# Recursion in JavaScript

# (Function Calling Itself + Problem Decomposition Model)

**Subtitle:** Learn how functions solve complex problems by calling themselves and breaking tasks into smaller identical subproblems.

**Website Name:** haas.dev
**Website Link:** https://dev-roast-app.vercel.app

## 🧠 Introduction

Recursion is where beginners usually get confused because it breaks the normal thinking model:

- normal function → runs once
- recursion → function runs itself repeatedly

But recursion is not “infinite looping logic.”

It is a **structured way of breaking a problem into smaller versions of itself**.

## ⚙️ Step 1: What is Recursion?

Recursion means:

A function calling itself until a stopping condition is met.

### Simple Example:

```
function count(n) {

  if (n === 0) return;


  console.log(n);

  count(n - 1);

}


count(5);
```

### Output:

5

4

3

2

1

# 🧠 Step 2: Why Recursion Exists

Some problems are naturally recursive:

- tree structures
- file systems
- nested categories
- mathematical sequences

Instead of loops, recursion makes logic cleaner.

## Mental Model:

Problem → Smaller Problem → Smaller Problem → Base Case → Stop

# ⚙️ Step 3: Two Core Parts of Recursion

Every recursive function MUST have:

## 1. Base Case (Stopping Condition)

if (n === 0) return;

👉 prevents infinite loop

## 2. Recursive Case (Self Call)

count(n - 1);

## Structure:

Function = Base Case + Recursive Call

# 🔄 Step 4: How Recursion Works Internally (Call Stack View)

Example:

```
function test(n) {
  if (n === 0) return;
  test(n - 1);
}

test(3);
```

## Stack Flow:

test(3)

→ test(2)

→ test(1)

→ test(0)

→ STOP

Then stack unwinds.

# 🧱 Step 5: Recursion vs Loop

| Feature | Loop | Recursion |
|---|---|---|
| Structure | iterative | self-calling |
| Memory | less | more (stack) |
| Readability | simple | better for nested logic |
| Use case | simple repetition | complex structures |

# 🌍 Step 6: Real-World Use Cases

## 1. File System Traversal

- folders inside folders
- recursive scanning

## 2. UI Trees (React, DOM)

- components inside components
- nested rendering

## 3. Organizational Structures

- departments
- sub-departments
- teams

# ⚠️ Step 7: Stack Overflow Problem

## ❌ Bad Recursion:

```
function loop() {

 loop();

}


loop();
```

## Result:

❌ Stack Overflow

## Why?

- no base case
- infinite function calls

# 🧠 Step 8: Classic Example — Factorial

```
function factorial(n) {

 if (n === 1) return 1;


 return n * factorial(n - 1);
```

}

console.log(factorial(5));

## Breakdown:

5 × 4 × 3 × 2 × 1 = 120

# 🔄 Step 9: Recursion Flow Visualization

factorial(5)

= 5 * factorial(4)

= 5 * 4 * factorial(3)

= 5 * 4 * 3 * factorial(2)

= 5 * 4 * 3 * 2 * factorial(1)

= 120

# ⚠️ Step 10: Common Mistakes

## ❌ 1. Missing base case

leads to infinite loop

## ❌ 2. Wrong condition logic

stopping too early or too late

## ❌ 3. Overusing recursion

loop would be better

## ❌ 4. Not understanding stack behavior

causes confusion in debugging

# 🧪 Step 11: Mini Exercises

Exercise 1

Print numbers from 10 to 1 using recursion

Exercise 2

Calculate sum of numbers using recursion

Exercise 3

Reverse a string using recursion

# 🧠 Step 12: Mini Quiz

1. What is recursion?
2. What is a base case?
3. Why does recursion use more memory?
4. When should recursion be avoided?

# 🚀 Step 13: Thinking Upgrade

If you understand recursion:

- you can solve tree problems
- you can understand compilers and engines better
- you can design scalable logic systems

👉 Recursion = thinking in breakdown patterns, not loops.

# 📌 Step 14: Summary

- Recursion = function calling itself
- Must have base case + recursive case
- Uses call stack internally
- Powerful for nested structures
- Can cause stack overflow if misused
- Essential for advanced JavaScript systems