Fundamentos de Programação 2025-2026

Programming Fundamentals

Class #10 - Recursion

Overview

- An alternative way to solve problems
- Stacks, frames and program stack
- Recursive functions (in Python)
- Application Examples
- Iterative vs Recursive functions
- Common mistakes

- Navigating in a maze

ALTERNATIVE WAYS OF SOLVING (SOME) PROBLEMS

Problem: Finding the minimum in a List

- Iterative Solution
  - **Idea:** Keep track of the smallest value seen so far.
  - Pseudo Code

```
function findMinIterative(list):
    minValue ← list[0]
    for each element in list starting from index 1:
        if element < minValue:
            minValue ← element
    return minValue
```

Problem: Finding the minimum in a List

- We can think of another way to determine the minimum:
  - Compare the **first element** with the **minimum of the remaining elements**.
  - But how do we determine the minimum of the remaining elements?
  - By applying the **same idea again**: compare its first element with the minimum of its remaining elements.
  - And so on… until we reach a list with only one element, which is trivially its own minimum.
- This is a **Recursive Solution**, that breaks the problem into smaller sublists until only one element remains.

```
function findMinRecursive(list):
    if length(list) = 1:
        return list[0]
    else:
        subMin ← findMinRecursive(list[1:])
        if list[0] < subMin:
            return list[0]
        else:
            return subMin
```

Problem: Factorial(n)

Iterative

Idea: Multiply numbers from 1 up to n step by step

```
function factorialIter(n):
    result ← 1
    for i from 2 to n:
        result ← result * i
    return result
```

Recursive

Idea: Factorial of n is n times factorial of (n-1)

```
function factorialRec(n):
  if n == 0 or n == 1:
     return 1 // base case
  else:
    return n * factorialRec(n-1)
```

Problem: Paint all the divisions of a house

- **Iterative**

  Idea: Go through each division one by one and paint it

```
function paintHouseIter(divisions):
    for each division in divisions:
        paint(division)
```

- **Recursive**

  Idea: Paint the first division, then paint the rest

```
function paintHouseRecursive(divisions):
    if divisions is empty:
        return      # base case
    else:
        paint(divisions[0])
        paintHouseRecursive(divisions[1:])  # the rest
```

This alternative approach is interesting

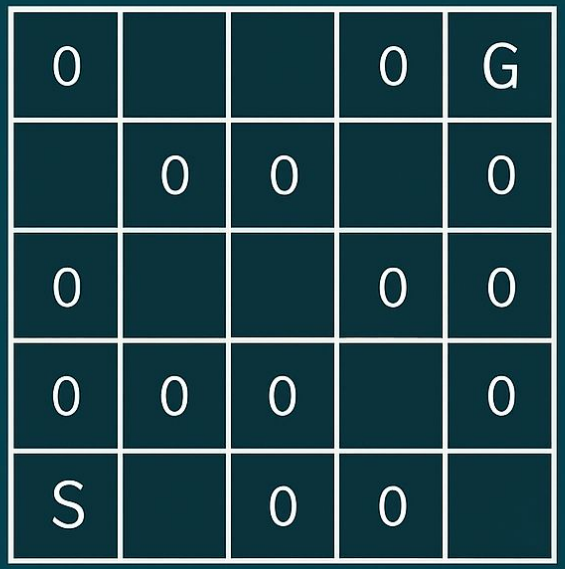
- Step-by-step reduction:
  - Each call works on a smaller problem
- Base case clarity:
  - The simplest case anchors the recursion
- Self-similarity:
  - The same comparison pattern repeats at every level

A New Problem: Navigating a Maze

- **Scenario:**
  - You are given a 2D grid representing a maze. Each cell can be:
    - Open (0)
    - Blocked/Wall (1)
    - The goal (G)
    - The start (S)
  - You start at a given cell and must find a path to the goal, moving only up, down, left, or right.

Problem

Maze Navigation Problem



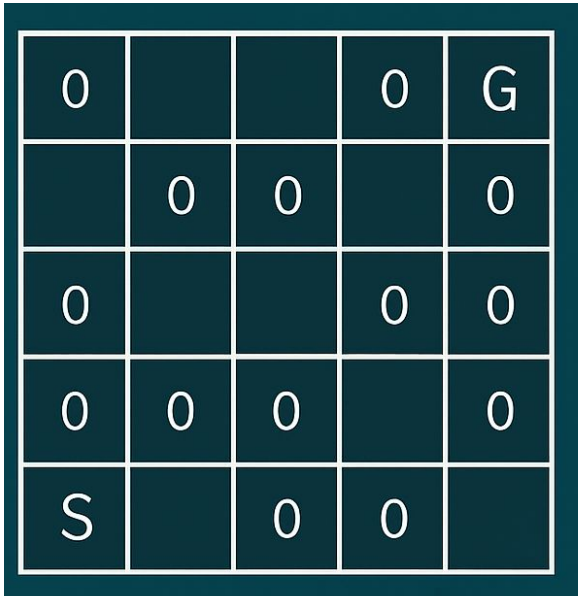

```
find_path(maze, x, y):
 if out_of_bounds(maze, x,y
    or maze[x][y] == 1:
       return False
 if maze[x][y] = 'G':
    return True
 mark_as_visited(maze, x, y)
return ( find_path(maze, x+1, y)
    or  find_path(maze, x-1, y)
        find_path(maze, x-1, y)
)       find_path(maze, x,y- 1)
```

Possible solution (pseudo code)

```
function findPath(maze, x, y):
    if x or y is out of bounds:
        return false

    if maze[x][y] is a wall or already visited:
        return false

    if maze[x][y] is the goal:
        return true

    mark maze[x][y] as visited

    if findPath(maze, x + 1, y): return true   // move down
    if findPath(maze, x - 1, y): return true   // move up
    if findPath(maze, x, y + 1): return true   // move right
    if findPath(maze, x, y - 1): return true   // move left

    unmark maze[x][y] (optional, for backtracking visualization)
    return false
```

Call Stack Trace (Simplified)

- Assume the path from (4,0) to (0,4) is valid and goes through open cells.
- Here's how the stack builds and unwinds:




```
Call 1: findPath(4, 0)   ← Start
Call 2: findPath(3, 0)   ← Move up
Call 3: findPath(2, 0)   ← Move up
Call 4: findPath(1, 0)   ← Move up
Call 5: findPath(0, 0)   ← Move up
Call 6: findPath(0, 1)   ← Move right → blocked
Call 7: findPath(0, 3)   ← Skip to open path
Call 8: findPath(0, 4)   ← Goal reached ✅
```

Why Recursion to solve our problem

- The problem naturally breaks into subproblems:
  - “From this cell, can I reach the goal?”
- Recursive calls explore each direction, backtracking when a path is blocked.
- The base cases are: reaching the goal, hitting a wall or stepping out of bounds.
- Recursive cases explore neighbors.

What do we need?

- To be possible to call a function inside itself

- Guarantee that the process ends

Before we start our topic …

SOME USEFUL INFORMATION

Stack

- A stack is a **linear data structure** that stores elements in a specific order.
- It follows the principle LIFO (**Last In, First Out**).

- Analogy: **A stack of plates / books**:
  - You add on top.
  - You remove from the top.
  - You cannot directly access in the middle without removing those above.

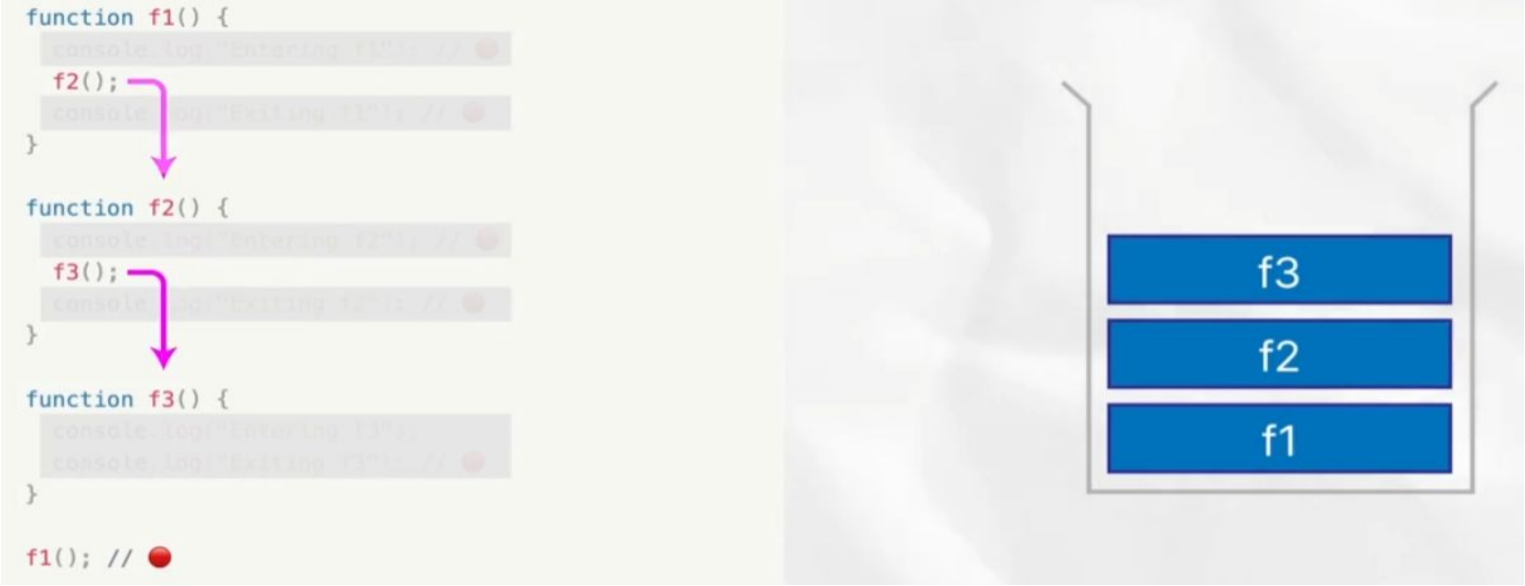
- Key Operations:
  - **Push**: Add an element to the top.
  - **Pop**: Remove the element from the top.

Frames

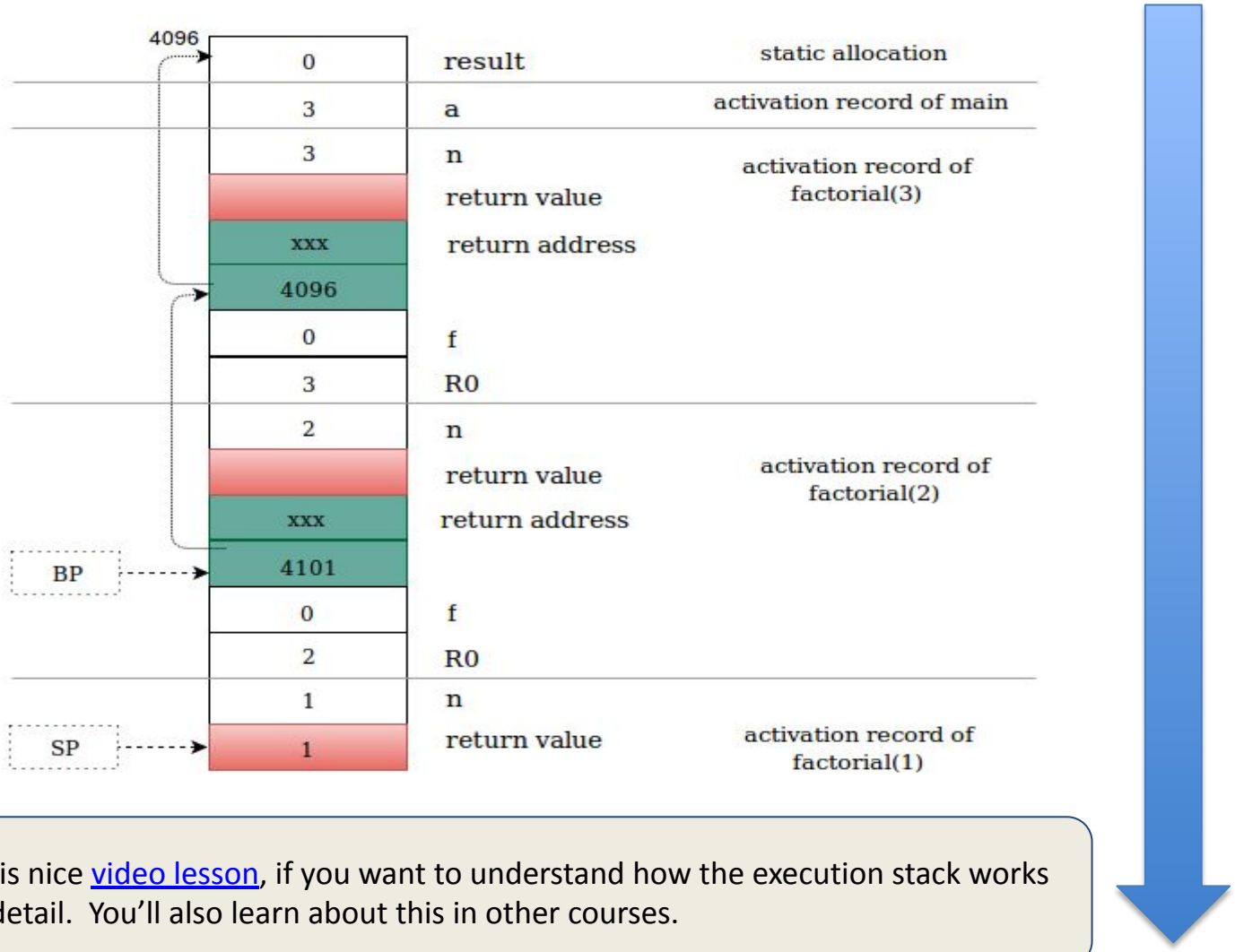
- A frame (or activation record) is a block of memory created when a function is called.
- It stores all the information needed to execute and return from that function.
- Each frame stores the local context of a single function call.
- Frames are managed in a call stack.
  - Also named the *program stack* or *execution stack*.

Execution Stack

- Each new function call is placed on top of the stack.
- When a **function is called**, its local variables and return address are pushed onto the stack.
- When a **function finishes**, its stack frame is popped off, and control returns to the function just below it.



Execution Stack (in memory)

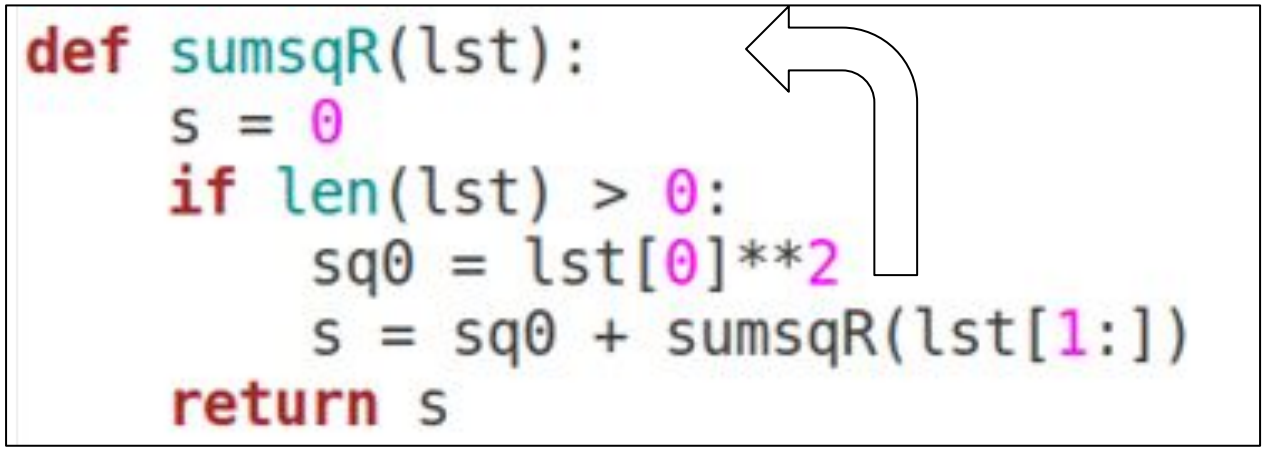


Watch this nice video lesson, if you want to understand how the execution stack works in more detail. You'll also learn about this in other courses.

RECURSION

Recursive functions

- A **recursive function** is a function that calls itself.



```
def sumsqR(lst):
    s = 0
    if len(lst) > 0:
        sq0 = lst[0]**2
        s = sq0 + sumsqR(lst[1:])
    return s
```

- Notice that there is no loop instruction, but code gets executed several times, anyway.
- How does it work?

How recursion works

- What happens when we call `sumsqR([1, 2, 3])` ?

```
sumsqR([1, 2, 3])
|    sumsqR([2, 3])
|    |    sumsqR([3])
|    |    |    sumsqR([])
|    |    |    └>0
|    |    └>9    ( = 3**2 + 0 )
|    └>13    ( = 2**2 + 9 )
└>14    ( = 1**2 + 13 )
```

Play ▶

- Notice that at one point, there are 4 frames in memory.
  - 4 variables named lst, 4 named s, 3 named sq0, but all distinct!
- Each frame stores the local context of a single function call.

The rules for recursion termination

To guarantee that a recursive function **terminates**, it must respect some **rules**:

1. There must be cases that can be solved without recursive calls. These are called the **base cases**.
   - In `sumsqR`, the base case is `len(lst)==0`, that returns 0.
2. In the **other cases**, the context passed to recursive calls **must always differ** from the context received.*
   - In `sumsqR`, the argument `lst[1:] != lst` (always)
3. In successive recursive calls, the context **must converge towards the base cases**.
   - In `sumsqR`, the `lst` is shortened on each call, until it's empty.

* The ***context*** is the set of arguments (and global values) that have an impact on the base case / recursive case selection.

Why Use Recursive Functions?

- **Elegant Solutions** to Complex Problems
  - Recursion simplifies problems that have a natural hierarchical or self-similar structure
    - e.g., trees, graphs, fractals.
  - It allows for concise and readable code, especially when iterative solutions are cumbersome.

- **Divide-and-Conquer** Strategy
  - Recursion is ideal for breaking down problems into smaller subproblems
    - e.g., merge sort, quicksort.
  - Each recursive call tackles a simpler version of the original problem.

Why Use Recursive Functions?

- **Natural Fit for Mathematical Definitions**
  - Many algorithms are defined recursively
    - e.g., factorial, Fibonacci, GCD
  - Recursion mirrors mathematical induction,
    - making it intuitive for mathematically inclined learners.

- **Foundation** for Advanced Concepts
  - Enables understanding of backtracking, dynamic programming, and functional programming paradigms.
  - Builds mental models for stack frames, base cases, and termination conditions.

Recursion vs repetition

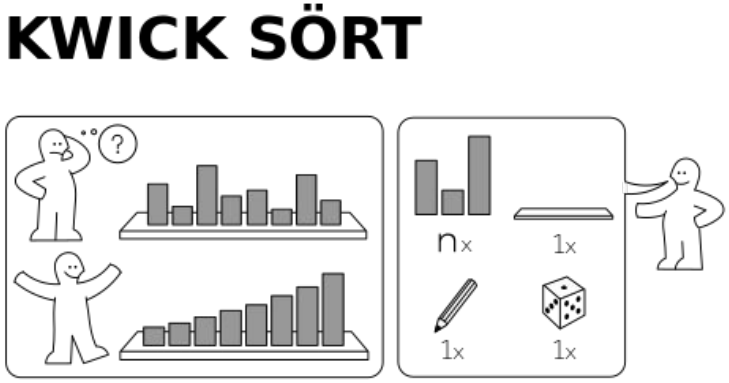
- Any problem that can be solved by repetition may be solved by recursion, and vice-versa.
- For certain complex problems, recursive solutions are usually more concise and easier to understand.
- Recursive implementations may incur extra time and memory cost because of function calls and stack usage.
- If the problem has a simple iterative solution, that is usually the most efficient, too.

Recursive vs Iterative: Performance & Practicality

- Memory Usage
  - Recursive functions use the call stack, which can lead to stack overflow for large inputs.
  - Iterative solutions are sometimes available with constant memory, making them more efficient.
- Speed
  - Recursion introduces overhead from repeated function calls.
  - Iteration is generally faster due to direct control flow.
- Scalability
  - Recursive solutions are less suitable for large datasets unless tail-call optimization is available (not in Python).
  - Iterative solutions scale well and handle large inputs reliably.

APPLICATION EXAMPLES

Example: quicksort



Example: quicksort

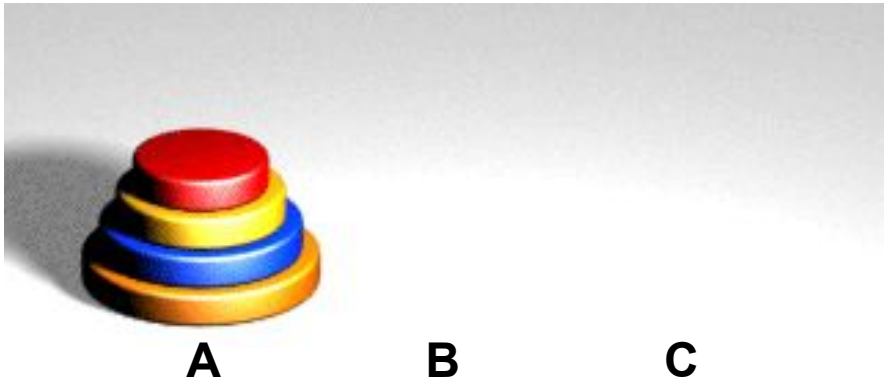
- The quicksort algorithm (by C.A.R. Hoare) goes like this:
  1. Pick one of the values in the list (e.g. the first) and store in T.
  2. Put values smaller than T into a list L1, the others into a list L2.
  3. Sort L1 and L2 (using same algorithm, by the way)
  4. Result is L1 + [T] + L2.

```
def qsorted(lst):
    if len(lst)<=1:     # no need to sort
        return lst[:]   # just return a copy
    T = lst[0]
    L1 = [x for x in lst[1:] if x<T]
    L2 = [x for x in lst[1:] if x>=T]
    return qsorted(L1) + [T] + qsorted(L2)
```

- This is simple to understand and quite efficient!
- NOTE: The actual quicksort modifies the list *in-place* (without creating extra lists), and is a bit harder to write.

Example: Towers of Hanoi

- The Towers of Hanoi puzzle
  - By Édouard Lucas, 1883.
- Move tower from A to C, using B temporarily.
  - Move only one disk at a time;
  - No disk may be put on top of a smaller disk.



André Karwath aka Aka, CC BY-SA 2.5, via Wikimedia Commons

- Now solve it in 4 lines of code!

Play ▶

Writing recursive functions

To develop a recursive function to solve some problem, follow these guidelines:

1. **Define** the **arguments** you need, what they **mean**, and the **expected result**, as rigorously as possible.

2. **Assume** the function will work. Describe how the solution to a problem can be obtained by **modifying** the solutions to **smaller** versions of the problem.
   - This will be the recursive part of the algorithm.

3. **Determine** the base cases: which conditions have a trivial solution?
   - This will be the non-recursive part of the algorithm.
   - Hint: base cases are usually outside the domain of the recursive part.

- While in step 2, you may realize that you need extra arguments. In that case, go back to step 1 and add the arguments you need.

SOME COMMON MISTAKES BEGINNERS MAKE WITH RECURSION IN PYTHON

And How to Identify & Fix Them

Forgetting the Base Case

- **Mistake:** Writing recursive functions without a clear base case.
- **Consequence:** Infinite recursion → program crashes (stack overflow).
- **Example:**

```
def factorial(n):
    return n * factorial(n-1)
    # no base case!
```

- **Tip:** Always ask: *"When does the recursion stop?"*

Incorrect or Weak Base Case

- **Mistake:** Base case exists but doesn't cover all stopping conditions.
- **Consequence:** Wrong results or partial termination.
- **Example:**

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

- **Tip:** Test base cases with smallest inputs (0, 1, empty list).

Missing Progress Toward Base Case

- **Mistake:** Recursive call doesn't move closer to termination.
- **Consequence:** Infinite recursion again.
- **Example:**

```
def countdown(n):
    if n == 0:
        return
    countdown(n)    # no change in n
```

- **Tip:** Ensure each recursive call reduces the problem size.

Confusing Iteration with Recursion

- **Mistake:** Writing recursive code that mimics loops without need.
- **Consequence:** More complex, less efficient than iteration.
- **Example:**

```
def printList(lst, i=0):
    if i == len(lst):
        return
    print(lst[i])
    printList(lst, i+1)
```

- **Tip:** Ask: *"Is recursion really needed here?"*

Ignoring Stack Depth Limits

- **Mistake:** Assuming recursion works for very large inputs.
- **Consequence:** Stack overflow error.

- **Example:**

  factorial(10000) in Python.

Overlapping Work / Inefficiency

- **Mistake:** Recomputing the same subproblems repeatedly.
- **Consequence:** Exponential runtime
  - e.g., Fibonacci.
- **Example:**

```
def fib(n):
    if n <= 1:
        return n
    return fib(n-1) + fib(n-2)
           # repeats work!
```

- **Tip:** Use *memoization* or iterative alternatives.

Recursive Fibonacci with Memoization

```
fib_memo = {}  # cache
def fib(n):
     # Check cache
    if n in fib_memo:
        print(f"fib({n}) already calculated = {fib_memo[n]}")
        return fib_memo[n]
    # Base cases
    if n == 0:
        return 0
    elif n == 1:
        return 1
    # Recursive case
    result = fib(n - 1) + fib(n - 2)
    fib_memo[n] = result
    return result
```

```
# Example usage
for i in range(10):
    print(f"fib({i}) = {fib(i)}")
```

Play ▶

You'll learn about memoization and other techniques in another course!

RETURNING TO OUR INITIAL PROBLEM …

Possible Solution in Python

- Code: examples/demo_maze_solution.py