Tutorial 9

Recursion: smaller problems, call stack, and classic examples

18YZALG – Basics of Algorithmization, Summer Semester 2026

Today

1. What recursion means and when it is useful.
2. The call stack mental model.
3. Classic examples on numbers, arrays, and strings.
4. Divide-and-conquer example: recursive binary search.
5. Nested structures: a tree example.
6. Common bugs, debugging, and recursion vs iteration.

Why recursion at all?

- Some problems contain a smaller version of themselves.
- Some data is naturally nested: folders inside folders, nodes inside trees, expressions inside parentheses.
- Some algorithms repeatedly shrink the search space: for example, binary search keeps only one half.

Main idea

Recursion is useful when the problem has a **self-similar structure**: solve a smaller instance of the *same kind* of problem, then finish the current one.

Working definition

Recursion

A function is recursive if it solves a problem by calling **itself** on a **smaller input** until it reaches a case that can be solved directly.

Important clarification

The function is not "doing everything at once". Each call handles **one instance** of the same contract, with its own parameters and local variables.

The three ingredients of safe recursion

1. Base case

Smallest input where the answer is immediate.

2. Recursive case

Call the same function on a smaller instance.

3. Progress

Every recursive call moves toward the base case.

If one ingredient is missing ...

No base case → infinite recursion. No progress → infinite recursion. Wrong combine step → wrong answer.

The recursive leap of faith

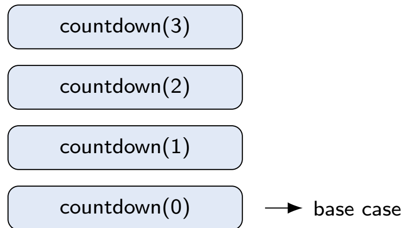
- When designing recursion, **assume** the recursive call already works correctly on the smaller input.
- Then focus only on the current step: **which smaller input** do I call on, and **how do I use its answer**?
- This stops you from trying to mentally expand the whole call tree every time.

A good question to ask

"If I trust the smaller call, what remains for the current frame to do?"

Mental model: the call stack

- Each function call gets its own **parameters**, **local variables**, and **return address**.
- Active calls are stored in a stack: the most recent call returns first.
- Recursion ends when the deepest call reaches the base case and returns upward.



Warm-up example: countdown

Code

```
def countdown(n):
    if n == 0:
        print("Lift off!")
        return
    print(n)
    countdown(n - 1)
```

Questions

What is the base case? Why is the recursive call smaller? What will `countdown(3)` print?

Tracing `countdown(3)`

`Call`	**What this frame does**
`countdown(3)`	print 3, then call `countdown(2)`
`countdown(2)`	print 2, then call `countdown(1)`
`countdown(1)`	print 1, then call `countdown(0)`
`countdown(0)`	print `Lift off!`, then return

Key observation

Because `print(n)` happens **before** the recursive call, the numbers appear on the way **down**. If we moved it after the call, they would appear on the way **up**.

Example 1: factorial

Specification

```
factorial(n)
Input: n >= 0
Output: n!
```

Recursive code

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

Contract

`factorial(n)` returns the product $1 \cdot 2 \cdot 3 \cdots n$.

How `factorial(4)` returns

Expansion

$$\begin{aligned}
\texttt{factorial(4)} &= 4 \cdot \texttt{factorial(3)} \\
&= 4 \cdot 3 \cdot \texttt{factorial(2)} \\
&= 4 \cdot 3 \cdot 2 \cdot \texttt{factorial(1)} \\
&= 4 \cdot 3 \cdot 2 \cdot 1 \cdot \texttt{factorial(0)} \\
&= 4 \cdot 3 \cdot 2 \cdot 1 \cdot 1 = 24
\end{aligned}$$

Call stack story

Calls go **down** until `n = 0`.
Answers come **back up**:

$$1 \to 1 \to 2 \to 6 \to 24$$

A recipe for designing recursive functions

1. Write a precise contract: what should `f(x)` return?
2. Choose the smallest easy input and solve it directly.
3. Decide how to create one smaller instance of the same problem.
4. Use the answer from the smaller call to finish the current one.
5. Test tiny inputs first: base case, one step above base, then a normal case.

Example 2: sum of the first n elements

Recursive idea

```
def sum_first(A, n):
    if n == 0:
        return 0
    return sum_first(A, n - 1) + A[n - 1]
```

Contract

`sum_first(A, n)` returns the sum of `A[0] + A[1] + ... + A[n-1]`.
The smaller problem is "sum of the first $n-1$ elements".

Tracing `sum_first([4,1,7,2], 4)`

Expansion

$$
\begin{aligned}
\texttt{sum_first(A,4)} &= \texttt{sum_first(A,3)} + 2 \\
&= \texttt{sum_first(A,2)} + 7 + 2 \\
&= \texttt{sum_first(A,1)} + 1 + 7 + 2 \\
&= \texttt{sum_first(A,0)} + 4 + 1 + 7 + 2 \\
&= 0 + 4 + 1 + 7 + 2 = 14
\end{aligned}
$$

Why this example matters

The contract is not "sum the whole array". It is **sum the first *n* elements**. Good recursion starts with a good contract.

Example 3: reverse a string

Code

```
def reverse_string(s):
    if s == "":
        return ""
    return reverse_string(s[1:]) + s[0]
```

- Base case: empty string.
- Smaller problem: reverse everything except the first character.
- Combine step: put the first character at the **end**.

Quick check: understand the string example

Questions

For `reverse_string("DOG")`, answer these before we compute anything:

1. What is the base case?
2. Which expression creates the smaller input?
3. Why is the result `"GOD"` and not `"ODG"`?

Hint

The order of concatenation matters: `reverse_string(s[1:]) + s[0]`.

Not all recursion removes just one element

- In `factorial` and `sum_first`, the problem shrank by one step.
- In many algorithms, recursion shrinks the problem much faster by discarding a large part at once.
- Binary search is the standard example: keep only the half that could still contain the answer.

Still the same principle

It is still recursion because we solve the **same kind of problem** on a **smaller input interval**.

Example 4: recursive binary search

Code

```
def binary_search(A, left, right, x):
    if left > right:
        return -1
    mid = (left + right) // 2
    if A[mid] == x:
        return mid
    if x < A[mid]:
        return binary_search(A, left, mid - 1, x)
    return binary_search(A, mid + 1, right, x)
```

Precondition

A must already be **sorted**. Otherwise the "discard half" logic is invalid.

Tracing a binary search call

Search for 23 in $[3, 5, 8, 12, 19, 23, 31, 44]$

left	right	mid	A[mid]	Decision
0	7	3	12	keep right half
4	7	5	23	found

What got smaller?

Not the whole array object, but the **active search interval** $[left, right]$.

Where recursion feels natural

- Mathematical definitions: factorial, powers, Fibonacci-style definitions.
- Prefix / suffix problems on strings or arrays.
- Divide-and-conquer algorithms.
- Tree-shaped or nested structures.

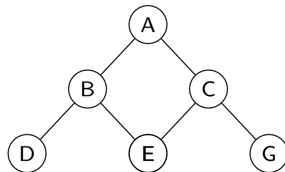
Examples from real tasks

- Count files in nested folders.
- Evaluate an arithmetic expression tree.
- Search a node in an organizational chart.

Example 5: count nodes in a binary tree

Code

```
def tree_size(node):
    if node is None:
        return 0
    return 1 + tree_size(node.left) \
             + tree_size(node.right)
```



Idea

Count the current node, then count the left subtree and the right subtree.

Why the tree example is so natural

- A tree is already built from smaller trees.
- The base case is clear: an empty child contributes 0.
- The recursive rule matches the definition of the object itself.

This is the big picture

Recursion often feels easiest when the **shape of the data** already suggests a recursive definition.

Common bugs in recursion

1. **Missing or wrong base case**: the function never knows when to stop.
2. **No progress**: the recursive call does not move closer to the base case.
3. **Forgetting to return or combine**: the smaller answer is computed, but never used correctly.
4. **Wrong contract**: the recursive call solves a different problem than the current frame expects.

Debugging checklist

- Test the smallest inputs first.
- Ask: does **every path** eventually reach a base case?
- Ask: is the input **strictly smaller** in each recursive call?
- Write one trace by hand: call values on the way down, return values on the way up.
- If needed, print the current parameters at function entry while debugging.

Useful habit

When recursion fails, do not guess. Trace two or three calls exactly.

Recursion vs iteration

Task	Recursion	Iteration
Countdown / factorial	Clear and short	Also easy
Sum many consecutive values	Usually less natural	Often simplest
Binary search	Elegant and short	Also common in practice
Tree traversal / nested structures	Very natural	Usually more bookkeeping
Very deep linear problems	Risky in Python	Usually safer

Rule of thumb

Prefer recursion when it makes the structure of the solution **clearer**. Prefer iteration for long, simple loops with no natural nesting.

A practical note about Python

- Every recursive call uses extra stack space.
- Python intentionally limits recursion depth to avoid a crash.
- So recursion is excellent for **conceptual clarity**, but not every deeply repeated process should be written recursively in Python.

Translation

Recursive *thinking* is always useful. Recursive *code* is useful when the depth stays reasonable.

Quick check: recursion or iteration?

For each task, choose a likely good first approach

1. Compute `factorial(6)`.
2. Add all numbers from 1 to 1,000,000.
3. Count nodes in a tree-shaped structure.
4. Search for 42 in a sorted array.

Quick check: spot the bug

Bug A

```
def f(n):
    if n == 0:
        return 0
    f(n - 1) + n
```

Bug B

```
def g(n):
    if n == 0:
        return 0
    return g(n) + 1
```

Questions

What is wrong in each function? Which one misses a **return**, and which one makes **no progress**?