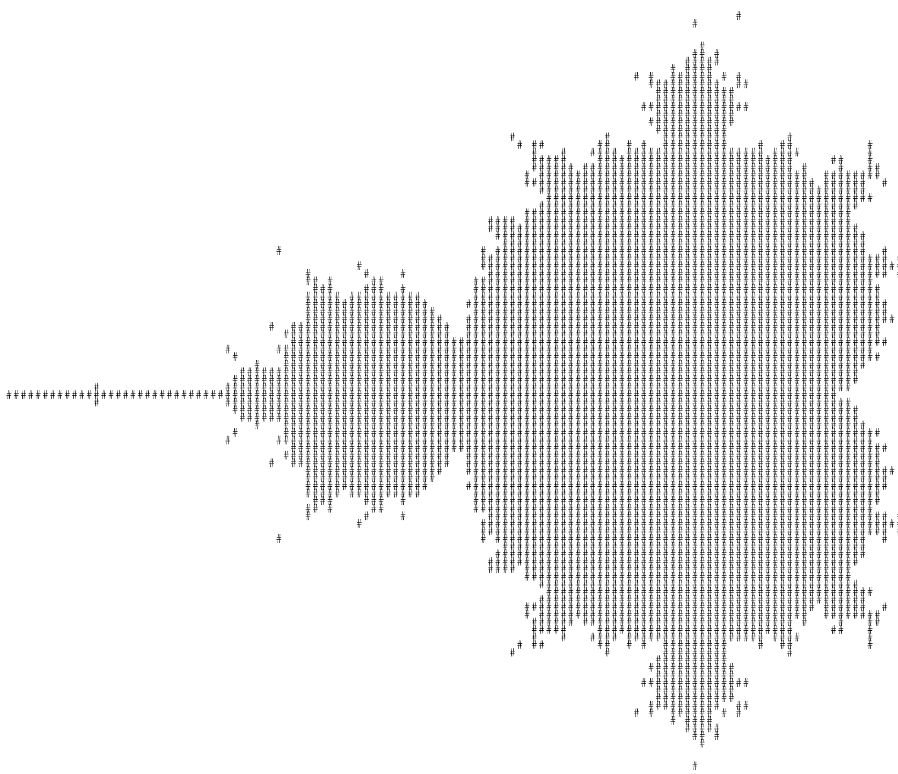
```
def in_or_out(start_x, start_y, limit):
    next_term = mandelbrot_seq(start_x, start_y)

    def helper(x, y, limit):
        if limit <= 0:
            return True
        elif x ** 2 + y ** 2 > 4:
            return False
        else:
            next_x, next_y = next_term(x, y)
            return helper(next_x, next_y, limit - 1)

    return helper(start_x, start_y, limit)
```

The *Mandelbrot set* is the set of all points (x, y) in the plane for which the Mandelbrot sequence starting at (x, y) does not escape to infinity. An approximate picture of this set can be seen by plotting all the points `(x,y)` where `in_or_out(x, y, limit)` returns `True` (don't worry if you don't understand this code):

```
for y in range(50, -50, -1):
    for x in range(-100, 25):
        if in_or_out(x/50, y/50, 20):
            print('#', end='')
        else:
            print(' ', end='')
    print()
```



A major feature of this problem is that it exercises both higher order functions and recursive problem solving skills.

Students might find the use of `next_term` to be a little bit tricky. If this is giving your students trouble, ask them what sort of information the line `next_x, next_y = next_term(x, y)` gives them. Hopefully, this will help them deduce that `next_term` has to be a function that takes in two numbers and returns two numbers, and they'll make the connection with the previous part.

The use of `limit` as a recursive argument might be novel to some students, but it's a pretty common idea that crops up across all types of problems. I'd try to emphasize this to your students.