

## 2장: 순환

---

- 목차
  - 1. 순환의 소개
  - 2. 거듭 제곱 값 계산
  - 3. 피보나치 수열의 계산
  - 4. 하노이 탑 문제

# 1. 순환(recursion)이란?

- 알고리즘이나 함수가 수행 도중에 자기 자신을 다시 호출하여 문제를 해결하는 기법
- 정의자체가 순환적으로 되어 있는 경우에 적합한 방법
- (예제)

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

피보나치 수열

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

이항계수

$${}_nC_k = \begin{cases} 1 & n=0 \text{ or } n=k \\ {}_{n-1}C_{k-1} + {}_{n-1}C_k & \text{otherwise} \end{cases}$$

하노이의 탑

이진탐색



# 팩토리얼 프로그래밍 #1

- 팩토리얼의 정의

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

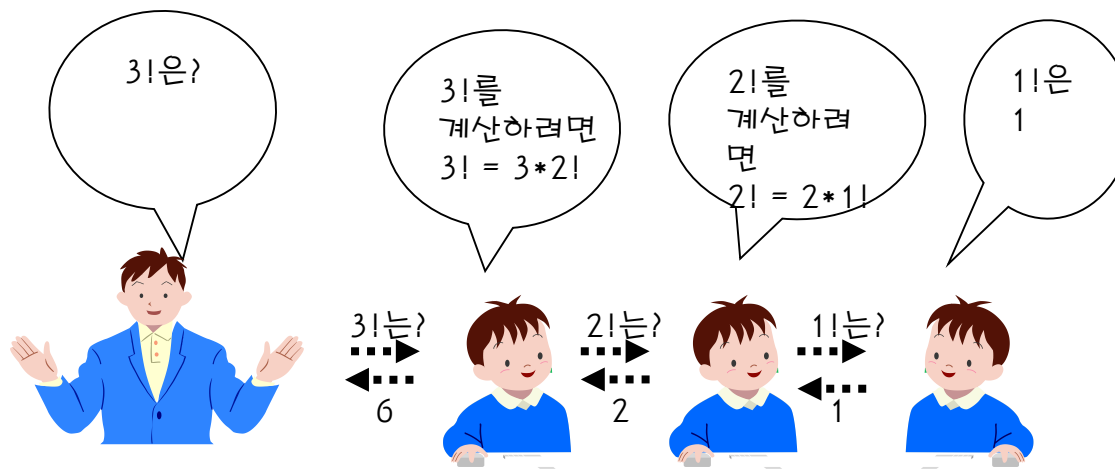
- 팩토리얼 프로그래밍 #1:
  - 위의 정의대로 구현
  - (n-1)! 팩토리얼을 구하는 서브 함수 factorial\_n\_1를 따로 제작

```
int factorial(int n)
{
    if( n <= 1 ) return(1);
    else return (n * factorial_n_1(n-1) );
}
```

# 팩토리얼 프로그래밍 #2

- 팩토리얼 프로그래밍 #2:
  - (n-1)! 팩토리얼을 현재 작성중인 함수를 다시 호출하여 계산(순환 호출)

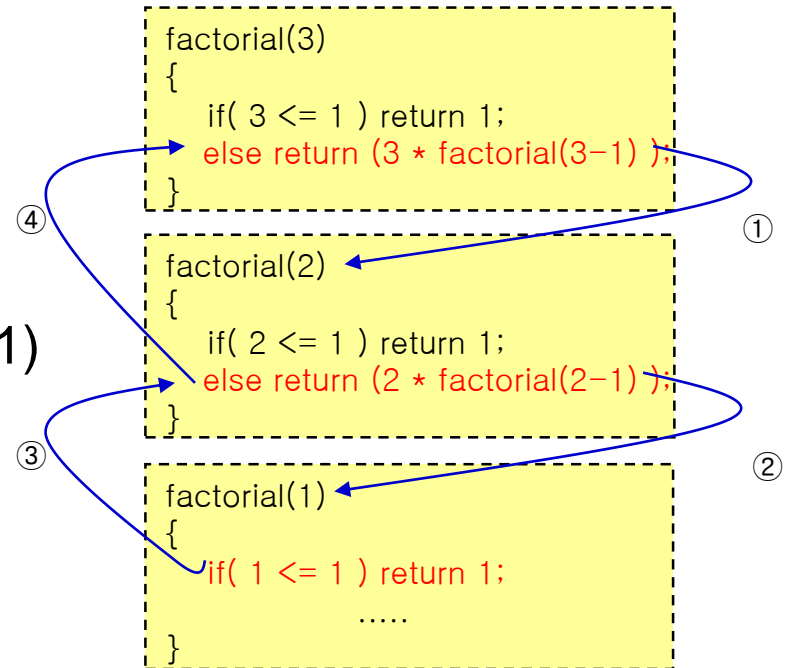
```
int factorial(int n)
{
    if( n <= 1 ) return(1);
    else return (n * factorial(n-1) );
}
```



# 순환호출순서

- 팩토리얼 함수의 호출 순서

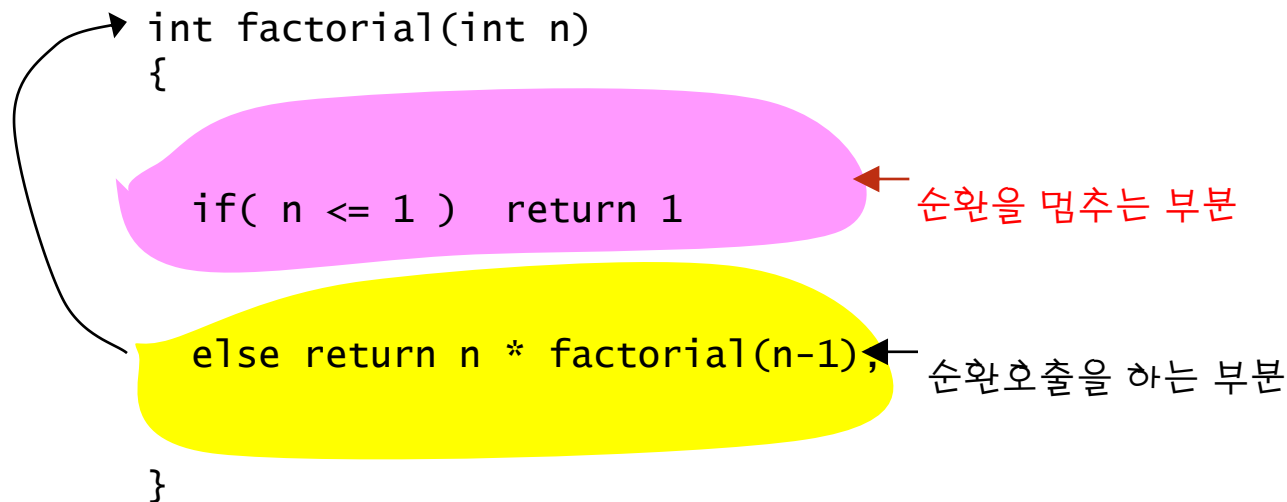
$\text{factorial}(5) = 5 * \text{factorial}(4)$   
 $= 5 * 4 * \text{factorial}(3)$   
 $= 5 * 4 * 3 * \text{factorial}(2)$   
 $= 5 * 4 * 3 * 2 * \text{factorial}(1)$   
 $= 5 * 4 * 3 * 2 * 1$   
 $= 120$



# 순환 알고리즘의 구조

- 순환 알고리즘은 다음과 같은 부분들을 포함한다.

- 순환 호출을 하는 부분
- 순환 호출을 멈추는 부분



- 만약 순환 호출을 멈추는 부분이 없다면?
- 시스템 오류가 발생할 때까지 무한정 호출하게 된다.

# 순환 <-> 반복

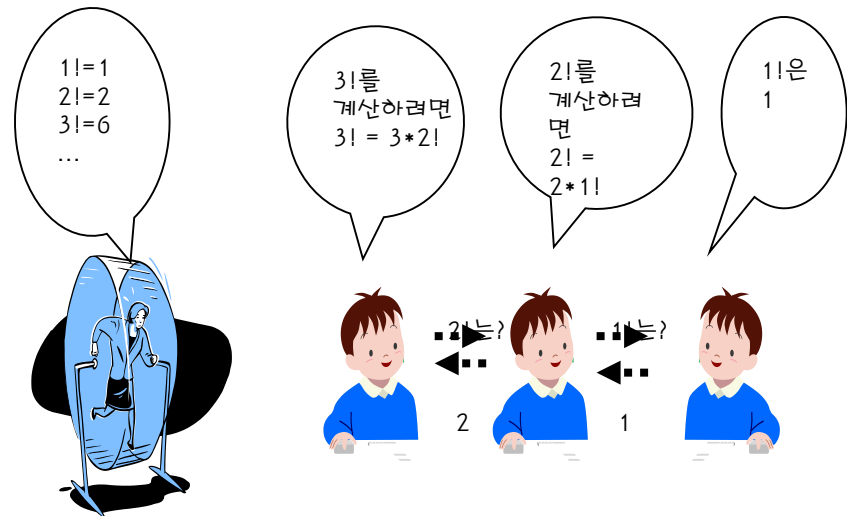
- 컴퓨터에서의 되풀이
  - 순환(recursion): 순환 호출 이용
  - 반복(iteration): for나 while을 이용한 반복
- 대부분의 순환은 반복으로 바꾸어 작성할 수 있다.

- 순환

- 순환적인 문제에서는 자연스러운 방법
- 함수 호출의 오버헤드

- 반복

- 수행속도가 빠름
- 순환적인 문제에 대해서는 프로그램 작성이 아주 어려울 수도 있음



# 팩토리얼의 반복적 구현

---

$$n! = \begin{cases} 1 & n = 1 \\ n * (n-1) * (n-2) * \Lambda * 1 & n \geq 2 \end{cases}$$

```
int factorial_iter(int n)
{
    int k, v=1;
    for(k=n; k>0; k--)
        v = v*k;
    return(v);
}
```



## 2. 거듭제곱값 프로그래밍

---

- 순환적인 방법이 반복적인 방법보다 더 효율적인 예
- 숫자  $x$ 의  $n$ 제곱값을 구하는 문제:  $x^n$
- 반복적인 방법

```
double slow_power(double x, int n)
{
    int i;
    double r = 1.0;
    for(i=0; i<n; i++)
        r = r * x;
    return(r);
}
```

# 거듭제곱값 프로그래밍 #2

- 순환적인 방법

```
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);
```



```
double power(double x, int n)
{
    if( n==0 ) return 1;
    else if ( (n%2)==0 )
        return power(x*x, n/2);
    else return x*power(x*x, (n-1)/2);
}
```

즉  $n$ 이 짝수이면 다음과 같이 계산하는 것이다.

$$\begin{aligned} \text{power}(x, n) &= \text{power}(x^2, n / 2) \\ &= (x^2)^{n/2} \\ &= x^{2(n/2)} \\ &= x^n \end{aligned}$$

만약  $n$ 이 홀수이면 다음과 같이 계산하는 것이다.

$$\begin{aligned} \text{power}(x, n) &= x \cdot \text{power}(x^2, (n-1) / 2) \\ &= x \cdot (x^2)^{(n-1)/2} \\ &= x \cdot x^{n-1} \\ &= x^n \end{aligned}$$

# 거듭제곱값 프로그래밍 분석

- 순환적인 방법의 시간 복잡도
  - $n$ 이 100이면,  $100 \rightarrow 50 \rightarrow 25 \rightarrow 12 \rightarrow 6 \rightarrow 3 \rightarrow 1$  로 크기가 줄어든다.
  - 만약  $n$ 이 2의 제곱이라고 가정하면 다음과 같이 문제의 크기가 줄어든다.

$$2^n \rightarrow 2^{n-1} \rightarrow \dots \rightarrow 2^2 \rightarrow 2^1 \rightarrow 2^0$$

- 반복적인 방법과 순환적인 방법의 비교

	반복적인 함수 slow_power	순환적인 함수 power
시간복잡도	$O(n)$	$O(\log_2 n)$
실제수행속도	7.17초	0.47초

### 3. 피보나치 수열의 계산

- 순환 호출을 사용하면 비효율적인 예
- 피보나치 수열

0, 1, 1, 2, 3, 5, 8, 13, 21, ...



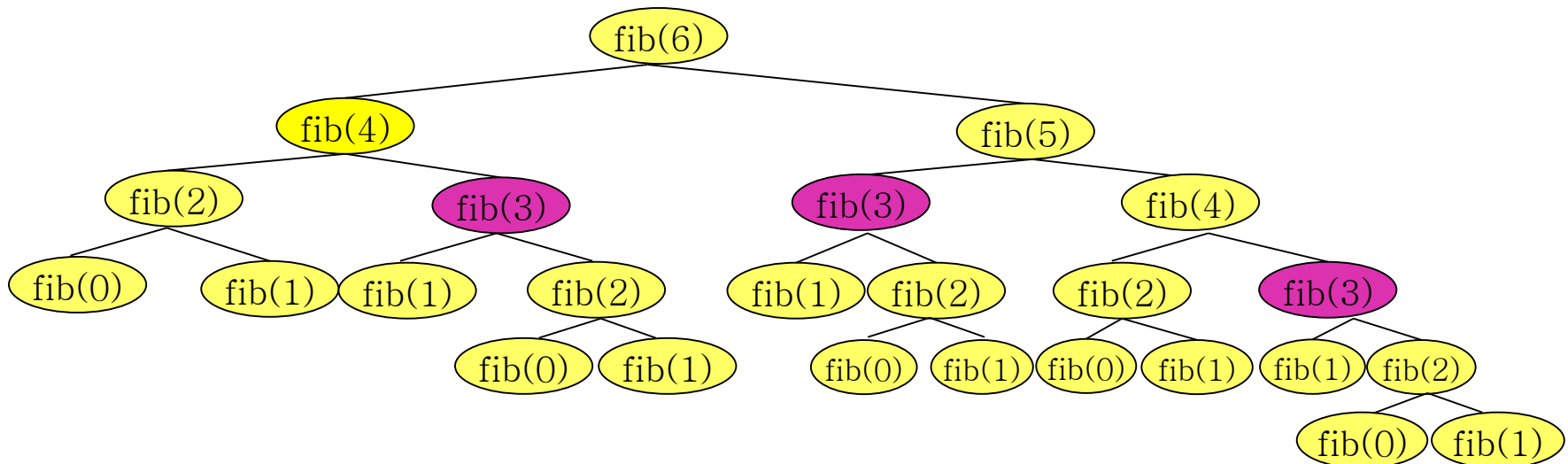
$$fib(n) = \begin{cases} 0 & n = 0 \\ 1 & n = 1 \\ fib(n-2) + fib(n-1) & otherwise \end{cases}$$

- 순환적인 구현

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

# 피보나치 수열의 계산 #2

- 순환 호출을 사용했을 경우의 비효율성
  - 같은 항이 중복해서 계산됨
  - 예를 들어 `fib(6)`을 호출하게 되면 `fib(3)`이 3번이나 중복되어서 계산됨
  - 이러한 현상은 `n`이 커지면 더 심해짐



# 피보나치 수열의 반복구현

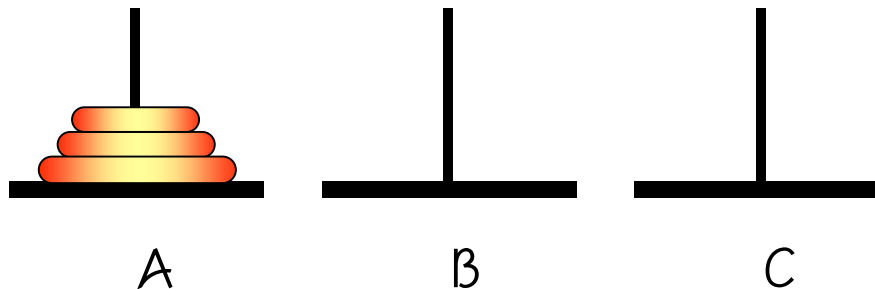
- 반복 구조를 사용한 구현

```
fib_iter(int n)
{
    if( n < 2 ) return n;
    else
    {
        int i, tmp, current=1, last=0;
        for(i=2;i<=n;i++)
        {
            tmp = current;
            current += last;
            last = tmp;
        }
        return current;
    }
}
```

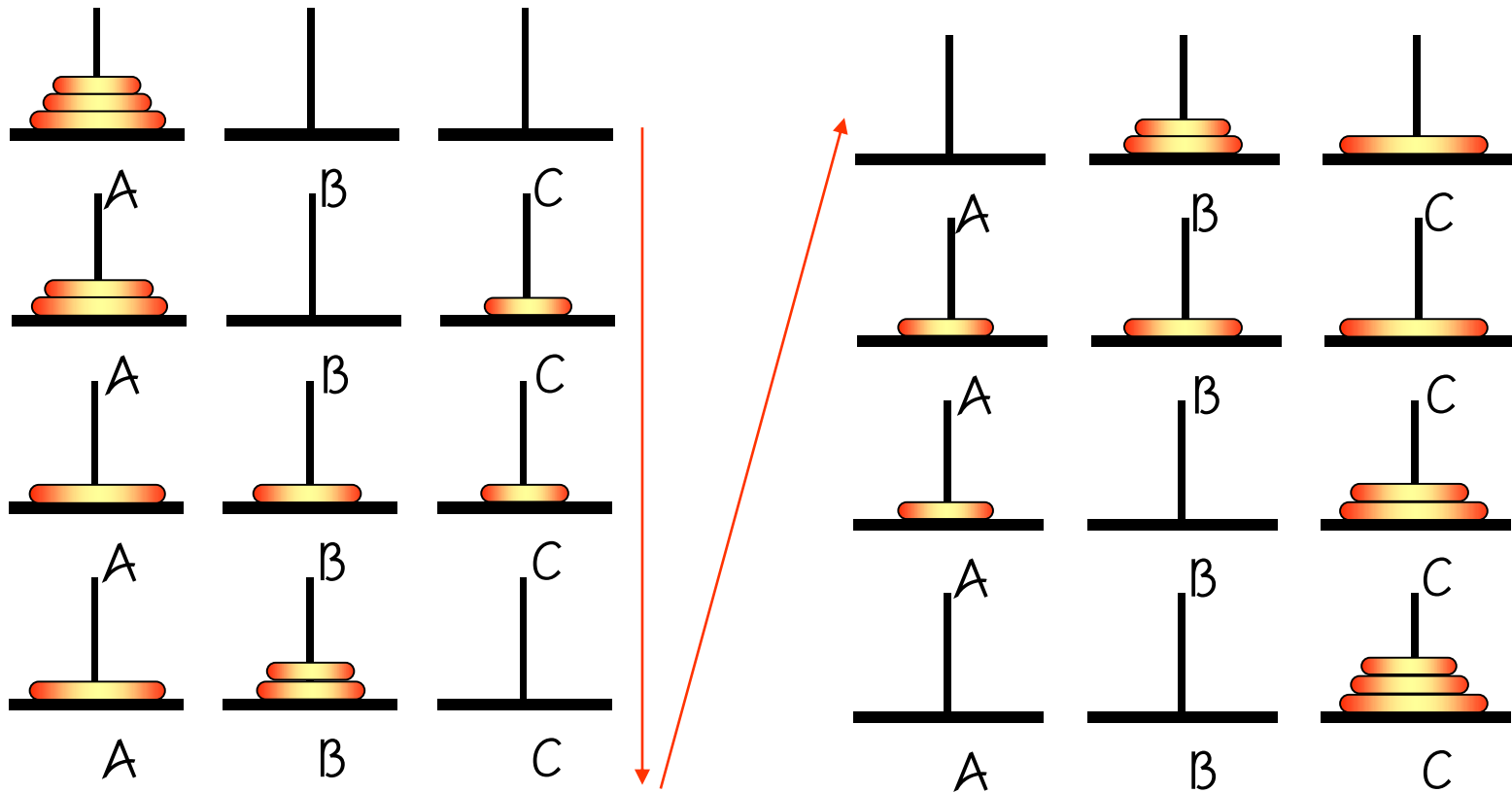


## 4. 하노이 탑 문제

- 문제는 막대 A에 쌓여있는 원판  $n$ 개를 막대 C로 옮기는 것이다. 단 다음의 조건을 지켜야 한다.
  - 한 번에 하나의 원판만 이동할 수 있다
  - 맨 위에 있는 원판만 이동할 수 있다
  - 크기가 작은 원판위에 큰 원판이 쌓일 수 없다.
  - 중간의 막대를 임시적으로 이용할 수 있으나 앞의 조건들을 지켜야 한다.

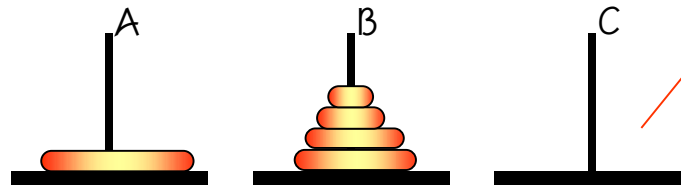
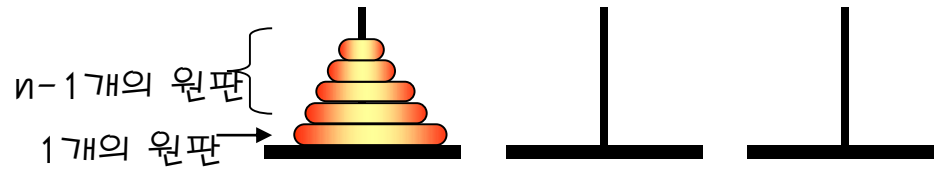


# n=3인 경우의 해답

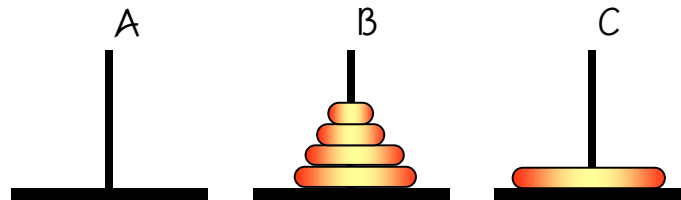




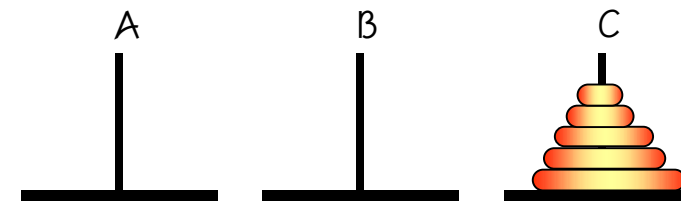
# 일반적인 경우에는?



C를 임시버퍼로 사용하여 A에 쌓여있는  $n-1$ 개의 원판을 B로 옮긴다.



A의 가장 큰 원판을 C로 옮긴다.



A를 임시버퍼로 사용하여 B에 쌓여있는  $n-1$ 개의 원판을 C로 옮긴다.

A B C

# 남아있는 문제는?

- 자 그러면 어떻게  $n-1$ 개의 원판을 A에서 B로, 또 B에서 C로 이동하는가?
- (힌트) 우리의 원래 문제가  $n$ 개의 원판을 A에서 C로 옮기는 것임을 기억하라.
- 따라서 지금 작성하고 있는 함수를 파라미터를  $n-1$ 로 바꾸어 순환호출하면 된다.

```
// 막대 from에 쌓여있는 n개의 원판을 막대 tmp를 사용하여 막대 to
// 옮긴다.
void hanoi_tower(int n, char from, char tmp, char to)
{
    if (n==1)
    {
        from에서 to로 원판을 옮긴다.
    }
    else
    {
        hanoi_tower(n-1, from, to, tmp);
        from에 있는 한 개의 원판을 to로 옮긴다.
        hanoi_tower(n-1, tmp, from, to);
    }
}
```

# 하노이탑 최종 프로그램

- $n-1$ 개의 원판을 A에서 B로 옮기고  $n$ 번째 원판을 A에서 C로 옮긴 다음,  $n-1$ 개의 원판을 B에서 C로 옮기면 된다.

```
#include <stdio.h>
void hanoi_tower(int n, char from, char tmp, char to)
{
    if( n==1 ) printf("원판 1을 %c 에서 %c으로 옮긴다.\n",from,to);
    else
    {
        hanoi_tower(n-1, from, to, tmp);
        printf("원판 %d을 %c에서 %c으로 옮긴다.\n",n, from, to);
        hanoi_tower(n-1, tmp, from, to);
    }
}
main()
{
    hanoi_tower(4, 'A', 'B', 'C');
}
```