

CHAP 2:재귀 (Recursion)



Chapter 02. 재귀(Recursion)

함수의 재귀적 호출의 이해

Recursion (재귀 or 순환)

- 알고리즘이나 함수가 수행도중 자기 자신을 다시 호출하여 문제를 해결하는 방법
- 정의 자체가 순환적으로 되어 있는 경우에 적합한 방법

Recursion (재귀 or 순환)

- 00위키 발췌 -

어느 한 컴퓨터공학과 학생이 유명한 교수님을 찾아가 물었다.

"재귀함수가 뭔가요?"

"잘 들어보게. 옛날옛날 한 산 꼭대기에 이세상 모든 지식을 통달한 선인이 있었어. 마을 사람들은 모두 그 선인에게 수많은 질문을 했고, 모두 지혜롭게 대답해 주었지. 그의 답은 대부분 옳았다고 하네.

그런데 어느날, 그 선인에게 한 선비가 찾아와서 물었어.

"재귀함수가 뭔가요?"

"잘 들어보게. 옛날옛날 한 산 꼭대기에 이세상 모든 지식을...

Recursion 예

- (예제)

- 팩토리얼 값 구하기

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

- 피보나치 수열

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

- 이항계수

$${}^nC_k = \begin{cases} 1 & n=0 \text{ or } n=k \\ {}^{n-1}C_{k-1} + {}^{n-1}C_k & \text{otherw} \end{cases}$$

- 하노이의 탑
- 이진탐색

재귀함수의 기본적인 이해 1

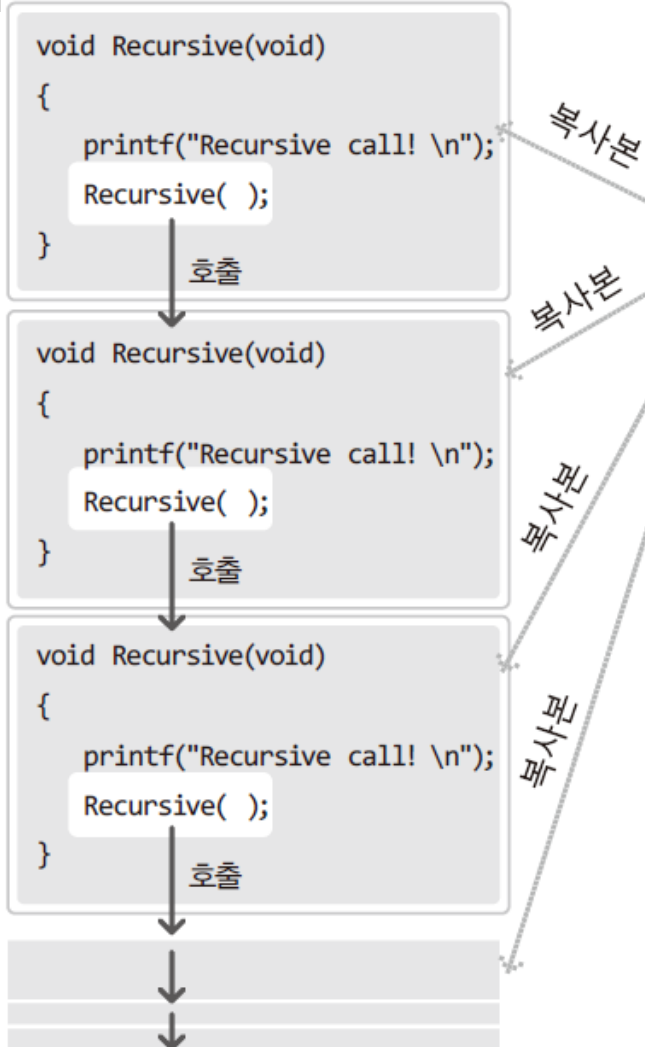
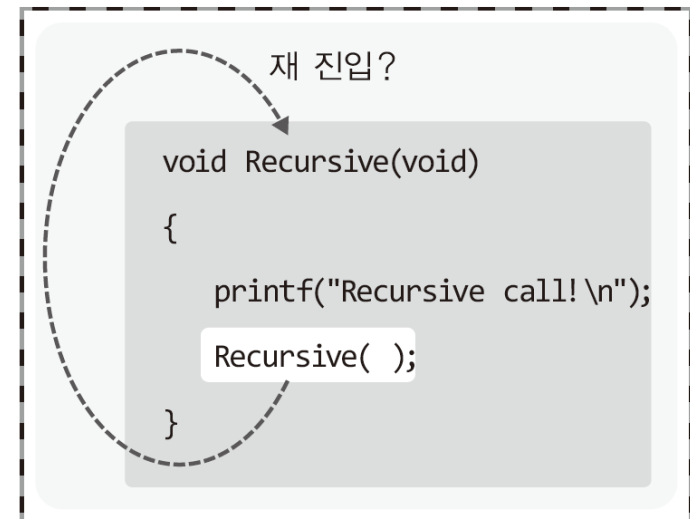
재귀함수의 호출 원리

원 본

```
void Recursive(void)
{
    printf("Recursive call! \n");
    Recursive( );
}
```

재귀? 재진입? 자기호출? 자기복재?

재귀함수의 이해



재귀함수의 기본적인 이해 2

```
void Recursive(int num)
{
    if(num <= 0)        // 재귀의 탈출조건
        return;        //재귀의 탈출!
    printf("Recursive call! %d \n", num);
    Recursive(num-1);
}

int main(void)
{
    Recursive(3);
    return 0;
}
```

재귀함수의 간단한 예

재귀는 탈출조건이 필수적으로 필요!!

실행결과

```
Recursive call! 3
Recursive call! 2
Recursive call! 1
```




c u r s i v e F u n

재귀함수의 디자인 사례

$$n! = n \times (n-1) \times (n-2) \times (n-3) \times \dots \times 2 \times 1$$

$(n-1)!$



$$n! = n \times (n-1)!$$

0!=1
1!=1
3!=1*2*3
5!=1*2*3*4*5
n!=n*(n-1)!

수학적 함수 정의를 사용할 수 있는 재귀!!

$$f(n) = \begin{cases} n \times f(n-1) & \dots n \geq 1 \\ 1 & \dots n = 0 \end{cases}$$

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

```
else
    return n * Factorial(n-1);
```


팩토리얼의 재귀적 구현

```
int Factorial(int n)
{
    if(n == 0)
        return 1;
    else
        return n * Factorial(n-1);
}

int main(void)
{
    printf("1! = %d \n", Factorial(1));
    printf("2! = %d \n", Factorial(2));
    printf("3! = %d \n", Factorial(3));
    printf("4! = %d \n", Factorial(4));
    printf("9! = %d \n", Factorial(9));
    return 0;
}
```

팩토리얼 구현 결과



ursive Factori

실행결과

```
1! = 1
2! = 2
3! = 6
4! = 24
9! = 362880
```

Chapter 02. 재귀(Recursion)

재귀의 활용

재귀와 반복

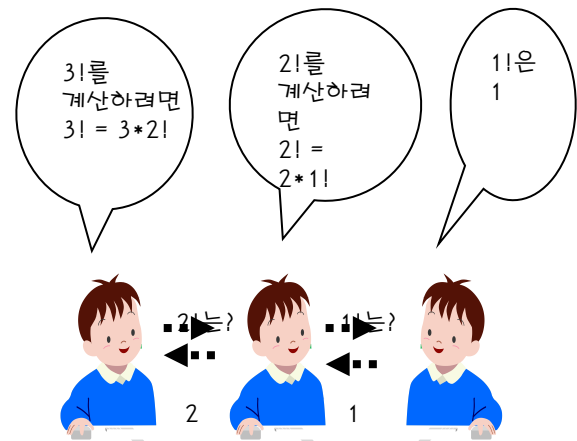
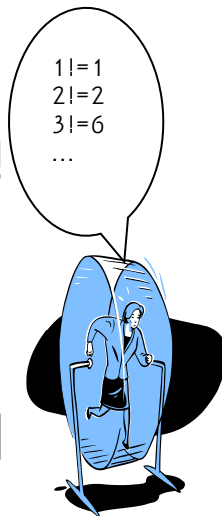
- 컴퓨터에서의 되풀이
 - 순환(recursion): 순환 호출 이용
 - 반복(iteration): for나 while을 이용한 반복
- 대부분의 재귀는 반복으로 바꾸어 작성할 수 있다.

- 재귀

- 순환적인 문제에서는 자연스러운 방법
- 함수 호출의 오버헤드 발생

- 반복

- 수행속도가 빠르다.
- 순환적인 문제에 대해서는 프로그램 작성이 아주 어려울 수도 있다.



팩토리얼 (반복)

$$n! = n * (n-1) * (n-2) * \dots * 1 \quad \begin{matrix} n=1 \\ n \geq 1 \end{matrix}$$

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

거듭제곱 문제

- 재귀적인 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);
}
```

거듭제곱 문제

```
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);
```

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.

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

거듭제곱 문제

- 재귀적인 방법

```
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);
}
```

재귀 주의점

- 함수의 처리 = 호출관계 + 호출순서
 - 호출관계는 함수와 함수간의 호출 관계
 - 호출 순서는 호출 관계에 대해 각 함수가 실제 호출되는 순서
 - 재귀에서 호출 순서의 전체파악은 현실적으로 어렵다.
 - 파악해도 의미가 별로 없다....

피보나치 수열 1: 이해

피보나치 수열

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55

피보나치 수열의 구성

수열의 n 번째 값 = 수열의 $n-1$ 번째 값 + 수열의 $n-2$ 번째 값

피보나치 수열의 표현

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

피보나치 수열 2: 코드의 구현

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

```
int Fibo(int n)
{
    if(n == 1)
        return 0;

    else if(n == 2)
        return 1;

    else
        return Fibo(n-1) + Fibo(n-2);
}
```

피보나치 수열 3: 완성된 예제

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

int main(void)
{
    int i;
    for(i=1; i<15; i++)
        printf("%d ", Fibo(i));

    return 0;
}
```



o n a c c i F u n

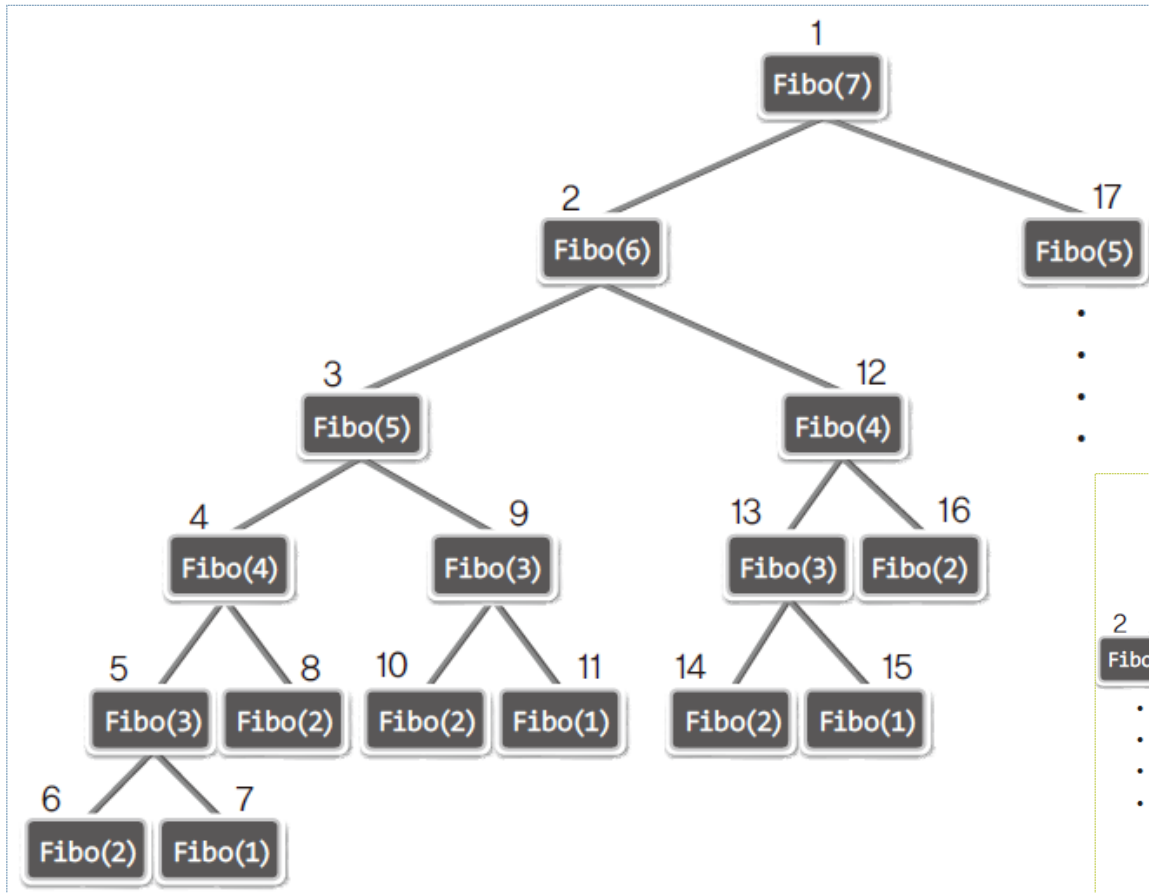
실행결과

0	1	1	2	3	5	8	13	21	34	55	89	144	233
---	---	---	---	---	---	---	----	----	----	----	----	-----	-----

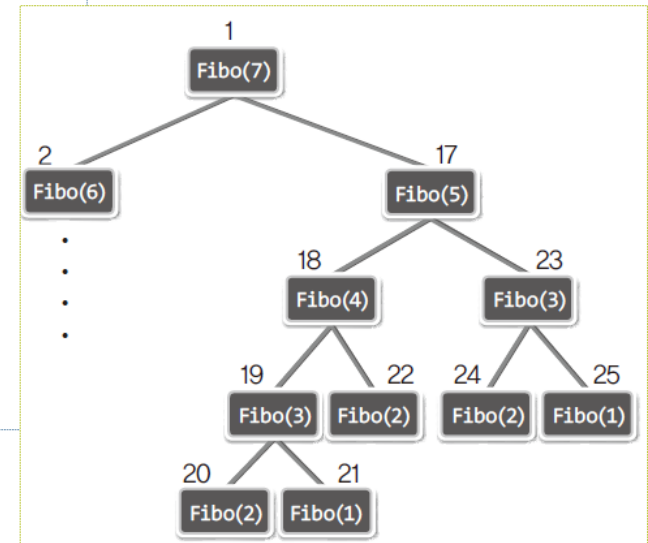
피보나치 수열 4: 함수의 흐름

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

피보나치 수열 4: 함수의 흐름



FuncProcedu1



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;
    }
}
```

이진 탐색 알고리즘의 재귀구현 1



rsive Binary Search

이진 탐색의 알고리즘의 핵심

1. 탐색 범위의 중앙에 목표 값이 저장되었는지 확인
2. 저장되지 않았다면 탐색 범위를 반으로 줄여서 다시 탐색 시작

이진 탐색의 종료에 대한 논의

```
int BSearchRecur(int ar[], int first, int last, int target)
{
    if(first > last)
        return -1;
    . . . .
}
```

first와 last는 각각 탐색의 시작과 끝을 나타내는 변수
// -1의 반환은 탐색의 실패를 의미

이진 탐색 알고리즘의 재귀구현 2

탐색 대상의 확인!

```
int BSearchRecur(int ar[], int first, int last, int target)
{
    int mid;
    if(first > last)
        return -1;

    탐색의 대상에서 중심에 해당하는 인덱스 값 계산
    mid = (first+last) / 2;
    if(ar[mid] == target) 타겟이 맞는지 확인!
        return mid;
    . . . .
}
```


이진 탐색 알고리즘의 재귀구현 3

계속되는 탐색

```
int BSearchRecur(int ar[], int first, int last, int target)
{
    int mid;
    if(first > last)
        return -1;
    mid = (first+last) / 2;

    if(ar[mid] == target)
        return mid;
    else if(target < ar[mid])
        return BSearchRecur(ar, first, mid-1, target);
    else
        return BSearchRecur(ar, mid+1, last, target);
}
```

앞부분을 대상으로 재 탐색

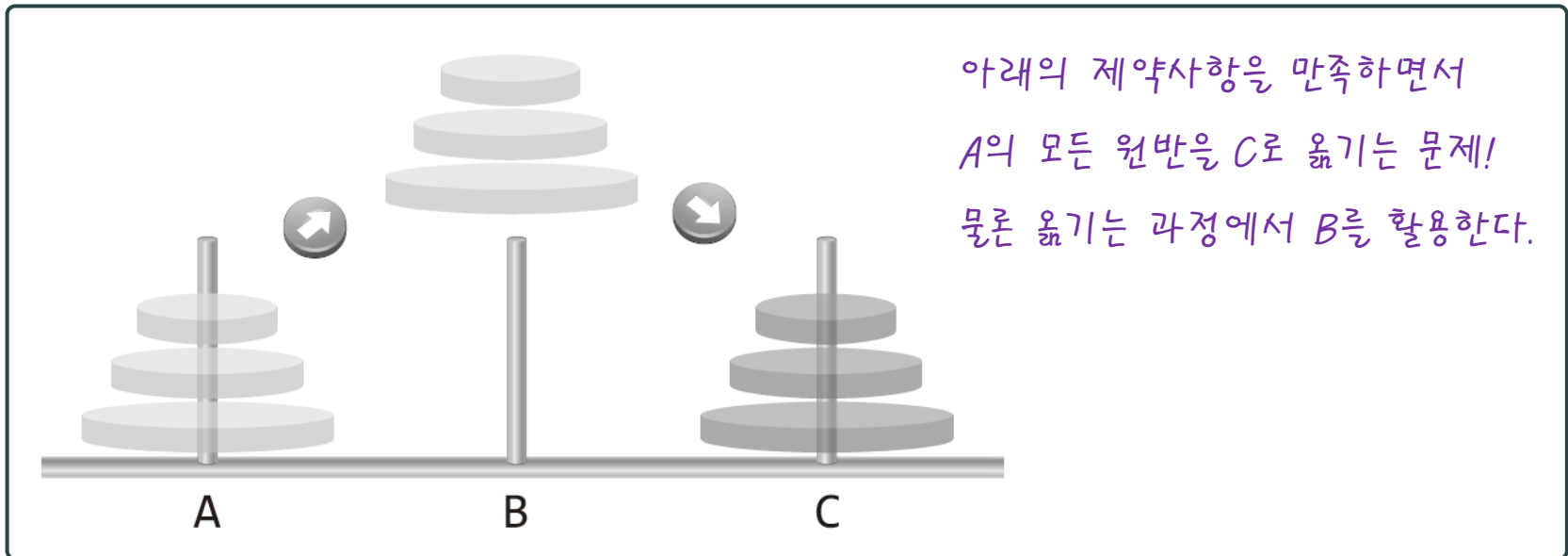
뒷부분을 대상으로 재 탐색

Chapter 02. 재귀(Recursion)

하노이 타워

하노이 타워 문제의 이해

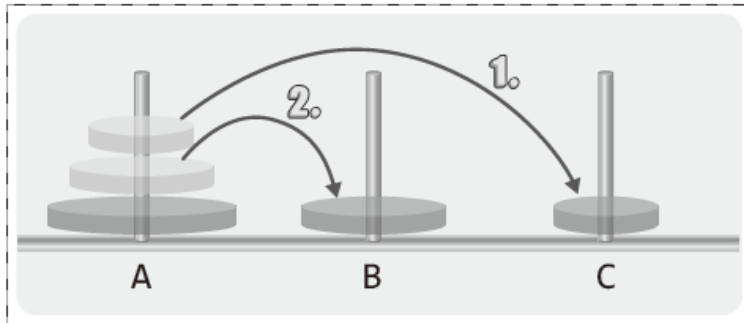
원반을 A에서 C로 이동하기



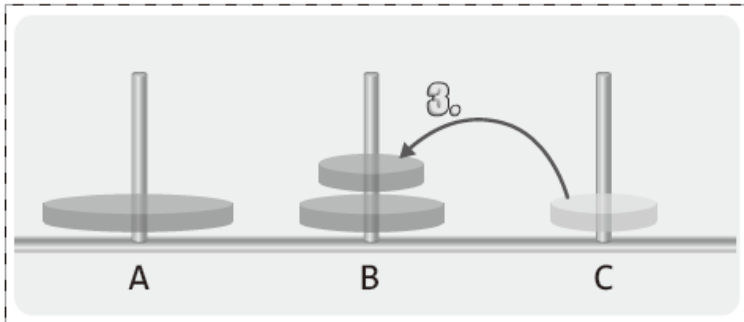
제약사항

- 원반은 한 번에 하나씩만 옮길 수 있습니다.
- 옮기는 과정에서 작은 원반의 위에 큰 원반이 올려져서는 안됩니다.

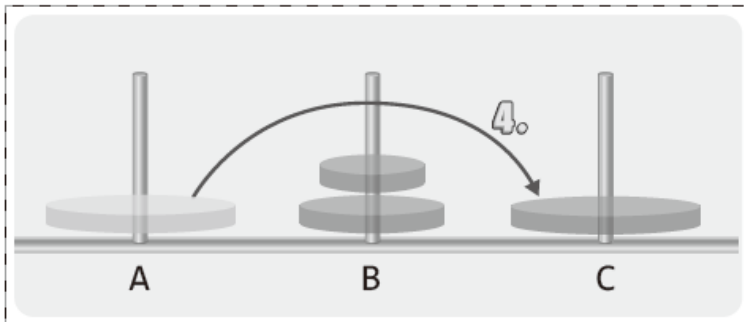
하노이 타워 문제 해결의 예 1



▶ [그림 02-8: 문제해결 1/5]

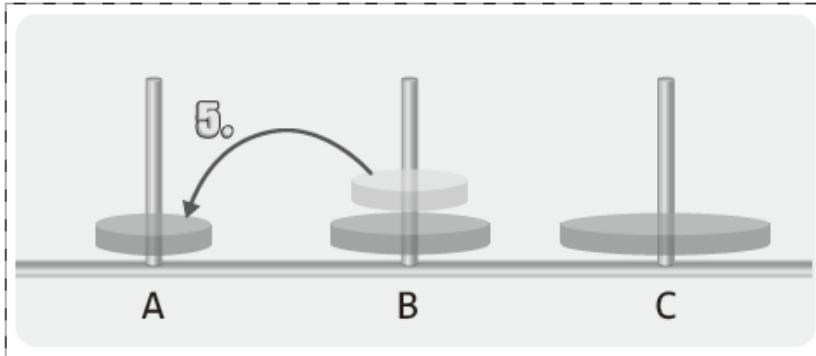


▶ [그림 02-9: 문제해결 2/5]

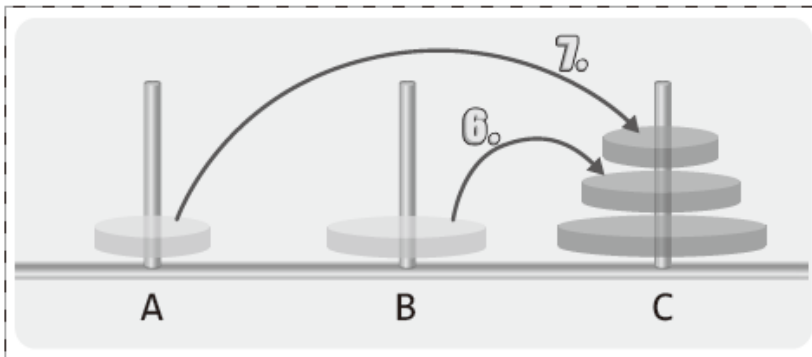


▶ [그림 02-10: 문제해결 3/5]

하노이 타워 문제 해결의 예 2



▶ [그림 02-11: 문제해결 4/5]

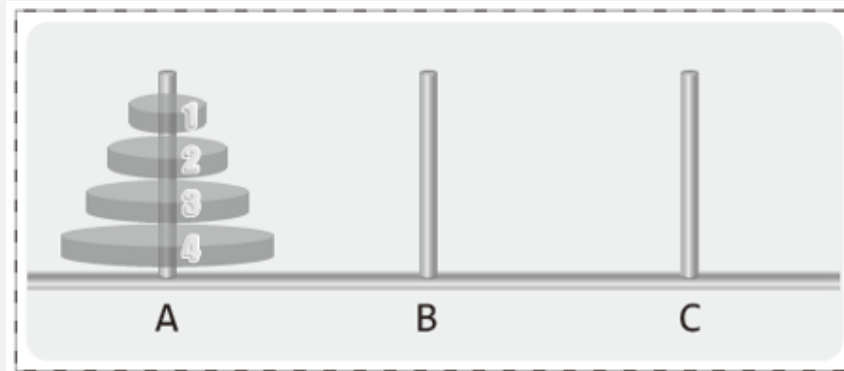


▶ [그림 02-12: 문제해결 5/5]

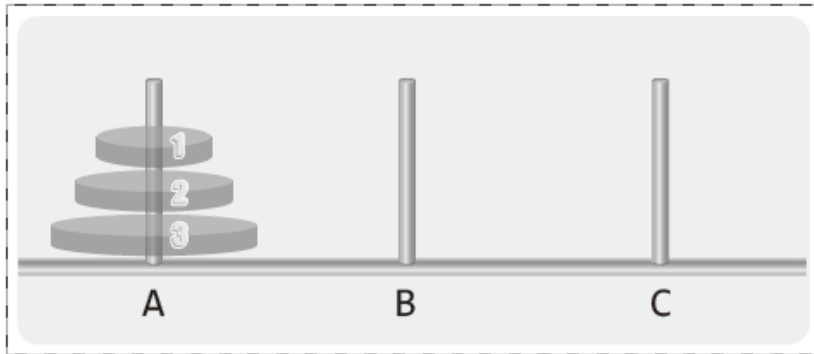
지금 보인 과정에서 반복의 패턴을 찾아야 문제의 해결을 위한 코드를 작성할 수 있다!.

원반 옮기기..

- (1,2,3,4)를 C로 이동
- (4) → C (1,2,3)을 B로 먼저..
- (1,2,3) → B로 , (1,2)를 C로 먼저..
- (1,2,3) → C로 , (1)을 B로 먼저..
- (1) → B로
- 즉, n 개의 원반을 옮기는 문제는 $n-1$ 개의 원반을 옮기는 문제로 해결해 나갈수 있다.



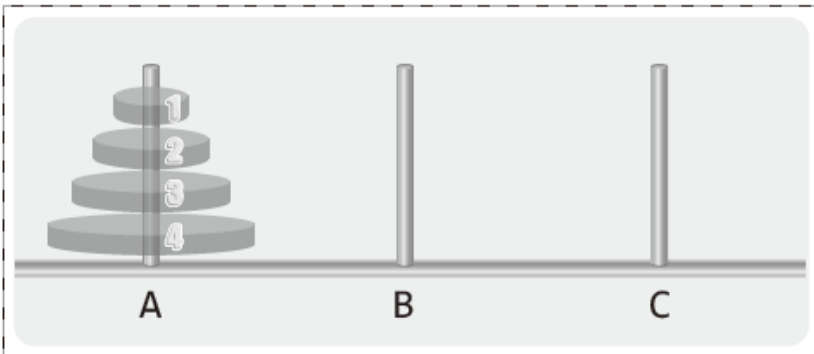
반복되는 일련의 과정을 찾기 위한 힌트



A의 세 원반을 C로 옮기기 위해서는 **원반 3을 C로** 옮겨야 한다. 그리고 이를 위해서는 원반 1과 2를 우선 **원반 B로** 옮겨야 한다.

▶ [그림 02-13: 원반이 3개인 하노이 타워]

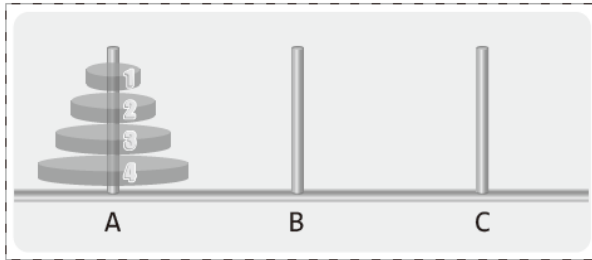
위와 아래의 두 예를 통해서 문제의 해결에 있어서 반복이 되는 패턴이 있음을 알 수 있다.



A의 네 원반을 C로 옮기기 위해서는 **원반 4를 C로** 옮겨야 한다. 그리고 이를 위해서는 원반 1과 2와 3을 우선 **원반 B로** 옮겨야 한다.

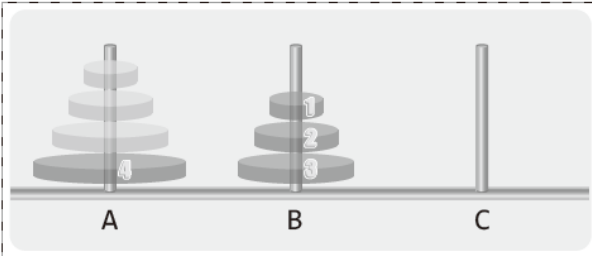
▶ [그림 02-14: 원반이 4개인 하노이 타워]

하노이 타워의 반복패턴 연구 1



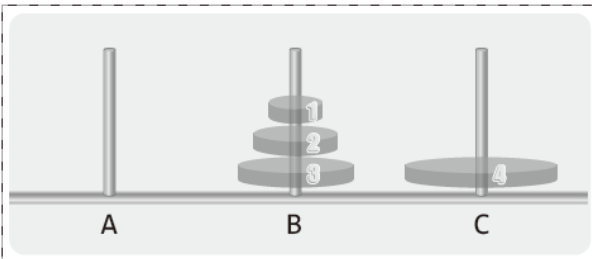
목적. 원반 4개를 A에서 C로 이동

▶ [그림 02-14: 원반이 4개인 하노이 타워]



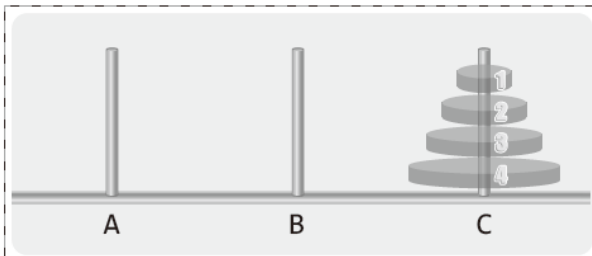
1. 작은 원반 3개를 A에서 B로 이동

▶ [그림 02-15: 반복패턴 1/3]



2. 큰 원반 1개를 A에서 C로 이동

▶ [그림 02-16: 반복패턴 2/3]



3. 작은 원반 3개를 B에서 C로 이동

▶ [그림 02-17: 반복패턴 3/3]

하노이 타워의 반복패턴 연구 2

목적. 원반 4개를 A에서 C로 이동

목적. 큰 원반 n 개를 A에서 C로 이동

1. 작은 원반 3개를 A에서 B로 이동

1. 작은 원반 $n-1$ 개를 A에서 B로 이동

2. 큰 원반 1개를 A에서 C로 이동

2. 큰 원반 1개를 A에서 C로 이동

3. 작은 원반 3개를 B에서 C로 이동

3. 작은 원반 $n-1$ 개를 B에서 C로 이동

하노이 타워 문제의 해결 1

하노이 타워 함수의 기본 골격

```
void HanoiTowerMove(int num, char from, char by, char to)
{
    HanoiTowerMove(4,'A','B','C')
    원반 num의 수에 해당하는 원반을 from에서 to로
    . . . .
    이동을 시키되 그 과정에서 by를 활용한다.
}
```

- | | |
|--------------------------|--------------------------------------|
| 목적. 큰 원반 n개를 A에서 C로 이동 | HanoiTowerMove(num, from, by, to); |
| 1. 작은 원반 n-1개를 A에서 B로 이동 | HanoiTowerMove(num-1, from, to, by); |
| 2. 큰 원반 1개를 A에서 C로 이동 | printf(. . . .); |
| 3. 작은 원반 n-1개를 B에서 C로 이동 | HanoiTowerMove(num-1, by, from, to); |

하노이 타워 문제의 해결 2

목적. 큰 원반 n 개를 A에서 C로 이동

1. 작은 원반 $n-1$ 개를 A에서 B로 이동

2. 큰 원반 1개를 A에서 C로 이동

3. 작은 원반 $n-1$ 개를 B에서 C로 이동

HanoiTowerMove(num, from, by, to);

HanoiTowerMove(num-1, from, to, by);

printf(. . .);

HanoiTowerMove(num-1, by, from, to);

```
void HanoiTowerMove(int num, char from, char by, char to)
{
    if(num == 1)          // 이동할 원반의 수가 1개라면
    {
        printf("원반1을 %c에서 %c로 이동 \n", from, to);
    }
    else
    {
        HanoiTowerMove(num-1, from, to, by);
        printf("원반%d을(를) %c에서 %c로 이동 \n", num, from, to);
        HanoiTowerMove(num-1, by, from, to);
    }
}
```

하노이 타워 문제의 해결 3

```
void HanoiTowerMove(int num, char from, char by, char to)
{
    if(num == 1)        // 이동할 원반의 수가 1개라면
    {
        printf("원반1을 %c에서 %c로 이동 \n", from, to);
    }
    else
    {
        HanoiTowerMove(num-1, from, to, by);
        printf("원반%d을(를) %c에서 %c로 이동 \n", num, from, to);
        HanoiTowerMove(num-1, by, from, to);
    }
}

int main(void)
{
    // 막대A의 원반 3개를 막대B를 경유하여 막대C로 옮기기
    HanoiTowerMove(3, 'A', 'B', 'C');
    return 0;
}
```



io i T o w e r S o

실행결과

원반1을 A에서 C로 이동
원반2을(를) A에서 B로 이동
원반1을 C에서 B로 이동
원반3을(를) A에서 C로 이동
원반1을 B에서 A로 이동
원반2을(를) B에서 C로 이동
원반1을 A에서 C로 이동