Recursion & Induction (Practical)

What Problem This Solves

Recursion is solving a problem by solving smaller versions of the same problem.

Useful for:

- **Hierarchical data**: File systems, DOM trees, org charts
- **Divide-and-conquer**: Binary search, merge sort
- **Backtracking**: Generating permutations, solving puzzles
- **Mathematical sequences**: Fibonacci, factorials

When the structure is self-similar, recursion makes code cleaner.

Intuition & Mental Model

Think: Russian Nesting Dolls

Each doll contains a smaller version of itself:

```
Largest doll
  → Contains medium doll
    → Contains small doll
      → Contains tiny doll
        → Contains smallest doll (base case)
```

Recursion: Solve current level by opening it up to reveal smaller problem.

Base case: Smallest doll that doesn't open (stop condition).

Core Concepts

1. Anatomy of Recursion

Every recursive function needs:

1. **Base case**: When to stop
2. **Recursive case**: Call itself with smaller input
3. **Progress toward base**: Each call must get closer to stopping

```
function factorial(n) {
  // Base case
  if (n <= 1) return 1;

  // Recursive case (smaller input)
  return n * factorial(n - 1);
}

// factorial(5)
// = 5 * factorial(4)
// = 5 * (4 * factorial(3))
```

```
// = 5 * (4 * (3 * factorial(2)))
// = 5 * (4 * (3 * (2 * factorial(1))))
// = 5 * (4 * (3 * (2 * 1)))
// = 120
```

---

2. The Call Stack

Each function call adds a frame to the stack:

```javascript
function countdown(n) {
  if (n === 0) {
    console.log("Done!");
    return;
  }
  console.log(n);
  countdown(n - 1);
}

countdown(3);
```

Call stack visualization:

```
countdown(3)
  console.log(3)
  countdown(2)
    console.log(2)
    countdown(1)
      console.log(1)
      countdown(0)
        console.log("Done!")
        return
      return
    return
  return
```

Stack memory: Each call takes space → O(n) space complexity

---

3. Common Recursive Patterns

Tree Traversal

```javascript
const fileSystem = {
  name: 'root',
  type: 'folder',
  children: [
    { name: 'file1.txt', type: 'file' },
    {
      name: 'subfolder',
      type: 'folder',
      children: [
```

```
        { name: 'file2.txt', type: 'file' }
      ]
    }
  ]
};

function printFiles(node) {
  // Base case: file (leaf node)
  if (node.type === 'file') {
    console.log(node.name);
    return;
  }

  // Recursive case: folder (process children)
  for (const child of node.children) {
    printFiles(child);
  }
}

printFiles(fileSystem);
// Output: file1.txt, file2.txt
```

Divide and Conquer

```
// Binary search (recursive)
function binarySearch(arr, target, left = 0, right = arr.length - 1) {
  // Base case: not found
  if (left > right) return -1;

  const mid = Math.floor((left + right) / 2);

  // Base case: found
  if (arr[mid] === target) return mid;

  // Recursive case: search smaller half
  if (arr[mid] > target) {
    return binarySearch(arr, target, left, mid - 1);
  } else {
    return binarySearch(arr, target, mid + 1, right);
  }
}
```

Accumulation

```
// Sum array recursively
function sum(arr, index = 0) {
  // Base case: past end
  if (index >= arr.length) return 0;

  // Recursive case: current + rest
  return arr[index] + sum(arr, index + 1);
```

```
}

sum([1, 2, 3, 4, 5]); // 15
```

---

4. Recursion vs Iteration

Same problem, two approaches:

```
// Recursive
function factorialRec(n) {
  if (n <= 1) return 1;
  return n * factorialRec(n - 1);
}

// Iterative
function factorialIter(n) {
  let result = 1;
  for (let i = 2; i <= n; i++) {
    result *= i;
  }
  return result;
}
```

When to use each:

Recursion	Iteration
Tree/graph structures	Simple loops
Naturally recursive problems	Performance-critical code
Clearer code	Limited stack depth
Divide-and-conquer	Tail recursion (if no TCO)

Tradeoff: Recursion is often clearer but uses more memory (call stack).

---

5. Tail Recursion (Optimization)

Tail call: Recursive call is the last operation

```
// NOT tail recursive (multiplication after recursive call)
function factorial(n) {
  if (n <= 1) return 1;
  return n * factorial(n - 1); // ← Must remember n
}

// Tail recursive (uses accumulator)
function factorialTail(n, acc = 1) {
  if (n <= 1) return acc;
  return factorialTail(n - 1, n * acc); // ← Last operation
```

```
}

// Some engines optimize tail calls to not grow stack
```

JavaScript caveat: Most engines don't optimize tail calls (yet).

---

6. Mathematical Induction

Induction proves recursion is correct.

Structure:

1. **Base case**: Prove P(0) or P(1) is true
2. **Inductive step**: Prove if P(k) is true, then P(k+1) is true
3. **Conclusion**: P(n) is true for all n

Example: Prove sum of first n numbers = n(n+1)/2

```
function sumToN(n) {
  if (n === 1) return 1; // Base case
  return n + sumToN(n - 1); // Inductive step
}

// Base: sum(1) = 1 = 1(1+1)/2 ✓
// Inductive:
//   Assume sum(k) = k(k+1)/2
//   Then sum(k+1) = (k+1) + k(k+1)/2
//                 = (k+1)(1 + k/2)
//                 = (k+1)(k+2)/2 ✓
```

We won't prove everything, but induction explains why recursion works.

Software Engineering Connections

1. DOM Traversal

```
// Find all elements with class
function findByClass(node, className, results = []) {
  // Base case: check current node
  if (node.classList?.contains(className)) {
    results.push(node);
  }

  // Recursive case: check children
  for (const child of node.children) {
    findByClass(child, className, results);
  }

  return results;
}
```

```
// Usage
findByClass(document.body, 'highlight');
```

2. JSON Deep Clone

```
function deepClone(obj) {
  // Base cases
  if (obj === null || typeof obj !== 'object') {
    return obj;
  }

  if (Array.isArray(obj)) {
    return obj.map(item => deepClone(item));
  }

  // Recursive case: clone object
  const cloned = {};
  for (const key in obj) {
    cloned[key] = deepClone(obj[key]);
  }
  return cloned;
}
```

3. React Component Trees

```
function CommentThread({ comment }) {
  return (
    <div className="comment">
      <p>{comment.text}</p>
      {/* Recursive rendering of replies */}
      {comment.replies?.map(reply => (
        <CommentThread key={reply.id} comment={reply} />
      ))}
    </div>
  );
}

// Handles arbitrary nesting depth
```

4. File System Operations

```
async function getTotalSize(path) {
  const stats = await fs.stat(path);

  // Base case: file
  if (stats.isFile()) {
    return stats.size;
  }
```

```
  // Recursive case: directory
  const files = await fs.readdir(path);
  const sizes = await Promise.all(
    files.map(file => getTotalSize(path + '/' + file))
  );
  return sizes.reduce((sum, size) => sum + size, 0);
}
```

5. Memoization (Fixing Slow Recursion)

```
// Slow: O(2ⁿ) - recalculates same values
function fib(n) {
  if (n <= 1) return n;
  return fib(n - 1) + fib(n - 2);
}

// Fast: O(n) - cache results
function fibMemo(n, cache = {}) {
  if (n <= 1) return n;
  if (cache[n]) return cache[n];

  cache[n] = fibMemo(n - 1, cache) + fibMemo(n - 2, cache);
  return cache[n];
}

// Or using closure
function createFib() {
  const cache = {};
  return function fib(n) {
    if (n <= 1) return n;
    if (cache[n]) return cache[n];
    cache[n] = fib(n - 1) + fib(n - 2);
    return cache[n];
  };
}
```

6. Backtracking

```
// Generate all permutations
function permute(arr, current = [], results = []) {
  // Base case: complete permutation
  if (current.length === arr.length) {
    results.push([...current]);
    return;
  }

  // Recursive case: try each unused element
  for (const item of arr) {
```

```
    if (current.includes(item)) continue;

    current.push(item);
    permute(arr, current, results);
    current.pop(); // Backtrack
  }

  return results;
}

permute([1, 2, 3]);
// [[1,2,3], [1,3,2], [2,1,3], [2,3,1], [3,1,2], [3,2,1]]
```

Common Misconceptions

❌ "Recursion is always slower"

Sometimes clearer and not meaningfully slower. For tree structures, recursion is natural and performant.

❌ "Recursion = stack overflow"

Only if too deep or missing base case. Most recursive problems have reasonable depth.

```
// Safe: depth = tree height (usually log n)
function treeHeight(node) {
  if (!node) return 0;
  return 1 + Math.max(treeHeight(node.left), treeHeight(node.right));
}

// Dangerous: depth = n (for large n)
function sumToN(n) {
  if (n === 0) return 0;
  return n + sumToN(n - 1); // Depth = n
}
// sumToN(100000) → stack overflow
```

❌ "Can't convert recursion to iteration"

Any recursion can be converted to iteration (using explicit stack). Not always clearer.

```
// Recursive
function factorial(n) {
  if (n <= 1) return 1;
  return n * factorial(n - 1);
}

// Iterative equivalent
function factorialIter(n) {
  let result = 1;
  while (n > 1) {
    result *= n;
```

```
    n--;
  }
  return result;
}
```

❌ "Memoization fixes all slow recursion"

Only if subproblems repeat. Unique subproblems still take time.

```
// Memoization helps (repeated subproblems)
fib(5) calls fib(3) multiple times → cache helps

// Memoization doesn't help (unique subproblems)
factorial(5) calls factorial(4) once → cache doesn't help
```

Practical Mini-Exercises

Exercise 1: Flatten Nested Array

```
const nested = [1, [2, [3, [4]], 5], 6];
// Expected: [1, 2, 3, 4, 5, 6]
```

Write recursive function to flatten.

▶ Solution

Exercise 2: Count Files in Directory

Given a file tree structure, count total files:

```
const tree = {
  type: 'folder',
  children: [
    { type: 'file' },
    {
      type: 'folder',
      children: [
        { type: 'file' },
        { type: 'file' }
      ]
    },
    { type: 'file' }
  ]
};
```

▶ Solution

Exercise 3: Optimize Fibonacci

Fix this slow implementation:

```
function fib(n) {
  if (n <= 1) return n;
  return fib(n - 1) + fib(n - 2);
}

// fib(40) takes seconds!
```

▶ Solution

Summary Cheat Sheet

Recursion Template

```
function recursive(input) {
  // Base case(s)
  if (/* stopping condition */) {
    return /* simple result */;
  }

  // Recursive case
  // 1. Make input smaller
  // 2. Call self
  // 3. Combine results
  return /* combine */( recursive(/* smaller input */) );
}
```

Common Patterns

```
// Tree traversal
function traverse(node) {
  if (!node) return;
  process(node);
  traverse(node.left);
  traverse(node.right);
}

// Accumulator
function sum(arr, i = 0, acc = 0) {
  if (i >= arr.length) return acc;
  return sum(arr, i + 1, acc + arr[i]);
}

// Backtracking
function backtrack(choices, path, results) {
  if (/* complete */) {
    results.push([...path]);
    return;
  }
  for (const choice of choices) {
```

```
    path.push(choice);
    backtrack(choices, path, results);
    path.pop();
  }
}
```

When to Use

Use Recursion	Use Iteration
Trees/graphs	Simple sequences
Nested structures	Performance-critical
Divide-and-conquer	Large depth
Natural self-similarity	Memory-constrained

Optimization

```
// Memoization
const cache = new Map();
function memoized(n) {
  if (cache.has(n)) return cache.get(n);
  const result = /* compute */;
  cache.set(n, result);
  return result;
}
```

Next Steps

Recursion is a powerful tool for elegant solutions to naturally self-similar problems. You now understand when and how to use it effectively.

Next, we'll explore **probability basics**—understanding randomness and uncertainty in systems.

Continue to: 06-probability-basics.md