Recursive

KUCSEPotato

January 2026

1 Recursive Functions

Recursive function(재귀 함수)는 함수의 정의 과정에서 자기 자신을 호출하는 함수를 의미한다. 재귀 함수의 개념은 수학적 정의에서 기원하며, 수학에서 재귀가 사용되는 대표적인 정의로는 자연수, 팩토리얼, 피보나치 수열 등이 있다.

Examples of Recursive Definitions

Definition of Natural Numbers 자연수 집합 $\mathbb{N}$은 다음과 같이 재귀적으로 정의할 수 있다.

- $1 \in \mathbb{N}$
- $n \in \mathbb{N}$이면 $n+1 \in \mathbb{N}$

이는 집합을 자기 자신을 이용하여 정의한 대표적인 예이다.

Factorial 팩토리얼 함수는 다음과 같이 재귀적으로 정의된다.

$$n! = n \cdot (n-1)! \quad (n \geq 2),$$

여기서 기저 사례는

$$1! = 1$$

이다.

Fibonacci Sequence 피보나치 수열은 다음과 같은 재귀 관계로 정의된다.

$$F(n) = F(n-1) + F(n-2) \quad (n \geq 3),$$

기저 사례는

$$F(1) = 1, \quad F(2) = 1$$

이다.

위와 같은 정의들은 재귀적 정의를 사용하지 않는다면, 동일한 규칙을 모든 경우에 대해 간결하고 일반적으로 표현하기 어렵다. 이는 프로그래밍에서도 동일하게 적용된다. 초기 프로그래밍 언어들(e.g. assembly, Fortran)은 오늘날 우리가

말하는 형태의 "재귀성(recursion)"을 자연스럽게 표현하기 어려웠다. 그 결과, 재귀적으로 정의되는 문제(트리 순회, 분할정복, 백트래킹 등)를 구현하려면 개발자가 *직접* 상태(state)와 복귀 지점(return address)을 관리해야 했고, 이는 코드의 복잡도를 크게 증가시키며 버그 가능성을 높였다.

2 Why Recursion Was Introduced into Programming

재귀가 프로그래밍에 도입된 배경을 한 문장으로 요약하면 다음과 같다.

> 재귀는 **컴퓨터를 위해서**가 아니라, **사람이 문제를 정의하고 이해하는 방식**을 코드로 직접 옮기기 위해 도입되었다.

2.1 Recursion as a Language for Definitions

많은 문제는 "큰 문제"가 "같은 형태의 더 작은 문제"로 표현되는 구조를 갖는다. 재귀는 이러한 구조를 코드로 그대로 반영하는 표현 수단이다. 예를 들어, 다음과 같은 문장은 매우 자연스럽다.

- (트리) 트리는 하나의 노드와 여러 개의 (자식) 트리로 구성된다.
- (분할정복) 정렬은 "왼쪽 절반 정렬"과 "오른쪽 절반 정렬"을 수행한 뒤 두 결과를 합치는 과정으로 정의될 수 있다.
- (탐색) 현재 선택 이후의 문제는 "현재 선택을 고정한 상태에서 남은 부분"을 푸는 문제로 환원된다.

이처럼 문제 정의 자체가 재귀적일 때, 재귀 함수는 그 정의를 코드로 직접 구현할 수 있게 해준다. 즉, 재귀는 단순한 "반복문의 대체"가 아니라 **문제를 기술하는 언어적 장치**로 이해하는 것이 중요하다.

2.2 Early Programming and the Burden of Manual State Management

초기 환경에서 "함수가 끝난 뒤 어디로 돌아가야 하는가"(복귀 주소)와 "현재까지의 중간 결과는 어디에 저장해야 하는가"(지역 변수, 매개변수 등)를 체계적으로 처리하기는 쉽지 않았다. 특히 다음과 같은 형태의 계산은 자연스럽게 "이전 상태를 저장하고 다시 돌아와 이어서 처리"해야 한다.

- 트리/그래프 탐색: 한 노드의 처리가 끝나면 부모(또는 이전 분기)로 돌아가 다른 자식을 처리해야 한다.
- 백트래킹: 선택을 진행하다가 불가능해지면 이전 선택으로 되돌아가 다른 선택지를 시도해야 한다.

재귀 호출이 가능한 언어에서는 이 "되돌아감"이 **호출 스택(call stack)**에 의해 자동으로 관리된다. 반면 재귀 표현이 어렵거나 없던 환경에서는, 개발자가 직접 스택 자료구조를 구현하거나 점프(jump) 중심의 코드를 구성하여 상태를 관리해야 했다. 이 과정은 문제의 핵심 로직보다 "기계적인 상태 관리 코드"가 더 커지는 현상을 만들 수 있다.

3 Call Stack: The Mechanism Behind Recursion

재귀의 핵심 실행 메커니즘은 호출 스택이다. 함수 호출은 실행 도중 필요한 정보를 스택에 쌓고, 함수가 종료되면 마지막에 쌓인 정보를 꺼내어 이전 위치로 복귀한다 (LIFO: Last-In First-Out).

3.1 What Is Stored in a Stack Frame?

일반적으로 한 번의 함수 호출은 다음 정보를 포함하는 *스택 프레임(stack frame)*을 생성한다.

- **Return address**: 함수가 종료된 뒤 돌아갈 코드 위치
- **Parameters**: 매개변수 값
- **Local variables**: 지역 변수 및 임시 값
- **Saved registers / bookkeeping**: 구현(컴파일러/ABI)에 따라 필요한 보조 정보

이 구조 덕분에, 재귀 함수는 "현재 문제를 해결하던 문맥"을 잃지 않고 더 작은 문제로 내려갔다가 다시 돌아와 후속 처리를 수행할 수 있다.

3.2 Base Case and Termination

재귀가 올바르게 동작하려면 반드시 다음 두 조건이 필요하다.

1. **기저 사례(base case)**가 존재해야 한다.
2. 재귀 호출은 입력 크기를 줄이거나(예: $n \mapsto n-1$, $n \mapsto n/2$), 종료를 향해 진행하는 **감소(decrease)**를 가져야 한다.

이를 만족하지 못하면 무한 재귀(infinite recursion)가 발생하며, 현실의 시스템에서는 스택이 한정되어 있으므로 **스택 오버플로우(stack overflow)**로 종료된다.

4 Recursion Patterns in Problem Solving

재귀가 유용한 대표 유형을 세 가지로 나누어 정리한다.

4.1 Linear Recursion

입력 크기가 한 단계씩 감소하는 형태이다. 예를 들어 다음 점화식과 대응된다.

$$T(n) = T(n-1) + O(1) \quad \Rightarrow \quad T(n) \in O(n).$$

대표 예시는 팩토리얼, 단순 누적합 등이 있다.

4.2 Divide and Conquer

문제를 여러 개의 더 작은 문제로 분할한 뒤 결과를 결합한다. 표준 형태는 다음과 같다.

$$T(n) = aT\left(\frac{n}{b}\right) + f(n).$$

예를 들어 병합 정렬은

$$T(n) = 2T(n/2) + O(n) \Rightarrow T(n) \in O(n \log n).$$

별찍기(프랙탈) 문제처럼 "이전 패턴을 3×3 블록으로 확장"하는 문제 또한 이 범주에 속한다.

4.3 Backtracking / Search Tree

각 단계에서 여러 선택지(branch)가 존재하고, 조건을 만족하지 못하면 되돌아가 다른 선택지를 시도한다. 이 유형은 상태 공간(state space)이 트리 형태로 전개되며 최악의 경우 지수 시간이 될 수 있다. 따라서 **가지치기(pruning)**를 통해 탐색 공간을 줄이는 것이 중요하다.

5 Recursion vs. Iteration

중요한 사실 하나는 다음과 같다.

> 이론적으로 재귀는 반복(iteration)으로 항상 변환 가능하다.

즉, 재귀는 "할 수 없는 일을 가능하게 만드는 기능"이라기보다 **표현력과 가독성을 높이는 방법**이다. 재귀 호출을 반복문으로 바꾸는 과정은 대체로 다음 아이디어와 연결된다.

- 재귀 호출은 "암묵적 스택(호출 스택)"을 사용한다.
- 반복문 구현은 "명시적 스택(자료구조)"을 사용한다.

예를 들어 트리 DFS는 재귀로 쓰면 정의가 간결하지만, 반복문으로도 스택을 이용해 구현할 수 있다. 따라서 실전에서는 다음의 균형을 고려한다.

- **가독성/정의의 일치**: 재귀가 더 자연스러운가?
- **스택 깊이 제한**: 입력 크기에서 재귀 깊이가 안전한가?
- **성능**: 호출 오버헤드 또는 중복 계산이 문제인가?

6 When Recursion Becomes Inefficient

재귀가 항상 좋은 선택은 아니다. 특히 다음 상황에서는 주의가 필요하다.

6.1 (1) Overlapping Subproblems

서로 다른 경로에서 같은 부분 문제가 반복 계산되는 경우, 순수 재귀는 매우 비효율적일 수 있다. 예를 들어 피보나치 수열의 순진한 재귀 구현은

$$T(n) \approx T(n-1) + T(n-2) + O(1)$$

의 형태를 가지며 지수적으로 느려진다. 이때는 **메모이제이션(memoization)**을 도입하여 Top-down DP로 바꾸면 중복 계산을 제거할 수 있다.

6.2 (2) Deep Recursion

입력 크기에 비례하여 깊이가 커지는 선형 재귀는 언어/환경에 따라 스택 한계를 초과할 수 있다. 이 경우 반복문으로 변환하거나, DP(바텀업)로 재설계하는 것이 안전하다.

7 Summary

재귀가 프로그래밍에 도입된 이유는 단순히 "편리해서"가 아니라, **재귀적으로 정의되는 문제를 사람이 이해하는 방식 그대로 표현**하기 위함이다. 특히 트리/그래프, 분할정복, 백트래킹과 같이 구조적으로 재귀성이 내재된 문제에서 재귀는 정의와 구현을 일치시키고 상태 관리를 단순화한다. 다만 중복 계산과 스택 깊이 제한을 고려하여, 필요할 경우 메모이제이션 또는 반복문(명시적 스택)으로의 변환을 함께 사용할 수 있어야 한다.