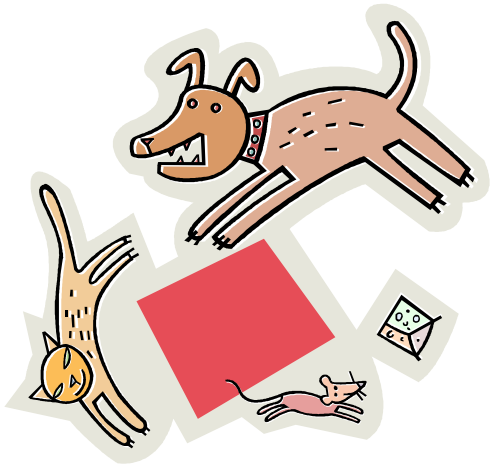
2장 순환 (RECURSION)

순환(recursion)이란?

- 알고리즘이나 함수가 수행 도중에 자기 자신(함수)을 다시 호출하여 문제를 해결하는 기법

- 정의 자체가 순환적으로 되어 있는 경우에 적합한 방법



팩토리얼 프로그래밍 #1

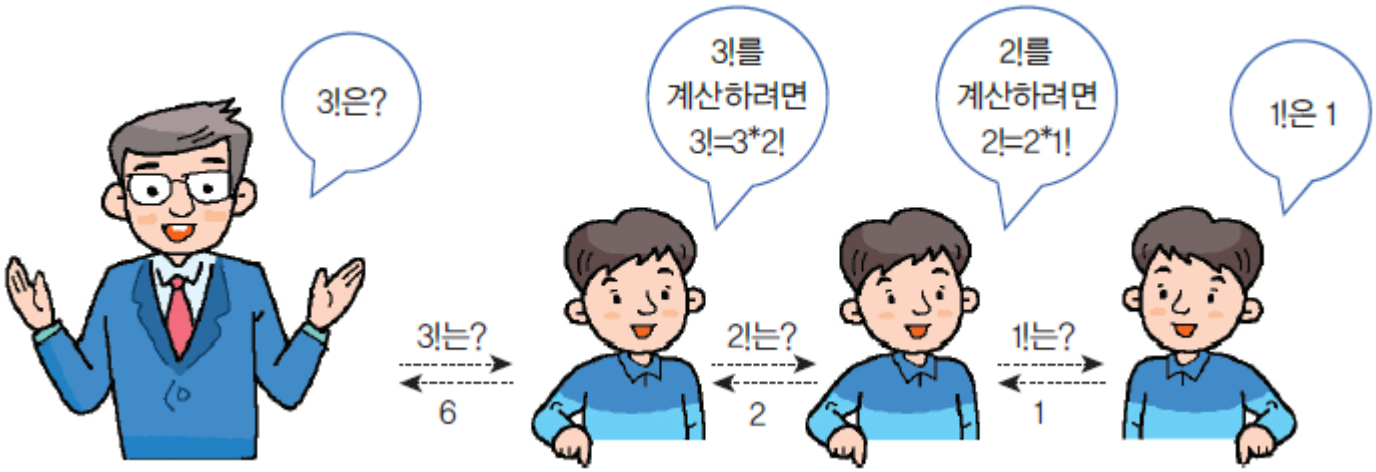
- 팩토리얼의 정의

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

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

팩토리얼 프로그래밍 #2

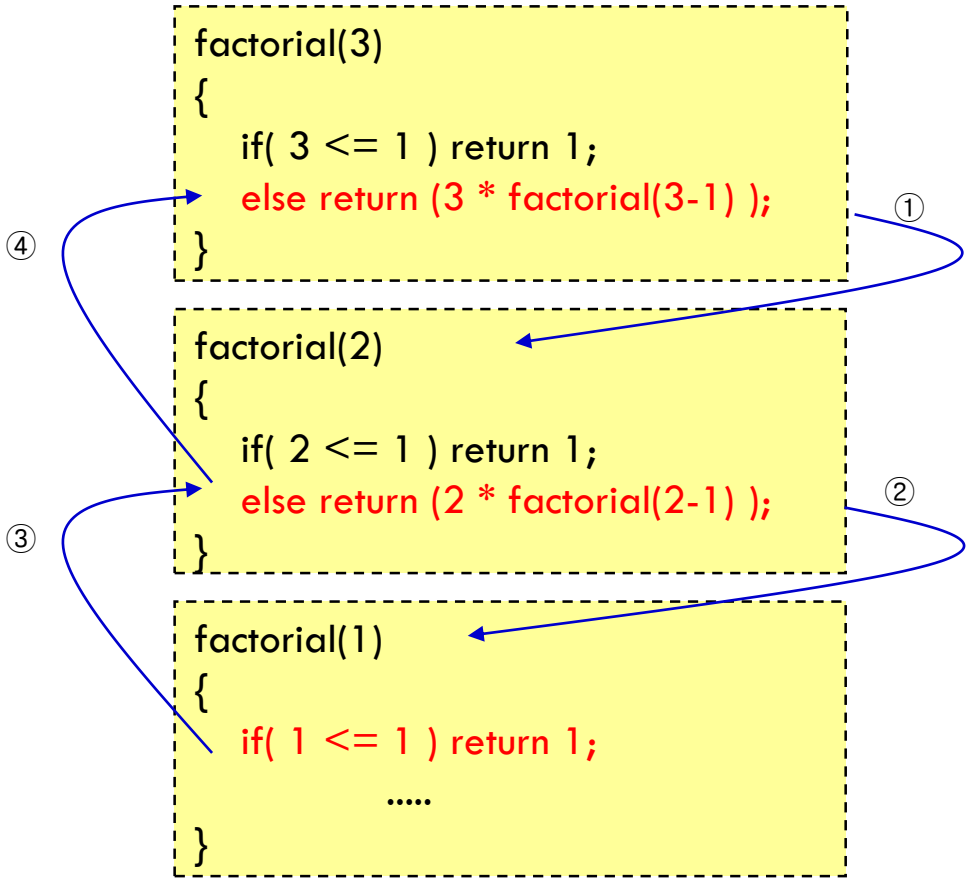

```
int factorial(int n)
{
    if( n <= 1 ) return(1);
    else return (n * factorial(n-1) );
}
```



순환호출순서

- 팩토리얼 함수의 호출 순서

factorial(3) = 3 * factorial(2)
= 3 * 2 * factorial(1)
= 3 * 2 * 1
= 6



순환 알고리즘의 구조


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

순환을 멈추는 부분

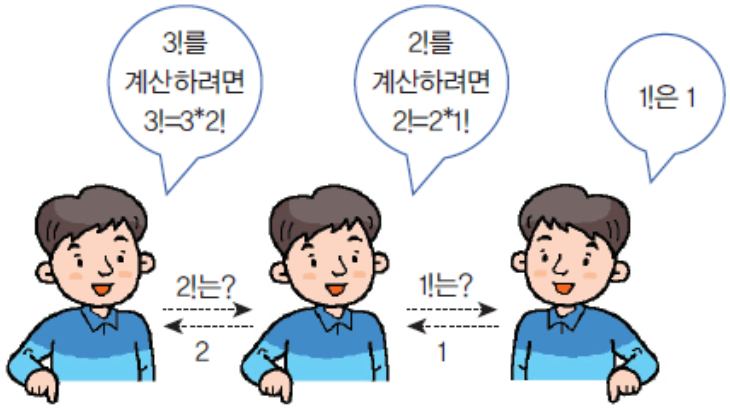
순환 호출을 하는 부분

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

순환 <-> 반복

- 대부분의 순환은 반복으로 바꾸어 작성할 수 있다.

(a) 반복



(b) 순환

팩토리얼의 반복적 구현

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

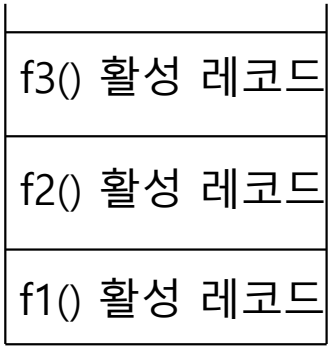
```
int factorial_iter(int n)
{
      int k, v=1;
      for(k=n; k>0; k--)
          v = v*k;
      return(v);
}
```

순환과 반복 비교

- 일반적으로 순환 프로그래밍 <-> 반복 프로그래밍 변환이 가능함
- 문제 정의가 순환적인 경우
  - 순환적 프로그래밍이 더 쉽고, 이해하기 편함
  - 반복문 작성 시에 훨씬 코드도 길어지고 이해하기도 힘듦
- 프로그램 효율적인 측면에서는 반복문이 유리
  - 순환 프로그래밍
    - 함수 호출의 부담: 활성 레코드 생성/보존(다음 페이지 참조), 실행 코드 점프 등
  - 둘다 가능하고, 코드 길이가 더 커지지 않는다면 반복문으로 프로그래밍 하는 것을 선호

활성 레코드(Activation Record)

- 함수 호출 순서: f1() -> f(2) -> f3()



- 호출되는 함수의 각 활성 레코드가 스택 형태로 저장
- 활성 레코드 항목
  - Return Address: 자신을 호출한 함수의 복귀 주소
  - 레지스터 값들: 호출된 함수가 실행되기 전에 보존해야 할 CPU 레지스터 값들
  - 함수 매개변수: 해당 함수 호출 시에 전달된 인수(파라미터)
  - 지역 변수: 함수 내에 사용되는 지역 변수
  - …

거듭제곱 값 프로그래밍

- 순환적인 방법이 더 효율적인 예제
- 숫자 x의 n제곱 값을 구하는 문제: x^n

- 반복적인 방법

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

순환 프로그래밍 작성 방법

- 순환 프로그래밍: 선언적(Declarative) 프로그래밍
- 선언적(Declarative) 프로그래밍
  - 프로그래밍의 목표를 명시. 구체적인 알고리즘을 명시하지 않음
  - 명령형 프로그래밍에 익숙한 개발자는 이해가 어렵다는 단점이 있음
- 명령형(Imperative) 프로그래밍
  - 프로그래밍의 목표보다 구체적인 알고리즘을 명시

- 순환 프로그래밍: 정의를 이해하고, 이를 재귀적으로 분석
  - 종료 조건
  - 문제의 정의 (n과 같은 차수는 점차적으로 낮아져야 함)

거듭제곱(x^n) 프로그래밍: 순환 방식 1

- 알고리즘 $x^n = x * x^{n-1}$ 정의를 이용

```
power(x, n)

if n==0
        then return 1;
else
        then return x*power(x, n-1);
```

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

거듭제곱(x^n) 프로그래밍: 순환 방식 2

- 알고리즘 $x^n = (x^2)^{n/2}$ 정의를 이용

```
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);
}
```

거듭제곱 값 프로그래밍 분석

- 순환 방식2의 시간 복잡도
  - 만약 n = 2^k 로 가정, 다음과 같이 문제의 크기가 줄어든다.

$$2^k \rightarrow 2^{k-1} \rightarrow 2^{k-2} \rightarrow \ldots \rightarrow 2^1 \rightarrow 2^0$$

예) 100 -> 50 -> 25 -> 12 -> 6 -> 3 -> 1

- 반복적인 방법과 순환적인 방법의 비교

	반복적인 함수 slow_power	순환적인 함수 power
시간복잡도	O(n)	O(logn)
실제수행속도	7.17초	0.47초

(2^{500}을 1,000,000번 계산 시 수행속도)

피보나치 수열의 계산

- 순환 호출을 사용하면 비효율적인 예
- 피보나치 수열

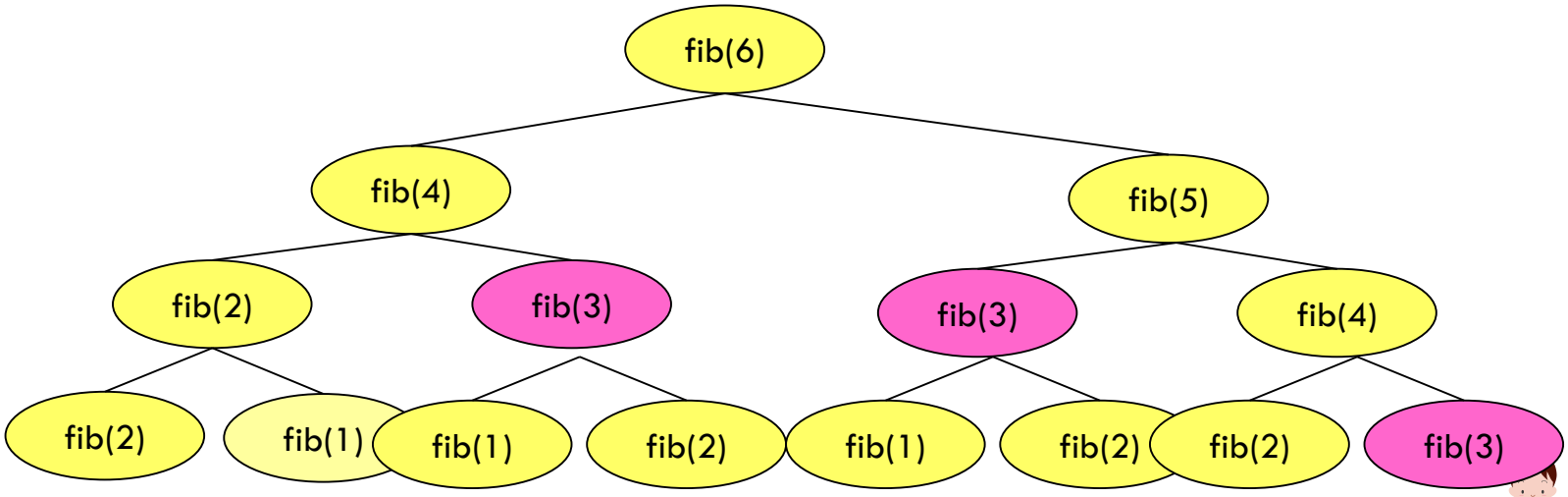
  0,1,1,2,3,5,8,13,21,…

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

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

피보나치 순환 방식 알고리즘의 비효율성

- 같은 항이 중복해서 계산됨
  - 예를 들어 fib(6)을 호출하게 되면 fib(3)이 3번이나 중복되어서 계산됨
- 이러한 현상은 n이 커지면 더 심해짐
  - fib(6)은 fib() 함수 25번 호출, fib(30)은 300 만번 함수 호출



피보나치 수열 순환 알고리즘 복잡도

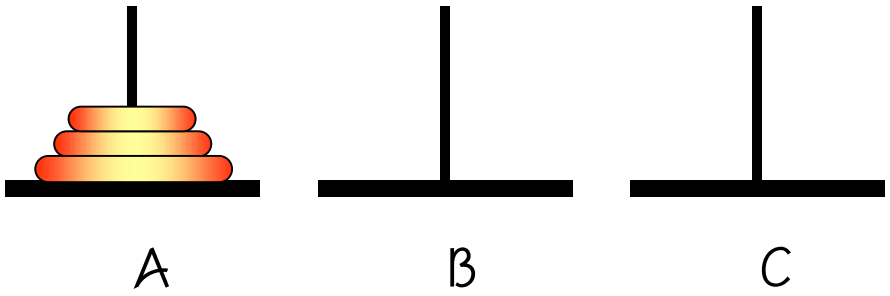
- 복잡도 함수
  - $T(n) = T(n-1) + T(n-2) + O(1)$
- 시간 복잡도: $O(2^n)$
  - 하나의 피보나치 함수 fib()는 매 스텝 2개의 fib() 함수를 호출
  - 1 -> 2 -> 4 -> 8 -> 16 -> 32 -> 64 -> …

피보나치 수열의 반복 구현

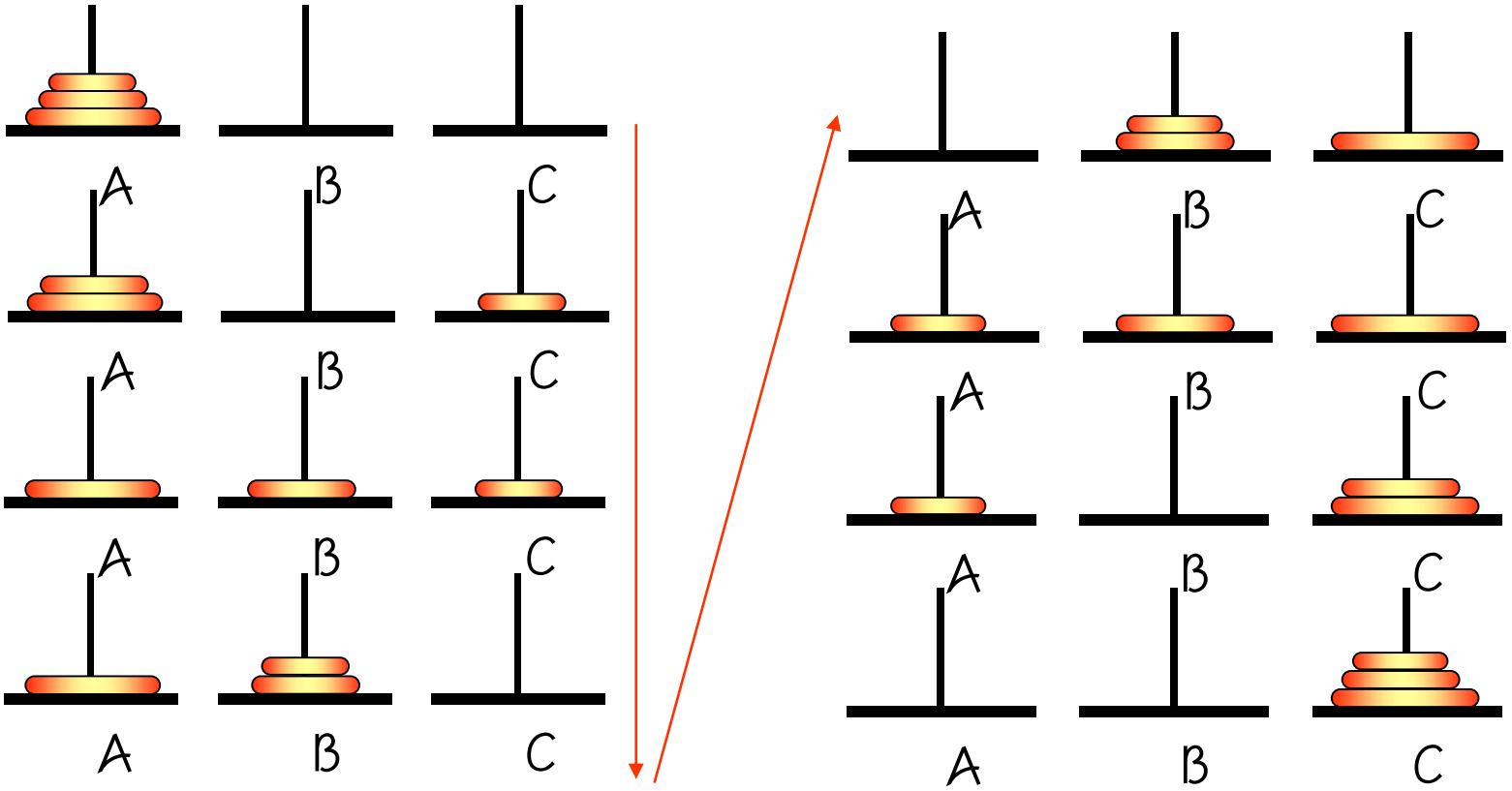

```
int fib_iter(int n)
{
    int pp = 0;
    int p = 1;
    int result = 0;

    if (n == 0) return 0;
    if (n == 1) return 1;

    for (int i = 2; i <= n; i++) {
        result = p + pp;
        pp = p;
        p = result;
    }
    return result;
}
```

하노이 탑 문제 (순환의 대표적 알고리즘)

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

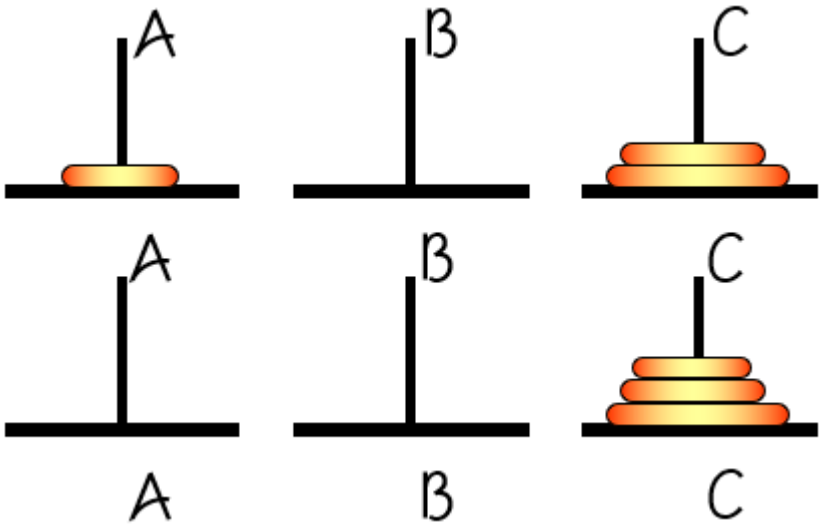


n=3인 경우의 해답

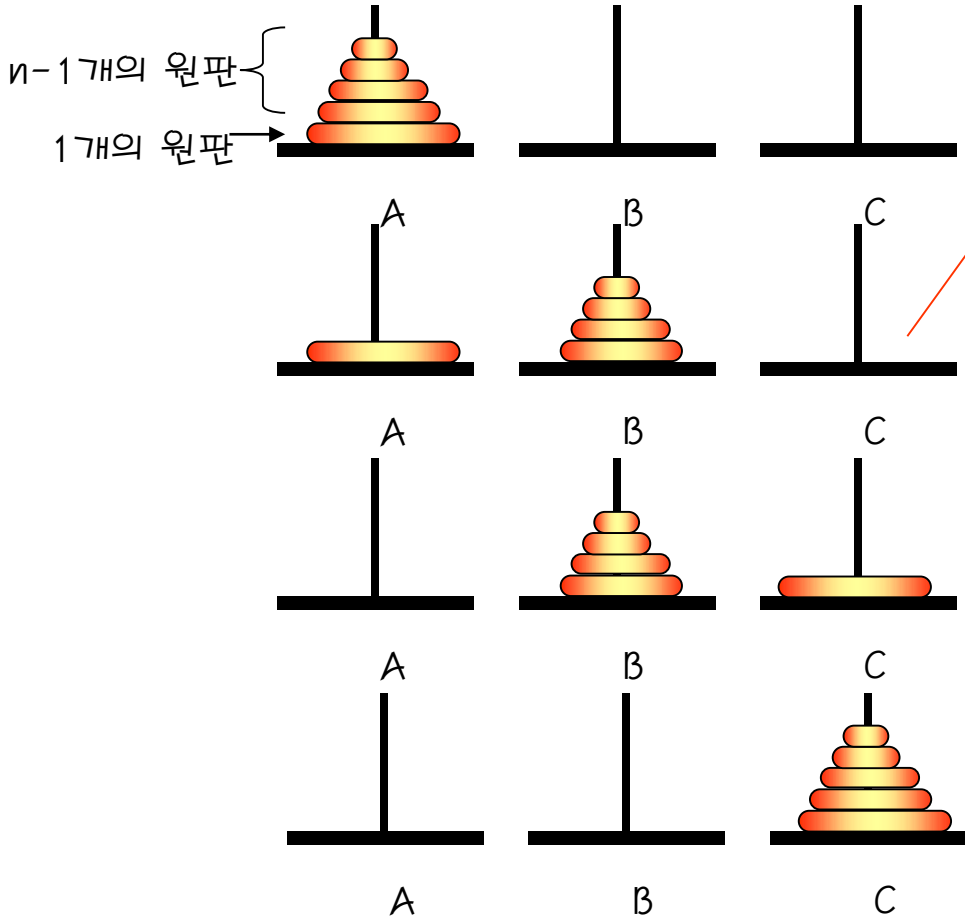


종료 조건

- n == 1인 경우
  - 마지막 원반(1)을 A에서 C로 이동함



문제의 정의



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

A의 가장 큰 원판을 C로 옮긴다.

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

하노이탑 알고리즘


```
hanoi (n, From, Tmp, To)  // hanoi(4, A, B, C)

// 종료 조건 명시
if n == 1 then 마지막 원판을 from에서 to로 이동한다.

// 문제의 정의
else then
        hanoi(n-1, From, To, Tmp)
        from에 있는 큰 원판을 To로 이동한다.
        hanoi(n-1, Tmp, From, To)
```

하노이탑 구현 프로그램


```c
#include <stdio.h>

void hanoi_tower(int n, char from, char tmp, char to)
{
    if (n == 1)
    {
        printf("원판 1을 %c에서 %c로 이동함\n", from, to);
    }
    else
    {
        hanoi_tower(n-1, from, to, tmp);
        printf("원판 %d을 %c에서 %c로 이동함\n", n, from, to);
        hanoi_tower(n-1, tmp, from, to);
    }
    return;
}

int main(void)
{
    int n;

    printf("탑 높이 n 입력 : ");
    scanf("%d", &n); printf("\n\n");
    hanoi_tower(n, 'A', 'B', 'C');
    return 0;
}
```

실행결과


```
탑높이 n 입력: 4


원판 1을 A에서 B로 이동함
원판 2을 A에서 C로 이동함
원판 1을 B에서 C로 이동함
원판 3을 A에서 B로 이동함
원판 1을 C에서 A로 이동함
원판 2을 C에서 B로 이동함
원판 1을 A에서 B로 이동함
원판 4을 A에서 C로 이동함
원판 1을 B에서 C로 이동함
원판 2을 B에서 A로 이동함
원판 1을 C에서 A로 이동함
원판 3을 B에서 C로 이동함
원판 1을 A에서 B로 이동함
원판 2을 A에서 C로 이동함
원판 1을 B에서 C로 이동함

--------------------------------
Process exited after 3.663 seconds with return value 0
계속하려면 아무 키나 누르십시오 . . .
```

하노이탑 알고리즘 총평

- 하노이 순환 알고리즘은 비교적 난해한 알고리즘
  - 순환 알고리즘 자체가 대표적인 선언적 프로그래밍 방식이므로,
  - 기존의 명령형 프로그래밍에 익숙한 사람에게 이해하기 어려운 방식임
- 프로그래밍 결과를 자세히 따라 가보고, 여러가지 순환 프로그래밍 경험을 가지기 바람
- 순환 알고리즘이 안되면 Divide-&-Conquer, Dynamic Programming과 같은 고수준 알고리즘을 익히기 힘듦

Q & A