CS 134:
Recursion (2)

Announcements & Logistics

- **Lab 6 due today/tomorrow 10 pm**
  - Remember to test your `hIndex` function thoroughly
  - What are some good test cases?
    - Empty tuples, Singletons, tuples of zeroes
    - Tuples with duplicate citation counts
    - Is there a default return value outside conditionals?
  - If matplotlib is complaining, you can always use lab machines
- **HW 6** will be posted this afternoon
    - Covers sorting, dictionaries, sets

Do You Have Any Questions?

Recap: Recursive Approach to Problem Solving

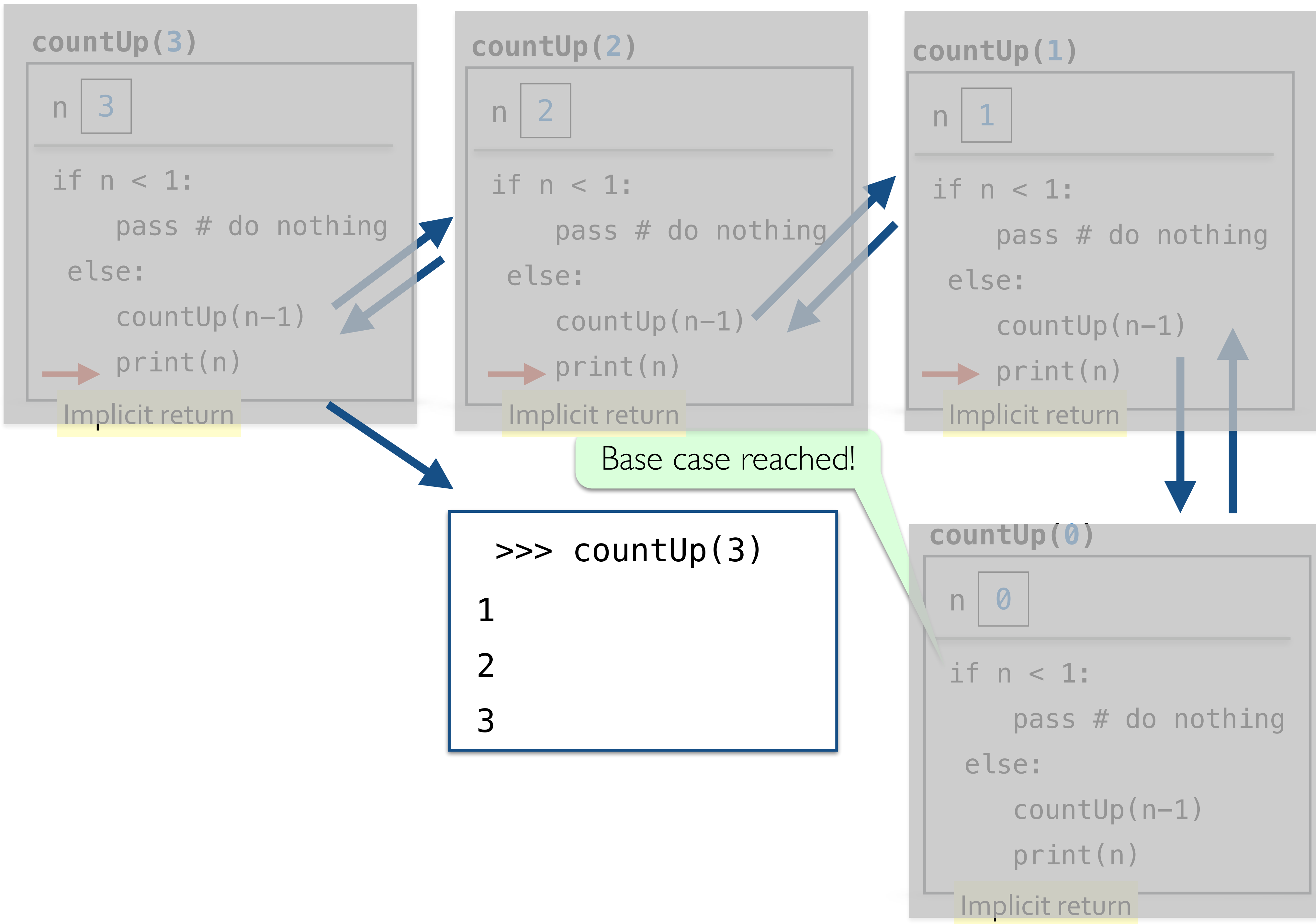
- A recursive function is a function **that calls itself**
- A recursive approach to problem solving has two main parts:
  - **Base case(s).** When the problem is **so small**, we solve it directly, without having to reduce it any further
  - **Recursive step.** Does the following things:
    - Performs 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)

Review: countUp(n)

- Write a recursive function that prints integers from `1` up to `n` (without using any loops)
- Recursive definition of countUp:
  - Base case: `n = 0`, do nothing
  - Recursive rule: `call countUp(n-1), print(n)`

```
def countUp(n):
    '''Prints out integers from 1 up to n'''
    if n < 1:
        pass # do nothing
    else:
        countUp(n-1)
        print(n)
```

Function Frame Model to Understand `countUp`


countUp(3)
n 3
if n < 1:
pass # do nothing
else:
countUp(n-1)
print(n)
Implicit return
countUp(2)
n 2
if n < 1:
pass # do nothing
else:
countUp(n-1)
print(n)
Implicit return
countUp(1)
n 1
if n < 1:
pass # do nothing
else:
countUp(n-1)
print(n)
Implicit return
Base case reached!
countUp(0)
n 0
if n < 1:
pass # do nothing
else:
countUp(n-1)
print(n)
Implicit return
>>> countUp(3)
1
2
3

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 countUpGotcha(n):
    '''Prints ints from 1 up to n’''
    if n < 1:
        pass # do nothing
    else:
        countUpGotcha(n)
        print(n)
```

Subproblem not getting smaller!

GOTCHA #2

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

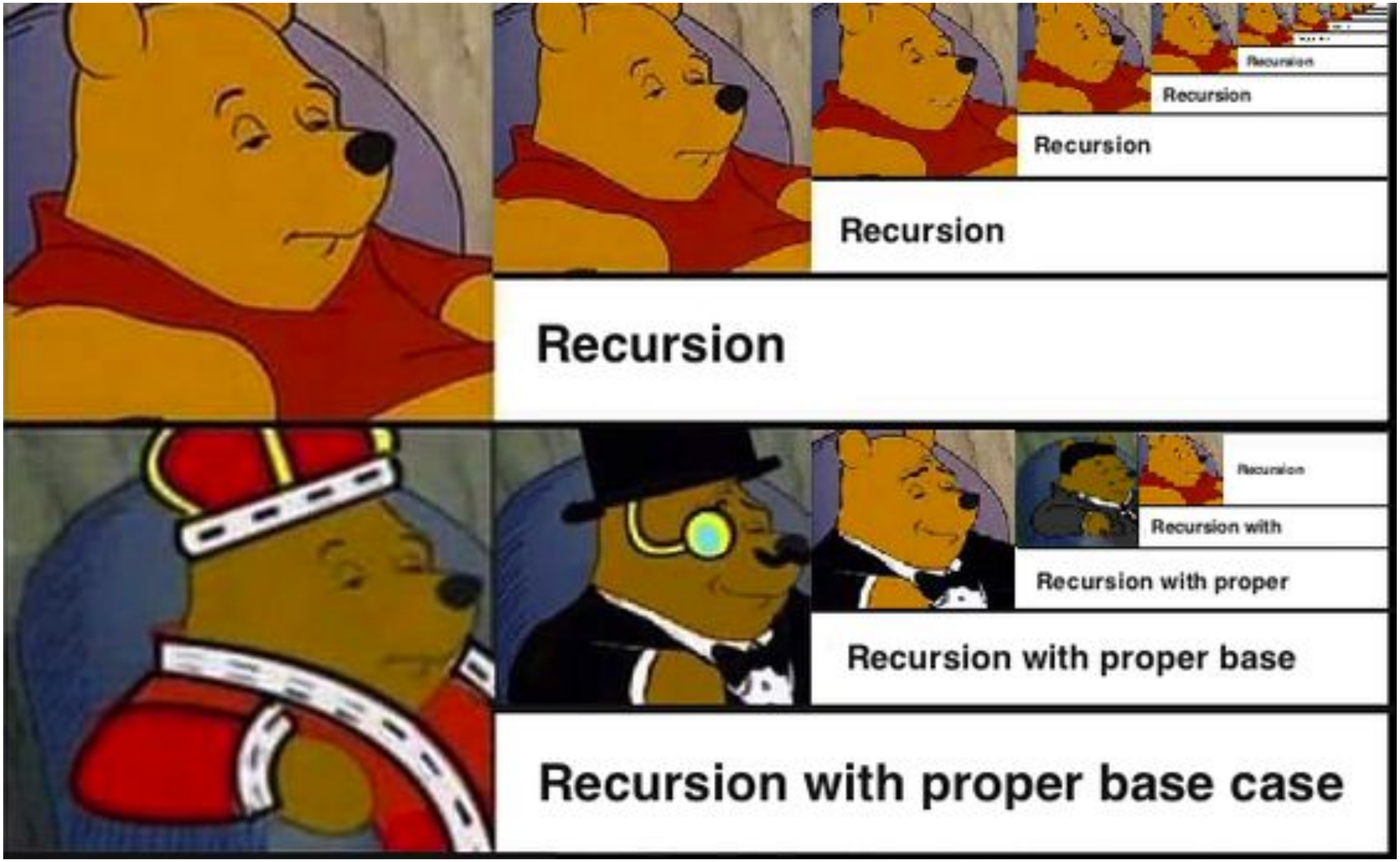
```
def printHalvesGotcha(n):
    if n > 0:
        print(n)
        printHalvesGotcha(n/2)
```

Always 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


I'm going to write a recursive function
With a base case, right?
I'm going to write a recursive function
With a base case, right?
I'm going to write a recursive function
With a base case, right?


Recursion
Recursion
Recursion
Recursion
Recursion
Recursion with
Recursion with proper
Recursion with proper base
Recursion with proper base case

Today’s Plan

- Comparing iterative vs. recursive ideas and discussing trade offs
- Some live coding involving the implementation of recursive vs. iterative functions

Iterative Approach to `sumList`

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

```
def sumListIterative(numList):
    sum = 0
    for num in numList:
        sum += num
    return sum
```

```
sumListIterative([3, 4, 20, 12, 2, 20])
```

```
61
```

Recursive approach to `sumList`

- Let's say the name of our list is `numList`
- **Base case:** `numList` is empty, return 0
- **Recursive rule:** return first element of `numList` plus result from calling `sumList` on rest of the elements of the list.
- One way to think of the recursive rule: say the list has numbers `[6, 3, 6, 5]`
  - `sum([6, 3, 6, 5]) = 6 + sum([3, 6, 5])`
  - `sum([3, 6, 5]) = 3 + sum([6, 5])`
  - `sum([6, 5]) = 6 + sum([5])`
  - `sum([5]) = 5 + sum([])`
- And for the base case we have `sum([])` returns `0`

Recursive approach to `sumList`

```
def sumList(numList):
    """Returns sum of given list"""
    if numList == []:
        return 0
    else:
        return numList[0] + sumList(numList[1:])
```

```
sumList([3, 4, 20, 12, 2, 20])
```

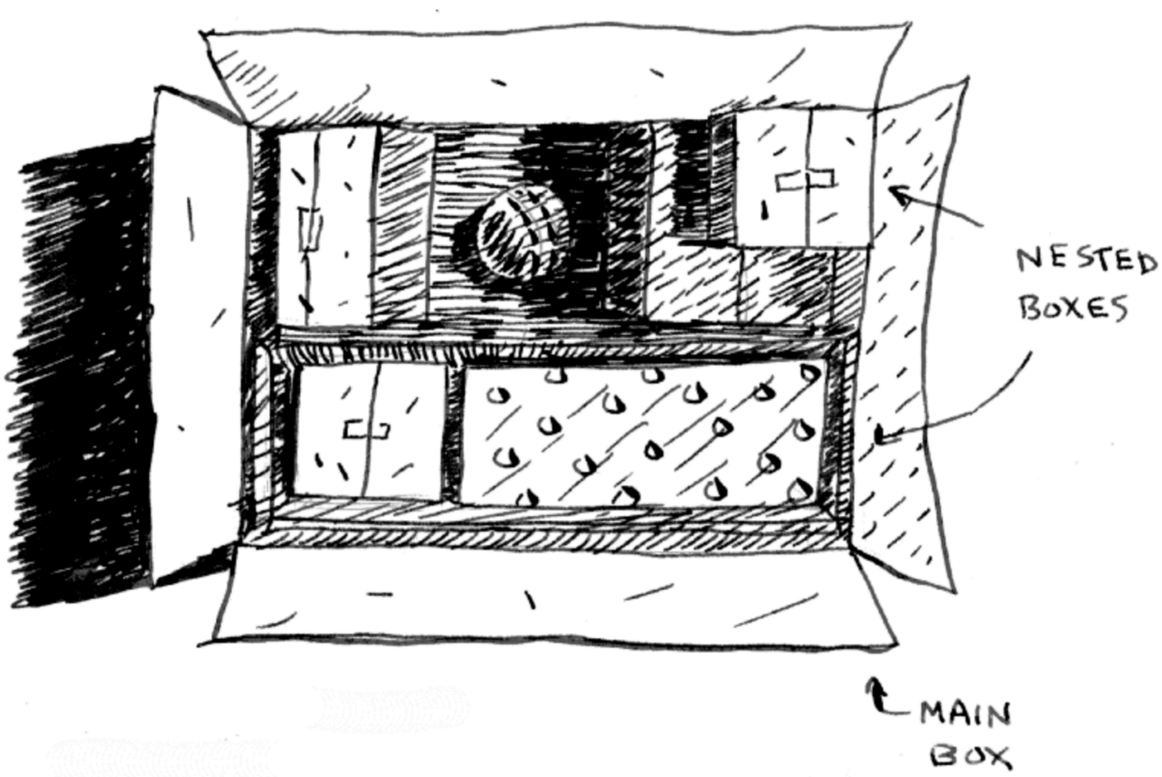
```
61
```

What's The Big Deal With Recursion?

- So far, it seems like there's not a whole lot to gain from learning recursion if we already know about iterative methods
- However, in some cases you'll find that the recursive solution can be described in a more elegant manner, resulting in fewer lines of code
- And fewer lines of code often correlates with less debugging!
- We'll start simple and build up to a scenario that demonstrates a tangible benefit to learning recursion

A Simple Real World Task

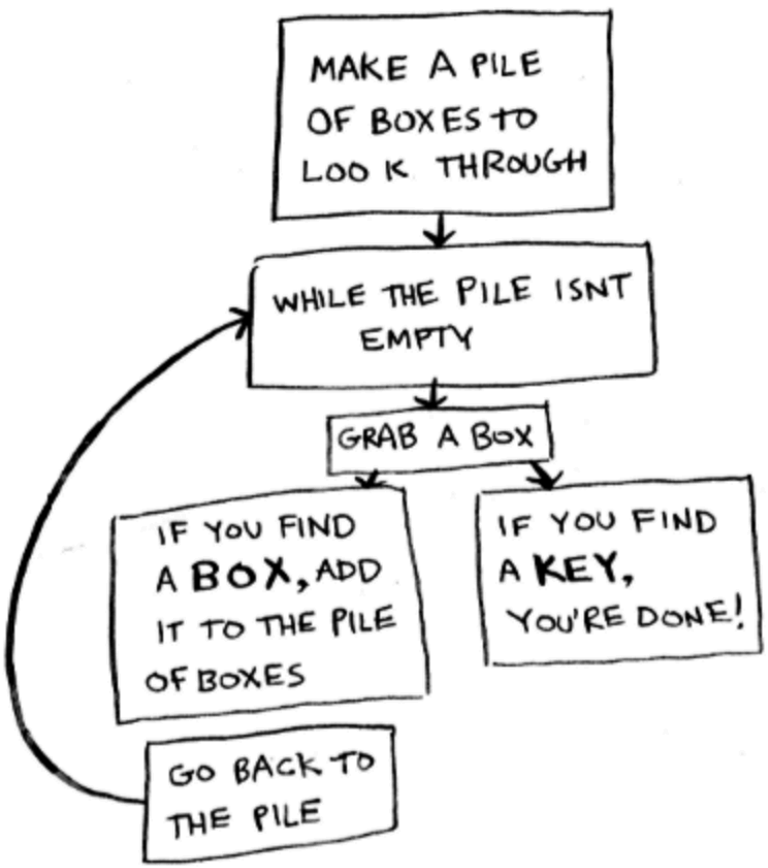
- Consider trying to find a key that is lost in a pile of boxes within boxes.
- It seems like a silly analogy to begin with, but we'll see that this task is quite similar to trying to find a file on your computer!



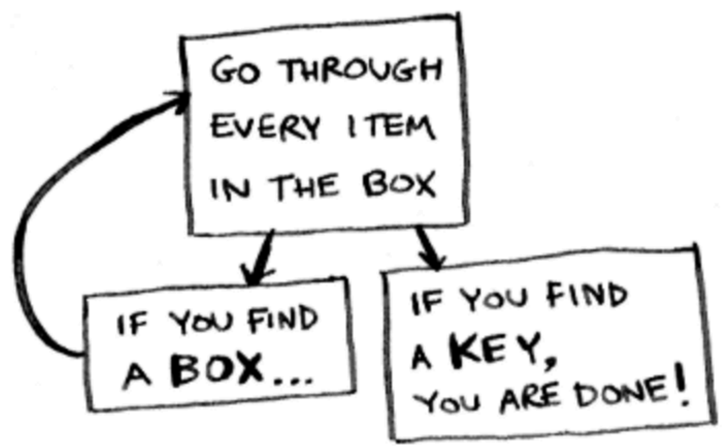
Credit to Aditya Bhargava for the nice illustrations

Comparing Approaches To Finding The Key

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



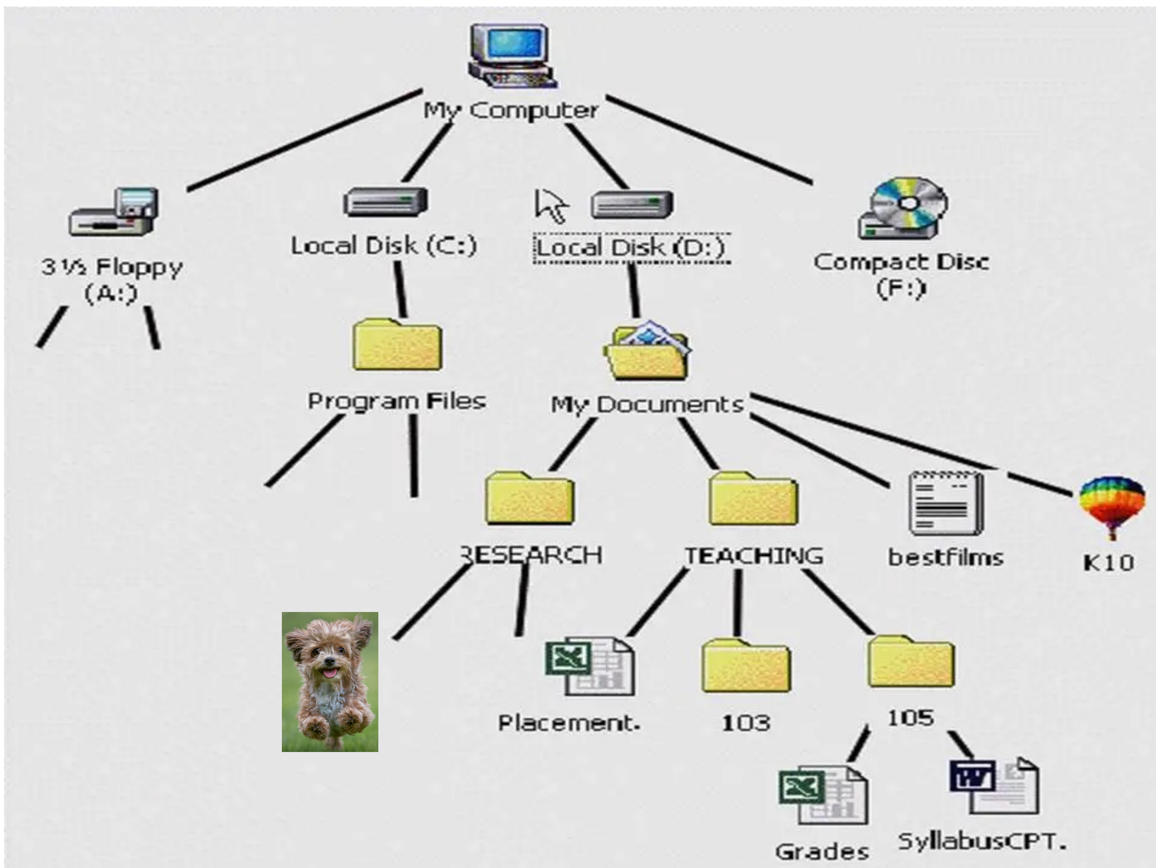
Iterative Approach



Recursive Approach

Searching For A File On Our Computer

- We'll now do a Jupyter notebook exercise to compare iterative and recursive approaches to finding a file on our computer — instead of boxes within boxes, we'll have folders within folders getting in the way of finding the picture of a puppy



Pros and Cons of Recursion

- Pros:
  - Can lead to syntactically simpler programs
  - Many tasks, such as exploring and building file systems, computer networks, or data structures used in machine learning, are best written as recursive programs
  - Because of the first 2 points, you will often see a lot of recursive computer code or pseudocode out in the real world
- Cons:
  - Recursive procedures often have more computational overhead than iterative ones because of repeated function calls
  - Recursion has a steeper learning curve (but can be very rewarding once you get the hang of it — simplifies notation, amount of code you write, etc.)
  - To understand recursion you must understand recursion (an old CS folklore joke about the steep learning curve)

Next Time

- Turtle and graphical recursion!