Level 1 Recall

Original function:

```
def compoundRecursion(principal, compounded, duration, rate, numberOfRecursions):
    if numberOfRecursions == 0:
        totalDuration = compounded * duration
    elif numberOfRecursions != 0:
        totalDuration = duration
    if duration == 0:
        return principal
    else:
        newDuration = totalDuration - 1
        amount = principal * (1 + (rate / compounded))
        return compoundRecursion(amount, compounded, newDuration, rate, 1)
```

Base case:

```
if duration == 0:
    return principal
```

Recursive calls:

```
else:
    return compoundRecursion(amount, compounded, newDuration, rate, 1)
```

Reflection

I understood from memory that the `return` statement usually represented the base case where things will end. However, I couldn't remember if you wanted us to include the `if` statement as part of the base case, so I did the desmos on the 2.7 lesson and found you did want that.

I also understood from memory that the recursive call takes place when the function calls itself, and I saw that it called itself in the else branch.

Level 3 Strategic Thinking

2. Convert the following recursively defined equation into a recursive function/algorithm

$f(n) = f(n-1) + 29.53$, for $f(0) = 125$

```
def recursive_arith(n):
    # base case
    if n == 0:
        return 125

    # recursive call
    return recursive_arith(n - 1) + 29.53
```

Reflection

I made the recursive function from my own knowledge. I made an actual python file and ran it with `recursive_arith(5)` and also did $f(n) = 29.53n + 125$ with $f(5)$ to check that I did it correctly.

Level 4 Extended Thinking

3. Take this iteratively defined function and rewrite it as recursive.

Original function

```
def getTotalIterative(n, total):
    while n >= 0:
        print(n)
        total = total + n
        n -= 1

    return total
```

Recursive version:

```
def getTotalIterative(n, total):
    print(n)

    # base case
    if n == 0:
        return 0

    # recursive call
    return n + getTotalIterative(n - 1, total)
```

7. Given the following recursive function in python, code the inverse function from decimal to binary.

Original function:

```
def binToDec(binary, i):
    decimal = 0
    if binary > 0:
        decimal = decimal + binary % 10 * 2**i + binToDec(binary // 10, i + 1)
    return decimal
```

Recursive Inverse function:

```
def decToBin(decimal, i):
    binary = 0

    # base case, decimal num gets smaller with each call
    # until it reaches 0, at which the binary equivalent is 0,
    # so end the recursion and just add 0 to the binary number
    if decimal == 0:
        return 0

    # recursive call here
    # decimal % 2 tells us if there should be a 1 or 0 at that location
    # 10^i tells us at which digit from the right to put the 1 or 0
    # decimal // 2 removes the rightmost digit from the next call
    # add 1 to i to signal that we want to move left one digit
    binary = binary + (decimal % 2) * 10**i + decToBin(decimal // 2, i + 1)
    return binary
```

Reflection

For problem 3, I ran the original function in python to see what it did. From there, it was pretty easy to do a recursive version without any external tools.

I did problem 7 because I knew how difficult it would be and I thought it'd be a nice challenge. I probably spent an hour on it. At first, I didn't even know what was going on in the original function, so I asked ChatGPT to break it down for me. Link to the conversation here.

Even despite this, I still had trouble making the decimal version. Eventually, I got it without using any external sources. I immediately knew that I wanted to increment i by 1 each function call and that I wanted to call with a different decimal value each time, but I didn't know much else.

I used a calculator and kept trying stuff like "decimal % 10", "decimal // 10", "decimal % 2", etc. to see if it got me the results I wanted. I knew that 7_{10} should be $1 + 10 + 100$, and that should be three recursive calls for three digits. Eventually, I found that "decimal // 2" did the trick.

I made comments like this to convert 7_{10} and 4_{10} to binary to understand what math I had to do with each call:

```
# dec = 7; i = 0; want 1
# dec = 3; i = 1; want 10
# dec = 1; i = 2; want 100
# dec = 0; i = 3; want 0

# dec = 4; i = 0; want 0
# dec = 2; i = 1; want 0
# dec = 1; i = 2; want 100
# dec = 0; i = 3; want 0
```

From these comments, I found that "decimal % 2" correctly told me whether there should be a 1 or 0 in that digit, and I already knew from the beginning that 10**i was going to be involved to get to the tens, hundreds, thousands, etc. places.

Then, I tested my code with different decimal values and correctly got the binary outputs for 4_{10}, 7_{10}, 1_{10}, 0_{10}, and 125_{10}.