MarxisSolution — Call Stack Visualizer

📚 Recursion Cheat Sheet

🔷 What is Recursion?

Recursion is a programming technique where a function calls itself to solve a problem by breaking it down into smaller, similar subproblems.

Two Essential Components:

Component	Description
Base Case	The stopping condition that prevents infinite recursion
Recursive Case	The function calls itself with a smaller input

The 4 Steps of Recursion:

1. **Define a base case** — Identify the simplest case with a known solution
2. **Define a recursive case** — Break the problem into smaller subproblems
3. **Ensure termination** — Verify each recursive call moves toward the base case
4. **Combine solutions** — Use the results from recursive calls to solve the original problem

Concept	Description
Call Stack	A memory structure that tracks active function calls
Stack Frame	A block of memory storing function arguments, local variables, and return address
Push	Adding a new frame to the top of the stack (function call)
Pop	Removing a frame from the top of the stack (function return)

MarxisSolution — Call Stack Visualizer

📚 Recursion Cheat Sheet

Concept	Description
Stack Depth	The number of frames currently on the stack
Stack Overflow	Occurs when the stack grows beyond available memory

🔷 Call Stack Visualization

How Recursion Uses the Stack:

text

```
factorial(5) called
  └── factorial(4) called
      └── factorial(3) called
          └── factorial(2) called
              └── factorial(1) called → Base case reached
              └── factorial(1) returned → 1
          └── factorial(2) returned → 2
      └── factorial(3) returned → 6
  └── factorial(4) returned → 24
factorial(5) returned → 120
```

🔷 Common Algorithms

1. Factorial

Definition: n! = n × (n-1) × (n-2) × ... × 1

Code:

c

```c
int factorial(int n) {
   if (n <= 1) return 1;        // Base case
   return n * factorial(n - 1);  // Recursive case
}
```

Time Complexity: O(n) | **Space Complexity:** O(n)

Input	Result
factorial(0)	1
factorial(1)	1
factorial(5)	120
factorial(7)	5040
factorial(10)	3,628,800

2. Fibonacci

Definition: fib(n) = fib(n-1) + fib(n-2), where fib(0) = 0, fib(1) = 1

Code:

c

```
int fibonacci(int n) {
   if (n <= 1) return n;            // Base case
   return fibonacci(n - 1) + fibonacci(n - 2); // Recursive case
}
```

Time Complexity: $O(2^n)$ | **Space Complexity:** O(n)

Input	**Result**	**Sequence**
fibonacci(0)	0	0
fibonacci(1)	1	0, 1
fibonacci(5)	5	0, 1, 1, 2, 3, 5

Input	Result	Sequence
fibonacci(6)	8	0, 1, 1, 2, 3, 5, 8
fibonacci(10)	55	0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55

3. Power (x^n)

Definition: $x^n = x \times x \times \ldots \times x$ (n times)

Code:

c

```
int power(int x, int n) {
   if (n == 0) return 1;        // Base case
   return x * power(x, n - 1);    // Recursive case
}
```

Time Complexity: O(n) | **Space Complexity:** O(n)

Input	Result
power(2, 0)	1
power(2, 3)	8
power(2, 5)	32
power(3, 4)	81
power(5, 3)	125

🔷 **Types of Recursion**

MarxisSolution — Call Stack Visualizer

📚 Recursion Cheat Sheet

Type	Description	Example
Linear Recursion	Each call makes at most one recursive call	Factorial
Tree Recursion	Each call makes multiple recursive calls	Fibonacci
Tail Recursion	The recursive call is the last operation	Some languages optimize this
Direct Recursion	Function calls itself directly	Factorial
Indirect Recursion	Function A calls B, B calls A	Mutual recursion

🔷 Common Mistakes to Avoid

Mistake	Explanation
❌ **No Base Case**	Function calls itself forever → Stack overflow
❌ **Wrong Base Case**	Base case is never reached → Infinite recursion
❌ **Not Making Progress**	Input doesn't move toward base case → Infinite recursion
❌ **Returning Wrong Type**	Ensure recursive calls match the function's return type

🔷 Quick Reference

Term	Definition
Recursive Call	A function calling itself with a smaller input

MarxisSolution — Call Stack Visualizer

📚 Recursion Cheat Sheet

Term	Definition
Base Case	The condition that stops recursion
Stack Frame	Memory block for each function call
Stack Depth	Number of active function calls
Stack Overflow	When the call stack runs out of memory
Memoization	Caching results to avoid re-computation

🔷 Complexity Summary

Algorithm	Time Complexity	Space Complexity
Factorial	O(n)	O(n)
Fibonacci	$O(2^n)$	O(n)
Power (x^n)	O(n)	O(n)

🔷 Interview Tips

1. **Identify the base case first** — Always start by defining when recursion should stop
2. **Trace the recursion** — Walk through a small example on paper or using a visualizer
3. **Avoid re-computation** — Use memoization for overlapping subproblems
4. **Consider tail recursion** — Some languages optimize tail calls
5. **Know your complexity** — Understand both time and space complexity

📌 Use the Call Stack Visualizer

Watch recursion in action! Visit: https://marxissolution.com/call-stack-visualizer/