**When to Use Recursion Instead of Iteration**

**Excellent and very important question.**  
The key insight is that *any problem solvable with iteration (loops) can be solved with recursion, and vice versa*. They are equivalent in terms of computational power.

However, some problems are **inherently recursive** in nature. Their structure is self-similar, meaning the problem itself is defined in terms of smaller instances of the same problem. For these, a recursive solution is often the most intuitive, natural, and elegant approach.

An iterative solution would be possible but would require manually simulating recursion using a stack data structure, making it more complex and less readable.

**1. Problems Defined Recursively by Nature**

These are situations where the mathematical or conceptual definition itself is recursive:

* **The Fibonacci Sequence**: **(but recursion is not a performance efficient solution. This can be done for learning purpose)** F(n) = F(n-1) + F(n-2)
* **Factorial of a number**: **(but recursion is not a performance efficient solution. This can be done for learning purpose)**
* n! = n \times (n-1)!
* **Tree Data Structure**:  
  A tree is defined as a root node with a set of child nodes, each of which is itself a tree.
* **File System**:  
  A directory contains files and other directories, which in turn contain more files and directories. This forms a recursive structure.

**2. Problems Involving Tree Traversal and Tree-shaped Data**

This is the most natural and powerful use case for recursion.

* **Depth-First Search (DFS) in Trees**: Pre-order, In-order, and Post-order traversals fit recursion perfectly.
* **Calculating Tree Properties**:  
  Example:  
  $ height(node) = 1 + \max(height(node.left), height(node.right)) $  
  Problems include:
  + Finding the height of a tree
  + Summing all nodes
  + Checking if a tree is symmetric

**3. Problems Involving Backtracking**

Backtracking explores multiple possibilities and abandons invalid paths. Recursion provides a neat way to handle this logic.

* **Generating Permutations or Combinations**:  
  Recursive choices handle element placement and remaining elements.
* **Solving Puzzles**:  
  Examples:
  + **N-Queens Problem**
  + **Sudoku Solver**  
    Both follow the pattern: *make a choice → explore → undo and try next*.

**4. Problems Involving Divide and Conquer**

Divide and Conquer problems naturally map to recursion, since they reduce a big problem into smaller subproblems of the same type.

* **Merge Sort**: Recursively split into halves, then merge results.
* **Quick Sort**: Pick pivot → sort subarrays recursively.
* **Binary Search**: Check middle element, then search left or right half recursively.

**Summary Table**

|  |  |  |
| --- | --- | --- |
| Problem Type | Why Recursion is Ideal | Classic Example |
| **Recursively Defined** | Code directly mirrors the mathematical definition. | Fibonacci, Factorial ( Both are not of high performance solution. Can be used for educative purpose) |
| **Tree Operations** | Data structure is recursive; code navigates it easily | DFS, Tree height |
| **Backtracking** | Fits the "choose → explore → unchoose" call stack | N-Queens, Permutations |
| **Divide and Conquer** | Breaks into smaller identical sub-problems naturally | Merge Sort, Quick Sort |

**Conclusion**

Although any recursive algorithm can be rewritten iteratively by simulating a stack, recursion shines when problems have a naturally recursive **structure** or **definition**.  
It allows you to describe a solution very intuitively: *"solve this by solving a smaller version of it."*