

# Lecture 10: Recursion Exercises

Guided example, laboratory activities, and exam-style independent practice

# Exercise structure

- **Guided example:** solved together to see the recursive pattern in action
- **Laboratory 1:** recursive traversal of a hierarchical organization
- **Laboratory 2:** recursive search and counting in nested packages
- **Independent exercises:** three compact tasks that together simulate about one hour of exam practice

# Guided: Count Files in Nested Folders

## Problem context

- A folder may contain files and other folders
- We assume the structure of the target folder saved in a variable `folder` of type `list`
- A **file** is represented by a **string**
- A **subfolder** is represented by a nested list
- The goal is to **count all files at any depth**.

```
folder = [  
    "report.pdf",  
    "notes.txt",  
    [  
        "photo.jpg",  
        ["archive.zip", "data.csv"]  
    ]  
]  
  
# expected result: 5
```

# Guided: Recursive Plan

## Implementation tasks

- Define `count_files(folder)`
- Initialize a counter for the current folder
- For each item: if it is a **file**, count 1
- If it is a **subfolder**, call `count_files(...)` again
- Return the total count.

## Recursive structure

- Base idea: an empty folder contains 0 files
- Recursive idea: a folder may contain smaller folders of the same kind
- Question to ask: what should happen when the current item is itself a folder?

To check the type of an object:

```
type(item) == list    #True or False  
type(item) == string  #True or False  
...
```

# Additional insight: Recursion vs. Iteration

Same problem: count files in nested folders at any depth.

## RECURSIVE · calls itself

```
def count_files(folder):
    total = 0
    for item in folder:
        if type(item) == list:
            total += count_files(item)
        else:
            total += 1
    return total
```

## ITERATIVE · uses an explicit stack

```
def count_files(folder):
    total = 0
    stack = [folder]
    while stack:
        current = stack.pop()
        for item in current:
            if type(item) == list:
                stack.append(item)
            else:
                total += 1
    return total
```

Both return 5 for the example folder. Both visit every file and subfolder once.

# Additional insight: Why Recursion, Then?

An explicit stack stores the folders we still need to explore.

## 1 Start

```
stack = [folder]  
Main folder is waiting.
```



## 2 Explore

```
current = stack.pop()  
Count files; add subfolders.
```



## 3 Repeat

```
while stack:  
Stop when nothing is left.
```

## Why might we choose recursion ?

The code mirrors the folder structure: a folder can contain smaller folders. It is often easier to read.

## Why might we choose iteration ?

It avoids deep recursive calls, but we must manage the stack ourselves.

**Key idea: neither is always faster. Both do the same work; the main difference is how it is organized.**

# Additional insight: Complexity

ADVANCED EXTENSION

A simple way to describe how work grows as the input gets bigger:

$O(n)$  · “linear growth” ·  $\propto n$

If there are twice as many elements to inspect, we expect roughly twice as much work.

$O(1)$  · “constant growth”

The amount of work stays about the same, even when the input becomes larger.

## For our folder-counting problem

Both algorithms must **inspect every file and subfolder**. If  $n$  is the total number of elements, both take  $O(n)$  time.

Big O describes how an algorithm scales, not its exact execution time in seconds.

# Additional insight: Time and Memory

ADVANCED EXTENSION

## Same work, different memory requirements

### RECURSIVE

**Time:  $O(n)$**

Every file and subfolder is visited once.

**Memory:  $O(d)$**

Python remembers only the active function calls.  
More nested levels mean more memory.

### ITERATIVE

**Time:  $O(n)$**

Every file and subfolder is visited once.

**Memory:  $O(n)$  worst case**

The stack stores folders waiting to be explored, also from different levels. More waiting folders mean more memory. Its size therefore depends on both depth and branching.

**$n$  = total elements**

**$d$  = maximum nesting depth**

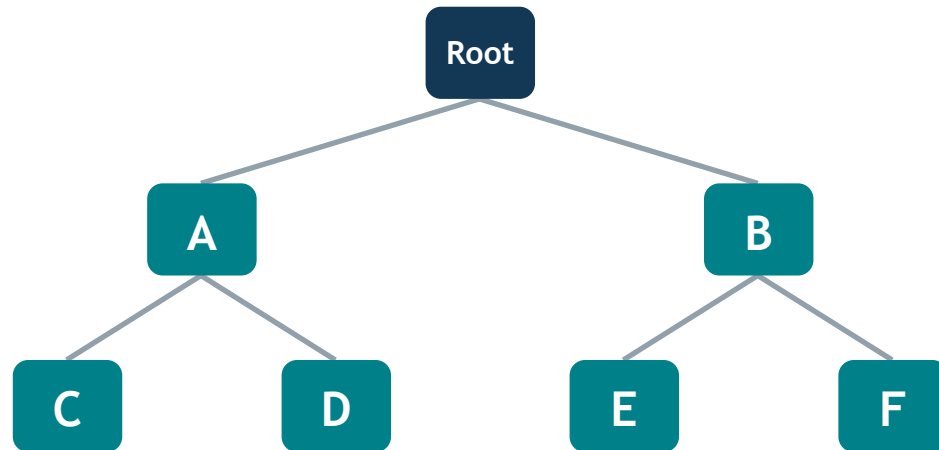
**$w$  = most folders waiting in the stack**

Neither uses less memory in every situation;  $O(w)$  can be as large as  $O(n)$ .

# Two Ways to Explore a Hierarchy: DFS vs. BFS

ADVANCED EXTENSION

The difference is the order in which we visit the elements



## DFS • Depth-First Search

Explore one branch fully before the next.

Root → A → C → D → B → E → F

## BFS • Breadth-First Search

Explore all nodes at one level first.

Root → A → B → C → D → E → F

## Complexity (for $n$ nodes)

Time:  $O(n)$  for both (each node is visited once)

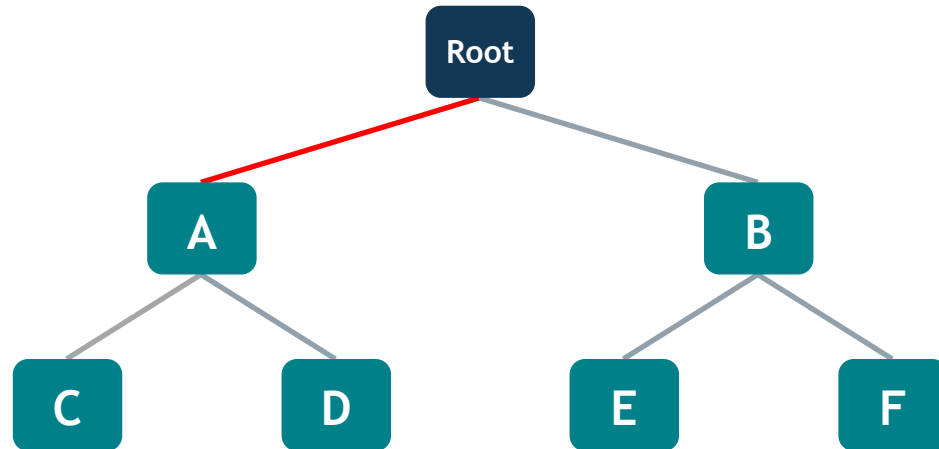
Extra memory: DFS  $O(h)$  (tree height); BFS  $O(w)$  (maximum level width).

DFS is naturally recursive; BFS is usually iterative with a queue.

# Two Ways to Explore a Hierarchy: DFS vs. BFS

ADVANCED EXTENSION

The difference is the order in which we visit the elements



## DFS • Depth-First Search

Root → A

## BFS • Breadth-First Search

Explore all nodes at one level first.

Root → A → B → C → D → E → F

## Complexity (for $n$ nodes)

Time:  $O(n)$  for both (each node is visited once)

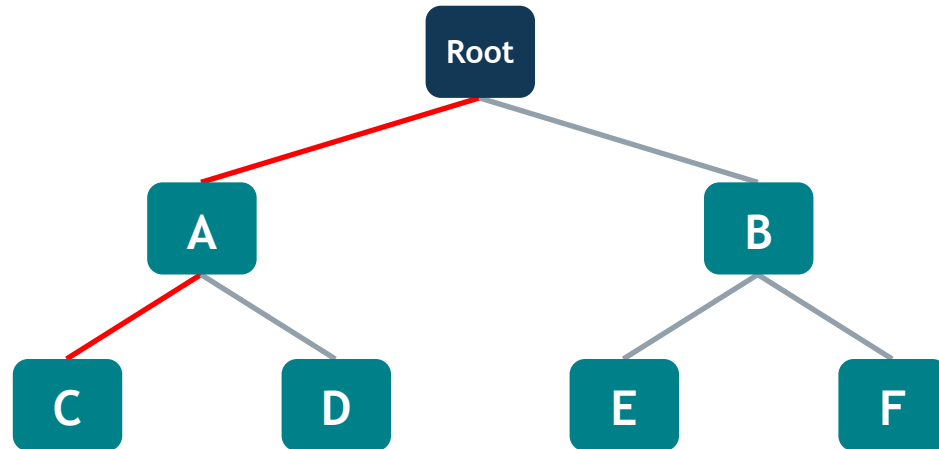
Extra memory: DFS  $O(h)$  (tree height); BFS  $O(w)$  (maximum level width).

DFS is naturally recursive; BFS is usually iterative with a queue.

# Two Ways to Explore a Hierarchy: DFS vs. BFS

ADVANCED EXTENSION

The difference is the order in which we visit the elements



## DFS • Depth-First Search

Root → A → C

## BFS • Breadth-First Search

Explore all nodes at one level first.

Root → A → B → C → D → E → F

## Complexity (for $n$ nodes)

Time:  $O(n)$  for both (each node is visited once)

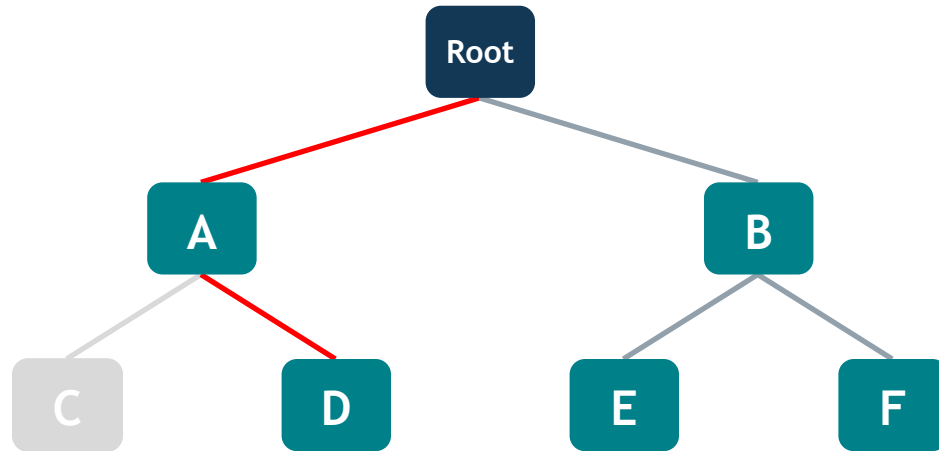
Extra memory: DFS  $O(h)$  (tree height); BFS  $O(w)$  (maximum level width).

DFS is naturally recursive; BFS is usually iterative with a queue.

# Two Ways to Explore a Hierarchy: DFS vs. BFS

ADVANCED EXTENSION

The difference is the order in which we visit the elements



## DFS · Depth-First Search

Root → A → ~~B~~ → D

## BFS · Breadth-First Search

Explore all nodes at one level first.

Root → A → B → C → D → E → F

## Complexity (for $n$ nodes)

Time:  $O(n)$  for both (each node is visited once)

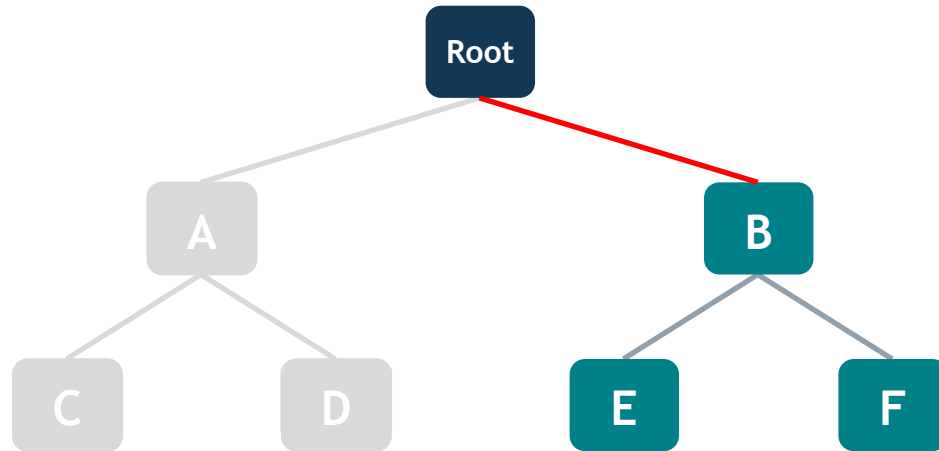
Extra memory: DFS  $O(h)$  (tree height); BFS  $O(w)$  (maximum level width).

DFS is naturally recursive; BFS is usually iterative with a queue.

# Two Ways to Explore a Hierarchy: DFS vs. BFS

ADVANCED EXTENSION

The difference is the order in which we visit the elements



## DFS · Depth-First Search

Root → ~~A~~ → ~~C~~ → ~~D~~ → B

## BFS · Breadth-First Search

Explore all nodes at one level first.

Root → A → B → C → D → E → F

## Complexity (for n nodes)

Time:  $O(n)$  for both (each node is visited once)

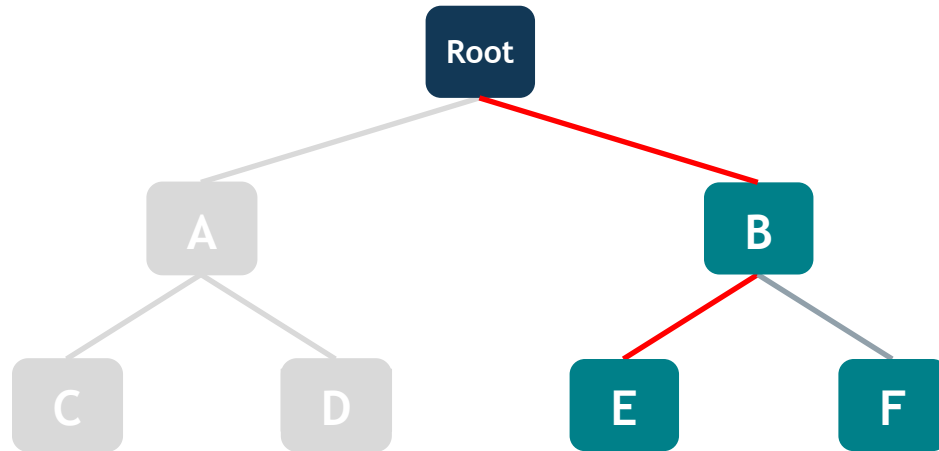
Extra memory: DFS  $O(h)$  (tree height); BFS  $O(w)$  (maximum level width).

DFS is naturally recursive; BFS is usually iterative with a queue.

# Two Ways to Explore a Hierarchy: DFS vs. BFS

ADVANCED EXTENSION

The difference is the order in which we visit the elements



## DFS · Depth-First Search

Root → ~~A~~ → ~~C~~ → ~~D~~ → B → E

## BFS · Breadth-First Search

Explore all nodes at one level first.

Root → A → B → C → D → E → F

## Complexity (for n nodes)

Time:  $O(n)$  for both (each node is visited once)

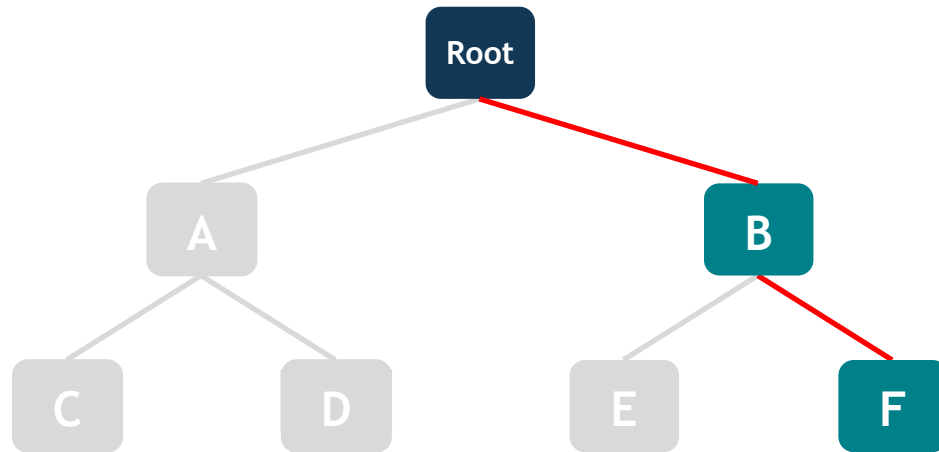
Extra memory: DFS  $O(h)$  (tree height); BFS  $O(w)$  (maximum level width).

DFS is naturally recursive; BFS is usually iterative with a queue.

# Two Ways to Explore a Hierarchy: DFS vs. BFS

ADVANCED EXTENSION

The difference is the order in which we visit the elements



## DFS · Depth-First Search

Root → ~~A~~ → ~~C~~ → ~~D~~ → B → ~~E~~ → F

## BFS · Breadth-First Search

Explore all nodes at one level first.

Root → A → B → C → D → E → F

## Complexity (for n nodes)

Time:  $O(n)$  for both (each node is visited once)

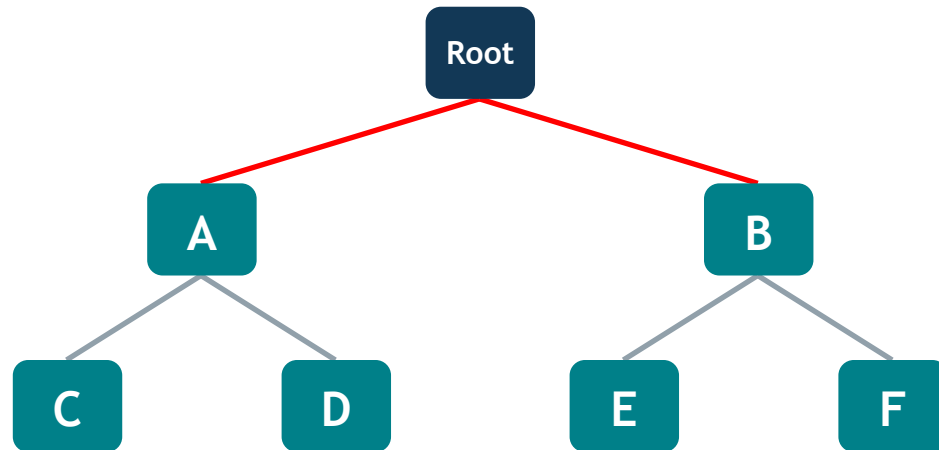
Extra memory: DFS  $O(h)$  (tree height); BFS  $O(w)$  (maximum level width).

DFS is naturally recursive; BFS is usually iterative with a queue.

# Two Ways to Explore a Hierarchy: DFS vs. BFS

ADVANCED EXTENSION

The difference is the order in which we visit the elements



## DFS • Depth-First Search

Explore one branch fully before the next.

Root → A → C → D → B → E → F

## BFS • Breadth-First Search

Root → A → B

## Complexity (for n nodes)

Time:  $O(n)$  for both (each node is visited once)

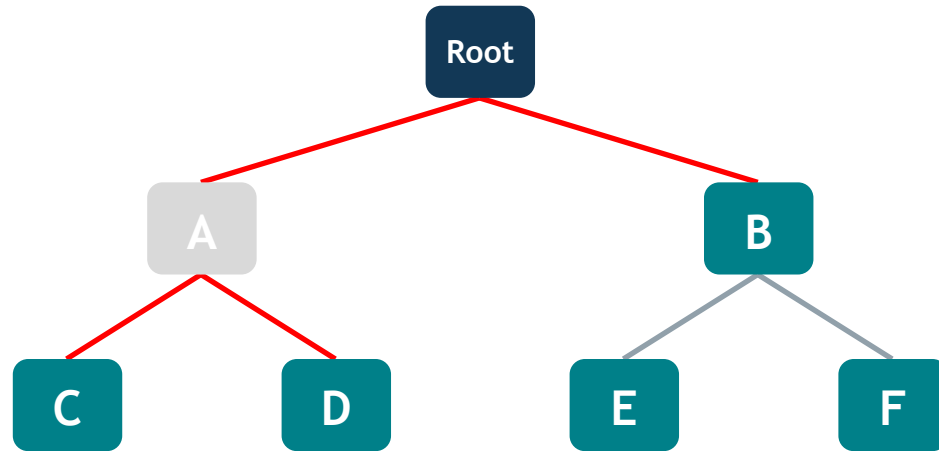
Extra memory: DFS  $O(h)$  (tree height); BFS  $O(w)$  (maximum level width).

DFS is naturally recursive; BFS is usually iterative with a queue.

# Two Ways to Explore a Hierarchy: DFS vs. BFS

ADVANCED EXTENSION

The difference is the order in which we visit the elements



## DFS · Depth-First Search

Explore one branch fully before the next.

Root → A → C → D → B → E → F

## BFS · Breadth-First Search

Root → ~~A~~ → B → C → D

## Complexity (for n nodes)

Time:  $O(n)$  for both (each node is visited once)

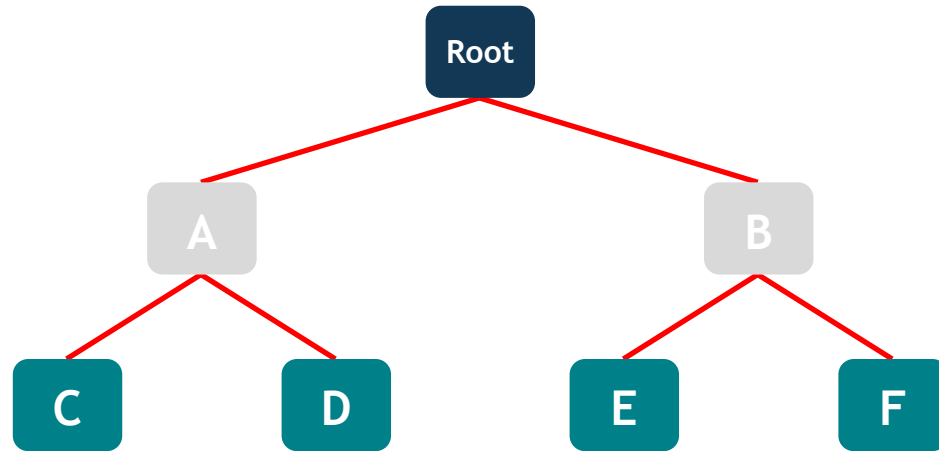
Extra memory: DFS  $O(h)$  (tree height); BFS  $O(w)$  (maximum level width).

DFS is naturally recursive; BFS is usually iterative with a queue.

# Two Ways to Explore a Hierarchy: DFS vs. BFS

ADVANCED EXTENSION

The difference is the order in which we visit the elements



## DFS · Depth-First Search

Explore one branch fully before the next.

Root → A → C → D → B → E → F

## BFS · Breadth-First Search

Root → ~~A~~ → ~~B~~ → C → D → E → F

## Complexity (for n nodes)

Time:  $O(n)$  for both (each node is visited once)

Extra memory: DFS  $O(h)$  (tree height); BFS  $O(w)$  (maximum level width).

DFS is naturally recursive; BFS is usually iterative with a queue.

# Laboratory 1: Company Team Hierarchy

## Problem context

- A company stores its organization chart as nested dictionaries
- Each employee has a name and a team
- The team is a list of employees with the same structure
- The program must analyze and print the whole hierarchy.

```
company = {  
    "name": "Director",  
    "team": [  
        {"name": "Sales Manager",  
         "team": [  
             {"name": "Alice", "team": []},  
             {"name": "Bob", "team": []}  
         ]},  
        {"name": "IT Manager",  
         "team": [{"name": "Carol",  
                   "team": []}]  
        }  
    ]  
}
```

# Laboratory 1: Tasks and hints

Given the provided structure of an organization chart:

## 1. Count all employees

- Implement `count_people(employee)`
- The function must return the total number of people in the hierarchy starting from `employee`
- The employee passed to the function must also be included in the count

## 2. Print the organization hierarchy

- Implement `print_team(employee, level=0)`
- Print the name of every employee
- Use indentation to show the hierarchy: direct team members are indented one level more than their manager

## 3. Test your functions

- Apply both functions to the provided organization chart
- Verify that the total number of employees is correct and that the printed hierarchy reflects the organization structure
- Test both functions again on an organization containing only one employee.

```
Director
  Sales Manager
    Alice
    Bob
  IT Manager
    Carol
```

# Laboratory 2: Nested Shipment Manifest

## Problem context

- A logistics center represents a shipment as a nested list
- A string is a real item in the shipment
- A list is another container inside the shipment
- The system must count items and search inside all nested containers.

```
shipment = [  
    "Laptop",  
    ["Mouse", "Keyboard"],  
    [  
        "Monitor",  
        ["Cable", "Adapter"]  
    ]  
]
```

```
# 6 items in total
```

# Laboratory 2: Tasks and hints

Given the provided nested shipment structure:

## 1. Count all items in the shipment

- Implement `count_items(package)`
- Return the **total** number of actual items contained in package, including items inside any nested packages
- Packages themselves must not be counted as items

## 2. Search for an item

- Implement `contains_item(package, target)`
- Ask the user for the name of an item to search for
- Search through the entire shipment, including all nested packages and print the result
- Return `True` if target is found at any level and `False` otherwise

## 3. Test your functions

- Use `count_items()` to print the total number of items in the provided shipment
- Use `contains_item()` to search for at least:
  - one item that is present inside a nested package;
  - one item that is not present in the shipment.
- Test both functions also on a package containing no nested packages.

# Independent practice

Three compact exercises; together they simulate about one hour of exam practice

# Individual 1: Sensor Measurements Analyzer

## Problem context

- A monitoring system receives a list of numeric sensor measurements (list of floats)
- The user provides a threshold
- The program must analyze the list **recursively**, without using loops inside the requested recursive functions.

## Required functions

- `count_above(values, threshold)`: count how many measurements are greater than the threshold
- `maximum(values)`: return the largest value in a non-empty list
- The original list must not be modified
- Hint: reason about the first measurement and the remaining measurements.

# Individual 2: Package Manifest

## Problem context

- A shipment may contain objects or other packages
- Objects are strings; packages are lists
- The nesting depth is not known in advance
- The program must work regardless of how deeply items are nested.

## Required functions

- `count_items(package)`: count all objects contained in the package.
- `contains_item(package, target)`: check whether a target object appears anywhere
- Return Boolean values for search results
- Hint: an element is either directly processed or recursively explored.

# Individual 3: Organization Depth Report

## Problem context

- An organization is represented as nested employee records
- Each employee record contains a name and a team
- The program must determine how deep the hierarchy is and whether a name appears anywhere.

## Required functions

- `hierarchy_depth(employee)`: return the maximum depth of the hierarchy.
- `find_employee(employee, name)`: return True if the name is found in any level.
- An employee with no team has depth 1.
- Hint: compute the depth of each subordinate, then keep the largest one.

# Checklist for recursive solutions

- Base case: what is the simplest input that can be solved immediately ?
- Recursive case: how is the current problem reduced to a smaller one ?
- Returned value: what exactly should each call return to its caller ?
- Combination step: do we return the recursive result directly or use it in an operation ?
- Test small cases first: empty structures, one element, and one nested level.