



**METROPOLITAN  
TIRANA  
UNIVERSITY**

**Course: Data Structures and Algorithms**

# Recursion

Evis Plaku

# Welcome to recursion ... again and again

To understand  
recursion,  
you must first  
understand recursion



# Introduction to recursion

# Real-world examples & motivation

Recursion simplifies complex problems by breaking them into smaller, manageable subproblems

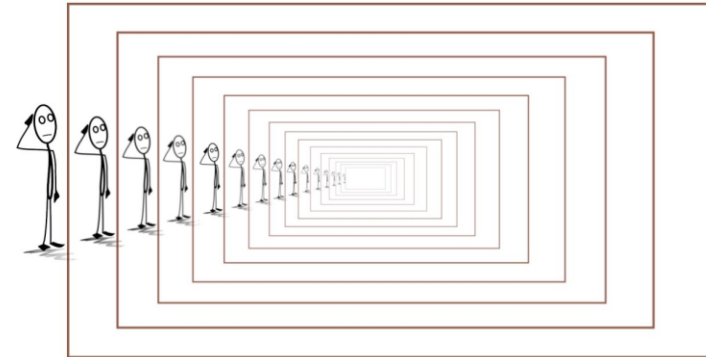
- File system navigation: operating systems use recursion to list directories and files
- Web crawlers & ai decision trees: search engines and AI models rely on recursive exploration
- Game development & pathfinding: recursive algorithms solve mazes, generate maps, and plan moves



# How recursion works: a function calling itself

- Recursion occurs when a function calls itself to solve a smaller version of the same problem
- **Self-referencing functions:** a function repeatedly calls itself with modified input
- **Call stack & execution flow:** each function call is stored in memory until resolved

Recursion is used in divide and conquer, dynamic programming, backtracking



# Key components: base case and recursive step

Two essential components: the **base case** to stop the recursion and the **recursive case** to continue it

- Condition that stops the recursion. Without it, the function would keep calling itself forever
  - In base case, recursion ends and starts returning results
- } **Base case**
- 
- The function calls itself, usually with a smaller or simpler input, moving towards the base case
  - The recursive case breaks the problem into smaller steps
- } **Recursive step**

# Key components: base case and recursive step



**Goal:** break down problem in smaller manageable steps

## Intuition

- The **base case** is the first step (the simplest solution)
- The recursive case is moving one step down, gradually reaching the base case



## Linking to mathematical induction

- **Base case:** verify the smallest case ( $n = 1$ )
- **Inductive hypothesis:** assume the problem works for a smaller case ( $n = k$ )
- **Inductive step:** prove it works for the next step ( $n = k + 1$ )




# Basic recursion examples



# Factorial calculation

- **Base case:** stops recursion when  $n == 1$
- **Recursive case:** calls  $factorial(n - 1)$  until base case is reached
- **Stack unwinding:** returns the multiplied results as recursion unwinds

$$n! = n \cdot (n - 1) \cdot (n - 2) \cdot \dots \cdot 2 \cdot 1$$

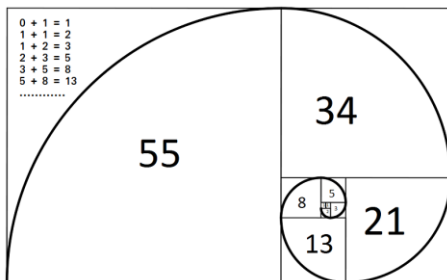


```
public int factorial(int n) {  
    if (n == 1) // Base Case  
        return 1;  
    else // Recursive Case  
        return n * factorial(n - 1);  
}
```

$$n! = \begin{cases} n & n == 1 \\ n \cdot (n - 1)!, & n > 1 \end{cases}$$

# Fibonacci calculation

- Fibonacci sequence follows a recursive pattern where each term is the sum of the two previous terms
- Base case: stops when  $n == 0$  or  $n == 1$
- Recursive step: calls  $Fibonacci(n - 1) + Fibonacci(n - 2)$



1, 1, 2, 3, 5, 8, 13, 21, 24, ...



```
public int fibonacci(int n) {  
    if (n == 0 || n == 1) // Base Case  
        return n;  
    else // Recursive Case  
        return fibonacci(n - 1) + fibonacci(n - 2);  
}
```

$$f(n) = \begin{cases} 1 & n \leq 1 \\ f(n - 1) + f(n - 2) & n > 1 \end{cases}$$

# Sum of first N numbers

- The sum of the first  $N$  numbers can be computed recursively by adding  $N$  to the sum of first  $N - 1$  numbers
- **Base case:** stops when  $n == 1$ , return 1
- **Recursive step:** calls  $n + \text{sum}(n - 1)$  until reaching the base case

$$1 + 2 + 3 + \dots + (N - 1) + N$$



```
public int sum(int n) {  
    if (n == 1) // Base Case  
        return 1;  
    else // Recursive Case  
        return n + sum(n - 1);  
}
```

# Common pitfalls in recursion

- Infinite recursion: occurs when the base case is missing or incorrect
- Stack overflow: too many recursive calls can exceed memory limits
- Redundant computation: recomputing the same values in overlapping subproblems

- Ensure a proper stopping condition to prevent endless calls
- Use iteration or optimize with tail recursion
- Implement memoization or switch to dynamic programming

# Comparing recursion with iteration

# Iteration versus recursion

## Recursion


- Calls itself to smaller subproblems
- Uses call stack, high memory overhead
- Can be slower
- Simpler for recursive problems like trees and graphs

## Iteration

- Uses loops for repetition
- Constant memory usage
- Generally faster
- Easier to debug and optimize

# Factorial: recursive versus iterative approach


## Recursion



```
public int factorialRec(int n) {  
    if (n == 1)        // Base Case  
        return 1;    // Recursive Case  
    return n * factorialRec(n - 1);  
}
```

- Elegant and simple but uses extra stack space ( $O(n)$  memory)
- $O(n)$  time,  $O(n)$  space (due to call stack)

## Iteration



```
public int factorialIter(int n)  
{    int result = 1;  
    for (int i = 2; i <= n; i++)  
        result *= i;  
    return result;  
}
```

- More efficient ( $O(1)$  memory) but requires explicit looping
- $O(n)$  time,  $O(1)$  space (more memory-efficient)

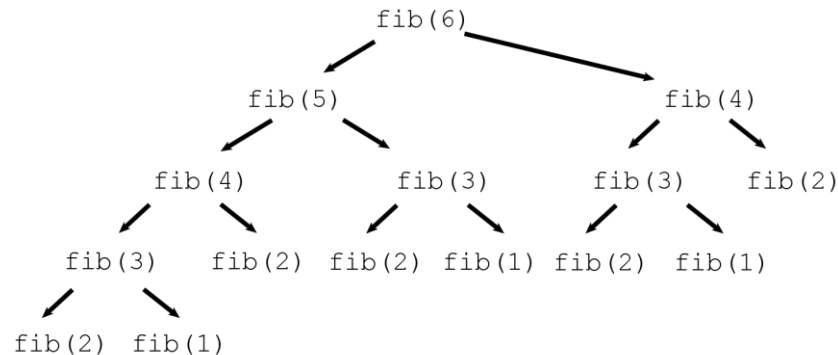
Iteration is generally preferred unless  
recursion improves readability


# Fibonacci: recursive versus iterative approach

## Recursion



```
public int fibonacci(int n) {  
    if (n == 0 || n == 1) // Base Case  
        return n;  
    else // Recursive Case  
        return fibonacci(n - 1) + fibonacci(n - 2);  
}
```



-  Inefficient due to repeated calculations, leading to exponential time complexity ( $O(2^n)$ )
- Exponential growth: each call spawns two more calls, leading to redundant calculations

- Stack overhead: large **n** causes deep recursion, increasing memory usage
- **Better alternative:** use iteration or dynamic programming (stay tuned) for improved efficiency



# Fibonacci: recursive versus iterative approach

## Iteration



```
public int fibonacciIter(int n) {  
    if (n == 0 || n == 1) return n;  
  
    int a = 0, b = 1, sum;  
    for (int i = 2; i <= n; i++) {  
        sum = a + b;  
        a = b;  
        b = sum;  
    }  
    return b;  
}
```

- Uses two variables to track previous Fibonacci numbers, updating them iteratively
- Each value is computed only once, unlike the recursive approach
- Time complexity:  $O(n)$ : computes each Fibonacci number in a single pass
- Space complexity:  $O(1)$ : uses only a few integer variables, no extra memory for recursion

# When to use recursion? Best cases and trade-offs

- **Divide and conquer problems:** like merge sort, quick sort, and binary search
- **Tree and graph traversals:** such as depth-first search (dfs) and pre-order/post-order traversal
- **Combinatorial problems:** like generating permutations or combinations
- **Backtracking problems:** solving puzzles like sudoku or the n-queens problem

Avoid recursion when the problem can be solved with simple loops, especially for small inputs.

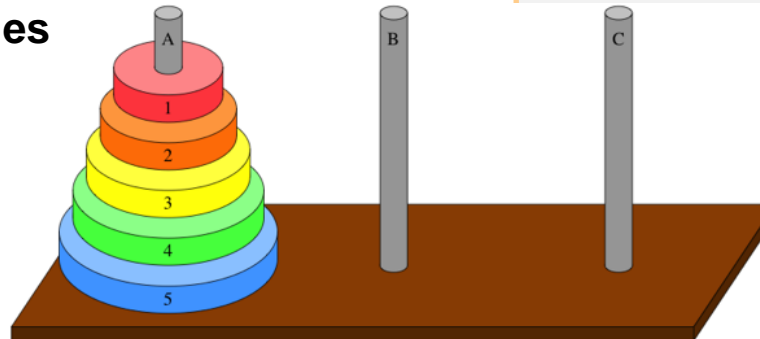
Recursion should be avoided for problems where deep recursion might lead to stack overflow or high memory usage

# Advanced recursion

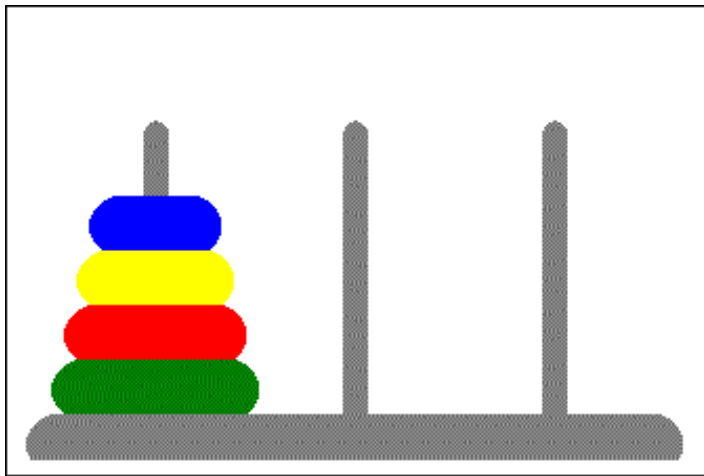
# Towers of Hanoi: a classic puzzle

- You have three rods and a set of disks of different sizes
- The disks are initially stacked on one rod, with the largest at the bottom and the smallest at the top
- **The goal is to move all disks to another rod following the rules**

1. Only one disk can be moved at a time
2. Each move consists of taking the top disk from one stack and placing it on another rod
3. No disk may be placed on top of a smaller disk



# Towers of Hanoi: strategy



- If only one disk is left, move it directly to the destination rod
- This is the simplest case and does not require further recursion
- Move the top  $n - 1$  disks from the source rod to the auxiliary rod, using the destination rod as temporary storage
- Move the  $n$ th disk (largest disk) directly to the destination rod
- Finally, move the  $n - 1$  disks from the auxiliary rod to the destination rod, using the source rod as temporary storage



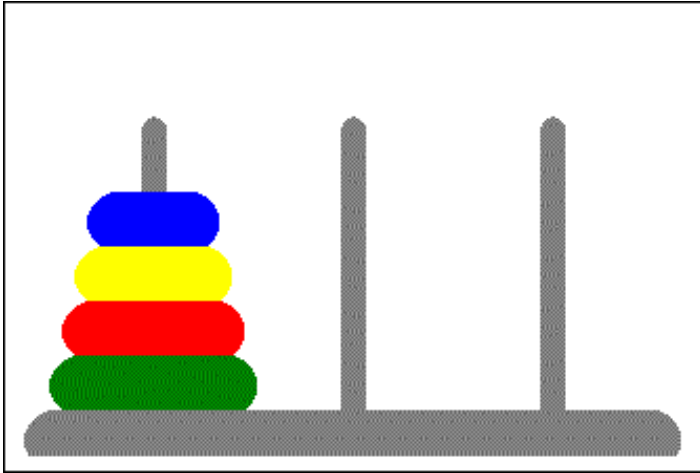
# Java implementation of Tower of Hanoi

```
public void solve(int n, String source, String temp, String destination) {  
    // Base case: If only one disk, move it directly  
    if (n == 1) {  
        System.out.println("Move disk 1 from " + source + " to " + destination);  
        return;  
    }  
  
    // Move n-1 disks from source to temp  
    solve(n - 1, source, destination, temp);  
  
    // Move the nth disk from source to destination  
    System.out.println("Move disk " + n + " from " + source + " to " + destination);  
  
    // Move n-1 disks from temp to destination  
    solve(n - 1, temp, source, destination);  
}
```

- Solves the problem in  $2^n - 1$  moves

- Base case:  
moves a single disk directly to the destination
- Recursive calls:  
move  $n - 1$  disks between rods before and after moving the largest disk

# Towers of Hanoi: complexity analysis



- The function calls itself twice for each disk
  1. To move  $n - 1$  disks from source to temporary rod
  2. To move  $n - 1$  disks from temporary rod to destination
- Recurrence:  $T(n) = 2 \cdot T(n - 1)$  leading to  $O(2^n)$  complexity
- **Total moves:** each recursive call splits the problem in half, making the number of calls double with each additional disk

# Quick sort as an example of recursion



Efficient divide-and-conquer algorithm that partitions and recursively sorts subarrays

- Choose a pivot to partition the array into smaller and larger elements
- Efficient for large datasets with average time complexity of  $O(n \log n)$
- In-place sorting, requiring no additional storage besides the input array

6 5 3 1 8 7 2 4



# Quick sort



```
function quickSort(array, low, high):  
    if low < high:  
        pivotIndex = partition(array, low, high)  
        quickSort(array, low, pivotIndex - 1)  
        quickSort(array, pivotIndex + 1, high)
```



```
function partition(array, low, high):  
    pivot = array[high]  
    i = low - 1  
    for j from low to high - 1:  
        if array[j] < pivot:  
            i = i + 1  
            swap(array[i], array[j])  
    swap(array[i + 1], array[high])  
    return i + 1
```

- Recursively divides the array based on the pivot index
- Sorts subarrays on either side of the pivot
- Rearranges the array so smaller elements are left of the pivot
- Returns pivot index, placing the pivot in its sorted position

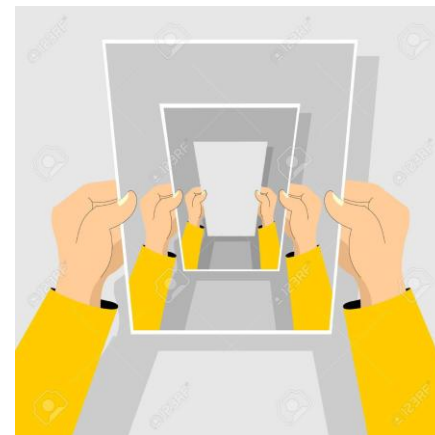
[Click here for Java implementation](#)

# Key Takeaways







Recursion simplifies many problems but comes with trade-offs in efficiency and memory

- Recursion: a method where a function calls itself to solve problems
- Base case & recursive case: essential components for preventing infinite recursion and solving the problem step-by-step
- Ideal for problems with self-similar subproblems, like tree traversals and backtracking



# Helpful resources on recursion

-  [Recursion tree visualizer](#)
-  [Coding with John](#): short tutorial on recursion fundamentals
-  A clear and structured [introduction](#) to recursion
-  [LeetCode](#) set of recursive problems

Recursion: because sometimes,  
going in circles is the only way to get somewhere

