CSE2010 자료구조론

Week 2: Recursion

ICT융합학부 한진영

순환(Recursion)

- 알고리즘이나 함수가 수행 도중에 자기 자신을 다시 호출하여 문제를 해결하는 방법
  - 정의자체가 순환적으로 되어 있는 경우에 적합
- 함수의 순환(재귀)호출(recursive call) 사용
  - 문제 또는 사용되는 자료구조가 재귀적으로 정의되어 있을 때
  - 문제 해결이 용이, 알고리즘 정확성 증명이 용이
  - 시간, 공간 사용이 비효율적임

순환(Recursion) 예

- 팩토리얼

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

- 피보나치 수열

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

- 이항계수

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

- 하노이탑, 이진탐색, ...

팩토리얼 구현

- 팩토리얼 정의

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

- 팩토리얼 구현

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

팩토리얼 호출 순서

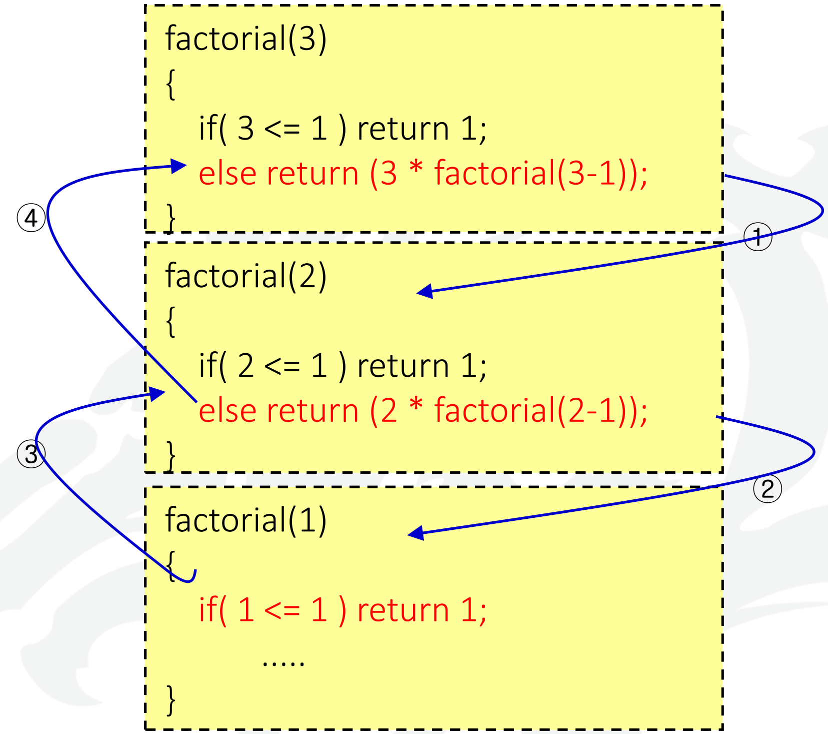
- factorial(3)?

factorial(3) = 3 * factorial(2)

= 3 * 2 * factorial(1)

= 3 * 2 * 1

= 6



순환 알고리즘의 구조

- 순환 알고리즘은 다음의 부분을 포함
  - 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 호출 멈추는 부분이 없다면?
  - 시스템 오류가 발생할 때까지 무한정 호출

순환 vs. 반복

- 컴퓨터에서의 되풀이
  - “순환(recursion)”
    - 순환적인 문제에서는 자연스러운 방법
    - 함수 호출의 오버헤드
  - “반복(iteration)”
    - 수행속도가 빠름
    - 순환적인 문제에 대해서는 프로그램 작성이 어려울 수 있음
- 대부분의 순환은 반복으로 바꾸어 작성할 수 있음

팩토리얼 함수의 반복구현

- 팩토리얼 함수 반복적 정의

$$n! = 1 \qquad \text{if} \quad n = 0$$

$$n! = \underbrace{n \times (n-1) \times (n-2) \times \ldots \times 1}_{\text{loop}} \qquad \text{if} \quad n > 0$$

```
int factorial_iter(int n)
{
    int i, f = 1;
    for(i=n; i>-; i--)
            f = f * i;
    return(f);
}
```

복잡도 비교

- 공간 복잡도

$$S_I(n) = \Theta(1)$$

$$S_R(n) = \Theta(n)$$

- 시간 복잡도

$$T_I(n) = n = \Theta(n)$$

$$\left.\begin{aligned}
T_R(0) &= 0 \\
T_R(n) &= 1 + T_R(n-1) \\
&= 1 + 1 + T_R(n-2) \\
&= 1 + 1 + 1 + T_R(n-3) \\
&\cdots \\
&= \underbrace{1 + 1 + \ldots + 1}_{} + T_R(0)
\end{aligned}\right\} n$$

$$\begin{aligned}
&= n \\
&= \Theta(n)
\end{aligned}$$

거듭제곱 프로그래밍(1)

- 숫자 x의 n제곱값을 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 프로그래밍(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);

즉 n이 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 n이 홀수이면 다음과 같이 계산하는 것이다.

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

```
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);
}
```

시간 복잡도 비교

- 순환적인 방법
  - 만약 n이 2의 거듭 제곱인 2^k 이라고 가정하면 다음과 같이 문제의 크기가 줄어듬
  - n = 2^k -> k=logn

$$\underbrace{2^k \rightarrow 2^{k-1} \rightarrow \cdots 2^2 \rightarrow 2^1 \rightarrow 2^0}_{k=\log n}$$

- 반복적인 방법과의 비교

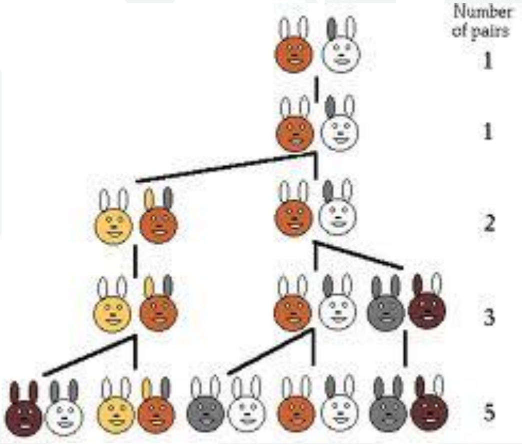
	반복적인 함수 slow_power	순환적인 함수 power
시간복잡도	O(n)	O(logn)

이 경우에, 순환적인 방법이 반복적인 방법보다 시간 복잡도 측면에서 더 효율적임!

토끼 번식 문제

문제	**토끼장에 바로 막 태어난 암수 토끼 한쌍을 넣었다. 다음 가정 하에 일년 후 이 토끼장에는 몇쌍이 있겠는가?**
가정	**(1) 토끼 한쌍은 새로 태어난지 한달후 성숙해진다.** **(2) 토끼 한쌍은 성숙해진지 한달후부터 매달 새로운 암수 토끼 한쌍을 낳는다.** **(3) 토끼는 죽지 않는다.**

- 여섯달 후의 토끼의 쌍 수는?
  - F_i : i번째 달의 토끼 쌍수
  - r_i : i번째 태어난 토끼 쌍
  - R_i : 성숙한 r_i



$F_0 = 0$

$F_1 = 1$ (최초의 토끼한 쌍 r_1)

$F_2 = 1$ (R_1)

$F_3 = 2$ $(R_1, (R_1 \rightarrow) r_2)$

$F_4 = 3$ $(R_1, R_2, (R_1 \rightarrow) r_3)$

$F_5 = 5$ $(R_1, R_2, R_3, (R_1 \rightarrow) r_4, (R_2 \rightarrow) r_5)$

$F_6 = 8$ $(R_1, R_2, R_3, R_4, R_5, (R_1 \rightarrow) r_6, (R_2 \rightarrow) r_7, (R_3 \rightarrow) r_8)$

피보나치 수열

- i번째 달의 토끼 쌍의 수

  = $(i-1)$ 번째 달의 토끼 쌍 + i 번째 달에 새로 태어난 토끼 쌍

  = $(i-1)$ 번째 달의 토끼 쌍 + $(i-1)$ 번째 달의 성숙한 토끼 쌍

  = $(i-1)$ 번째 달의 토끼 쌍 + $(i-2)$ 번째 달의 토끼 쌍

$$F_i = F_{i-1} + F_{i-2}$$ 피보나치 수열

- 피보나치 수열 정의

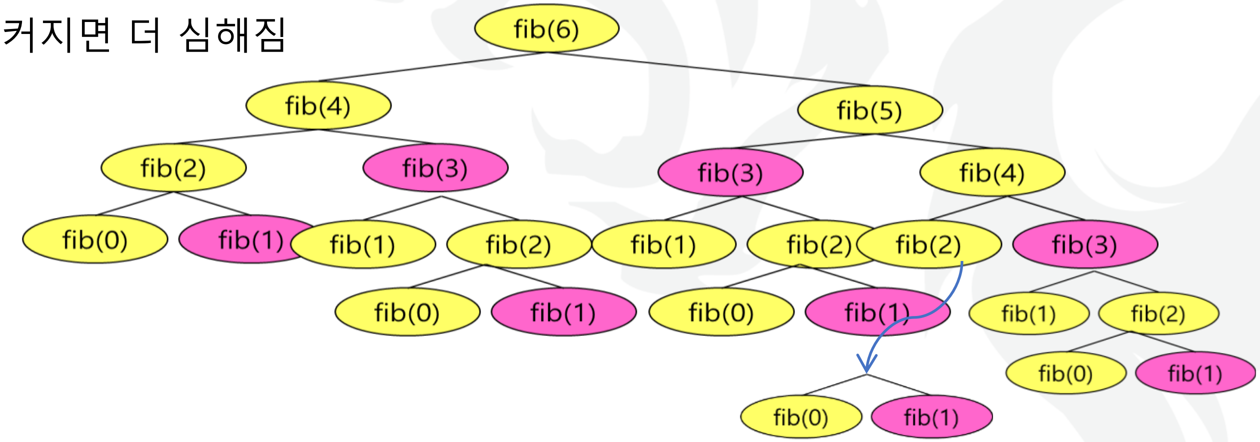
$$fib(n) \begin{cases} 0 & n = 0 \\ 1 & n = 1 \\ fib(n-2) + fib(n-1) & otherwise \end{cases}$$

피보나치 수열의 순환적 계산 방법

- 순환적인 방법

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

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



피보나치 수열의 반복적 계산

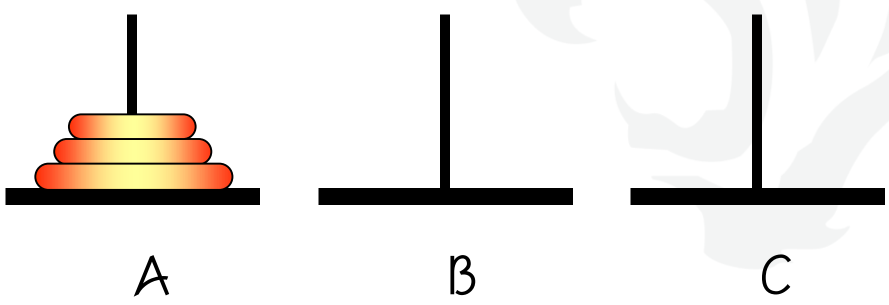
- 반복 구조를 사용한 구현

```
int 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;
	}
}
```

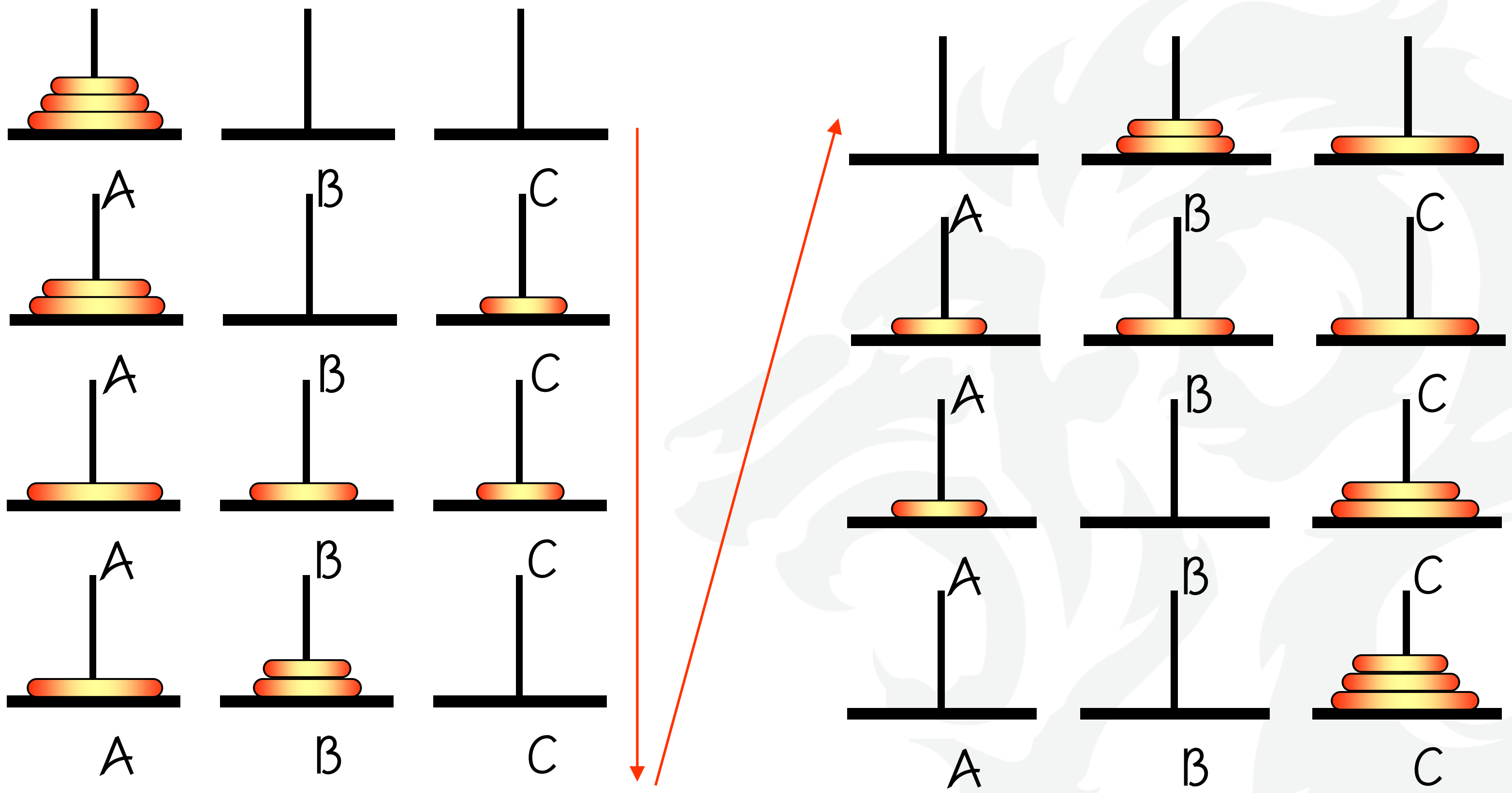
이 경우에, 반복적인 방법이 순환적인 방법보다 더 효율적임!

하노이탑 문제

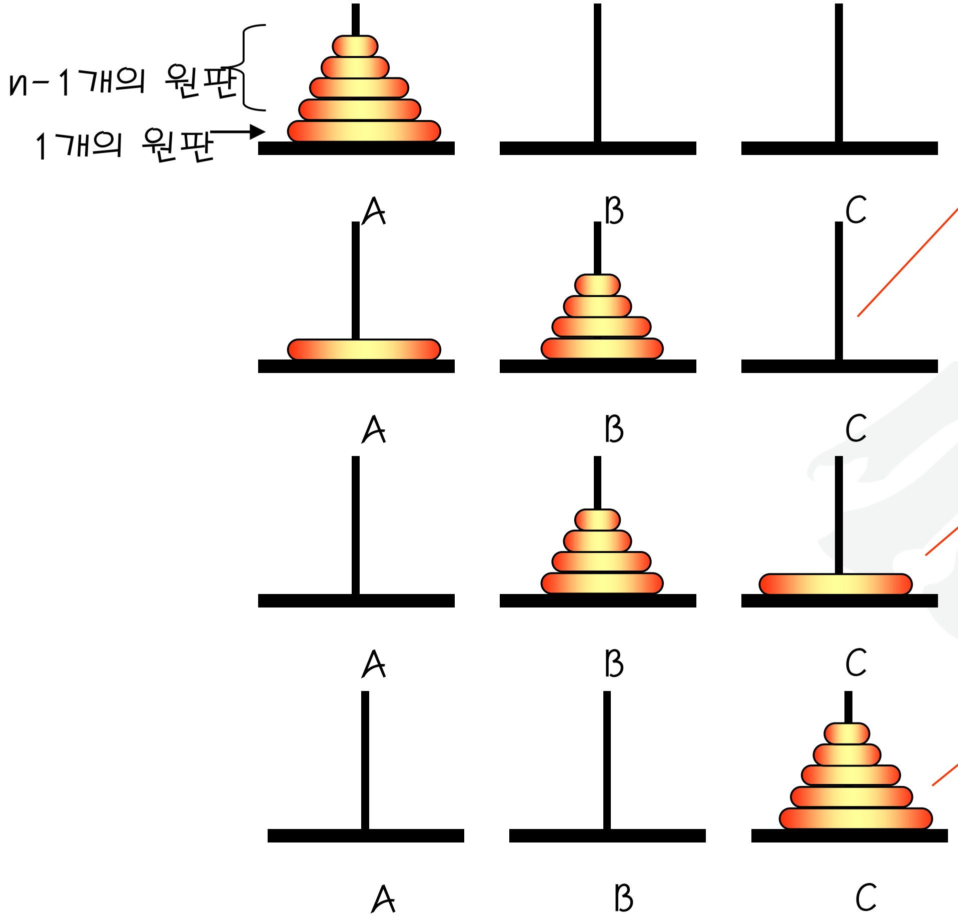
- 하노이탑 문제: 막대 A에 쌓여있는 원판 n개를 막대 C로 옮기는 것
- 다음의 조건을 지켜야 함
  - 한 번에 하나의 원판만 이동할 수 있음
  - 맨 위에 있는 원판만 이동할 수 있음
  - 크기가 작은 원판 위에 큰 원판이 쌓일 수 없음
  - 중간의 막대를 임시적으로 이용할 수 있으나 앞의 조건들을 지켜야 함



하노이탑 문제: n=3인 경우



하노이탑 문제: 일반적인 경우



C를 임시버퍼로 사용하여 A에 쌓여있는 n-1개의 원판을 B로 옮김

A의 가장 큰 원판을 C로 옮김

A를 임시버퍼로 사용하여 B에 쌓여있는 n-1개의 원판을 C로 옮김

하노이탑 문제: 순환방법 도입

- 어떻게 n-1개의 원판을 A에서 B로, 또 B에서 C로 이동하는가?
  - (힌트) 우리의 원래 문제는 n개의 원판을 A에서 C로 옮기는 것임
- 작성하고 있는 함수의 파라미터를 n-1로 바꾸어 순환 호출!

```
// 막대 from에 쌓여있는 n개의 원판을 막대 tmp를 사용하여 막대 to로 옮긴다.
void hanoi_tower(int n, char from, char tmp, char to)
{
   if (n==1){
      from에서 to로 원판을 옮긴다.
   }
   else{
      (1) hanoi_tower(n-1, from, to, tmp); //from의 맨 밑에 있는 원판을 제외한
     나머지 원판들을 tmp로 옮긴다.
      (2) from에 있는 한 개의 원판을 to로 옮긴다.
      (3) hanoi_tower(n-1, tmp, from, to); // tmp의 원판들을 to로 옮긴다.
   }
}
```

하노이탑 문제의 구현

- 기본 아이디어: n-1개의 원판을 A에서 B로 옮기고, n번째 원판을 A에서 C로 옮긴 다음, n-1개의 원판을 B에서 C로 옮김

```
#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');
}
```

Week 2: Recursion