

Computational Structures in Data Science

Recursion

Week 4, Summer 2024. 7/8 (Mon)

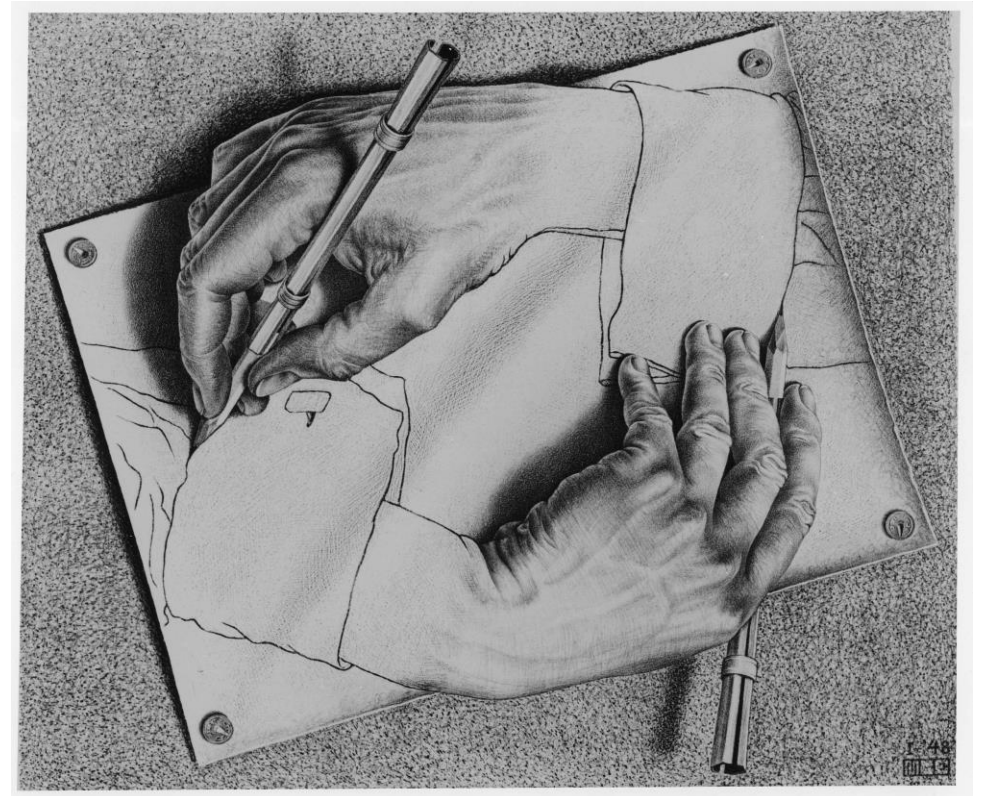
Lecture 11

Berkeley
UNIVERSITY OF CALIFORNIA

Recursion



M. C. Escher : Drawing Hands



Announcements

- HW04, Lab04 due today (11:59 PM PST)
- “(Urgent) Midterm Exam Scheduling” due **TOMORROW** (7/9 11:59 PM PST)
- HW05, Lab05 due 7/10 (Wed)
- Project01 (“Maps”) is out!
 - Checkpoint: due 7/10 (Wed)
 - Full project: due 7/18
- New weekly course survey out in Gradescope (optional, +0.5 extra credit pts)

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

- Recursion
- Recursion in Python
 - Iteration (for/while) vs recursion
- Identifying recursive structure of problems

(Optional) vee / Fractals

- `python3 -i 11-Recursion.py`
- This uses Turtle Graphics.
 - The turtle module is really cool, but **not** something you need to learn
- **vee** is the one recursive problem that doesn't have a base case
 - But fractals in general are a fun way to visualize self-similar structures
- Use the following keys to play with the demo
 - Space to draw
 - C to Clear
 - Up to add "vee" to the functions list
 - Down to remove the "vee" functions from the list.
- [Some cool variations on vee, seen in Snap! \(the language of CS10\)](#)
- [More Fractals](#)

Recursion: a broad definition

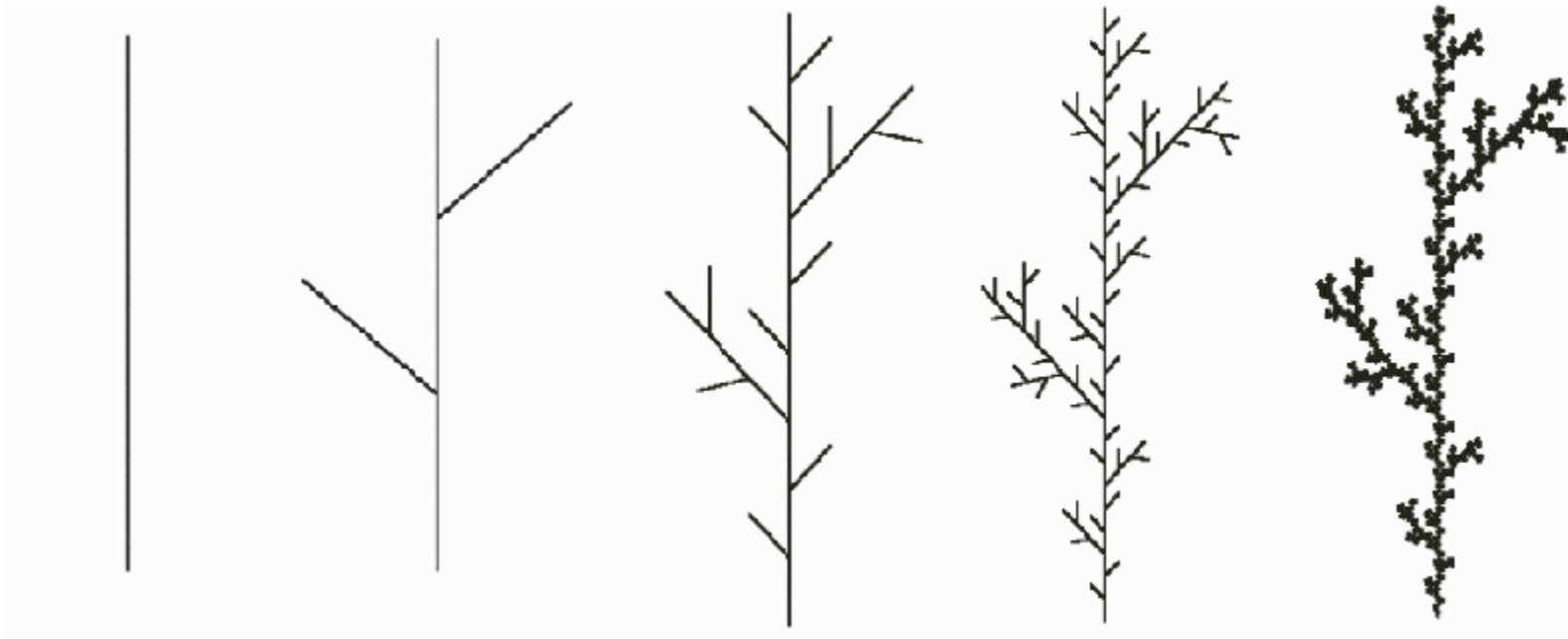
- **Recursion** occurs when the definition of a concept or process **depends on a simpler or previous version of itself**. (from: [Wikipedia](#))



A visual form of recursion known as the [Droste effect](#). The woman in this image holds an object that contains a smaller image of her holding an identical object, which in turn contains a smaller image of herself holding an identical object, and so forth. 1904 Droste [cocoa](#) tin, designed by Jan Misset

Recursion application: plants/trees

- Fun application: artificial plants/trees built via recursively-defined rules



Source:
<http://guyhaas.com/bfoit/itp/RecursionInNature/RecursionInNature.html>

(Step 1) Start with a line segment.

(Step 2) Replace the line segment with 5 line segments as pictured, each $\frac{1}{3}$ the length of the original.

(Step N) Replace each segment in step $n-1$ with a reduced copy of the step $n-1$ figure.

Recursion application: plants/trees

[Figure 6](#) shows the compounding of some of the inflorescences. These pictures were all done with simple recursion.

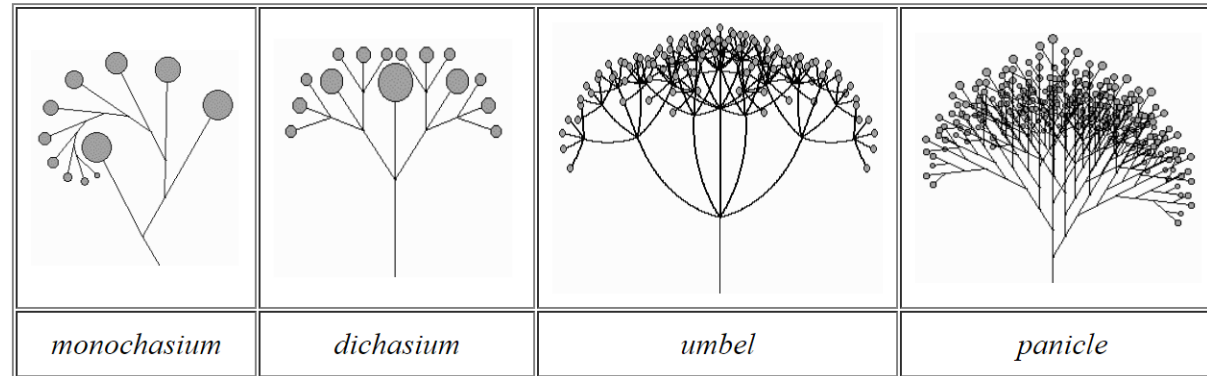


Figure 6: *Compound inflorescences*

[Figure 7](#) shows some imaginary inflorescences obtained by using random numbers to vary segment lengths and angles and taking artistic liberties with the above.

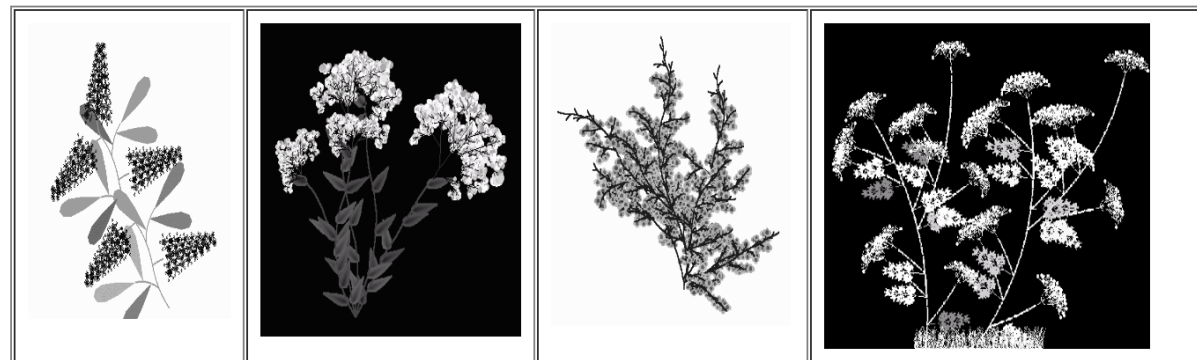


Figure 7: *Imaginary inflorescences*

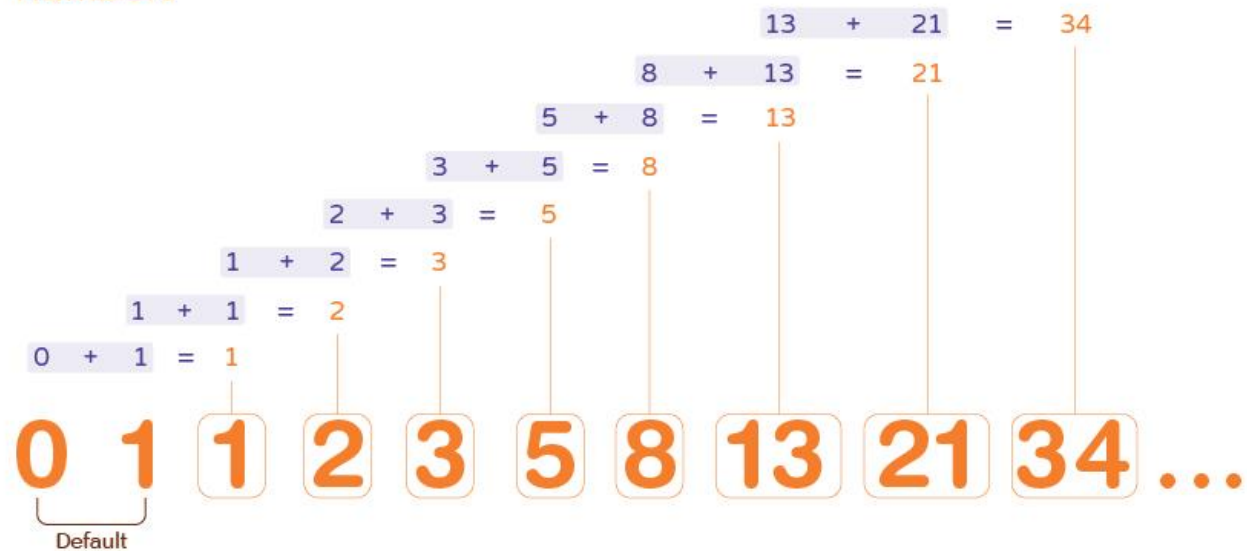
Recursion in Computer Science / Math

- Sometimes, a problem or process is easiest to describe via a recursive definition.
- Fibonacci sequence: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...
 - Recursive definition: $\text{Fibonacci}(n) = \text{Fibonacci}(n - 1) + \text{Fibonacci}(n - 2)$
-

Fibonacci Sequence

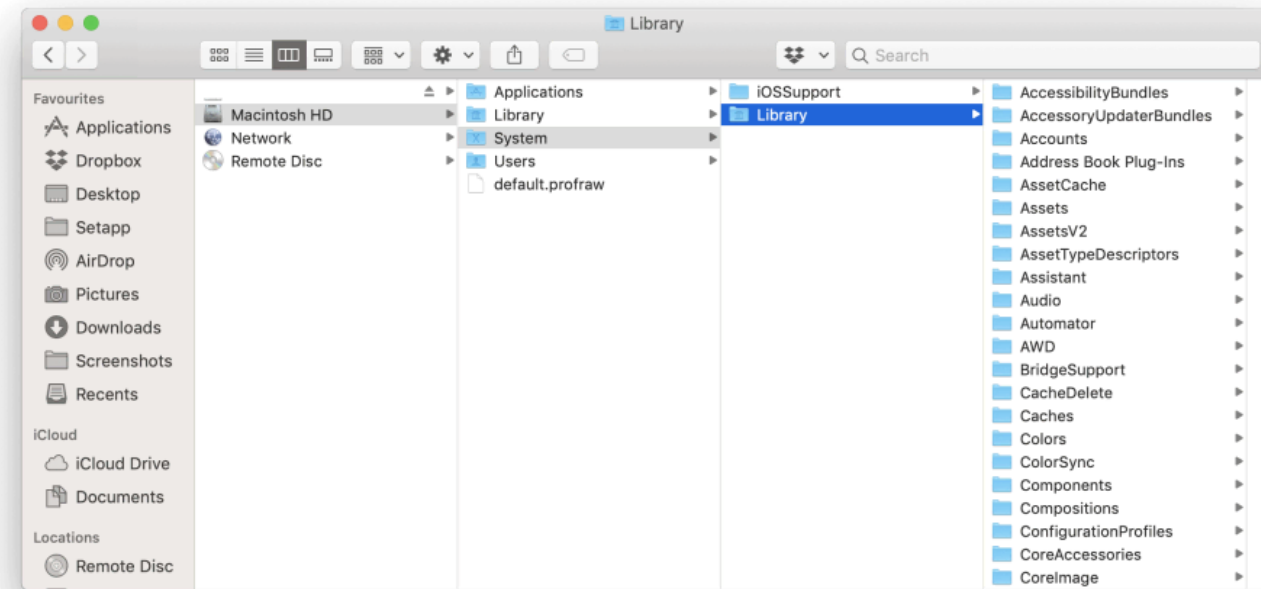
Numbers

MATH
MONKS



Recursion in Computer Science / Math

- Searching for a file in a file system
 - To find a file: starting from the current directory, examine the files in the current directory.
 - If there is another directory, then (recursively) search for the file in that subdirectory.
 - In that subdirectory: if there is a subdirectory, search for the file in that subdirectory
 - ...and so on...



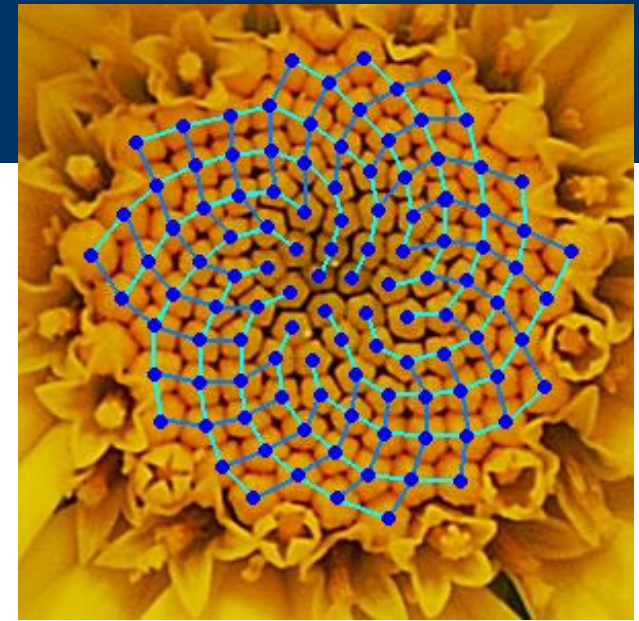
Source: <https://mac-optimization.bestreviews.net/how-to-restore-system-files-on-macos/>

A Folder contains:

- Files
 - Folders
- ← Recursion!

Recursion In Practice

- Key idea: A recursive function operates by solving smaller sub-problems
- “Smaller sub-problems” -> recursive function calls
- Compared to a for-loop, while loop, we will *not* directly specify how many times we need to make a function call.



Toy Example: Countdown

A recursive function calls itself in its body:

```
def countdown(n):  
    if n == 0:  
        print('Blastoff!')  
    else:  
        print(n)  
        countdown(n - 1)
```



Recursive call!

```
>>> countdown(5)  
5  
4  
3  
2  
1  
Blastoff
```


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 current 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 current problem.

Computational Structures in Data Science

Recursion



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

Example: Palindromes

- **Definition:** Palindromes are the same word forwards and backwards.
- Examples:
 - C88C
 - racecar
 - LOL
 - a man a plan a canal panama
 - aibohphobia 🐈
 - The fear of palindromes.
 - <https://czechtheworld.com/best-palindromes/#palindrome-words>



“Bob” – Weird Al Yankovic
(Lyrics contain only palindromes!)
“I, man, am Regal, a German am I
Never odd or even
If I had a Hi-Fi
Madam, I'm Adam
Too hot to hoot”

Example: Palindromes

- Palindromes are the same word forwards and backwards.
- How to define an ``is_palindrome(word)`` function?
- **One way:** check if the reversed string is the same as the input string

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

```
>>> is_palindrome("too hot to hoot")
```

```
True
```

```
>>> is_palindrome("meow")
```

```
False
```

Example: Reverse (iteratively)

- **Question**: How to define `reverse_str()` via iteration (for/while)?
- Notably, let's not use the `[::-1]` shortcut. For practice, let's implement it once with a for loop, and another with a while loop.

```
def reverse_str_for(input_str):  
    # FILL ME IN
```

```
def reverse_str_while(input_str):  
    """  
    >>> reverse_str_while('hello')  
    'olleh'  
    """  
    # FILL ME IN
```

Example: Reverse (iteratively)

- **Question:** How to define `reverse_str()` via iteration (for/while)?
- Notably, let's not use the `[::-1]` shortcut. For practice, let's implement it once with a for loop, and another with a while loop.

```
def reverse_str_for(s):  
    result = ''  
    for letter in s:  
        result = letter + result  
    return result
```

Note: there are many other ways to do this. For instance, in the while-loop implementation we could have iterated backwards over the string via indexing.

```
def reverse_str_while(s):  
    """  
    >>> reverse_str_while('hello')  
    'olleh'  
    """  
    result = ''  
    while s:  
        first = s[0]  
        s = s[1:]  
        result = first + result  
    return result
```

Example: Reverse (recursively)

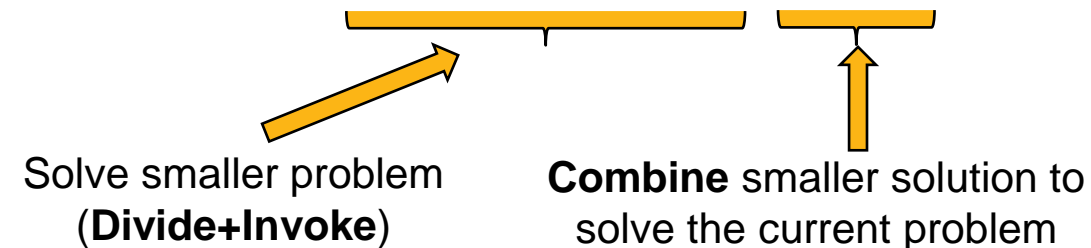
- **Question**: How to define `reverse_str()` as a recursive process in the following manner?

- **Base case(s)**, the problem is simple enough to be solved directly
- **Recursive case(s)**. A recursive case has three components:
 - **Divide** the current 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 current problem.

"hello" -> "olleh"

(Hint) Recursive structure of problem:
If I reversed the substring "ello" -> "olle", how do I use this partial result "olle" to solve my current problem of reversing "hello"?

Answer: "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]`



Example: Reverse (recursively)

- **Question**: How to define `reverse_str()` as a recursive process in the following manner?

- **Base case(s)**, the problem is simple enough to be solved directly
- **Recursive case(s)**. A recursive case has three components:
 - **Divide** the current 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 current problem.

"hello" -> "olleh"

(Hint) Base cases

What are the simplest ("trivial") strings to reverse?

Empty string, string with one character

`reverse_str("") -> ""`

`reverse_str("o") -> "o"`

Example: Reverse (recursively)

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

- **Base case(s)**, the problem is simple enough to be solved directly
- **Recursive case(s)**. A recursive case has three components:
 - **Divide** the current 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 current problem.

"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]`

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  
    # FILL ME IN  
    # Recursive cases  
    # FILL ME IN
```

Question: how to fill in the base cases?

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  
    # FILL ME IN
```

Question: how to fill in the recursive cases?

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]
```

How does the recursion work?

- Our algorithm in words:
 - Take the first letter, put it at the end
 - The beginning of the string is the reverse of the rest.

`reverse('ABC')`

→ `reverse('BC') + 'A'`

→ `reverse('C') + 'B' + 'A'`

→ `'C' + 'B' + 'A'`

→ `'CBA'`

```
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]
```

Iteration vs Recursion: Sum Numbers

Iteratively
(For loop):

```
def sum_for(n):  
    s = 0  
    for i in range(0, n + 1):  
        s = s + i  
    return s
```

Iteratively
(While loop):

```
def sum_while(n):  
    s = 0  
    i = 0  
    while i < n:  
        i = i + 1  
        s = s + i  
    return s
```

Recursively

```
def sum_recurse(n):  
    if n == 0:  
        return 0  
    return n + sum_recurse(n - 1)
```

Aside: closed form solutions (Math)

Sometimes, a recursive math function will have a closed form solution!

In this class: even if you know the closed-form solution, and we ask you to implement it recursively: don't just use the closed-form solution 😊

```
def sum_recurse(n):  
    if n == 0:  
        return 0  
    return n + sum_recurse(n - 1)
```

```
def sum_closed_form(n):  
    return (n * (n + 1)) / 2
```

```
def fib(n):  
    if n <= 1:  
        return n  
    return fib(n - 1) + fib(n - 2)
```

```
g = (1 + 5**.5) / 2 # Golden ratio.  
def fib_closed_form(N):  
    return int((g**N - (1-g)**N) / 5**.5)
```

For the curious (beware, deep math!):
https://en.wikipedia.org/wiki/Fibonacci_sequence#Closed-form_expression

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.

In Python, how does it work under the hood?

- Each recursive call gets its own local variables
 - Just like any other function call
- Computes its result (possibly using additional calls)
 - Just like any other function call
- Returns its result and returns control to its caller
 - Just like any other function call
- The function that is called happens to be itself
 - Called on a simpler problem
 - Eventually stops on the simple base case

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

indexing an element of a sequence

Slicing a sequence of elements

- Recursion over sequence length

Recall: Iteration

```
def sum_of_squares(n):  
    accum = 0  
    for i in range(1, n+1):  
        accum = accum + i*i  
    return accum
```

1. Initialize the "base" case of no iterations

2. Starting value

3. Ending value

4. New loop variable value

Recursion Key concepts – by example

1. Test for simple "base" case

2. Solution in simple "base" case

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

3. Assume recursive solution to simpler problem

4. "Combine" the simpler part of the solution, with the recursive case

In words

- The sum of no numbers is zero
- The sum of 1^2 through n^2 is the
 - sum of 1^2 through $(n-1)^2$
 - plus n^2

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

Why does it work

```
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
```

Questions

- **Question:** for each, in what order do we sum the squares ?

```
def sum_of_squares(n):  
    accum = 0  
    for i in range(1,n+1):  
        accum = accum + i**2  
    return accum
```

```
accum = 0  
accum += 1**2  
accum += 2**2  
...  
accum += n**2
```

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

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

Output = (((((0) + 1**2) + 2**2) + ...) + n**2

Output = n**2 + ((n-1)**2 + ((n-2)**2 + ...) + 0)))

Hint: Python always evaluates left-to-right

Trust ...

- The recursive “leap of faith” works as long as we hit the base case eventually
- What happens if we don't?
 - “Infinite Loop”



Example: infinite loop (recursion)

- This code will never finish

```
def sum_recurse_inf_loop(n):  
    # base case  
    if n == 0:  
        return 0  
    # BUG: I forgot to do (n-1). Infinite loop!  
    return n + sum_recurse_inf_loop(n)
```

Aside: By default, Python has a maximum limit on how much recursive “depth” is allowed before it terminates the program.

If Python didn’t have this max limit, then our code would have run forever!

Or, to be more precise: eventually our CPU would run out of memory and crash. This is because function frames use up memory. Infinite call frames => infinite required memory => out-of-memory error.

To adjust max recursive depth limit, see: [sys.setrecursionlimit\(\)](#)



```
>>> sum_recurse_inf_loop(3)  
Traceback (most recent call last):  
  File  
    "C:\Users\Eric\c88c\lectures\lecture11\lecture11.py",  
    line 96, in <module>  
      print(sum_recurse_inf_loop(3))  
  File  
    "C:\Users\Eric\c88c\lectures\lecture11\lecture11.py",  
    line 94, in sum_recurse_inf_loop  
      return n + sum_recurse_inf_loop(n)  
...  
[Previous line repeated 995 more times]  
File  
  "C:\Users\Eric\c88c\lectures\lecture11\lecture11.py",  
  line 91, in sum_recurse_inf_loop  
    if n == 0:
```

RecursionError: maximum recursion depth exceeded in comparison

Why Recursion?

- “After Abstraction, Recursion is probably the 2nd biggest idea in this course”
- “It’s tremendously useful when the problem is self-similar”
- “It’s no more powerful than iteration, but often leads to more concise & better code”
- “It’s more ‘mathematical’”
- “It embodies the beauty and joy of computing”
- ...

(Aside) Tips for success

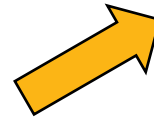
- In my view, this class has suddenly increased in scope/difficulty.
- **Litmus test:** at this point, we ideally expect you to be fairly fluent with Python:
 - Notably, be comfortable reading Python, writing Python from scratch, modifying existing Python
- It's OK if you're not there yet, but do keep this as your "north star".
- **Programming is a craft you can only learn by doing** (and not only by watching lectures or reading books/slides/etc).

(Aside) Tips for success

- **Warning:** if you're struggling with basic Python syntax, **you are at risk of falling behind**, as the rest of the course will continue rapidly building on top of Python.
- Tip: If you're feeling weak here, I'd prioritize getting comfortable with Python first before embarking on more complicated topics like recursion.
 - if you'd like, reach out on Ed for ideas on how to catch up!

Litmus test: at this point, we expect you to be able to define this function on your own. Considered an “easy” problem.

Tip: If this doesn't feel “easy” yet, practice until it does!



```
# Write a function that, given a list
# of numbers, returns a new list with
# every number squared. Use a for-loop.
def square_nums(nums):
    """
    >>> square_nums([2, 4, 5, 10])
    [4, 16, 25, 100]
    """
    out = []
    for num in nums:
        out.append(num ** 2)
    return out
```

(Aside) Tips for success. Any questions?

- Tip for success: **deliberate practice**, and **being organized**.
- **“Deliberate practice”**: targeted practice on topics you know you aren’t comfortable with.
- It’s not time-efficient to practice things you are already comfortable with.
 - Though, any practice is better than no practice. But, we don’t have infinite time/energy, so best use it strategically
- Ex: “I know I’m not comfortable with recursion. Let me spend an hour doing recursion practice problems / asking my TA (or Ed) about this tricky recursion problem, etc.
- **“Being organized”**: a surprising amount of success in this course (and, college/life in general) is keeping up with assignment due dates. Establish good organizational habits early!
- “Meta learning”: aka “learning how to learn effectively”. Very important life skill.
- At the end of the day, we (the teaching staff) want you to succeed!