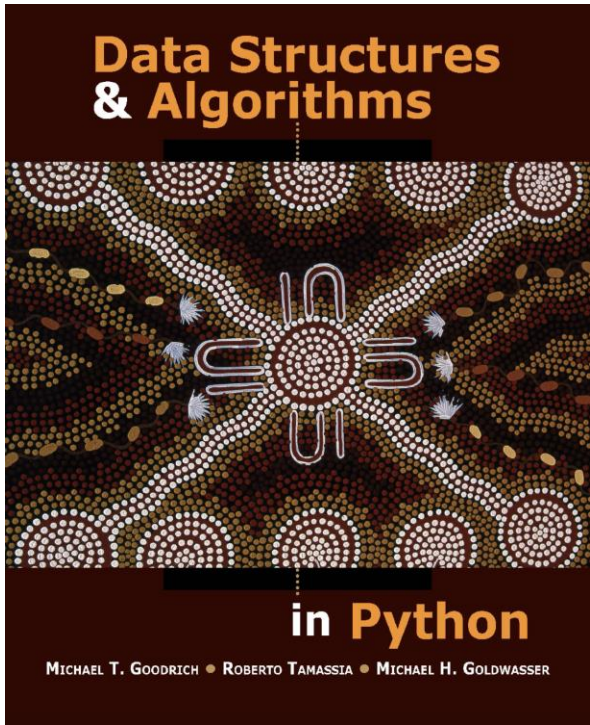
RECURSION

School of Artificial Intelligence

PREVIOUSLY ON DS&A

- Asymptotic Analysis
  - Big-O notation
  - Big-Theta notation
  - Big-Omega notation
- Recursion
  - Factorial, English Ruler, Binary Search, File System
- Recursion analysis
  - File System
    - Armotisation
    - Tree structure
    - More on tree algorithm analysis later



THREE-WAY SET DISJOINTNESS

- 3 sequences of numbers: A, B and C
  - No individual sequence contains duplicate values
  - The intersection of the three sequences is empty
  - There is no element x such that $x \in A, x \in B, and\ x \in C$
  - Complexity?

```
1  def disjoint1(A, B, C):
2    """Return True if there is no element common to all three lists."""
3    for a in A:
4      for b in B:
5        for c in C:
6          if a == b == c:
7            return False       # we found a common value
8    return True                # if we reach this, sets are disjoint
```

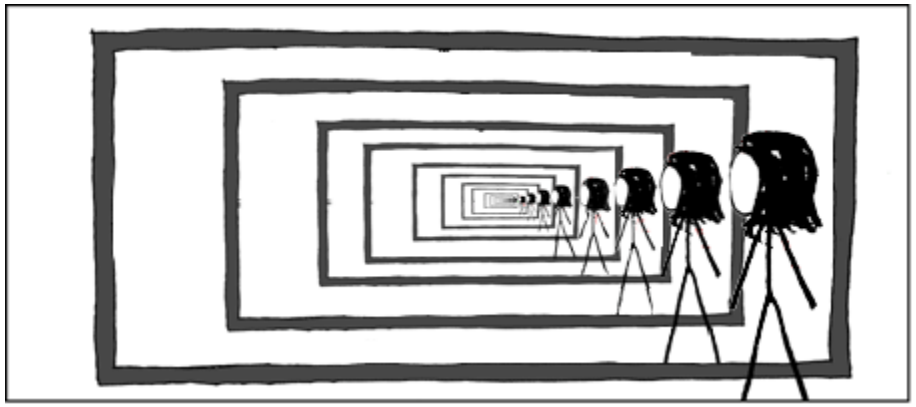
THREE-WAY SET DISJOINTNESS

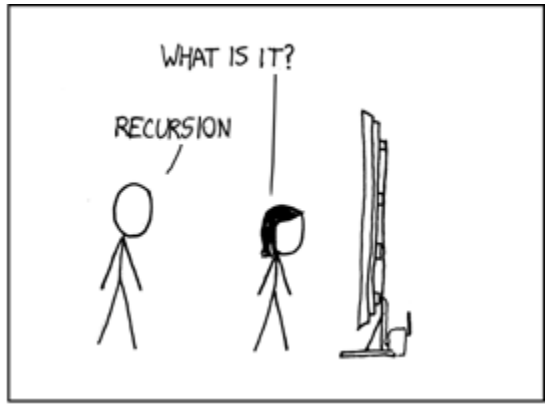
- 3 sequences of numbers: A, B and C
  - No individual sequence contains duplicate values
  - The intersection of the three sequences is empty
  - There is no element x such that $x \in A, x \in B, and\ x \in C$
  - Improved version: $O(n^2)$

```
1  def disjoint2(A, B, C):
2    """Return True if there is no element common to all three lists."""
3    for a in A:
4      for b in B:
5        if a == b:              # only check C if we found match from A and B
6          for c in C:
7            if a == c          # (and thus a == b == c)
8              return False     # we found a common value
9    return True                # if we reach this, sets are disjoint
```

THIS LECTURE

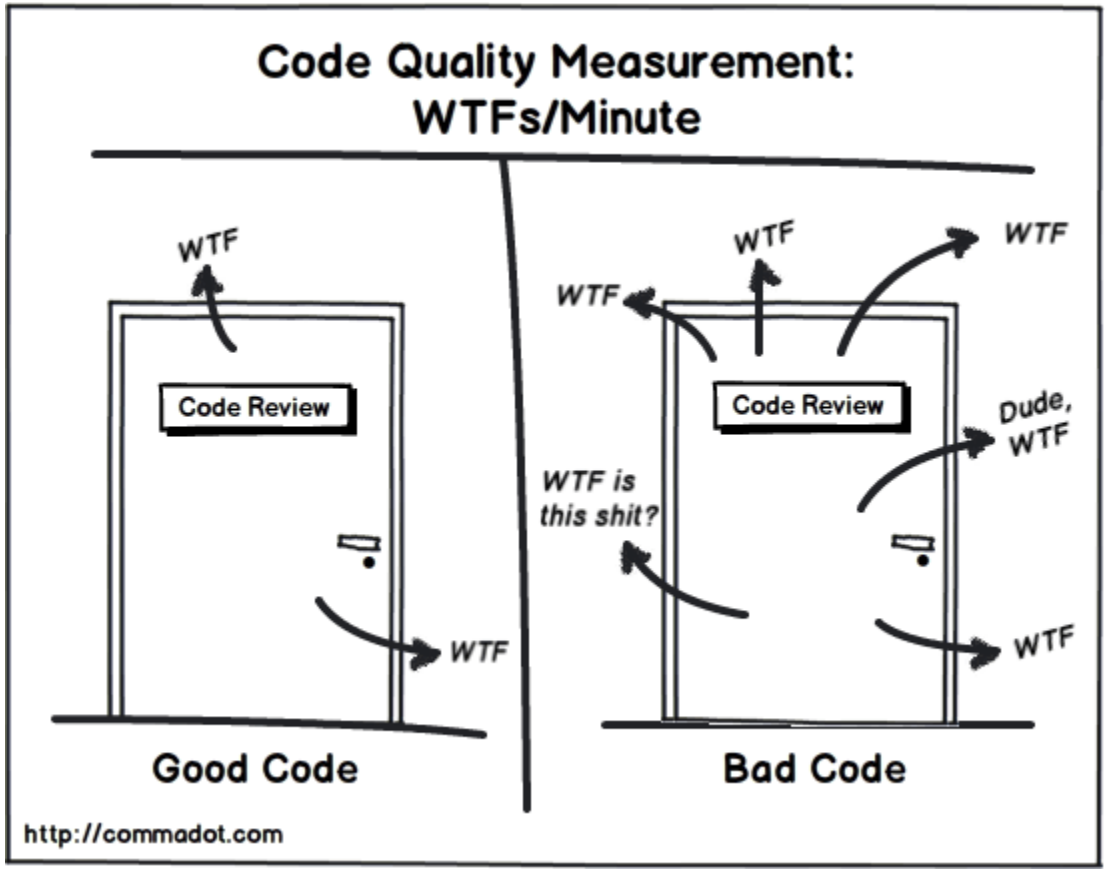
- Recursion
  - Bad recursions
  - Types of recursions





RECURSION

- What can go wrong?



ELEMENT UNIQUENESS (AGAIN)

- Find if there are duplicate elements in a sequence
- With recursive algorithm
- Logic
  - If n = 1, elements are unique
  - If n >=2, elements are unique iff (if and only if)
    - the first n-1 elements are unique
    - The last n-1 elements are unique
    - First != last

```
1  def unique3(S, start, stop):
2    """Return True if there are no duplicate elements in slice S[start:stop]."""
3    if stop − start <= 1: return True                  # at most one item
4    elif not unique(S, start, stop−1): return False    # first part has duplicate
5    elif not unique(S, start+1, stop): return False    # second part has duplicate
6    else: return S[start] != S[stop−1]                 # do first and last differ?
```

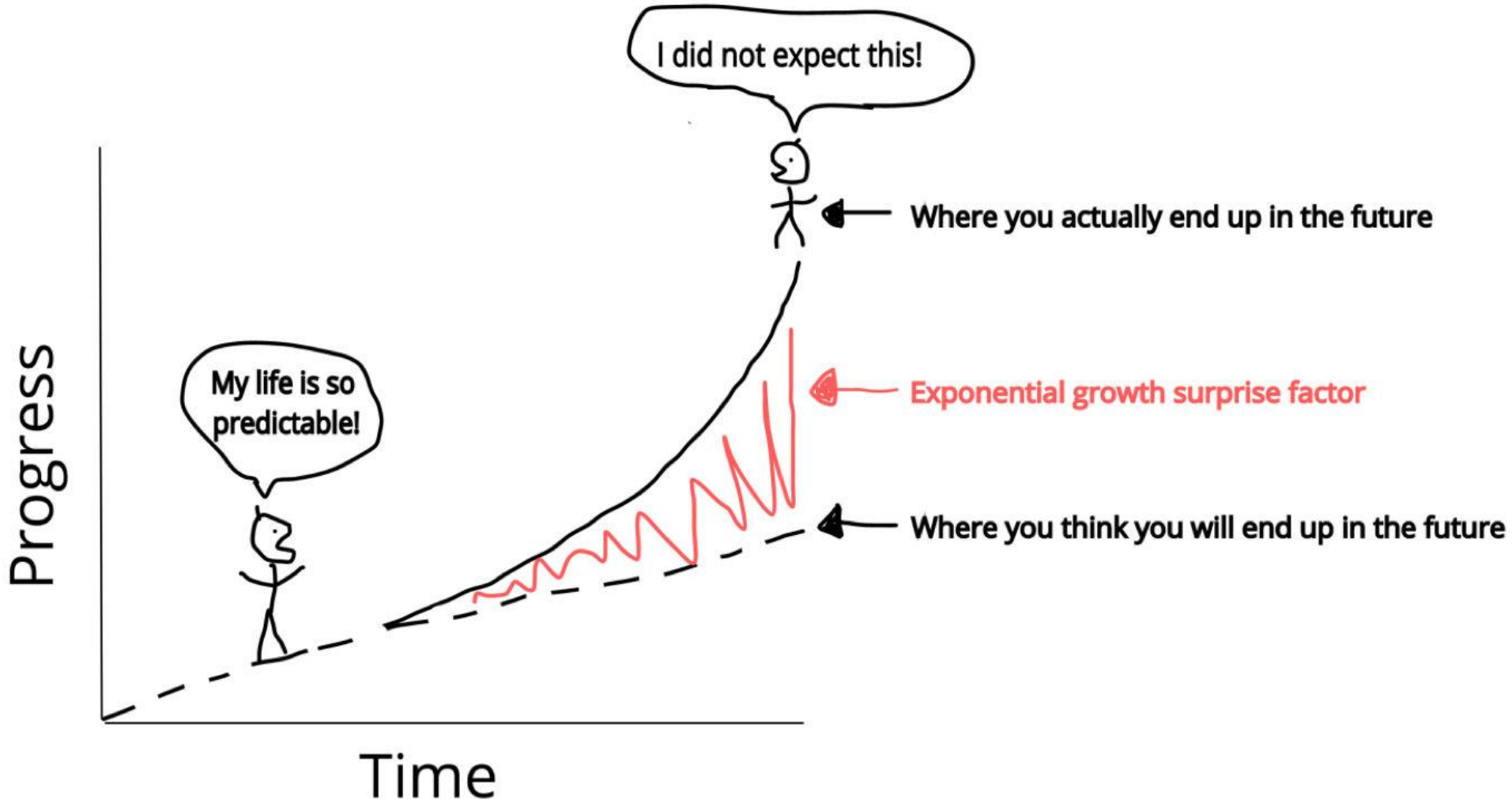
ELEMENT UNIQUENESS (AGAIN)

- Find if there are duplicate elements in a sequence
- Complexity?
  - Line 3: O(1)
  - Line 6: O(1)
  - Overall time: proportional to # of invocations
  - But how many?

```
1  def unique3(S, start, stop):
2    """Return True if there are no duplicate elements in slice S[start:stop]."""
3    if stop − start <= 1: return True                  # at most one item
4    elif not unique(S, start, stop−1): return False    # first part has duplicate
5    elif not unique(S, start+1, stop): return False    # second part has duplicate
6    else: return S[start] != S[stop−1]                 # do first and last differ?
```

ELEMENT UNIQUENESS (AGAIN)

- F
- (



FIBONACCI FUNCTION

- Previously on Fibonacci function
  - F(0) = 0
  - F(1) = 1
  - F(n) = F(n-1) + F(n-2)

```
1  def bad_fibonacci(n):
2    """Return the nth Fibonacci number."""
3    if n <= 1:
4      return n
5    else:
6      return bad_fibonacci(n−2) + bad_fibonacci(n−1)
```

FIBONACCI FUNCTION

- Previously on Fibonacci function
  - F(0) = 0
  - F(1) = 1
  - F(n) = F(n-1) + F(n-2)
- Efficiency?
  - C_n : # of calls to bad_fibonacci() performed
  - # of calls doubles for each two consecutive indeices
  - $O(2^n)$

```
1  def bad_fibonacci(n):
2    """Return the nth Fibonacci number."""
3    if n <= 1:
4      return n
5    else:
6      return bad_fibonacci(n−2) + bad_fibonacci(n−1)
```

$$
\begin{aligned}
c_0 &= 1 \\
c_1 &= 1 \\
c_2 &= 1 + c_0 + c_1 = 1 + 1 + 1 = 3 \\
c_3 &= 1 + c_1 + c_2 = 1 + 1 + 3 = 5 \\
c_4 &= 1 + c_2 + c_3 = 1 + 3 + 5 = 9 \\
c_5 &= 1 + c_3 + c_4 = 1 + 5 + 9 = 15 \\
c_6 &= 1 + c_4 + c_5 = 1 + 9 + 15 = 25 \\
c_7 &= 1 + c_5 + c_6 = 1 + 15 + 25 = 41 \\
c_8 &= 1 + c_6 + c_7 = 1 + 25 + 41 = 67
\end{aligned}
$$

FIBONACCI FUNCTION

- What is the problem with the current Fibonacci function?

FIBONACCI FUNCTION

- What is the problem with the current Fibonacci function?
- In F(n-1), we compute F(n-3) and F(n-2)
- For F(n), we need to compute F(n-1) and F(n-2) again
- In the textbook: returning a tuple

```
1  def good_fibonacci(n):
2    """Return pair of Fibonacci numbers, F(n) and F(n-1)."""
3    if n <= 1:
4      return (n,0)
5    else:
6      (a, b) = good_fibonacci(n-1)
7      return (a+b, a)
```

FIBONACCI FUNCTION

- What is the problem with the current Fibonacci function?
- In F(n-1), we compute F(n-3) and F(n-2)
- For F(n), we need to compute F(n-1) and F(n-2) again
- But returning a tuple may not be what typical computer programs do
- Complexity?
- Dynamic Programming

```
memo = { }
fib(n):
        if n in memo: return memo[n]
        else: if n ≤ 2 : f = 1
              else: f = fib(n − 1) + fib(n − 2)
              memo[n] = f
              return f
```

MAXIMUM RECURSIVE DEPTH IN PYTHON

- Infinite loop:
  - a = 0
  - while (a < n):
    - print(a)
- Infinite Recursion

```
def fib(n):
    return fib(n)
```

- Remember: each call to the recursion function should progress towards the base case
  - E.g. parameter value – 1
- Python designers: "We are going to give you a limit on how many recursive calls can be made in your program"

MAXIMUM RECURSIVE DEPTH IN PYTHON

- Maximum recursive depth
  - 1000 by default
  - RuntimeError is thrown when this limit is reached
    - "Maximum recursion depth exceeded."
  - Sufficient for many applications
  - Some algorithms have recursive depth proportional to n

```
import sys
old = sys.getrecursionlimit( )      # perhaps 1000 is typical
sys.setrecursionlimit(1000000)      # change to allow 1 million nested calls
```

FURTHER EXAMPLES OF RECURSION

- Linear recursion
  - The body of the function makes at most one new recursive call
  - Factorial()
  - Binary_search()
  - Recursion trace: a single sequence of calls
  - Summing the elements of a sequence recursively

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
4	3	6	2	8	9	3	2	8	5	1	7	2	8	3	7

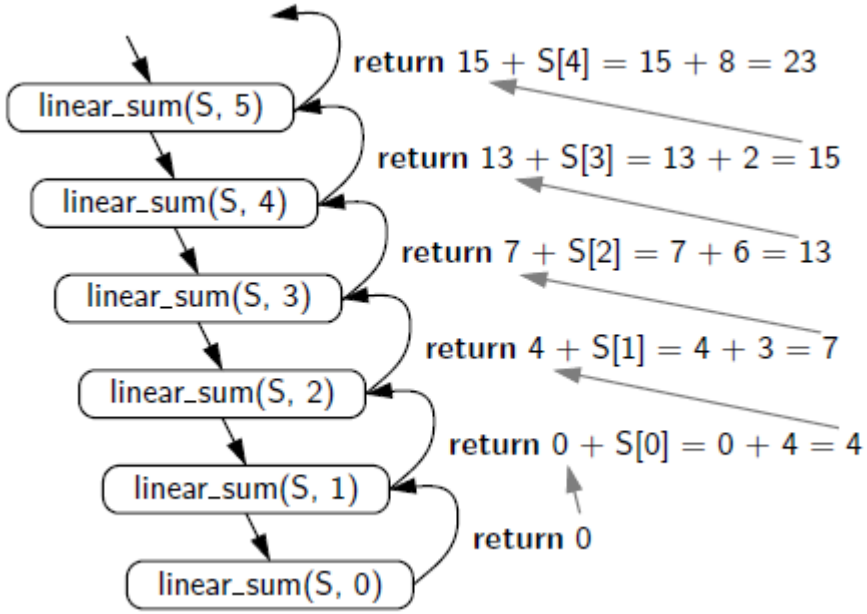
```
1  def linear_sum(S, n):
2    """Return the sum of the first n numbers of sequence S."""
3    if n == 0:
4      return 0
5    else:
6      return linear_sum(S, n−1) + S[n−1]
```

FURTHER EXAMPLES OF RECURSION

- Linear recursion
  - The body of the function makes at most one new recursive call
  - Factorial()
  - Binary_search()
  - Recursion trace: a single sequence of calls
  - Summing the elements of a sequence recursively

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
4	3	6	2	8	9	3	2	8	5	1	7	2	8	3	7

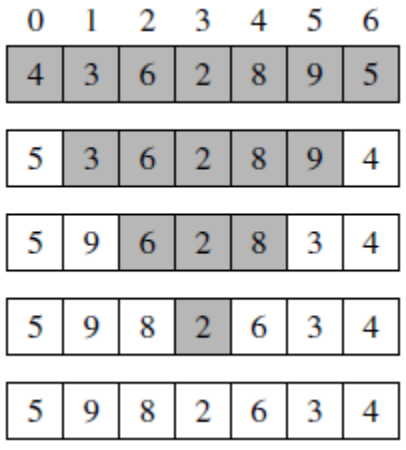
```
1  def linear_sum(S, n):
2    """Return the sum of the first n numbers of sequence S."""
3    if n == 0:
4      return 0
5    else:
6      return linear_sum(S, n−1) + S[n−1]
```



FURTHER EXAMPLES OF RECURSION

- Linear recursion
  - Reversing a sequence with recursion
    - Temp = S[1]
    - S[1] = S[n-1]
    - S[n-1] = temp
    - So on for S[2] and S[n-2]
    - Textbook: you are likely to annoy your interviewer if you use a Python coding style

```
1  def reverse(S, start, stop):
2    """Reverse elements in implicit slice S[start:stop]."""
3    if start < stop - 1:                          # if at least 2 elements:
4      S[start], S[stop-1] = S[stop-1], S[start]   # swap first and last
5      reverse(S, start+1, stop-1)                 # recur on rest
```



FURTHER EXAMPLES OF RECURSION

- Linear recursion
  - Raising a number x to an non-negative integer n
  - Power(x, n) = x^n

$$power(x,n) = \begin{cases} 1 & \text{if } n = 0 \\ x \cdot power(x, n-1) & \text{otherwise.} \end{cases}$$

  - Can this be improved?

```
1  def power(x, n):
2    """Compute the value x**n for integer n."""
3    if n == 0:
4      return 1
5    else:
6      return x * power(x, n-1)
```

FURTHER EXAMPLES OF RECURSION

- Linear recursion
  - Raising a number x to an non-negative integer n
  - Power(x, n) = x^n
  - Can this be improved?
    - Let k = floor(n/2)
    - $(x^k)^2 = x^n$ (n is even), x^{n-1} (n is odd) = x. $(x^k)^2$

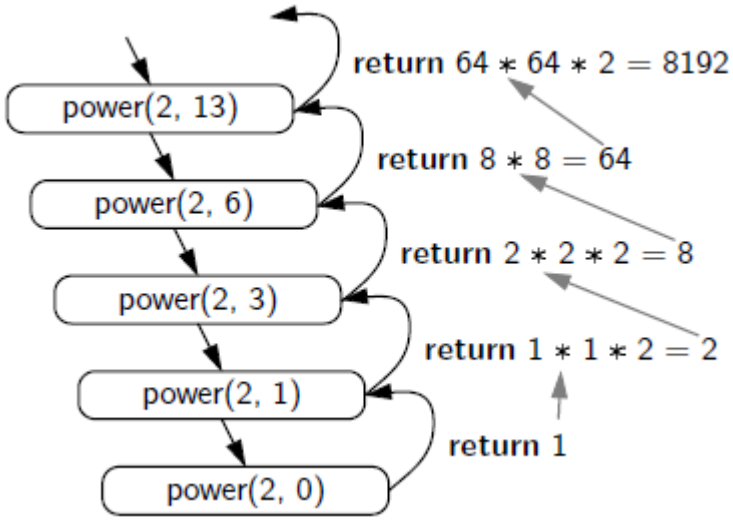
$$power(x,n) = \begin{cases} 1 & \text{if } n = 0 \\ x \cdot \left(power\left(x, \lfloor \frac{n}{2} \rfloor\right)\right)^2 & \text{if } n > 0 \text{ is odd} \\ \left(power\left(x, \lfloor \frac{n}{2} \rfloor\right)\right)^2 & \text{if } n > 0 \text{ is even} \end{cases}$$

```
1  def power(x, n):
2    """Compute the value x**n for integer n."""
3    if n == 0:
4      return 1
5    else:
6      partial = power(x, n // 2)          # rely on truncated division
7      result = partial * partial
8      if n % 2 == 1:                      # if n odd, include extra factor of x
9        result *= x
10     return result
```

FURTHER EXAMPLES OF RECURSION

- Linear recursion
  - Raising a number x to an non-negative integer n
  - Power(x, n) = x^n
  - Can this be improved?
    - Let k = floor(n/2)
    - $(x^k)^2 = x^n$ (n is even), x^{n-1} (n is odd) = x. $(x^k)^2$
  - Complexity?

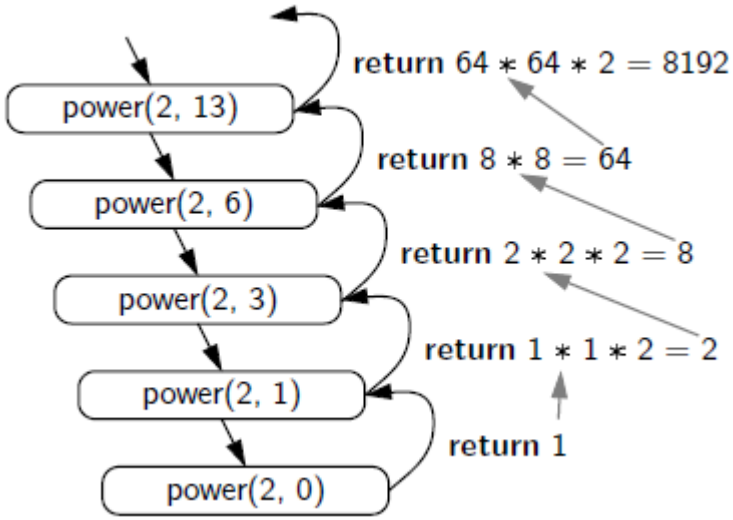
```
1  def power(x, n):
2    """Compute the value x**n for integer n."""
3    if n == 0:
4      return 1
5    else:
6      partial = power(x, n // 2)          # rely on truncated division
7      result = partial * partial
8      if n % 2 == 1:                      # if n odd, include extra factor of x
9        result *= x
10     return result
```



FURTHER EXAMPLES OF RECURSION

- Linear recursion
  - Raising a number x to an non-negative integer n
  - Power(x, n) = x^n
  - Can this be improved?
    - Let k = floor(n/2)
    - $(x^k)^2 = x^n$ (n is even), x^{n-1} (n is odd) = x. $(x^k)^2$
  - Complexity: O(log n)

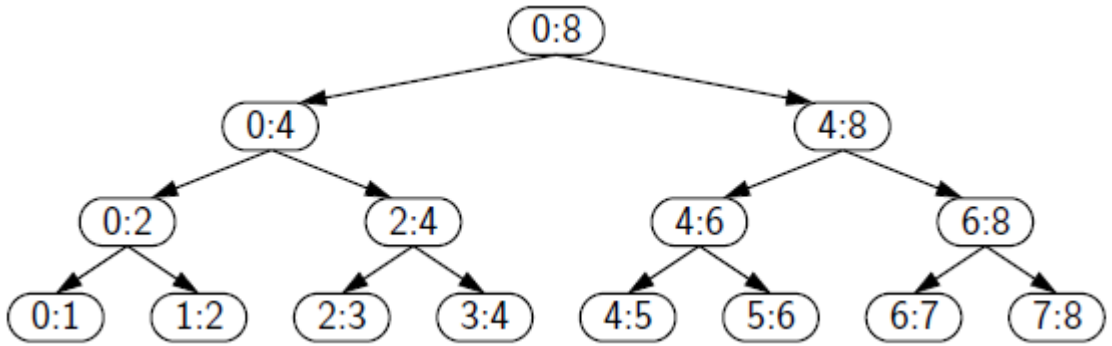
```
1  def power(x, n):
2    """Compute the value x**n for integer n."""
3    if n == 0:
4      return 1
5    else:
6      partial = power(x, n // 2)          # rely on truncated division
7      result = partial * partial
8      if n % 2 == 1:                      # if n odd, include extra factor of x
9        result *= x
10     return result
```



FURTHER EXAMPLES OF RECURSION

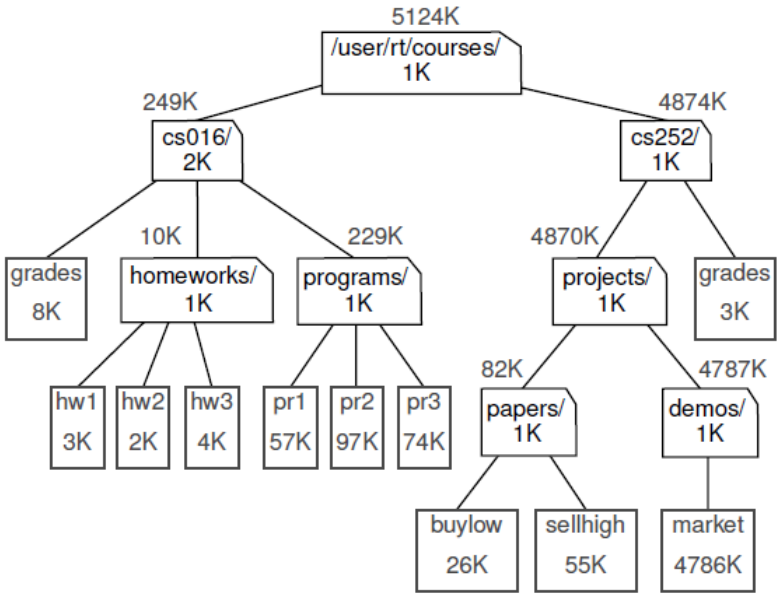
- Binary Recursion
  - The body of the function makes two recursive calls
  - English ruler
  - Bad_Fibonacci()
  - Summing n elements of a sequence
  - Depth: 1 + log n
  - Running time: O(n)

```
1  def binary_sum(S, start, stop):
2    """Return the sum of the numbers in implicit slice S[start:stop]."""
3    if start >= stop:                    # zero elements in slice
4      return 0
5    elif start == stop-1:                # one element in slice
6      return S[start]
7    else:                                # two or more elements in slice
8      mid = (start + stop) // 2
9      return binary_sum(S, start, mid) + binary_sum(S, mid, stop)
```



FURTHER EXAMPLES OF RECURSION

- Multiple Recursion
  - A function makes more than two recursive calls
  - File system disk space



Algorithm DiskUsage(path):
Input: A string designating a path to a file-system entry
Output: The cumulative disk space used by that entry and any nested entries
total = size(path) {immediate disk space used by the entry}
if path represents a directory **then**
for each child entry stored within directory path **do**
total = total + DiskUsage(child) {recursive call}
return *total*

DESIGNING RECURSIVE ALGORITHMS

- A recursion typically is made of:
  - Base case(s): at least one
    - Test input for base cases
    - Base case(s) should not use recursion
  - Recursive step(s):
    - Linear
    - Binary
    - Multiple
    - Progress towards the base case(s)
- Design of the function to facilitate recursion
  - binary_search(data, target)
  - binary_search(data, target, low, high)

DESIGNING RECURSIVE ALGORITHMS

- Eliminating tail recursion
  - Recursion
    - Must maintain activation records: keep track of the state
    - When memory is a limited resource
      - Use non-recursive algorithms
  - Use of stack
    - To convert recursive algorithm into a non-recursive one
    - Less memory footprint
- Tail recursion
  - If any recursive call made from one context is the very last operation in the context
  - With the computed value returned immediately
  - Must be a linear recursion

DESIGNING RECURSIVE ALGORITHMS

- Tail recursion
  - If any recursive call made from one context is the very last operation in the context
  - With the computed value returned immediately
  - Must be a linear recursion
  - Factorial function is NOT a tail recursion
    - return n*factorial(n-1)

```
1  def binary_search(data, target, low, high):
2    """ Return True if target is found in indicated portion of a Python list.
3
4    The search only considers the portion from data[low] to data[high] inclusive.
5    """
6    if low > high:
7      return False                          # interval is empty; no match
8    else:
9      mid = (low + high) // 2
10     if target == data[mid]:               # found a match
11       return True
12     elif target < data[mid]:
13       # recur on the portion left of the middle
14       return binary_search(data, target, low, mid - 1)
15     else:
16       # recur on the portion right of the middle
17       return binary_search(data, target, mid + 1, high)
```

DESIGNING RECURSIVE ALGORITHMS

- Tail recursion: converted to non-recursive algorithm

```
1  def binary_search(data, target, low, high):
2    """Return True if target is found in indicated portion of a Python list.
3
4    The search only considers the portion from data[low] to data[high] inclusive.
5    """
6    if low > high:
7      return False                          # interval is empty; no match
8    else:
9      mid = (low + high) // 2
10     if target == data[mid]:               # found a match
11       return True
12     elif target < data[mid]:
13       # recur on the portion left of the middle
14       return binary_search(data, target, low, mid - 1)
15     else:
16       # recur on the portion right of the middle
17       return binary_search(data, target, mid + 1, high)
```

```
1  def binary_search_iterative(data, target):
2    """Return True if target is found in the given Python list."""
3    low = 0
4    high = len(data)-1
5    while low <= high:
6      mid = (low + high) // 2
7      if target == data[mid]:               # found a match
8        return True
9      elif target < data[mid]:
10       high = mid - 1                      # only consider values left of mid
11     else:
12       low = mid + 1                       # only consider values right of mid
13   return False                            # loop ended without success
```

QUIZ FOR THIS WEEK

- Problem setting
  - Building of 100 floors
  - If marble (a glass ball) drops form the Nth floor or above, it breaks
  - If it is dropped from any floor below, it does not break
  - Find minimal # of drops to find N
  - You are allowed to break 2 marbles

QUIZ FOR THIS WEEK

- Immediately discard: binary search
- Most inefficient approach: from 0 to 100
- Approach
  - Left: try n and marble breaks
    - Try n-1 times worst case
  - Right: try n if marble does not break
    - Next floor to try: n + n-1
      - Why?

THANKS

See you in the next session!