CS134:
More Recursion

Join Women in Computer Science + UnICS for a

Gourd Decorating Party

Who: All Computer Scientists!
What: Gourd Decorating + Snacks!
Where: CS Common Room (TCL 3rd floor)
When: Thursday 10/24, 3:30-4:30pm

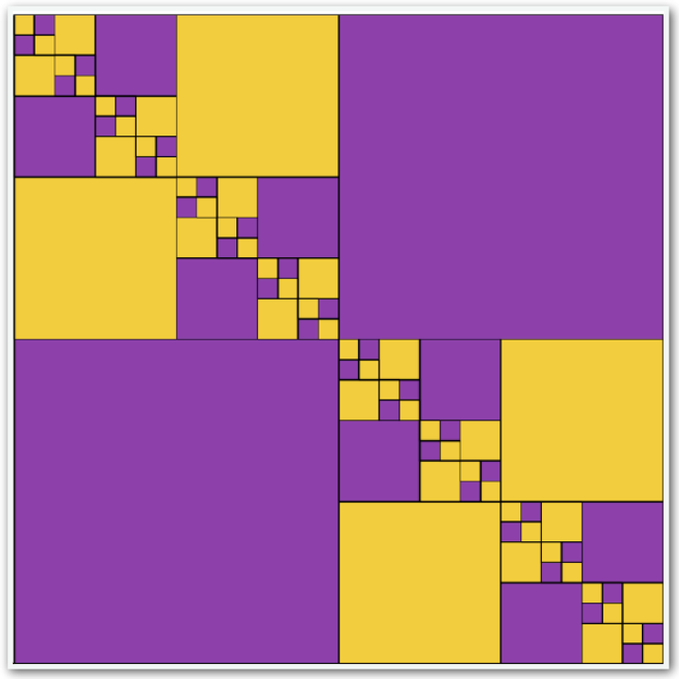
Announcements & Logistics

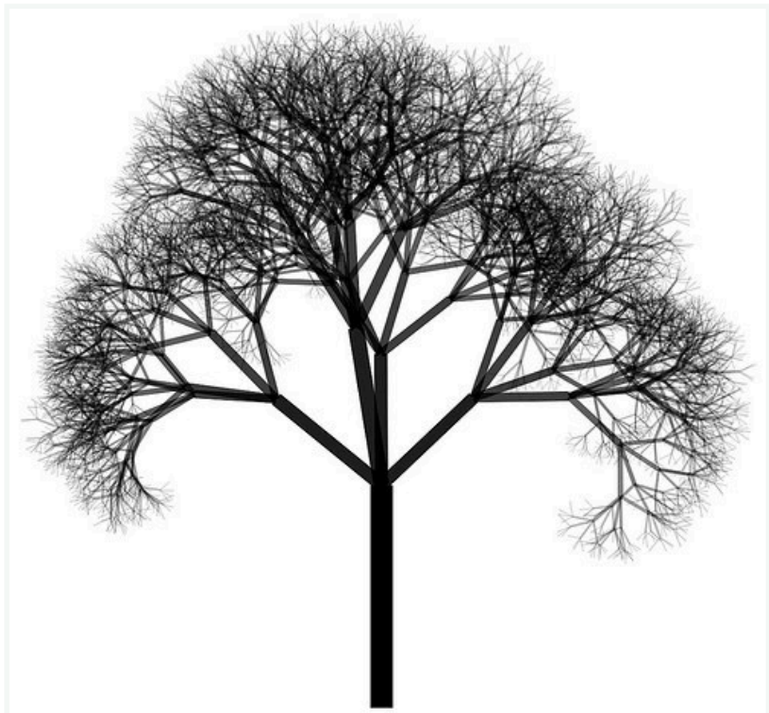
- Lab 6 (Dictionaries) Due today/tomorrow
  - WATCH THE LECTURE VIDEO ON DICTIONARIES
- **HW 6** on Gradescope will be released today
- Lab 7, 8, and 9 are **partner labs**
  - Pair programming is an important skill as well as a vehicle for learning
  - Fill-out the **partner form** Lida emailed Monday
    - Due THURSDAY/TOMORROW

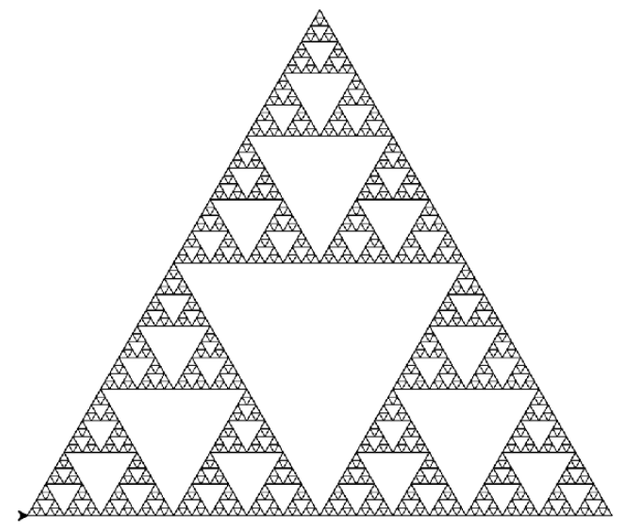
Do You Have Any Questions?

Last Time

- Introduction to recursion
  - Alternative to iteration
  - New problem solving paradigm
- Function frame model to understand recursion behind the scenes







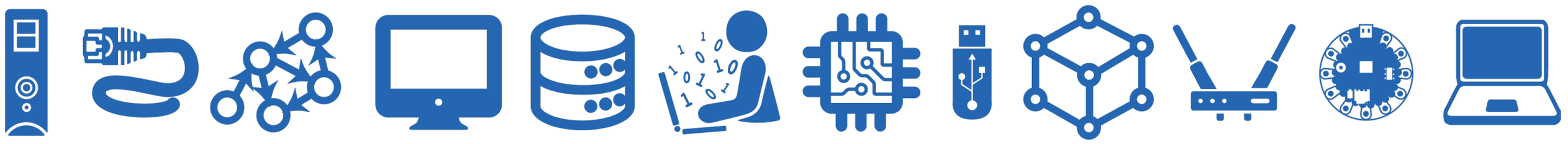
Last Time: Recursive Approach to Problem Solving

- A recursive function is a function **that calls itself**
- A recursive approach to problem solving has two main parts:
  - **Base case(s).** When the problem is **so small**, we solve it directly, without having to reduce it any further
  - **Recursive step.** Does the following things:
    - Performs an action that contributes to the solution
    - **Reduces** the problem to a smaller version of the same problem, and calls the function on this **smaller subproblem**
- The recursive step is a form of "wishful thinking" (also called the inductive hypothesis)

Today's Plan

- [More] practice translating recursive ideas into recursive programs
- Examining the relationship between **recursive** and **iterative** programs
  - That is, how do recursive ideas relate to the iterative ideas (for loops, while loops) we've covered so far?

More Recursion: `count_up`



count_up(n)

- Write a recursive function that prints integers from `1` up to `n`
- Recursive definition of count_up:
  - Base case: `n == 1, print(n) # last time!`
  - Recursive rule: `call count_up(n-1), print(n)`

```
>>> count_up(5)
1
2
3
4
5
```

```
>>> count_up(4)

1
2
3
4
```

```
>>> count_up(3)

1
2
3
```

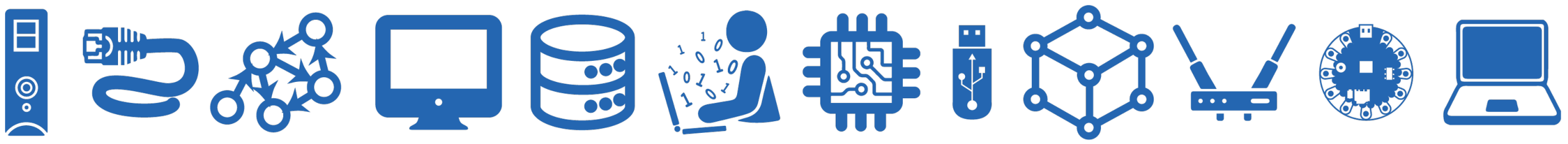
`count_up(n)`

- Unlike `count_down(n)` the print statement is **after** the recursive function call (*why*?)
- By printing **after** the recursive call, the print statement gets executed "on the way back" from recursive calls

```
>>> count_up(5)
1
2
3
4
5
```

```
def count_up(n):
    '''Prints out integers from 1 up to n'''
    if n == 1:
        print(n)
    else:
        count_up(n-1)
        print(n)
```

Function Frame Model to Understand `count_up`



count_up(4)

n 4

```
if n == 1:
    print(n)
else:
    count_up(n-1)
    print(n)
```

count_up(3)

n 3

```
if n == 1:
    print(n)
else:
    count_up(n-1)
    print(n)
```

count_up(2)

n 2

```
if n == 1:
    print(n)
else:
    count_up(n-1)
    print(n)
```

Base case reached!

count_up(1)

n 1

```
if n == 1:
    print(n)
else:
    count_up(n-1)
    print(n)
```

```
>>> count_up(4)
```

count_up(4)

```
n 4

if n == 1:
    print(n)
else:
    count_up(n-1)
→   print(n)
```

count_up(3)

```
n 3

if n == 1:
    print(n)
else:
    count_up(n-1)
→   print(n)
```

count_up(2)

```
n 2

if n == 1:
    print(n)
else:
    count_up(n-1)
→   print(n)
```

Base case reached!

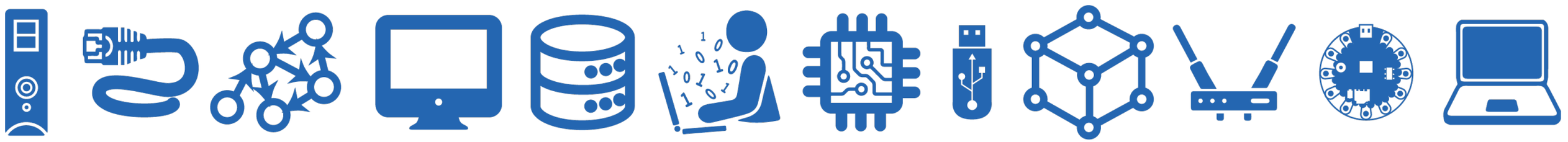
count_up(1)

```
n 1

if n == 1:
→   print(n)
else:
    count_up(n-1)
    print(n)
```

```
>>> count_up(4)
1
2
3
4
```

Recursion GOTCHAs!



GOTCHA #1

- If the problem that you are solving recursively **is not getting smaller**, that is, you are not getting closer to the base case --- **infinite recursion**!
- Never reaches the base case

```
def count_down_gotcha(n):
    '''Prints ints from 1 up to n’''
    if n == 1:  # Base case
        print(n)
    else:       # Recursive case
        print(n)
        count_down_gotcha(n)
```

Subproblem not getting smaller!

GOTCHA #2

- Missing base case/unreachable base case--- another way to cause **infinite recursion**!

```
def print_halves_gotcha(n):
    """Prints n, n/2, down to ... 1"""
    if n == 0:
        print(n)
    else:
        print(n)
        return print_halves_gotcha(n/2)
```

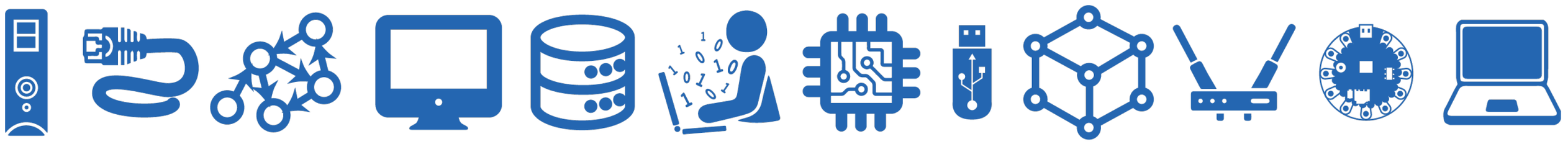
Never true!

"Maximum recursion depth exceeded"

- In practice, the infinite recursion examples will terminate when Python runs out of resources for creating function call frames, leads to a "maximum recursion depth exceeded" error message

Recursion vs. Iteration:

`sum_list`



sum_list

- **Goal:** Write a function to sum up a list of numbers
- Iterative approach? (i.e., using loops?)

Iterative Approach to `sum_list`

- **Goal:** Write a function to sum up a list of numbers
- Iterative approach:

```
def sum_list_iterative(num_lst):
    sum = 0
    for num in num_lst:
        sum += num
    return sum
```

```
>>> sum_list_iterative([3, 4, 20, 12, 2, 20])
61
```

sum_list

- **Goal:** Write a function to sum up a list of numbers
- Recursive approach?

Recursive approach to `sum_list`

- **Base case:**
  - `num_lst` is empty, return `0`
- **Recursive rule:**
  - Return first element of `num_lst` plus result from calling `sum_list` on rest of the elements of the list.

- Example: Suppose `num_lst = [6, 3, 6, 5]`
  - `sum_list([6, 3, 6, 5]) = 6 + sum_list([3, 6, 5])`
  - `sum_list([3, 6, 5]) = 3 + sum_list([6, 5])`
  - `sum_list([6, 5]) = 6 + sum_list([5])`
  - `sum_list([5]) = 5 + sum_list([])`
- For the base case we have `sum_list([])` returns `0`

Recursive approach to `sum_list`

- **Base case:**
  - `num_lst` is empty, return `0`
- **Recursive rule:**
  - Return first element of `num_lst` plus result from calling `sum_list` on rest of the elements of the list.

- Example: Suppose `num_lst = [6, 3, 6, 5]`
  - `sum_20([6, 3, 6, 5]) = 6 + sum_14([3, 6, 5])`
  - `sum_14([3, 6, 5]) = 3 + su11st([6, 5])`
  - `sum_11([6, 5]) = 6 + s5ist([5])`
  - `sum_5([5]) = 5 + 0_list([])`
- For the base case we have `sum_list([])` returns `0`

Recursive approach to `sum_list`

```
def sum_list(num_lst):
    """Returns sum of given list"""
    if not num_lst:
        return 0
    else:
        return num_lst[0] + sum_list(num_lst[1:])
```

```
>>> sum_list([3, 4, 20, 12, 2, 20])
61
```

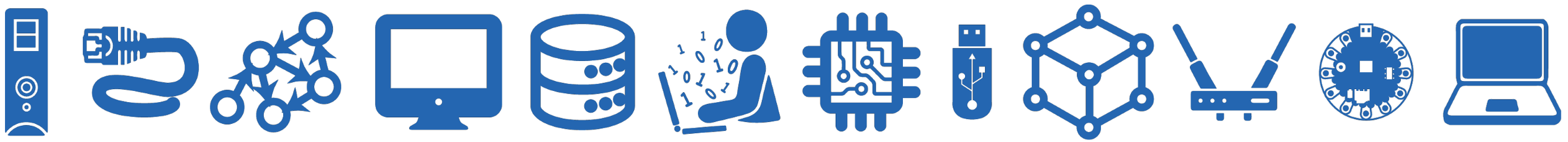
Compare `sum_list` approaches

- Compare/Contrast:

```
def sum_list_iterative(num_lst):
    sum = 0
    for num in num_lst:
        sum += num
    return sum
```

```
def sum_list(num_lst):
    if num_lst == []:
        return 0
    else:
        return num_lst[0] + sum_list(num_lst[1:])
```

Why Recursion?

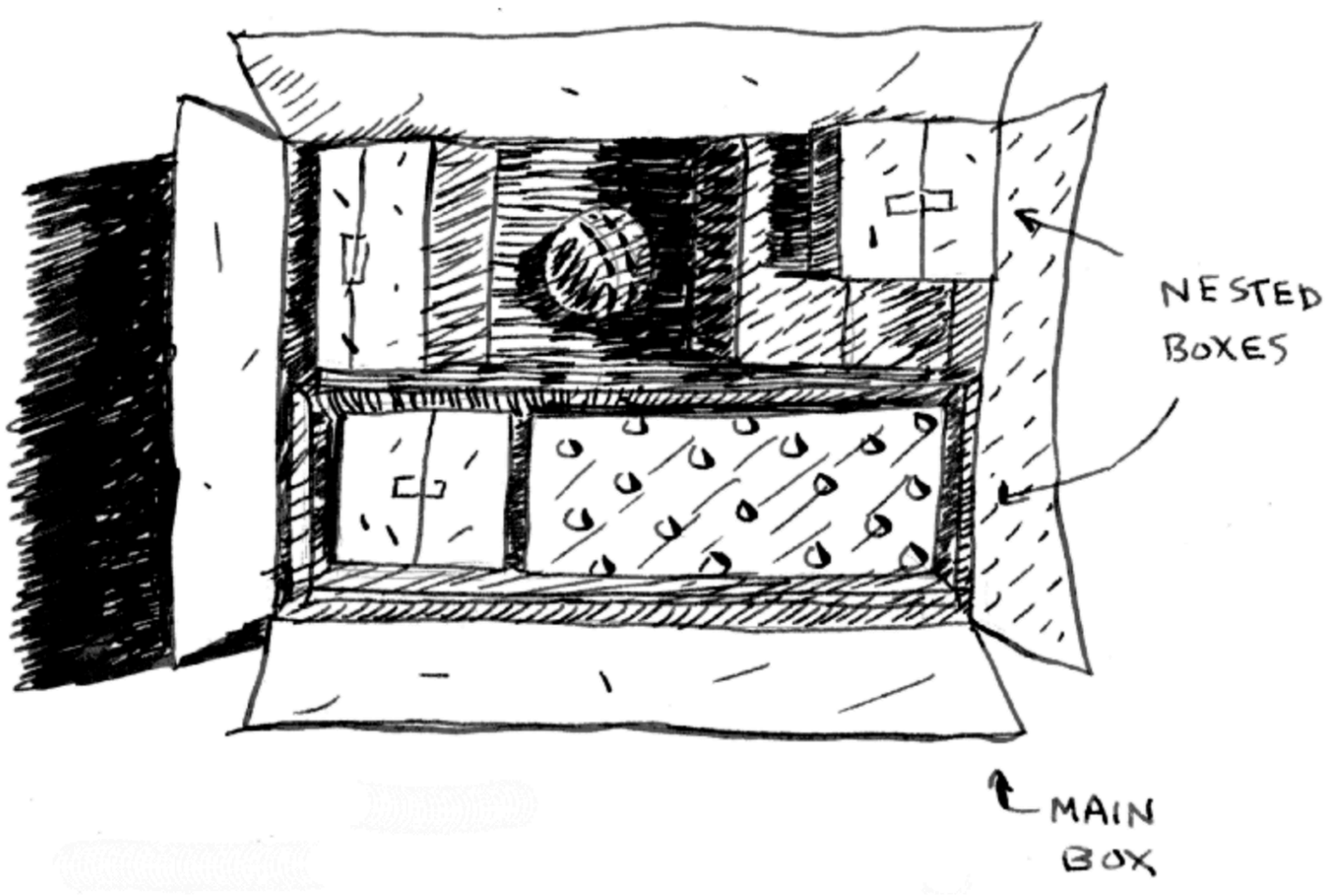


What's The Big Deal With Recursion?

- Why choose recursion over iteration?
- The recursive solution can be more elegant, resulting in fewer lines of code
- Fewer lines of code often correlates with less debugging!
- Let's consider a simple real world example

A Simple Real World Task

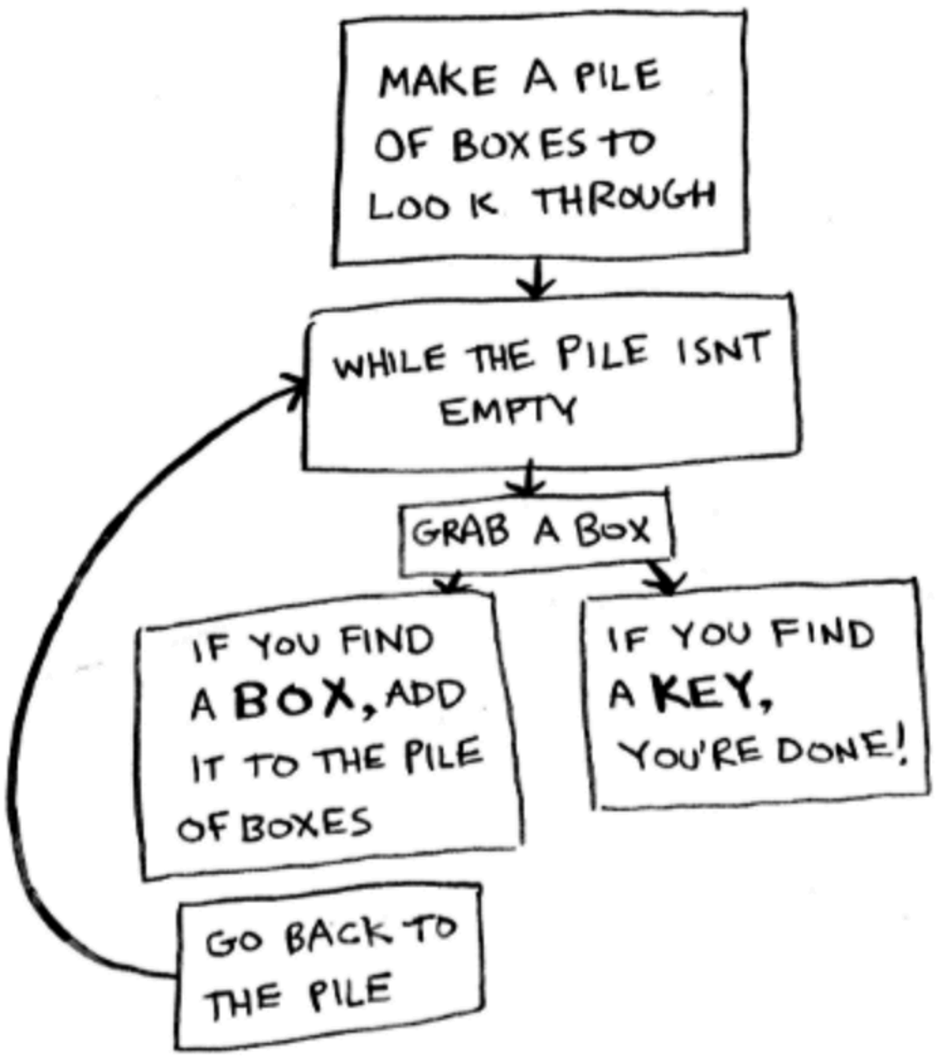
- Consider trying to find a key that is lost in a pile of boxes within boxes.
- (This task is quite similar to trying to find a file on your computer!)



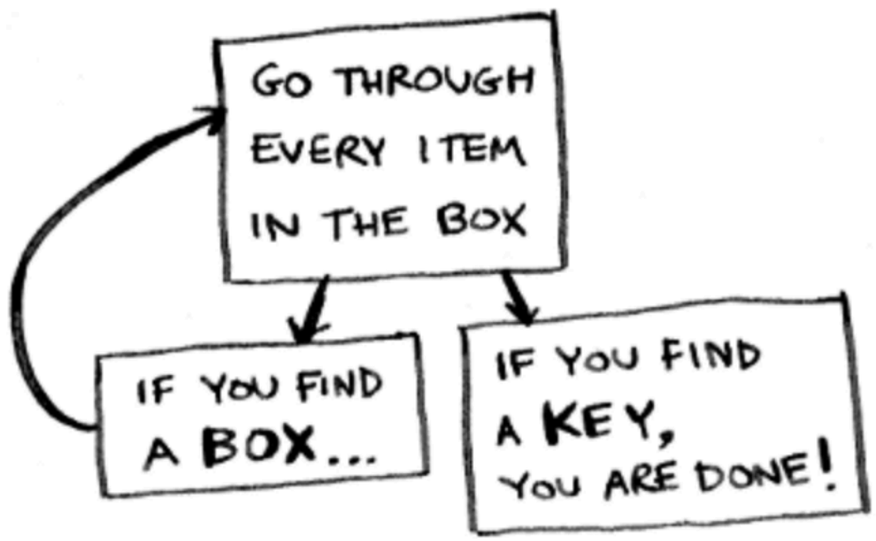
Credit to Aditya Bhargava for the nice illustrations

Comparing Approaches To Finding The Key

- In this case, it's much easier to describe the algorithm using a recursive approach

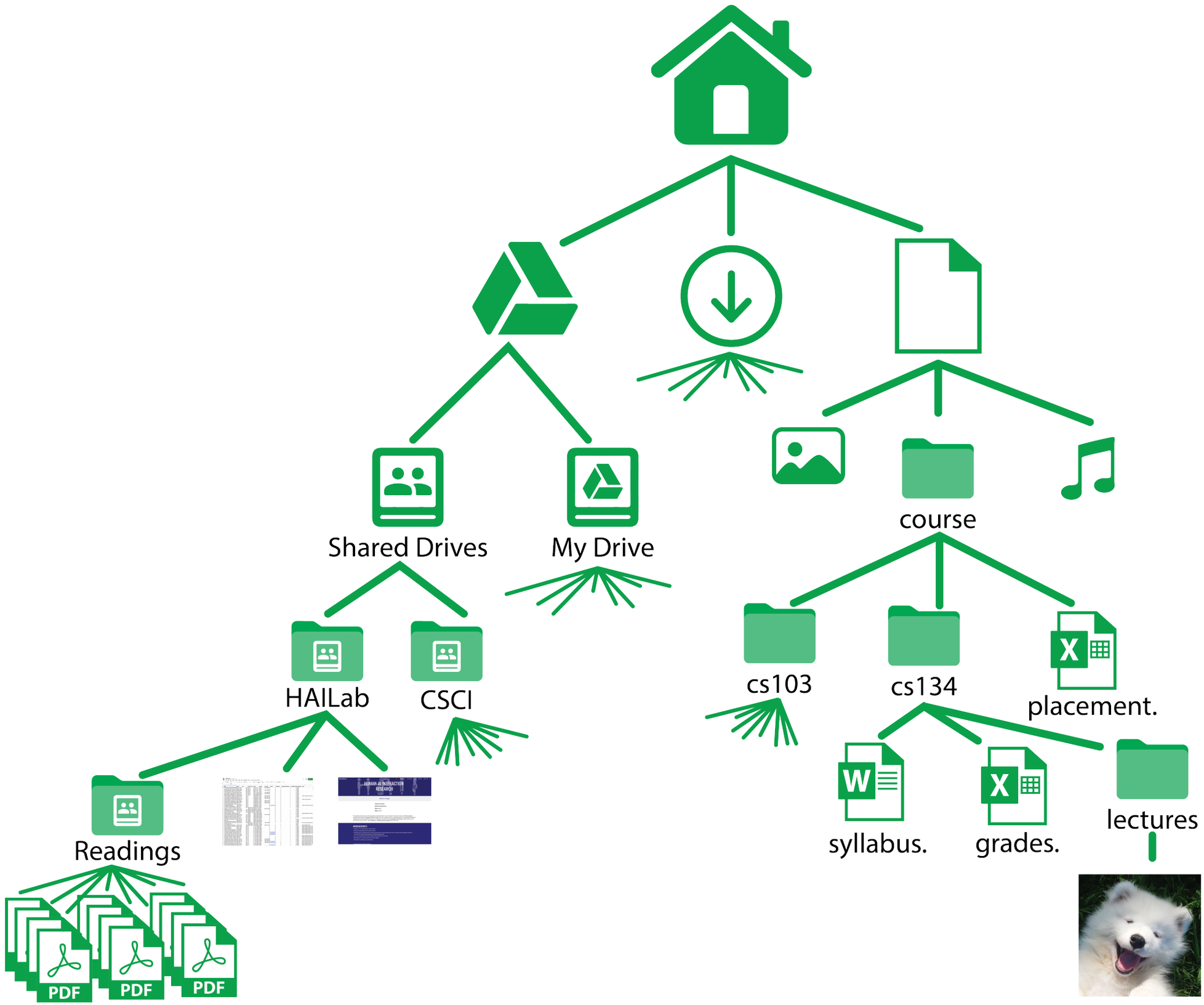


Iterative Approach

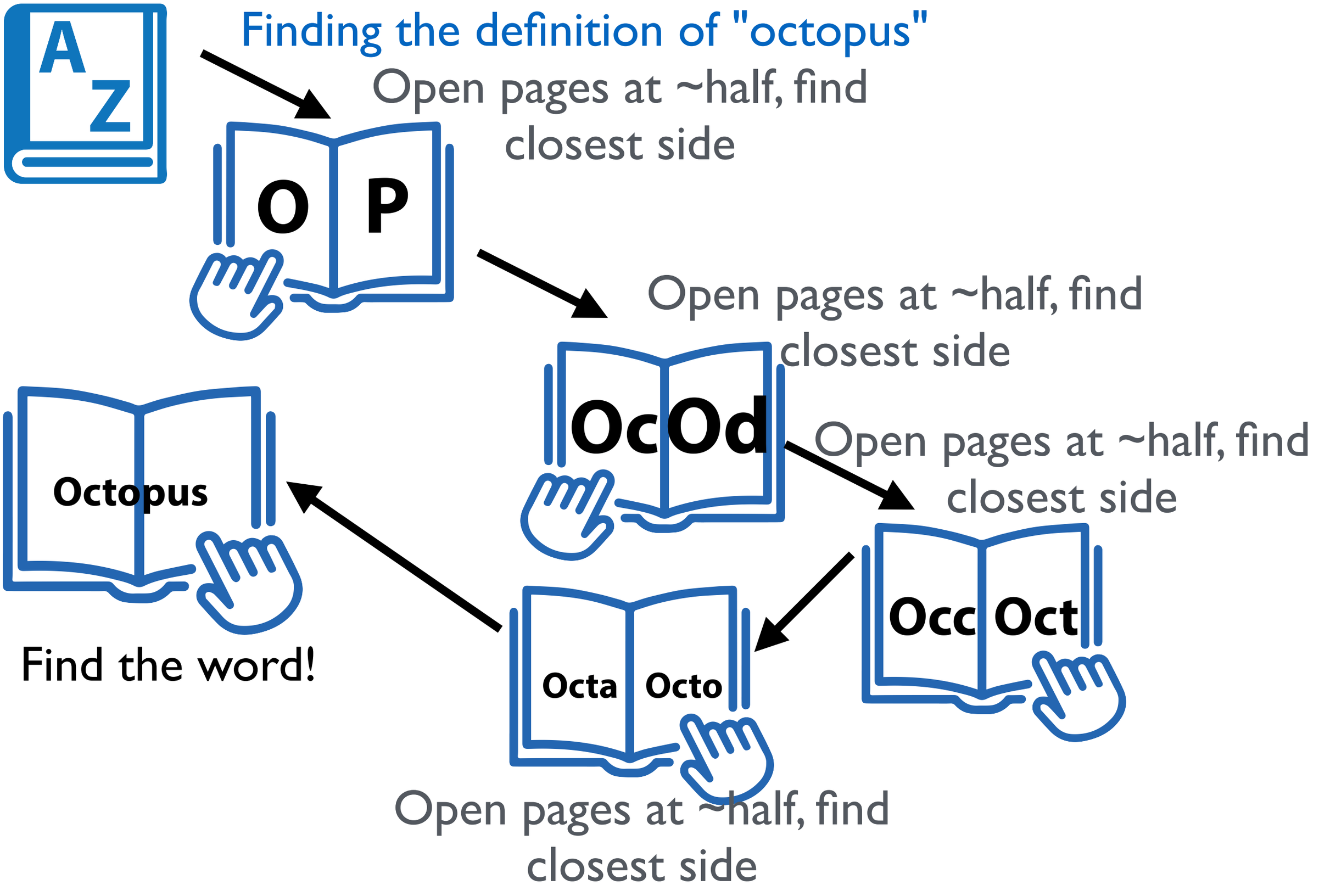


Recursive Approach

Similar: Searching For A File On Our Computer



Similar: Finding a Word in a Dictionary



Finding a File: Iterative

- Compare/Contrast:

```
def file_found_iterative(files, target):
    """
    Iterative function that returns True if the list of files
    contains the target file and False otherwise.
    """
    for file in files:
        if file == target:
            return True
    return False
```

```
>>> files = ["homework", "puppy", "films"]
>>> file_found_iterative(files, "puppy")
True
>>> file_found_iterative(files, "kitten")
False
```

Finding a File: Recursive

- Compare/Contrast:

```
def file_found(files, target):
    """
    Recursive function that returns True if the list of files
    contains the target file and False otherwise.
    """
    if files == []:
        return False
    else:
        return files[0] == target or file_found(files[1:], target)
```

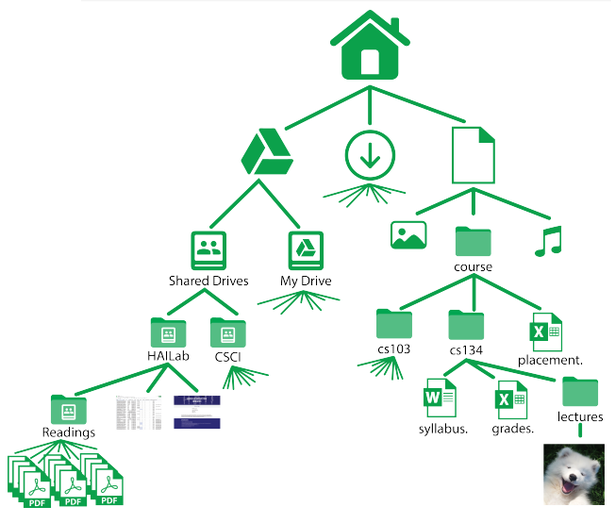
```
>>> files = ["homework", "puppy", "films"]
>>> file_found(files, "puppy")
True
>>> file_found(files, "kitten")
False
```

Compare `file_found` approaches

- Compare/Contrast:

```
def file_found_iterative(files, target):
    for file in files:
        if file == target:
            return True
    return False
```

```
def file_found(files, target):
    if files == []:
        return False
    else:
        return files[0] == target or file_found(files[1:], target)
```



But what if we have folders in folders in folders?

Finding a File in Nested Structures: Iterative

```
def file_found_iterative(folder, target):
    # create an initial list or pile of files/folders to look through
    pile = []
    for item in folder:
        pile+=item
    # keep looking while the pile isn't empty
    while len(pile) > 0:

        # get and remove the last item from the pile
        item = pile[-1]
        pile = pile[:-1]

        # if the item is a folder (list) add each item
        # inside the folder onto the pile
        if type(item) == list:
            for obj in item:
                pile += obj

        # otherwise check if the current file is our target
        elif item == target:
            return True

    return False
```

Finding a File in Nested Structures: Iterative

```
def file_found_iterative(folder, target):
    # create an initial list or pile of files/folders to look through
    pile = []
    for item in folder:
        pile+=item
    # keep looking while the pile isn't empty
    while len(pile) > 0:
        # get and remove the last item from the pile
        item = pile[-1]
        pile = pile[:-1]
        # if the item is a folder (list) add each item
        # inside the folder onto the pile
        if type(item) == list:
            for obj in item:
                pile += obj
        # otherwise check if the current file is our target
        elif item == target:
            return True
    return False
```

```
>>> nested_folders = ["homework", ["cat", "puppy", "goat"], ["films", "books"]]
>>> file_found(nested_folders, "puppy")
True
>>> file_found(nested_folders, "poems")
False
```

Finding a File in Nested Structures: Recursive

```
def file_found(folder, target):
    # base case
    if folder == []:
        return False
    else:
        first_item = folder[0]

        # check if the first item is a folder (list), if so, recurse
        if type(first_item) == list and first_item != []:
            return file_found(first_item, target) or
                                file_found(folder[1:], target)

        # otherwise, item is a file name, so we check if it's our
        # target, or if the remaining files/folders contain our target
        return first_item == target or file_found(folder[1:], target)
```

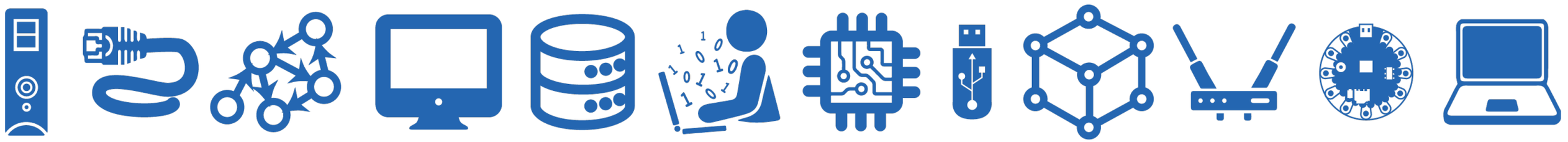
```
>>> nested_folders = ["homework", ["cat", "puppy", "goat"], ["films", "books"]]
>>> file_found(nested_folders, "puppy")
True
>>> file_found(nested_folders, "poems")
False
```

Compare `file_found` approaches

```
def file_found_iterative(folder, target):
    # create an initial list or pile of files/folders to look through
    pile = []
    for item in folder:
        pile+=item
    # keep looking while the pile isn't empty
    while len(pile) > 0:
        # get and remove the last item from the pile
        item = pile[-1]; pile = pile[:-1]
        # if the item is a folder (list) add each item
        # inside the folder onto the pile
        if type(item) == list:
            for obj in item:
                pile += obj
        # otherwise check if the current file is our target
        elif item == target:
            return True
    return False
```

```
def file_found(folder, target):
    if folder == []: # base case
        return False
    else:
        first_item = folder[0]
        # check if the first item is a folder (list), if so, recurse
        if type(first_item) == list and first_item != []:
            return file_found(first_item, target) or file_found(folder[1:], target)
        # otherwise, item is a file name, so we check if it's our
        # target, or if the remaining files/folders contain our target
        return first_item == target or file_found(folder[1:], target)
```

More Examples



Exercise: Forming Base Case & Recursive Rules

- The easiest way to understand recursion is to first see examples of it
- Let's start by examining a familiar recursive definition in mathematics
- The set of natural numbers can be defined as follows:
  - **1** is a natural number
  - If **n** is a natural number, then **n+1** is a natural number

- Building blocks of a recursive idea:
  1. **Base case(s):** **1** is a natural number
  2. **Recursive rule(s):** If **n** is a natural number, then **n+1** is a natural number

Exercise: Forming Base Case & Recursive Rules

- How would you define the concept of exponentiation a^n as a base case and a recursive rule (assuming `n >= 0`)

- A recursive definition:
  - Base case:
  - Recursive rule:

Exercise: Forming Base Case & Recursive Rules

- How would you define the concept of exponentiation a^n as a base case and a recursive rule (assuming `n >= 0`)

- A recursive definition:
  - Base case: $a^0 = 1$
  - Recursive rule: $a^n = a * a^{n-1}$

Translating Recursive Ideas To Programs

- Recursive definition for a^n:
  - **Base case:** $a^0 = 1$
  - **Recursive rule:** $a^n = a * a^{n-1}$

```
def power(a, n):
    """
    Returns a^n. Assumes n >= 0.
    """
    if n == 0:
        return 1
    else:
        return a * power(a, n-1)
```

```
>>> print(power(5, 0))
>>> print(power(5, 4))
1
625
```

Exercise: Forming Base Case & Recursive Rules

- Similarly, how would you define the concept of factorial `n!` as a base case and a recursive rule (assuming `n >= 0`)

- A recursive definition:
  - **Base case:** `0! = 1`
  - **Recursive rule:** `n! = n * (n-1)!`

Translating Recursive Ideas To Programs

- Recursive definition for `n!`:
  - **Base case:** `0! = 1`
  - **Recursive rule:** `n! = n * (n-1)!`

```
def factorial(n):
    """
    Returns a!. Assumes n >= 0.
    """
    if n == 0:
        return 1
    else:
        return a * factorial(n-1)
```

```
>>> print(factorial(1))
>>> print(factorial(5))
1
120
```

Exercise: Forming Base Case & Recursive Rules

- Let's examine a more complicated series known as the Fibonacci sequence.
- The Fibonacci sequence is a series of numbers that starts with `0` and `1`, and where each successive number is the sum of the two preceding ones

```
0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ..
```

- A recursive definition:
  - Base cases: $F_0 = 0$ and $F_1 = 1$
  - Recursive rule: $F_n = F_{n-1} + F_{n-2}$

Translating Recursive Ideas To Programs

- Recursive definition for Fibonacci:
  - **Base cases:** $F_0 = 0,\ F_1 = 1$
  - **Recursion:** $F_n = F_{n-1} + F_{n-2}$

```
def fibonacci(n):
    """
    Returns nth Fibonnaci number
    """
    if n <= 1:
        return n
    else:
        return fibonacci(n-1) + fibonacci(n-2)
```

```
>>> print(fibonacci(5))
>>> print(fibonacci(6))
>>> print(fibonacci(7))
5
8
13
```

The end!