Mastering Recursion: A Step-by-Step Approach

Note: This post is an adaptation and translation inspired by a blog post by Eddy Song. I found it insightful and wanted to share this with minor refinement.

Recursion can be challenging to understand, even with numerous online resources available. My initial apprehension toward recursion stemmed from a computer science professor who once described it as "evil." As a result, I avoided recursion until I discovered that it is essential when working with certain data structures. This realization initially led to a sort of "brain overflow."

Ironically, the best way to approach recursion is to not overthink it. Don't try to visualize every call and return step in the recursive flow. Instead, break the problem down into four simple parts: necessity, base case, decomposition, and assembly.

The 4 Steps to Recursion

1. Is Recursion Necessary?

Not all problems require recursion. In fact, many problems can be solved more efficiently with iteration. However, recursion is necessary for certain tasks, such as depth-first search (DFS) and dynamic programming, which build directly on recursive logic.

2. Base Case

The base case is the condition that stops the recursion. It's similar to the base case in mathematical induction—it's the simplest input where the function can return a result without further recursive calls. This case ensures that the function doesn't run indefinitely.

Think of the base case as a situation where an immediate answer is possible, often with inputs of size 0 or 1:

- **Integers**: The base case could be when the input is `0` or `1`.
- **Arrays/Strings**: The base case is when the length is `0` or `1`.
- **Trees**: The base case is when the input node is `null` or has no children.

Identify the return type for the base case based on the problem. For example, if you need a sum, the base case should return an integer.

3. Decomposition

Decompose the input so that each recursive call brings you closer to the base case. This means modifying the argument passed to the recursive call:

- **Integers**: Pass `n - 1` or `n - 2`.

- **Arrays/Strings**: Remove an element to reduce the size (e.g., `[1, 2, 3, 4]` → `[2, 3, 4]` → `[3, 4]`).
- **Linked Lists**: Move to `next` or `next->next`.
- **Trees**: Pass a child node.

The goal is for the recursive input to approach the base case. If you're unsure how to decompose the input, move on to the assembly step and revisit decomposition later.

4. Assembly

Assembly is the process of combining the results from partial solutions to form the final answer. This happens when the recursion reaches the base case and starts to return. You don't need to visualize every step; instead, consider:

- **The step immediately above the base case**: This is the simplest assembly. For example, if the base case is `n == 1`, think about what happens when `n == 2`.
- **A few steps above**: Consider up to three steps above the base case. For instance, if the base case is `n == 1`, think about what happens at `n == 4`.

Use the **Recursive Leap of Faith**: assume the recursive call works correctly without tracing every detail.

Applying the 4 Steps: The Reverse String Problem (Source LeetCode)

Problem Statement: Write a function that reverses a string. The input is given as an array of characters `s`.

Example:

- **Input**: `s = ["h", "e", "l", "l", "o"]`
- **Output**: `["o", "l", "l", "e", "h"]`

Step-by-Step Solution

1. Is Recursion Necessary?
No, recursion isn't required for this problem—an iterative solution is sufficient. But to practice recursion, we'll use it here.

2. Base Case
What's the simplest condition where we can return a result directly? When the length of the input string is `0`, return an empty string. When the length is `1`, return the string as-is.

3. Decomposition
We need to decompose the input so it eventually reaches the base case. For the string "hello":

- Remove the first character: `"hello"` → `"ello"` → `"llo"` → `"lo"` → `"o"`.

4. Assembly
Now, think about how to use the result of a subproblem to build the full solution:

- **Right above the base case**: If the input is `"hi"`, the base case for `"i"` returns `"i"`. To get `"ih"`, append `"h"` to `"i"`.
- **Three steps above**: For the input `"gray"`, the output should be `"yarg"`. Assume the recursive call on `"ray"` returns `"yar"`. To build `"yarg"`, append `"g"` to `"yar"`.

General Pattern

To generalize, add the first character of the current input to the result of the recursive call:

- `reverseString("hi")` returns `"i" + "h"`.
- `reverseString("gray")` returns `"yar" + "g"`.

Code Implementation

```
std::string reverseString(const std::string& s) {
    // Base case: if the string is empty, return an empty string
    if (s.empty()) {
        return "";
    }
    // Recursive case: call reverseString on the substring without the first character
    // and append the first character to the result
    return reverseString(s.substr(1)) + s[0];
}
```

Final Thoughts

This step-by-step approach outlines how to understand and apply recursion effectively. While recursion isn't always necessary, practicing with simple problems builds a foundation for tackling more complex recursive algorithms.

Source

https://velog.io/@eddy_song/you-can-solve-recursion
https://leetcode.com/problems/reverse-string/