Data Structures

Week 12 – Recursion: reducing a problem to a smaller instance of itself

Dr. Ferenc Koczka · Eszterházy Károly University · 2026

Review

- **Call stack** (week 7): every function call pushes a frame onto the stack, returning pops it
- **Linked list:** one element + the rest of the list (which is also a list!)
- **Binary search:** we halve the problem and continue in the same way

Questions for today

- What is recursion, and how does it work in the computer?
- How do we write a correct recursive function?
- When is recursion good, and when is it dangerous?

What is recursion?

Recursion: a function **calls itself** on a **smaller** instance of the problem.

Every recursive function has two parts:

1. **Base case:** the simplest case, solved directly – here the recursion **stops**
2. **Recursive case:** the problem is reduced to a **smaller** instance of the same problem

Factorial: $n! = 1 \cdot 2 \cdot \ldots \cdot n$

$0! = 1$ **(base case)**

$n! = n \cdot (n - 1)!$ **(recursive case)**

In mathematics its counterpart is **mathematical induction**: if the base case is true, and the larger case follows from the smaller one, then it is true for every case.

Factorial

Pseudocode

```
Function Fact(n)
  If n ≤ 1 then
    Return 1                    // base case
  else
    Return n · Fact(n − 1)      // recursive case
  End if
End function
```

Java

```java
static long fact(int n) {
    if (n <= 1) return 1;
    return n * fact(n - 1);
}
```

Go

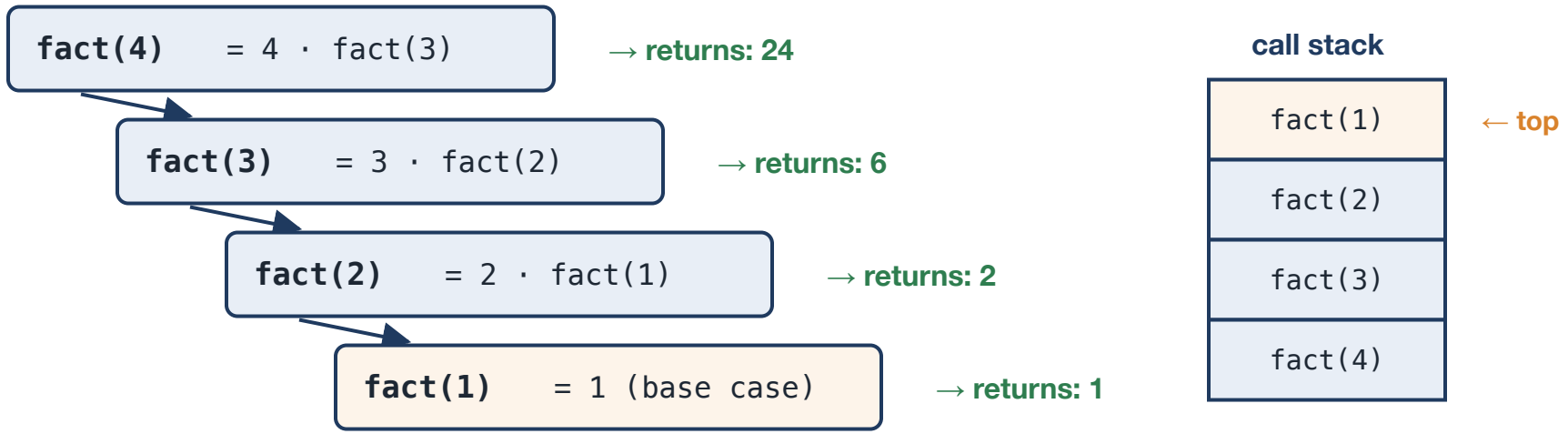
```go
func fact(n int) int64 {
    if n <= 1 {
        return 1
    }
    return int64(n) * fact(n-1)
}
```

Output of the program (`Factorial`):

```
fact(4) called
  fact(3) called
    fact(2) called
      fact(1) called
      fact(1) = 1
    fact(2) = 2
  fact(3) = 6
fact(4) = 24
```

How does it work? The call stack

Evaluating fact(4): calls going down (stack grows), returns going up (stack shrinks)



- Every call gets its own **frame**: its own parameter **n**, its own return address
- The multiplications are performed only **backwards** (while returning)
- Space: **O(n)** – n frames on the stack

Correct recursion – three rules

1. **There is a base case** that gives a result without recursion
2. Every recursive call **gets closer** to the base case (smaller n, shorter list, smaller range)
3. **Trust the recursion:** assume that the call returns the correct result for the smaller problem, and only think about how the solution of the larger problem is built from it

Without a base case, or without getting closer to it: **infinite recursion** → the stack fills up Java: `StackOverflowError` (after about 10 000–30 000 calls) · Go: the stack grows, but crashes at 1 GB

Kinds of recursion

Kind	Meaning	Example
Direct	f calls itself	factorial
Indirect	f calls g, g calls f	even(n) → odd(n – 1)
Linear	one recursive call	factorial, sum, length of a list
Tree (multiple)	several recursive calls	Fibonacci, Towers of Hanoi, tree traversal
Tail recursion	the recursive call is the last operation	`gcd(b, a mod b)`

Tail recursion can be turned into a loop mechanically (some compilers do this automatically – Java and Go do not).

Efficiency: the Fibonacci example

Fibonacci numbers

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

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, …

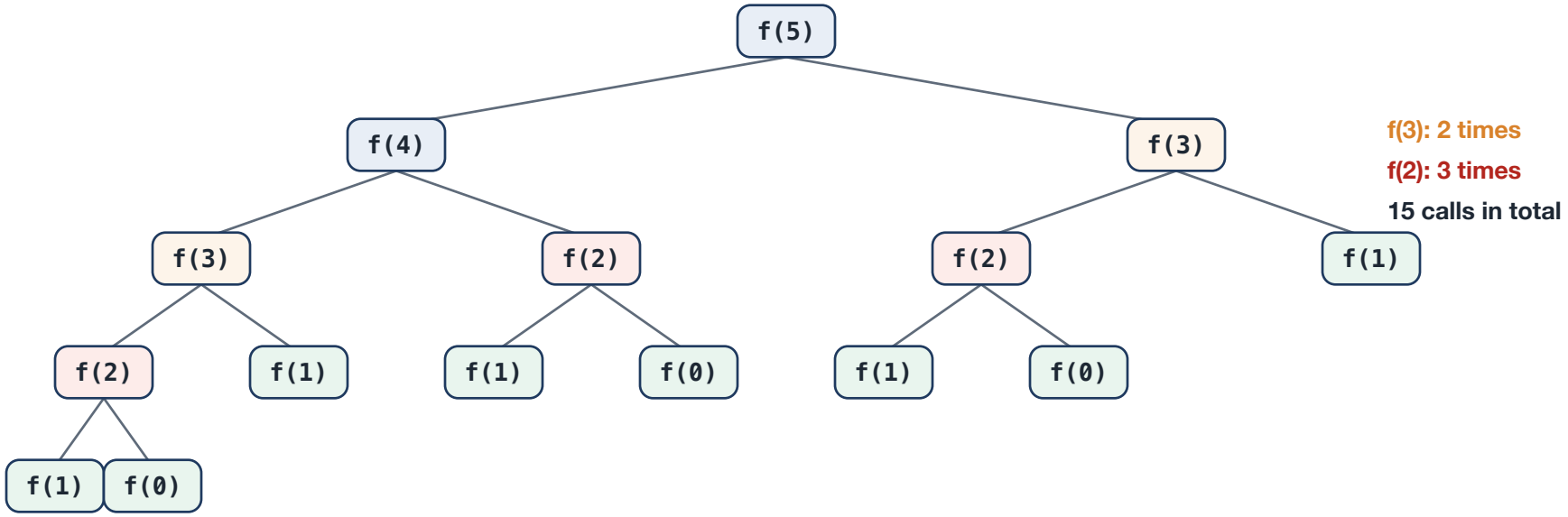
```
Function Fib(n)
  If n < 2 then
    Return n
  else
    Return Fib(n - 1) + Fib(n - 2)
  End if
End function
```

```
static long fib(int n) {
    if (n < 2) return n;
    return fib(n - 1) + fib(n - 2);
}
```

Simple and follows the definition exactly – but…

The call tree of the naive Fibonacci

Call tree of fib(5): the same values are recomputed again and again



f(3): 2 times

f(2): 3 times

15 calls in total

Measurements

n	fib(n)	naive calls	memoized	time (naive)
10	55	177	19	0.0 ms
20	6 765	21 891	39	0.1 ms
30	832 040	2 692 537	59	2.6 ms
40	102 334 155	**331 160 281**	79	**320 ms**

- The naive version is **exponential**: $O(\phi^n) \approx O(1.618^n)$ – minutes for n = 50, millions of years for n = 100
- **Memoization:** the values already computed are stored in a **hash table** → **O(n)**
- **Iteration:** with two variables, O(n) time, **O(1)** space

Improved versions

Memoization (Java)

```java
static HashMap<Integer, Long> memo = new HashMap<>();

static long fibMemo(int n) {
    if (n < 2) return n;
    Long cached = memo.get(n);
    if (cached != null) return cached;
    long r = fibMemo(n - 1) + fibMemo(n - 2);
    memo.put(n, r);
    return r;
}
```

Iteration (Go)

```go
func fibIter(n int) int {
    a, b := 0, 1
    for i := 0; i < n; i++ {
        a, b = b, a+b
    }
    return a
}
```

Memoization is the basic idea of **dynamic programming**: every subproblem is solved only once.

Classic examples

Recursive thinking: “first + rest”

Sum of an array: the first element + the sum of the rest

```
static int sum(int[] a, int from) {
    if (from == a.length) return 0;
    return a[from] + sum(a, from + 1);
}
```

Reversing a string: the rest reversed + the first character

```
static String reverse(String s) {
    if (s.isEmpty()) return s;
    return reverse(s.substring(1)) + s.charAt(0);
}
```

Palindrome: the first = the last, and the inside is a palindrome too

```
func isPalindrome(r []rune) bool {
    if len(r) < 2 {
        return true
    }
    return r[0] == r[len(r)-1] &&
        isPalindrome(r[1:len(r)-1])
}
```

Length of a linked list: 1 + the length of the rest

```
func length(p *Node) int {
    if p == nil {
        return 0
    }
    return 1 + length(p.Next)
}
```

Fast exponentiation – divide and conquer

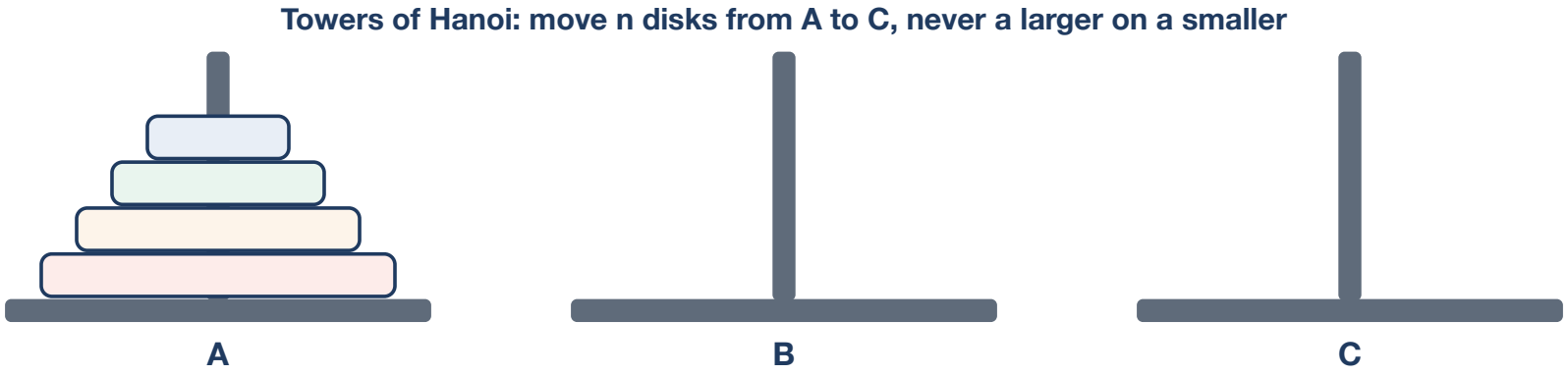
Computing x^{13} instead of 12 multiplications:

- $x^{13} = (x^6)^2 \cdot x \quad x^6 = (x^3)^2 \quad x^3 = (x^1)^2 \cdot x \quad \rightarrow$ **5** multiplications in total

```
Function Power(x, n)
  If n = 0 then Return 1
  half ← Power(x, n div 2)
  If n mod 2 = 0 then Return half · half
  else Return half · half · x
End function
```

- Every call **halves** n → **O(log n)** calls (like binary search)
- The same pattern: **recursive binary search**, merge sort, quicksort (week 13)

Towers of Hanoi




```
Procedure Hanoi(n, from, to, via)
  If n > 0 then
    Hanoi(n – 1, from, via, to)
    Move(from, to)
    Hanoi(n – 1, via, to, from)
  End if
End procedure
```

- Number of moves: **$2^n - 1$** → $O(2^n)$
- n = 3: 7 moves · n = 20: 1 048 575 moves
- n = 64: at one move per second about **585 billion years**

Much harder to write iteratively!

Backtracking: permutations

All orders of `A, B, C`: ABC, ACB, BAC, BCA, CAB, CBA

```
static void permute(List<Character> prefix, boolean[] used, char[] items) {
    if (prefix.size() == items.length) { print(prefix); return; }  // base case
    for (int i = 0; i < items.length; i++) {
        if (used[i]) continue;
        used[i] = true;  prefix.add(items[i]);          // decision
        permute(prefix, used, items);                   // the rest recursively
        prefix.remove(prefix.size() - 1);  used[i] = false;   // backtracking
    }
}
```

- n elements have **n!** permutations – O(n · n!)
- The same pattern: sudoku, maze, 8 queens, subsets

Recursion or iteration?

	Recursion	Iteration
Code	short, follows the definition	often longer
Space	O(depth) on the stack	usually O(1)
Speed	cost of the function calls	faster
Danger	stack overflow	–
Natural application	**trees**, divide and conquer, backtracking	simple loops, lists

Every recursion can be rewritten as iteration (with an own **stack** if necessary), and vice versa.
Rule of thumb: if the problem is **tree-shaped** (several recursive calls), recursion is much simpler; if it is linear, rather write a loop.

Summary

- **Recursion:** the function calls itself on a **smaller** problem
- Needed: a **base case** + the calls must **get closer** to the base case
- How it works: **call stack** – every call with its own frame, space O(depth)
- **Tree recursion** can easily become exponential (naive Fibonacci) → **memoization** or iteration
- Classic examples: factorial, exponentiation (O(log n)), Towers of Hanoi (2^n – 1), permutations
- The natural tool of **trees** (weeks 14–15) and of the fast sorting algorithms (week 13)

Next week: sorting II – merge sort and quicksort.

Review questions

1. What is the maximal depth of the call stack during the call `Fact(5)`?
2. What is wrong? `Function F(n): Return n + F(n – 1)`
3. How many times is the naive `Fib` function called while computing `Fib(5)`?
4. What does it print? `Procedure P(n): If n > 0 then P(n – 1); Print(n)` – `P(3)`
5. And if `Print(n)` is moved before `P(n – 1)`?
6. In how many moves can 5 disks be moved in the Towers of Hanoi?

Practice

✎ Demo programs

Program	Java	Go	What does it show?
Factorial	`Factorial.java`	`w12/factorial`	indented trace of the calls, overflow
Fibonacci	`Fibonacci.java`	`w12/fibonacci`	number of calls, memoization, time
Examples	`RecursionExamples.java`	`w12/examples`	sum, power, reverse, palindrome, binary search
Hanoi	`Hanoi.java`	`w12/hanoi`	printing the moves, $2^n - 1$
Stack depth	`StackDepth.java`	`w12/stackdepth`	`StackOverflowError` in Java, growing stack in Go
Permutations	`Permutations.java`	`w12/permutations`	backtracking

✎ Exercises

Solve them **with recursion** (without loops)!

1. **Digit sum:** `2026 → 10`
2. **Binary representation:** `50 → "110010"` (compare with the stack of week 7!)
3. **Maximum:** the largest element of an array
4. **Counting:** how many times does **x** occur in an array?
5. **List backwards:** print the elements of a linked list in reverse order!
6. **Length of a list** recursively
7. **Binomial coefficient:** C(n, k) = C(n – 1, k – 1) + C(n – 1, k) – why is it slow for large n?

Solutions: `Exercises12.java`, `w12/exercises`

✎ Homework

1. Write the binomial coefficient **with memoization**, and measure the speedup for C(30, 15)!
2. Write a recursive function that prints **all subsets** of a set! How many are there?
3. **Maze:** on a character map (`#` wall, `.` path) find a path from the top left corner to the bottom right one with backtracking!
4. Rewrite the Towers of Hanoi solution **iteratively** with an own stack! (Hint: push the “tasks still to be done” onto the stack.)