제 9 장

함수와 변수

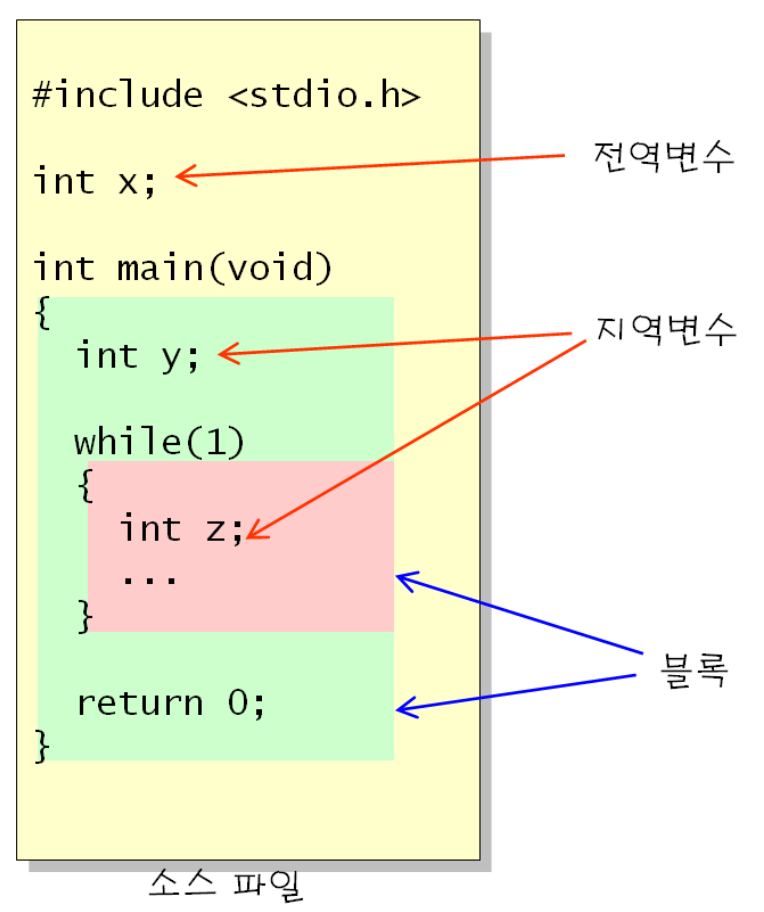
변수의 속성

- 변수는 선언 그 자체만으로도 많은 속성을 나타내고 있으며, 더불어 선언되는 위치에 따라서도 또 다른 속성들을 나타낸다
- 변수의 기본 속성 : 이름, 타입, 바이트 크기, 가질 수 있는 값 범위
- 변수 선언 위치 : 블록 내부 혹은 블록 외부
  - 유효 영역(scope), 생존 시간(life time), 연결(linkage) 및 추상성(abstraction)
- 변수에 붙여주는 저장유형지정자에 따라
  - auto, register
  - static, extern

범위(scope) : 변수가 사용 가능한 범위, 가시성
생존 시간(lifetime): 메모리에 존재하는 시간
연결(linkage): 다른 영역에 있는 변수와의 연결 상태

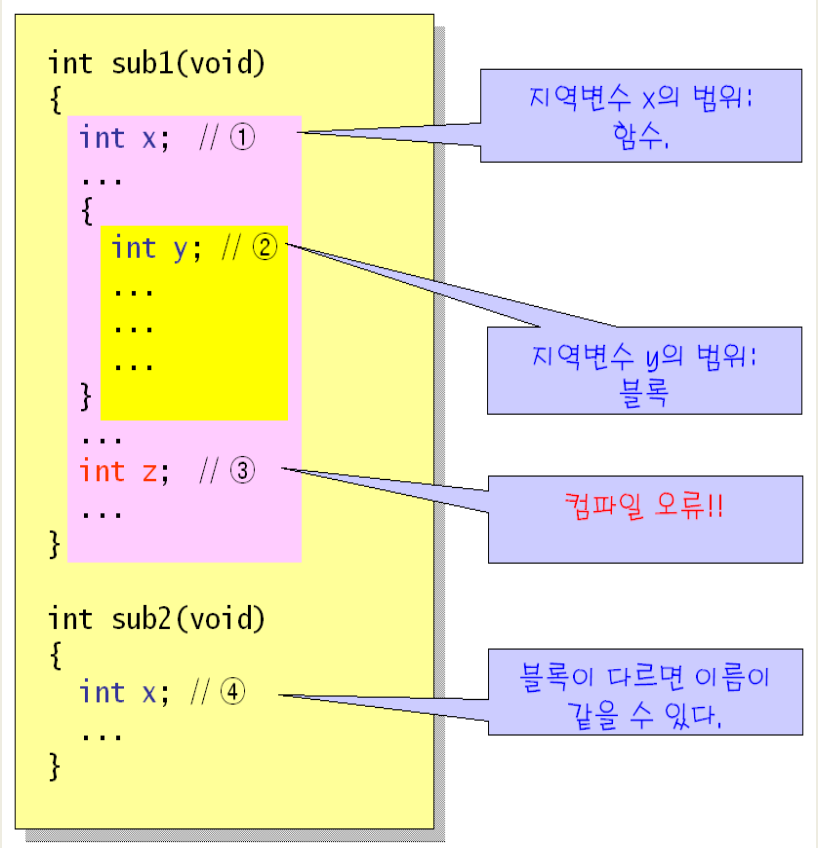
영역(scope)

- scope: 변수가 유효성을 가지고 접근되고 사용되는 영역
- 이는 변수가 선언되는 위치에 의해 결정된다.
- 영역의 종류
  - 파일 (file scope) : 함수의 외부에서 선언, 전역(global) 변수
  - 함수 (function scope)
  - 블록 (block scope): 블록 안에서 선언, 지역(local) 변수
  - 함수 원형 : 함수의 원형에 등장하는 매개 변수에 적용



지역 변수(local variable)

- 지역변수(Local variable) : 선언 위치가 블록의 내부인 변수
- 블록의 도입부에 선언되어야 함
  - 블록이 시작해서 종료하면 자동적으로 사라진다
  - automatic variable
  - 보통 stack에 변수의 저장소가 할당된다
- 유효영역-블록
  - 블록 (block scope): 선언된 블록 내부에서만 접근 가능
- 생존시간
  - 블록의 시작 부터 블록의 종료 시까지
- 선언 지역만 다르면 동일한 이름을 사용할 수 있다



```
int sub1(void)
{
  int x;   // 함수의 시작 부분에 선언
  int y;   // 오류가 아님
  ...
}
```

```
void sub1(void)
{
        int x;              // 지역 변수 x 선언
        x = 0;              // OK
        {
                int y;      // 지역 변수 y 선언
                x = 1;      // OK
                y = 2;      // OK
        }
        x = 3;              // OK
        y = 4;              // ① 컴파일 오류!!
}
```

지역 변수 예제

```
#include <stdio.h>

int main(void)
{
        int i;

        for(i = 0;i < 5; i++)
        {
                int temp = 1;
                printf("temp = %d₩n", temp);
                temp++;
        }
        return 0;
}
```

블록이 시작할 때마다 생성되어 초기화된다.

```
temp = 1
temp = 1
temp = 1
temp = 1
temp = 1
```

함수의 매개 변수

```
#include <stdio.h>
int inc(int counter);

int main(void)
{
        int i;

        i = 10;
        printf("함수 호출전 i=%d₩n", i);
        inc(i);
        printf("함수 호출후 i=%d₩n", i);
        return 0;
}
int inc(int counter)
{
  counter++;
  return counter;
}
```

값에 의한 호출
(call by value)

매개 변수도 일종의
지역 변수임

```
함수 호출전 i=10
함수 호출후 i=10
```

전역 변수

- 전역변수(global variable) : 선언 위치가 함수 블록의 외부인 변수
- 보통 프로그램 파일의 시작 부분에 선언된다
  - 주메모리에 변수의 저장소가 할당된다
- 유효영역-파일
  - 선언된 위치 아래에 존재하는 모든 함수에서 접근 가능
  - 파일 (file scope): 선언된 프로그램 파일 전체에서 접근 가능
- 생존시간
  - 프로그램 파일의 시작 부터 종료 시까지
- 동일한 이름을 사용할 수 없다

```
#include <stdio.h>
int x = 123;

void sub1()
{
        printf("In sub1()  x=%d₩n",x);
}
void sub2()
{
        printf("In sub2()  x=%d₩n",x);
}
int main(void)
{
        sub1();
        sub2();

        return 0;
}
```

```
In sub1()   x=123
In sub2()   x=123
```

전역 변수의 초기값과 생존 기간

```c
#include <stdio.h>
int counter; // 전역 변수

void set_counter(int i)
{
        counter = i;        // 직접 사용 가능
}
int main(void)
{
        printf("counter=%d₩n", counter);

        counter = 100;              // 직접 사용 가능
        printf("counter=%d₩n", counter);

        set_counter(20);
        printf("counter=%d₩n", counter);

        return 0;
}
```

```
counter=0
counter=100
counter=20
```

* 전역 변수의 초기값은 0

*생존 기간은 프로그램 시작부터 종료

전역 변수의 사용 예

```c
// 전역 변수를 사용하여 프로그램이 복잡해지는 경우
#include <stdio.h>
void f(void);

int i;
int main(void)
{
        for(i = 0;i < 5; i++)
        {
                f();
        }
        return 0;
}
void f(void)
{
        for(i = 0;i < 10; i++)
                printf("#");
}
```

##########

같은 이름의 전역 변수와 지역 변수

```
// 동일한 이름의 전역 변수와 지역 변수
#include <stdio.h>

int sum = 1;        // 전역 변수

int main(void)
{
    int i = 0;
    int sum = 0;        // 지역 변수

    for(i = 0; i <= 10; i++)
    {
        sum += i;
    }
    printf("sum = %d\n", sum);

    return 0;
}
```

지역 변수가 전역 변수를 가린다.

```
sum = 55
```

생존 기간(life-time)과 저장유형 지정자

- 정적 할당(static allocation):
  - 프로그램 실행 시간 동안 계속 유지
- 자동 할당(automatic allocation):
  - 블록에 들어갈때 생성
  - 블록에서 나올때 소멸
- 생존 기간을 결정하는 요인
  - 변수가 선언된 위치 : local, global
  - 저장 유형 지정자 : static local, static global
- 저장 유형 지정자
  - auto
  - register
  - static
  - extern

저장 유형 지정자 auto

- 변수를 선언한 위치에서 자동으로 만들어지고 블록을 벗어나게 되며 자동으로 소멸되는 저장 유형을 지정
- 지역 변수는 auto가 생략되어도 자동 변수가 된다.

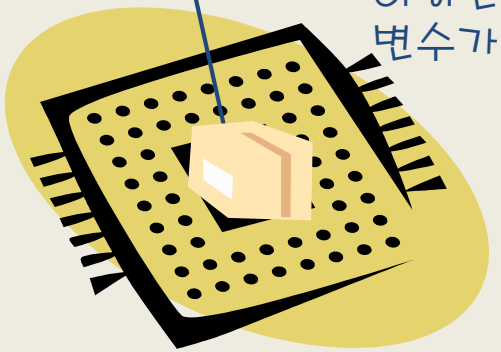
```
int main(void)
{
  auto int sum = 0;
  int i = 0;
  ...
  ...
}
```

전부 자동 변수로서 함수가 시작되면 생성되고 끝나면 소멸된다.

저장 유형 지정자 register

- CPU 레지스터(register)에 변수를 저장
- 저장 용량이 제한을 받으므로, 선언한다고 모두 저장되지는 않는다

```
register int i;
for(i = 0;i < 100; i++)
  sum += i;
```



저장 유형 지정자 extern

extern1.c

```
#include <stdio.h>

int x;              // 전역 변수
extern int y;       // 현재 소스 파일의 뒷부분에 선언된 변수
extern int z;       // 다른 소스 파일의 변수
int main(void)
{
        extern int x;    // 전역 변수 x를 참조한다. 없어도된다.

        x = 10;
        y = 20;
        z = 30;

        return 0;
}
int y;          // 전역 변수
```

컴파일러에게 변수가 다른 장소에 선언되었음을 알린다

extern2.c

```
int z;
```

연결(binding)

- *연결(binding, linkage):* 다른 장소의 변수들을 서로 매핑
  - 외부 연결(external binding)
  - 내부 연결(internal binding)
  - 무연결(no binding)

- 전역 변수만이 연결을 가질 수 있다

- static 지정자를 사용한다
  - static이 없으면 외부 연결
  - static이 있으면 내부 연결

연결 예제(extern, static+전역변수)

linkage1.c

```c
#include <stdio.h>
int all_files; // 다른 소스 파일에서도 사용할 수 있는 전역 변수
static int this_file; // 현재의 소스 파일에서만 사용할 수 있는 전역 변수
extern void sub();

int main(void)
{
        sub();
        printf("%d₩n", all_files);
        return 0;
}
```

연결

linkage2.c

```c
extern int all_files;
void sub(void)
{
        all_files = 10;
}
```

10

저장 유형 지정자 static

- **static** + 지역변수
  - 지역변수에 static을 붙이는 경우 블록이 종료되더라도 소멸하지 않고 생존시간이 프로그램 종료 시까지로 바뀌며, 저장된 값도 그대로 유지된다
- **static** + 전역변수
  - 전역변수에 static을 불이는 경우 파일 내부에서만 접근 가능하며 파일의 외부에서의 참조를 허용하지 않는다. 즉 연결을 허용하지 않으며 파일 외부에서는 전혀 액세스 할 수 없다
  - Abstract Data Type(ADT) 구현 시 이용
- **static** + 함수
  - 함수 앞에 static을 불이는 경우 마찬가지로 파일 내부에서만 호출 가능하며 파일의 외부에서의 호출을 허용하지 않는다. 즉 연결을 허용하지 않으며 파일 외부에서는 호출하지 못한다
  - Abstract Data Type(ADT) 구현 시 이용

저장 유형 지정자(static + 지역변수)

```
#include <stdio.h>
void sub(void);

int main(void)
{
        int i;
        for(i = 0;i < 3; i++)
                sub();
        return 0;
}
void sub(void)
{
        int auto_count = 0;
        static int static_count = 0;

        auto_count++;
        static_count++;
        printf("auto_count=%d\n", auto_count);
        printf("static_count=%d\n", static_count);
}
```

자동 지역 변수

정적 지역 변수로서 static을 붙이면 지역 변수가 정적 변수로 된다.

```
auto_count=1
static_count=1
auto_count=1
static_count=2
auto_count=1
static_count=3
```

static + 함수

main.c

```c
#include <stdio.h>

extern void f2();
int main(void)
{
        f2();
        return 0;
}
```

static이 붙는 함수는 파일 안에서만 사용할 수 있다

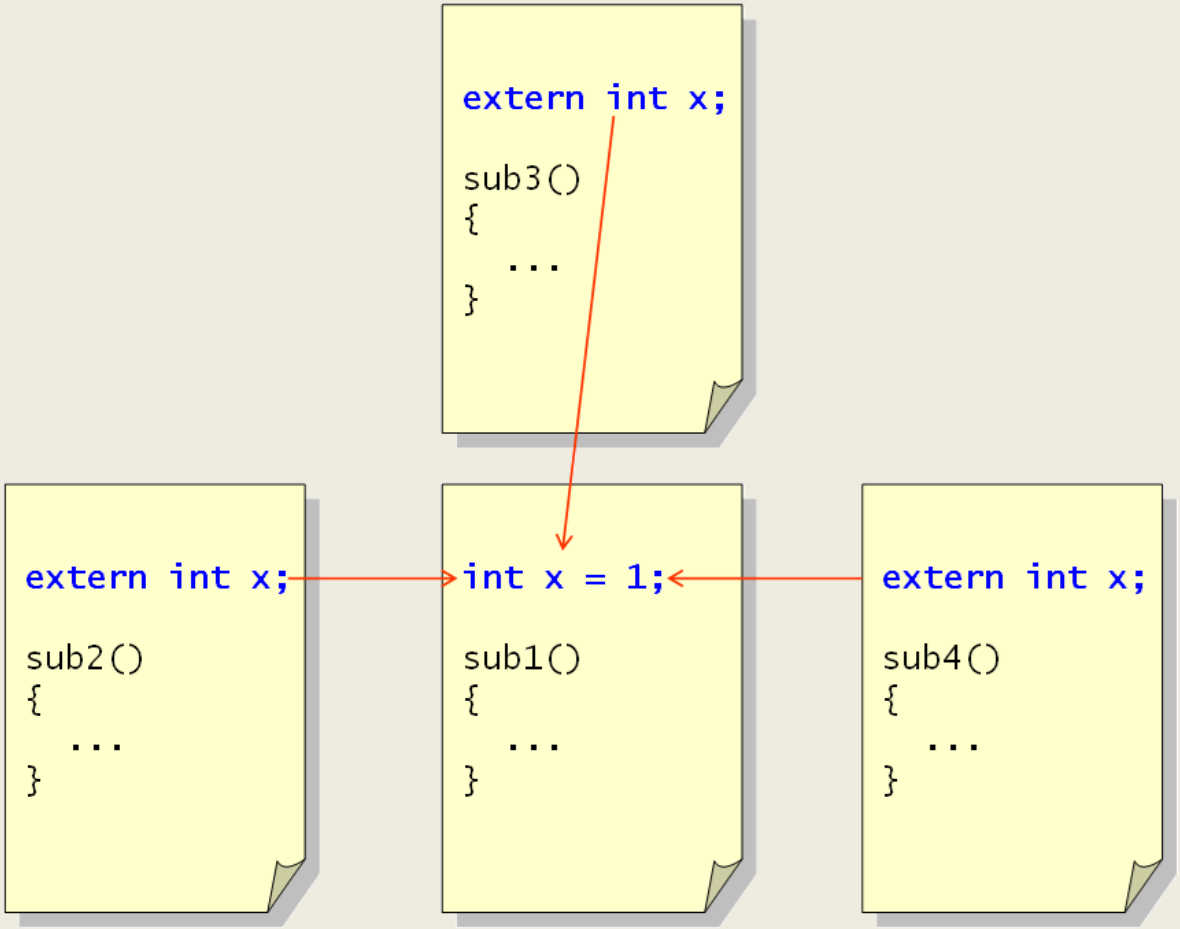
sub.c

```c
static void f1()
{
        printf("f1()이 호출되었습니다.₩n");
}

void f2()
{
        f1();
        printf("f2()가 호출되었습니다.₩n");
}
```

다중 소스 파일

- 연결은 흔히 다중 소스 파일에서 변수들을 연결하는데 사용된다.



저장 유형 정리

- 일반적으로는 *자동 저장 유형* 사용 권장
- 자주 사용되는 변수는 *레지스터 유형*
- 변수의 값이 함수 호출이 끝나도 그 값을 유지하여야 할 필요가 있다면 *지역 정적*
- 만약 많은 함수에서 공유되어야 하는 변수라면 *외부 참조 변수*

저장 유형	*키워드*	*정의되는 위치*	*범위*	*생존 시간*
자동	auto	함수 내부	지역	임시
레지스터	register	함수 내부	지역	임시
정적 지역	static	함수 내부	지역	영구
전역	없음	함수 외부	모든 소스 파일	영구
정적 전역	static	함수 외부	하나의 소스 파일	영구
외부 참조	extern	함수 외부	모든 소스 파일	영구

예제

main.c

```c
#include <stdio.h>
unsigned random_i(void);
double random_f(void);

extern unsigned call_count;        // 외부 참조 변수

int main(void)
{
        register int i;              // 레지스터 변수

        for(i = 0; i < 10; i++)
                printf("%d ", random_i());

        printf("₩n");

        for(i = 0; i < 10; i++)
                printf("%f ", random_f());

        printf("₩n함수가 호출된 횟수= %d ₩n", call_count);
        return 0;
}
```

예제

random.c

```
// 난수 발생 함수
#define SEED 17
#define MULT 25173
#define INC  13849
#define MOD  65536

unsigned int call_count = 0;        // 전역 변수
static unsigned seed = SEED;        // 정적 전역 변수

unsigned random_i(void)
{
        seed = (MULT*seed + INC) % MOD;
        call_count++;
        return seed;
}
double random_f(void)
{
        seed = (MULT*seed + INC) % MOD;
        call_count++;
        return seed / (double) MOD;
}
```

48574 61999 40372 31453 39802 35227 15504 29161 14966 52039
0.885437 0.317215 0.463654 0.762497 0.546997 0.768570 0.422577 0.739731 0.455627 0.720901
함수가 호출된 횟수 = 20

순환함수(recursive function)

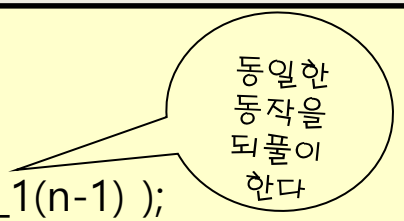
- 순환(혹은 재귀, recursion) : 알고리즘이나 함수가 수행 도중에 자기 자신을 다시 호출하여 문제를 해결하는 기법

- 팩토리얼의 정의 예

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

- 팩토리얼 프로그래밍 #1: (n-1)! 팩토리얼을 구하는 서브 함수를 따로 제작

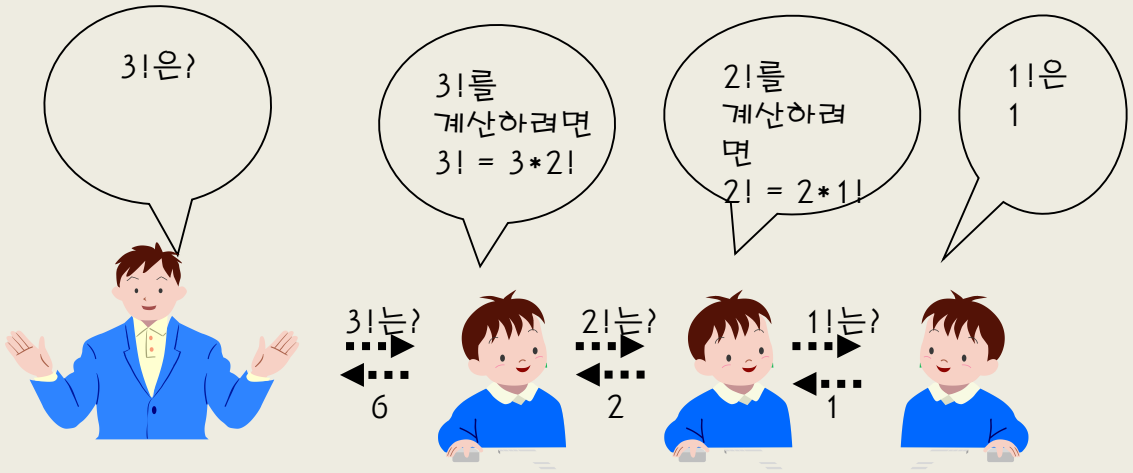
```
int factorial(int n)
{
    if( n == 1 ) return(1);
    else return (n * factorial_n_1(n-1) );
}
```



Factorial 함수

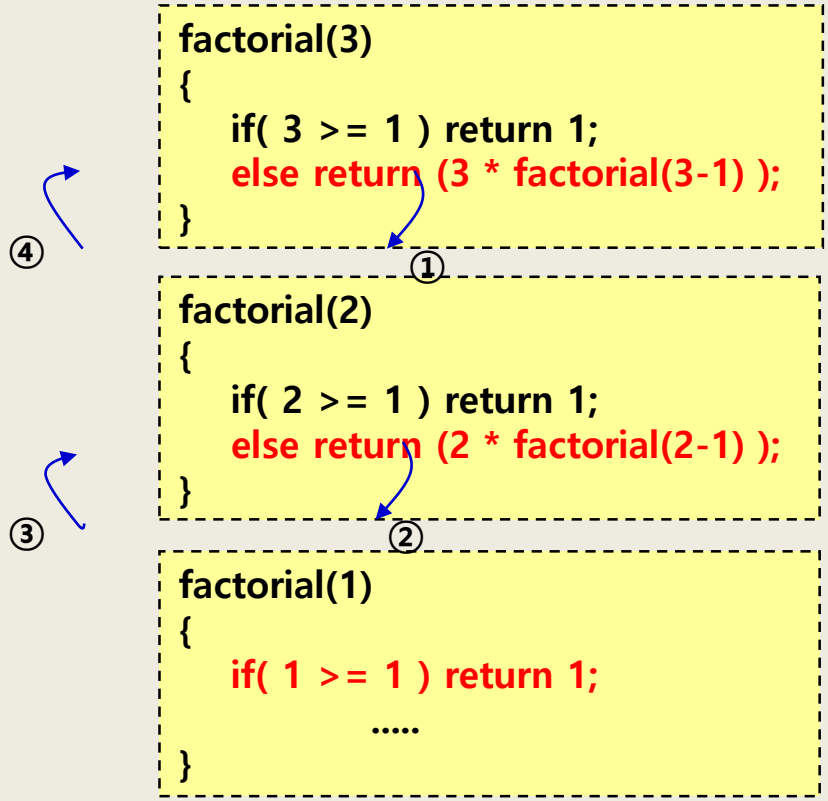
- 팩토리얼 프로그래밍 #2: (n-1)! 팩토리얼을 현재 작성중인 함수를 다시 호출하여 계산(순환 호출)

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



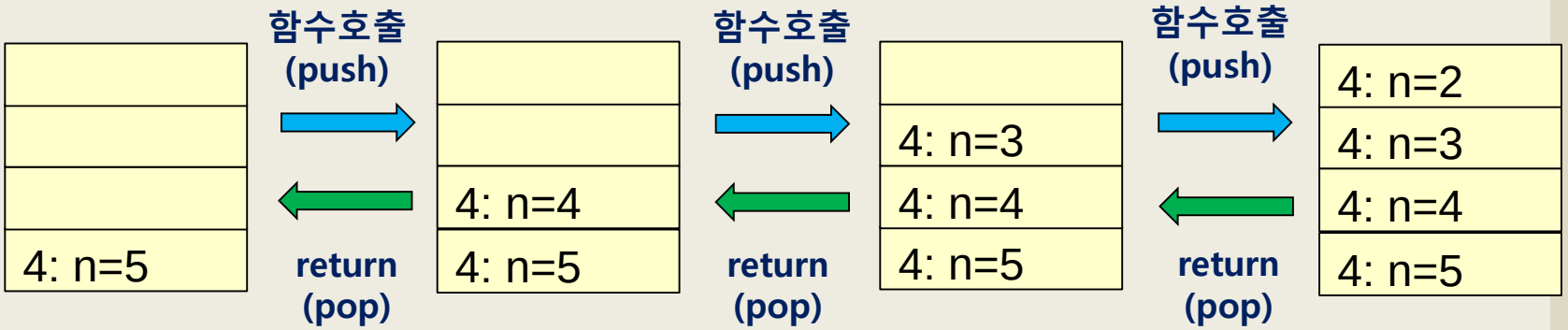
- 팩토리얼의 호출 순서

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



```
int factorial(int n) {
1:   if(n==1)
2:         return 1;
3:   x = factorial(n-1);
4:   x = n*x;
5:   return x;
}
```

예를 들어서 factorial(5)



순환 알고리즘의 구조

- 순환 알고리즘은 다음과 같은 부분들을 포함하여야 한다
  - 순환 호출을 하는 부분
  - 순환 호출을 멈추는 부분

```
int factorial(int n)
{

  if( n == 1 )  return 1

  else return n * factorial(n-1);

}
```

순환을 멈추는 부분

순환호출을 하는 부분

- 만약 순환 호출을 멈추는 부분이 없다면?.
  시스템 오류가 발생할 때까지 무한 호출

순환 ←→ 반복

- 컴퓨터에서의 반복 작업
  - 순환(recursion): 순환 호출 이용
  - 반복(iteration): for나 while을 이용한 반복

- 대부분의 순환은 반복으로 바꾸어 작성할 수 있다.

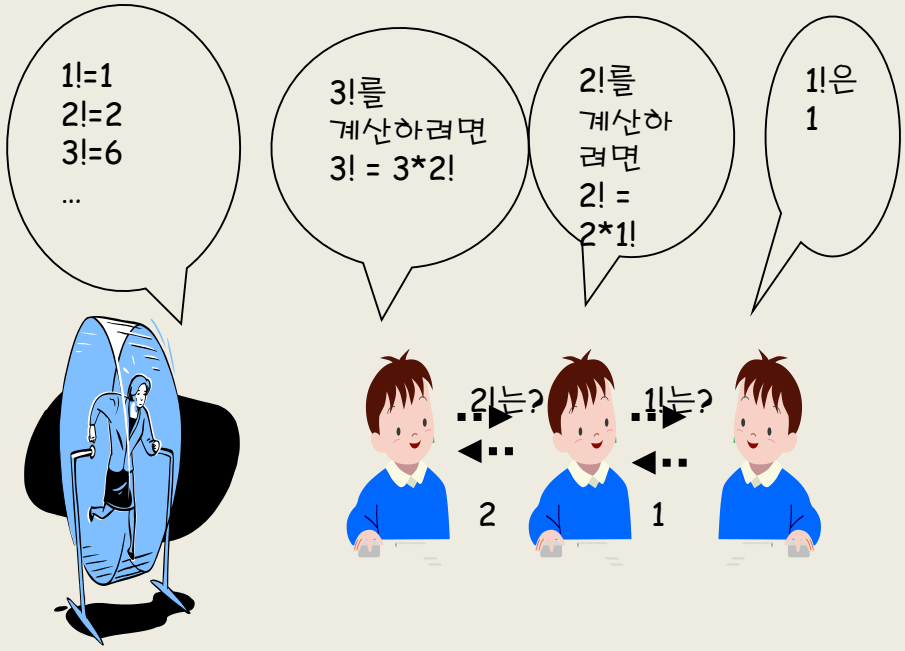
- 순환

  가독성이 매우 좋은 자연스러운 방법이나, 함수 호출의 오버헤드

- 반복

  수행속도가 빠르다.

  순환적인 문제에 대해서는 프로그램 작성이 아주 어려울 수도 있다.



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, value=1;
    for(k=1; k<=n; k++)
        value = value*k;
    return(value);
}
```

재귀함수와 반복함수 사용의 비교

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

recursive function
재귀함수

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

iterative function
반복함수

거듭제곱 구하기 #1

- 순환적인 방법이 반복적인 방법보다 더 효율적인 예
- 숫자 x의 n 승을 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 구하기 #2

- 순환적인 방법

```
power(x, n)

if n=0
        then return 1;
else if n이 짝수
        then return power(x^2,n/2);
else if n이 홀수
        then return x*power(x^2, (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);
}
```

- 순환적인 방법의 시간 복잡도
  - 만약 n이 2의 제곱이라고 가정하면 다음과 같이 문제의 크기가 줄어든다.

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

- 반복적인 방법과 순환적인 방법의 비교

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

피보나치 수열

- 순환 호출을 사용하면 비효율적인 예
- 피보나치 수열

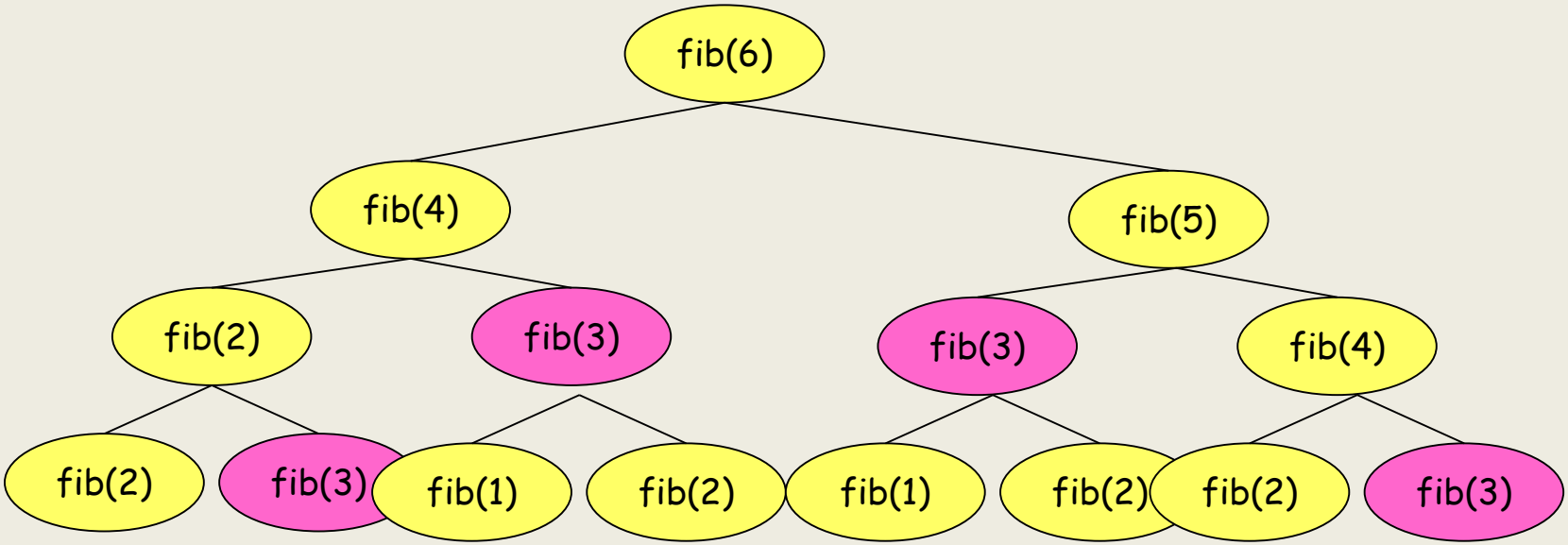
  0,1,1,2,3,5,8,13,21,...

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

- 순환적인 구현

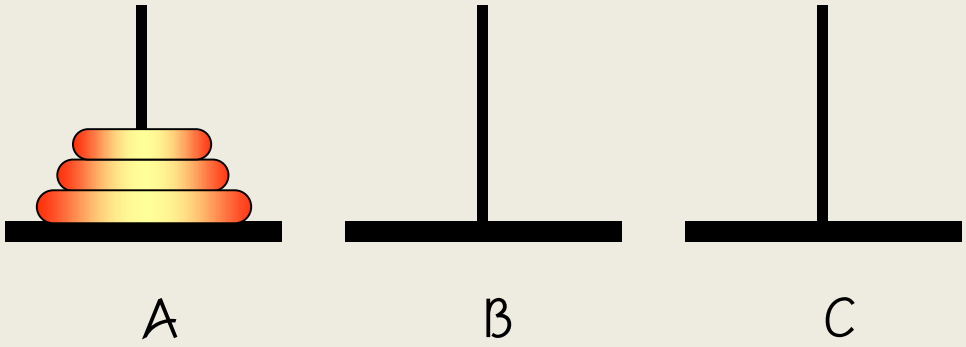
```
int fib(int n)
{
    if( n==0 ) return 0;
    if( n==1 ) return 1;
    return (fib(n-1) + fib(n-2));
}
```

- 순환 호출을 사용했을 경우의 비효율성
  - 같은 항이 중복해서 계산됨
  - 예를 들어 fib(6)을 호출하게 되면 fib(3)이 4번이나 중복되어서 계산됨
  - 이러한 현상은 n이 커지면 더 심해짐

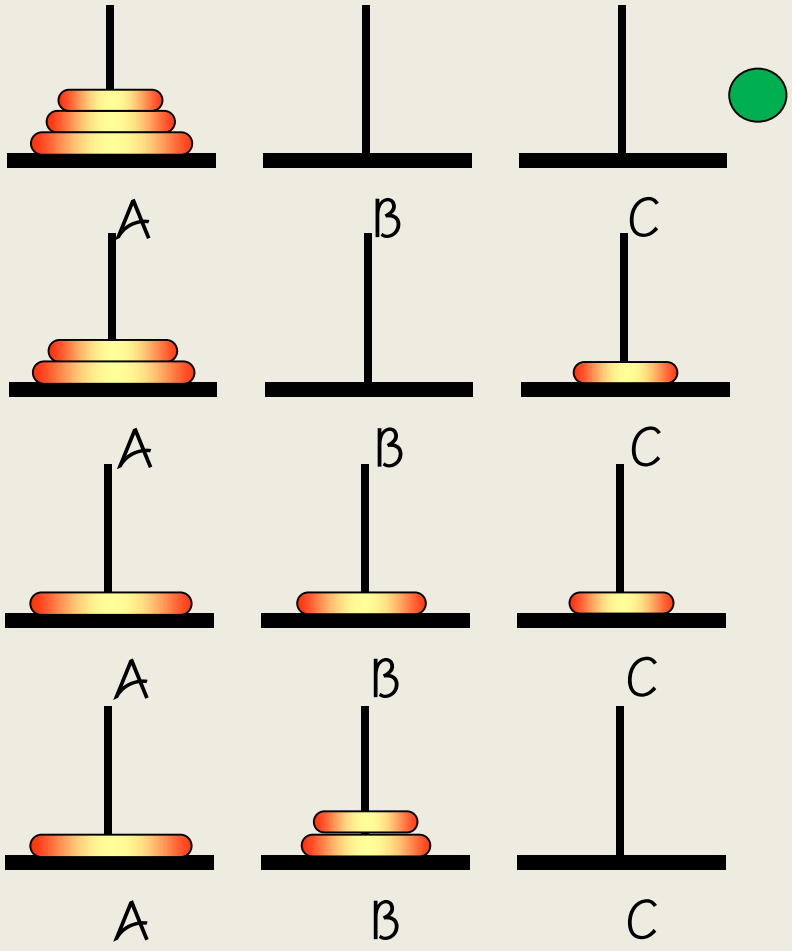


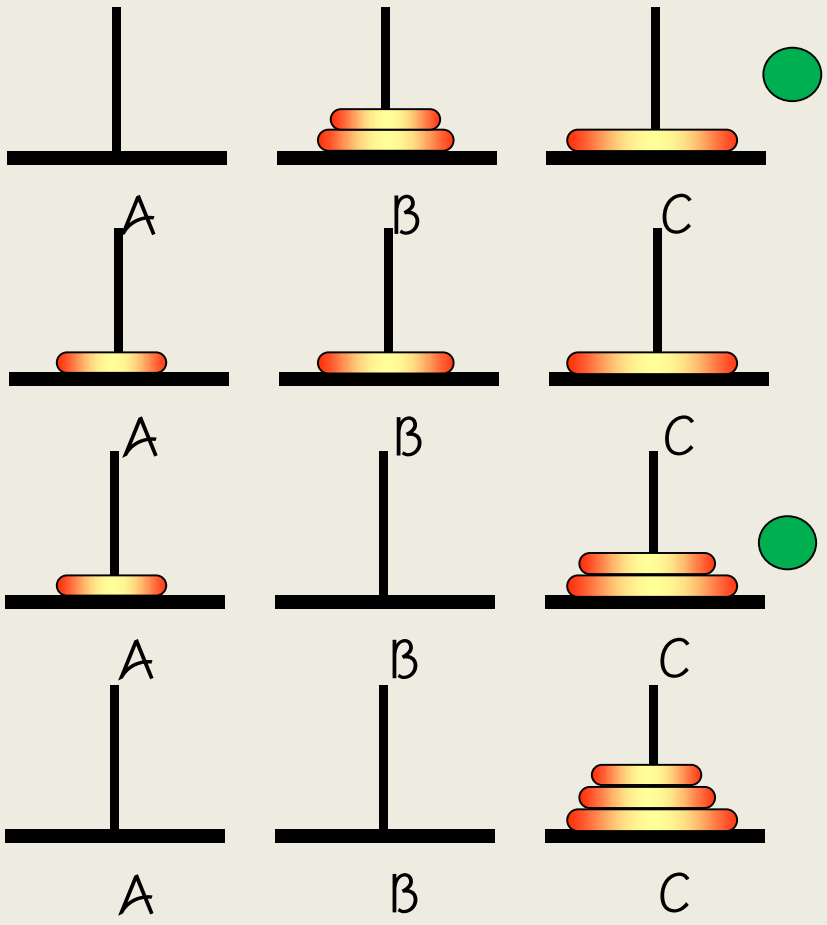
하노이 타워

- 문제는 막대 A에 쌓여있는 원판 3개를 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 원판 위에 큰 원판이 쌓일 수 없다.
  - 중간의 막대를 임시적으로 이용할 수 있으나 앞의 조건들을 지켜야 한다.



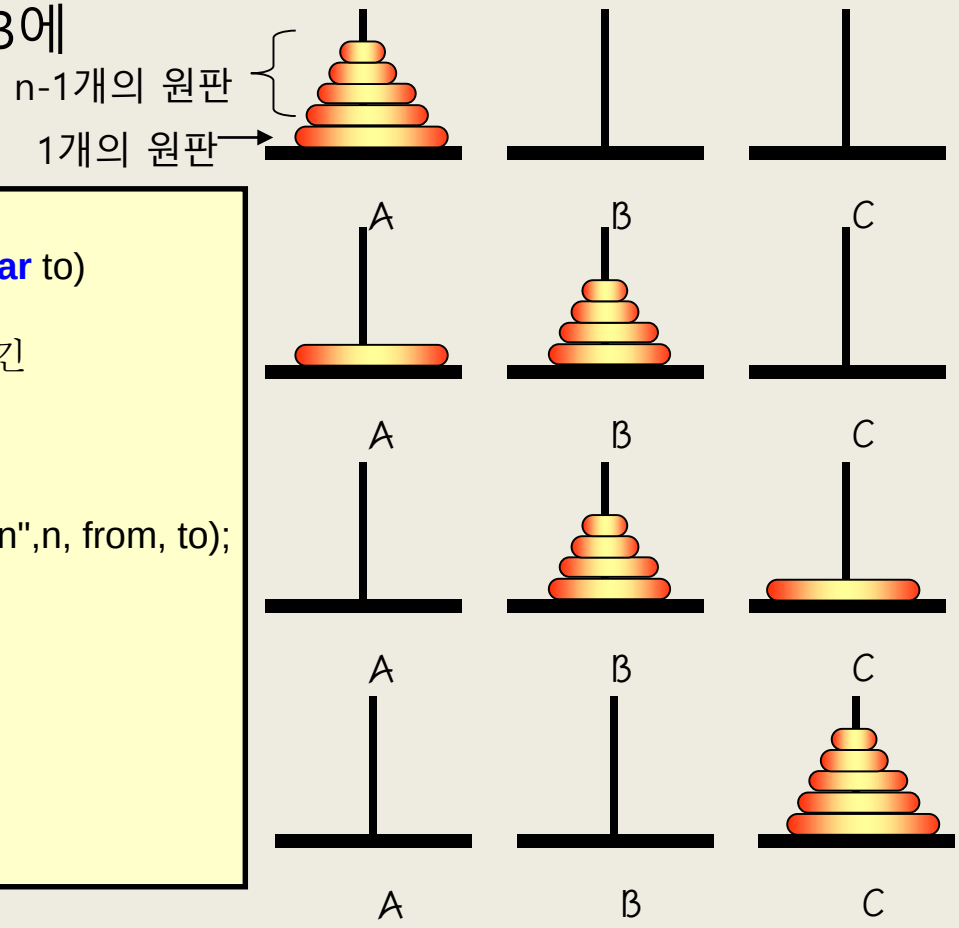
3개의 원판인 경우의 해답



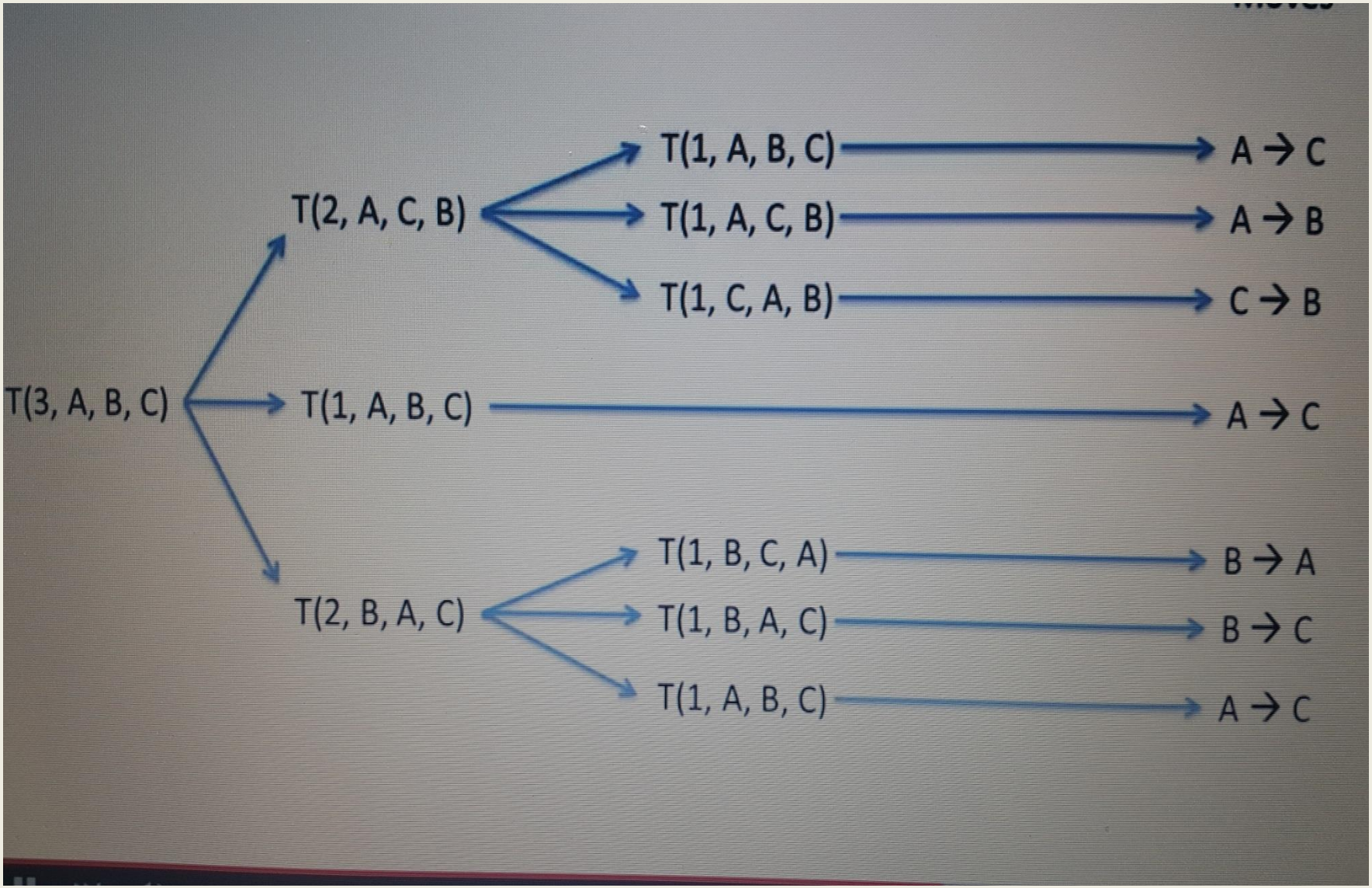


n개의 원판인 경우

- n-1개의 원판을 A에서 B로 옮기고 n번째 원판을 A에서 C로 옮긴 다음, n-1개의 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');
}
```


T(3, A, B, C)
T(2, A, C, B)
T(1, A, B, C)
A → C
T(1, A, C, B)
A → B
T(1, C, A, B)
C → B
T(1, A, B, C)
A → C
T(2, B, A, C)
T(1, B, C, A)
B → A
T(1, B, A, C)
B → C
T(1, A, B, C)
A → C

Algorithm

```
/*N = Number of disks          Beg, Aux, End are the pegs
*/
T(N, Beg, Aux, End)
Begin
   if N = 1 then
      Print: Beg → End;
   else
      Call T(N-1, Beg, End, Aux);
      Call T(1, Beg, Aux, End);
      Call T(N-1, Aux, Beg, End);
   endif
End
```

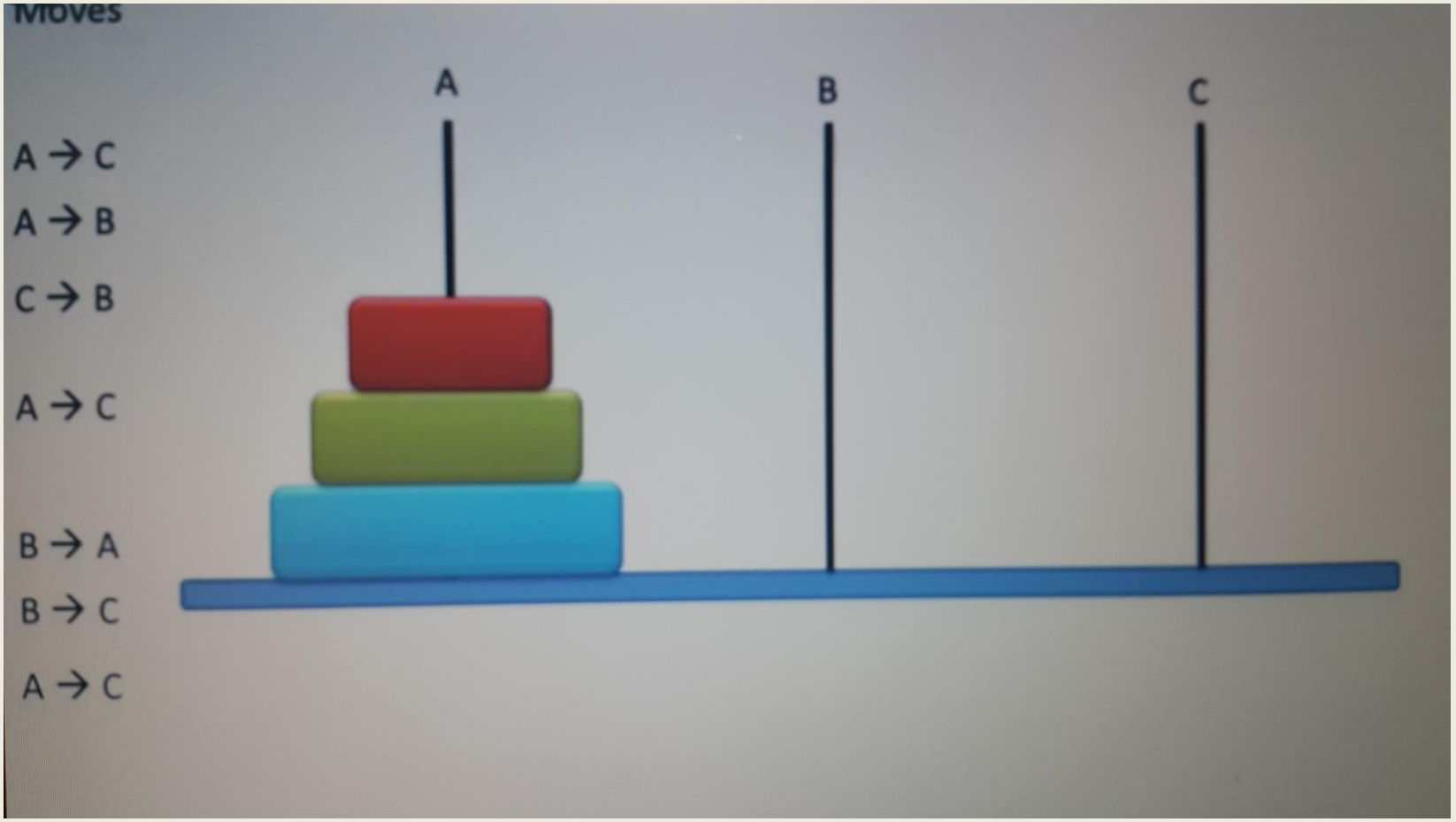
Moves required

If there are N disks then we can solve the game in minimum

$2^N - 1$ *moves*

Example: N = 3

Minimum moves required $= 2^3 - 1 = 7$


Moves
A → C
A → B
C → B
A → C
B → A
B → C
A → C
A
B
C

Moves

A → C ✓

A → B ✓

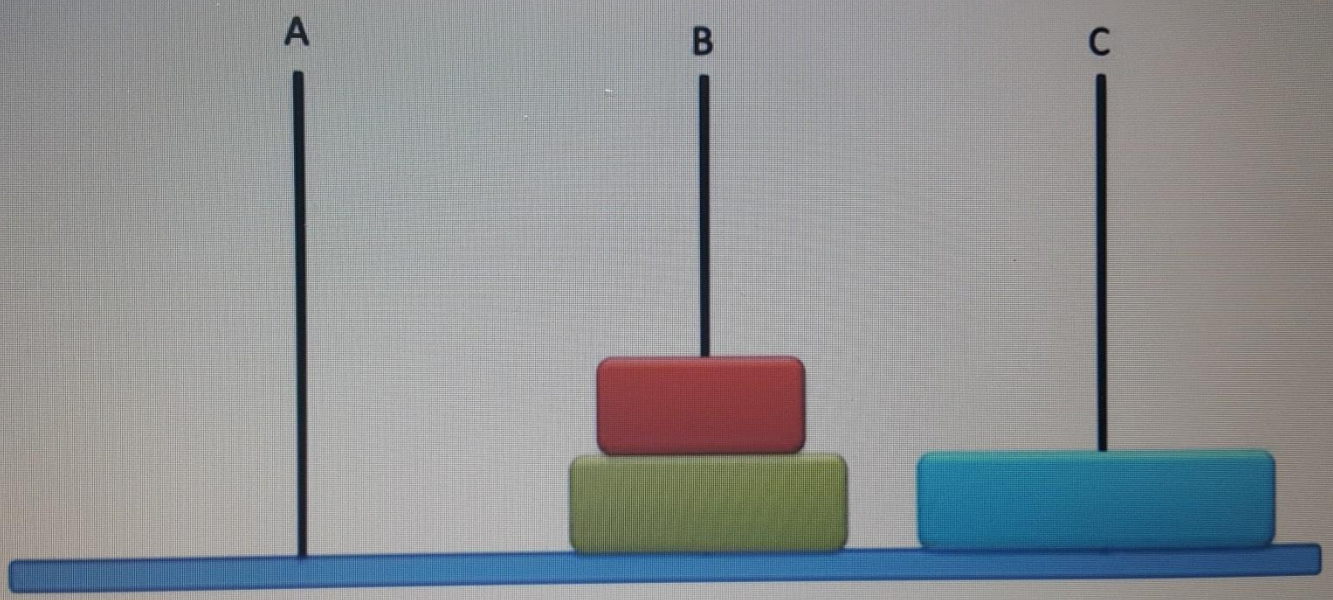
C → B ✓

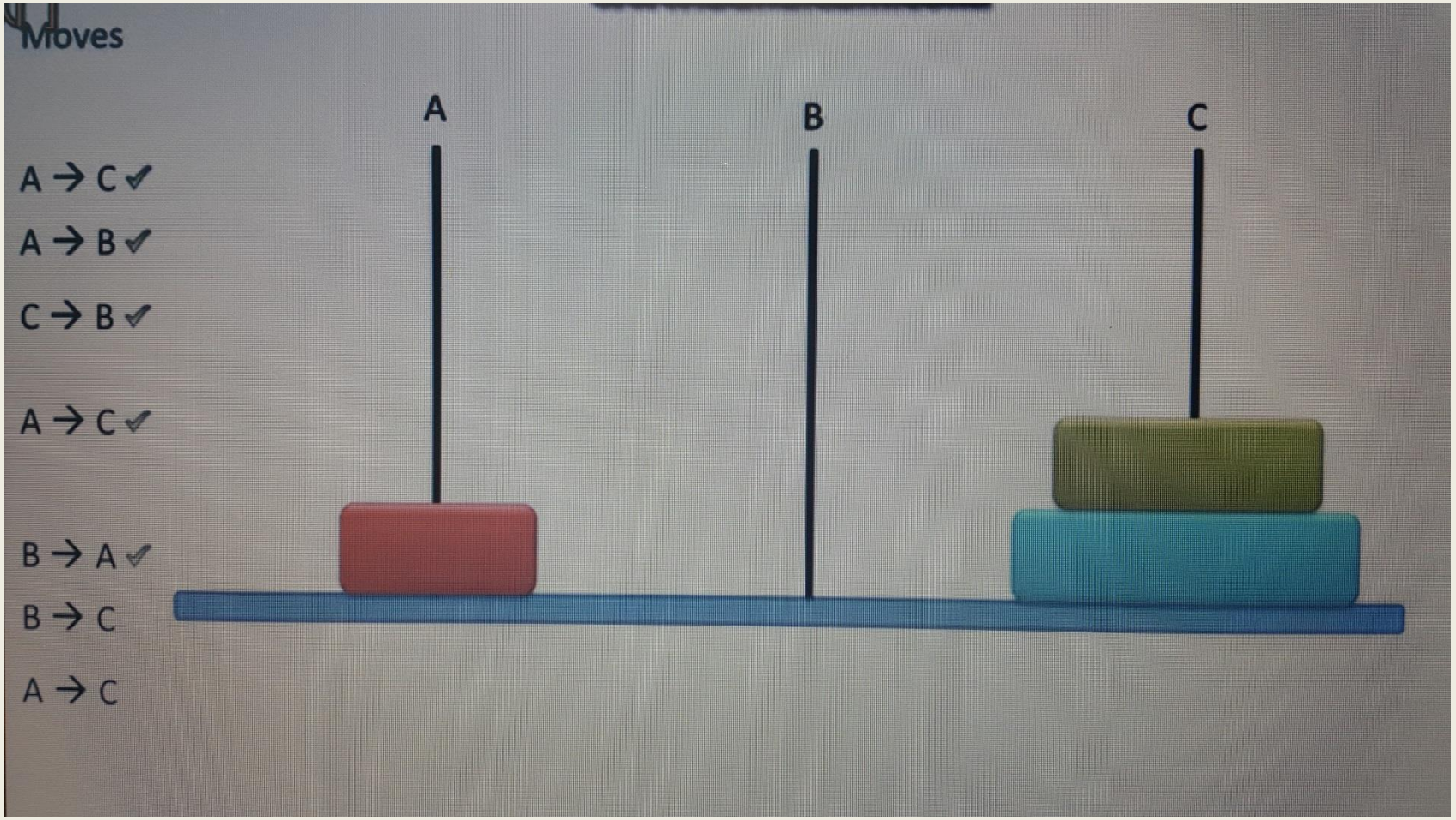
A → C

B → A

B → C

A → C




Moves
A → C ✓
A → B ✓
C → B ✓
A → C ✓
B → A ✓
B → C
A → C
A
B
C