

Course Algorithms and Data Structures · Lecture 5

Recursion

Textbook: A. Levitin, Sections 2.4, 2.5, 4.4

The idea

You stand in a long queue and want to know your position. You ask the person in front of you, they ask the person in front of them... The first person says “I’m number 1”, and the answers come back.

Base case

a small problem solved directly (the first person in the queue)

Recursive case

the same problem, but smaller (ask the person in front, add 1)

Every recursive call must move closer to the base case.

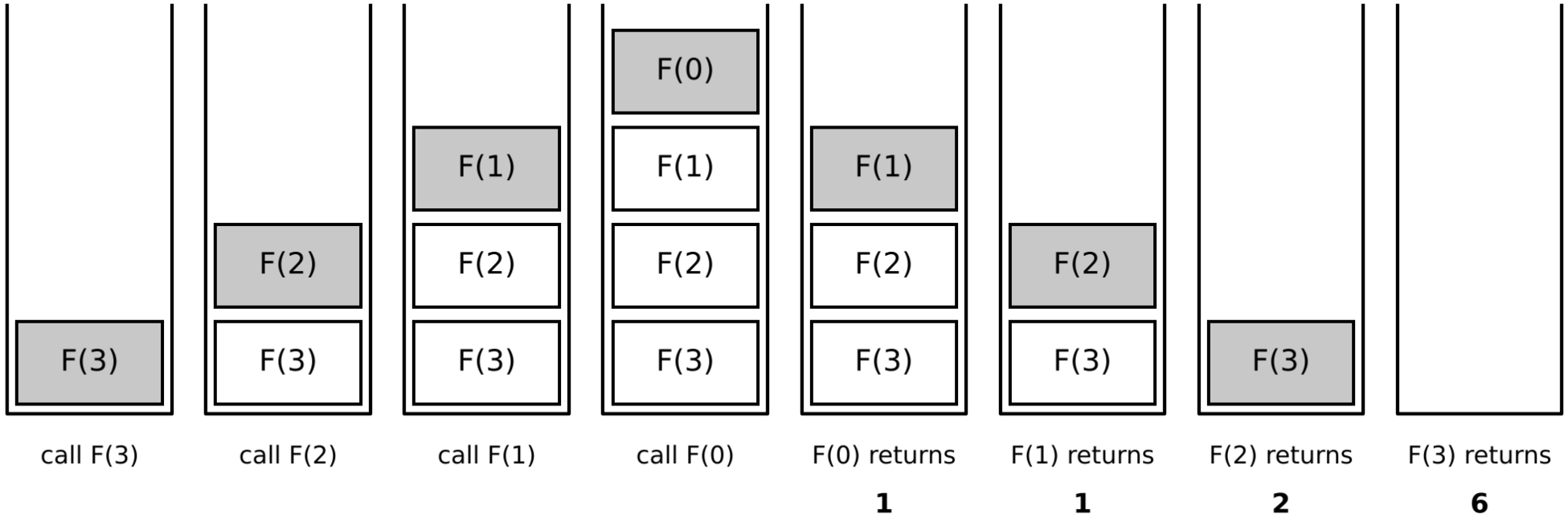
Example: factorial

$$n! = (n - 1)! \cdot n,\ 0! = 1$$

```
ALGORITHM F(n)
    //Computes n! recursively
    if n = 0 return 1          //base case
    else return F(n − 1) * n   //recursive case
```

$$F(3) = F(2) \cdot 3 = (F(1) \cdot 2) \cdot 3 = ((F(0) \cdot 1) \cdot 2) \cdot 3 = 6$$

How the computer does it: the call stack



Calls go **down** until the base case, then the answers come back **up**.

Recursion uses memory

- Each waiting call keeps a **stack frame**
- F(n) has n + 1 frames at the deepest point: **O(n) memory**; a loop needs O(1)
- The stack is limited: too deep, or no base case → **stack overflow**
- Python’s default limit is about 1000 calls (RecursionError)

Counting operations: a recurrence

M(n) = number of multiplications in F(n):

$$\mathbf{M(n) = M(n - 1) + 1,\ M(0) = 0}$$

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

So factorial is O(n), like a loop. This “unrolling” is the standard method.

Sum of an array, recursively

```
ALGORITHM Sum(A[0..n − 1], n)
    //Returns A[0] + A[1] + … + A[n − 1]
    if n = 0 return 0
    else return Sum(A, n − 1) + A[n − 1]
```

Same pattern as the factorial: n additions, O(n) time, O(n) stack memory.

Binary search, recursively

```
ALGORITHM BinSearchRec(A[0..n − 1], l, r, K)
    if l > r return −1              //empty part
    m ← [(l + r) / 2]
    if K = A[m] return m            //found
    else if K < A[m]
        return BinSearchRec(A, l, m − 1, K)
    else return BinSearchRec(A, m + 1, r, K)
```

Each call works on half the part: about $\log_2 n$ calls, $O(\log n)$.

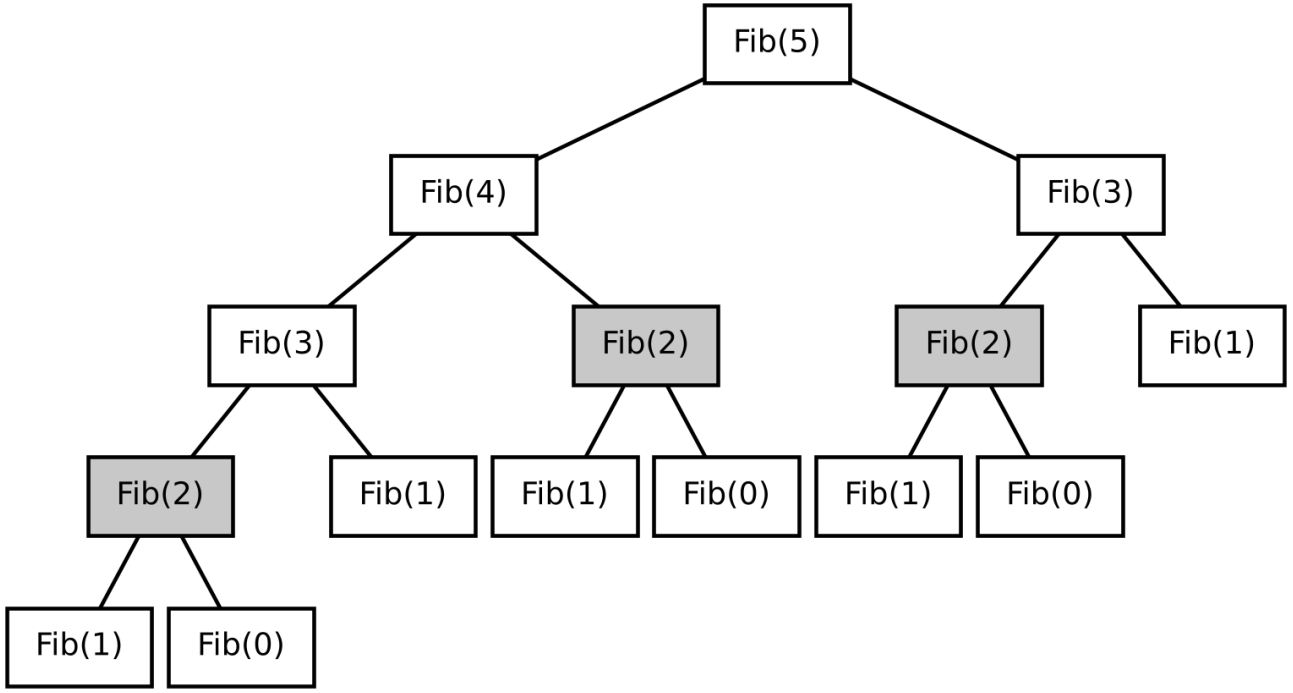
Fibonacci numbers

0, 1, 1, 2, 3, 5, 8, 13, ... each number is the sum of the two before it

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

```
ALGORITHM Fib(n)
    if n ≤ 1 return n
    else return Fib(n − 1) + Fib(n − 2)
```

Why naive recursion is slow



Fib(2) is computed 3 times. The work grows like 1.6^n: Fib(50) needs more than 10^{10} additions.

The fix: a loop

```
ALGORITHM FibLoop(n)
    if n ≤ 1 return n
    a ← 0;  b ← 1
    for i ← 2 to n do
        c ← a + b
        a ← b
        b ← c
    return b
```

Fib(50): 49 additions instead of 10^{10}

Recursion or a loop?

	Recursion	Loop
Code	often shorter, close to the definition	sometimes longer
Memory	a frame for every active call	usually O(1)
Danger	stack overflow, repeated work	infinite loop
Natural for	trees, divide and conquer, graphs	counting, going through an array

Common mistakes

- **No base case**, or one that is never reached
- The recursive call is **not smaller**
- **Wrong base value**: if F(0) returned 0, every factorial would be 0
- **Repeated subproblems**, like naive Fibonacci: correct but exponentially slow

Summary

- A recursive algorithm solves a smaller copy of the same problem; it needs a base case and a recursive case.
- Waiting calls are kept on the call stack, so recursion uses memory.
- Write a recurrence and unroll it: factorial O(n), binary search O(log n).
- Repeated subproblems make recursion exponential; a loop fixes naive Fibonacci.

Review questions

1. What are the two parts of every recursive algorithm? What happens if the base case is missing?
2. Draw the call stack for F(4), the recursive factorial. How many frames are on the stack at the deepest point?
3. Why is the recursive Fibonacci algorithm slow? How can we make it fast?

Reading: Levitin, Sections 2.4, 2.5, 4.4; Bhargava, Chapter 3.