Recursion Cheat Sheet

What is Recursion?

Recursion is a powerful programming technique where a function calls itself directly or indirectly to solve a problem. It's a method of solving a computational problem where the solution depends on solutions to smaller instances of the same problem [1]. Think of it as breaking down a complex problem into smaller, more manageable sub-problems until a simple, solvable base case is reached. This self-referential nature is what defines recursion.

Key Components of a Recursive Function:

Every well-formed recursive function must have two main components:

1. **Base Case:** This is the condition that stops the recursion. Without a base case, the function would call itself indefinitely, leading to a stack overflow. The base case provides a direct solution for the simplest instance of the problem.
2. **Recursive Step:** This is where the function calls itself with a modified input, moving closer to the base case. The problem is broken down into a smaller sub-problem of the same type.

How Recursion Works: The Call Stack

When a recursive function is called, each call is placed onto a data structure called the **call stack**. The call stack operates on a Last-In, First-Out (LIFO) principle. When a function is called, its execution context (including local variables and parameters) is pushed onto the stack. When the function returns, its context is popped off the stack. In recursion, this means that the most recent recursive call is executed first, and once it returns, the previous call resumes execution. This process continues until the base case is reached, and then the results propagate back up the stack as each function call returns.

Recursion in C++: Examples

Let's explore some classic examples of recursion implemented in C++.

1. Factorial Calculation

The factorial of a non-negative integer `n` is the product of all positive integers less than or equal to `n`. The factorial of 0 is 1. This can be defined recursively as:

- `n! = n * (n-1)!` for `n > 0`
- `0! = 1` (Base Case)

```cpp
#include <iostream>

long long factorial(int n) {
    // Base case: if n is 0, return 1
    if (n == 0) {
        return 1;
    }
    // Recursive step: n * factorial of (n-1)
    return n * factorial(n - 1);
}

int main() {
    int number = 5;
    std::cout << "Factorial of " << number << " is: " << factorial(number) << std::endl; // Output: 120
    return 0;
}
```

Explanation:

When `factorial(5)` is called: 1. `factorial(5)` calls `factorial(4)` 2. `factorial(4)` calls `factorial(3)` 3. `factorial(3)` calls `factorial(2)` 4. `factorial(2)` calls `factorial(1)` 5. `factorial(1)` calls `factorial(0)` 6. `factorial(0)` returns `1` (base case) 7. `factorial(1)` receives `1` and returns `1 * 1 = 1` 8. `factorial(2)` receives `1` and returns `2 * 1 = 2` 9. `factorial(3)` receives `2` and returns `3 * 2 = 6` 10. `factorial(4)` receives `6` and returns `4 * 6 = 24` 11. `factorial(5)` receives `24` and returns `5 * 24 = 120`

2. Fibonacci Sequence

The Fibonacci sequence is a series of numbers where each number is the sum of the two preceding ones, usually starting with 0 and 1. The sequence begins: 0, 1, 1, 2, 3, 5,

8, 13, ...

- `Fib(n) = Fib(n-1) + Fib(n-2)` for `n > 1`
- `Fib(0) = 0` (Base Case 1)
- `Fib(1) = 1` (Base Case 2)

```cpp
#include <iostream>

int fibonacci(int n) {
    // Base cases
    if (n == 0) {
        return 0;
    }
    if (n == 1) {
        return 1;
    }
    // Recursive step
    return fibonacci(n - 1) + fibonacci(n - 2);
}

int main() {
    int number = 7;
    std::cout << "Fibonacci of " << number << " is: " << fibonacci(number) <<
std::endl; // Output: 13
    return 0;
}
```

Explanation:

When `fibonacci(7)` is called, it recursively breaks down into `fibonacci(6) + fibonacci(5)`, and so on, until it hits the base cases `fibonacci(0)` or `fibonacci(1)`. The results then sum up as the calls return.

Advantages and Disadvantages of Recursion

Advantages:

- **Elegance and Readability:** Recursive solutions can often be more elegant and easier to understand for problems that are inherently recursive (e.g., tree traversals, fractal generation).
- **Reduced Code Size:** For certain problems, recursion can lead to more concise code compared to iterative solutions.

- **Problem Decomposition:** It naturally breaks down complex problems into smaller, identical sub-problems.

Disadvantages:

- **Memory Overhead:** Each recursive call adds a new frame to the call stack. If the recursion depth is too large, it can lead to a stack overflow error.
- **Slower Execution:** Function calls have overhead (saving context, passing parameters), which can make recursive solutions slower than iterative ones for some problems.
- **Debugging Difficulty:** Tracing the execution flow of a recursive function can be challenging due to the nested calls.
- **Performance Issues (for some problems):** Naive recursive implementations (like the Fibonacci example above) can lead to redundant calculations, significantly impacting performance. This can often be mitigated using memoization or dynamic programming.

When to Use Recursion?

Recursion is particularly well-suited for problems that can be defined in terms of smaller, similar sub-problems. Common scenarios include:

- **Tree and Graph Traversal:** Algorithms like Depth-First Search (DFS) are naturally recursive.
- **Divide and Conquer Algorithms:** Examples include Merge Sort and Quick Sort.
- **Mathematical Functions:** Factorial, Fibonacci, power functions.
- **Parsing and Language Processing:** Compilers often use recursion to parse syntax trees.
- **Fractals and Recursive Patterns:** Generating self-similar geometric shapes.

Iteration vs. Recursion

Almost every recursive problem can also be solved iteratively using loops (e.g., `for`, `while`). The choice between recursion and iteration often depends on the problem's

nature, performance requirements, and readability preferences.

Feature	Recursion	Iteration
Definition	Function calls itself	Uses loops (for, while)
Termination	Base case	Loop condition
Memory	More (uses call stack)	Less (no call stack overhead)
Speed	Generally slower (function call overhead)	Generally faster
Code Size	Can be more concise	Can be more verbose
Readability	Elegant for naturally recursive problems	Straightforward for sequential problems
Debugging	More challenging	Easier

Conclusion

Recursion is a fundamental concept in computer science that offers an elegant way to solve complex problems by breaking them down into simpler, self-similar parts. While it comes with certain overheads, its power in handling specific problem structures, especially those involving hierarchical data, makes it an indispensable tool in a programmer's arsenal. Understanding its mechanics, advantages, and disadvantages is crucial for writing efficient and maintainable code.

References

[1] GeeksforGeeks. "What is Recursion?". Available at: https://www.geeksforgeeks.org/dsa/what-is-recursion/