Results

Shang Xiao — 1st Attempt

100%

10
Out of 10 points

271:14:23
Time for this attempt

This assessment has unlimited attempts. Take Now

Attempt History

Results	Points	Score	(Highest score is kept)
Attempt 1	10 of 10	100%	(Highest score)

Your Answers:

1 3 / 3 points

What are the 3 parts of a recursive function?

Recursive functions are related to a math concept known as induction. All of them have three things in common:

1. Base case: the case that works for sure and does not depend on anything else, the case that stands on its own, for example: the first case of factorial, which is factorial of 1, written as 1!, is defined to be 1 and the first two numbers that are in the Fibonacci sequence are defined to be 0 and 1.

2. Inductive step: I often interpret this as "the key point" because this is the core of any recurssion function. If we know the solution for n-1 then we can use it to solve n. This steo can also be formulas, for example: the formula for the nth case for the Fibonacci sequence is defined as $F_n = F_{n-1} + F_{n-2}$. We can use this recursive step to code up the solution for n with the following code: (remeber the first element in the sequence starts with 0 because Python starts the first element in its lists from 0)

```
def r_fib(n):
    if n == 0:
        result = 0
    elif n == 1:
        result = 1
    else:
        result = r_fib(n - 2) + r_fib(n - 1)
    return result

def main():
    print(r_fib(6))

main()
```

3. Verify if the repeated recursive step will lead to the base case. We have to be careful here because sometimes it does not and that is why we need to perform this third step as a complete resursive function. Within the resursive function, the function must be calling itself recursively, hence the name recursion.

Correct

2 3 / 3 points

Write a recursive function which, given a list of numbers, finds the smallest number in the list.

```
def FindSmallestNumber(numbers):
    if(len(numbers) == 1):
        return numbers[0]
    return min(numbers[0], FindSmallestNumber(numbers[1:]))
```

Correct

3 2 / 2 points

What is one thing that recursion is bad at?

Eating memories. If we were to write the Fibonacci sequence in two versions:

1. The *while* loop version

```
def fib(n):
    if n == 0:
        result = 0
    elif n == 1:
        result = 1
    else:
        i = 1
        current = 1
        previous = 0
        while n > i:
            i += 1
            next = previous + current
            previous = current
            current = next
        result = current
    return result
```

2. The recursion version

```
def r_fib(n):
    if n == 0:
        result = 0
    elif n == 1:
        result = 1
    else:
        result = r_fib(n - 2) + r_fib(n - 1)
    return result
```

If we set the parameter equal to 10 for both versions, the time difference that takes both to run and hand back us the results are not noticeable; however, if we set the parameter to 999 this time, the while loop version will also hand back the answer to us immediately, but the recursion version takes much longer time to hands back the answer. This is because the recursion algorithm repeatedly computes what it already computed over and over again until it reaches the largest element. When computing the Fibonacci of 10, the recursion function actually computing Fibonacci of 1 over 50 times.

This is a problem of recursive algorithms: stack growth. Since recursion is functions calling themselves, each time a function makes a nested call to another function, another memory is allocated to the execution stack. When handling large numbers in this example, recursion is NOT the way to go since it might take much longer to process.

Correct

4 2 / 2 points

You have written a recursive function that is not working as intended. What are two things you can do to debug the recursive function?

1. Use the *print* statement in a enhanced way. We need to more explicitly indicate where are we for each recursive function call, this means we do not use sequential calls with *start* and *end*, but use recursive calls that indicate where are we currently at, such as *start*, *start nested*, *end nested*, *end*.

```
def r_factorial(n):
    print("START", n)
    if n == 1:
        product = 1
    else:
        product = r_factorial(n - 1) * n
    print("END",n,"RETURNING:",product)
    return product

def main():
    # print(factorial (4))
    print(r_factorial (4))

main()
```

2. Use indentation to indicate each level of the recursion. The indentation marker can be any sign, such as < or \.

```
def r_factorial(n, level):
    print("<"*level, "START", n)
    if n == 1:
        product = 1
    else:
        product = r_factorial(n - 1, level + 1) * n
    print("<"*level, "END",n,"RETURNING:",product)
    return product

def main():
    # print(factorial (4))
    print(r_factorial (4, 0))

main()
```

Correct