Computational Structures in Data Science

Recursion (2)

Week 4, Summer 2024. 7/9 (Tues)

Lecture 12

Berkeley
UNIVERSITY OF CALIFORNIA

Announcements

- HW06, Lab06 out today! (Due: 7/13)
- Project01 (“Maps”) Checkpoint due tomorrow (7/10)
- “(Urgent) Midterm Exam Scheduling” due TONIGHT (7/9 11:59 PM PST)
- Study tip: past C88C exams can be found here: https://c88c.org/sp24/articles/resources.html#past-midterms
  - Take a look to get a sense of what C88C exams tend to look like. (I highly, highly encourage this)

Announcements: Midterm scheduling!

- IMPORTANT: Complete the “Midterm Exam Scheduling” form on Gradescope
  - Due: Tuesday July 9th, 11:59 PM PST
  - It is required that every student fill this out. If you don’t fill this out, you will risk missing the midterm.
    - Please, please do this ASAP! Thank you!
- (DSP students with +50% exam time) We have sent you a Google Form to schedule your exam. Please fill this out ASAP!
  - But, you should still fill out the above Gradescope “Midterm Exam Scheduling” form as well!

Midterm content

- Midterm will cover content from start of course up to (and including) OOP+Inheritance, aka:
  - Start (inclusive): Lecture 01: "Welcome & Intro" (6/17)
  - End (inclusive): Lecture 15: "OOP – Inheritance" (7/15)
- Midterm will be done through Zoom + Gradescope
- Study tip: past C88C exams can be found here: https://c88c.org/sp24/articles/resources.html#past-midterms
  - Take a look to get a sense of what C88C exams tend to look like. (I highly, highly encourage this)
    - "Be prepared" – Boy Scouts
    - "Luck is when preparation meets opportunity" – Roman philosopher Seneca

Midterm logistics

- The midterm will be held over Zoom + Gradescope
  - You must have your camera + screen sharing on during the entire exam, and we will be doing screen+camera recording.
- You must take the exam in a quiet room with no other students present
- Things to bring to the exam (and nothing else!):
  - Photo ID. Ideally your UCB student ID, but anything with your name + photo is fine, eg: Passport, driver's license, etc.
  - (Optional) Five (5) pages of handwritten (not typed!) notes
  - (Optional, recommended) Additional blank scratch paper, pencil/pen/eraser.
- We will provide everyone with a 1-2 page digital PDF of additional reference
- Other than the above notes, the exam will be closed book, closed notes.
- (For more info, stay tuned for an Ed post)

Lecture overview

- More recursion practice
  - Recursion over sequences
    - “Deep” recursion

The Recursive Process

Recursive solutions involve two major parts:

- Base case(s), the problem is simple enough to be solved directly
- Recursive case(s). A recursive case has three components:
  - **Divide** the problem into one or more simpler or smaller parts
  - **Invoke** the function (recursively) on each part, and
  - **Combine** the solutions of the parts into a solution for the problem.

Why learn recursion?

- Recursive data is all around us!
  - Take CS61B (data structures), CS70 (discrete math), CS164 (Programming Languages), Data 101 (Data Eng) for more examples where you'll encounter recursion
- Trees (post-midterm) and Graphs are structures which are recursive in nature.
  - E.g. A social network is a graph of friends with connections to other friends, with connections to other friends.
  - Analyzing "chains" of data, can benefit from recursion
- Next Lecture: Problems that "branch" out:
  - generating subsets and permutations
  - calculating Fibonacci numbers

Computational Structures in Data Science

Palindromes

Learning Objectives

- Compare Recursion and Iteration to each other
  - Translate some simple functions from one method to another
- Write a recursive function
  - Understand the base case and a recursive case

(Recall) Example: Reverse (recursively)

- Now that we've identified the recursive process for string reversal, let's translate it into Python code

```
"hello" -> "olleh"
```

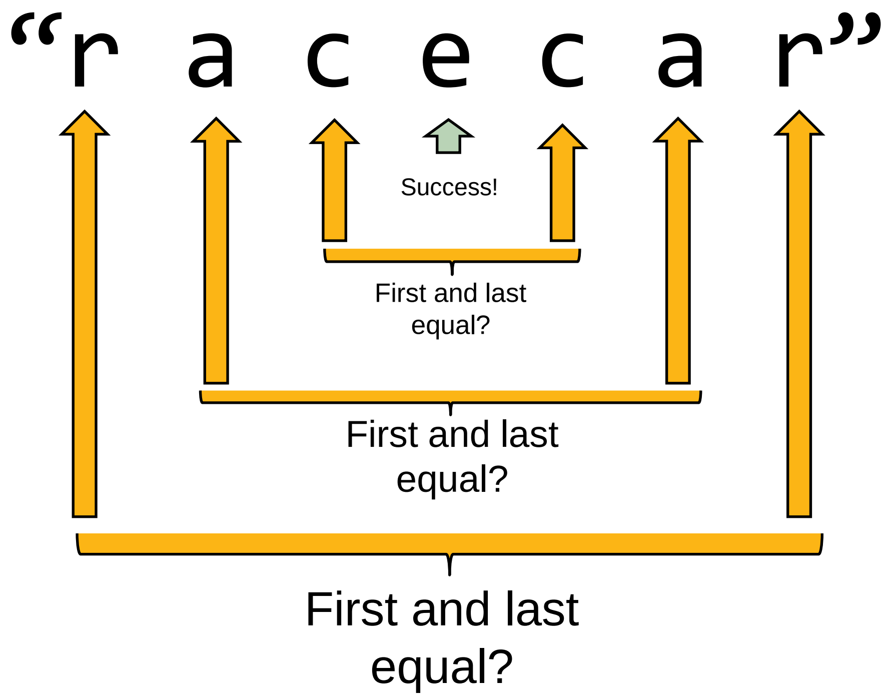
Base cases
What are the simplest ("trivial") strings to reverse?
reverse_str("") -> ""
reverse_str("o") -> "o"

Recursive structure
"olle" + "h" -> "olleh", or in code:
reverse_str("hello") is reverse_str("ello") + "h"
Or, more generally:
reverse_str(s) is reverse_str(s[1:]) + s[0]

```
def reverse_str(s):
    # Base cases
    if len(s) == 0 or len(s) == 1:
        return s
    # Recursive cases
    return reverse_str(s[1:]) + s[0]
```

Palindrome – Alternative Approach

- Instead of using reverse_str(), let's try a different recursive approach to `is_palindrome()`
- **Idea**: Compare first / last letters, working our way towards the middle
-



```
def is_palindrome(s):
    return s == reverse_str(s)

>>> is_palindrome("too hot to hoot")
True
>>> is_palindrome("meow")
False
```

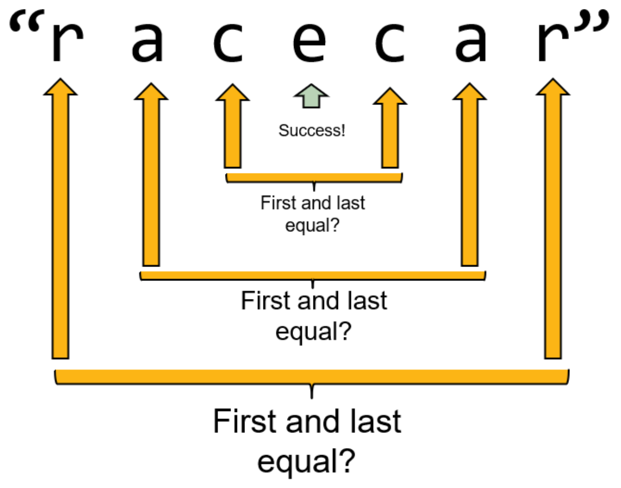
Question: can you express this approach as a recursive process?

What are the **base cases**?

What are the **recursive cases**?

Palindrome – Alternative Approaches

- Idea: Compare first / last letters, working our way towards the middle
- Base Case?
  - What is the *smallest* word that is a palindrome?
    - A 1-letter word!
    - A 0 letter word? Maybe?
  - Recursive case?
    - If the first and last letter are the same, check the "inner word“ (aka s[1:-1])
    - If they're not → return False

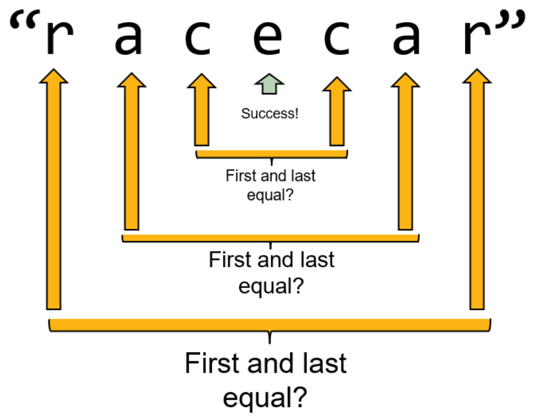


```
def is_palindrome(s):
    # FILL ME IN
```

Question: implement this recursive process as a Python recursive function

Palindrome – Alternative Approaches

- Idea: Compare first / last letters, working our way towards the middle
- Base Case?
  - What is the *smallest* word that is a palindrome?
    - A 1-letter word!
    - A 0 letter word? Maybe?
  - Recursive case?
    - If the first and last letter are the same, check the "inner word" (aka s[1:-1])
    - If they're not → return False



```
def is_palindrome(s):
    if len(s) <= 1:
        return True
    if s[0] != s[-1]:
        return False
    else:
        return is_palindrome(s[1:-1])
```

(Quick Python check)

- Question: how to write this as a conditional expression?

```python
def is_palindrome(s):
    if len(s) <= 1:
        return True
    if s[0] != s[-1]:
        return False
    else:
        return is_palindrome(s[1:-1])
```

Answer:

```python
def is_palindrome(s):
    if len(s) <= 1:
        return True
    return False if s[0] != s[-1] else is_palindrome(s[1:-1])
```

(Quick Python check)

- Question: does changing the basecase from (A) to (B) still work? Aka “<=“ vs “<“

```
def is_palindrome_a(s):
    if len(s) <= 1:
        return True
    if s[0] != s[-1]:
        return False
    else:
        return is_palindrome(s[1:-1])
```

```
def is_palindrome_b(s):
    if len(s) < 1:
        return True
    if s[0] != s[-1]:
        return False
    else:
        return is_palindrome(s[1:-1])
```

Answer: yes, it still works!

Base cases: when to start/stop?

- In your studies, you may find that one recursive function's implementation has different base cases than yours (eg more, or fewer), but both implementations work!
- Reasoning: one can always add "more" base cases and still have a correct solution
- Some base cases can be "redundant", like this `len(s) == 2` base case.
  - However, it's "harmless" and doesn't impact correctness.
- Tip: try to reduce the number of "redundant" base cases in your code.
  - But not at the expense of readability (an "art", etc)

```python
def is_palindrome(s):
    if len(s) <= 1:
        return True
    if s[0] != s[-1]:
        return False
    else:
        return is_palindrome(s[1:-1])

def is_palindrome_more(s):
    if len(s) <= 1:
        # '', 'a'
        return True
    if len(s) == 2:
        # 'aa'
        return s[0] == s[-1]
    if s[0] != s[-1]:
        return False
    else:
        return is_palindrome_more(s[1:-1])
```

Computational Structures in Data Science

Recursion With Lists

Berkeley
UNIVERSITY OF CALIFORNIA

Exercise: Return minimum of a list (recursion)

- Let's define a recursive function that, given a list of numbers, returns the smallest value.
- Question: what is the recursive structure of this function? (Base cases, recursive cases?)

```
>>> print(min_r([3, 2, -1, 42]))
-1
```

Base cases?
If list has only one number, return that number (easy)

Recursive cases?
Return the smaller of (1) the first number in the input list, and (2) the smallest number in the rest of the list (eg lst[1:])

Another Example – Finding a Minimum

indexing an element of a sequence

```
def first(s):
    """Return the first element in a sequence."""
    return s[0]
def rest(s):
    """Return all elements in a sequence after the first"""
    return s[1:]
```

Slicing a sequence of elements

```
def min_r(s):
    """Return minimum value in a sequence."""
    if
```

Base Case

```
    else:
```

Recursive Case

- Recursion over sequence length

Exercise: Return minimum of a list (recursion)

- Question: why isn't the base case \`len(s) == 0\`?

Answer: the minimum value of an empty list is ill defined. So, it makes sense that, in our Python code, we don't have a base case to cover it.

(Aside) If one wants to get a little fancy: technically you could declare that "the minimum value of an empty list is +Infinity". In Python, +Infinity is \`float('inf')\`.

With this, you'll find that things will actually work! Which is interesting. But, kind of weird too.

Python's stance? It throws an error if you call \`min()\` on an empty list. Not a bad design choice IMO!

```
def first(s):
    return s[0]
def rest(s):
    return s[1:]

def min_r(s):
    """Return minimum value in a sequence."""
    if len(s) == 1:
        return first(s)
    else:
        return min(first(s), min_r(rest(s)))
```

```
>>> print(min_r([3, 2, -1, 42]))
-1
```

Exercise: min_deep

- Suppose we want to find the smallest value in a list that can contain either (1) a number, of (2) a list.
- Question: what is the recursive structure? (Base cases, recursive cases)

```
>>> min_deep([1, -1, 4])
-1
>>> min_deep([1, [2], [3, [-4]], 9])
-4
```

Base cases:
If the input length is 1 AND the first element type is a number: return the number

Recursive case:
If the input length is 1 AND the first element type is a list: return the min_deep() of the first element

Else: return the smaller of the two:
(1) The smallest value in the first element, and (2) calling min_deep() on the rest of the sequence.

New: two recursive cases!

This is required because the first list element may itself be a list that we need to recurse into. Wow!

Question: implement this as a Python recursive function!

Hint: use `isinstance(thing, list)` to see if the type of something is a list type.

Exercise: min_deep

- Suppose we want to find the smallest value in a list that can contain either (1) a number, of (2) a list.

Base cases:
If the input length is 1 AND the first element type is a number: return the number

Recursive case:
If the input length is 1 AND the first element type is a list: return the min_deep() of the first element

Else: return the smaller of the two:
(1) The smallest value in the first element, and (2) calling min_deep() on the rest of the sequence.

```python
def min_deep(lst):
    if len(lst) == 1 and not isinstance(lst[0], list):
        return lst[0]
    elif len(lst) == 1 and isinstance(lst[0], list):
        return min_deep(lst[0])
    else:
        if isinstance(lst[0], list):
            first_min = min_deep(lst[0])
        else:
            first_min = lst[0]
        return min(first_min, min_deep(lst[1:]))
```

```
>>> min_deep([1, -1, 4])
-1
>>> min_deep([1, [2], [3, [-4]], 9])
-4
```

Compare: min_r() vs min_deep()

```
def min_r(nums):
    if len(nums) == 1:
        return nums[0]
    return min(nums[0], min_r(nums[1:]))

>>> min_r([3, 2, -1, 42])
-1
```

```
def min_deep(lst):
    if len(lst) == 1 and not isinstance(lst[0], list):
        return lst[0]
    elif len(lst) == 1 and isinstance(lst[0], list):
        return min_deep(lst[0])
    else:
        if isinstance(lst[0], list):
            first_min = min_deep(lst[0])
        else:
            first_min = lst[0]
        return min(first_min, min_deep(lst[1:]))

>>> min_deep([1, -1, 4])
-1
>>> min_deep([1, [2], [3, [-4]], 9])
-4
```

Takeaway: min_deep() is indeed more complicated, but the two functions share the **same guiding principles**:

First: identify the recursive structure (base cases, recursive cases).
Then: translate into Python code.

Min_deep() + keeping track of depth

Question: what if we wanted to modify min_deep() to return not only the minimum value, but also the depth at which the minimum value was found? What changes do we need to make?

Hint: define a helper function that keeps track of the state as an argument.

```
def min_deep_2(lst):
    if len(lst) == 1 and not isinstance(lst[0], list):
        return lst[0]
    elif len(lst) == 1 and isinstance(lst[0], list):
        return min_deep(lst[0])
    else:
        if isinstance(lst[0], list):
            first_min = min_deep(lst[0])
        else:
            first_min = lst[0]
        return min(first_min, min_deep(lst[1:]))
```

```
# return value is: (min_val, int depth)
>>> min_deep_2([1, -1, 4])
(-1, 0)
>>> min_deep_2([1, [2], [3, [-4]], 9])
(-1, 2)
```

Min_deep() + keeping track of depth

```
def min_deep_2_helper(lst, cur_depth):
    if len(lst) == 1 and not isinstance(lst[0], list):
        return lst[0], cur_depth
    elif len(lst) == 1 and isinstance(lst[0], list):
        return min_deep_2_helper(lst[0], cur_depth + 1)
    else:
        if isinstance(lst[0], list):
            first_min, first_depth = min_deep_2_helper(lst[0], cur_depth + 1)
        else:
            first_min = lst[0]
            first_depth = cur_depth
        rest_min, rest_depth = min_deep_2_helper(lst[1:], cur_depth)
        if first_min < rest_min:
            return first_min, first_depth
        else:
            return rest_min, rest_depth

def min_deep_2(lst):
    return min_deep_2_helper(lst, 0)
```

(wow, a lot happening there!)

```
# return value is: (min_val, int depth)
>>> min_deep_2([1, -1, 4])
(-1, 0)
>>> min_deep_2([1, [2], [3, [-4]], 9])
(-1, 2)
```

Min_deep() + keeping track of depth

```
def min_deep_2_helper(lst, cur_depth):
    if len(lst) == 1 and not isinstance(lst[0], list):
        return lst[0], cur_depth

    elif len(lst) == 1 and isinstance(lst[0], list):
        return min_deep_2_helper(lst[0], cur_depth + 1)
    else:
        if isinstance(lst[0], list):
            first_min, first_depth = min_deep_2_helper(lst[0], cur_depth + 1)
        else:
            first_min = lst[0]
            first_depth = cur_depth
        rest_min, rest_depth = min_deep_2_helper(lst[1:], cur_depth)
        if first_min < rest_min:
            return first_min, first_depth
        else:
            return rest_min, rest_depth

def min_deep_2(lst):
    return min_deep_2_helper(lst, 0)
```

Base case: if the input is trivial (list with a single number), then the output depth is the same as the current depth.

Ex: min_deep_2_helper([-1], 0) -> (-1, 0)

```
# return value is: (min_val, int depth)
>>> min_deep_2([1, -1, 4])
(-1, 0)
>>> min_deep_2([1, [2], [3, [-4]], 9])
(-1, 2)
```

Min_deep() + keeping track of depth

```
def min_deep_2_helper(lst, cur_depth):
    if len(lst) == 1 and not isinstance(lst[0], list):
        return lst[0], cur_depth
    elif len(lst) == 1 and isinstance(lst[0], list):
        return min_deep_2_helper(lst[0], cur_depth + 1)
    else:
        if isinstance(lst[0], list):
            first_min, first_depth = min_deep_2_helper(lst[0], cur_depth + 1)
        else:
            first_min = lst[0]
            first_depth = cur_depth
        rest_min, rest_depth = min_deep_2_helper(lst[1:], cur_depth)
        if first_min < rest_min:
            return first_min, first_depth
        else:
            return rest_min, rest_depth

def min_deep_2(lst):
    return min_deep_2_helper(lst, 0)
```

Recursive case: if the input is a list with a single sublist, then we need to step into one level lower, aka increase our depth by 1.

This is why we pass `cur_depth + 1` into the recursive call.

```
# return value is: (min_val, int depth)
>>> min_deep_2([1, -1, 4])
(-1, 0)
>>> min_deep_2([1, [2], [3, [-4]], 9])
(-1, 2)
```

Min_deep() + keeping track of depth

```python
def min_deep_2_helper(lst, cur_depth):
    if len(lst) == 1 and not isinstance(lst[0], list):
        return lst[0], cur_depth
    elif len(lst) == 1 and isinstance(lst[0], list):
        return min_deep_2_helper(lst[0], cur_depth + 1)
    else:
        if isinstance(lst[0], list):
            first_min, first_depth =
min_deep_2_helper(lst[0], cur_depth + 1)
        else:
            first_min = lst[0]
            first_depth = cur_depth
        rest_min, rest_depth = min_deep_2_helper(lst[1:], cur_depth)
        if first_min < rest_min:
            return first_min, first_depth
        else:
            return rest_min, rest_depth

def min_deep_2(lst):
    return min_deep_2_helper(lst, 0)
```

```
# return value is: (min_val, int depth)
>>> min_deep_2([1, -1, 4])
(-1, 0)
>>> min_deep_2([1, [2], [3, [-4]], 9])
(-1, 2)
```

Recursive case: this departs from the original. Reasoning:

The smallest value can come from one of two sources: the first element, or from the recursive call on the rest of the sequence.

Each of these two sources will have their own depth associated with their respective smallest values.

Ex: for the input lst=[[1], [2, [3]]], cur_depth=0, the two recursive calls would yield:
First element: min_val=1, depth=1
Rest: min_val=3, depth=2

As the global min value is min_val=1, we want to make sure we also return its associated depth=1.

Thus, there is some **additional bookkeeping** required to keep track of which depth corresponds to which source.

Min_deep() + keeping track of depth

```python
def min_deep_2_helper(lst, cur_depth):
    if len(lst) == 1 and not isinstance(lst[0], list):
        return lst[0], cur_depth
    elif len(lst) == 1 and isinstance(lst[0], list):
        return min_deep_2_helper(lst[0], cur_depth + 1)
    else:
        if isinstance(lst[0], list):
            first_min, first_depth =
min_deep_2_helper(lst[0], cur_depth + 1)
        else:
            first_min = lst[0]
            first_depth = cur_depth
        rest_min, rest_depth = min_deep_2_helper(lst[1:], cur_depth)
        if first_min < rest_min:
            return first_min, first_depth
        else:
            return rest_min, rest_depth

def min_deep_2(lst):
    return min_deep_2_helper(lst, 0)
```

```python
# return value is: (min_val, int depth)
>>> min_deep_2([1, -1, 4])
(-1, 0)
>>> min_deep_2([1, [2], [3, [-4]], 9])
(-1, 2)
```

Recursive case:

With that said:

First, we take a look at the first element. If it's a list, we recurse into it (and increment `cur_depth + 1` as we're going down in depth).

If it's a number, we don't need to increment `cur_depth`.

Min_deep() + keeping track of depth

```python
def min_deep_2_helper(lst, cur_depth):
    if len(lst) == 1 and not isinstance(lst[0], list):
        return lst[0], cur_depth
    elif len(lst) == 1 and isinstance(lst[0], list):
        return min_deep_2_helper(lst[0], cur_depth + 1)
    else:
      if isinstance(lst[0], list):
            first_min, first_depth = min_deep_2_helper(lst[0], cur_depth +
1)
        else:
            first_min = lst[0]
            first_depth = cur_depth
            rest_min, rest_depth =
min_deep_2_helper(lst[1:], cur_depth)
        if first_min < rest_min:
            return first_min, first_depth
        else:
            return rest_min, rest_depth

def min_deep_2(lst):
    return min_deep_2_helper(lst, 0)
```

Recursive case:

Next, we recurse on the rest of the list. Notably, `cur_depth` remains the same since we’re remaining at the same depth.

```python
      # return value is: (min_val, int depth)
      >>> min_deep_2([1, -1, 4])
      (-1, 0)
      >>> min_deep_2([1, [2], [3, [-4]], 9])
      (-1, 2)
```

Min_deep() + keeping track of depth

```
def min_deep_2_helper(lst, cur_depth):
    if len(lst) == 1 and not isinstance(lst[0], list):
        return lst[0], cur_depth
    elif len(lst) == 1 and isinstance(lst[0], list):
        return min_deep_2_helper(lst[0], cur_depth + 1)
    else:
      if isinstance(lst[0], list):
            first_min, first_depth = min_deep_2_helper(lst[0], cur_depth +
1)
        else:
            first_min = lst[0]
            first_depth = cur_depth
        rest_min, rest_depth = min_deep_2_helper(lst[1:], cur_depth)
            if first_min < rest_min:
                return first_min, first_depth
            else:
                return rest_min, rest_depth

def min_deep_2(lst):
    return min_deep_2_helper(lst, 0)
```

Recursive case:

Finally, to determine the correct depth to return (first_depth vs rest_depth), we compare first_min vs rest_min.

```
# return value is: (min_val, int depth)
>>> min_deep_2([1, -1, 4])
(-1, 0)
>>> min_deep_2([1, [2], [3, [-4]], 9])
(-1, 2)
```

Min_deep() + keeping track of depth

```
def min_deep_2_helper(lst, cur_depth):
    if len(lst) == 1 and not isinstance(lst[0], list):
        return lst[0], cur_depth
    elif len(lst) == 1 and isinstance(lst[0], list):
        return min_deep_2_helper(lst[0], cur_depth + 1)
    else:
      if isinstance(lst[0], list):
            first_min, first_depth = min_deep_2_helper(lst[0], cur_depth +
1)
        else:
            first_min = lst[0]
            first_depth = cur_depth
        rest_min, rest_depth = min_deep_2_helper(lst[1:], cur_depth)
        if first_min < rest_min:
            return first_min, first_depth
        else:
            return rest_min, rest_depth
```

```
def min_deep_2(lst):
    return min_deep_2_helper(lst, 0)
```

Here, we have `min_deep_2()` call our helper function `min_deep_2_helper()`, and initialize the cur_depth to 0 (by definition).

```
# return value is: (min_val, int depth)
>>> min_deep_2([1, -1, 4])
(-1, 0)
>>> min_deep_2([1, [2], [3, [-4]], 9])
(-1, 2)
```

Computational Structures in Data Science

Understanding Order of Execution

Why does it work

One tip: draw out the function call stacks to see how the computation is executed

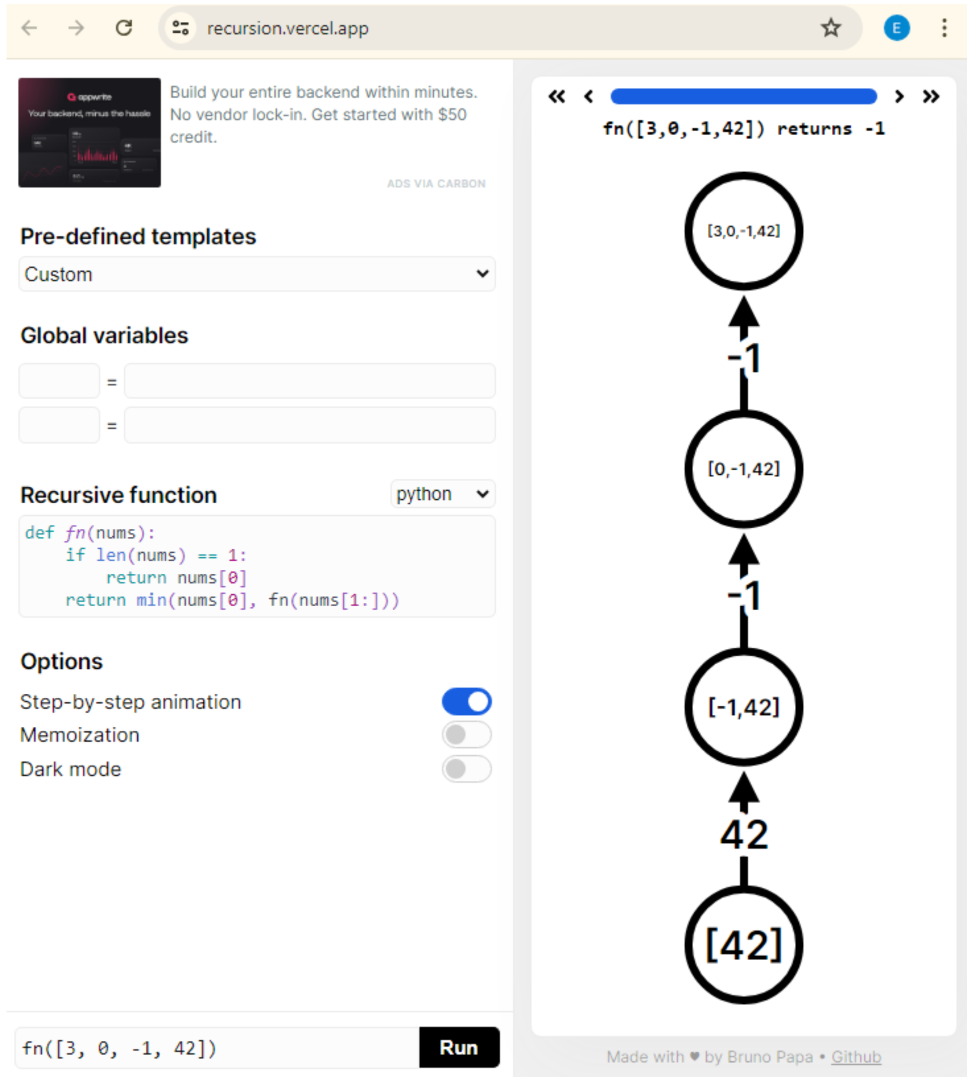
```
sum_of_squares(3)

# sum_of_squares(3) => sum_of_squares(2) + 3**2
#                   => sum_of_squares(1) + 2**2 + 3**2
#                   => sum_of_squares(0) + 1**2 + 2**2 + 3**2
#                   => 0 + 1**2 + 2**2 + 3**2 = 14
```

```
def sum_of_squares(n):
    if n < 1:
        return 0
    else:
        return sum_of_squares(n-1) + n**2
```

Exercise: Return minimum of a list (recursion)

- Demo: Python recursion visualization tools
  - https://recursion.vercel.app/
    - Tip: site is a little finicky, must call your function "fn()"

```python
def fn(nums):
    if len(nums) == 1:
        return nums[0]
    return min(nums[0], fn(nums[1:]))
```


recursion.vercel.app

Build your entire backend within minutes. No vendor lock-in. Get started with $50 credit.

ADS VIA CARBON

Pre-defined templates

Custom

Global variables

Recursive function python

```python
def fn(nums):
    if len(nums) == 1:
        return nums[0]
    return min(nums[0], fn(nums[1:]))
```

Options

Step-by-step animation: on
Memoization: off
Dark mode: off

`fn([3, 0, -1, 42])` Run

`fn([3,0,-1,42]) returns -1`

Made with ♥ by Bruno Papa • Github

Lecture overview. Any questions?

- More recursion practice
  - Recursion over sequences
    - "Deep" recursion