<참고 교재 : C언어로 쉽게 풀어쓴 자료구조, 천인국 등>

<2장, 순환(RECURSION), 김성현, 2019107783>.

2.1 순환의 소개	2.2. 팩토리얼 (Factorial)	2.3 거듭제곱 (Power)	2.4 피보나치 (Fibonacci) 수열	2.5 하노이 탑 (Hanoi tower)
- 순환이란?	- 팩토리얼 (순환)	- 거듭제곱 (반복)	- 피보나치 수열 (순환)	- 하노이 탑
- 순환의 보기	- 팩토리얼 (반복)	- 거듭제곱 (순환)	- 피보나치 수열 (반복)	

2.1 순환의 소개.

- 순환이란?

p.40:

□ 순환 (Recursion) 이란 ... → A라는 걸 정의할 때, A라는 걸 이용해서 정의를 한다~

- 어떤 것을 정의할 때 자기 자신의 정의를 사용하여 정의하는 방법.
- 순환은 수학이나 컴퓨터 과학에서 어떤 문제를 풀기 위하여 많이 사용되는 방법이다. (특히 프로그래밍)

⇒ 어떤 알고리즘이나 함수가 자기 자신을 호출하여 문제를 해결하는 프로그래밍 기법!

□ 순환을 사용한 정의 : base case / induction step.

- base case : 순환을 사용하지 않고 정의하는 부분.
- induction step : 순환을 사용하여 정의하는 부분 (최종적으로 base case의 정의를 사용).

□ 순환으로 정의된 집합 (특성) ➡ 수학적 귀납법 (mathematical induction)을 사용하여 쉽게 증명 가능.
(ex: 프로그램의 특성 증명 등).

↳ 순환으로 정의된 집합들의 여러가지 특성들은 수학적 귀납법으로 "쉽게" 증명할 수 있다!! 라는 거네.

★ • 알고리즘 ⇒ 그 '증명'이 제일 중요 (잘 돌아가는지)

➡ 이럴 때 '순환'으로 되어있으면 증명하기가 편하다. ~.

p.41 □ 순환의 보기 (순환의 예)

2장
- 팩토리얼 (Factorial)
- 거듭제곱 (Power)
- 피보나치 (Fibonacci) 수열
- 하노이 탑 (hanoi tower)

그 뒤에
- 트리 순회 (Tree traverse) - 트리의 노드들을 방문하면서 하고 싶은 일을 하기
- 이진 탐색 (Binary search) - 사전 같은데서 이진 탐색으로 쉽게 찾기
- 합병 정렬 (Merge sort) → 정렬 (줄세우기)
- 퀵 정렬 (Quick sort)

프로그래밍에서 정렬이 많이 쓰이고, 자주 많이 사용할 정렬 방법 중 하나

2.2 팩토리얼(Factorial).

p.40 ~ p.43.
(순환의 예).

- 팩토리얼 (Factorial) (순환).

  ㅁ 팩토리얼 정의 (순환) (팩토리얼이란 함수를 스스로 '순환'으로 정의)

  ☆ (=) ⇒ $n! = \begin{cases} 1 & (n \le 1) \\ n*(n-1)! & \text{otherwise.} \end{cases}$

  ① base case (순환 사용X)
  ② Induction step (순환 사용O)

  → 순환!
  $n = 2, 3, 4 \ldots$
  → 자기 자신을 사용해서 정의!

  ㅁ 팩토리얼 함수 (순환).

상황 함수 짜기서 쉬운데... 결과/내용은 보면, 해석하긴 쉽지 않다... 재밌는 Topic이고 반복도 해야!

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

자기자신을 또 씀 (1을 쓰진 않았지만) ⇒ 팩토리얼을 정의할 때 팩토리얼을 썼다...!

- 팩토리얼 함수 (순환) 수행.

순환 ⇒ 재귀함수

ㅁ factorial(3) 수행 과정.

factorial(3) ① — ③ 순서
= (3 * factorial(2)) ② — ②
= (3 * (2 * factorial(1))) ③ — ①
= (3 * (2 * 1))
= (3 * 2) → factorial(3) 아직 안 끝났어!
= 6.

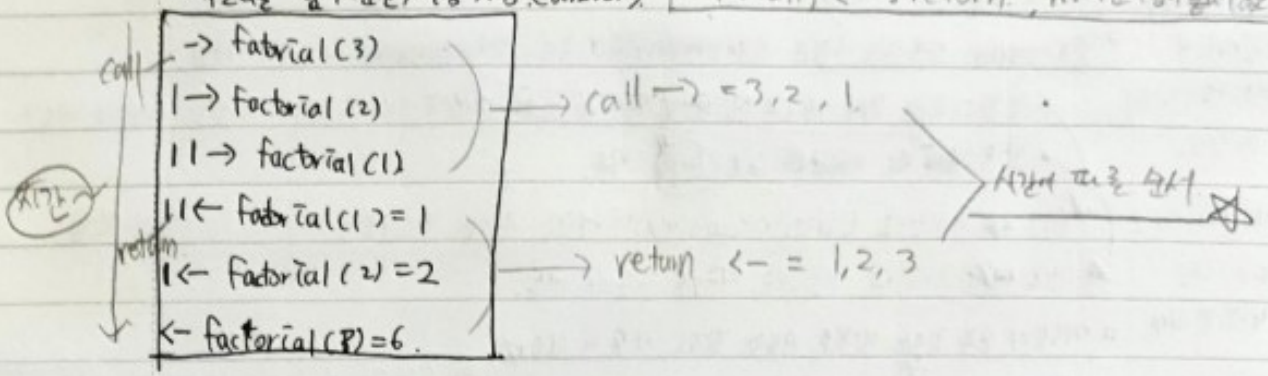
```
fatorial(3)
{
  if(3 <= 1) return 1;
  else return (3* factorial(3-1));
}

fatorial(2)
{
  if (2 <= 1) return 1;
  else return (2* factorial(2-1));
}

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

불린 순서와 끝나는 순서를 잘 알아야 한다!

중요!
factorial(3) 이라는게 불려져서 factorial(2)가 또 불려지는데,
└ 이건 factorial(3)이 끝나기 전에 불려지는 거야!

Q.
⇒ 시간적으로 fatorial(3)이 먼저 불렸냐, factorial(2)가 먼저 불렸냐?
⇒ 당연히 factorial(3)이 먼저 불렸지~

• 팩토리얼 함수(순환) 수행 과정 (Behavior) ->: call, <-: return, ||: 이전 수행이 끝나기 기다리는 상태



p.44. ~ p.45

• 순환 호출의 내부적인 구현

1. - 메서의 함수가 자기 자신을 다시 호출하는 것은 다른 함수를 호출하는 것과 동일.

➡ 복귀주소가 시스템 스택(stack)에 저장, 호출되는 함수를 위한 매개변수(parameter)와 지역변수를 위한 메모리 할당 받음.

호출될 함수를 위한 스택(stack)에서의 공간 ⇒ 활성 레코드(activation record).

2. 위의 준비가 끝나면 호출된 함수의 시작위치로 점프하여 수행을 시작.

↳ 만약 호출된 함수가 자기 자신이라면 자기 자신의 시작위치로 점프하게 되는 것.

3. 호출된 함수가 끝나게 되면 시스템 스택(stack)에서 복귀주소를 꺼내, 호출한 함수로 되돌아가게 됨.

※ 순환 호출이 계속 중첩될수록, 시스템 스택에서는 활성 레코드들이 쌓이게 된다.

p.46

• 순환함수의 구조 (순환 알고리즘의 구조).

- 순환함수는 두 부분으로 구성됨.

2 part
- ✓ Base case : 비 순환 부분 (순환을 멈추게 하는데 사용됨)
- ✓ Induction step : 순환 부분 (최종적으로 비 순환부분을 사용하게 됨.)

```
int factorial(int n)
{
  if (n <= 1) return 1;          ← Base case (비 순환 부분)
  else return (n * factorial(n-1));   ← Induction step (순환 부분)
}
```

컴퓨터 반복적 ⇔ 순환 ⇔ 반복

p.46~47. • 순환 및 반복 (장점 ↔ 단점)

• 되풀이하는 것 → 많은 컴퓨터 알고리즘의 특징임.

□ 컴퓨터에서의 반복적인 수행은 순환(recursion) 혹은 반복(iteration)으로 표현 가능함.

- 순환: 순환 함수를 이용 (주어진 문제 해결을 위해 자신을 다시 호출하여 작업을 수행하는 방식)
- 반복: for 혹은 while과 같은 loop를 이용.

• 바로게 되풀이하는 컴퓨터의 강한 능력 중 하나

순환 ➡ 본질적으로 순환적인 (recursive) 문제나 그러한 자료구조를 다루는 프로그램에 적합.

반복 ➡ 간명하고 효율적으로 되풀이를 구현하는 방법.

□ 대부분의 순환 함수는 반복을 사용한 함수로 바꿀 수 있음.

• 기본적으로 반복과 순환 → 문제해결 능력이 같음 → 순환 호출이 끝에서 이루어지는 순환 : 꼬리순환 (tail recursion)

• 순환 : 장점 : 여러 문제에서도 반복에 비해 알고 리즘이 훨씬 명확하고 간결하지 (반복 알고리즘으로 실제 바꾸어 쓸 수 있음.)

단점 : 일반적으로는 함수 호출을 많이 하므로 반복에 비해 수행속도 면에서는 떨어짐.

→ 알고리즘 설명은 순환으로 하고 실제 프로그램에서는 그것을 반복 버전으로 구현하는 경우도 많음 (물론 순환이 더 쉬운 제거도 존재함!!)

ex) 팩토리얼 (반복)

p.47. □ 팩토리얼 정의 (반복)

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

recursion 아님 X (factorial의 정의에 따라서!)

□ 팩토리얼 함수 (반복)

```
int factorial_iteration(int n)
{
    int i, f = 1;

    for (i = n; i > 0; i--)
        f = f * i;
    return (f);
}
```

$n! = n*(n-1)!$ ↔ $n! = n*(n-1)*(n-2)\cdots*1$

p.48. • 순환과 반복중 어떤 형태가 바람직할까?

- 문제의 정의가 순환적으로 되어있는 경우 => 순환으로 작성하는 것이 훨씬 더 쉽다.
  순환형태의 코드가 이해하기 더 쉬움.
  → 프로그램의 가독성 ↑ + 코딩 도 더 간단

but, 순환적 코드의 약점: 실행 시간!

but... 순환을 반드시 사용해야 하는 경우가 있으니까... 순환을 반드시 피해야 하는 것은 중요한 개념

p.48 • 순환의 원리.

- 순환의 원리 → 문제를 하나도 해결하지 않고 순환 호출만 하고있는 것은 절어어빌...!
  - 문제의 일부를 해결한 다음, 나머지 문제에 대하여 순환 호출을 하는 것임...!

ex)
```
factorial(int n)
{
  if (n <= 1) return 1;
  else return (n * factorial(n-1))
}
```
(n: 해결된 부분, factorial(n-1): 남아있는 부분.)

• 분할 정복 (divide and conquer) : 주어진 문제를 더 작은 동일한 문제들로 분해하여 해결하는 방법.

→ 순환호출이 일어날 때마다 문제의 크기가 작아진다는 것!
  - 문제의 크기가 점점 작아지면 풀기가 쉬워지고 결국 쉽게 풀기 쉬운 문제가 된다...!!!

p.50 • 순환 알고리즘의 성능.

- 반복 알고리즘과 순환 알고리즘의 성능을 비교해보자. (feat. 팩토리얼)

• 반복 알고리즘의 시간 복잡도 = for을 사용하여 n번 → O(n) ☆

• 순환 알고리즘의 시간 복잡도 = 한번 순환 호출때마다 1번의 곱셈수행, 순환호출은 n회 일어남 → O(n) ☆

(둘 다 동일함.)

⇒ 반복, 순환 알고리즘의 시간 복잡도는 같지만 (O(n)),

순환 호출의 경우 여분의 기억공간에 더필요 + 함수 호출 위해 함수의 매개변수들을 스택에 저장하는 것과 같은 사전 작업이 상당 필요.

⇒ (현실적으로) 수행시간도 더 걸림.

"결론적으로, 순환 알고리즘은 이해하기 쉽다는 점과 짧게 프로그램을 작성할 수 있는 장점이 있는 대신, 수행시간과 기억 공간의 사용에 있어서는 비효율적인 경우가 많다.

순환 호출 시에는 호출이 일어날 때마다 그들만의 활성화 상태를 기억해야 하므로 여분의 기억공간이 필요한 것이다."

팩토리얼 : 반복 > 순환 (효율성)

2.3 거듭제곱값 계산. (순환 > 반복)

p. 50 ~ 53.

• 거듭제곱 (Power) (반복)

```
double power_iteration(double x, int n)
{
  int i;
  double p=1;

  for(i=0; i<n; i++)
    p = p*x;
  return(p);
}
```

• 거듭제곱 (순환).

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

base case

inductive step

x·x한 것을 n/2번 곱한다 ⇒ $x^n = (x \cdot x)^{\frac{n}{2}}$ (n=짝수)

⇒ $x^n = x \cdot (x \cdot x)^{\frac{(n-1)}{2}}$ (n=홀수)

<Recursion>

p. 51

• $power(x,n) = power(x^2, n/2)$ <n=짝수>

$= (x^2)^{n/2}$

$= x^{2(n/2)}$

$= x^n$

• $power(x,n) = x \cdot power(x^2, (n-1)/2)$ <n=홀수>

$= x \cdot (x^2)^{(n-1)/2}$

$= x \cdot x^{(n-1)}$

$= x^n$

p.52 <순환호출 일어나서 복귀하는 순서>

power(2,10)

10 → 짝

1 → power(4,5)

5 → 홀

11 → power(16, 2)

2 → 짝

111 → power(256, 1)

1 → 홀

1111 → power(65536, 0)

1111 ← return 1;

111 ← return 256;

11 ← return 256;

1 ← return 1024;

← return 1024;

p.53 • 거듭제곱 분석.

input data = n

▫ 거듭제곱의 **시간 복잡도 분석.**

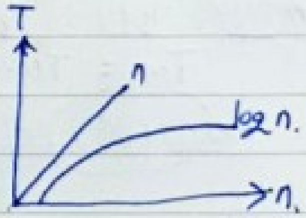
<거듭제곱>
실제는 반복이 쉽지만,
효율성, 즉 시간은
순환이 더 좋다!

- ▫ 반복 : n회의 곱하기 연산 → O(n)
- ▫ 순환 : (log n)회의 곱하기 및 나누기 연산 → O(log n)

전체 연산 횟수 k = log n ☆ └ 한번 부를 때마다 거의 ½씩 줄어드니까.

▫ 거듭제곱 문제는 순환 방법이 더 효율적임.

	반복 (power_iteration)	순환 (power)
시간 복잡도	O(n)	O(log n)
실제 수행 속도.	7.17초	0.47초



2.4 피보나치 (Fibonacci) 수열의 계산.

(수를 연속으로 다른걸 더한걸 나열한 수열)

p.53 ~ p.56.

□ 피보나치 수열 정의 (순환).

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

n=0, n=1] base case

otherwise — Induction step.

보기) 0, 1, 1, 2, 3, 5, 8, 13, 21, ······ ☆ 항상 앞에서 것 2개를 더해야 한다! ☆

(n가지의 전과 그직전 개의 합의 도달)

•순환은 단순한 작성과 높은 가독성이 있지만, but, 똑같은 계산을 몇 번씩 반복한다면 아주 단순한 경우라 할지라도 계산시간이 엄청나게 길어질 수 있다.

□ 피보나치 수열 함수 (순환)

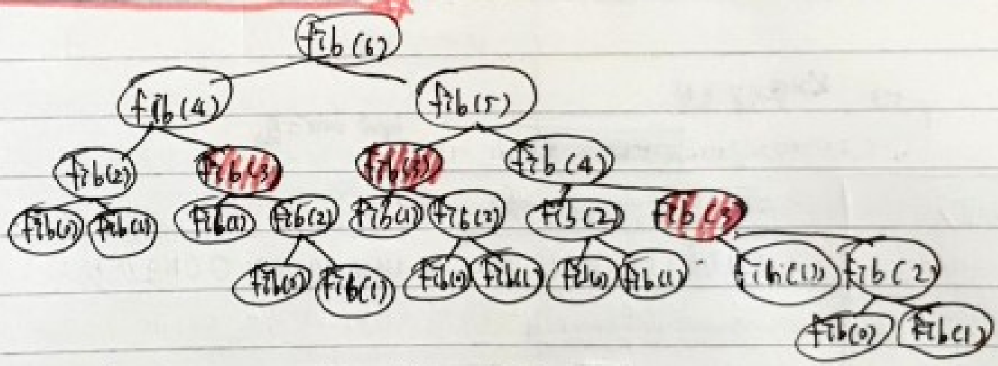
```
int fib(int n)
{
    if(n == 0) return 0;
    if(n == 1) return 1;
    return fib(n-2) + fib(n-1);
}
```

• 피보나치 수열 (순환) 수행.

□ 순환을 사용했을 경우 (아래 그림) 비효율성을 발견할 수 있음.

□ 같은 항을 중복해서 계산함 (ex. fib(6) 계산 시 fib(3)을 3번 계산.)

□ 이러한 현상은 n이 커지면 더 심해짐.



p.55 ✱ 순환적인 피보나치 수열 알고리즘의 시간 복잡도 (순환식으로 표현)

$$T(n) = T(n-1) + T(n-2) + C.$$

$\Rightarrow O(2^n)$ ☆

fib(6) // 1번 호출

fib(5) // 1번 호출

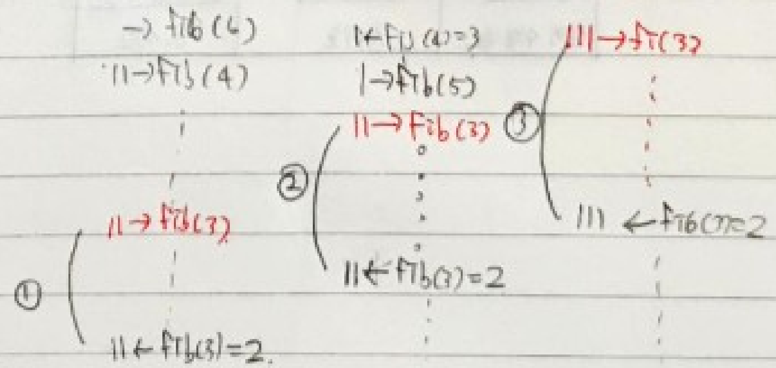
fib(4) // 2번 호출

fib(3) // 3번 호출

fib(2) // 5번 호출

fib(1) // 8번 호출.

$\Rightarrow$ ☆ $O(2^n)$ 복잡도 패턴 ☆



• 피보나치 수열 (반복)

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

  → 그전 그거랑 그거 + 그그거를 더한걸 저장 시켜줘야 하니 3개 필요
  int pp = 0;      //fib(n-2)
  int p = 1;       //fib(n-1)
  int result;      //fib(n)

  for (int i = 2; i <= n; i++) {
    result = p + pp;
    pp = p;   → p가 pp가 되고 (2번째 시점에서)
    p = result;  → 그다음 시점에서 result가 p가 됨
  }
```

✲ 순환, 반복이 좋다, 나쁘다가 아닌

상황에 따라서 효율적인게 달라지니

(경계)

우리는 문제의 "특성을

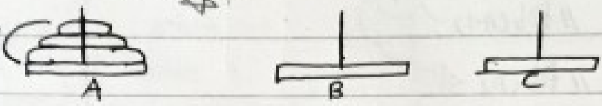
잘 파악해서 더 효율적인 것을 사용한다...!!

2.5 하노이 탑 (Hanoi tower) 문제.

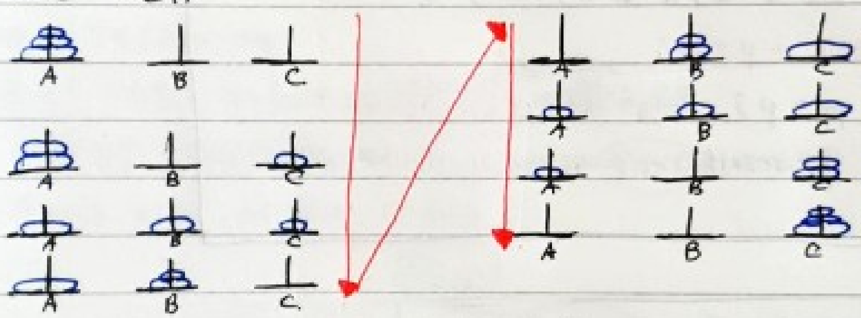
p.56~p.62. • 하노이 탑 (Hanoi tower) — 순환의 파워를 가장 극명하게 보여주는 예제

□ 하노이 탑 문제는 여러 규칙을 지키면서 막대 A에 쌓여 있는 원판 n개를 막대 C로 옮기는 것이다.

① 한 번에 하나의 원판만 이동할 수 있다.
② 맨 위에 있는 원판만 이동할 수 있다.
③ 크기가 작은 원판 위에 큰 원판을 쌓을 수 없다.
④ 중간의 막대를 임시적으로 이용할 수 있으나 앞의 조건들을 지켜야 한다.



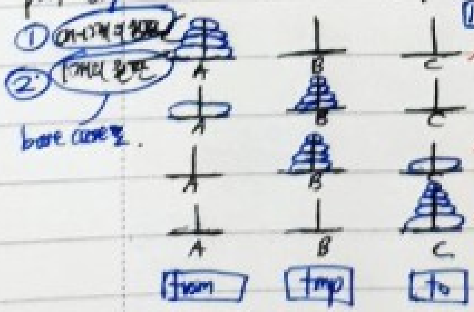
• n=3일 때 풀이.



p.58 * 4개의 경우 조금 더 복잡해짐. 더 나아가 n개의 원판이 있는 경우를 해결하려면 상당히 복잡해짐.
=> 순환적으로 생각하면 쉽게 해결될 문제임. 순환이 일어날수록 문제의 크기가 작아져야 한다.
└ 이동해야하는 디스크의 갯수

Induction step으로

p.59~60 • 일반적인 n일 때 풀이



1. (n-1)개의 원판을 A(=from)에서, C(=to)를 임시 버퍼로 사용하여, B(=tmp)로 옮긴다.
2. 원판 n을, A(=from)에서 C(=to)로 옮긴다. — base case
3. (n-1)개의 원판을 B(=tmp)에서, A(=from)를 임시 버퍼로 사용하여 C(=to)로 옮긴다. — Induction step

이것들을 순환하게 된다!!

1. A의 (n-1)개를 A→B로 (남은 C를 임시로 경유)

5. B의 (n-1)개를 B→C로 (남은 A를 임시로 경유)

p.60-61.

• (n-1)을 때 풀이는 순환을 사용. Recursion!

□ 어떻게 (n-1)개의 원판을 A(=from)에서 C(=to)를 임시버퍼로 사용하여, B(=tmp)로 옮기는가?
또 (n-1)개의 원판을 B(=tmp)에서, A(=from)를 임시 버퍼로 사용하여, C(=to)로 옮기는가?
→ 함수의 재귀적 argument는 바뀌어야 한다.

□ 결국 hanoi의 첫째 매개변수를 (n-1)로 바꾸고, 나머지 매개변수를 from/to/tmp 를 tmp/from/to 로 바꾸어서 순환 호출하면 된다.

A B C
from tmp to

```
// n개의 원판을 from에서, tmp를 임시 버퍼로 사용하여, to로 옮긴다.
void hanoi_tower(int n, char from, char tmp, char to)
{
    if (n == 1) {                                   // Base case.
        원판 1을 from에서 to로 옮긴다.
    } else {                                        // Induction step.
        hanoi_tower(n-1, from, to, tmp);   (A C B)
        원판 n을 from에서 to로 옮긴다.
        hanoi_tower(n-1, tmp, from, to);   (B A C)
    }
}
```

hanoi_tower(n-1, <from>, <tmp>, <to>) ☆ 이 함수!!

• 하노이 탑 (완전)

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


int main(void)
{
  hanoi_tower(4, 'A', 'B', 'C');
}
```

Recursion!

p.61 • 하노이 탑 (순환) 수행

Recursion

// n=4일 때,

원판 1을 A에서 B로 옮긴다.
원판 2를 A에서 C로 옮긴다.
원판 1을 B에서 C로 옮긴다.
원판 3을 A에서 B로 옮긴다.
원판 1을 C에서 A로 옮긴다.
원판 2를 C에서 B로 옮긴다.
원판 1을 A에서 B로 옮긴다.
원판 4를 A에서 C로 옮긴다.
원판 1을 B에서 C로 옮긴다.
원판 2를 B에서 A로 옮긴다.
원판 1을 C에서 A로 옮긴다.
원판 3을 B에서 C로 옮긴다.
원판 1을 A에서 B로 옮긴다.
원판 2를 A에서 C로 옮긴다.
원판 1을 B에서 C로 옮긴다.

3개의 원판을 A → B로 (C 경유)

① (4-1)개의 원판을 A (=from)에서,
C(=to)를 임시 버퍼로 사용하여, B(=tmp)로 옮긴다.

② 1개의 원판 4를, A (=from)에서
C (=to)로 옮긴다.

③ (4-1)개의 원판을 B(=tmp)에서,
A (=from)을 임시 버퍼로 사용하여,
C (=to)로 옮긴다.

5번의 3단계로 옮기게 된다!!!

3개의 원판을 B에서 C로 (A는 경유)

n=3일 때

① ② ③ ① ② ③

Q : n=4일때 하노이 타워를 푸르면?

A : n=3인게 한 번 불려지고,
그게 return하고!
또 n=3인게 한 번 불려지고!
그놈이 또 return하고!
그래서 n=4인게 return이 된다!!!

결국 Tip=> 하노이 탑의 원판 이동 최소는 2^4-1 이다.

p.62 • 반복적인 형태로 바꾸기 어려운 순환

① return n*factorial(n-1);
② return factorial(n-1)*n;

- 꼬리 순환 (tail recursion)은 ①처럼 순환 호출이 순환 함수의 맨 끝에서 이루어진 형태의 순환이다.
  └ 꼬리 순환의 경우 알고리즘을 쉽게 반복적인 형태로 변환이 가능.
- but, ②와 같은 머리 순환 (head recursion)의 경우, 하노이 탑 문제처럼 여러 군데에서 자기 자신을 호출하는 경우 (multi recursion)는 쉽게 반복적인 코드로 바꿀 수 없다.
  (명시적 스택을 만들어서 순환을 simulation 할 수도 있음).

→ 동일한 알고리즘을 꼬리 순환과 머리 순환 양쪽으로 표현할 수 있다면...
당연히 "꼬리 순환"으로 작성!!! ☆