

# LAB 04

---

## Recursion, Tree Recursion, Python Lists


























Mingxiao Wei

[mingxiaowei@berkeley.edu](mailto:mingxiaowei@berkeley.edu)

Feb 14, 2023

# FROM LAST TIME... 👁️👁️

Describe your feelings about the midterm with one emoji (or whatever you like)

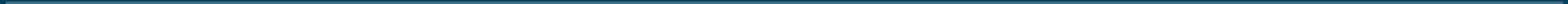
							:)	:))	: -)	cool
								 	[doge]	it was ok
								: 0	: (	stressful

# LOGISTICS

---

- Lab 04 due Wed 02/15 - make sure to have a submission by then
- Homework 03 due Fri 02/17
- Regrade requests for MT1 are due tomorrow
- Go to office hour/post on Ed if you need help on anything! We are here to support you :))

# 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:
  - 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), **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 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:

```
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  $m$ .

- Recursive case: Since each integer part is up to  $m$ , at each step, where one number (part) in the partition is generated, 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. Do not 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  $m$ .

- 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, we've successfully partitioned  $n$  so that there's nothing left to partition. In this case, return 1, since we found one valid partition.
  - $n < 0$  - 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 successful.
  - $m == 0$  - 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 a 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.

```
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
```

# PYTHON LISTS



# PYTHON LISTS - INTRO

---

- A data structure that can store multiple elements in a defined order.
- Each element can be of any type, even a list itself.
- Created by wrapping square brackets around comma-separated expressions
- `len(<seq>)` is a built-in function that takes in a sequence and returns the length (i.e., the number of elements in the sequence)

```
>>> a = ['a', 'b']
>>> b = [1, 2]
>>> len(a)
2
>>> c = [a, b, [3, 4], 5]
>>> c
[['a', 'b'], [1, 2], [3, 4], 5]
>>> len(c)
4
```

# PYTHON LISTS - INDEXING

- Each element has its corresponding index, starting from 0 (i.e., "zero-indexed")
  - The index of the last element is `len(lst)-1`
- When index is negative, it means starting from the end of the list
  - `-i` is equivalent to `len(lst)-i`
  - `lst[-1]` is the last element, `lst[-2]` is the second to last element, etc.

```
>>> a = ['a', 'b']
>>> b = [1, 2]
>>> a[-1]
'b'
>>> c = [a, b, [3, 4], 5]
>>> c
[['a', 'b'], [1, 2], [3, 4], 5]
>>> c[1][1]
1
```

# PYTHON LISTS - OTHER OPERATIONS

---

- Concatenation - use `+` to concatenate multiple lists together

```
>>> a = ['say']
>>> b = ['cheese', '!']
>>> a + b # same as add(a, b)
['say', 'cheese', '!']
```

- Repetition - use `*` to duplicate a list certain times

```
>>> a = [6, 1]
>>> a * 3 # same as a + a + a, or mul(a, 3)
[6, 1, 6, 1, 6, 1]
>>> add(mul(a, 2), mul(['a'], 2)) # a * 2 + ['a'] * 2
[6, 1, 6, 1, 'a', 'a']
```



# PYTHON LISTS - OTHER OPERATIONS

---

- Checking element existence - use `in` to check whether or not some value is contained in the list
- To check for non-existence, use `elem not in lst` or `not (elem in lst)`

```
>>> a = ['o', 'p', ['oops']]
```

```
>>> 'p' in a
```

```
True
```

```
>>> 'oops' in a # in only checks the elements of the outer list
```

```
False
```

```
>>> 'op' not in a
```

```
True
```

```
>>> not('op' in a)
```

```
True
```

# LIST SLICING

```
lst[ <start> : <end> : <step size> ]
```

- List slicing creates a copy of part or all of the list.
  - It takes elements at index `start + step`, `start + 2 * step`, ... and stops before `end`, and makes those selected elements into a new list
- `start`
  - index to start at, *inclusive*, default to 0
- `end`
  - index to end by, *exclusive*, default to `len(lst)`\*
  - when negative, counts from the end of the list, similar to negative indexing
- `step size`
  - the difference between indices to include, default to 1
  - negative steps means stepping backwards

\* when step is positive - when step is negative, start index defaults to the end of the list and end index defaults to the start of the list

# LIST SLICING

```
>>> lst = [1, 2, 3, 4, 5]
>>> lst[1:3]
[2, 3]
>>> lst[1:]
[2, 3, 4, 5]
>>> lst[:-2]
[1, 2, 3]
>>> lst[1::2]
[2, 4]
>>> lst[::-1] # reverse the list
[5, 4, 3, 2, 1]
>>> lst[5:9] # list slicing won't cause an index error
[]
```

Takeaway: list slicing picks elements at indices `start`, `start + step`, `start + 2 * step`, ... and stops before `end`, and makes those selected elements into a new list

# NOW IT'S LAB TIME 🤠

---

- Get started on the lab and raise your hand whenever you need help!
- Get to know your neighbors and collaborate if you'd like!
- Slides: [go.cs61a.org/mingxiao-index](https://go.cs61a.org/mingxiao-index)
- Leave any anonymous feedback here: [go.cs61a.org/mingxiao-anon](https://go.cs61a.org/mingxiao-anon)

AND REMEMBER TO GET  
CHECKED OFF! 🧺

---

[go.cs61a.org/mingxiao-att](https://go.cs61a.org/mingxiao-att)

The secret phrase is ...  
(NOT 3 dots! I'll announce it 🙊)

