Recursion

BBM 101 - Introduction to Programming I
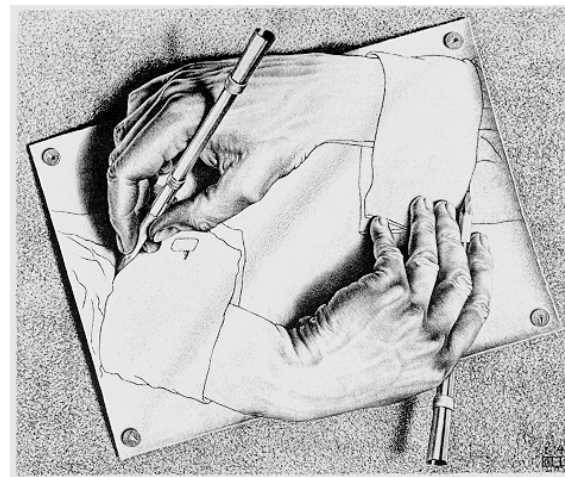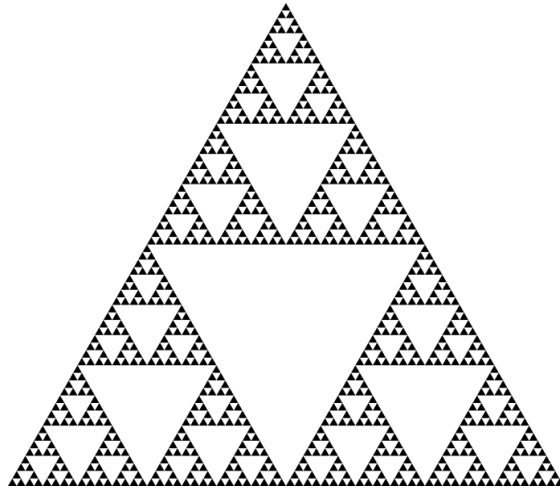
Hacettepe University
Fall 2016

Fuat Akal, Aykut Erdem, Erkut Erdem

Recursive functions

- A function is called **recursive** if the body of that function calls itself, either directly or indirectly.

- **Implication:** Executing the body of a recursive function may require applying that function

Drawing Hands, by M. C. Escher (lithograph, 1948)

Iterative algorithms

- Looping constructs (e.g. while or for loops) lead naturally to **iterative** algorithms

- Can conceptualize as capturing computation in a set of "state variables" which update on each iteration through the loop

Iterative multiplication by successive additions
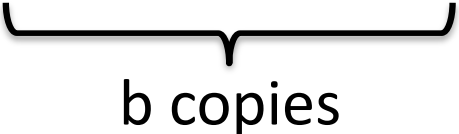
- Imagine we want to perform multiplication by successive additions:
  – To multiply a by b, add a to itself b times

- State variables:
  – i – iteration number; starts at b
  – result – current value of computation; starts at 0

- Update rules
  – i←i -1; stop when 0
  – result ← result + a

Iterative multiplication by successive additions
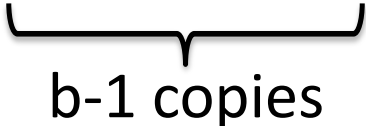
```python
def iterMul(a, b):
    result = 0
    while b > 0:
        result += a
        b -= 1
    return result
```

Recursive version

- An alternative is to think of this computation as:

$$a * b = \underbrace{a + a + \ldots + a}_{b \text{ copies}}$$

$$= \underbrace{a + a + \ldots + a}_{b\text{-}1 \text{ copies}}$$

$$= a + a * (b - 1)$$

Recursion

- This is an instance of a **recursive** algorithm
  - Reduce a problem to a simpler (or smaller) version of the same problem, plus some simple computations
    - **Recursive step**
  - Keep reducing until reach a simple case that can be solved directly
    - **Base case**
- a*b=a; if b=1 (Base case)
- a * b = a + a * (b-1); otherwise (Recursive case)

Recursive multiplication

```
def recurMul(a,b):
    if b == 1:
        return a
    else:
      return a + recurMul(a,b-1)
```

Let's try it out

```
def recurMul(a,b):
    if b == 1:
        return a
    else:
        return a +
    recurMul(a,b-1)
```

recurMul	→

Procedure4
  (a, b)
   if b == 1:
      return a
   else:
      return a +
recurMul(a, b-1)

Let's try it out

```
def recurMul(a,b):
    if b == 1:
        return a
    else:
        return a +
    recurMul(a,b-1)


recurMul(2,3)
```

recurMul	→

a	2
b	3

Procedure4
 (a, b)
  if b == 1:
     return a
  else:
     return a +
recurMul(a, b-1)

Let's try it out

```
def recurMul(a,b):
    if b == 1:
        return a
    else:
        return a +
    recurMul(a,b-1)


recurMul(2,3)
```

recurMul	

a	2
b	3

Procedure4
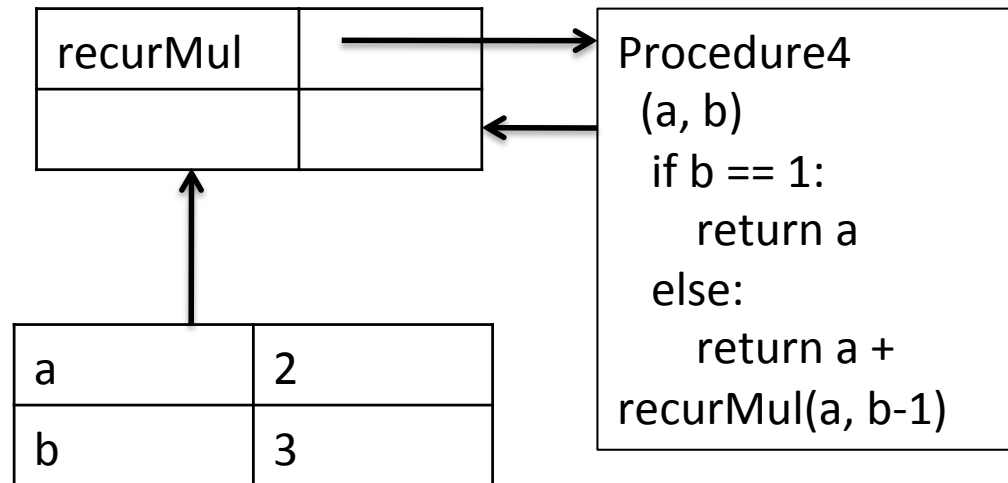 (a, b)
   if b == 1:
      return a
   else:
      return a +
recurMul(a, b-1)

Let's try it out

```
def recurMul(a,b):
    if b == 1:
        return a
    else:
        return a +
    recurMul(a,b-1)


recurMul(2,3)
```
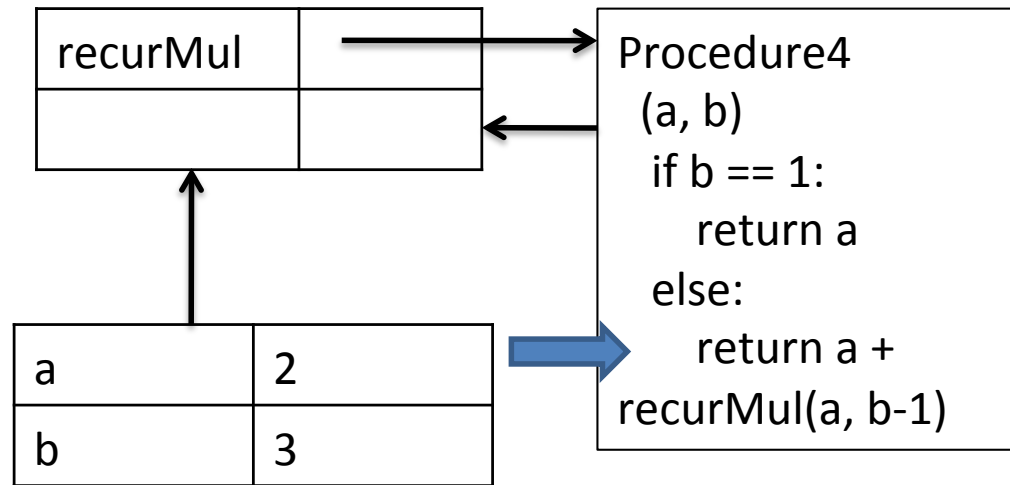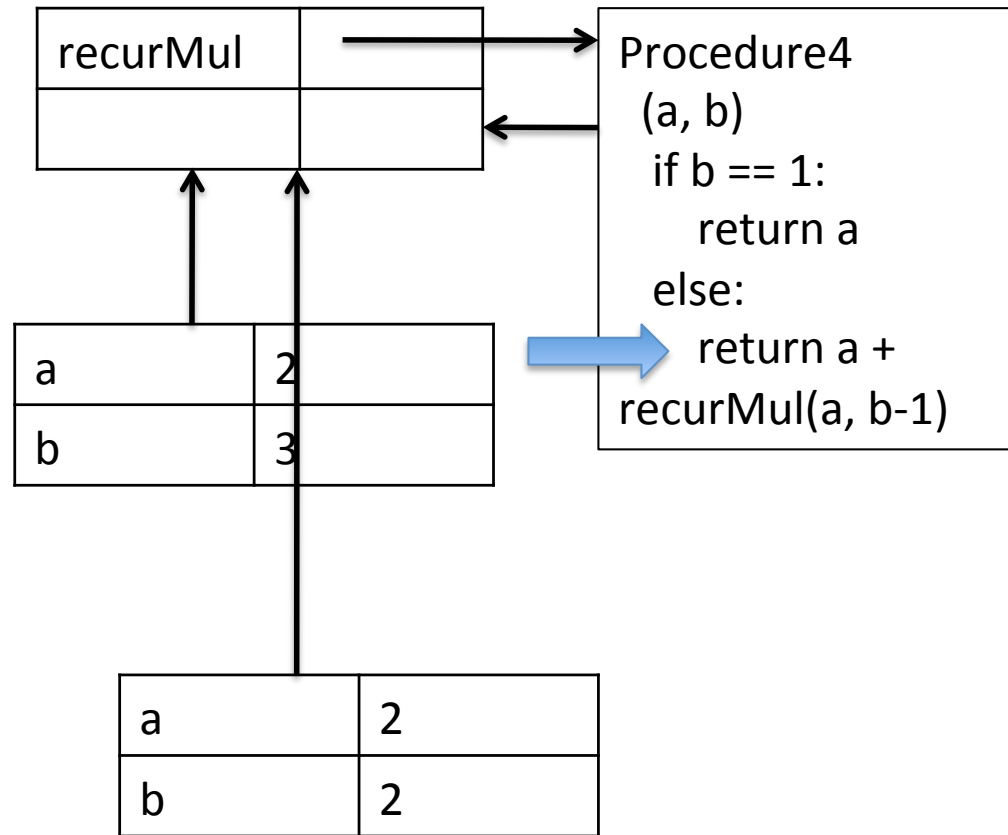
recurMul	

Procedure4
(a, b)
if b == 1:
return a
else:
return a +
recurMul(a, b-1)

a	2
b	3

a	2
b	2

Let's try it out

```
def recurMul(a,b):
    if b == 1:
        return a
    else:
        return a +
    recurMul(a,b-1)


recurMul(2,3)
```

recurMul	→

Procedure4
(a, b)
if b == 1:
return a
else:
→ return a +
recurMul(a, b-1)

a	2
b	3

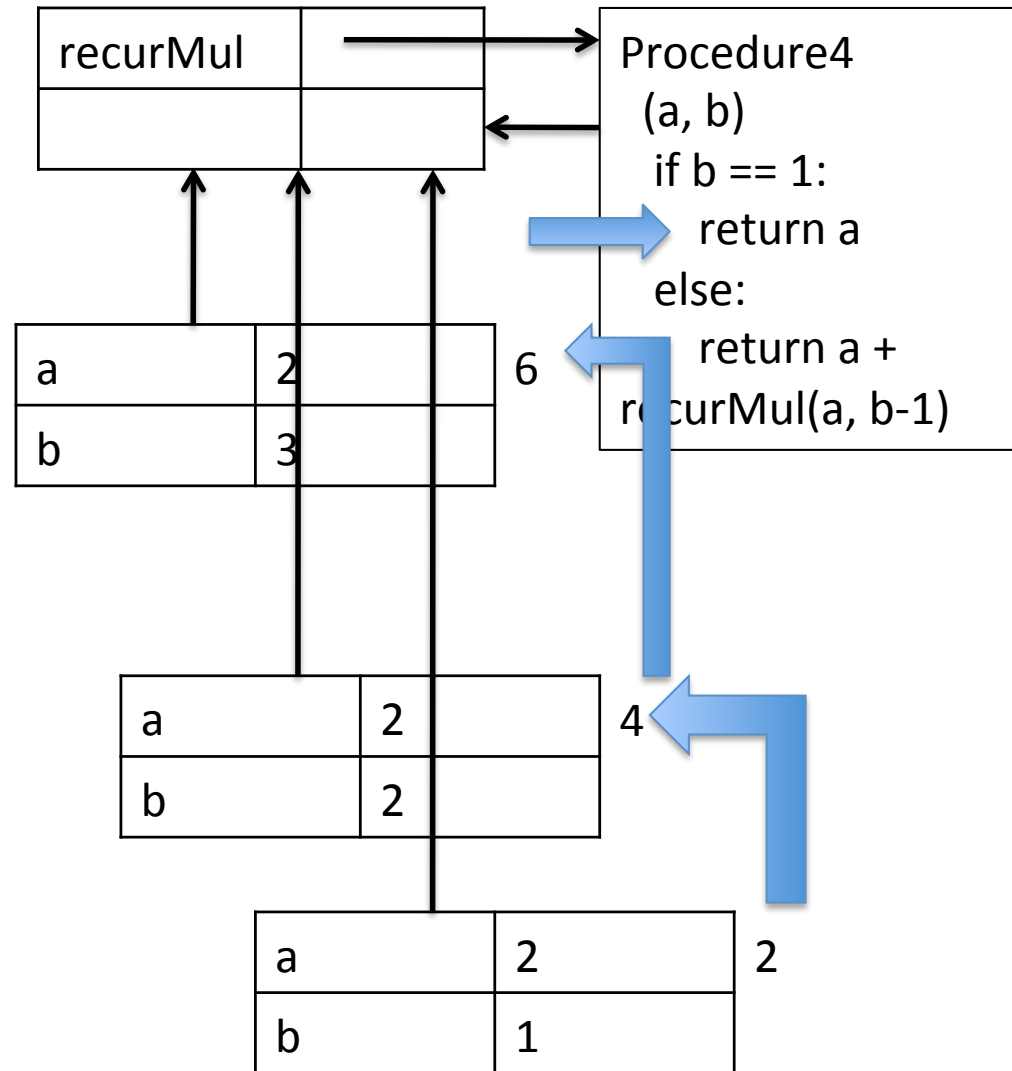| a	2	
b	2	

a	2
b	1

Let's try it out

```
def recurMul(a,b):
    if b == 1:
        return a
    else:
        return a +
    recurMul(a,b-1)


recurMul(2,3)
```

recurMul	→

Procedure4
  (a, b)
   if b == 1:
     return a
   else:
     return a +
 recurMul(a, b-1)

a	2		6
b	3		

a	2		4
b	2		

a	2		2
b	1		

The Anatomy of a Recursive Function

- The def statement header is similar to other functions

- Conditional statements check for base cases

- Base cases are evaluated without recursive calls

- Recursive cases are evaluated with recursive calls

```
def recurMul(a,b):
    if b == 1:
        return a
    else:
        return a + recurMul(a,b-1)
```

Inductive reasoning

- How do we know that our recursive code will work?

- iterMul terminates because b is initially positive, and decrease by 1 each time around loop; thus must eventually become less than 1

- recurMul called with b = 1 has no recursive call and stops

- recurMul called with b > 1 makes a recursive call with a smaller version of b; must eventually reach call with b = 1

Mathematical induction

- To prove a statement indexed on integers is true for all values of n:

  - Prove it is true when n is smallest value (e.g. n = 0 or n = 1)

  - Then prove that if it is true for an arbitrary value of n, one can show that it must be true for n+1

Example

- 0+1+2+3+…+n=(n(n+1))/2
- Proof
  - If n = 0, then LHS is 0 and RHS is 0*1/2 = 0, so true
  - Assume true for some k, then need to show that
    - 0 + 1 + 2 + … + k + (k+1) = ((k+1)(k+2))/2
    - LHS is k(k+1)/2 + (k+1) by assumption that property holds for problem of size k
    - This becomes, by algebra, ((k+1)(k+2))/2
  - Hence expression holds for all n >= 0

What does this have to do with code?

- Same logic applies

```
def recurMul(a, b):
    if b == 1:
        return a
    else:
        return a + recurMul(a, b-1)
```

- Base case, we can show that recurMul must return correct answer
- For recursive case, we can assume that recurMul correctly returns an answer for problems of size smaller than b, then by the addition step, it must also return a correct answer for problem of size b
- Thus by induction, code correctly returns answer

Sum digits of a number

```python
def split(n):

    """Split positive n into all but its last digit and its last digit."""

    return n // 10, n % 10


def sum_digits(n):

    """Return the sum of the digits of positive integer n."""

    if n < 10:

        return n

    else:

        all_but_last, last = split(n)

        return sum_digits(all_but_last) + last
```

<u>Verify the correctness of this recursive definition.</u>

Some observations

- Each recursive call to a function creates its own environment, with local scoping of variables

- Bindings for variable in each frame distinct, and not changed by recursive call

- Flow of control will pass back to earlier frame once function call returns value

The "classic" recursive problem

- Factorial

n! = n * (n-1) * ... * 1

$$= \begin{cases} 1 & \text{if } n = 0 \\ n * (n-1)! & \text{otherwise} \end{cases}$$
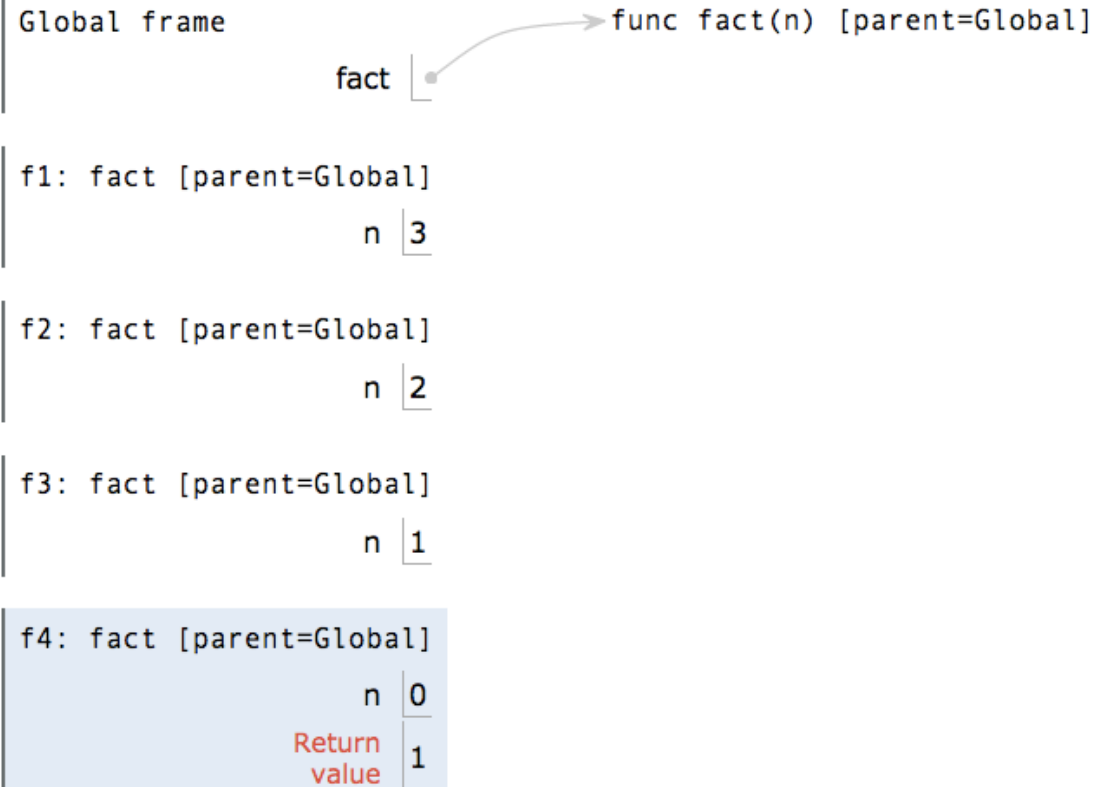
Recursion in Environment Diagrams

```
1  def fact(n):
2      if n == 0:
3          return 1
4      else:
5          return n * fact(n-1)
6
7  fact(3)
```

Recursion in Environment Diagrams

```
1  def fact(n):
2      if n == 0:
3          return 1
4      else:
5          return n * fact(n-1)
6
7  fact(3)
```

(Demo)

Recursion in Environment Diagrams

```
1  def fact(n):
2      if n == 0:
3          return 1
4      else:
5          return n * fact(n-1)
6
7  fact(3)
```

(Demo)

- The same function **fact** is called multiple times

Global frame

fact → func fact(n) [parent=Global]

f1: fact [parent=Global]
    n 3

f2: fact [parent=Global]
    n 2

f3: fact [parent=Global]
    n 1

f4: fact [parent=Global]
    n 0
    Return value 1

Recursion in Environment Diagrams

```
1  def fact(n):
2      if n == 0:
3          return 1
4      else:
5          return n * fact(n-1)
6
7  fact(3)
```

(Demo)

- The same function **fact** is called multiple times
- Different frames keep track of the different arguments in each call

Global frame

fact → func fact(n) [parent=Global]

f1: fact [parent=Global]
n 3

f2: fact [parent=Global]
n 2

f3: fact [parent=Global]
n 1
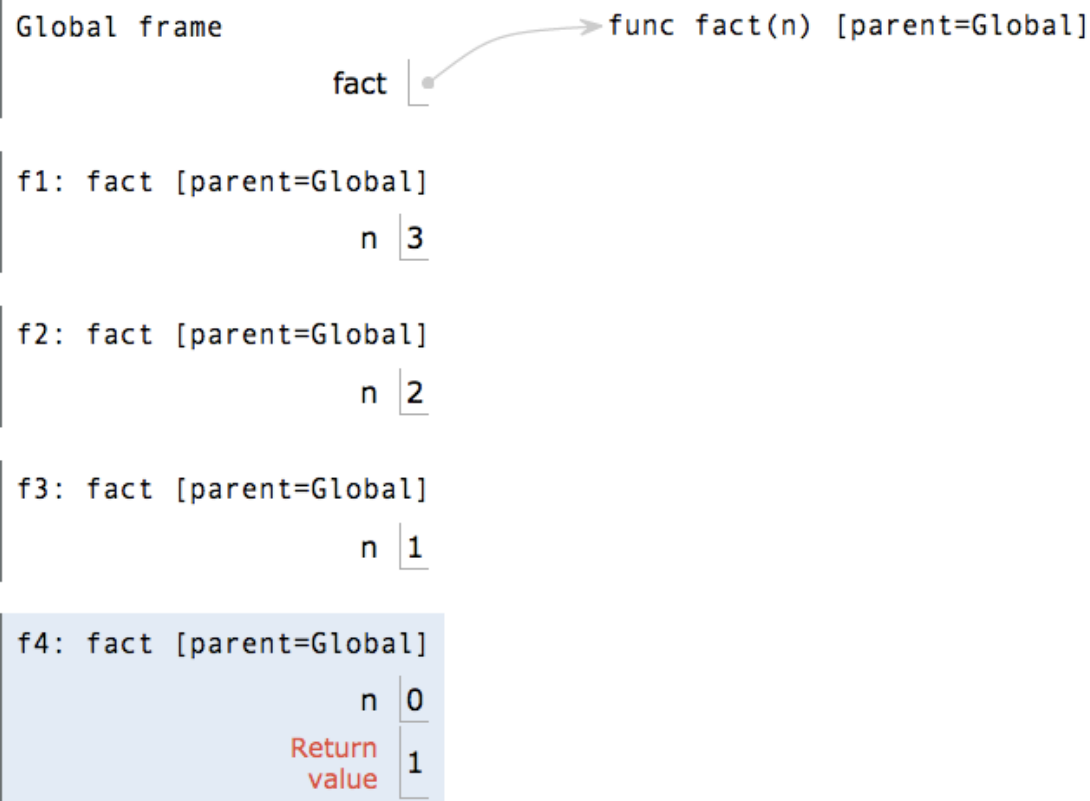
f4: fact [parent=Global]
n 0
Return value 1

Recursion in Environment Diagrams

```
1  def fact(n):
2      if n == 0:
3          return 1
4      else:
5          return n * fact(n-1)
6
7  fact(3)
```

(Demo)

- The same function **fact** is called multiple times
- Different frames keep track of the different arguments in each call
- What **n** evaluates to depends upon the current environment
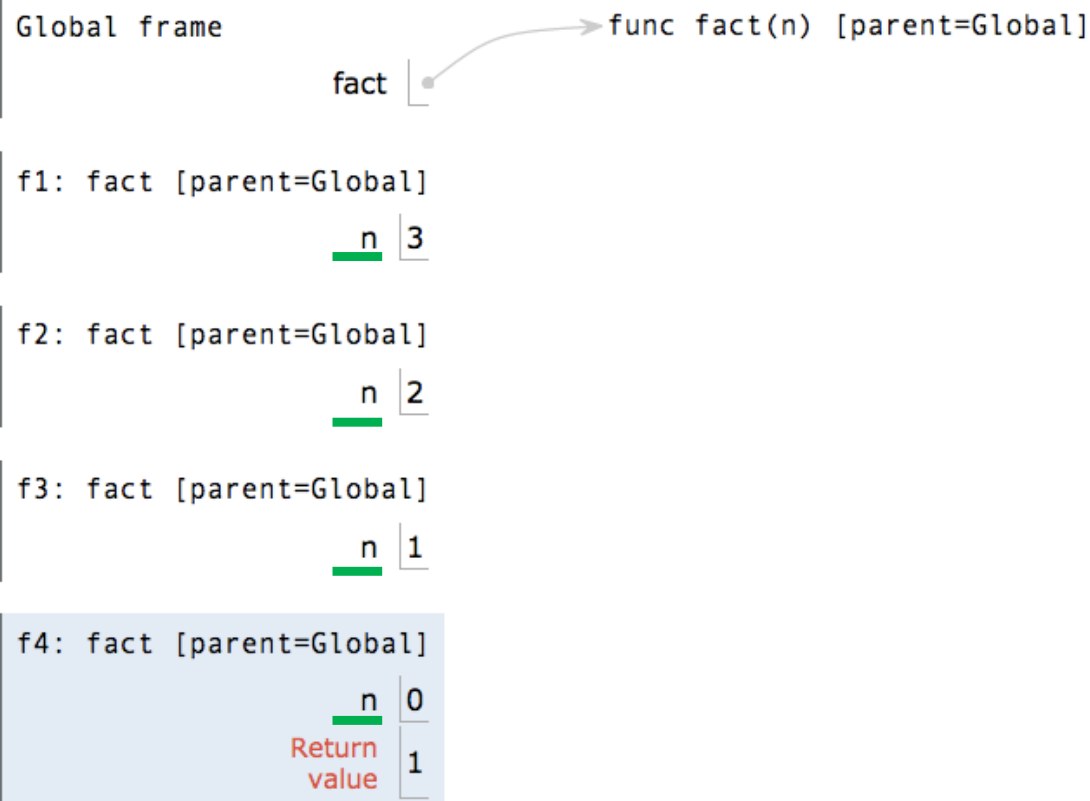
Recursion in Environment Diagrams

```
1  def fact(n):
2      if n == 0:
3          return 1
4      else:
5          return n * fact(n-1)
6
7  fact(3)
```

(Demo)

- The same function **fact** is called multiple times

- Different frames keep track of the different arguments in each call

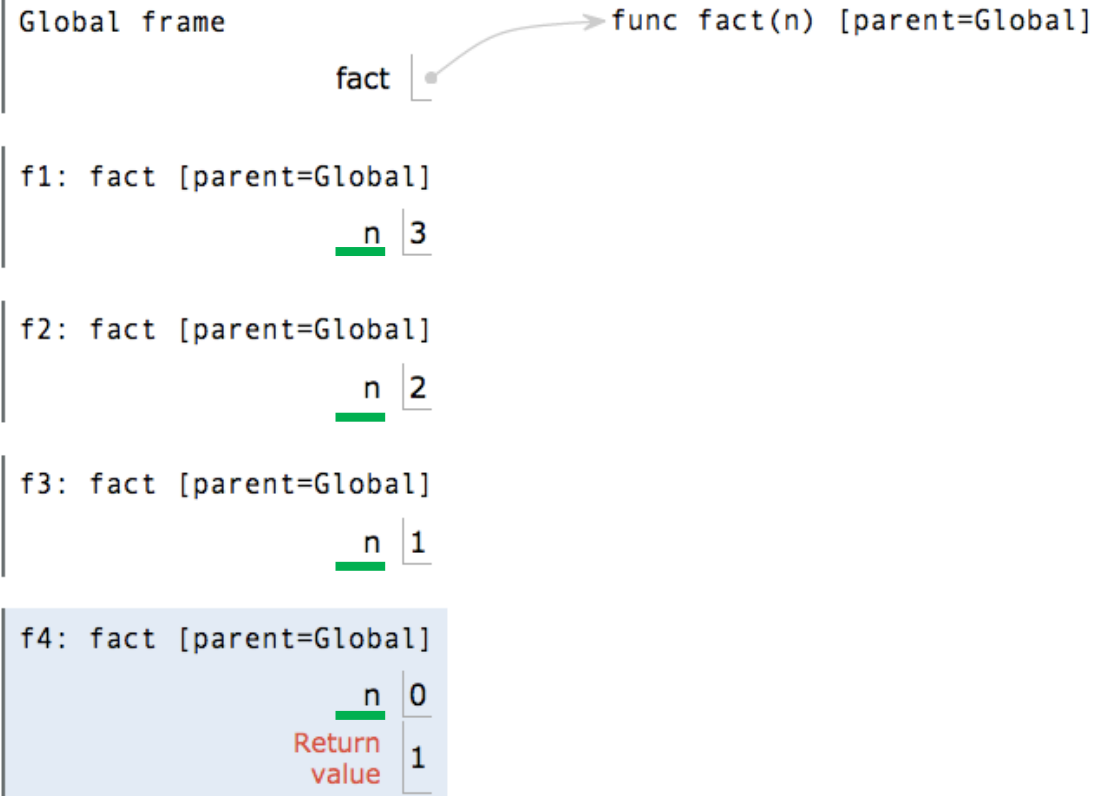- What **n** evaluates to depends upon the current environment

- Each call to **fact** solves a simpler problem than the last: smaller **n**

Iteration vs Recursion

$$4! = 4 \cdot 3 \cdot 2 \cdot 1 = 24$$

$$n! = \prod_{k=1}^{n} k$$

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot (n-1)! & \text{otherwise} \end{cases}$$

Iteration vs Recursion

$$4! = 4 \cdot 3 \cdot 2 \cdot 1 = 24$$

Using while:

```python
def fact_iter(n):
    total, k = 1, 1
    while k <= n:
        total, k = total*k, k+1
    return total
```

Math:

$$n! = \prod_{k=1}^{n} k$$

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot (n-1)! & \text{otherwise} \end{cases}$$

Names: `n, total, k, fact_iter`

Iteration vs Recursion

$$4! = 4 \cdot 3 \cdot 2 \cdot 1 = 24$$

Using while:

```python
def fact_iter(n):
    total, k = 1, 1
    while k <= n:
        total, k = total*k, k+1
    return total
```

Using recursion:

```python
def fact(n):
    if n == 0:
        return 1
    else:
        return n * fact(n-1)
```

Math:

$$n! = \prod_{k=1}^{n} k$$

$$n! = \prod_{k=1}^{n} k$$

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot (n-1)! & \text{otherwise} \end{cases}$$

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot (n-1)! & \text{otherwise} \end{cases}$$

Names: total, k, fact_iter

n, fact

Recursion on non-numerics

- How could we check whether a string of characters is a palindrome, i.e., reads the same forwards and backwards
  - "Able was I ere I saw Elba" – attributed to Napolean
  - "Are we not drawn onward, we few, drawn onward to new era?"
  - "Ey Edip Adana'da pide ye"

How to we solve this recursive?

- First, convert the string to just characters, by stripping out punctuation, and converting upper case to lower case

- Then
  - Base case: a string of length 0 or 1 is a palindrome
  - Recursive case:
    - If first character matches last character, then is a palindrome if middle section is a palindrome

Example

- 'Able was I ere I saw Elba'→ 'ablewasiereisawleba'
- isPalindrome('ablewasiereisawleba') is same as
  - 'a' == 'a' and isPalindrome('blewasiereisawleb')

Palindrome or not?

```python
def toChars(s):
    s = s.lower()
    ans = ''
    for c in s:
        if c in 'abcdefghijklmnopqrstuvwxyz':
            ans = ans + c
    return ans

def isPal(s):
    if len(s) <= 1:
        return True
    else:
        return s[0] == s[-1] and isPal(s[1:-1])

def isPalindrome(s):
    return isPal(toChars(s))
```

Divide and conquer

- This is an example of a "divide and conquer" algorithm
  - Solve a hard problem by breaking it into a set of sub-problems such that:
  - Sub-problems are easier to solve than the original
  - Solutions of the sub-problems can be combined to solve the original

Global variables

- Suppose we wanted to count the number of times fib calls itself recursively

- Can do this using a global variable

- So far, all functions communicate with their environment through their parameters and return values

- But, (though a bit dangerous), can declare a variable to be global – means name is defined at the outermost scope of the program, rather than scope of function in which appears

Example

```
def fibMetered(x):
    global numCalls
    numCalls += 1
    if x == 0 or x == 1:
        return 1
    else:
        return fibMetered(x-1) + fibMetered(x-2)

def testFib(n):
    for i in range(n+1):
        global numCalls
        numCalls = 0
        print('fib of '+str(i)+' = '+str(fibMetered(i)))
        print('fib called ' + str(numCalls) + ' times')
```

Global variables

- Use with care!!
- Destroy locality of code
- Since can be modified or read in a wide range of places, can be easy to break locality and introduce bugs!!

Mutual recursion

- **Mutual recursion** is a form of **recursion** where two functions or data types are **defined** in terms of each other.

The Luhn Algorithm

- A simple checksum formula used to validate a variety of identification numbers, such as credit card numbers, IMEI numbers, etc.

The Luhn Algorithm

- From Wikipedia: http://en.wikipedia.org/wiki/Luhn_algorithm
- **First:** From the rightmost digit, which is the check digit, moving left, double the value of every second digit; if product of this doubling operation is greater than 9 (e.g., 7 * 2 = 14), then sum the digits of the products (e.g., 10: 1 + 0 = 1, 14: 1 + 4 = 5)
- **Second:** Take the sum of all the digits

1	3	8	7	4	3	
2	3	1+6=7	7	8	3	= 30

- The Luhn sum of a valid credit card number is a multiple of 10

The Luhn Algorithm

```python
def luhn_sum(n):
    """Return the digit sum of n computed by the Luhn algorithm"""
    if n < 10:
        return n
    else:
        all_but_last, last = split(n)
        return luhn_sum_double(all_but_last) + last


def luhn_sum_double(n):
    """Return the Luhn sum of n, doubling the last digit."""
    all_but_last, last = split(n)
    luhn_digit = sum_digits(2 * last)
    if n < 10:
        return luhn_digit
    else:
        return luhn_sum(all_but_last) + luhn_digit
```

Tree Recursion

- Tree-shaped processes arise whenever executing the body of a recursive function makes more than one recursive call.

Tree Recursion

- Fibonacci numbers
- Leonardo of Pisa (aka Fibonacci) modeled the following challenge
  - Newborn pair of rabbits (one female, one male) are put in a pen
  - Rabbits mate at age of one month
  - Rabbits have a one month gestation period
  - Assume rabbits never die, that female always produces one new pair (one male, one female) every month from its second month on.
  - How many female rabbits are there at the end of one year?

Fibonacci

- After one month (call it 0) – 1 female

- After second month – still 1 female (now pregnant)

- After third month – two females, one pregnant, one not

- In general, females(n) = females(n-1) + females(n-2)

  - Every female alive at month n-2 will produce one female in month n;

  - These can be added those alive in month n-1 to get total alive in month n

Month	Females
0	1
1	1
2	2
3	3
4	5
5	8
6	13

Fibonacci

- Base cases:
  - Females(0) = 1
  - Females(1) = 1
- Recursive case
  - Females(n) = Females(n-1) + Females(n-2)

Fibonacci

```
def fib(n):
    """assumes n an int >= 0
    returns Fibonacci of n"""
    assert type(n) == int and n >= 0
    if n == 0:
        return 1
    elif n == 1:
        return 1
    else:
        return fib(n-2) + fib(n-1)
```

A tree-recursive process

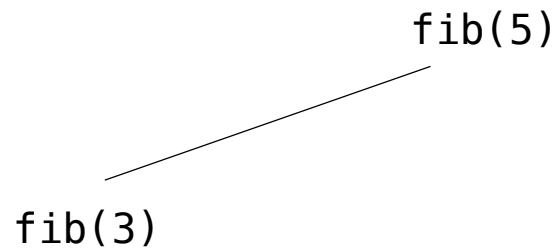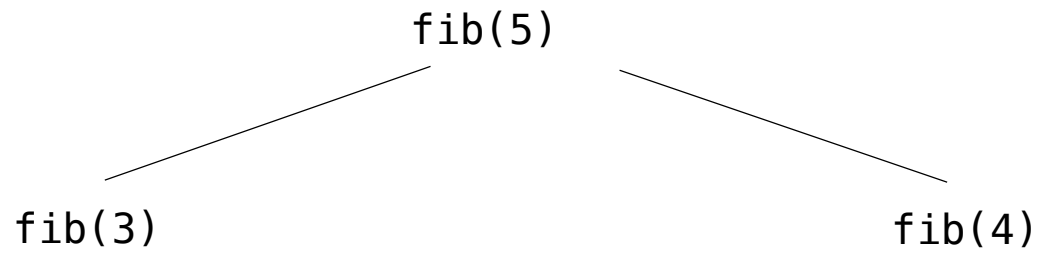- The computational process of fib evolves into a tree structure

A tree-recursive process

- The computational process of fib evolves into a tree structure
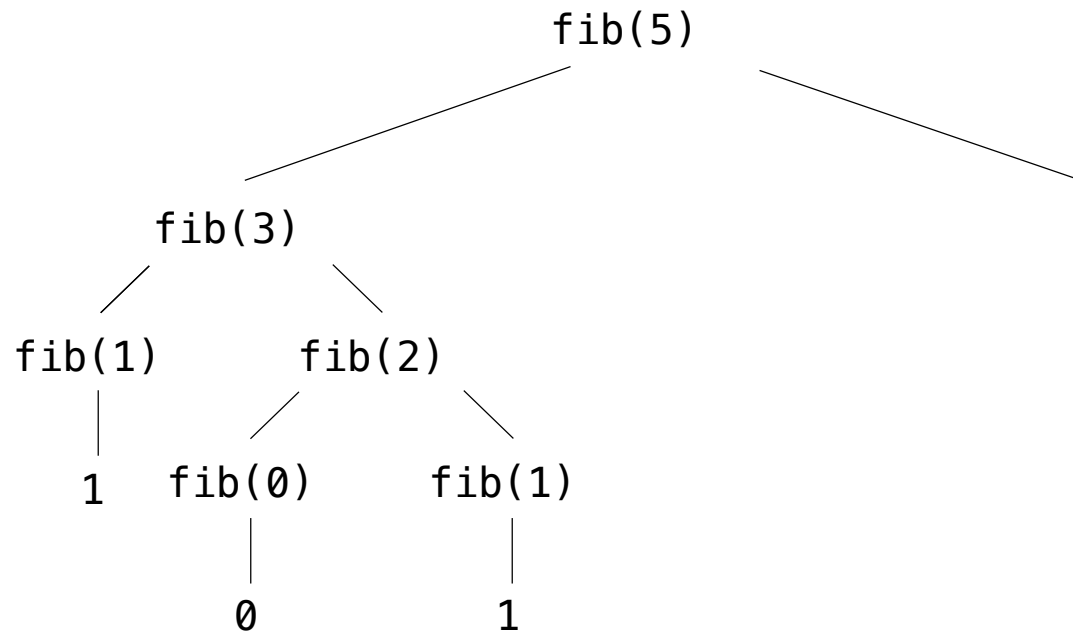
A tree-recursive process

- The computational process of fib evolves into a tree structure

A tree-recursive process

- The computational process of fib evolves into a tree structure

A tree-recursive process

- The computational process of fib evolves into a tree structure

```
                          fib(5)
                 
          fib(3)
         /      \
    fib(1)      fib(2)
       |         /    \
       1     fib(0)   fib(1)
                |        |
                0        1
```

A tree-recursive process

- The computational process of fib evolves into a tree structure

A tree-recursive process

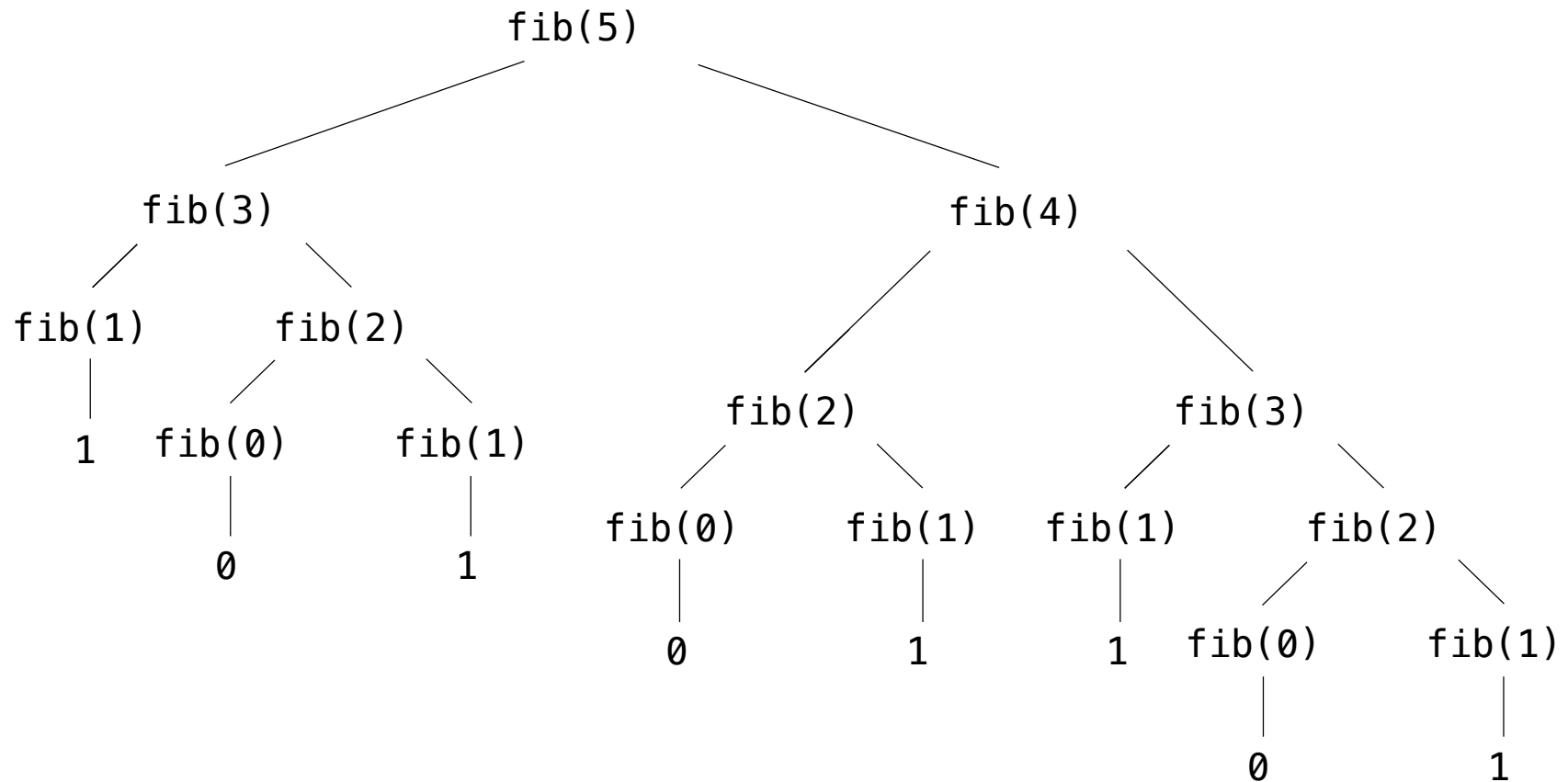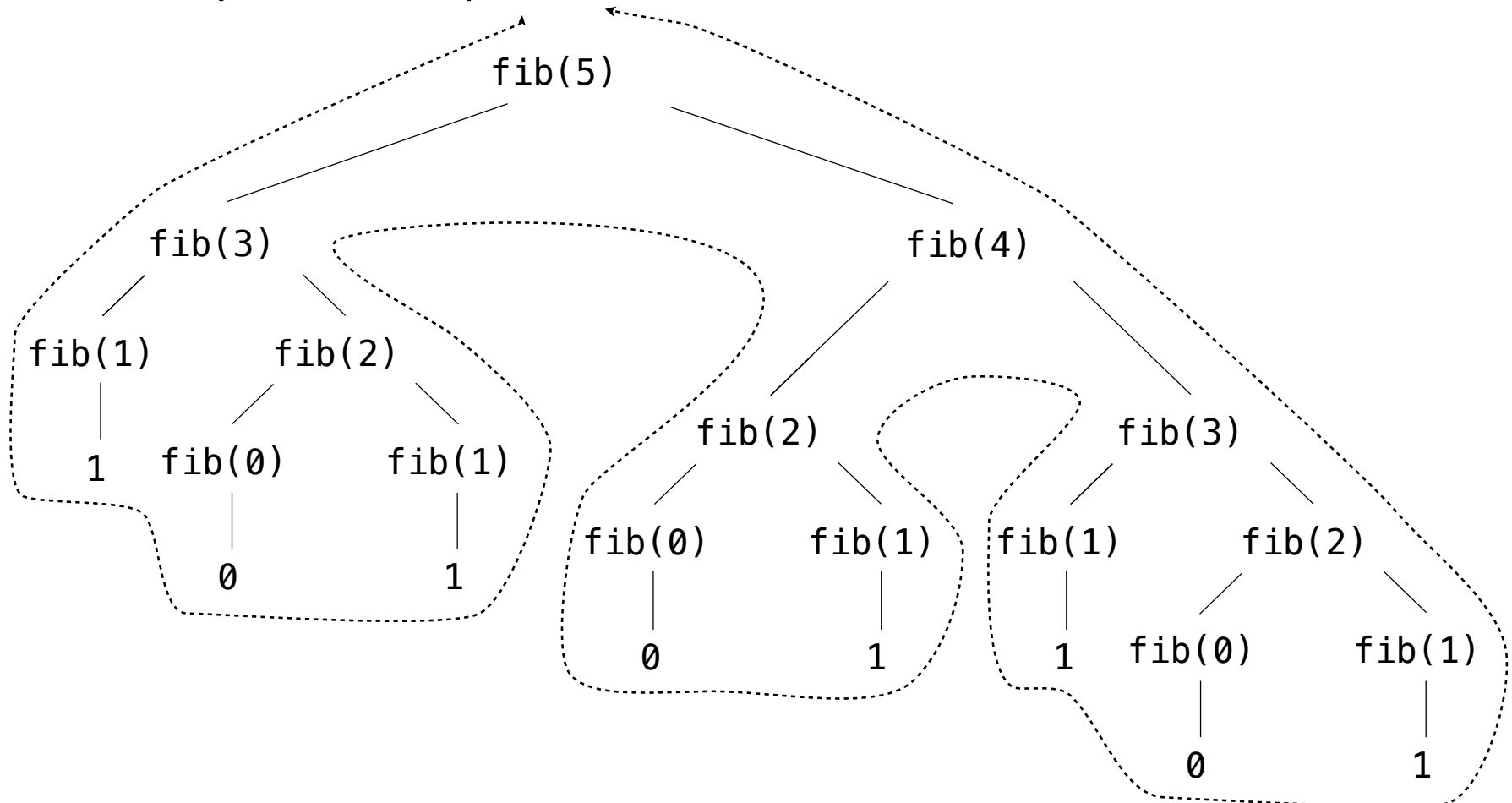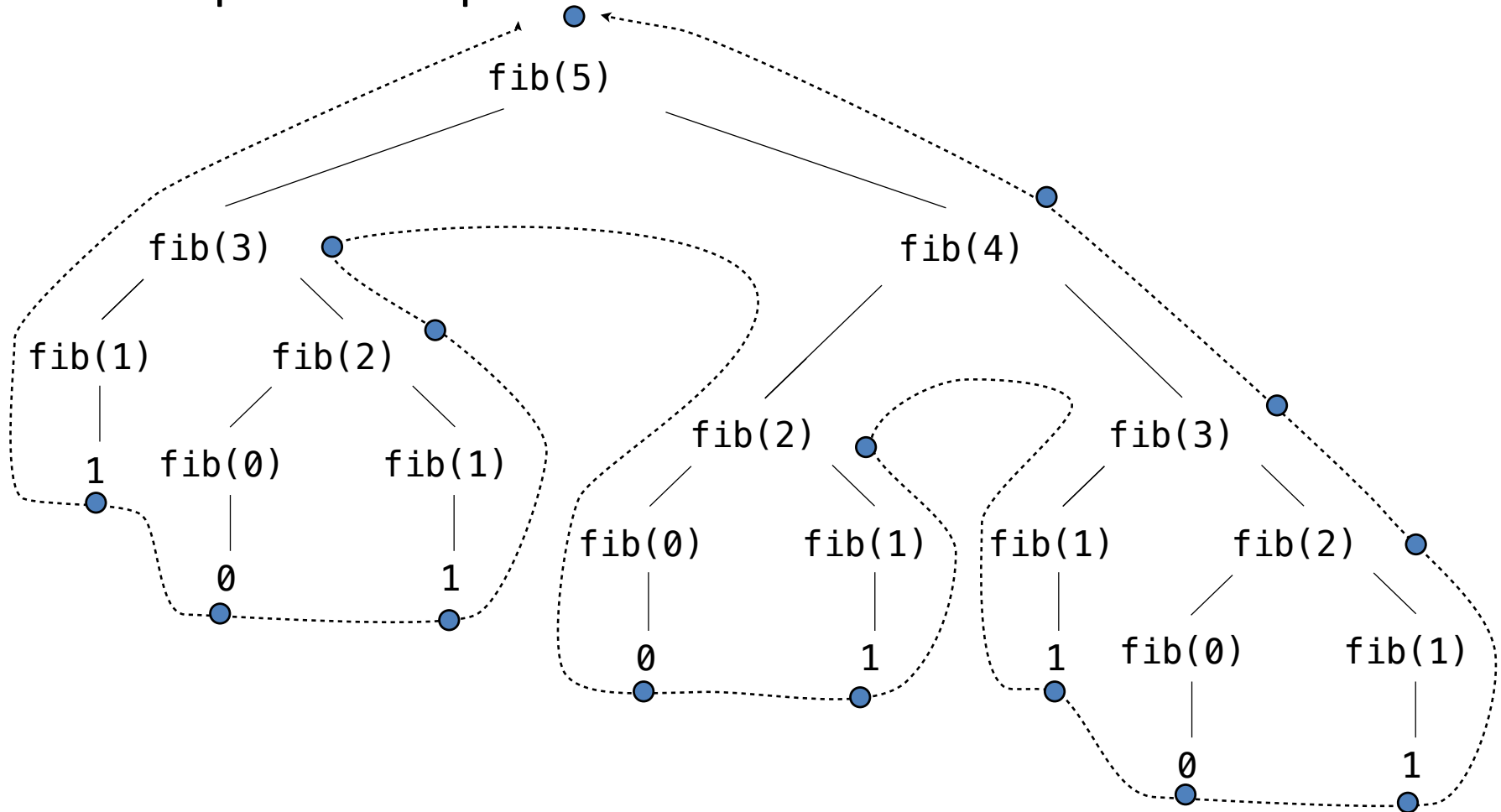- The computational process of fib evolves into a tree structure

A tree-recursive process

- The computational process of fib evolves into a tree structure

Pitfalls of Recursion

- With recursion, you can compose compact and elegant programs that fail spectacularly at runtime.

- Missing base case

- No guarentee of convergence

- Excessive space requirements

- Excessive recomputation

Missing base case

```
def H(n):
    return H(n-1) + 1.0/n;
```

- This recursive function is supposed to compute Harmonic numbers, but is missing a base case.
- If you call this function, it will repeatedly call itself and never return.

No guarantee of convergence

```
def H(n):
    if n == 1:
        return 1.0
    return H(n) + 1.0/n
```

- This recursive function will go into an infinite recursive loop if it is invoked with an argument n having any value other than 1.

- Another common problem is to include within a recursive function a recursive call to solve a subproblem that is not smaller.
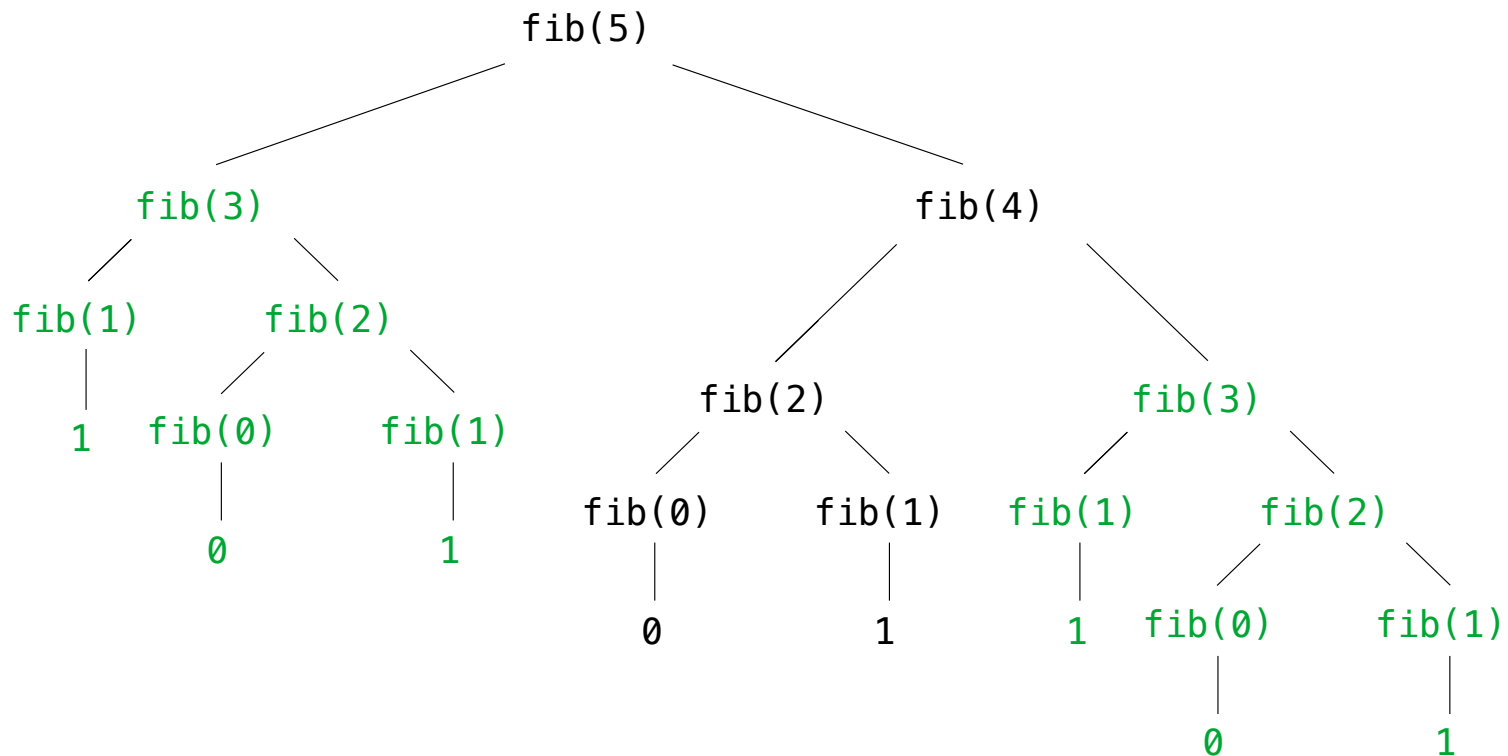
Excessive space requirements

- Python needs to keep track of each recursive call to implement the function abstraction as expected.

- If a function calls itself recursively an excessive number of times before returning, the space required by Python for this task may be prohibitive.

```
def H(n):
    if n == 0:
        return 0.0
    return H(n-1) + 1.0/n
```

- This recursive function correctly computes the nth harmonic number.

- However, we cannot use it for large *n* because the recursive depth is proportional to *n*, and this creates a StackOverflowError.

Excessive recomputation

- A simple recursive program might require exponential time (unnecessarily), due to excessive recomputation.

- For example, fib is called on the same argument multiple time

```
                              fib(5)
              /                             \
         fib(3)                               fib(4)
        /      \                           /          \
   fib(1)    fib(2)                   fib(2)            fib(3)
      |      /    \                  /     \           /      \
      1  fib(0) fib(1)           fib(0)  fib(1)    fib(1)   fib(2)
            |      |                |       |         |       /    \
            0      1                0       1         1   fib(0)  fib(1)
                                                             |       |
                                                             0       1
```

Computational Complexity of Recursive Algorithms: Linear Complexity

- Complexity can depend on number of recursive calls

```
def fact(n):
    if n == 1:
        return 1
    else:
        return n*fact(n-1)
```

- Number of recursive calls?
  - Fact(n), then fact(n-1), etc. until get to fact(1)
  - Complexity of each call is constant
  - *O(n)*

Computational Complexity of Recursive Algorithms: Exponential Complexity

```python
def genSubsets(L):
    res = []
    if len(L) == 0:
        return [[]] #list of empty list
    smaller = genSubsets(L[:-1])
    # get all subsets without last element
    extra = L[-1:]
    # create a list of just last element
    new = []
    for small in smaller:
        new.append(small+extra)
    # for all smaller solutions, add one with last element
    return smaller+new
    # combine those with last element and those without
```

Computational Complexity of Recursive Algorithms: Exponential Complexity

```python
def genSubsets(L):
    res = []
    if len(L) == 0:
        return [[]]
    smaller = genSubsets(L[:-1])
    extra = L[-1:]
    new = []
    for small in smaller:
        new.append(small+extra)
    return smaller+new
```

- Assuming append is constant time

- Time includes time to solve smaller problem, plus time needed to make a copy of all elements in smaller problem

Computational Complexity of Recursive Algorithms: Exponential Complexity

```
def genSubsets(L):
    res = []
    if len(L) == 0:
        return [[]]
    smaller = genSubsets(L[:-1])
    extra = L[-1:]
    new = []
    for small in smaller:
        new.append(small+extra)
    return smaller+new
```

- But important to think about size of smaller

- Know that for a set of size k there are 2^k cases

- So to solve need $2^{n-1} + 2^{n-2} + \ldots + 2^0$ steps

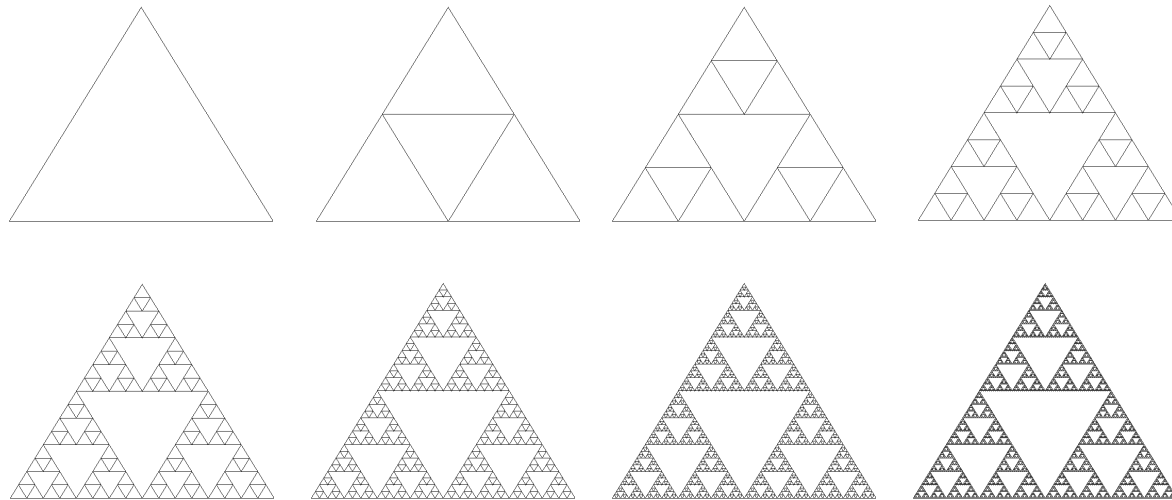- Math tells us this is $O(2^n)$

Recursive Graphics

- Simple recursive drawing schemes can lead to pictures that are remarkably intricate – **Fractals**

- For example, an *H-tree of order n* is defined as follows:
  - The base case is null for $n = 0$.
  - The reduction step is to draw, within the unit square three lines in the shape of the letter H four H-trees of order n-1.
  - One connected to each tip of the H with the additional provisos that the H-trees of order n-1 are centered in the four quadrants of the square, halved in size.
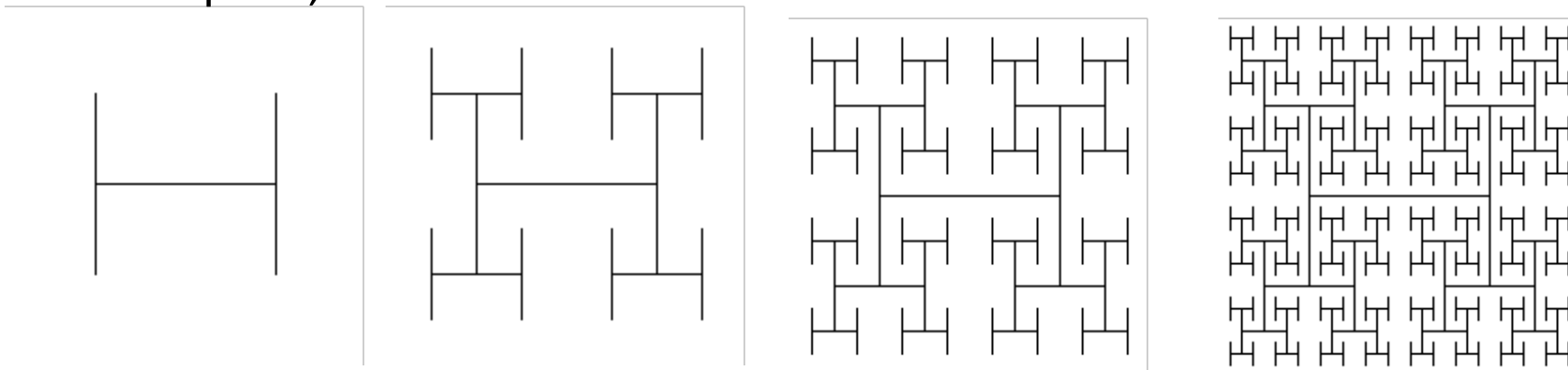
More recursive graphics

- Sierpinski triangles

- Recursive trees