Understanding Recursion

A Step-by-Step Approach

Programming Fundamentals

December 4, 2024

What is Recursion?

- A method where a function calls itself
- Solves problems by breaking them into smaller subproblems
- Two main components:
  - Base case(s)
  - Recursive case(s)

Key Concepts

- **Base Case**
  - Condition where recursion stops
  - Returns value directly without recursion
- **Recursive Case**
  - Problem broken into smaller subproblems
  - Function calls itself with modified parameters

Example 1: Factorial

- Definition: n! = n $\times$ (n-1)!
- Base case: 0! = 1! = 1

```
function fact = factorial_recursive(n)
    if n == 0 || n == 1
        fact = 1;
    else
        fact = n * factorial_recursive(n-1);
    end
end
```

Factorial Call Stack

5! calculation:

1. factorial(5) $\rightarrow$ 5 $\times$ factorial(4)
2. factorial(4) $\rightarrow$ 4 $\times$ factorial(3)
3. factorial(3) $\rightarrow$ 3 $\times$ factorial(2)
4. factorial(2) $\rightarrow$ 2 $\times$ factorial(1)
5. factorial(1) $\rightarrow$ 1

Then unwind: $1 \rightarrow 2 \rightarrow 6 \rightarrow 24 \rightarrow 120$

Example 2: Fibonacci Sequence

- Each number is sum of previous two
- Base cases: F(1) = F(2) = 1

```
function fib = fibonacci_recursive(n)
    if n <= 2
        fib = 1;
    else
        fib = fibonacci_recursive(n-1) +
             fibonacci_recursive(n-2);
    end
end
```

Recursion vs Iteration

Recursion

- Elegant and clear
- Memory intensive
- Natural for tree structures

Iteration

- More efficient
- Less memory usage
- Better for linear problems

Example 3: Binary Search

```
function index = binary_search_recursive(arr, target, le
    if left > right
        index = -1;
        return;
    end

    mid = floor((left + right)/2);
    if arr(mid) == target
        index = mid;
    elseif arr(mid) > target
        index = binary_search_recursive(arr, target, lef
    else
        index = binary_search_recursive(arr, target, mid
    end
end
```

Classic Example: Tower of Hanoi

- Problem:
  - Move n disks from source to target
  - Using auxiliary pole
  - Larger disk cannot be on smaller disk
- Recursive solution:
  1. Move n-1 disks to auxiliary
  2. Move largest disk to target
  3. Move n-1 disks to target

Tower of Hanoi Implementation

```
function tower_of_hanoi(n, source, auxiliary, target)
    if n == 1
        fprintf('Move disk 1 from %s to %s\n',
                source, target);
        return;
    end

    tower_of_hanoi(n-1, source, target, auxiliary);
    fprintf('Move disk %d from %s to %s\n',
            n, source, target);
    tower_of_hanoi(n-1, auxiliary, source, target);
end
```