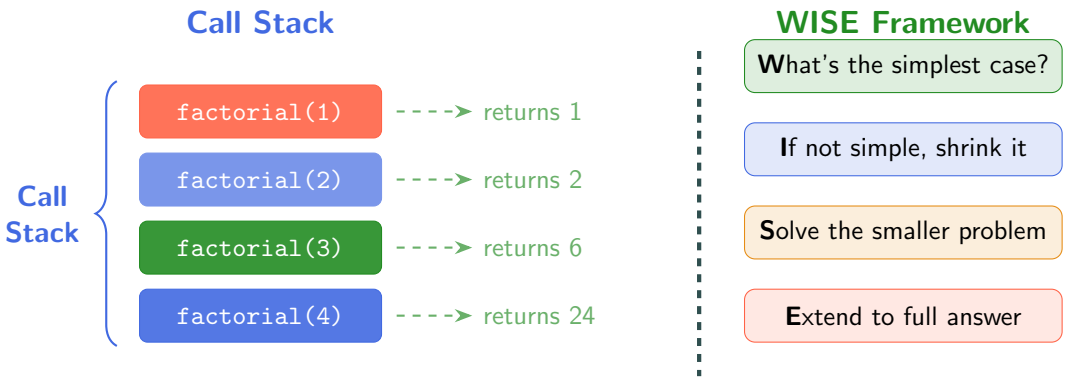
# Programming 2: Lab 03

## Recursion Foundations & Debugging Recursive Functions

A Walkthrough Tutorial




Comp 111 — Forman Christian College

Spring 2026

---

Estimated Time: ∼**3 hours**

## How to Use This Lab

This lab is a **walkthrough tutorial**—it is designed to guide you through learning, not to test you. Work through each section at your own pace:

- **Read** the short explanations and study the diagrams.
- **Type** every code listing yourself (don't copy-paste!). Muscle memory matters.
- **Run** every example and check that your output matches.
- **Complete** the exercises. They are graded by difficulty:
  - ★ Easy — Immediate practice. Follow the pattern you just saw.
  - ★★ Medium — Requires some thinking. Combines concepts.
  - ★★★ Hard — Stretch goal. It's OK to need help!
- **Ask for help** whenever you're stuck for more than 5 minutes.

**Before You Begin**

This lab assumes you completed Lab 02. You should be comfortable with **Git** (staging, committing, pushing), running Python files from the **Command Prompt**, and writing simple **unit tests** with `assert`.

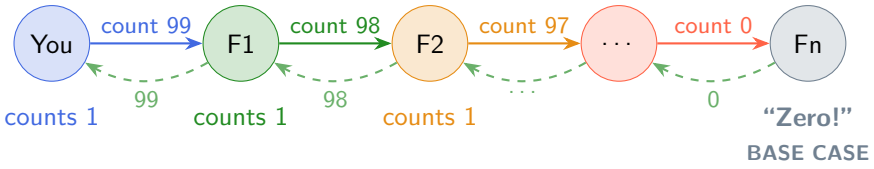
# 1 What is Recursion?

**Recursion** is when a function **calls itself** to solve a smaller version of the same problem. Every recursive function needs exactly two ingredients:

1. **Base case** — when to *stop* (the answer is obvious).
2. **Recursive case** — break the problem into a *smaller* version and call yourself.

**The Lazy Helper Analogy**

Imagine you need to count 100 sheep. Instead of counting them all yourself, you count **one** sheep and ask a friend to count the remaining 99. That friend counts one and asks *their* friend to count 98, and so on. The last person sees **zero** sheep and says "Zero!" (the **base case**). Then the answers bubble back up: 0, 1, 2, . . . , 100.




## 1.1 Walkthrough: Repeat Message

Create a file called `repeat.py` and type the following:

```python
# repeat.py
def repeat_message(msg, n):
    # Base case: nothing left to print
    if n <= 0:
        print("All done!")
        return

    # Recursive case: print once, then repeat fewer times
    print(f"{msg} ({n} left)")
    repeat_message(msg, n - 1)

repeat_message("Hello", 4)
```

Run it:

```
python repeat.py
```

```
Hello (4 left)
Hello (3 left)
Hello (2 left)
Hello (1 left)
All done!
```

Notice the two key parts: the `if n <= 0` check is the **base case** (when to stop), and `repeat_message(msg, n` is the **recursive case** (a smaller version of the same problem).

## 1.2 Exercises: What is Recursion?

1. ★ Easy Type and run the `repeat_message` function above. Verify that your output matches.

2. ★ Easy Change the call to `repeat_message("Hi", 2)`. What output do you get? What about `repeat_message("Hi", 0)`?

3. ★ Easy Write a recursive function `countdown(n)` that prints numbers from `n` down to 1, then prints `"Blastoff!"`. (Hint: the base case is `n <= 0`.)

4. ★★ Medium Write a recursive function `count_up(n)` that prints from 1 *up to* `n`. (Hint: think about how you `print(n)`.)

5. ★★ Medium Write a recursive function `countdown_by_two(n)` that counts down by 2: `10, 8, 6, 4, 2, Blastoff!`. What should the base case be?

6. ★★ Medium What happens if you forget the base case? Remove the `if n <= 0` block from `repeat_message` and try calling `repeat_message("Hi", 3)`. What error do you get? (Press Ctrl+C if it takes too long.)

7. ★★★ Hard Write a recursive function `print_stars(n)` that prints a triangle of stars:

```
*
**
***
****
```

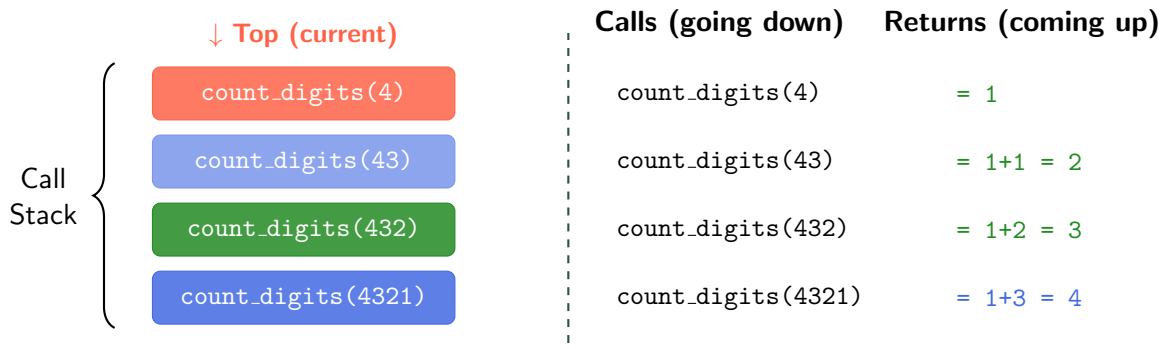
   when called with `print_stars(4)`. (Hint: first recurse to print the smaller triangle, then print a row of `n` stars.)

## 2 The Call Stack

When a function calls itself, Python keeps track of each call on a **call stack**. Think of it as a stack of sticky notes—each call adds a note on top, and when a call finishes, its note is removed. This is **LIFO**: Last In, First Out.

**How the Call Stack Works**

1. Each function call creates a new **stack frame** with its own local variables.
2. Frames stack up as calls go deeper.
3. When a function returns, its frame is **popped off** and the return value goes to the caller.
4. Python limits the stack to ~1000 frames to prevent crashes.




## 2.1 Walkthrough: count_digits with Print Tracing

Create a file called `count_digits.py` and type the following:

```python
# count_digits.py
def count_digits(n):
    print(f"  Calling count_digits({n})")
    # Base case: single digit
    if n < 10:
        print(f"  Base case! Returning 1")
        return 1

    # Recursive case: 1 for current digit + count the rest
    result = 1 + count_digits(n // 10)
    print(f"  count_digits({n}) returning {result}")
    return result

answer = count_digits(4321)
print(f"Final answer: {answer}")
```

Run it:

```
python count_digits.py
```

```
  Calling count_digits(4321)
  Calling count_digits(432)
  Calling count_digits(43)
  Calling count_digits(4)
  Base case! Returning 1
  count_digits(43) returning 2
  count_digits(432) returning 3
  count_digits(4321) returning 4
Final answer: 4
```

Notice how the calls go *down* (4321, 432, 43, 4) and the returns come back *up* (1, 2, 3, 4). Each call strips the last digit using `n // 10`. This is the call stack in action!

## 2.2 Exercises: The Call Stack

1. ★ Easy Type and run the `count_digits` function above. Verify your output matches.
2. ★ Easy On paper, trace the call stack for `count_digits(73)`. Draw each frame as a box and show the return values.
3. ★★ Medium Write a recursive function `factorial(n)` that computes $n! = n \times (n-1) \times$

$\ldots \times 1$ with $0! = 1$. Add print tracing like the walkthrough above. Run `factorial(5)` and verify you get 120.

4. ★★ Medium What happens if you call `count_digits` on a very deeply nested computation like recursive factorial of 1500? Try `factorial(1500)` (without print tracing) and observe the error. What is Python's recursion limit?

5. ★★ Medium Write a recursive function `multiply(a, b)` that multiplies two positive integers using only addition and recursion (no loops). (Hint: $3 \times 4 = 3 + 3 + 3 + 3 = 3 + (3 \times 3)$.)

6. ★★ Medium Write a recursive function `sum_to(n)` that computes $1 + 2 + \ldots + n$. Add print tracing and verify `sum_to(5)` returns 15.

7. ★★ Medium On paper, draw the full call stack for `factorial(5)`. Show each frame being pushed, the base case, and then each frame being popped with its return value. How many frames are on the stack at the deepest point?

# 3 The WISE Framework

When you need to write a recursive function from scratch, follow the **WISE** framework—four questions that guide you to a solution every time:

| | |
|---|---|
| **W** — **W**hat's the simplest case? (base case) | `n < 10 → return n` |
| **I** — **I**f not simple, how do I shrink it? | `n → n // 10` |
| **S** — **S**olve the smaller problem (trust the recursion!) | `sum_digits(n // 10)` |
| **E** — **E**xtend the small solution to the full answer | `(n % 10) + that result` |

**The Recursive Leap of Faith**

When writing recursive code: **trust** that your function will correctly solve the smaller problem. If your base case is correct and you're making the problem smaller, **it will work**. Don't try to trace every call in your head—trust the structure.

## 3.1 Walkthrough: sum_digits with WISE

Let's write a recursive function to sum all the digits of a positive integer. For example, `sum_digits(123)` returns $1 + 2 + 3 = 6$. Create a file called `sum_digits.py`:

```
# sum_digits.py
def sum_digits(n):
    # W: What's the simplest case?
    #    A single digit is its own sum.
    if n < 10:
        return n

```

```python
    # I: If not simple, shrink it.
    #    Remove the last digit: n // 10
    # S: Solve the smaller problem.
    #    sum_digits(n // 10) gives the sum of the remaining digits.
    # E: Extend to full answer.
    #    Add the last digit (n % 10) to the smaller sum.
    return (n % 10) + sum_digits(n // 10)

print(sum_digits(123))      # 6   (1 + 2 + 3)
print(sum_digits(9))        # 9   (single digit)
print(sum_digits(4507))     # 16  (4 + 5 + 0 + 7)
```

```
python sum_digits.py
```

```
6
9
16
```

### 3.2 Exercises: The WISE Framework

1. ★ Easy Type and run the `sum_digits` function above. Test it with at least three different numbers.
2. ★ Easy For the function `string_length(s)` that counts characters *without* using `len()` (e.g., `string_length("hello")` returns 5), answer the four WISE questions *in English* before writing any code.
3. ★★ Medium Now implement `string_length(s)` **recursively** based on your WISE answers. Test it with `string_length("hello")` (expect 5), `string_length("a")` (expect 1), and `string_length("")` (expect 0).
4. ★★ Medium Write a recursive function `count_char(s, c)` that counts how many times the character `c` appears in string `s`. Use WISE: **W:** empty string returns 0; **I:** check first character and shrink; **S:** count in the rest; **E:** add 1 if match, 0 if not.
5. ★★ Medium Write a recursive function `remove_char(s, c)` that returns a new string with all occurrences of `c` removed. Example: `remove_char("hello", "l")` returns `"heo"`.
6. ★★ Medium Write a recursive function `all_digits_even(n)` that returns `True` if every digit of a positive integer is even. Examples: `all_digits_even(2468)` returns `True`; `all_digits_even(2461)` returns `False`. (Hint: check the last digit with `n % 10`.)
7. ★★★ Hard Write a recursive function `replace_char(s, old, new)` that replaces every occurrence of character `old` with `new`. Example: `replace_char("hello", "l", "r")` returns `"herro"`. Do not use Python's built-in `.replace()` method.

## 4 Debugging Recursive Functions

Recursion bugs can be tricky because you can't always "see" what's happening deep in the call stack. Fortunately, 90% of recursion bugs fall into just three categories:

**Missing Return**
`f(n-1)` instead of `return f(n-1)`
→ Returns `None`

**Wrong Base Case**
`if n == 0` when `n` can be negative
→ Infinite recursion

**Not Shrinking**
`f(n)` calls `f(n)` instead of `f(n-1)`
→ RecursionError

**⚠ `print` is NOT `return`!**

A very common beginner mistake: writing `print(result)` in the base case instead of `return result`. The function *displays* the value but returns `None`. The caller then gets `None` instead of the actual answer. **Always use `return` to send values back to the caller.**

## 4.1 Print Tracing with Depth Indentation

Adding **depth indentation** to your print statements makes it easy to see the call stack unfold:

```python
# debug_vowels.py
def count_vowels(s, depth=0):
    indent = "  " * depth
    print(f"{indent}count_vowels('{s}')")

    # Base case
    if s == "":
        print(f"{indent}  -> base case, returning 0")
        return 0

    # Recursive case
    rest = count_vowels(s[1:], depth + 1)
    if s[0].lower() in "aeiou":
        result = 1 + rest
    else:
        result = rest
    print(f"{indent}  -> returning {result}")
    return result

print("Answer:", count_vowels("hey"))
```

```
python debug_vowels.py
```

```
count_vowels('hey')
  count_vowels('ey')
    count_vowels('y')
      count_vowels('')
        -> base case, returning 0
      -> returning 0
    -> returning 1
  -> returning 1
Answer: 1
```

## 4.2 Walkthrough: Finding and Fixing Bugs

Here is a **buggy** function that is supposed to sum a list. It has two bugs—can you spot them?

```python
# buggy_sum.py (BUGGY!)
def sum_list_buggy(lst):
```

```
    if lst == []:
        print(0)              # Bug 1: print instead of return
    sum_list_buggy(lst[1:]) + lst[0]   # Bug 2: missing return
```

**Bug 1:** `print(0)` instead of `return 0`. The base case displays zero but returns `None`.

**Bug 2:** The recursive case computes the answer but doesn't `return` it.

**Fixed version:**

```
# sum_list_fixed.py
def sum_list(lst):
    if lst == []:
        return 0                          # Fix 1: return, not print
    return sum_list(lst[1:]) + lst[0]     # Fix 2: add return

print(sum_list([3, 7, 2]))    # 12
```

## 4.3 Exercises: Debugging Recursive Functions

1. ★ Easy Look at this code and identify the bug *without* running it:

```
def length(s):
    if s == "":
        return 0
    1 + length(s[1:])     # What's missing?
```

2. ★ Easy Look at this code and identify the bug:

```
def repeat(msg, n):
    if n == 0:
        return
    print(msg)
    repeat(msg, n)     # What's wrong here?
```

3. ★ Easy Type and run the buggy `sum_list_buggy` from the walkthrough. What error or output do you see? Then type the fixed version and verify it works.

4. ★★ Medium This function is supposed to count vowels but has a bug. Find and fix it:

```
def count_vowels(s):
    if s == "":
        return 0
    if s[0] in "aeiouAEIOU":
        return 1 + count_vowels(s)     # Bug!
    return count_vowels(s[1:])
```

5. ★★ Medium This function checks if all characters are uppercase. Find and fix the bug:

```
def all_upper(s):
    if s == "":
        return True
    if not s[0].isupper():
        return False
    all_upper(s[1:])     # Bug!
```

6. ★★★ Hard Write a recursive function `debug_product(lst, depth=0)` that computes the product of a list while printing indented trace output (like the depth tracing walkthrough). Test it with `debug_product([2, 3, 5])`.

7. ★★★ Hard This function flattens a nested list but has a bug. Find and fix it:

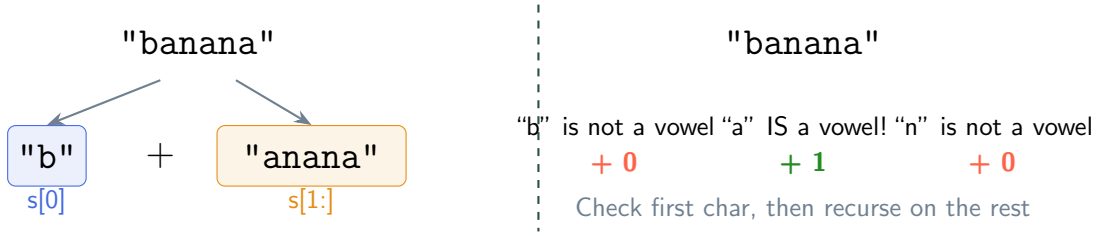
```python
def flatten(lst):
    if lst == []:
        return []
    first = lst[0]
    rest = flatten(lst[1:])
    if isinstance(first, list):
        return flatten(first) + rest
    else:
        return first + rest     # Bug!
```

(Hint: what type is `first`? Can you concatenate an integer with a list?)

8. ★★★ Hard Write a buggy recursive function *on purpose*—include one of the Big Three bugs. Give it to a classmate and see if they can find and fix the bug.

## 5 Recursion on Strings

Strings are a natural fit for recursion because you can always decompose a string into its **first character** and **the rest**:




### 5.1 Walkthrough: count_vowels and remove_char

Create a file called `string_recursion.py`:

```python
# string_recursion.py

def count_vowels(s):
    """Count vowels in a string recursively."""
    if s == "":
        return 0
    rest = count_vowels(s[1:])
    if s[0].lower() in "aeiou":
        return 1 + rest
    return rest

def remove_char(s, c):
    """Remove all occurrences of character c from string s."""
    if s == "":
        return ""
    rest = remove_char(s[1:], c)
    if s[0] == c:
        return rest
    return s[0] + rest

# Test count_vowels
print(count_vowels("banana"))     # 3
print(count_vowels("HELLO"))      # 2
print(count_vowels("xyz"))        # 0
```

```
25
26 # Test remove_char
27 print(remove_char("hello", "l"))      # "heo"
28 print(remove_char("banana", "a"))     # "bnn"
29 print(remove_char("aaa", "a"))        # ""
```

```
python string_recursion.py
```

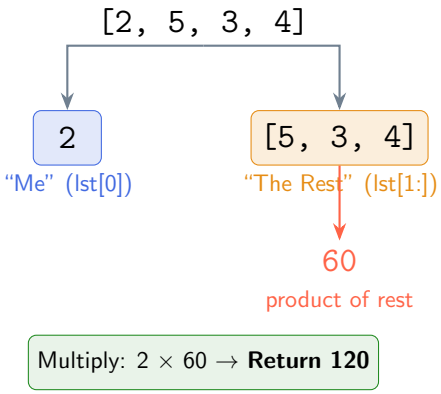
```
3
2
0
heo
bnn
```

### 5.2 Exercises: Recursion on Strings

1. ★ Easy Type and run both functions above. Verify your output matches.
2. ★ Easy Trace `remove_char("cat", "a")` on paper. Write each recursive call and the return value at each step.
3. ★★ Medium Write a recursive function `reverse(s)` that reverses a string. Example: `reverse("hello")` returns `"olleh"`. (Hint: `reverse(s[1:]) + s[0]`.)
4. ★★ Medium Write a recursive function `remove_spaces(s)` that returns a new string with all spaces removed. Example: `remove_spaces("h e l l o")` returns `"hello"`.
5. ★★ Medium Write a recursive function `is_palindrome(s)` that checks if a string reads the same forwards and backwards. Make it case-insensitive: `is_palindrome("Racecar")` should return `True`. (Hint: compare the first and last characters after lowercasing, then recurse on `s[1:-1]`.)
6. ★★★ Hard Write a recursive function `capitalize_words(s)` that capitalizes the first letter of every word. Example: `capitalize_words("hello world")` returns `"Hello World"`. (Hint: find the first space, capitalize the first letter, recurse on the rest.)
7. ★★★ Hard Write a recursive function `interleave(s1, s2)` that interleaves two strings character by character. Example: `interleave("abc", "123")` returns `"a1b2c3"`. If one string is longer, append the remaining characters. (Hint: take one character from each, then recurse.)

## 6 Recursion on Lists

Lists work just like strings for recursion: split into the **first element** (`lst[0]`) and **the rest** (`lst[1:]`). The base case is usually an empty list or a single-element list.




## 6.1 Walkthrough: product_list

Create a file called `list_recursion.py`:

```python
# list_recursion.py
def product_list(lst):
    """Compute the product of all elements in a list recursively."""
    # Base case: single element
    if len(lst) == 1:
        return lst[0]

    # Recursive case: first * product of the rest
    return lst[0] * product_list(lst[1:])

print(product_list([2, 5, 3, 4]))    # 120
print(product_list([7]))             # 7
print(product_list([1, 2, 3, 4]))    # 24
```

```
python list_recursion.py
```

```
120
7
24
```

## 6.2 Exercises: Recursion on Lists

1. ★ Easy Type and run the `product_list` function above. Test it with at least three different lists.

2. ★ Easy Write a recursive function `sum_list(lst)` that returns the sum of all elements in a list. Base case: empty list returns 0.

3. ★★ Medium Write a recursive function `find_max(lst)` that returns the largest element in a list. (Hint: compare the first element with the max of the rest.)

4. ★★ Medium Write a recursive function `count_occurrences(lst, target)` that counts how many times `target` appears in `lst`. Example: `count_occurrences([1, 2, 1, 3, 1], 1)` returns 3.

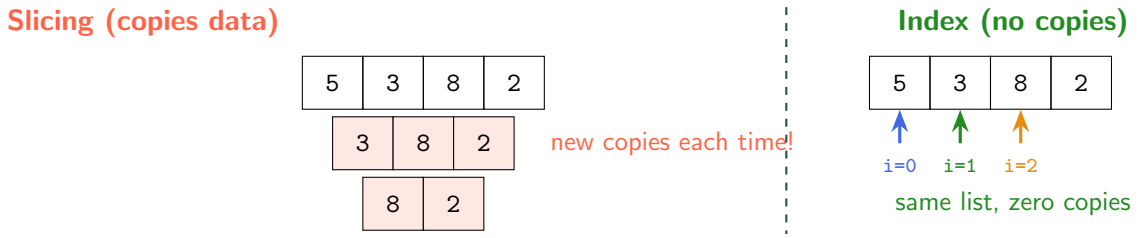
5. ★★ Medium Write a recursive function `contains(lst, target)` that returns `True` if `target` is in the list, `False` otherwise. Do not use the `in` operator.

6. ★★★ Hard Write a recursive function `is_sorted(lst)` that returns `True` if the list is in ascending order. Examples: `is_sorted([1, 3, 5])` returns `True`; `is_sorted([1, 5, 3])` returns `False`. (Hint: compare the first two elements, then recurse on the rest.)

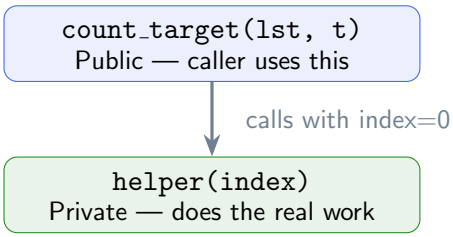
7. ★★★ Hard Write a recursive function `flatten(lst)` that takes a nested list and returns a flat list. Example: `flatten([1, [2, 3], [4, [5, 6]]])` returns `[1, 2, 3, 4, 5, 6]`. (Hint: use `isinstance(item, list)` to check if an element is a list.)

# 7 Helper Functions and Avoiding Slicing

Every time you write `lst[1:]`, Python creates a **brand new list**. For a list of $n$ items, this means $n + (n-1) + (n-2) + \ldots$ copies—that's $O(n^2)$ work! The fix: use an **index** that moves through the *same* list.




But now the function needs an extra `index` parameter that the caller shouldn't have to provide. The solution is a **helper function**:




## 7.1 Walkthrough: count_target — Slice vs. Helper

Create a file called `helper_demo.py`:

```
# helper_demo.py

# Version 1: Slicing (simple but slow for large lists)
def count_target_slice(lst, t):
    if lst == []:
        return 0
    match = 1 if lst[0] == t else 0
    return match + count_target_slice(lst[1:], t)

# Version 2: Helper function (efficient, no copies)
def count_target(lst, t):
    def helper(index):
        if index == len(lst):
            return 0
        match = 1 if lst[index] == t else 0
        return match + helper(index + 1)
```

```
    return helper(0)

# Both give the same answer
print(count_target_slice([1, 3, 1, 2, 1], 1))    # 3
print(count_target([1, 3, 1, 2, 1], 1))           # 3
```

```
python helper_demo.py
```

```
3
3
```

The caller writes `count_target([1, 3, 1, 2, 1], 1)`—clean and simple. The `index` is an **implementation detail** hidden inside the helper.

### 7.2 Exercises: Helper Functions

1. ★ Easy Type and run both versions of `count_target` above. Verify they give the same results.
2. ★ Easy Rewrite `product_list(lst)` from Section 6 using a helper function with an index parameter instead of slicing.
3. ★★ Medium Write a recursive function `count_vowels_helper(s)` using a nested helper function with an index instead of `s[1:]` slicing. Test it with the same inputs as Section 5.
4. ★★ Medium Write a recursive function `linear_search(lst, target)` that returns the **index** of `target` in `lst`, or `-1` if not found. Use a helper that carries the current index.
5. ★★★ Hard Write a recursive function `is_palindrome_helper(s)` using a helper with two indices: `left` starting at 0 and `right` starting at `len(s) - 1`, moving toward the center.
6. ★★★ Hard Write a recursive function `binary_search(lst, target)` that searches a **sorted** list for `target`. Use a helper with `low` and `high` parameters. Return the index of `target` or `-1` if not found. Test with `binary_search([1, 3, 5, 7, 9], 7)` (expect 3).

## 8 Recursion vs. Loops

Both recursion and loops can solve repetitive problems, but each has strengths:

**Recursion**

✓ Naturally fits trees, nesting
✓ Divide and conquer
✓ Backtracking algorithms
✗ 1000 call limit in Python
✗ Stack overhead per call

**Loops**

✓ Simple counting/accumulating
✓ No stack limit
✓ Performance critical code
✗ Can't handle arbitrary nesting
✗ Harder for divide & conquer

## 8.1 Walkthrough: Power — Recursive vs. Loop

Create a file called `compare.py`:

```python
# compare.py

# Recursive version
def power_recursive(base, exp):
    if exp == 0:
        return 1
    return base * power_recursive(base, exp - 1)

# Loop version
def power_loop(base, exp):
    result = 1
    for i in range(exp):
        result *= base
    return result

# Both give the same answer
print(power_recursive(2, 10))     # 1024
print(power_loop(2, 10))          # 1024

# But try a large exponent...
# print(power_recursive(2, 2000))  # RecursionError!
print(power_loop(2, 2000))          # Works fine (very large number)
```

```
python compare.py
```

```
1024
1024
```

For simple accumulation like computing a power, a loop is more practical. But for problems with recursive structure (trees, nested lists, divide and conquer), recursion is the natural choice.

**Use recursion when the problem "looks like itself at a smaller scale." Use loops when you're simply counting or accumulating. Many problems can be solved either way—choose the clearer approach.**

## 8.2 Exercises: Recursion vs. Loops

1. ★ Easy Type and run `compare.py` above. Uncomment the `power_recursive(2, 2000)` line. What error do you get?

2. ★ Easy Write a loop version `sum_to_loop(n)` that computes $1 + 2 + \ldots + n$. Compare the result with a recursive `sum_to(n)`.

3. ★★ Medium Write a loop version `reverse_loop(s)` that reverses a string using a `for` loop. Compare it with a recursive `reverse(s)` from Section 5.

4. ★★ Medium Which approach (recursion or loop) would you use for each scenario? Explain why.

   - Counting words in a sentence
   - Traversing all files in nested folders

- Computing the sum of a list of numbers
- Checking if a nested list (like `[1, [2, [3]]]`) contains a target value

**5.** ★★★ Hard Write both a recursive and a loop version of the Fibonacci sequence: `fib(n)` returns the $n$-th Fibonacci number (0, 1, 1, 2, 3, 5, 8, ...). Time both versions with `fib(30)` using:

```
import time
start = time.time()
result = fib(30)
print(f"Result: {result}, Time: {time.time() - start:.4f}s")
```

Which is faster? Why?

**6.** ★★★ Hard The recursive Fibonacci is slow because it recomputes the same values many times. Can you think of a way to make it faster? (Hint: what if you "remembered" values you already computed? We'll cover this technique in a future lab.)

## 9 Putting It All Together: The Recursive Text Analyzer

**Capstone Challenge**

This section combines **everything** you've learned: recursion, WISE framework, debugging, helper functions, and testing. Take it one step at a time. If you get stuck, re-read the relevant section above.

You will build a small **Recursive Text Analyzer**—a collection of functions that analyze text using *only recursion* (no loops allowed!).

**Project structure:**

```
text-analyzer/
├── analyzer.py
├── test_analyzer.py
└── main.py
```

Follow these steps:

**Step 1. Create the project folder:**

```
mkdir text-analyzer
cd text-analyzer
```

**Step 2. Create `analyzer.py`.** Type the following functions—all recursive, no loops:

```
# analyzer.py

def char_count(s):
    """Count total characters (like len, but recursive)."""
    if s == "":
```

```
        return 0
    return 1 + char_count(s[1:])

def word_count(s):
    """Count words (split by spaces)."""
    s = s.strip()
    if s == "":
        return 0
    # Find the first space
    space = find_char(s, " ")
    if space == -1:
        return 1   # No space = one word left
    return 1 + word_count(s[space + 1:])

def find_char(s, c):
    """Find index of character c in string s. Return -1 if not found."""
    def helper(index):
        if index == len(s):
            return -1
        if s[index] == c:
            return index
        return helper(index + 1)
    return helper(0)

def clean_text(s):
    """Remove all non-letter, non-space characters and lowercase."""
    if s == "":
        return ""
    first = s[0].lower()
    rest = clean_text(s[1:])
    if first.isalpha() or first == " ":
        return first + rest
    return rest

def is_palindrome_phrase(s):
    """Check if a phrase is a palindrome (ignoring spaces/punctuation).
       """
    cleaned = clean_text(s).replace(" ", "")
    def helper(left, right):
        if left >= right:
            return True
        if cleaned[left] != cleaned[right]:
            return False
        return helper(left + 1, right - 1)
    return helper(0, len(cleaned) - 1)

def most_common(s):
    """Find the most common letter (lowercase, ignoring spaces)."""
    cleaned = clean_text(s).replace(" ", "")
    if cleaned == "":
        return ""

    def count_in(text, c):
        if text == "":
            return 0
        return (1 if text[0] == c else 0) + count_in(text[1:], c)

    def best(text, current_best, best_count):
        if text == "":
            return current_best
        c = text[0]
        c_count = count_in(cleaned, c)
        if c_count > best_count:
```

```
            return best(text[1:], c, c_count)
        return best(text[1:], current_best, best_count)

    return best(cleaned, cleaned[0], 0)
```

**Step 3. Create `test_analyzer.py`.** Write tests for each function:

```
# test_analyzer.py
from analyzer import (char_count, word_count, find_char,
                      clean_text, is_palindrome_phrase,
                      most_common)

def test_char_count():
    assert char_count("hello") == 5
    assert char_count("") == 0
    assert char_count("a") == 1
    print("test_char_count: ALL PASSED")

def test_word_count():
    assert word_count("hello world") == 2
    assert word_count("one") == 1
    assert word_count("") == 0
    assert word_count("  spaces  ") == 1
    print("test_word_count: ALL PASSED")

def test_find_char():
    assert find_char("hello", "e") == 1
    assert find_char("hello", "z") == -1
    assert find_char("hello", "h") == 0
    print("test_find_char: ALL PASSED")

def test_clean_text():
    assert clean_text("Hello, World!") == "hello world"
    assert clean_text("abc123") == "abc"
    assert clean_text("") == ""
    print("test_clean_text: ALL PASSED")

def test_is_palindrome_phrase():
    assert is_palindrome_phrase("racecar") == True
    assert is_palindrome_phrase("A man a plan a canal Panama") == True
    assert is_palindrome_phrase("hello") == False
    print("test_is_palindrome_phrase: ALL PASSED")

def test_most_common():
    assert most_common("aabbc") == "a"
    assert most_common("hello") == "l"
    print("test_most_common: ALL PASSED")

# Run all tests
test_char_count()
test_word_count()
test_find_char()
test_clean_text()
test_is_palindrome_phrase()
test_most_common()
print("--- All tests passed! ---")
```

**Step 4. Run the tests:**

```
python test_analyzer.py
```

```
test_char_count: ALL PASSED
```

```
test_word_count: ALL PASSED
test_find_char: ALL PASSED
test_clean_text: ALL PASSED
test_is_palindrome_phrase: ALL PASSED
test_most_common: ALL PASSED
--- All tests passed! ---
```

If any test fails, use print tracing (Section 4) to debug!

**Step 5. Create `main.py`.** A demo program that uses your analyzer:

```python
# main.py
from analyzer import (char_count, word_count, clean_text,
                      is_palindrome_phrase, most_common)

text = "A man a plan a canal Panama"

print("=== Recursive Text Analyzer ===")
print(f"Text: '{text}'")
print(f"Characters: {char_count(text)}")
print(f"Words: {word_count(text)}")
print(f"Cleaned: '{clean_text(text)}'")
print(f"Palindrome? {is_palindrome_phrase(text)}")
print(f"Most common letter: '{most_common(text)}'")
```

**Step 6. Run the demo:**

```
python main.py
```

```
=== Recursive Text Analyzer ===
Text: 'A man a plan a canal Panama'
Characters: 27
Words: 7
Cleaned: 'a man a plan a canal panama'
Palindrome? True
Most common letter: 'a'
```

**Don't Forget to Commit!**

You've built a complete project. If you set up Git in Lab 02, commit your work:

```
git add analyzer.py test_analyzer.py main.py
git commit -m "Add recursive text analyzer with tests"
```

## Congratulations!

### Skills Unlocked!

**Recursion Foundations**

- ✓ Base cases and recursive cases
- ✓ Call stack tracing
- ✓ The WISE framework
- ✓ Recursion on numbers
- ✓ Recursion on strings
- ✓ Recursion on lists
- ✓ The Recursive Leap of Faith

**Debugging & Practical Skills**

- ✓ The Big Three recursion bugs
- ✓ Print tracing with depth
- ✓ Helper functions
- ✓ Avoiding slicing overhead
- ✓ Recursion vs. loops
- ✓ Testing recursive functions
- ✓ Building a recursive project

**Coming Up Next:** **Recursion Applications** — divide and conquer, binary search, and merge sort!