The Art of Data Structures

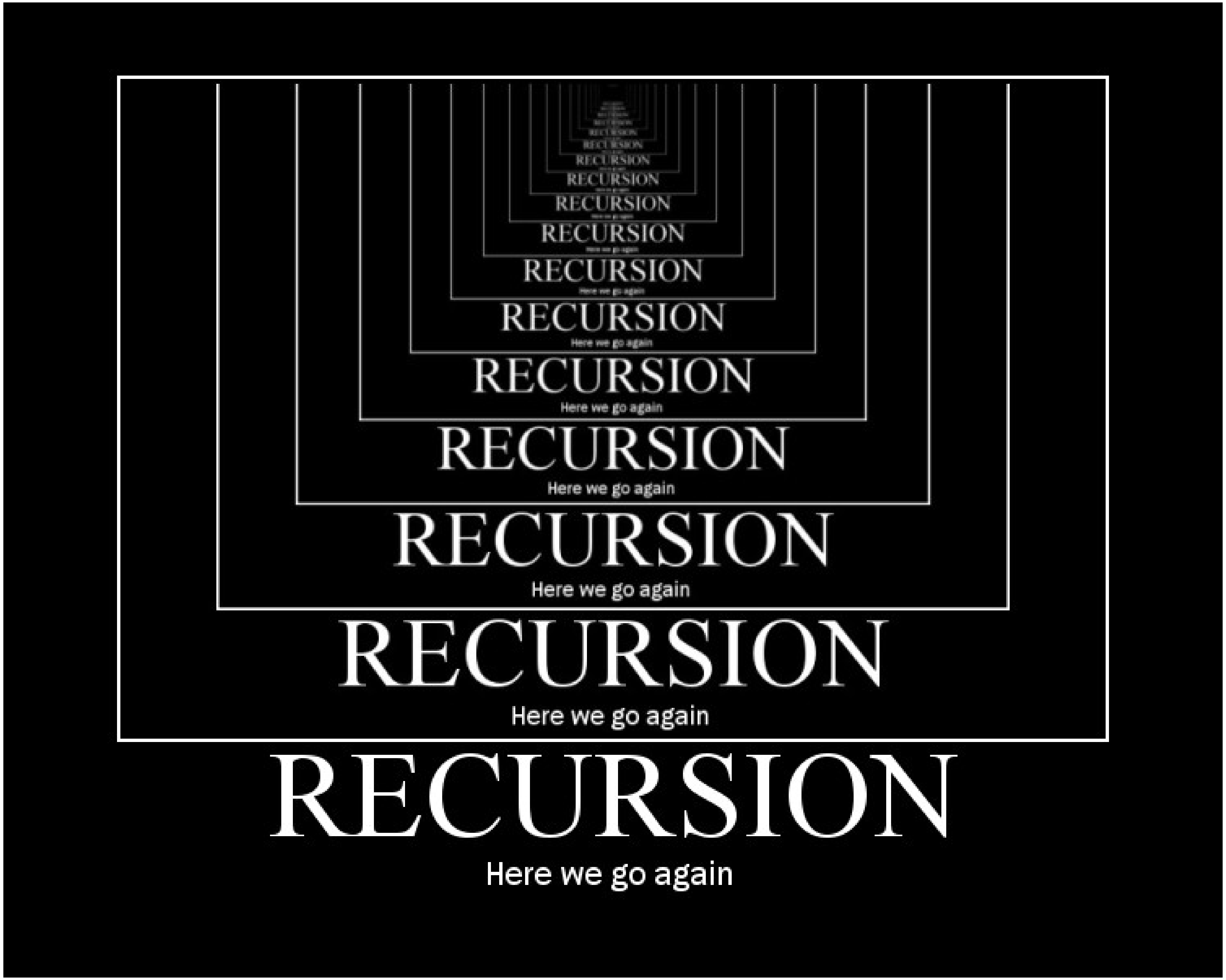
Recursion

Alan Beadle
CSC 162: The Art of Data Structures
UNIVERSITY of ROCHESTER

Agenda

- To understand that complex, difficult problems that may have a simple recursive solutions
- To learn how to formulate programs recursively
- To understand and apply the three laws of recursion
- To understand recursion as a form of iteration
- To implement the recursive formulation of a problem
- To understand how recursion is implemented by a computer system

Recursion


RECURSION
Here we go again
RECURSION
Here we go again
RECURSION
Here we go again
RECURSION
Here we go again
RECURSION
Here we go again
RECURSION
Here we go again

Recursion

- A description of something that refers to itself is called a *recursive definition (like this one for example)*

Recursion

- In mathematics, recursion is frequently used
- The most common example is the factorial:

  For example, 5! = 5(4)(3)(2)(1)

Recursion

- In other words,

$$n! = n(n-1)!$$

- Or

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

- This definition says that `0!` is `1`, while the factorial of any other number is that number times the factorial of one less than that number

Recursion

- Our definition is recursive, but definitely not circular. Consider 4!
  - 4! = 4(4-1)! = 4(3!)
  - What is 3!? We apply the definition again
    4! = 4(3!) = 4[3(3-1)!] = 4(3)(2!)
    4! = 4(3!) = 4(3)(2!) = 4(3)(2)(1!) = 4(3)(2)(1)(0!) = 4(3)(2)(1)(1) = 24

Recursion

- Factorial is not circular because we eventually get to 0!, whose definition does not rely on the definition of factorial and is just 1
- This is called a *base case* for the recursion
- When the base case is encountered, we get a closed expression that can be directly (often, trivially) computed

Recursion

Numbers Sum

- Suppose that you want to calculate the sum of a list of number, e.g. `[1, 3, 5, 7, 9]`

Recursion

Numbers Sum: Iterative

```
def list_sum(num_list):
    the_sum = 0
    for i in num_list:
        the_sum = the_sum + i
    return the_sum

print(list_sum([1,3,5,7,9]))
```

Recursion

Numbers Sum

- total = (1+(3+(5+(7+9))))
  total = (1+(3+(5+16)))
  total = (1+(3+21))
  total = (1 + 24)
  total = 25

- listSum(numList) =
  first(numList) +
  listSum(rest(numList))

Recursion

Numbers Sum: Recursive

```
def list_sum_rec(num_list):
  if len(num_list) == 1:
      return num_list[0]
  else:
      return num_list[0] + list_sum_rec(num_list[1:])

print(list_sum_rec([1,3,5,7,9]))
```

Recursion

- This is inefficient (lots of copying), but Python runs out of stack to keep track of the calls before the cost gets out of hand
- Note that the two implementations actually sum the elements in opposite order
  - We could make them do it in the same order like this:

Recursion

Numbers Sum: Recursive (better)

```
def listsum_rec2(the_sum, l):
    if len(l) == 0:
        return the_sum
    return listsum_rec2(the_sum + l[0], l[1:])

print(listsum_rec2(0, [1,3,5,7,9]))
```

Recursion

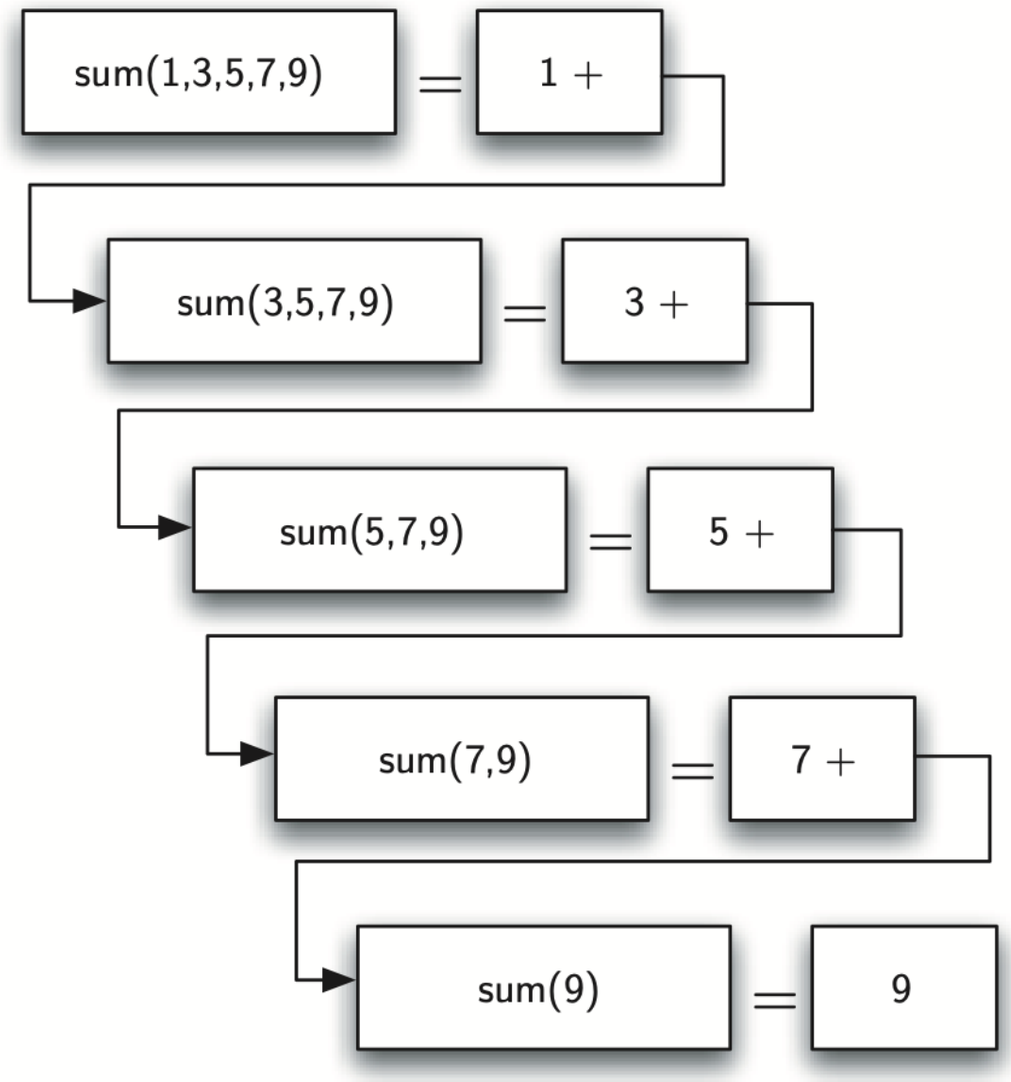
The Recursive Sum Function

```
def helper(sum, l):
    if len(l) == 0:
        return sum
    return helper(sum + l[0], l[1:])

def listsum_rec3(l):
    return helper(0, l)
```

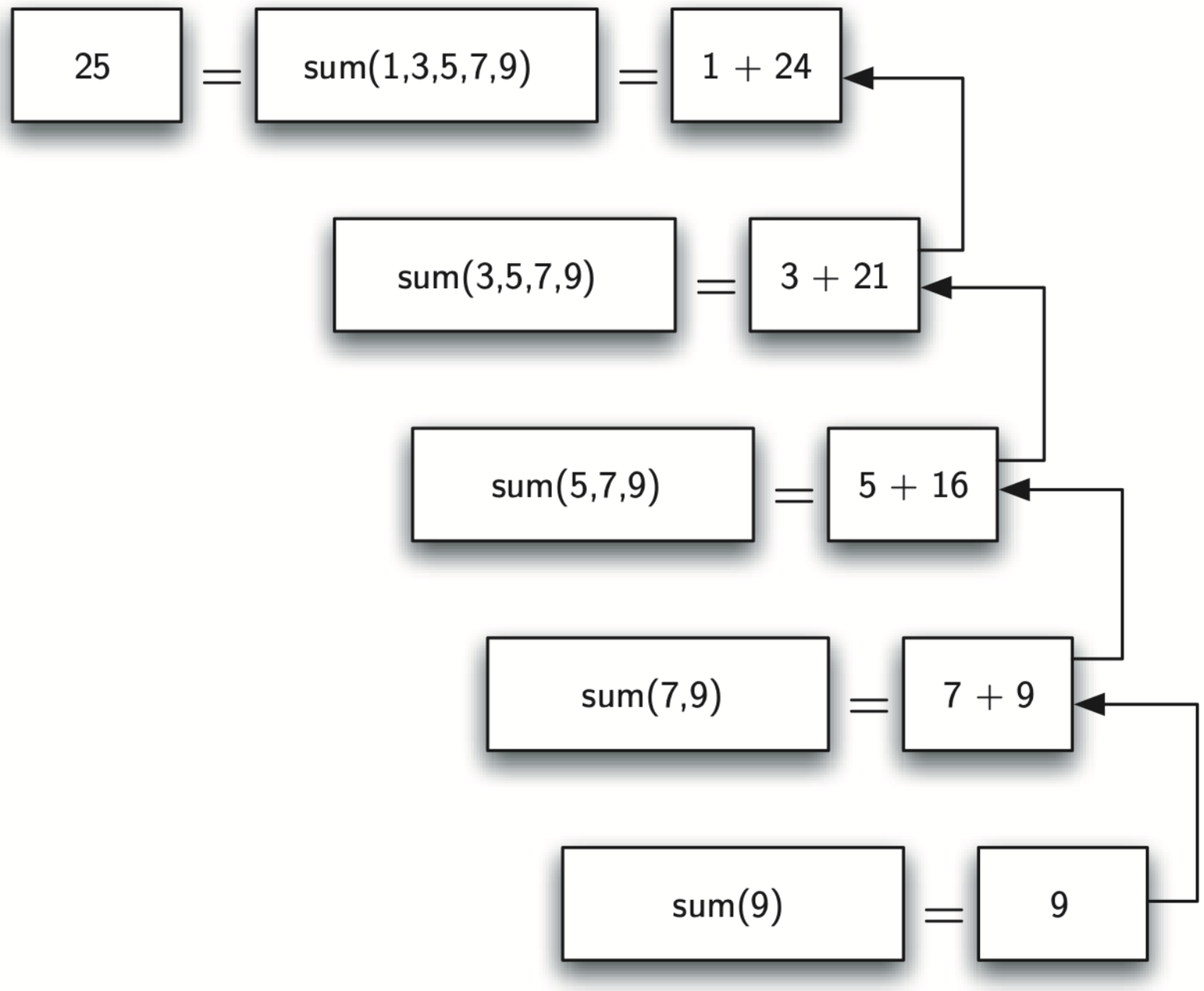
Recursion

Recursive Calls Adding a List of Numbers



Recursion

Recursive Calls Adding a List of Numbers



Recursion

The Three Laws of Recursion

1. A recursive algorithm must have a base case
2. A recursive algorithm must change its state and move toward the base case
3. A recursive algorithm must call itself, recursively

Recursion
Areas of Use

- Two fundamental computational concepts
  - Divide & Conquer: Solve a problem in terms of a smaller version of itself
  - Backtracking: Systematically explore a set of possible solutions

Factorial

Factorial

- We've seen previously that `fact` can be calculated using a loop accumulator
- If `fact` is written recursively...

Factorial

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

Factorial

- We’ve written a function that calls itself, a *recursive function*
- The function first checks to see if we’re at the base case (`n==0`). If so, return `1`
- Otherwise, return the result of multiplying `n` by the factorial of `n-1`, `fact(n-1)`

Factorial

```
>>> fact(4)
24
>>> fact(10)
3628800
>>> fact(100)
933262154439441526816992388562667004907159682643816214685929638952175999932299156089414639761565182862536979208272237582511852109168640000000000000000000000L
>>>
```

- Remember that each call to a function starts that function anew, with its own copies of local variables and parameters

Factorial

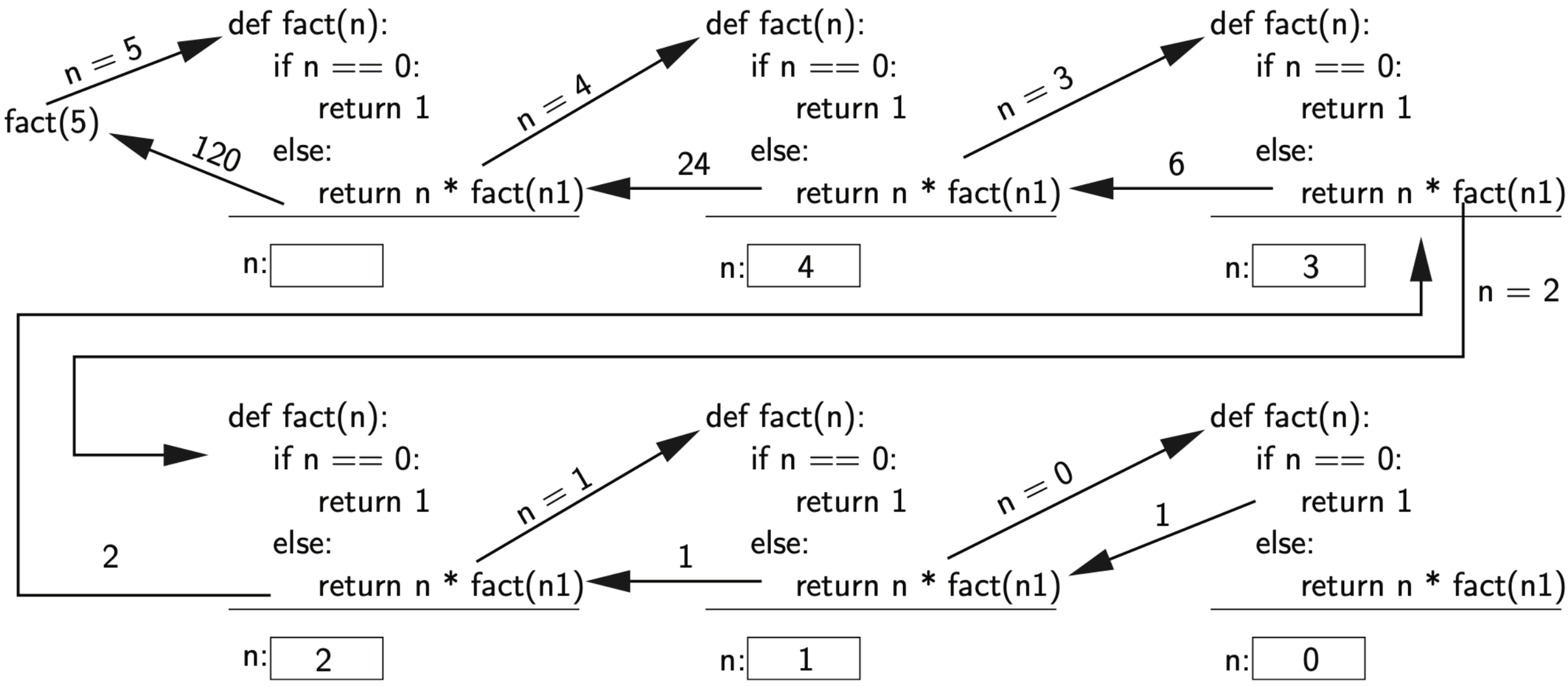


Figure 13.1: Recursive computation of 5!

Fibonacci

Fibonacci

- Sometimes one has to be careful with recursion
- In addition to limits on stack depth [`sys.setrecursionlimit(`*`limit`*`)`], and the cost of argument copying, some naive recursive algorithms are inherently expensive

Fibonacci

Iterative Fibonacci function – *O(n)*

```
def fib_iter(n):
    a = 0
    b = 1
    i = 0

    print(a)
    while i < n:
        t = a+b
        a = b
        b = t
        print(a)
        i += 1

    return a
```

Fibonacci

Naive recursive version, $O(2^n)$

```
def fib_rec_1(n):
if n < 2:
    return 1

return fib_rec_1(n-1) + fib_rec1(n-2)
```

Fibonacci

Good recursive version, O(n)

```
def fib_rec_2(n):
    def helper(a, b, i):
        if i == n:
            return b
        return helper(b, a + b, i + 1)

    return helper(1, 0, 0)
```

Recursion Exercise

Take this recursive function:

```
1  def fun(n):
2      if n == 0:
3          return []
4
5      return fun(n//2) + [n%2]
6
7  fun(25)
```

(a) Identify the line number for the *base case* in `fun(n)`:

(b) Identify the line number, and specific *recursive call* in `fun(n)`:

(c) Use the table below to determine the output of `fun(n)` when it is called with `n = 25` (line 7):

Recursion Level	n	`fun(n)` is called with...	is `n == 0`?	line 5, calculate `n//2`	line 5, calculate `n%2`
0 (start)	25	`fun(25)`	FALSE	12	1
1	12				
2					
3					
4					
5					

(d) The final returned value from calling `fun(25)`:

fun(25)

```
1 def fun(25):
2     if n == 0:
3         return []
4
5     return fun(25//2) + [25%2]
```

```
1 def fun(12):
2     if n == 0:
3         return []
4
5     return fun(12//2) + [12%2]
```

```
1 def fun(6):
2     if n == 0:
3         return []
4
5     return fun(6//2) + [6%2]
```

```
1 def fun(3):
2     if n == 0:
3         return []
4
5     return fun(3//2) + [3%2]
```

```
1 def fun(1):
2     if n == 0:
3         return []
4
5     return fun(1//2) + [1%2]
```

```
1 def fun(0):
2     if n == 0:
3         return []
4
5     return fun(n//2) + [n%2]
```

[1,1,0,0,1]

[] + [1] + [1] + [0] + [0] + [1] ⟶ [1, 1, 0, 0, 1]

[] + [1] + [1] + [0] + [0] ⟶ [1, 1, 0, 0]

[] + [1] + [1] + [0] ⟶ [1, 1, 0]

[] + [1] + [1] ⟶ [1, 1]

[] + [1] ⟶ [1]

[]

recursion level	n	`fun(n)` is called with…	is `n == 0`?	line 5, calc `n//2`	line 5, calc `n%2`
0 (start)	25	fun(25)	FALSE	12	1
1	12	fun(12)	FALSE	6	0
2	6	fun(6)	FALSE	3	0
3	3	fun(3)	FALSE	1	1
4	1	fun(1)	FALSE	0	1
5	0	fun(0)	TRUE	nothing, it returns	nothing, it returns

Int → Str

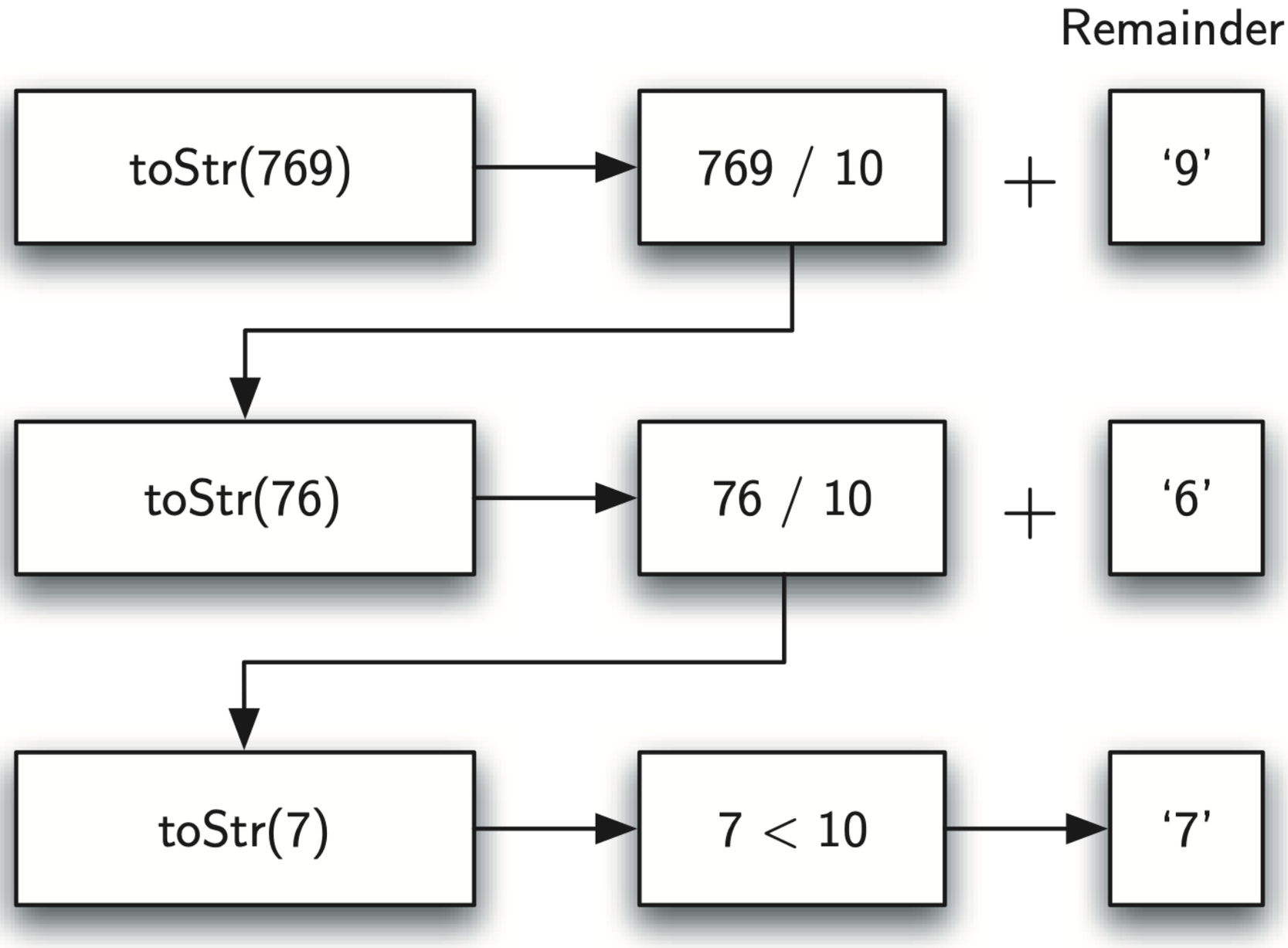
(in any base)

Int → Str
(in any base)

- Reduce the original number to a series of single-digit numbers
- Convert the single digit-number to a string using a lookup
- Concatenate the single-digit strings together to form the final result

Int → Str

in Base 10



Int → Str

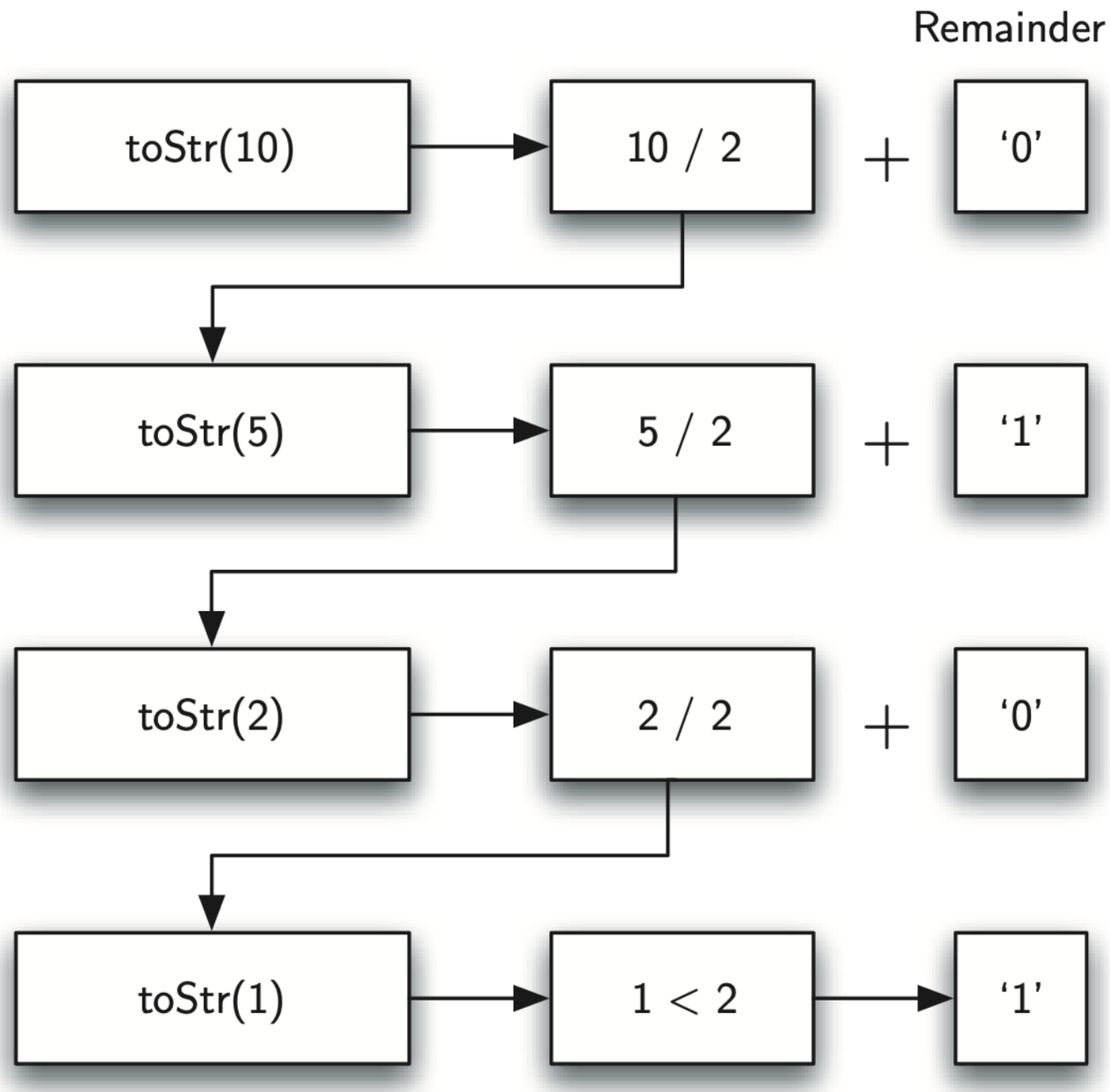
in Base 2–16

```
def to_str(n, base):
   convert_string = "0123456789ABCDEF"
   if n < base:
     return convert_string[n]
   else:
     return to_str(n // base, base) + convert_string[n % base]

print(to_str(1453, 16))
```

Int → Str

Decimal 10 to its Binary String



Int → Str

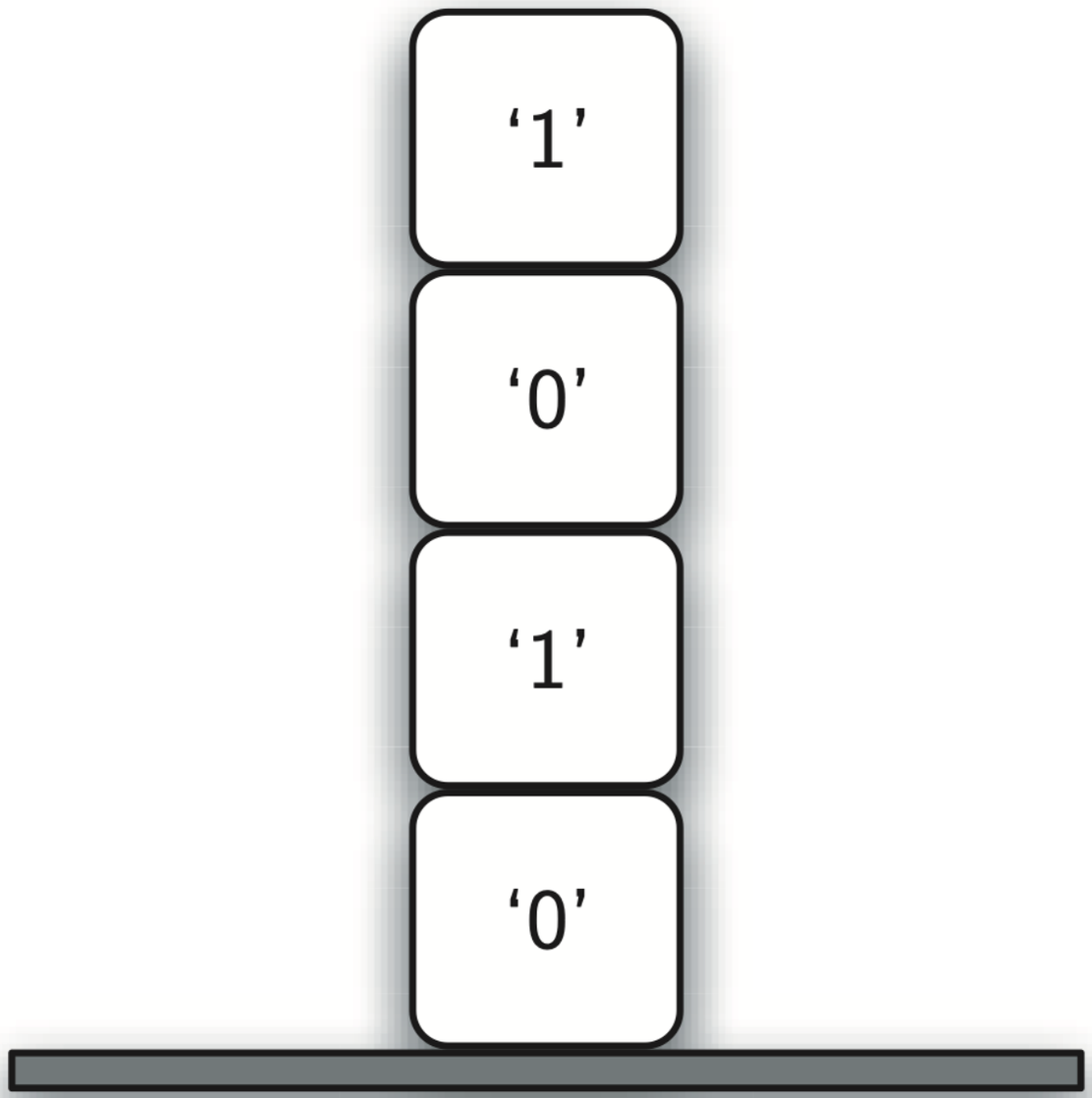
Pushing the Strings onto a Stack

```
r_stack = Stack()

def to_str(n, base):
    convert_string = "0123456789ABCDEF"
    if n < base:
        r_stack.push(convert_string[n])
    else:
        r_stack.push(convert_string[n % base])
        to_str(n // base, base)
```

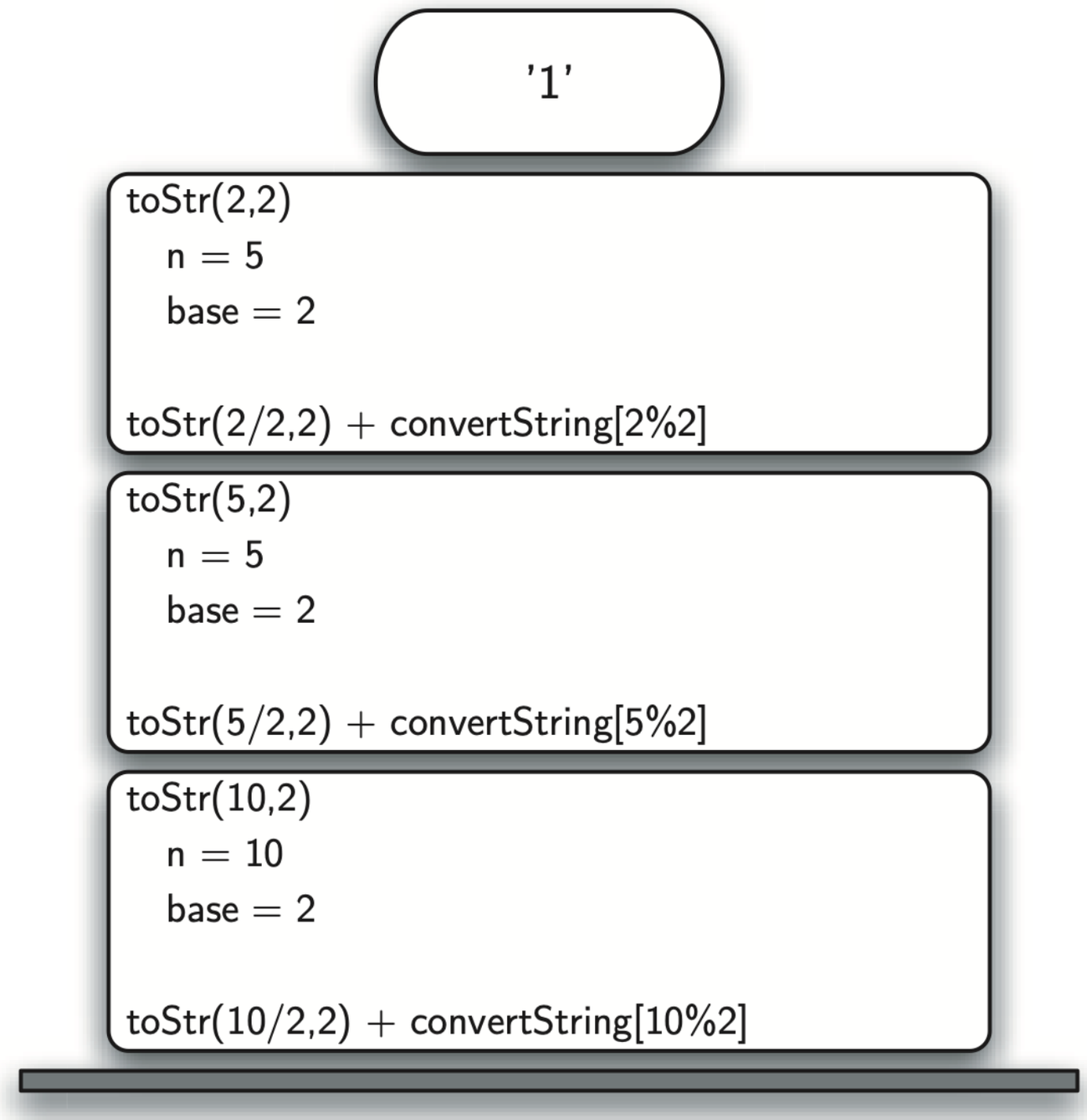
Int → Str

Strings Placed on the Stack



Int → Str

Call Stack: `toStr(10,2)`



String Permutations

Revisit Anagram Tester

String Permutations

Brute Force Anagram Testing

```
def permutation(s, prefix=""):
    n = len(s)
    if (n == 0):
        print(prefix)
    else:
        for i in range(n):
            permutation(s[0:i] + s[i+1:n], prefix + s[i])

permutation("ape")
```

Towers of Hanoi

A Complex Recursive Problem

Towers of Hanoi

Background

- Objective: move N disks from peg A to C can be reduced to three sub problems:

1. Move N-1 disks from peg A to intermediate peg B
2. Move the largest Disk N from peg A to target C
3. Move the N-1 parked disks from B to C

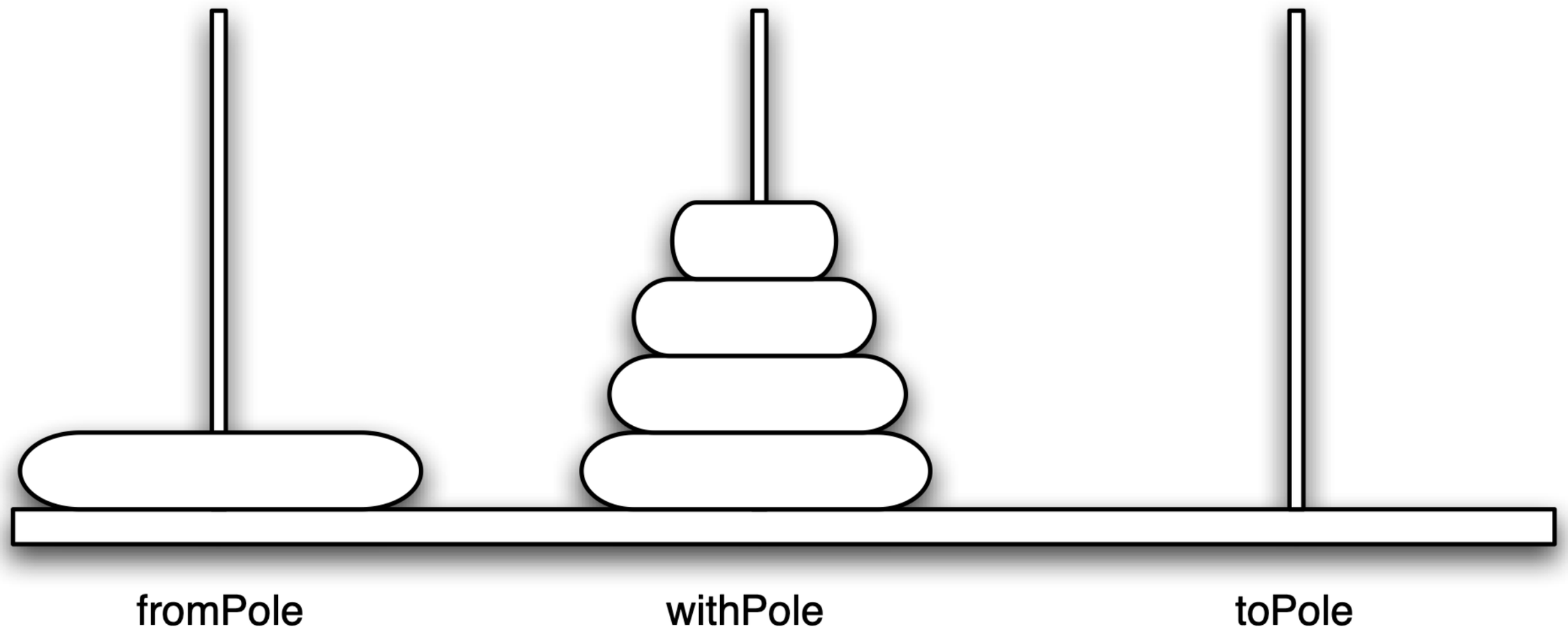
Towers of Hanoi

Background

- Tower of Hanoi (Wikipedia)
- Tower of Hanoi - 5 disks - 31 moves

Towers of Hanoi

An Example Arrangement of Disks



Towers of Hanoi

An Example Arrangement of Disks

- Move a tower of height-1 to an intermediate pole, using the final pole
- Move the remaining disk to the final pole
- Move the tower of height-1 from the intermediate pole to the final pole using the original pole

Towers of Hanoi

Python Code for the Tower of Hanoi

```
def move_tower(height, from_pole, to_pole, with_pole):
    if height >= 1:
        move_tower(height - 1, from_pole, with_pole, to_pole)
        move_disk(from_pole, to_pole)
        move_tower(height - 1, with_pole, to_pole, from_pole)

def move_disk(fp,tp):
    print("moving disk from",fp,"to",tp)

move_tower(3, "A", "B", "C")
```

Towers of Hanoi

An Iterative Version

Interesting secret: there's also an easy iterative solution, but it isn't anywhere near as intuitive

1. On every even-numbered move (starting with zero), move the little disk one pole "clockwise"

   If the total number of disks is even, the first move should be from `from_pole` to `with_pole`; if the total number of disks is odd, the first move should be from `from_pole` to `with_pole`

Towers of Hanoi

An Iterative Version

2. On every odd-numbered move, make the only legal move not involving the smallest disk (there can be only one)

Towers of Hanoi

Python Code for the Tower of Hanoi

```
def hanoi_iter(height, fromPole, toPole, withPole):
    if height % 2 == 0:
        poles = [fromPole, withPole, toPole]
    else:
        poles = [fromPole, toPole, withPole]
    stacks = [range(height, 0, -1), [height], [height]]
    for i in range(2**height-1):
        if i % 2 == 0:  # move little disk
            fd = (i//2)%3
            td = (i//2+1)%3
        else:           # move other disk
            fd = (i//2)%3
            td = (i//2+2)%3
            if (stacks[fd][len(stacks[fd])-1] >
                stacks[td][len(stacks[td])-1]):
                td = (i//2)%3
                fd = (i//2+2)%3
            stacks[td].append(list(stacks[fd]).pop())
        move_disk(poles[fd], poles[td])
```

Recursion Summary

Recursion Summary

- All recursive algorithms must have a base case
- A recursive algorithm must change its state and make progress toward the base case
- A recursive algorithm must call itself (recursively); Recursion can take the place of iteration in some cases

Recursion Summary

- Recursive algorithms often map very naturally to a formal expression of the problem you are trying to solve

Recursion Summary

- Recursion doesn't have to be any more expensive than iteration (though it is in Python)
- It's definitely more expressive: iteration can't capture recursion in the general case without an explicit stack

Questions?