LAB 04

Recursion, Tree Recursion, Python Lists

Mingxiao Wei
mingxiaowei@berkeley.edu

Sep 19, 2022

LOGISTICS 🏡

- Lab 04 (this lab) due Wed 09/21
- Homework 03 due Thu 09/22
    - There's an optional contest in the homework! - Not worth any credit, but if you want to have fun with higher order functions, go for it 😼
- If you late-enroll in the course, talk to me after section or email me - we have some new accommodation policies!
- The second round of scores has been released on howamidoing.cs61a.org. If you have any issues with your grade:
    - For assignments, submit a regrade request on howamidoing.cs61a.org - please keep it short (~ 2 sentences) since there's a character limit and words exceeding the limit will not go through!
    - For section attendance, email me. If you were not able to make it to section and emailed me beforehand, you should be marked as excused absence and receive attendance credit for that section.

HIGHLIGHT FROM LAST TIME 👀

If you could hold any number of pigeons, how many would you hold and why?

- "0 i am not risking getting pecked
- "1, handle w care"
- "One because I would like to bond on an individual level with my pigeon."
- "1 and make sure it doesn't poop"
- "2 so i can hold one in each hand"
- "I think 3 would be a solid number, one for each hand and a squishy bird for the middle :)"
- "3 cuz my lucky number is 3"
- "4 because two is too few and I want each pigeon to have a partner"
- "Four pigeons, two on the shoulders and two on my hands, as there is already someone who does this near Telegraph, Seeing another person holding as much pigeons would be strange."
- "300 bcz I want to be in the Guiness Book of World Records"
- "1000, I just wanna see if I can hold all of them."
- "9999999999999999999999999999999999"

INTRODUCING OUR AI'S 🥳

AI = Academic Intern

They are excellent helpers in labs!

- Christy Quang
- Gabriella Skavdahl
- Jiaxin Fang

RECURSION 🐚

INTRODUCTION

- A recursive function is a function that is defined in terms of itself - i.e., the function calls itself
- 3 main steps in a recursive function:
    1. Base case - the simplest function input, or the stopping condition for the recursion
    2. Recursive call on a smaller problem - calling the function on a smaller problem that our current problem depends on. We can assume that a recursive call on this smaller problem will give us the expected result as long as the we implement other parts correctly - the idea of the "recursive leap of faith".
    3. Solve the larger problem / Combination step - usually by using the result from the recursive call to figure out the result of our current problem

PROBLEM-SOLVING STRATEGIES

- Base case:
    - Usually hinted in the doctests - make sure to read them!
    - Think about different possibilities of "when to stop"
    - If multiple arguments change throughout the recursive calls, make sure to include a base case for *each* argument - this is especially relevant in tree recursion
    - If there are multiple base cases, think about whether or not the order of them matters
- Recursive call:
    - Think about how to break down the problem into smaller ones
    - Make sure that the base cases are *reachable* - that is, the argument changes toward the base case each time
- Combination step:
    - Assuming the implementation is correct (this is the recursive leap of faith), ask yourself: what will the recursive call return?

HELPER FUNCTIONS FOR RECURSION

- When to use:
    - Need to keep track of more variables than the given parameters of the outer function
    - Need the original parameter from the outer function for each recursive call AND some other parameters that change throughout the recursive call
- Where - usually, though not necessarily, nested within the original function
- How - define the helper function, and return a call to it with appropriate initial arguments

TREE RECURSION 🌴

TREE RECURSION

- A tree recursive function is a recursive function that makes more than one call to itself, resulting in a tree-like series of calls.
- For example, let's say we want to recursively calculate the n^{th} Fibonacci number, defined as:

```python
def fib(n):
    if n == 0 or n == 1:
        return n
    return fib(n - 1) + fib(n - 2)
```

- Now, what happens when we call `fib(4)`?
  - Each `fib(i)` node represents a recursive call to `fib`.
  - For `i >= 2`, each recursive call `fib(i)` makes another two recursive calls, which are to `fib(i - 1)` and `fib(i - 2)`.
  - Whenever we reach a `fib(0)` or `fib(1)` node, we can directly return 0 or 1, since these are our base cases.

COUNT PARTITIONS REVISIT

Given two positive integers `n` and `m`, return the number of ways in which `n` can be expressed as the sum of positive integer parts up to `m` in increasing order.

- Recursive case: Since each integer part is up to `m`, at each step, where each step generates one number in the partition, we have two choices:
    1. Use `m` to partition `n`, so that at the next step, `n` becomes `n - m`, and the largest possible part is still `m`
    2. Don't use `m`. So at the next step, `n` remains unchanged, but `m` becomes `m - 1` (we choose not to use the largest possible part, `m`, so the next largest possible one is `m - 1`)
- The two choices will result in two distinct sets of results, since in the first one we use `m` to partition, while in the second one we use at most `m - 1`
- Therefore, the total # of partitions = # partitions from choice 1 + # partitions from choice 2

COUNT PARTITIONS REVISIT

Given two positive integers `n` and `m`, return the number of ways in which `n` can be expressed as the sum of positive integer parts up to `m` in increasing order.

- Base case:
  - `n == 0` - note that since `n` and `m` are positive integers according to the problem description, when `n` is 0, it could only be the case where `n - m` results in 0 from the previous recursive call. In other words, when `n` is 0, it menas that we've successfully partitioned `n` so that there's nothing left to partition. In this case, return 1, since we found one valid parition.
  - `n < 0` - similarly, since the original input to the function must be positive integers, a negative `n` can only result from `n - m` from the last step. In this case, `m` was greater than `n` from the last step, indicating that the partition was not succeessful.
  - `m == 0` - also similarly, a negative `m` can only result from `m - 1` from the last step. Since the question requires that all parts of a partition are positive integers, such an partition is invalid.

COUNT PARTITIONS REVISIT

Given two positive integers `n` and `m`, return the number of ways in which `n` can be expressed as the sum of positive integer parts up to `m` in increasing order.

```python
def count_partitions(n, m):
    if n == 0:
        return 1
    elif n < 0 or m == 0:
        return 0
    else:
        with_m = count_partitions(n-m, m)
        without_m = count_partitions(n, m-1)
        return with_m + without_m
```

LIST COMPREHENSION 📄

LIST COMPREHENSION

- List comprehensions are a compact and powerful way of creating new lists out of sequences.

```
[<expr> for <var> in <seq> if <cond>]
```

- In English, this translates to:
    1. For each element in the sequence `<seq>`, bind it to the variable name `<var>`.
    2. If the element satisfy the condition `<cond>` (or skip this check if there's no condition), evaluate the expression `<expr>`, and add the value from `<expr>` to the resulting list.
- Note:
    - `if <cond>` is optional.
    - `<expr>` and `<cond>` may refer to `<var>`, which is essentially every element in the sequence

LIST COMPREHENSION - WWPD

```
>>>  lst_1 = [1, 2, 3, 4]
>>>  lst_2 = [5, 6, 7, 8]
>>>  [i ** 2 for i in lst_1]
[1, 4, 9, 16]
>>>  ['go' for j in lst_2 if j > 5]
['go', 'go', 'go']
>>>  [print(k) for k in lst_2 if k % 2 == 0]
6
8
[None, None]
```

ATTENDANCE! 🤠

go.cs61a.org/mingxiao-att

- The attendance form and slides are both linked on our section website!
- If you finish early, let me or any of the AI's know and we'll check you off
- Once again, please do remember to fill out the form by midnight today!!