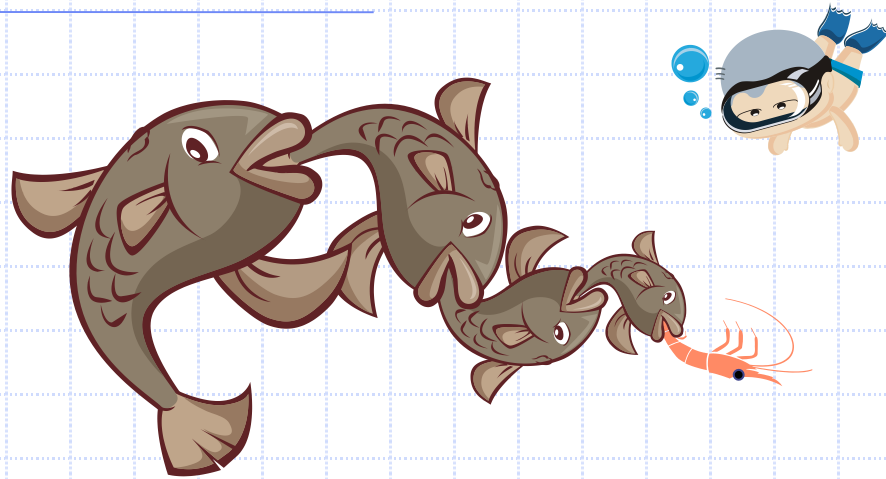


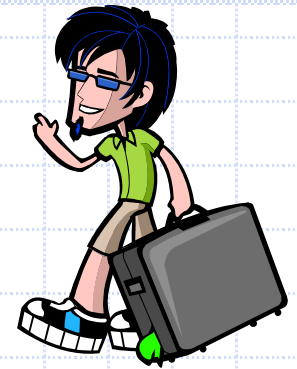
# 재귀



# Outline

- ◆ 2.1 재귀 알고리즘
- ◆ 2.2 재귀의 작동원리
- ◆ 2.3 재귀의 기본 규칙
- ◆ 2.4 응용문제

# 재귀 알고리즘

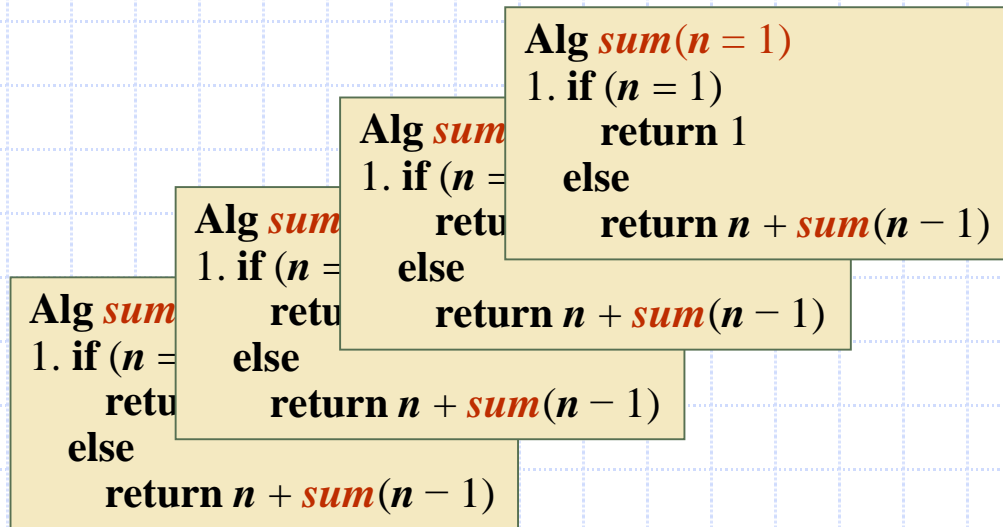


- ◆ 알고리즘 자신을 사용하여 정의된 알고리즘을 **재귀적**(recursive)이라고 말한다
  - **비재귀적**(nonrecursive) 또는 **반복적**(iterative) 알고리즘과 대조
- ◆ 재귀의 요소
  - **재귀 케이스**(recursion):  
차후의 재귀호출은 **작아진** 부문제들(subproblems)을 대상으로 이루어진다
  - **베이스 케이스**(base case):  
부문제들이 충분히 작아지면, 알고리즘은 재귀를 사용하지 않고 이들을 직접 해결한다

```
Alg sum(n)  
1. if (n = 1)           {base case}  
    return 1  
else                     {recursion}  
    return n + sum(n - 1)
```

# 작동 원리

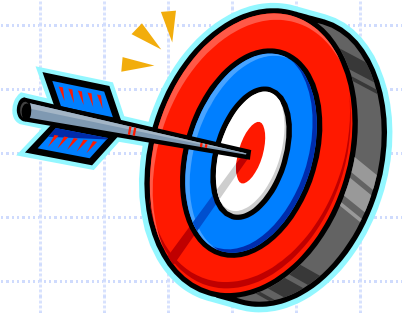
- ◆ 보류된 재귀호출(즉, 시작했지만 완료하지 않고 대기중인 호출들)을 위한 변수들에 관련된 저장/복구는 컴퓨터에 의해 자동적으로 수행



Parameters: *n* = 2  
Local variables:  
Return addr: ###

Parameters: *n* = 3  
Local variables:  
Return addr: ###

Parameters: *n* = 4  
Local variables:  
Return addr: ###



# 기본 규칙

## ◆ 베이스 케이스

- 베이스 케이스를 항상 가져야 하며, 이 부분은 재귀 없이 해결 가능

## ◆ 진행 방향

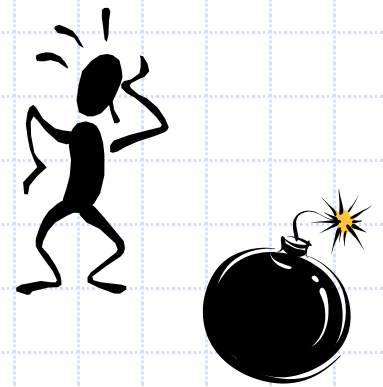
- 재귀적으로 해결되어야 할 경우, 재귀호출은 항상 베이스 케이스를 향하는 방향으로 진행

## ◆ 정상작동 가정

- 모든 재귀호출이 제대로 작동한다고 가정!

## ◆ 적절한 사용

- 꼭 필요할 때만 사용 - 저장/복구 때문에 성능 저하



# 나쁜 재귀

## ◆ 잘못 설계된 재귀

- 베이스 케이스: 없음
  - ◆ 예: `sum1`
- 재귀 케이스: 도달 불능 - 즉, 베이스 케이스를 향해 재귀하지 않음
  - ◆ 예: `sum2`

## ◆ 나쁜 재귀 사용의 영향

- 부정확한 결과
- 미정지(nontermination)
- 저장을 위한 기억장소 고갈

Alg `sum1(n)`

1. return  $n + \text{sum1}(n - 1)$

Alg `sum2(n)`

1. if ( $n = 1$ ) {base case}

    return 1

else {recursion}

    return  $n + \text{sum2}(n + 1)$

# 잘 설계된 재귀의 예

- ◆ `printDigits`는 재귀적  
`rPrintDigits`를 구동하여  
양의 정수를 한 라인에  
한 숫자씩 인쇄

*Enter a number*

3408

3

4

0

8

```
Alg printDigits()           {driver}  
1. write("Enter a number")  
2.  $n \leftarrow \text{read}()$   
3. if ( $n < 0$ )                {error check}  
    write("Negative number!")  
    else  
        rPrintDigits( $n$ )      {initial call}
```

```
Alg rPrintDigits( $n$ )          {recursive}  
1. if ( $n < 10$ )                {base case}  
    write( $n$ )  
    else                        {recursion}  
        rPrintDigits( $n/10$ )  
        write( $n \% 10$ )
```

# 응용문제: 재귀적 곱하기와 나누기

- A.  $a$ 와  $b$ 의 곱을 계산하는 재귀 알고리즘 `product(a, b)`를 작성하라
- B.  $a$ 를  $b$ 로 나눈 나머지를 계산하는 재귀 알고리즘 `modulo(a, b)`를 작성하라
- C.  $a$ 를  $b$ 로 나눈 몫을 계산하는 재귀 알고리즘 `quotient(a, b)`를 작성하라

## ◆ 주의

- 의사코드로 작성
- $a$ 와  $b$ 는 양의 정수
- 덧셈과 뺄셈을 제외한 산술연산자 사용 불가



# 해결

**Alg** *product*( $a, b$ )

**input** positive integer  $a, b$

**output** product of  $a$  and  $b$

```
1. if ( $b = 1$ )           {base case}
    return  $a$ 
else                     {recursion}
    return  $a + \text{product}(a, b - 1)$ 
```

**Alg** *modulo*( $a, b$ )

**input** positive integer  $a, b$

**output**  $a \% b$

```
1. if ( $a < b$ )           {base case}
    return  $a$ 
else                     {recursion}
    return  $\text{modulo}(a - b, b)$ 
```

**Alg** *quotient*( $a, b$ )

**input** positive integer  $a, b$

**output**  $a/b$

```
1. if ( $a < b$ )           {base case}
    return 0
else                     {recursion}
    return  $1 + \text{quotient}(a - b, b)$ 
```



# 응용문제: 하노이탑

## ◆ 하노이탑(towers of Hanoi) 문제

- 세 개의 말뚝:  $A, B, C$
- 초기 상황: 직경이 다른  $n > 0$  개의 원반이  $A$ 에 쌓여 있음
- 목표: 모든 원반을  $A$ 로부터  $C$ 로 옮김
- 이동 순서를 "move from  $x$  to  $y$ " 형식으로 인쇄할 것

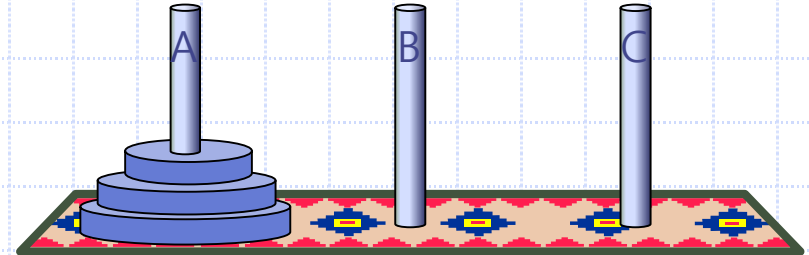
## ◆ 조건

- 한번에 한 개의 원반만을 이동
- 언제라도 직경이 큰 원반을 작은 원반 위에 놓지 말 것
- 남은 말뚝을 보조 말뚝으로 사용 가능



# 응용문제: 하노이탑 (conti.)

- ◆ 일반적으로,  $n$ 개의 원반에 대해  $2^n - 1$ 회의 이동이 필요하다
- ◆  $n = 1$ 인 경우, 1회의 이동
- ◆  $n = 2$ 인 경우, 3회의 이동
- ◆  $n = 3$ 인 경우, 7회의 이동
- ◆  $n = 64$ 인 경우,  $2^{64} - 1 = 1.844 \times 10^{19}$ 회의 이동
  - 1회 이동에 1초 걸린다고 가정하면, 이는 약 5,849억년!
  - "The end of the world"
  - $\approx$  태양의 수명



*For  $n = 3$ ,*

*move from A to C  
move from A to B  
move from C to B  
move from A to C  
move from B to A  
move from B to C  
move from A to C*

# 해결: 하노이탑

◆ **hanoi**는 아래의  
매개변수들을  
사용하여 재귀적  
**rHanoi**를 구동

- **n**: 이동해야 할 원반 수
- **from**: 출발 말뚝
- **aux**: 보조 말뚝
- **to**: 목표 말뚝

◆ 이중재귀(double recursion)의 한 예

Alg ***hanoi***(***n***)

1. ***rHanoi***(***n***, 'A', 'B', 'C') {initial call}
2. **return**

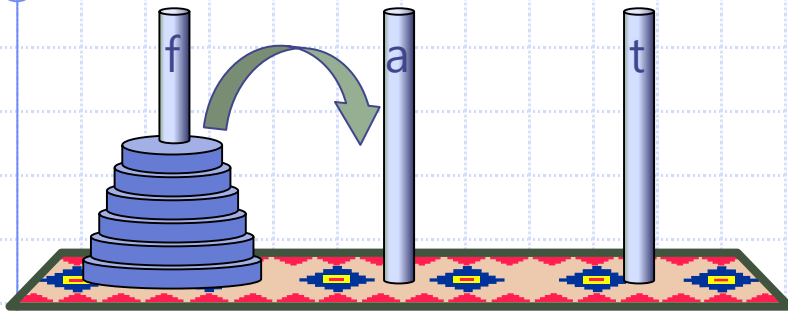
Alg ***rHanoi***(***n, from, aux, to***) {recursive}

**input** integer ***n***, peg ***from, aux, to***

**output** move sequence

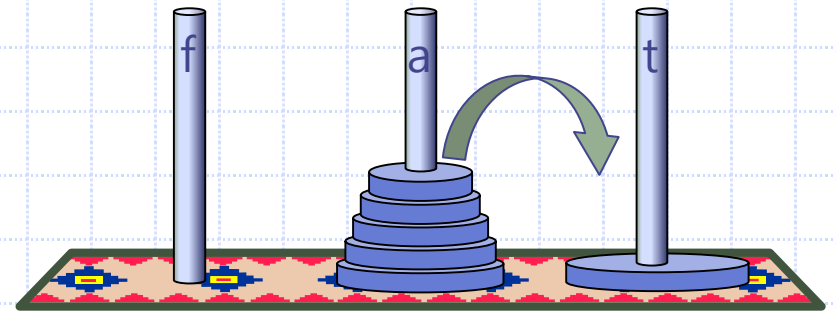
1. **if** (***n*** = 1) {base case}  
    **write**(“*move from*”, ***from***, “*to*”, ***to***)  
    **return**
2. ***rHanoi***(***n*** - 1, ***from, to, aux***) {recursion}
3. **write**(“*move from*”, ***from***, “*to*”, ***to***)
4. ***rHanoi***(***n*** - 1, ***aux, from, to***) {recursion}
5. **return**

# 해결: 하노이탑 (conti.)



(a)

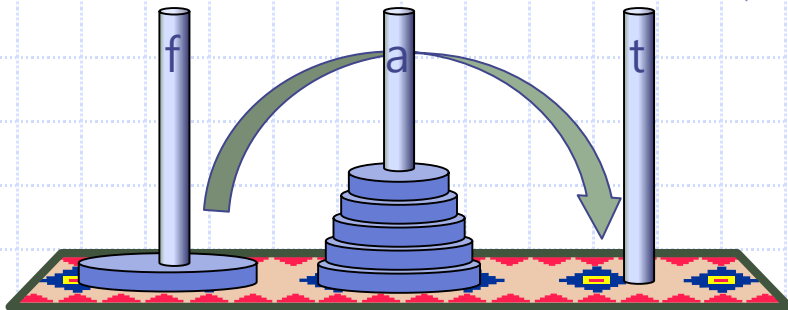
2.  $rTowers(n-1, from, to, aux)$



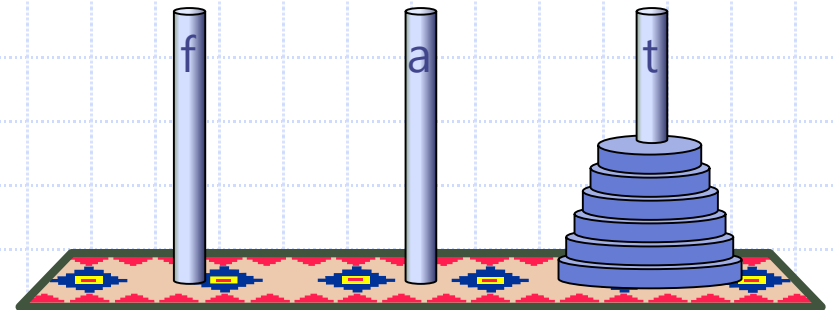
(c)

4.  $rTowers(n-1, aux, from, to)$

3. *move from from to to*



(b)



(d)