재귀(Recursion)

▼ 재귀(Recursion)란?

- 문제를 해결할 때 동일한 구조의 더 작은 문제를 해결함으로써 주어진 문제를 해결하는 방법
- 모든 재귀는 반복문으로 표현 가능하며, 문제를 더 작은 문제로 분해할 수 있거나 반복문의 중첩이 많아서 결과를 예측하기 힘든 경우 유용하게 사용
- 코드의 가독성과 유지보수성 향상

▼ 재귀 함수(Recursion Function)

- 스스로를 호출하는 함수

성능 문제

- 호출될 때마다 메모리에 스택이 쌓이고, 한계치 이상으로 호출이 되어 스택이 넘쳐버리면 메모리 부족으로 에러 발생
- jump가 잦아 반복문에 비해 시간이 오래 걸림
- 꼬리 재귀 최적화 기능을 통해 해결 : return 값이 함수 자체 뿐
  * 꼬리 재귀 최적화 (Tail Call Optimization) : 재귀 함수를 컴퓨터가 재해석하여 선형 알고리즘을 만들어 실행, 아무리 반복이 일어나도 스택이 넘치지 않음

▼ 재귀 호출(Recursive call)

- 재귀 함수는 실행 과정 중에 자기 자신을 호출하게 됨

직접 재귀

- 자기 자신의 메서드 내부에서 자기 자신의 메서드 호출

간접 재귀

- 자기 자신의 메서드 내부에서 호출이 아닌 다른 메서드를 통해 자기 자신의 메서드 호출

▼ 재귀적 사고

1. 재귀 함수의 입력값, 출력값 정의

: 문제를 가장 추상적 또는 가장 단순하게 정리

2. 문제를 쪼개고 경우의 수를 나누기

: 입력값이나 문제의 순서와 크기를 고려

3. 단순한 문제 해결하기

: 문제를 더이상 쪼갤 수 없을 때까지 쪼개고 가장 해결하기 쉬운 문제부터 해결. (= base case)

* base case : 더 이상 문제를 쪼갤 필요가 없는 종료된 경우.

4. 복잡한 문제 해결하고 코드 구현

: 남은 문제를 해결. 이를 recursive Case라고 함.

* recursive case : 문제를 쪼개서 풀어가는 경우

```
function factorial(num) {
  if(num === 1) { // 더이상 쪼갤 수 없는 경우(base case)
    return 1
  }
  return num * factorial(num - 1) // 재귀 호출(recursive case)
}
```

▼ 재귀와 반복의 차이

재귀

- 기본 경우에 도달하면 종료한다.
- 각 재귀 호출은 스택 프레임(메모리)에 부가 공간을 필요로 한다.
- 무한 재귀에 들어가게 되면 메모리 용량을 초과해서 스택 오버플로우를 초래하게 된다.
- 어떤 문제의 해답은 재귀적인 수식으로 만들기 쉽다.

반복

- 조건이 거짓일 때 종료한다.
- 각 반복이 부가 공간을 필요로 하지 않는다.
- 무한 루프는 추가 메모리가 필요로 하지 않으므로 무한히 반복된다.
- 반복적 해법은 재귀적 해법에 비해 간단하지 않을 때가 있다.

▼ 재귀 심화

1. **꼬리 재귀 (tail recursion in js)**
2. **하노이의 탑 재귀 (js tower of hanoi in recursion)**
3. **조합 재귀 함수 (js combination in recursion)**