## recursion

recursion (순환, 재귀) : 어떤 알고리즘이나 함수가 자기 자신을 호출하여 문제를 해결하는 기법

### 순환의 예

팩토리얼

$$n! = \begin{cases} 1 & n=0 \\ n * (n-1)! & n \geq 1 \end{cases}$$

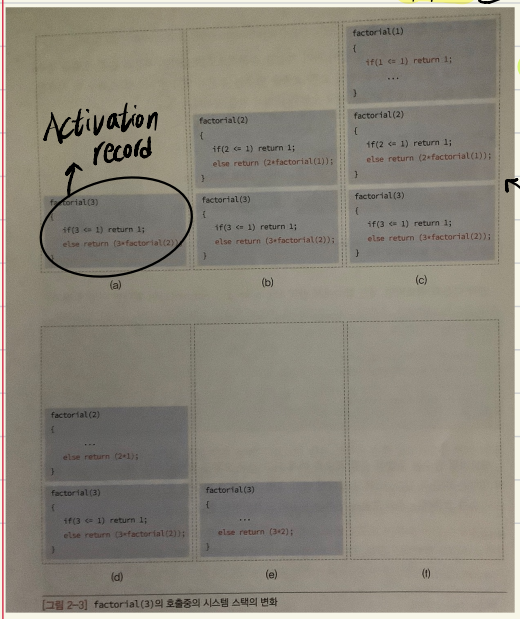
팩토리얼을 정의하는데 다시 팩토리얼을 사용

위와 같은 성의를 '순환적 성의'라고 한다.

### 순환 호출의 내부적인 구현

함수가 자기자신을 호출하는 것 == 다른 함수를 호출하는 것

== 복귀주소를 시스템 스택에 저장 + 매개변수, 지역변수 할당



[그림 2-3] factorial(3)의 호출중의 시스템 스택의 변화

**Activation record**

↳ 함수의 복귀 주소, 매개변수, 지역 변수 등의 정보를 기록하는 시스템 스택의 공간

함수가 호출될 때마다, Activation record 등장

## 순환호출의 조건

C, JAVA 등의 현대적인 프로그래밍 언어에서는 순환을 지원하지만,
FORTRAN, COBOL 등의 고전 언어에서는 지역변수가 정적으로 할당되기 때문에 지원 안함

⇒ 함수 호출마다 새로운 지역변수를 만들 수 있어야 순환 가능

## 순환 알고리즘의 구조

순환 알고리즘 = 순환을 멈추는 부분 + 순환 호출을 하는 부분

↳ if, 없다면?
⇒ 무한히 실행되다 에러

## 순환과 반복

반복(iteration)이란 for와 while 등의 반복구조로 되돌이 하는 방법이다.

↳ 일반적으로 순환보다 빠르다.
But! 순환이 반드시 필요한 경우도 존재한다.

ex)

| recursion | iteration |
|---|---|
| int factorial_re(int n){<br>if(n == 1) return 1;<br>else return n *<br>factorial_re(n-1);<br>} | int factorial_iter(int n) {<br>int i, result = 1;<br>for(i=0; i<n; i++) {<br>result *= i;<br>}<br>return result<br>} |

시간복잡도는 O(n)으로 같다.
But! 순환 알고리즘은 '함수'를 호출하고, 저장 공간을 더 사용하므로 비효율적인 경우가 많다.

## 순환의 원리

문제의 일부를 해결한 다음, 나머지 문제에 대하여 순환호출을 한다.

```
ex) int factorial_re(int n){
        if(n == 1) return 1;
        else return n * factorial_re(n-1)
    }
```

$n$ → 문제의 일부 해결

$factorial_re(n-1)$ → 나머지 문제

## 분할정복

↳ divide and conquer

주어진 문제를 더 작은 동일한 문제들로 분해하여 해결하는 방법

## 반복보다 빠른 순환

### 1. 거듭제곱 계산

반복을 사용한 경우 시간복잡도 : $O(n)$

순환을 사용한 경우

```
power(x, n) :
        if n == 0
            then return 1
        else if n이 짝수
            then return power(x², n/2)
        else if n이 홀수
            then return x * power(x², (n-1)/2)
```

문제의 크기가 $n$, $\frac{n}{2}$, $\frac{n}{4}$ … 순으로 줄어든다!

$$x^n = \begin{cases} 1 & n=0 \\ x^{2\times\frac{n}{2}} & n\text{이 짝수} \\ x * x^{2\times\frac{n-1}{2}} & n\text{이 홀수} \end{cases}$$

$n = 2^k$라 가정하자.
순환호출을 할 때마다, $n$의 크기가 절반이 되므로

$2^k, 2^{k-1}, \cdots, 2, 1$ 순으로 $n$의 값이 작아진다.

∴ 총 $k$번의 실행을 한다.
$k = \log_2 n$ 이므로 시간복잡도는 $O(\log_2 n)$ 이다.

## 2. 하노이 탑 문제

↳ 조건은 책 57p 참고

문제의 크기 $n$ : 이동하여야 하는 디스크의 개수

↳ 어떻게 작아지도록 할 것인가?

⇒ 1. $n-1$ 개의 원판을 temp로 옮김
2. 마지막 원판을 T0로 옮김
3. $n-1$ 개의 원판을 T0로 옮김

```
void hanoi_tower(int n, char from, char tmp, char to)
{
    if(n == 1) printf("%c to %c", from, to);
    else {
        hanoi_tower(n-1, from, to, tmp);    // n-1개를 tmp로 이동
        printf("%c to %c", from, to);
        hanoi_tower(n-1, tmp, from, to);    // n-1개를 tmp에서 to로 이동
    }
}
```

피보나치 수열

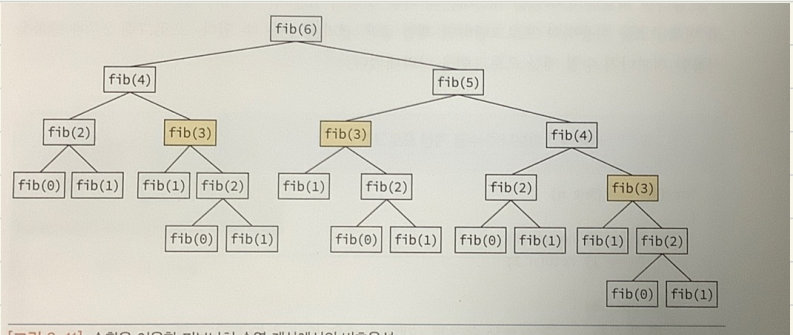
$$fib(n) \begin{cases} 0 & n=0 \\ 1 & n=1 \\ fib(n-1) + fib(n-2) & \text{otherwise} \end{cases}$$

↳ 함수가 순환적으로 정의되어있다!

⇓ 순환 알고리즘 사용

```
int fib(int n)
{
    if(n==0) return 0;
    if(n==1) return 1;
    return (fib(n-1) + fib(n-2));
}
```

But! 매우 비효율적이다.




[그림 2-11] 순환을 이용한 피보나치 수열 계산에서의 비효율성

Let $T(n)$ = 피보나치 수열에서 입력값 $n$에 따른 시간복잡도

then $T(0)=0$, $T(1)=1$, $T(n-2) < T(n-1)$ 이므로
$n \geq 2$ 일때 다음이 성립한다.

$$T(n) = T(n-1) + T(n-2) > 2T(n-2) > 2^2T(n-4) > \cdots > 2^{\frac{n}{2}} \times T(0) = 2^{\frac{n}{2}}$$

∴ 시간복잡도는 $O(2^n)$이다.

⇓

순환 보다는 반복을 쓰는 것이 좋다.

반복으로 바꾸기 어려운 순환

① return $n$ * factorial($n$-1) ⇒ 꼬리 순환
② return factorial($n$-1) * $n$ ⇒ 머리 순환

머리순환(head recursion)이나 다중 순환(multi recursion : 여러군데서 순환호출) 같은 경우, 순환을 반복으로 바꾸기 어렵다.