Middlebury

CSCI 146: Intensive Introduction to Computing

Fall 2025

Lecture 13: More Recursion

Goals for today

- Recursively draw images with the `turtle`.
- Apply techniques for improving the efficiency of recursive algorithms.
- Determine when to use recursion.

Review (steps for writing a recursive function):

1. Define the function header, including the parameters.
2. Define the recursive case.
3. Define the base case.
4. Put it all together.

Warmup: write a recursive function `rec_len` to calculate the length of a sequence.

Exercise from last class: a *recursive* palindrome checker.

Here is a loop-based implementation:

```python
def is_palindrome_loop(word):
    """
    Determines if a word is a palindrome.
    Args:
      word: word to check (str)
    Returns:
      True if the input word is a palindrome, False otherwise
    """
    for i in range(len(word) // 2):
      if word[i] != word[-i - 1]:
        return False
    return True
```

Examples: racecar, noon, kayak, madam, rotator

When you're done, try to extend it to ignore punctuation and spaces to handle *palindrome phrases*:

1. A Toyota
2. If I had a hi-fi,
3. UFO tofu
4. Never odd or even.
5. A man, a plan, a canal - Panama!

Possible implementation of the recursive palindrome checker.

```python
def is_palindrome_recursive(word):
    """
    Determines if a word is a palindrome.

    Args:
      word: word to check (str)
    Returns:
      True if the input word is a palindrome, False otherwise
    """
    if len(word) < 2:
        return True
    if word[0] != word[-1]:
        return False
    return is_palindrome_recursive(word[1:len(word) - 1])
```

③ convert to lower case

UFO tofu

If I had a hi-fi

preprocess:

① .split for words

② remove punctuation s = "!.,- _" (space)

import string

string.punctuation

str.replace(c, "") (empty string)

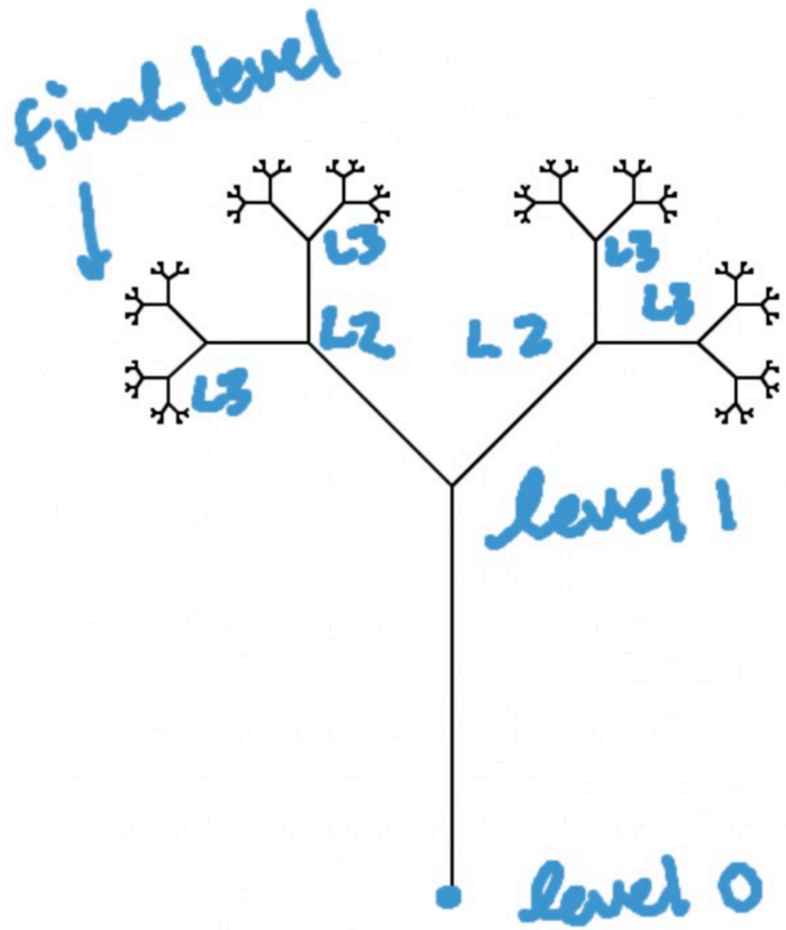
Drawing a spiral with the turtle.

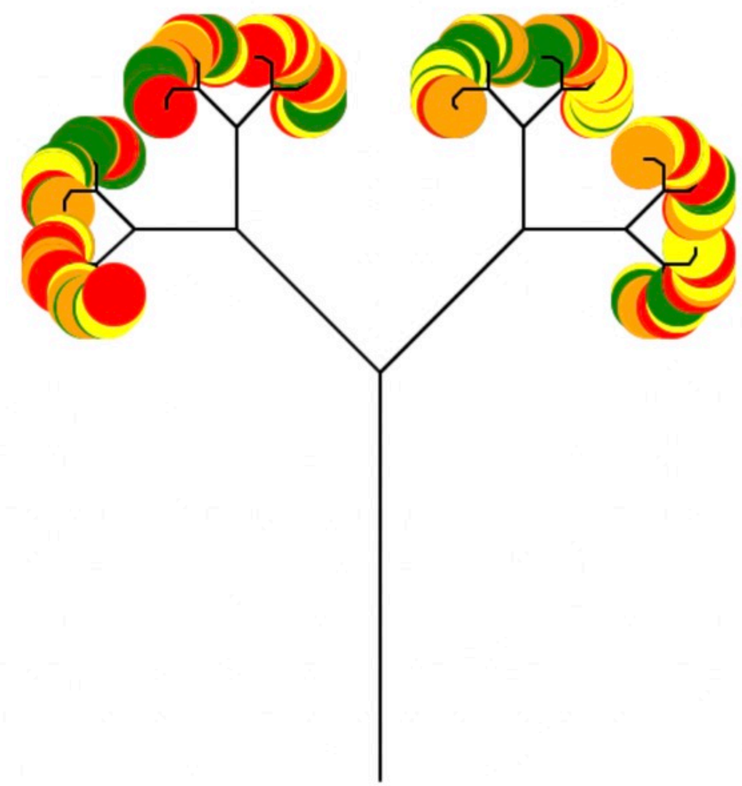

```python
import turtle as t

def spiral1(length, levels):
    """
    Draw a spiral with 'levels' segments with initial 'length'
    """
    # Implicit base case: do nothing if levels == 0
    if levels > 0:
        t.forward(length)
        t.left(30)
        spiral1(0.95 * length, levels - 1) # Recurse
```

```python
import turtle as t

def spiral2(length, levels):
    """
    Draw a spiral with 'levels' segments with initial 'length'
    """
    # Implicit base case: do nothing if levels == 0
    if levels > 0:
        t.forward(length)
        t.left(30)
        spiral2(0.95 * length, levels - 1) # Recurse
        t.right(30)
        t.backward(length)
```

Pending operations are useful to return to our starting point: let's draw a tree!

How can we draw this?





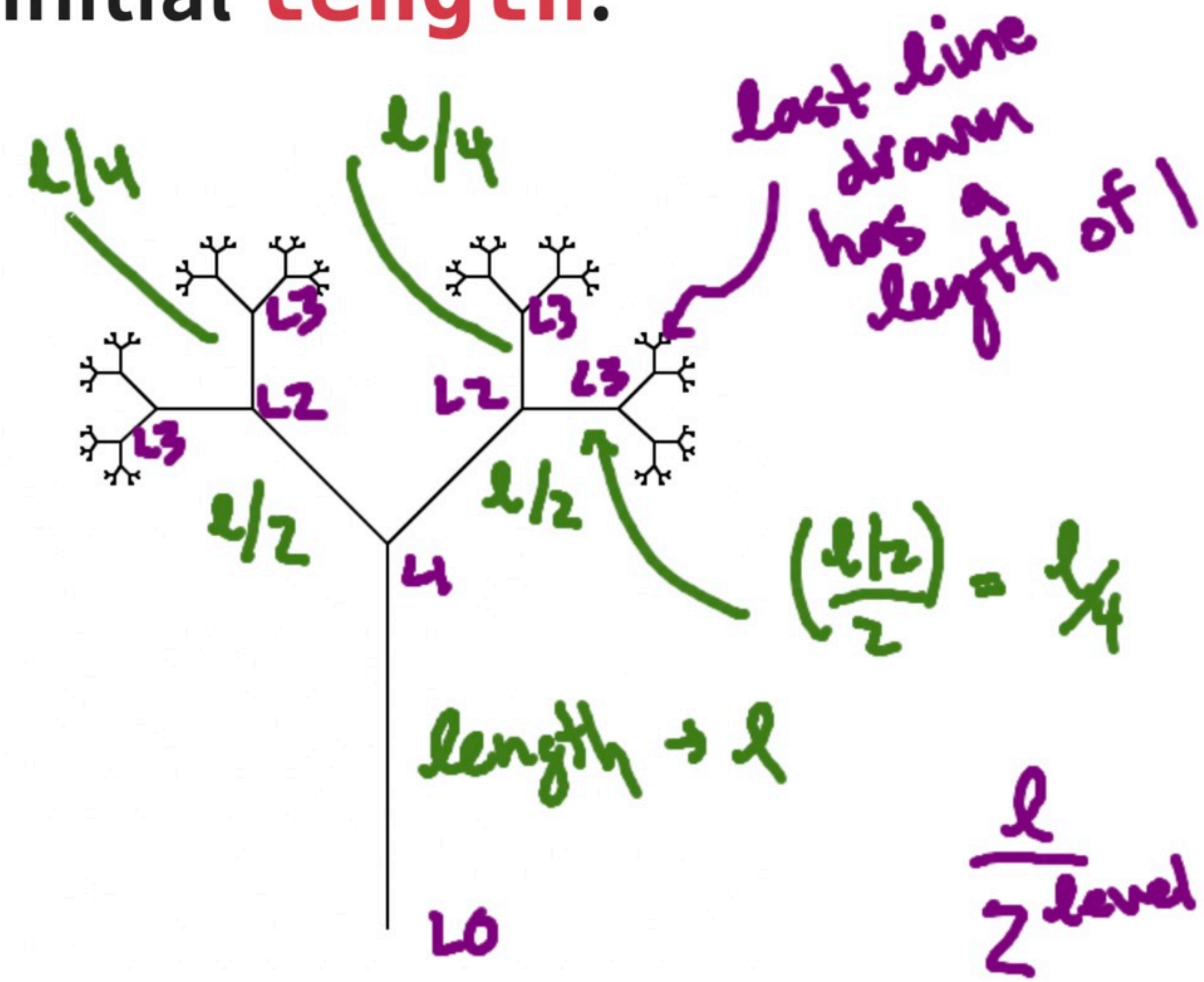
Pending operations are useful to return to our starting point: let's draw a tree!

```python
def draw_tree(length):
    """
    Draw a recursive tree and return to where the turtle started
    Args:
        length: length of initial tree trunk
    """
    if length > 0:
        t.forward(length)           # draw tree branch
        t.right(45)                 # prepare to draw right subtree
        draw_tree(length // 2)      # draw right subtree
        t.left(45 * 2)              # undo right turn, then turn left again
        draw_tree(length // 2)      # draw left subtree
        t.right(45)                 # undo left turn
        t.backward(length)          # trace back down the tree branch
```

How many "levels" are there in this tree? Let's extend our function to visualize this.

```python
def draw_tree(length, level=0):
    """
    Draw a recursive tree and return to where the turtle started
    Args:
        length: length of initial tree trunk
        level: current level (# branches from root to current location)
    """
    if length > 0:
        t.forward(length)                       # draw tree branch
        t.right(45)
        draw_tree(length // 2, level + 1)   # draw right subtree
        t.left(45*2)                            # undo right turn, then turn left again
        draw_tree(length // 2, level + 1)   # draw left subtree
        t.right(45)                             # undo left turn
        t.backward(length)                      # trace back down the tree branch
        t.write("L" + str(level), align="center")
```

We can also relate the total number of levels to the initial `length`.



$$\frac{l}{2^{level}}$$

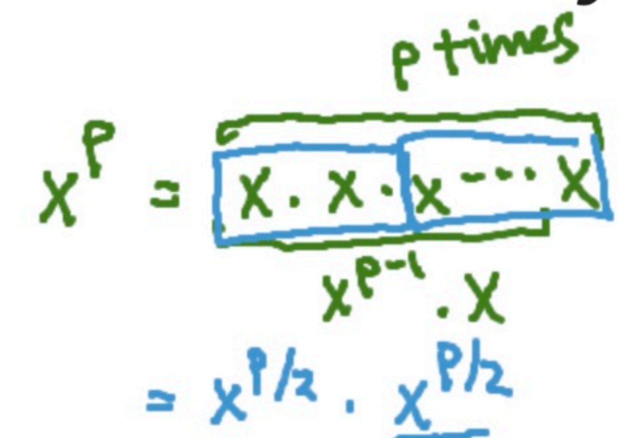
level	length drawn
0	l
1	$l/2$
2	$l/4$
3	$l/8$
⋮	⋮
level	1

$$\rightarrow \frac{length}{2^{level}} = 1$$

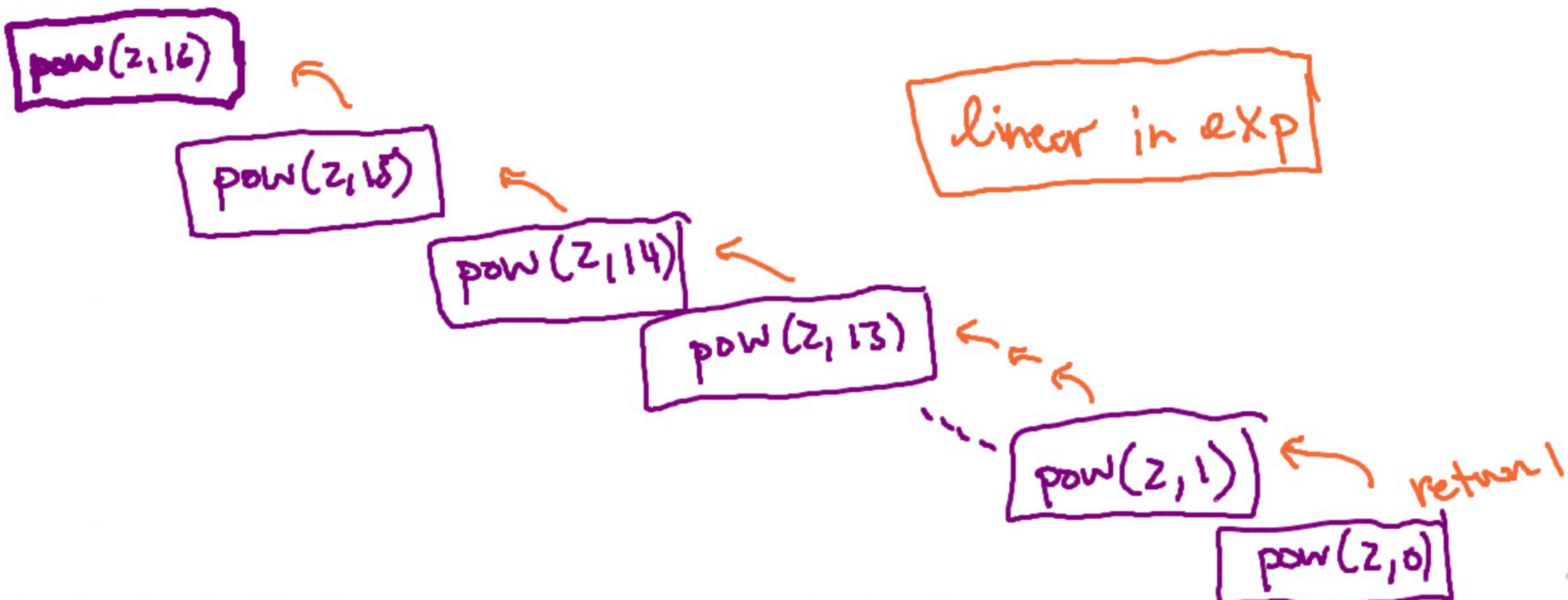
$$level = \log_2(length)$$

Another example: implementing pow recursively.

```python
def power(base, exp):
  if exp == 0:
      return 1
  else:
      return base * power(base, exp - 1)
```



If each call to power takes 1 second, how many seconds for power(2, 16)?



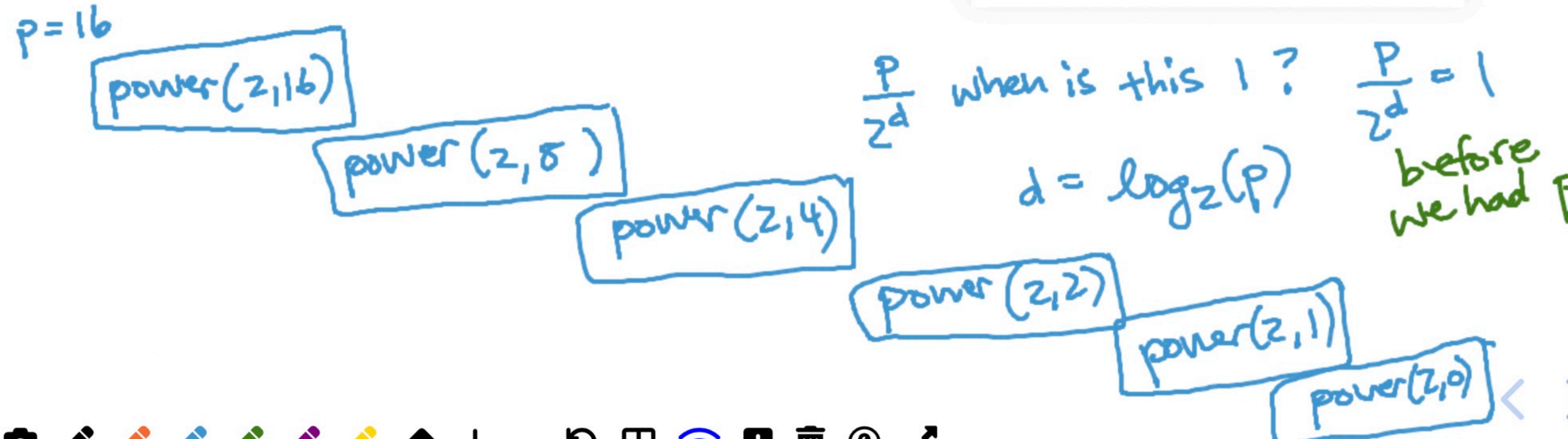
Can we do better?

What if we used the fact that $x^p = x^{\frac{p}{2}} x^{\frac{p}{2}}$?
Assume p is an even number for now (and actually a power of 2).

Is this what we want?

```python
def power(base, exp):
  if exp == 0:
      return 1
  else:
      return power(base, exp // 2) * power(base, exp // 2)
```

```python
def power(base, exp):
  if exp == 0:
      return 1
  else:
      y = power(base, exp // 2)
      return y * y
```



Extending our current `power` function to handle any `exp` (maintaining efficiency).

```
def power(base, exp):
    if exp == 0:
        return 1
    elif exp == 1:
        return base
    elif exp % 2 == 0:
        y = power(base, exp // 2)
        return y * y
    else:
        # exp is odd, so exp - 1 will be even
        return base * power(base, exp - 1)
```

Recursion is not always the best tool for the task.

Example: Fibonacci numbers are *defined* recursively, but a simple recursive implementation is inefficient:

0, 1, 1, 2, 3, 5, 8, 13, 21, ...

$$F_n = F_{n-1} + F_{n-2} \quad \text{with } F_0 = 0,\ F_1 = 1$$

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

Instead we can use something called "memoization".

```
calculated_fibs = {} # dictionary mapping n -> fn (could also be a list)
def fib(n):
    if n <= 1:
        return n
    elif n in calculated_fibs:
        return calculated_fibs[n]
    else:
        fn = fib(n - 1) + fib(n - 2)
        calculated_fibs[n] = fn
        return fn
```

(more in future CS classes)

Summary and Reminders

- Remember to (1) include a base case and (2) ensure your recursive case approaches the base case.
- Programming Assignment 4 final due date on Thursday.
- All Gradescope tests will be visible from now on.
- Use "Regrade Requests" form on the website. See Gradescope comments by clicking on **Code**.