Tail Recursion

Ibrahim Nasser

October 2025

1 Introduction

Recursion means a function calls itself to solve smaller parts of a problem until it reaches a **base case**, which stops the recursion. Every recursive function needs:

1. A **base case** (when to stop).
2. A **recursive case** (when to keep calling itself).

Example 1: Factorial

Mathematically:

$$n! = n \times (n-1)! \quad \text{with the base case} \quad 0! = 1$$

In Python:

```
def factorial(n):
    # Base case
    if n == 0:
        return 1
    # Recursive case
    return n * factorial(n - 1)
```

Walkthrough for `factorial(4)`:

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

Each call pauses until the smaller call finishes. The recursion "unwinds" once it hits the base case.

Example 2: Fibonacci Sequence

The Fibonacci sequence is defined as:

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

In Python:

```
def fibonacci(n):
    # Base cases
    if n == 0:
        return 0
    elif n == 1:
        return 1
    # Recursive case
    return fibonacci(n - 1) + fibonacci(n - 2)
```

Walkthrough for `fibonacci(4)`:

$$\begin{aligned} fibonacci(4) &= fibonacci(3) + fibonacci(2) \\ &= (fibonacci(2) + fibonacci(1)) + (fibonacci(1) + fibonacci(0)) \\ &= ((fibonacci(1) + fibonacci(0)) + 1) + (1 + 0) \\ &= ((1 + 0) + 1) + (1 + 0) = 3 \end{aligned}$$

Here you can see multiple recursive branches forming a call tree, because each call spawns two more calls.

2 Tail Recursion

In **tail recursion**, the recursive call is the *last operation* performed in the function. There is no computation left after the recursive call returns. This allows certain languages (not Python, but others like Scheme or Prolog) to reuse the same stack frame, converting recursion into an iterative process internally. This is known as **tail call optimization (TCO)**.

Tail-Recursive Factorial

A clean way to express tail recursion is by defining a main function that delegates to a helper carrying an accumulator. The helper performs the recursion, while the main function provides the initial accumulator value.

Here is a tail recursive implementation of the `fact` function:

```
def fact(n):
    def fact_helper(n, acc):
        # Base case
        if n == 0:
            return acc
        # Tail-recursive case
        return fact_helper(n - 1, n * acc)
    return fact_helper(n, 1)
```

Walkthrough for `fact(4)`:

$$\begin{aligned} fact(4) &= fact\_helper(4, 1) \\ &\to fact\_helper(3, 4) \\ &\to fact\_helper(2, 12) \\ &\to fact\_helper(1, 24) \\ &\to fact\_helper(0, 24) = 24 \end{aligned}$$

Each recursive call directly returns another call without additional computation. This makes the recursion tail-recursive, and languages with TCO can execute it in constant stack space.

Tail-Recursive Fibonacci

We can apply the same structure to the Fibonacci sequence. The helper carries the previous two Fibonacci numbers as parameters `(a, b)` (think of them as two pointers, `prev` and `current`).

```
def fib(n):
    def fib_helper(n, a, b):
        # Base case
        if n == 0:
            return a
        # Tail-recursive case
        return fib_helper(n - 1, b, a + b)
    return fib_helper(n, 0, 1)
```

Walkthrough for `fib(5)`:

$$\begin{aligned} fib(5) &= fib\_helper(5, 0, 1) \\ &\to fib\_helper(4, 1, 1) \\ &\to fib\_helper(3, 1, 2) \\ &\to fib\_helper(2, 2, 3) \\ &\to fib\_helper(1, 3, 5) \\ &\to fib\_helper(0, 5, 8) = 5 \end{aligned}$$

Here, `a` represents the previous Fibonacci number and `b` the current one. Each call shifts these accumulators forward until the base case is reached. See Fig 1 for illustration.

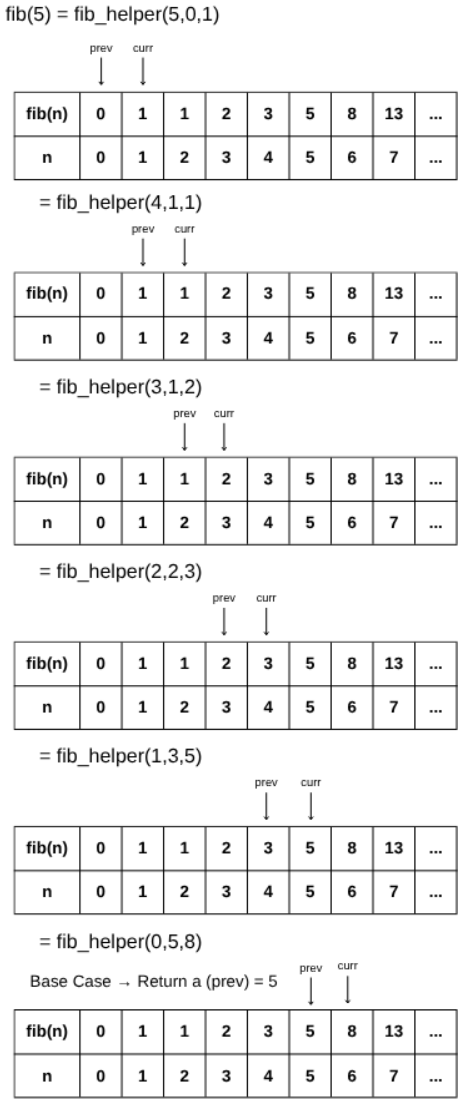

Figure 1: Fib Tail Recursion