Recursion Workbook for CS 2

Jeremy Evert

November 3, 2025

Contents

1

Welcome to Recursion

The Big Picture

Welcome to Chapter 14 — where loops learn to dream bigger.

Up to now, you've been mastering the building blocks of programming:

- In **Chapters 1–4**, you learned to build small decisions with logic and branching.
- In **Chapters 5–6**, you harnessed repetition through loops and functions.
- **Chapters 7–12** introduced ways to store, structure, and reuse data and code: strings, lists, dictionaries, modules, and files.
- In **Chapter 13**, you explored inheritance — the first taste of elegant self-reference in object-oriented programming.

Now comes recursion — the art of a function that calls itself. It's not just another way to repeat something; it's a deeper way to think.

Why Recursion Matters

Recursion is the moment when programming starts to feel like storytelling:

> "To solve this problem, I'll solve a smaller version of the same problem, until it becomes so simple it solves itself."

This chapter is where abstraction and problem-solving meet. Recursion helps you:

1. Break large problems into smaller, self-similar ones.
2. Write cleaner code for structures that naturally branch — like trees, directories, or nested data.
3. Understand the mathematical elegance behind algorithms like Fibonacci, quicksort, and binary search.

How It Fits the Course Flow

Recursion bridges **loops and algorithms**. Think of it as a new dimension added to functions:

$$\text{iteration} \Rightarrow \text{recursion} \Rightarrow \text{algorithmic thinking.}$$

By the end of this unit, you'll be able to:

- Identify when recursion is a good fit (and when it's not).
- Trace recursive calls like a detective following a trail of function frames.
- Design your own recursive algorithms for search, sorting, and pattern exploration.

What's Ahead in Chapter 14

Here's how this section aligns with your ZyBooks topics:

14.1 Recursive Functions
Learn the structure of a recursive definition.

14.2 Recursive Algorithm: Search
Explore how recursion simplifies search logic.

14.3 Debugging Recursion
Learn to use print statements to trace your way through the stack.

14.4 Creating a Recursive Function
Practice building and testing your own.

14.5 Recursive Math Functions
Apply recursion to classic math problems.

14.6 Exploration of All Possibilities
See how recursion enables exhaustive search.

14.7–14.8 Labs
Build Fibonacci and permutation generators — your first recursive masterpieces.

Mindset for Success

When you first see recursion, your brain may shout:

"Wait — it's calling itself? But... how does it stop?"

That's normal. Everyone wrestles with the base case and the recursive step. Recursion feels like magic until you learn the trick — and then you realize you've been doing it all along: thinking, teaching, and even living recursively.

So take a breath, trust the process, and remember:

Every problem that feels too big... can be made smaller.

Welcome to recursion. Let's dive down the rabbit hole.

2

Recursive Functions

The Heartbeat of Recursion

A **recursive function** is one that calls itself. That sounds almost mischievous at first — but recursion is not about infinite loops. It's about teaching a function how to do one small piece of a problem and then trusting it to repeat itself until the work is done.

Every recursive function has two essential parts:

1. **Base Case** — When the function stops calling itself. (This prevents infinite recursion.)
2. **Recursive Case** — Where the function calls itself with a smaller or simpler version of the problem.

2.1 The Countdown Example

Here's a simple example, like the one from your ZyBook, rewritten for clarity.

Listing 2.1: Recursive countdown example

```
def count_down(count):
    if count == 0:
        print("Go!")
    else:
        print(count)
        count_down(count - 1)

count_down(3)
```

Output:

```
3
2
1
Go!
```

Each time `count_down()` calls itself, Python pauses the current function, creates a new *stack frame* (a little memory box for local variables), and starts again. When the base case (`count == 0`) is reached, the call stack begins to unwind — printing each result in reverse order of completion.

Recursion isn't looping forward — it's diving down and climbing back up.

2.2 A Simple Mathematical Example: Sum of Numbers

Here's another warm-up, summing all numbers from 1 to `n`.

Listing 2.2: Recursive summation example

```python
def sum_to_n(n):
    if n == 0:
        return 0
    else:
        return n + sum_to_n(n - 1)

print(sum_to_n(5))
```

Call trace:

```
sum_to_n(5)
  = 5 + sum_to_n(4)
    = 5 + 4 + sum_to_n(3)
      = 5 + 4 + 3 + sum_to_n(2)
        = 5 + 4 + 3 + 2 + sum_to_n(1)
          = 5 + 4 + 3 + 2 + 1 + sum_to_n(0)
          = 15
```

You can see the recursive chain shrink by one each time until it bottoms out at 0.

2.3 A String Example: Spelling Backward

Recursion can work on strings too.

Listing 2.3: Recursive string reversal example

```python
def reverse_word(word):
    if len(word) <= 1:
        return word
    else:
        return reverse_word(word[1:]) + word[0]

print(reverse_word("hello"))
```

Output: `olleh`

Each call peels off the first letter, waits for the smaller word to be reversed, and then adds its letter to the end. It's like stacking pancakes, then flipping them one by one as you climb back up the stack.

2.4 A Creative Example: Nested Echo

Let's make recursion a little more human.

Listing 2.4: Recursive echo example

```python
def echo(message, depth):
    if depth == 0:
        print("...silence.")
    else:
        print("Echo:", message)
        echo(message, depth - 1)

echo("Is␣anyone␣there?", 3)
```

Output:

```
Echo: Is anyone there?
Echo: Is anyone there?
Echo: Is anyone there?
...silence.
```

This illustrates the recursive pattern beautifully:

Do something small → Shrink the problem → Trust the function.

2.5 Tracing a Recursive Call Stack

Every recursive call pauses the previous one. In your mind, picture a stack of plates:

1. Each new call adds a plate.
2. The base case stops adding plates.
3. As the calls return, you take the plates back off one at a time.

This "stack of plates" is literally called the **call stack**. You'll use this same concept when debugging recursion, searching trees, or walking directory structures.

2.6 Student Practice

Try writing your own recursive function for these challenges:

1. Print all even numbers from `n` down to 0.
2. Print each letter of a word on its own line, starting from the end.
3. Create a recursive function `count_vowels(word)` that returns the number of vowels in a string.

In the next chapter, we'll use recursion to solve problems that *loops can't easily reach* — recursive search and pattern exploration.

3

Recursive Algorithm: Search

From Repetition to Strategy

Loops and recursion both repeat work—but recursion lets us do it *intelligently*. Instead of plowing through every item one by one, a recursive algorithm can **divide and conquer**.

A recursive algorithm breaks a problem into smaller, self-similar versions of itself until the smallest case (the *base case*) can be solved directly.

3.1 A Familiar Analogy: The Guessing Game

Imagine your friend thinks of a number between 0 and 100. Each time you guess, your friend says "higher" or "lower."

If you always guess halfway between the possible range, you'll find the number in about $\log_2(100) \approx 7$ guesses.

That's **binary search**—recursion in action.

Listing 3.1: Recursive binary search for a number

```python
def binary_search(low, high, target):
    if low > high:
        print("Not␣found!")
        return
    mid = (low + high) // 2
    print(f"Searching␣{low}..{high}␣(mid={mid})")
    if mid == target:
        print("Found␣it!")
    elif target < mid:
        binary_search(low, mid - 1, target)
    else:
        binary_search(mid + 1, high, target)

binary_search(0, 100, 32)
```

This algorithm is recursive because it calls itself on smaller subranges each time. When the range collapses (`low > high`), the function ends.

Every recursive algorithm is a conversation with smaller versions of itself.

3.2 Recursive Search in a Sorted List

Now let's find a name in a list that's alphabetically sorted. This is a textual version of binary search.

Listing 3.2: Recursive search in a sorted list

```
def find(lst, item, low, high):
    if low > high:
        return -1  # Not found
    mid = (low + high) // 2
    if lst[mid] == item:
        return mid
    elif item < lst[mid]:
        return find(lst, item, low, mid - 1)
    else:
        return find(lst, item, mid + 1, high)

names = ["Adams,␣Mary", "Carver,␣Michael", "Domer,␣Hugo",
         "Fredericks,␣Carlo", "Liu,␣Jie"]

person = input("Enter␣last,␣first:␣")
pos = find(names, person, 0, len(names) - 1)
if pos >= 0:
    print(f"Found␣{person}␣at␣index␣{pos}")
else:
    print("Not␣found.")
```

Notice how the search range shrinks by half each time. Recursive binary search is powerful because it eliminates half the data at every step.

3.3 Thinking Recursively

A recursive algorithm always has these three traits:

1. **A clear goal:** What are we trying to find?
2. **A base case:** When do we stop searching?
3. **A recursive case:** How do we break the problem into smaller ones?

You'll see this pattern everywhere—from sorting algorithms (quicksort, mergesort) to tree traversal and directory searches.

3.4 Visualizing the Divide-and-Conquer Pattern

Each recursive call creates a branch in the "decision tree." Here's the mental image (now ASCII-safe):

```
Search [0..100]
 |- Guess 50 -> too high -> Search [0..49]
 |   |- Guess 25 -> too low -> Search [26..49]
 |   |   |- Guess 37 -> too high -> Search [26..36]
 |   |   |   |- Guess 31 -> too low -> Search [32..36]
 |   |   |   |   |- Guess 34 -> Found!
```

Each level of recursion focuses on a smaller search space. The call stack keeps track of where you came from.

3.5 Tracing the Call Stack

When recursion runs, Python keeps track of every unfinished call in the *call stack*. Think of it as a trail of sticky notes—each one says, “Come back to me when you’re done.”

```
binary_search(0, 100, 32)
 |-- binary_search(0, 49, 32)
 |-- binary_search(25, 49, 32)
 |-- binary_search(32, 36, 32)
 |-- Found!
```

Each time the recursive call returns, Python pops one frame off the stack. When the base case is reached, the stack empties gracefully. If you forget your base case—Python keeps stacking until it crashes! (Infinite recursion alert!)

3.6 Try It Yourself

Write your own recursive search for these problems:

1. Search for a letter in a string, returning its index.
2. Search for the smallest number in a sorted list (without using `min()`).
3. Modify `find()` to print the number of recursive calls made.

Challenge: Recursive Word Finder

For extra fun, try writing a recursive function that searches through nested lists:

Listing 3.3: Recursive word finder challenge

```python
def find_word(nested_list, word):
    for item in nested_list:
        if isinstance(item, list):
            if find_word(item, word):
                return True
        elif item == word:
            return True
    return False

data = [["dog", ["cat", "fish"]], ["hamster", ["parrot", "snake"]]]
print(find_word(data, "snake"))  # True
print(find_word(data, "whale"))  # False
```

This shows recursion applied to hierarchical data—a natural fit when structures contain smaller versions of themselves.

Closing Thought

Recursive search is not just faster—it's smarter. It doesn't look at everything; it looks *strategically*. That's what separates **repetition** from **recursion**—and **recursion** from **algorithmic thinking**.

Next: Debugging recursive calls — the art of seeing the invisible stack.

4

Adding Output Statements for Debugging

Why Debugging Recursion Feels Weird

When loops misbehave, we can print a single counter and know exactly where we are. When recursion misbehaves, it's like yelling down a canyon and getting ten voices back at once.

Recursion errors can hide in the call stack, so debugging means learning to *see the depth* of the function calls.

> Each recursive call is like a new mirror added to a hall of reflections. If you can see the pattern of echoes, you can find the bug.

4.1 Adding Output for Insight

One of the best debugging tools is a simple `print()` statement. By printing the current range or step, and indenting based on recursion depth, we can visualize what's happening.

Listing 4.1: Recursive search with indentation for debugging

```
def find(lst, item, low, high, indent=""):
    """Finds␣index␣of␣item␣in␣a␣sorted␣list,␣else␣returns␣-1."""
    print(f"{indent}Searching␣{low}..{high}")

    # Base case: Not found
    if low > high:
        print(f"{indent}Not␣found.")
        return -1

    mid = (low + high) // 2
    print(f"{indent}Checking␣index␣{mid}:␣{lst[mid]}")

    # Base case: Found
    if lst[mid] == item:
        print(f"{indent}Found␣{item}␣at␣index␣{mid}")
        return mid

    # Recursive case: Search smaller half
    elif item < lst[mid]:
        return find(lst, item, low, mid - 1, indent + "   ")

    # Recursive case: Search larger half
    else:
        return find(lst, item, mid + 1, high, indent + "   ")

# Example usage
names = ["Adams, Mary", "Carver, Michael", "Domer, Hugo",
         "Fredericks, Carlo", "Liu, Jie"]

print("Running debug search for 'Carver, Michael'...\n")
find(names, "Carver, Michael", 0, len(names) - 1)
```

Notice how the indentation grows deeper with each recursive call. The result is a visual "map" of the recursion depth — a living breadcrumb trail.

> Indentation isn't just about code style — it can become a debugging superpower.

4.2 What the Output Teaches Us

When you run this, you'll see the function's "journey":

```
Searching 0..4
Checking index 2: Domer, Hugo
   Searching 0..1
   Checking index 0: Adams, Mary
      Searching 1..1
      Checking index 1: Carver, Michael
      Found Carver, Michael at index 1
```

We can instantly see:

- The recursive narrowing of the search range.
- Which values are checked each time.
- How indentation reflects call depth.

4.3 Common Pitfalls to Watch For

Recursion is a mirror that shows both clarity and chaos. Here are the usual suspects that break your reflection:

1. **No Base Case:** The recursion never ends, causing a stack overflow.
2. **No Return on Recursive Call:** The answer is found, but lost on the way back up.
3. **Misplaced Print:** Messages print at the wrong depth, confusing the trace.

> A well-placed print can save an hour of staring at the screen.

4.4 Try It Yourself

1. Modify `find()` to count how many recursive calls were made.
2. Write a recursive function that prints a pyramid of indentation up to depth 5.
3. Create a recursive version of `sum()` that prints its partial results on the way up and down.
4. Comment out each print one at a time to see which ones are most useful for debugging.

4.5 Bonus Challenge: The "Echo Locator"

Write a recursive function called `echo()` that takes a message and depth:

```
def echo(message, depth):
    if depth == 0:
        print(message)
    else:
        print("␣" * depth + f"Echoing␣({depth})...")
        echo(message, depth - 1)
        print("␣" * depth + "Returning.")
```

Run `echo("Recursion rocks!", 4)` and watch the indentation reveal how recursion dives and returns.

Closing Thought

Debugging recursion is about listening to the rhythm of the calls. Add prints, indent the echoes, and soon you'll be able to *see* your program think.

Next: Creating Recursive Functions — from reflection to design.

Notes

Use this space for your own discoveries and recursive experiments.