

Indian Institute of Technology Madras
ZANZIBAR CAMPUS

Z5007 · PROGRAMMING AND DATA STRUCTURES

Thinking Recursively: Base Cases, Trust and the Call Stack

Lecture 15 · Monday 2 November 2026 · Canning, Broder & Lafore Ch 6

Dr. Innocent Nyalala | MTech Semester 1 | IIT Madras Zanzibar

Today's Plan

IIT Madras Zanzibar

1. (0–5 min) The idea: solve a big problem using the answer to a smaller one
2. (5–15 min) Anatomy: base case, recursive case, the call stack drawn frame by frame
3. (15–24 min) Trust the smaller call: sum of a list, fast power, binary search
4. (24–35 min) More than one call: the Fibonacci tree; the Towers of Hanoi
5. (35–40 min) Recursion versus loops; the cost of a recursive function
6. (40–50 min) Live demo, practice questions, your questions

Learning Outcomes

IIT Madras Zanzibar

By the end of this session, you will be able to:

Identify	the base case and the recursive case of a recursive function
Trace	a recursive call on paper: the calls going down and the returns coming back up
Draw	the call stack at any moment, with each frame's own local variables
Write	a recursive function by trusting the smaller call and checking the base case
Estimate	the cost of a recursive function by writing its recurrence

Key Terms for Today

IIT Madras Zanzibar

Term	Meaning	Example
Recursion	a function that calls itself on a smaller input	factorial(n) calls factorial(n − 1)
Base case	an input small enough to answer directly, with no call	factorial(1) = 1
Recursive case	the input is reduced, the function calls itself, and combines	n · factorial(n − 1)
Call stack	the computer's pile of unfinished calls, newest on top	f(4) under f(3) under f(2)
Frame	one call's record: its own parameters and local variables	n = 3
Recursion depth	the number of frames on the stack at the deepest point	4 for factorial(4)
RecursionError	Python's error when the depth passes its limit (1,000)	no base case
Recurrence	an equation for a function's cost in terms of smaller inputs	T(n) = T(n − 1) + c

Thinking Recursively

IIT Madras Zanzibar

→ **The Idea**

Anatomy of a Recursive Call

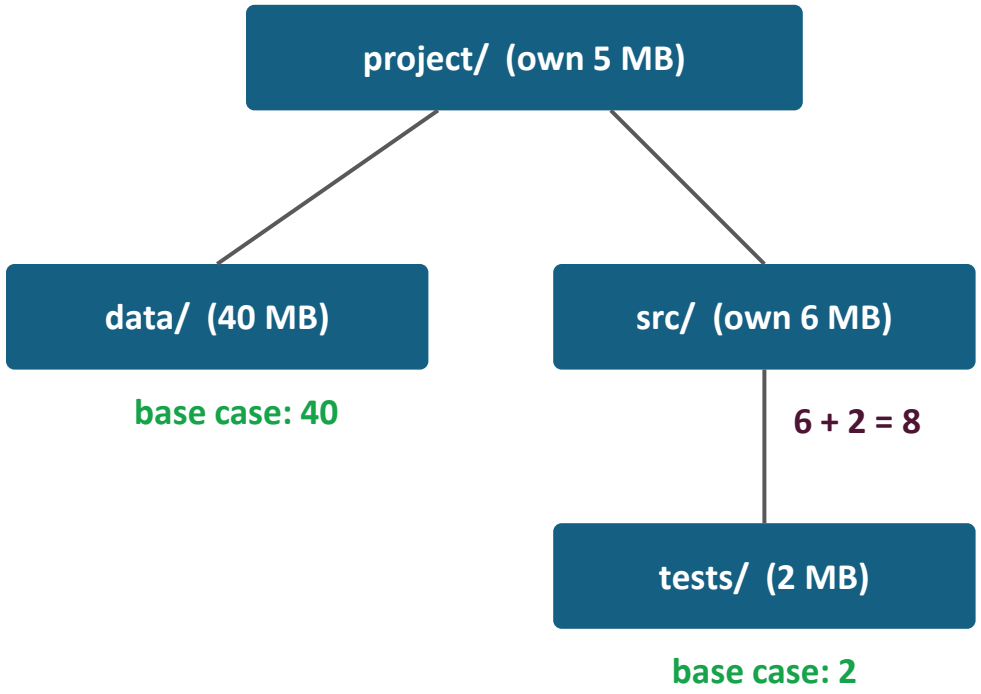
Trust the Smaller Call

More Than One Call: Fibonacci and Hanoi

Recursion, Loops and Cost

Demo and Practice

The Idea: How Much Disk Does This Folder Use?




```
size(folder) =
    its own files
  + size(each sub-folder)
```

A folder with no sub-folders: just add its files. That is where the asking stops (the base case).

project = 5 + 40 + 8 = 53 MB

? **Think:** where does the asking stop? Why can't a plain for-loop do this?

Recursion in One Slide: Three Rules

IIT Madras Zanzibar

5! = 5 × 4 × 3 × 2 × 1 = 5 × 4! so n! = n × (n – 1)! and 1! = 1

```
def factorial(n):                  # n: the number whose factorial we want
    if n <= 1:                     # is the problem already tiny?
        return 1                   # base case: answer directly, no more calls
    return n * factorial(n - 1)    # recursive case: call itself on n - 1, then times n
```

1. Base case

an input answered directly: n ≤ 1 gives 1

2. Smaller input

each call gets closer to the base case: n → n – 1

3. Combine

use the smaller answer to build yours: × n

? **Think:** what does factorial(3) return? Which line does factorial(1) run?

Definition: Recursion: solving a problem by solving a smaller copy of the same problem, down to a case small enough to answer directly.

The Two Pillars of Recursion: Base Case and Recursive Case

IIT Madras Zanzibar

1

Pillar 1: Base case

WHAT IT DOES: answers a tiny input directly, with no further call. It is what STOPS the recursion.

2

Pillar 2: Recursive case

WHAT IT DOES: calls the function on a SMALLER input, then uses that answer to build its own.

factorial(n)	when	what happens	calls?
base case	n <= 1	return 1	none: stops
recursive case	n > 1	return n * factorial(n - 1)	one, on n - 1

Both are needed: no base case: never stops. Input not getting smaller: never reaches the base case.

?

Think: for sum_to(n) = 1 + 2 + ... + n, which part is the base case and which is the recursive case?

Computing factorial(5), Step by Step

IIT Madras Zanzibar

Going down: each call waits

`factorial(5)`	`= 5 × factorial(4)  waits`	`returns 5 × 24 = 120`
`factorial(4)`	`= 4 × factorial(3)  waits`	`returns 4 × 6 = 24`
`factorial(3)`	`= 3 × factorial(2)  waits`	`returns 3 × 2 = 6`
`factorial(2)`	`= 2 × factorial(1)  waits`	`returns 2 × 1 = 2`
`factorial(1)`	`1 <= 1: base case, returns 1`	

Coming back up: each call finishes

Answer: factorial(5) = 5 × 4 × 3 × 2 × 1 = 120, from 5 calls

? **Think:** when factorial(1) runs, how many calls are waiting?

Thinking Recursively

IIT Madras Zanzibar

The Idea

→ **Anatomy of a Recursive Call**

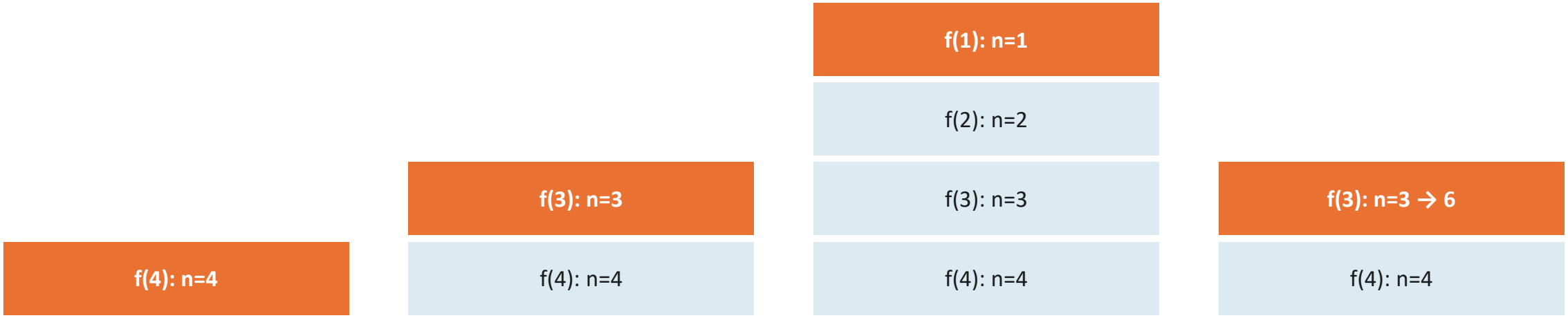
Trust the Smaller Call

More Than One Call: Fibonacci and Hanoi

Recursion, Loops and Cost

Demo and Practice

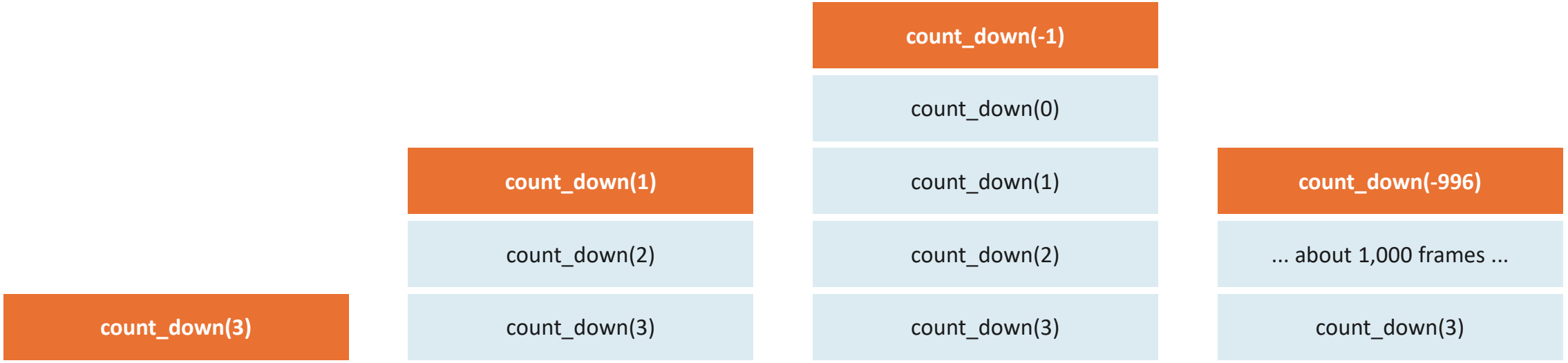
The Call Stack: One Frame per Unfinished Call



1. Each frame keeps its own n. Push on a call, pop on a return: last in, first out (the Week 3 stack).

? **Think:** factorial(n): how many frames at the deepest point? How much memory?

Stack Overflow: When the Stack Runs Out of Room



1. Every call adds a frame; a frame is removed only when its call returns.
2. No base case, so no call returns: frames pile up until Python's limit, about 1,000.
3. Python stops with RecursionError (a stack overflow in Java or C).

? **Think:** factorial(5000) has a correct base case. Does it overflow? Why?

What If There Is No Base Case?

IIT Madras Zanzibar

```
def count_down(n):      # n: the number to print
    print(n)            # print the current number
    count_down(n - 1)   # call itself with one less
    # no 'if ...: return' anywhere: nothing stops it
```

`count_down(3)`	prints 3
`count_down(2)`	prints 2
`count_down(1)`	prints 1
`count_down(0)`	prints 0
`count_down(-1)`	prints −1
…	… and on, forever

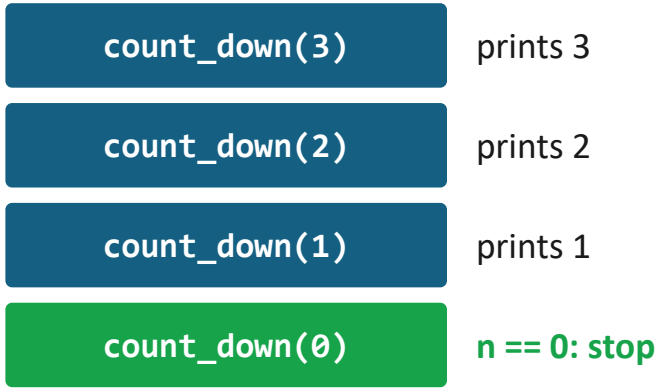
After about 1,000 calls: Python stops the program with RecursionError: maximum recursion depth exceeded. Meaning: no base case was ever reached.

? **Think:** would count_down(1) stop, since it starts so close to 0?

Adding the Base Case, and Making Sure We Reach It

IIT Madras Zanzibar

```
def count_down(n):
    if n == 0:          # base case: have we reached 0?
        return          # yes: stop, no more calls
    print(n)
    count_down(n - 1)   # n gets smaller: 3, 2, 1, 0
```



Output: 3 2 1 then it stops

Reach it: factorial with if n == 1 instead of n <= 1: factorial(0) → factorial(−1) → factorial(−2) … never equals 1: RecursionError. n <= 1 stops for 1, 0 and negatives.

? **Think:** with this fixed code, does count_down(−2) stop? How would you fix it?

Thinking Recursively

IIT Madras Zanzibar

The Idea

Anatomy of a Recursive Call

→ **Trust the Smaller Call**

More Than One Call: Fibonacci and Hanoi

Recursion, Loops and Cost

Demo and Practice

Sum of a List: What Each Name Means

IIT Madras Zanzibar

```
def sum_list(a, i=0):                    # a: the list;  i: the position to start adding from
    if i == len(a):                      # i is past the last item: nothing left to add
        return 0                         # base case: the sum of no numbers is 0
    return a[i] + sum_list(a, i + 1)     # this item + the sum of the items after it
```



`a`	= [4, 8, 15, 16]: the list to add up
`i = 0`	in the def line: start at the first item unless told otherwise
`len(a)`	= 4 items, so the positions are 0, 1, 2, 3
`i == len(a)`	i = 4: no item there, nothing left: return 0
`i + 1`	the same job, starting one position later: the rest of the list

? **Think:** a = [10, 20, 30]: what is len(a)? At which i does the function stop?

Computing sum_list([4, 8, 15, 16]), Step by Step

IIT Madras Zanzibar

Going down: each call waits		Coming back up: each call finishes
`sum_list(a, 0)`	`= a[0] + sum_list(a, 1) = 4 + ?`	`returns 4 + 39 = 43`
`sum_list(a, 1)`	`= a[1] + sum_list(a, 2) = 8 + ?`	`returns 8 + 31 = 39`
`sum_list(a, 2)`	`= a[2] + sum_list(a, 3) = 15 + ?`	`returns 15 + 16 = 31`
`sum_list(a, 3)`	`= a[3] + sum_list(a, 4) = 16 + ?`	`returns 16 + 0 = 16`
`sum_list(a, 4)`	`i = 4 = len(a): nothing left, returns 0`	

Answer: sum_list([4, 8, 15, 16]) = 4 + 8 + 15 + 16 = 43, from 5 calls

? **Think:** sum_list([2, 5, 7]): what is returned, and how many calls?

Fast Power: What x, n, // and % Mean

IIT Madras Zanzibar

```
def power(x, n):              # x: the base, n: the power.  power(2, 10) = 2 to the power 10
    if n == 0:                # any number to the power 0 is 1
        return 1              # base case
    half = power(x, n // 2)   # ONE recursive call: x to the power (n // 2)
    if n % 2 == 0:            # n even (remainder 0)
        return half * half    # e.g. 2^10 = 2^5 × 2^5
    return half * half * x    # n odd: e.g. 2^5 = 2^2 × 2^2 × 2
```

// divides, drops the remainder: `10 // 2 = 5` `5 // 2 = 2` `2 // 2 = 1` `1 // 2 = 0`

% gives the remainder: `10 % 2 = 0` → **even** `5 % 2 = 1` → **odd**

Why fast: the power halves every call: 2^10 needs 5 calls; 2^1,000,000 needs about 21.

? **Think:** what is 7 // 2? What is 7 % 2? Is 7 even or odd?

Computing power(2, 10), Step by Step

IIT Madras Zanzibar

	Going down: each call waits		Coming back up: each call finishes
power(2, 10)	half = power(2, 5)	10 // 2 = 5	half = 32, even: 32×32 = 1024
power(2, 5)	half = power(2, 2)	5 // 2 = 2	half = 4, odd: 4×4×2 = 32
power(2, 2)	half = power(2, 1)	2 // 2 = 1	half = 2, even: 2×2 = 4
power(2, 1)	half = power(2, 0)	1 // 2 = 0	half = 1, odd: 1×1×2 = 2
power(2, 0)	n == 0: base case, returns 1		

Answer: power(2, 10) = 1024 = 2 to the power 10, from 5 calls instead of 10 multiplications

? **Think:** power(3, 4): which calls are made, and what does each return?

Binary Search, Written Recursively

```
def bsearch(a, t, lo, hi):  # a: sorted list, t: target
    if lo > hi:             # lo..hi: part still to search
        return -1           # base case 1: empty, not found
    mid = (lo + hi) // 2    # the middle position
    if a[mid] == t:
        return mid          # base case 2: found it
    if a[mid] < t:          # target bigger: it is on the right
        return bsearch(a, t, mid + 1, hi)  # RECURSION
    return bsearch(a, t, lo, mid - 1)      # RECURSION
```

Recursion: the last two lines. bsearch calls itself on the right half or the left half.

call 1: bsearch(a, 23, 0, 6)



mid = 3: 15 < 23, go right

call 2: bsearch(a, 23, 4, 6)



mid = 5: 31 > 23, go left

call 3: bsearch(a, 23, 4, 4)



mid = 4: 23 found, return 4

? **Think:** which calls are made when we search for 42?

Thinking Recursively

IIT Madras Zanzibar

The Idea

Anatomy of a Recursive Call

Trust the Smaller Call

→ **More Than One Call: Fibonacci and Hanoi**

Recursion, Loops and Cost

Demo and Practice

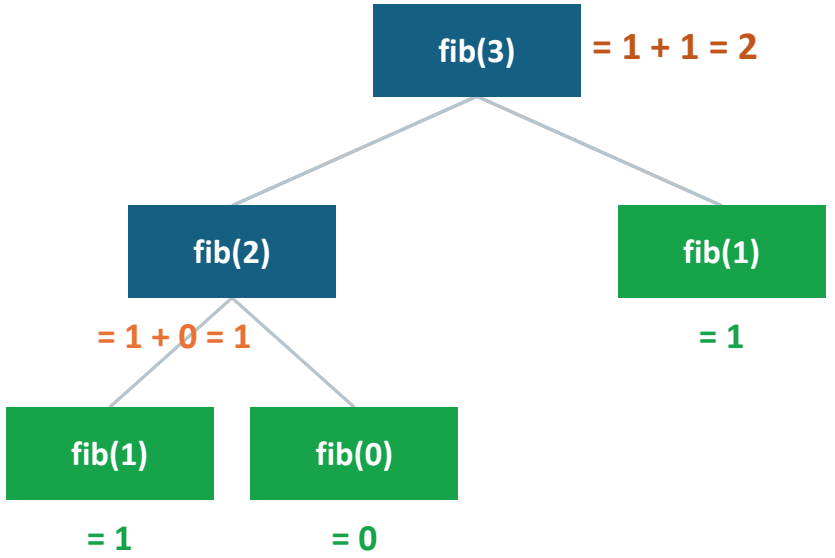
Fibonacci: Two Recursive Calls

IIT Madras Zanzibar

```
def fib(n):                          # n: which one we
want
    if n <= 1:                       # fib(0) = 0, fib(1)
= 1
        return n                     # base case
    return fib(n - 1) + fib(n - 2)   # TWO calls, then add
```

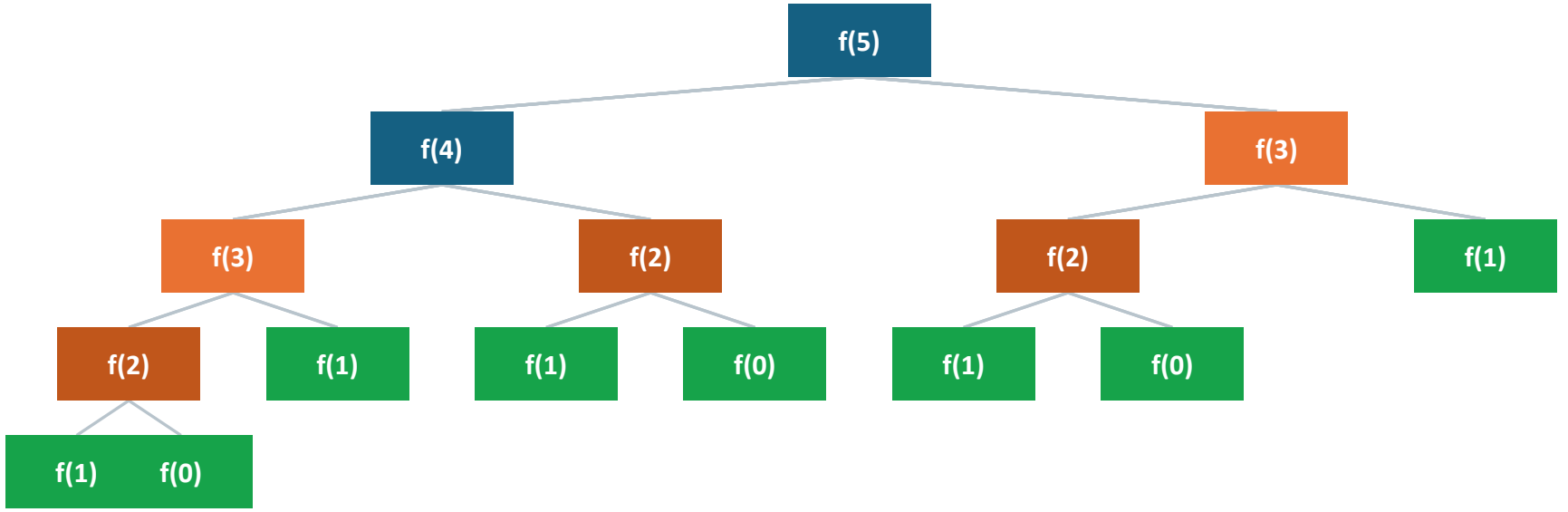
n	0	1	2	3	4	5	6
fib(n)	0	1	1	2	3	5	8

Each number = the two before it added: 2 + 3 = 5, 3 + 5 = 8



? **Think:** fib(4) = fib(3) + fib(2). Using this tree, what is fib(4)?

Why fib Gets Slow: Counting the Calls of fib(5)



1 call

2 calls

4 calls

6 calls

2 calls

green: base case orange: fib(3) done twice red: fib(2) done three times

total: 15

Rule: calls(n) = 1 + calls(n − 1) + calls(n − 2), calls(0) = calls(1) = 1

n	0	1	2	3	4	5	6	7	8	9	10
calls	1	1	3	5	9	15	25	41	67	109	177

Think: how many calls does fib(6) make?

The Towers of Hanoi: the Idea and the Code

IIT Madras Zanzibar

1 **1. Move n – 1 discs A → B**

out of the way (a smaller problem: trust it)

2 **2. Biggest disc A → C**

one move; nothing is on top of it now

3 **3. Move n – 1 discs B → C**

on top of the biggest disc (trust again)

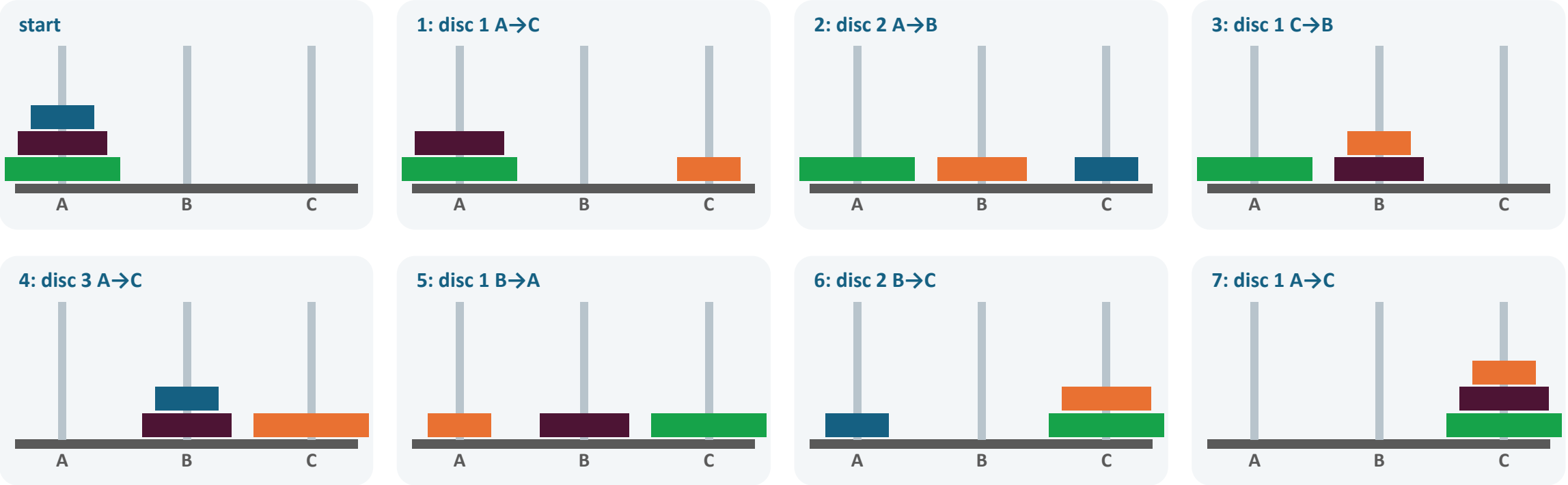
```
def hanoi(n, src, spare, dst):          # move n discs from src to dst
    if n == 0:                          # no discs to move?
        return                          # base case: do nothing
    hanoi(n - 1, src, dst, spare)       # 1: n-1 discs src -> spare
    print('move disc', n, src, '->', dst)  # 2: biggest: src -> dst
    hanoi(n - 1, spare, src, dst)       # 3: n-1 discs spare -> dst
```

`n`	how many discs to move
`src`	the peg they are on now
`dst`	the peg they must end on
`spare`	the third peg, for parking

Example: hanoi(3, 'A', 'B', 'C') means move 3 discs from A to C, parking on B. Lines 4 and 6 are the recursion.

? **Think:** in step 1, why may we put discs on C, where the biggest disc must end?

Hanoi with 3 Discs: All 7 Moves



Moves 1–3: two discs A → B · Move 4: biggest A → C · Moves 5–7: two discs B → C

? **Think:** with 4 discs, how many moves? On which move does the biggest disc move?

How Many Moves? Our First Recurrence

n discs	moves M(n)	pattern
1	1	$2^1 - 1$
2	3	$2^2 - 1$
3	7	$2^3 - 1$
4	15	$2^4 - 1$
10	1,023	$2^{10} - 1$
64	about 1.8×10^{19}	$2^{64} - 1$

$$M(n) = 2\,M(n-1) + 1, \quad M(0) = 0$$

1. Step 1 costs M(n − 1); step 2 costs 1; step 3 costs M(n − 1).
2. Solution: $M(n) = 2^n - 1$, so $O(2^n)$.
3. 64 discs at 1 move per second: about 585 billion years.

Definition: Recurrence: an equation giving the cost for input n in terms of the cost for smaller inputs, plus the work done in this call.

? **Think:** one more disc is added. What happens to the number of moves?

Thinking Recursively

IIT Madras Zanzibar

The Idea

Anatomy of a Recursive Call

Trust the Smaller Call

More Than One Call: Fibonacci and Hanoi

→ **Recursion, Loops and Cost**

Demo and Practice

Same Function, Two Ways: Recursion and a Loop

IIT Madras Zanzibar

Recursion

```
def factorial(n):
    if n <= 1:                  # base case
        return 1
    return n * factorial(n - 1)   # smaller call
```

Loop

```
def factorial(n):
    result = 1                  # plays the base case
    for i in range(2, n + 1):   # i = 2..n
        result = result * i     # multiply in i
    return result
```

factorial(4)	start	i = 2	i = 3	i = 4	answer
loop: result	1	2	6	24	24

Same answer: the loop builds 1, 2, 6, 24 going up; the recursion builds the same numbers coming back up the stack.

? **Think:** write sum_to(n) = 1 + 2 + ... + n both ways. What is the base case?

Recursion or a Loop?

IIT Madras Zanzibar

	Recursion	Loop
natural for	trees, divide and conquer, Hanoi, nested structures	walking a list, counting, simple repetition
memory	one frame per pending call: O(depth)	O(1) extra
limit	Python: about 1,000 frames	none
speed	a function call costs more than a loop step	faster for the same work

Rule of thumb: recursion that HALVES the input is shallow and safe; recursion that removes ONE item per call is n deep.

? **Think:** is recursive binary search on 1,000,000 items safe in Python? Why?

The Cost of a Recursive Function: Write Its Recurrence

IIT Madras Zanzibar

Function	Calls per level	Input shrinks to	Recurrence	Cost
factorial, sum_list	1	n − 1	T(n) = T(n − 1) + c	O(n)
power, binary search	1	n / 2	T(n) = T(n/2) + c	O(log n)
Hanoi	2	n − 1	T(n) = 2T(n − 1) + c	$O(2^n)$
merge sort (Wednesday)	2	n / 2	T(n) = 2T(n/2) + n	Wednesday

Unrolling factorial: $T(n) = T(n-1) + c = T(n-2) + 2c = \ldots = T(1) + (n-1)c \rightarrow O(n)$

? **Think:** one call on n − 1, plus a loop of n steps in the call. Recurrence? Cost?

A Recipe for Writing Any Recursive Function

IIT Madras Zanzibar

1. Smallest input?

one item left: i == len(a) − 1

2. Its answer?

that item: a[i]

3. Smaller problem?

trust max_list(a, i + 1)

4. Combine?

max(a[i], that)

```
def max_list(a, i=0):                      # a: the list;  i: start position
    if i == len(a) - 1:                    # i is the last position: one item left
        return a[i]                        # base case: the largest of one item is that item
    return max(a[i], max_list(a, i + 1))   # RECURSION: bigger of this item and the rest's largest
```

Order: write the base case first, then trust the smaller call, then write one combining step.

? **Think:** why is the base case 'one item left' and not 'no items left', as in sum_list?

Computing max_list([3, 9, 4]), Step by Step

IIT Madras Zanzibar

Going down: each call waits

Coming back up: each call finishes

`max_list(a, 0)`	`= max(a[0], max_list(a, 1)) = max(3, ?)`	`returns max(3, 9) = 9`
`max_list(a, 1)`	`= max(a[1], max_list(a, 2)) = max(9, ?)`	`returns max(9, 4) = 9`
`max_list(a, 2)`	`i = 2 = last position: returns a[2] = 4`	

Answer: max_list([3, 9, 4]) = 9, from 3 calls

? **Think:** max_list([7, 2]): which calls are made, and what is returned?

Thinking Recursively

IIT Madras Zanzibar

The Idea

Anatomy of a Recursive Call

Trust the Smaller Call

More Than One Call: Fibonacci and Hanoi

Recursion, Loops and Cost

→ **Demo and Practice**

Live Demo

IIT Madras Zanzibar

```
cd ~/PDS/Class_Demos/Week_05
python3 W05_Mon_Nov02_recursion.py factorial 5
python3 W05_Mon_Nov02_recursion.py power 2 10
python3 W05_Mon_Nov02_recursion.py search 23 3 8 12 15 23 31 42
python3 W05_Mon_Nov02_recursion.py hanoi 3
python3 W05_Mon_Nov02_recursion.py fib 5
python3 W05_Mon_Nov02_recursion.py deep
```

What to watch

1. Indentation = depth of the call stack.
2. Compare the Hanoi pictures and the fib(5) tree with the slides.

In-Class Exercise (6 minutes, in pairs)

IIT Madras Zanzibar

IN-CLASS TASK: base case first, then trust

1. Trace power(3, 5): list every call going down and every value returned coming back up.

2. def m(n): return 0 if n == 0 else 1 + m(n // 2). What is m(13)? What does m compute in general?

3. def count_down(n): print(n); count_down(n - 1). What goes wrong? Fix it so count_down(3) prints 3, 2, 1.

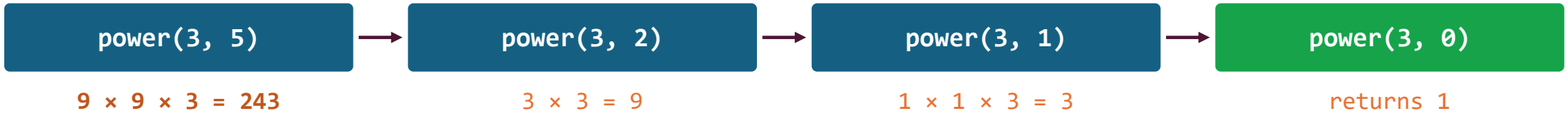
4. Write length(node): the number of nodes in a linked list starting at node, recursively. Base case?

5. Hanoi with 4 discs: how many moves? Which disc moves on move 8, and from where to where?

Solutions 1 and 2

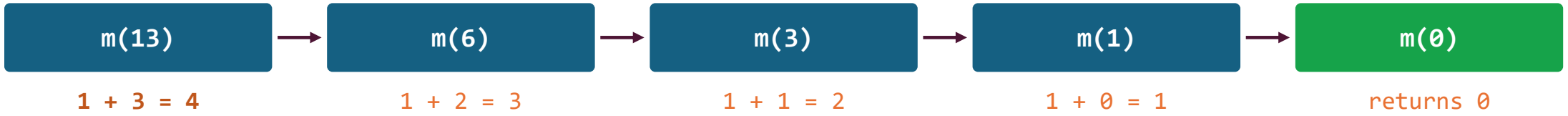
IIT Madras Zanzibar

1. power(3, 5): every call going down, every value coming back



5 // 2 = 2, 2 // 2 = 1, 1 // 2 = 0. Four calls; answer 243.

2. m(n) = 0 if n == 0, else 1 + m(n // 2). What is m(13)?



Meaning: m(n) counts how many times n can be halved before 0: the number of binary digits. 13 = 1101: 4 digits.

Solutions 3, 4 and 5

IIT Madras Zanzibar

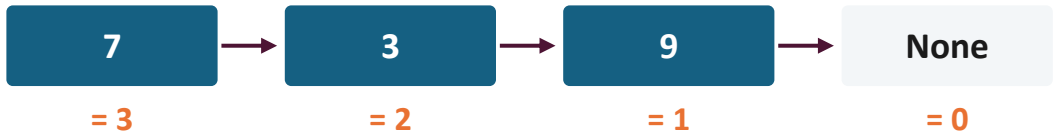
3. Add the missing base case

```
def count_down(n):
    if n == 0: return    # ADDED: base case
    print(n)
    count_down(n - 1)
```

now prints 3, 2, 1 and stops

4. length(node) on the list 7 → 3 → 9

```
def length(node):
    if node is None: return 0      # empty list
    return 1 + length(node.next)   # RECURSION
```



5. Hanoi with 4 discs: $2^4 - 1 = 15$ moves

1 Moves 1 to 7

discs 1, 2, 3 go A → B (the 3-disc puzzle)

2 Move 8

the biggest, disc 4, goes A → C

3 Moves 9 to 15

discs 1, 2, 3 go B → C, on top of disc 4

Common Mistakes

IIT Madras Zanzibar

1 **No base case, or never reached**

RecursionError; check that every call moves towards the base case

2 **Slicing on every call**

a[1:] copies the list: $O(n^2)$. Pass an index

3 **The same call twice**

store the result once: half = power(x, n // 2)

4 **Forgetting return**

return bsearch(...); otherwise the caller gets None

Summary and Exam Checkpoints

IIT Madras Zanzibar

Three rules

base case; a smaller input; combine the smaller answer

Call stack

one frame per unfinished call; each frame has its own variables

Trust

assume the smaller call works; check the base case and one step

Cost

write the recurrence: calls per level and input size → Big-O

Next

Wed 4 Nov: Merge sort: divide and conquer, and solving T(n) = 2T(n/2) + n

Thu 5 Nov: Quicksort; afternoon: ICA 2 (2:00–2:45) and Lab 4 on recursion and sorting

Fri 6 Nov: Recurrences and the Master Theorem

References and Further Reading

[1] Canning, J., Broder, A. & Lafore, R. (2022). Data Structures & Algorithms in Python. Addison-Wesley. Chapter 6 (Recursion).

[2] Cormen, T. H., Leiserson, C. E., Rivest, R. L. & Stein, C. (2022). Introduction to Algorithms, 4th ed. MIT Press. Section 2.3 (recursion and recurrences); Chapter 4.

[3] Goodrich, M. T., Tamassia, R. & Goldwasser, M. H. (2013). Data Structures and Algorithms in Python. Wiley. Chapter 4 (Recursion).

[4] Python documentation: sys.getrecursionlimit; RecursionError.