재귀 (Recursion)

강지훈
jhkang@cnu.ac.kr
충남대학교 컴퓨터융합학부

재귀적(Recursive) 이란?

점화식 (Recurrence Equation)

예: 계승 (Factorial)

- a_0 = 1 이고,
  0 보다 큰 정수 N 에 대해 a_N = N * a_{N-1} 이다.

- 수식으로 표현된 점화식

$$a_N = \begin{cases} 1 & \text{if } N = 0 \\ N \times a_{N-1} & \text{if } N \geq 1 \end{cases}$$

- a_3 의 값은?

점화식의 계산

- a_3 의 값을 계산하는 방법은?
  - $a_1 = 1 * a_0 = 1 * 1 = 1$
  - $a_2 = 2 * a_1 = 2 * 1 = 2$
  - $a_3 = 3 * a_2 = 3 * 2 = 6$

$$a_N = \begin{cases} 1 & \text{if } N = 0 \\ N \times a_{N-1} & \text{if } N \geq 1 \end{cases}$$

- a_3 의 계산의 또 다른 표현은?
  - $$\begin{aligned} a_3 &= 3 * a_2 \\ &= 3 * (2 * a_1) \\ &= 3 * (2 * (1 * a_0)) \\ &= 3 * (2 * (1 * 1)) \\ &= 3 * (2 * 1) = 3 * 2 = 6 \end{aligned}$$

예: 점화식

- Factorials

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

- Fibonacci Numbers

$$F_n = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ F_{n-1} + F_{n-2} & \text{if } n \geq 2 \end{cases}$$

- Binomial Coefficients

$${}_nC_m = \begin{cases} 1 & \text{if } m = 0 \text{ or } n = m \\ {}_{n-1}C_m + {}_{n-1}C_{m-1} & \text{if } n > m > 0 \end{cases}$$

점화식이란?

- Factorial 문제의 관찰로부터:
  - 형태가 축소적으로 반복:
    - N! 의 값을 알고자 할 때, (N-1)! 의 값을 안다면 그 값에 N 을 곱하면 우리는 N! 의 값을 얻을 수 있다.
    - 그러므로 N! 의 답을 얻기 위해서는 (N-1)! 의 답을 얻으면 된다.
  - 가장 간단한 형태의 답은 쉽게 얻을 수 있다:
    - 문제의 크기를 계속 줄여 간다면 언제인가는 답을 쉽게 얻을 수 있는 크기까지 줄여갈 수 있을 것이다. N! 의 경우 N 이 0 이 되면 결국 0! 의 값을 계산해야 하는데, 우리는 그 값이 1 임을 알고 있다.

- 이와 같은 내용을 하나의 식으로 표현할 수 있다: 우리는 그러한 식을 점화식 (recurrence equation) 이라고 한다.

❑ (해결 가능한) 점화식의 특징

- 점화식 (Recurrence Equation) 은:
  - 현재의 모습을 정의할 때, 현재의 것과 **모습은 동일**하되 **크기는 작아진 것**으로 표현된다.
    (크기만 줄어들었지, 풀어야 할 문제의 형태는 동일하다)
  - **크기가 가장 작은 경우**에 대해, **구체적인 모습**, 즉, 답을 쉽게 얻을 수 있다.
- 예제
  - **Factorial**
    - **n!** 은 (**n-1**)**!** 로 표현된다.
    - **0**! 은 **1** 이다.
  - **피보나치 수열**
    - **n** 번째 **피보나치 수**는 (**n-1**) 번째 **피보나치 수**와 (**n-2**) 번째 **피보나치 수**의 합으로 표현된다.
    - **0** 번째 수는 **0** 이고, **1** 번째 수는 **1** 이다.
  - **이항계수:** ${}_{n}C_{m}$
    - **n** 차의 **이항계수**는 (**n-1**) 차의 **이항계수**의 합으로 표현된다.
    - **0** 번째 항 (m=0) 이거나 **마지막** 항 (n=m) 이면, 그 값은 **1** 이다.

❑ 재귀적 정의의 일반적 특성: 계승

- 크기를 줄여가다 보면, 반드시 쉽게 답을 얻을 수 있는 상태에 도달한다. (재귀의 탈출)
- 이 예제에서는 0! 의 값이 1 인 것은 계승(factorial)의 정의로부터 바로 얻을 수 있는 값이다.

$$a_N = \begin{cases} 1 & \text{if } N = 0 \\ N \times a_{N-1} & \text{if } N \geq 1 \end{cases}$$

- 원래 문제보다 크기가 줄어들어야 한다.
- 이 예제에서는 n! 의 문제가 (n-1)! 의 값을 구하는 문제로 크기가 줄어들었다.

❑ 재귀적 정의의 일반적 특성: 이항계수

- 크기를 줄여가다 보면, 반드시 쉽게 답을 얻을 수 있는 상태에 도달한다. (재귀의 탈출)
- 이 예제에서는 (m=0) 이거나 (n=m) 이면, 이항계수(Binomial Coefficient) 의 정의로부터 바로 얻을 수 있는 값이다.

$$
{}_{n}C_{m} = \begin{cases} 1 & \text{if } m = 0 \text{ or } n = m \\ {}_{n-1}C_{m} + {}_{n-1}C_{m-1} & \text{if } n > m > 0 \end{cases}
$$

- 원래 문제보다 크기가 줄어들어야 한다.
- 이 예제에서는 ${}_{n}C$ 의 문제가 ${}_{n-1}C$ 의 값을 구하는 문제로 크기가 줄어들었다.

참고: 이항 계수 (Binomial Coefficients)

- 다음과 같은 다항식이 있다:
  $(1+x)^n = a_0 + a_1 \cdot x^1 + a_2 \cdot x^2 + a_3 \cdot x^3 + \ldots + a_n \cdot x^n$
- 그러면, 각 항의 계수 a_m 은:
  $a_m = {}_nC_m$
  즉, $(1+x)^n = {}_nC_0 + {}_nC_1 \cdot x^1 + {}_nC_2 \cdot x^2 + {}_nC_3 \cdot x^3 + \ldots + {}_nC_n \cdot x^n$

- 각 차수 별로 다항식의 계수를 살펴보면:

$(1+x)^0 =$ 1

$(1+x)^1 =$ $1 + 1 \cdot x^1$

\ /

$(1+x)^2 =$ $1 + 2 \cdot x^1 + 1 \cdot x^2$

\ / \ /

$(1+x)^3 =$ $1 + 3 \cdot x^1 + 3 \cdot x^2 + 1 \cdot x^3$

\ / \ / \ /

$(1+x)^4 =$ $1 + 4 \cdot x^1 + 6 \cdot x^2 + 4 \cdot x^3 + 1 \cdot x^4$

\ / \ / \ / \ /

$(1+x)^5 =$ $1 + 5 \cdot x^1 + 10 \cdot x^2 + 10 \cdot x^3 + 5 \cdot x^4 + 1 \cdot x^5$

N차의 항의 계수 값은, 이전 차수인 (N-1)차의 두 항의 계수 값의 덧셈으로 계산이 가능

${}_4C_2 = 6$
${}_4C_3 = 4$
${}_5C_3 = 10$
${}_5C_3 = {}_4C_2 + {}_4C_3$

재귀 (Recursion)

- Recurrence:
  - If there is a recurrence of something, it happens again.
- Recursion:
  - 동일한 모습이 되풀이 됨.
  - “Recurrence Equation 은 recursion 을 포함하고 있다.”
- Recursive:
  - Recursion 과 관계되거나 Recursion 을 포함하고 있는 상태.
- Recursive function:
  - 동일한 모습이 되풀이 되는 함수?
  - 즉, (크기만 줄어든 채로) 자기 자신을 call 하는 함수

재귀적 문제 풀이

- 어떤 복잡한 문제들은 재귀 (recursion) 를 사용하여 아주 간결하게 표현할 수 있다.
- 그러한 문제들은 그 자체가 재귀적 (recursive) 으로 정의된다.
  - 즉, 원래 문제 (P_n) 보다 크기가 작은 같은 형태의 문제 (P_{n-1}) 의 답 (S_{n-1}) 을 안다고 할 때,
  - 이 S_{n-1} 으로부터 원래 문제의 답 (S_n) 을 알아 낼 수 있다면:
  - 우리는 그러한 문제를 재귀적으로 풀 수 있다.

재귀적 문제 풀이는 수학 문제에만?

- 주어진 n 개의 값으로부터 최대값을 찾기
  - 최소값, 홀수 숫자의 개수, 전체 합, 등.
- 주어진 문자열을 역순으로 인쇄하기
- 하노이 탑 (Tower of Hanoi) 문제
- 드래곤 커브 (Dragon curves), 힐버트 커브 (Hilbert Curves), 등
- 그 밖에도 많이 있다

재귀 함수
(Recursive Function)

❑ 점화식은 모습 그대로 재귀함수로!

```
public int  fact (int n)
{
    if ( n == 0 )
        return ( 1 ) ;  // n! = 1 if (n==0)
    else
        return ( n*fact(n-1) ) ;  // n! = n*(n-1)!  if (n>= 1)
}  // end of fact()
```

$$fact(n) = \begin{cases} 1 & \text{if } n = 0 \\ n \times fact(n-1) & \text{if } n \geq 1 \end{cases}$$

$$a_n = \begin{cases} 1 & \text{if } n = 0 \\ n \times a_{n-1} & \text{if } n \geq 1 \end{cases}$$

❑ 프로그램으로서의 재귀함수의 작동 원리

```
public static void  main()
{
    int    nFact ;
    int    n = 2 ;

    nFact = fact (2) ;
    ......
}

public int  fact (int n)
{
    if (n==0)
        return ( 1 ) ;
    else
        return ( n * fact(n-1) ) ;
}  // end of fact()
```

?

```
nFact = fact(2) ; // a call to fact() for 2!
```

```
nFact = fact(2) ; // a call to fact() for 2!
```

```
public int fact (int n /2/ )
{
    if (n /2/ == 0)
        return (1) ;
    else
        return ( n /2/ * fact ((n-1) /1/ ) ) ;
}
```

```
nFact = fact(2) ; // a call to fact() for 2!
```

```
public int fact (int n /2/ )
{
    if (n /2/ == 0)
        return (1) ;
    else
        return ( n /2/ * fact ((n-1) /1/ ) ) ;
}
```

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

```
nFact = fact(2) ; // a call to fact() for 2!
```

```
public int fact (int n /2/ )
{
      if (n /2/ == 0)
            return (1) ;
      else
            return ( n /2/ * fact ((n-1) /1/ ) ) ;
}
```

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

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

```
nFact = fact(2) ; // a call to fact() for 2!
```

```
public int fact (int n /2/ )
{
      if (n /2/ == 0)
            return (1) ;
      else
            return ( n /2/ * fact ((n-1) /1/ ) ) ;
}
```

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

```
public int fact (int n /0/ )
{
      if (n /0/ == 0)
            return (1) ; [1]
      else
            return ( n * fact ((n-1) ) ) ;
}
```

```
nFact = fact(2) ; // a call to fact() for 2!
```

```
public int fact (int n /2/ )
{
    if (n /2/ == 0)
        return (1) ;
    else
        return ( n /2/ * fact ((n-1) /1/ ) ) ;
}
```

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

```
public int fact (int n /0/ )
{
    if (n /0/ == 0)
        return (1) ; [1]
    else
        return ( n * fact ((n-1) ) ) ;
}
```

```
nFact = fact(2) ; // a call to fact() for 2!
```

```
public int fact (int n /2/ )
{
    if (n /2/ == 0)
        return (1) ;
    else
        return ( n /2/ * fact ((n-1) /1/ ) ) ;
}
```

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

```
public int fact (int n /0/ )
{
    if (n /0/ == 0)
        return (1) ; [1]
    else
        return ( n * fact ((n-1) ) ) ;
}
```

```
nFact = fact(2) ; // a call to fact() for 2!
```

```
public int fact (int n /2/ )
{
        if (n /2/ == 0)
                return (1) ;
        else
                return ( n /2/ * fact ((n-1) /1/ ) ) ;
}
```

```
public int fact (int n /1/ )
{
        if (n /1/ == 0)
                return (1) ;
        else
                return ( n /1/ * fact ((n-1) /0/ ) ) ; [1]
}
```

```
public int fact (int n /0/ )
{
        if (n /0/ == 0)
                return (1) ; [1]
        else
                return ( n * fact ((n-1) ) ) ;
}
```

```
nFact = fact(2) ; // a call to fact() for 2!
```

```
public int fact (int n /2/ )
{
        if (n /2/ == 0)
                return (1) ;
        else
                return ( n /2/ * fact ((n-1) /1/ ) ) ;
}
```

```
public int fact (int n /1/ )
{
        if (n /1/ == 0)
                return (1) ;
        else
                return ( n /1/ * fact ((n-1) /0/ ) ) ; [1]
}
```

```
public int fact (int n /0/ )
{
        if (n /0/ == 0)
                return (1) ; [1]
        else
                return ( n * fact ((n-1) ) ) ;
}
```

```
nFact = fact(2) ; // a call to fact() for 2!
```

```
public int fact (int n /2/ )
{
    if (n /2/ == 0)
        return (1) ;
    else
        return ( n /2/ * fact ((n-1) /1/ ) ) ;
}
```

```
public int fact (int n /1/ )
{
    if (n /1/ == 0)
        return (1);
    else
        return ( n /1/ * fact ((n-1) /0/ ) ) ; [1]
}
```

```
public int fact (int n /0/ )
{
    if (n /0/ == 0)
        return (1) ; [1]
    else
        return ( n * fact ((n-1) ) ) ;
}
```

```
nFact = fact(2) ; // a call to fact() for 2!
```

```
public int fact (int n /2/ )
{
        if (n /2/ == 0)
                return (1) ;
        else
                return ( n /2/ * fact ((n-1) /1/ ) ) ; [2]
}
```

```
public int fact (int n /1/ )
{
        if (n /1/ == 0)
                return (1);
        else
                return ( n /1/ * fact ((n-1) /0/ ) ) ; [1]
}
```

```
public int fact (int n /0/ )
{
        if (n /0/ == 0)
                return (1) ; [1]
        else
                return ( n * fact ((n-1) ) ) ;
}
```

```
nFact = fact(2) ; // a call to fact() for 2!
```

```
public int fact (int n /2/ )
{
    if (n /2/ == 0)
        return (1) ;
    else
        return ( n /2/ * fact ((n-1) /1/ ) ) ; [2]
}
```

```
public int fact (int n /1/ )
{
    if (n /1/ == 0)
        return (1);
    else
        return ( n /1/ * fact((n-1) /0/ ) ) ; [1]
}
```

```
public int fact (int n /0/ )
{
    if (n /0/ == 0)
        return (1) ; [1]
    else
        return ( n * fact((n-1) ) ) ;
}
```

nFact = **fact(2)** ; // 재귀 함수 호출의 최종값 2 를 받는다

```
public int fact (int n /2/ )
{
    if (n /2/ == 0)
        return (1);
    else
        return ( n /2/ * fact ((n-1) /1/ ) ) ; [2]
}
```

```
public int fact (int n /1/ )
{
    if (n /1/ == 0)
        return (1);
    else
        return ( n /1/ * fact((n-1) /0/ ) ) ; [1]
}
```

```
public int fact (int n /0/ )
{
    if (n /0/ == 0)
        return (1) ; [1]
    else
        return ( n * fact((n-1) ) ) ;
}
```

nFact = **fact(2) [2]** ; // 등호의 오른쪽 값은 2 이다

```
public int fact (int n /2/ )
{
    if (n /2/ == 0)
        return (1);
    else
        return ( n /2/ * fact ((n-1) /1/ ) ) ; [2]
}
```

```
public int fact (int n /1/ )
{
    if (n /1/ == 0)
        return (1);
    else
        return ( n /1/ * fact((n-1) /0/ ) ) ; [1]
}
```

```
public int fact (int n /0/ )
{
    if (n /0/ == 0)
        return (1) ; [1]
    else
        return ( n * fact((n-1) ) ) ;
}
```

재귀 함수는 call 마다, 복사본이 정말 생기는가?

- 논리적으로는?
  - 재귀 call 을 할 때 마다 함수의 복사본이 생긴다고 생각하면 정확
  - 특히 디버깅 할 때
- 실제로는?
  - 함수의 코드는 하나만 존재
  - 새로운 함수 call 이 작동하는데 필요한 정보만 새롭게 확보:
    - Activation Record: 매개 변수, 지역 변수, Return Address (call 한 곳으로 돌아가 실행이 시작되어야 하는 곳의 메모리 상의 주소), Return Value, 등
  - Activation Record 의 Stack
- 모든 call 된 함수는 자신의 activation record 를 소유
  - 재귀함수를 포함한 일반적인 모든 함수
  - Call 하는 곳 (caller) 에서 call 당하는 함수 (callee) 의 activation record 를 생성해 주고, 함수가 시작하게 함.

재귀적 구조와 재귀 함수

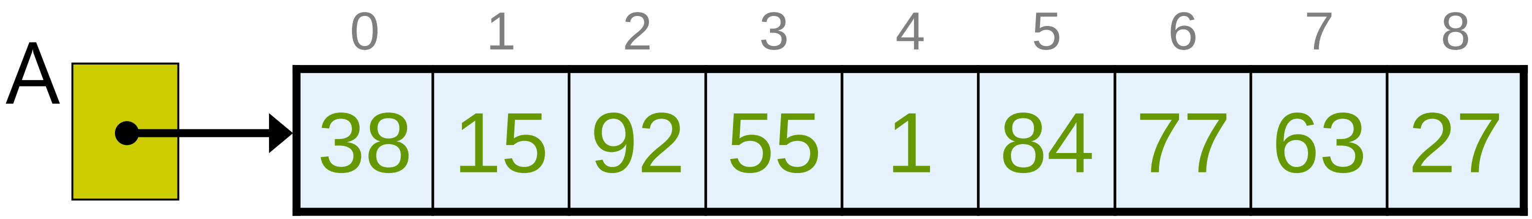
배열에서 최대값 찾기

배열에서 최대값을 찾으려면...

- 가장 단순한 경우는?
  - 배열의 크기가 1 이라면? (즉 원소가 1 개라면?)
- 일반적으로는?
  - 문제의 크기를 줄여보자:
    - 어떻게 크기를 줄일까?
  - 그 크기를 줄인 문제의 답을 안다면?
    - 어떻게 최종 답을 알 수 있을까?

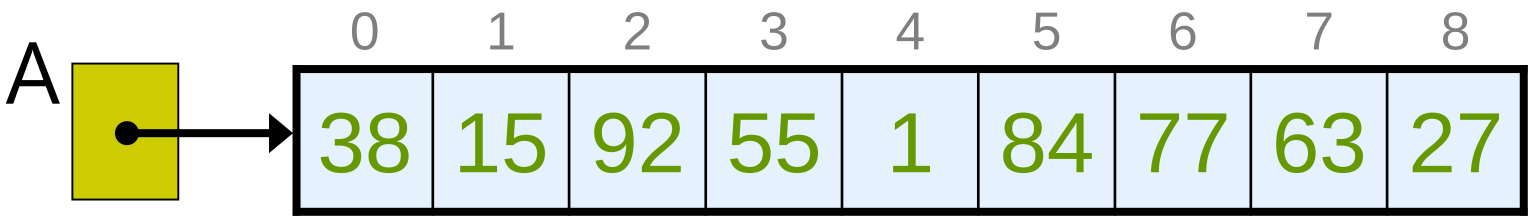
배열의 재귀적 구조 [1]

- A[0] 부터 A[8] 까지는 크기가 9 인 배열이다.

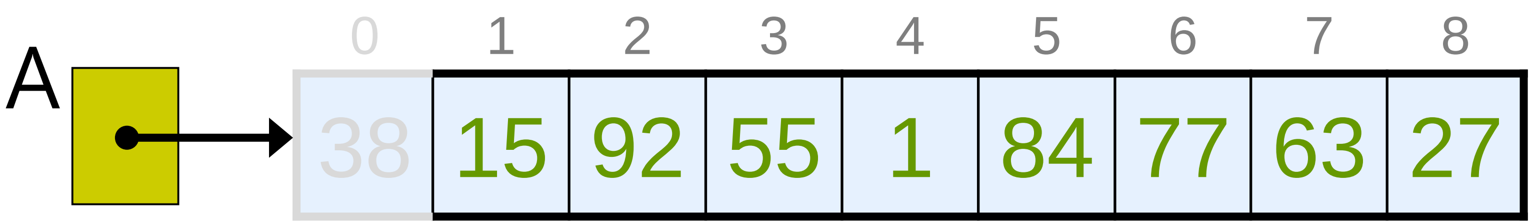


배열의 재귀적 구조 [2]

- A[0] 부터 A[8] 까지는 크기가 9 인 배열이다.



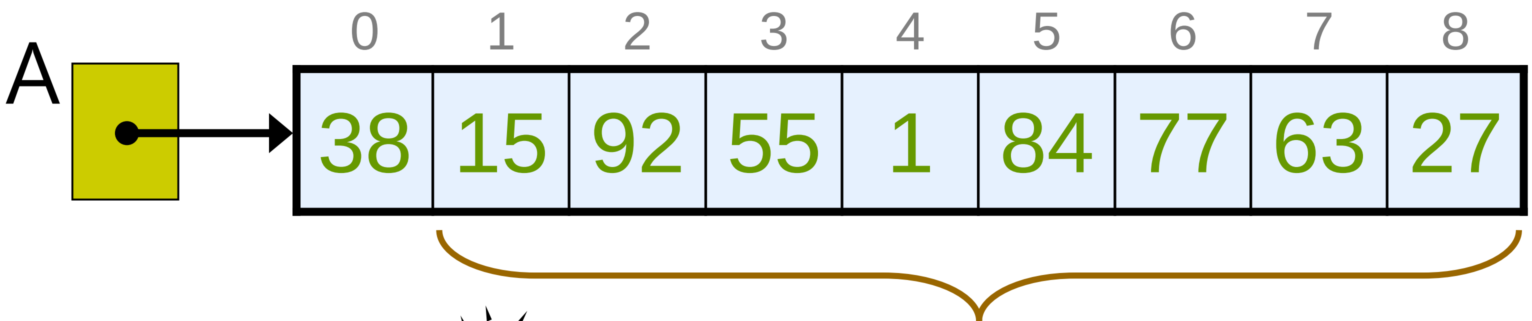
- 크기를 하나 줄여도, 여전히 배열!



  - 배열은 재귀적 구조 (recursive structure)를 가지고 있다.

배열에서 최대값 찾기 [1]

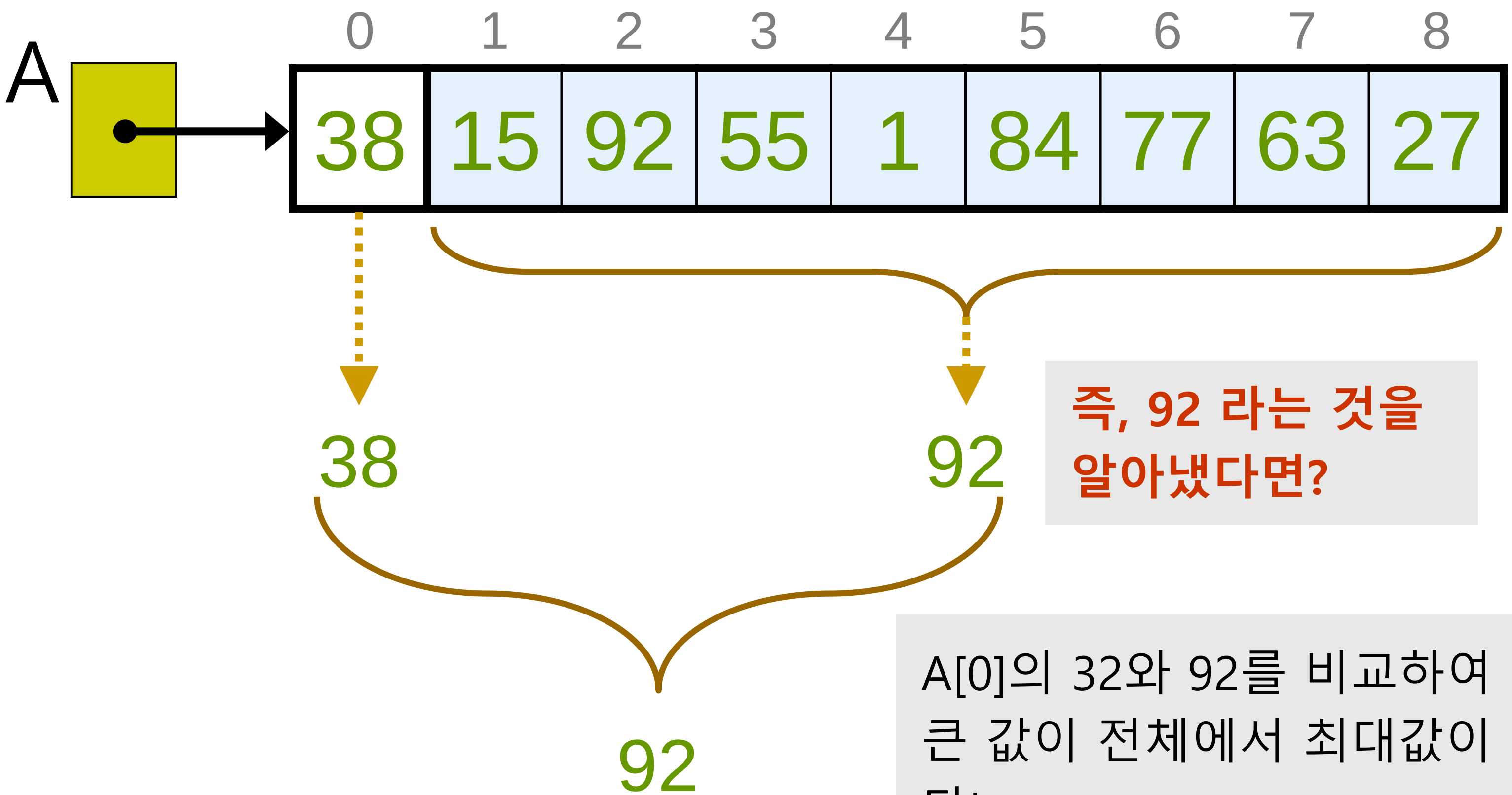
- A[0] 부터 A[8] 까지 중에서 최대값을 찾으려면?



크기가 하나 작아진 이 배열에서의
최대값을 알 수 있다고 하자.
그러면, 원래 배열에서의 최대값은?

배열에서 최대값 찾기 [4]

- A[1] 부터 A[8] 까지 중에서 최대값을 안다면?

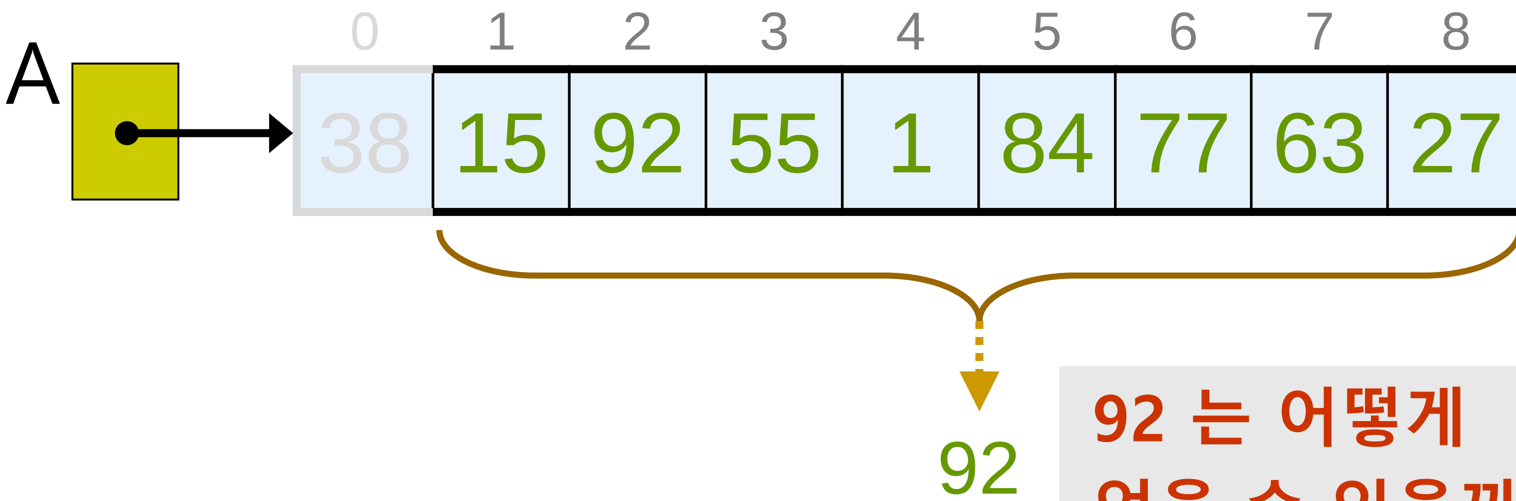


즉, 92 라는 것을 알아냈다면?

A[0]의 32와 92를 비교하여 더 큰 값이 전체에서 최대값이 된다!

❑ 배열에서 최대값 찾기 [5]

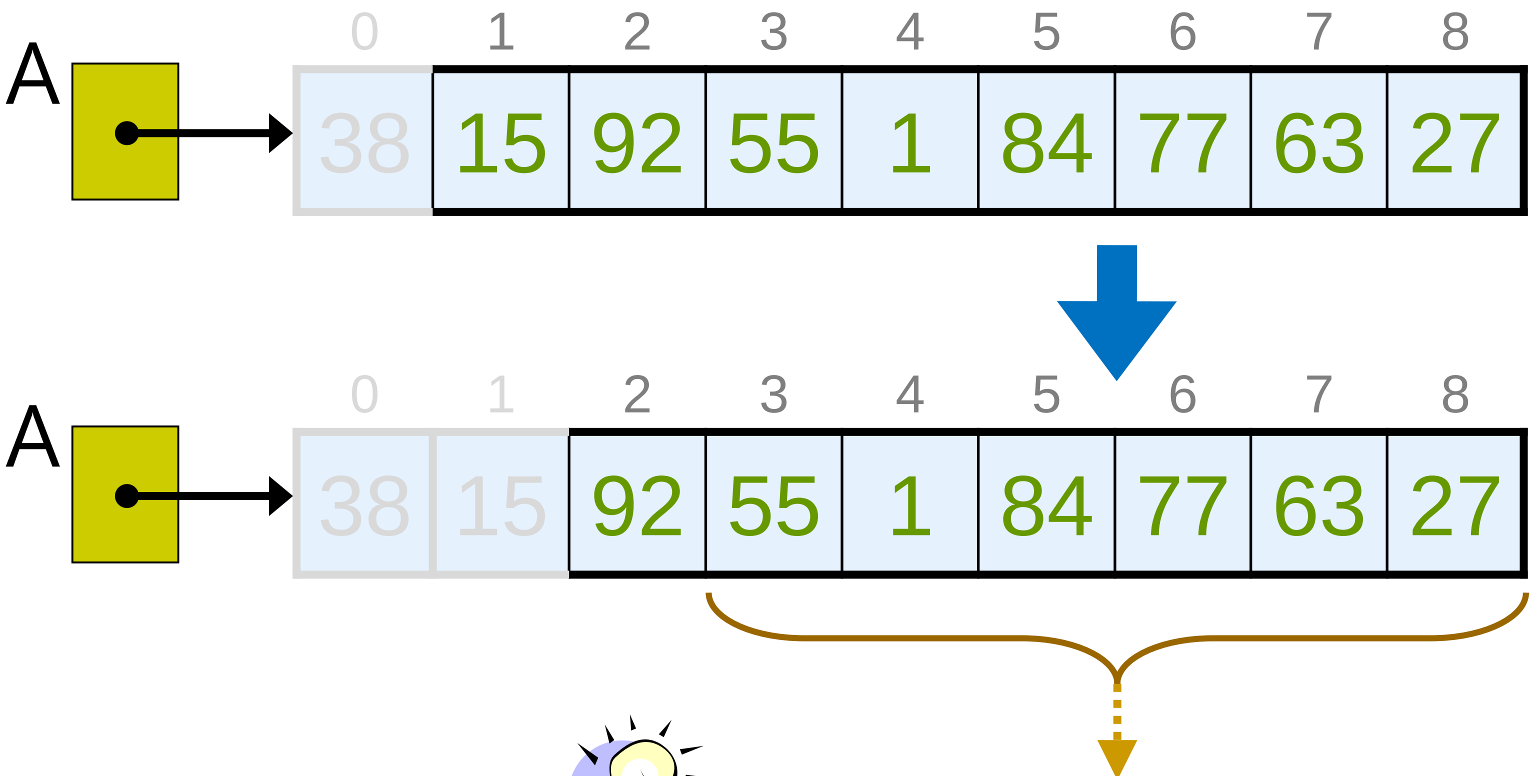
- A[1] 부터 A[8] 까지 중에서 최대값은 어떻게?



92 는 어떻게
얻을 수 있을까?

❑ 배열에서 최대값 찾기 [6]

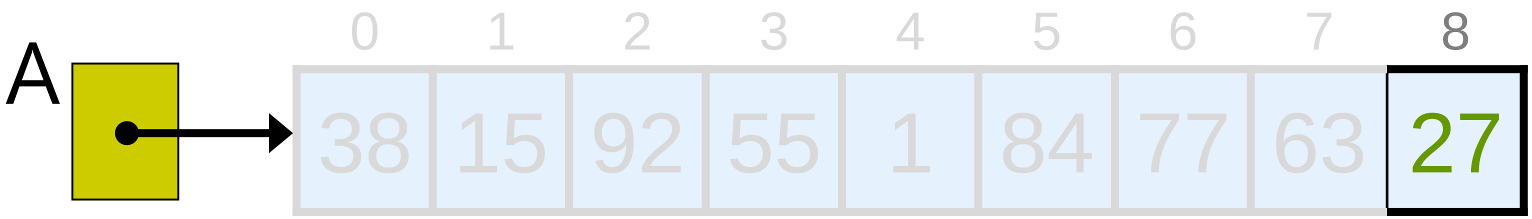
- A[1] 부터 A[8] 까지 중에서 최대값은 어떻게?



크기가 하나 작아진 이 배열에서의 최대값을 알 수 있다고 하자.
그러면, 원래 배열에서의 최대값은?

배열에서 최대값 찾기 [6]

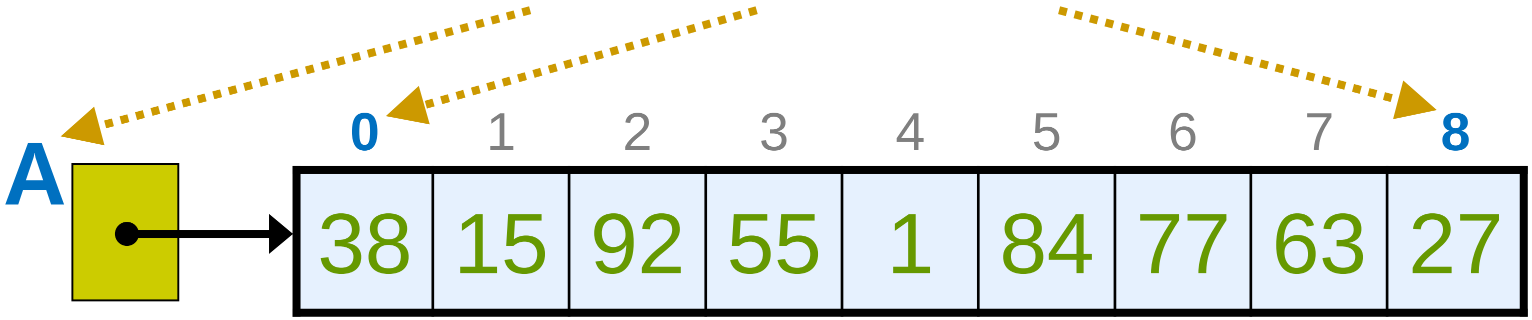
크기를 계속 줄여간다면, 어디까지?



- 배열에 원소가 하나만 있게 되면, 그 때의 최대값은?

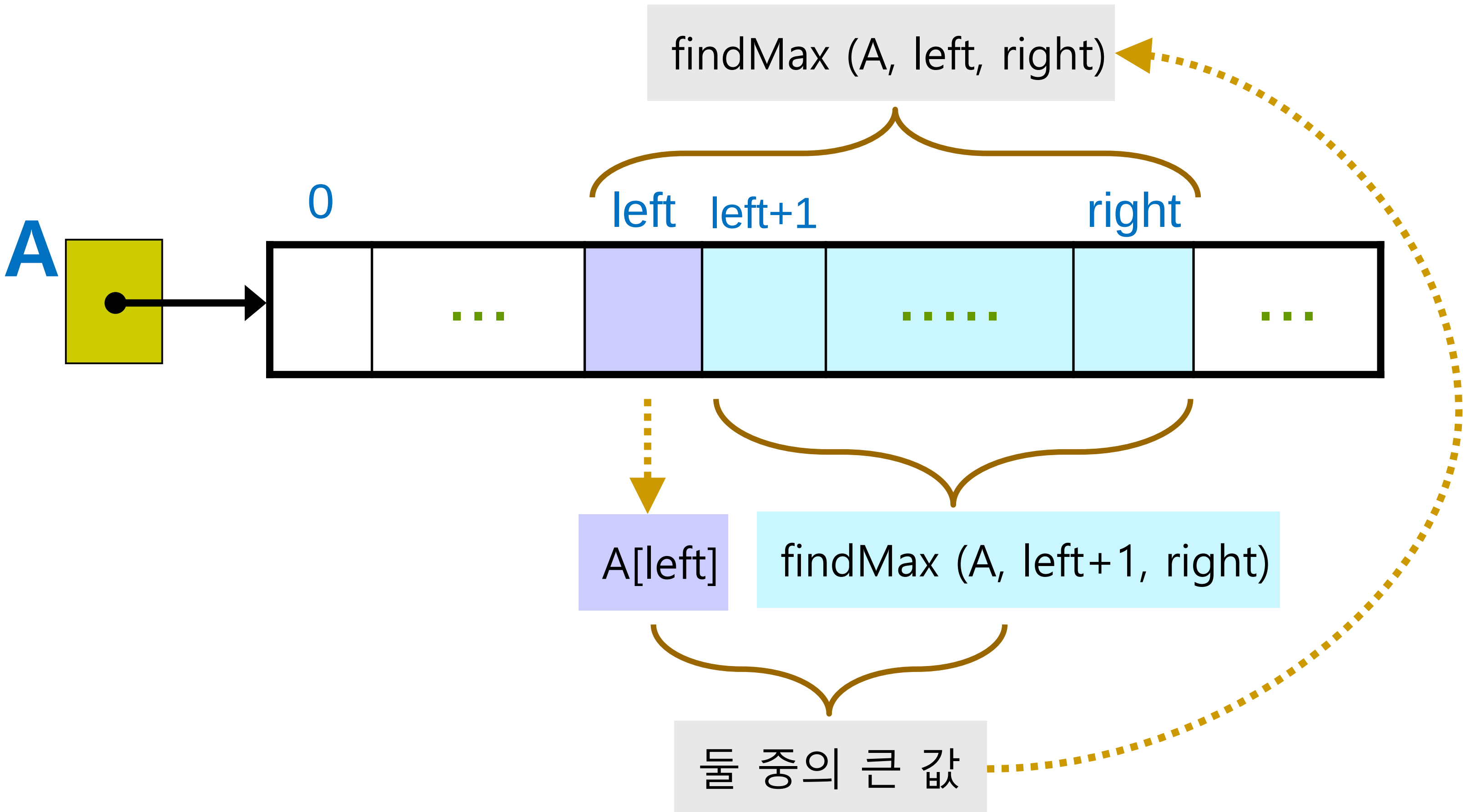
최대값 찾기 프로그램 [1]

- 어떻게 call 하면 좋을까?
  - int findMax (int **A**[], int **left**, int **right**) ;



  - 배열의 임의의 구간 내의 최대값을 찾을 수 있다.

❑ 최대값 찾기 프로그램 [2]


findMax (A, left, right)
0
left
left+1
right
A
…
……
…
A[left]
findMax (A, left+1, right)
둘 중의 큰 값

❑ 최대값 찾기 프로그램 [3]

```
public static void main(String[] args)
{
    int[] A = {32, 15, 99, ... ,27} ;
    int  max = findMax(A, 0, 8) ;
    ......
```

```
public static int  findMax (int[] A, int left, int right)
{
    if ( left == right ) {
        return  A[left] ;
    }
    else {
        int  maxOfSubPart = findMax (A, left+1, right) ;
        if ( A[left] >= maxOfSubPart ) {
            return  A[left] ;
        }
        else {
            return  maxOfSubPart  ;
        }
    }
}
```

Non-recursive version of findMax()

```
public static int findMax (int []A, int left, int right)
{
    int  max = A[left] ;
    int  curLoc = left + 1 ;
    while ( curLoc <= right ) {
        if ( max < A[curLoc] ) {
            max = A[curLoc] ;
        }
        curLoc++ ;
    }
    return  max ;
}
```

문자열 역순으로 인쇄하기

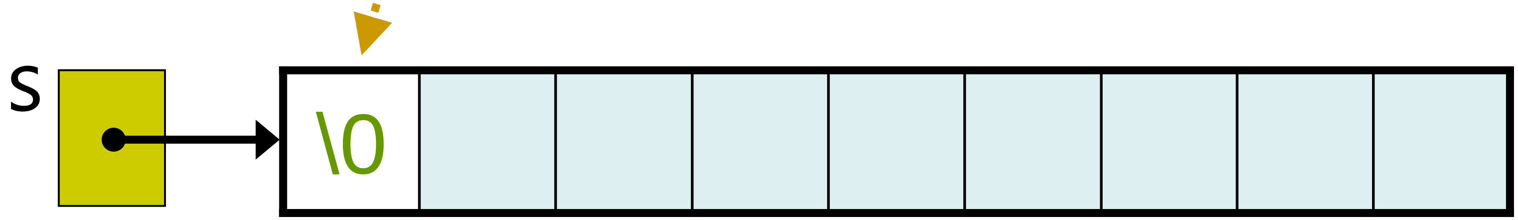
재귀적으로 생각하자!

- 가장 단순한 경우는?
  - 문자열이 비어있다면?
- 일반적으로는?
  - 문제의 크기를 줄여보자:
    - 어떻게 크기를 줄일까?
  - 그 크기를 줄인 문제의 답을 안다면?
    - 어떻게 최종 답을 알 수 있을까?

구체화 시켜보자: 가장 단순한 경우

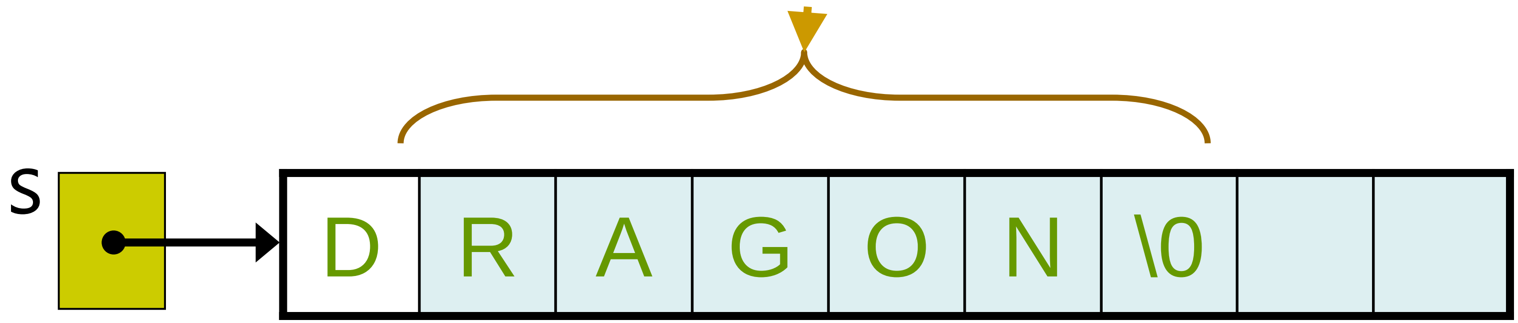
```
public void printInReverse (char[] s, int from) {
    if (문자열 s 가 비어 있다면) {
        // (가장 단순한 경우) 할 일이 없다
    }
    else { // (일반적인 경우) 문제의 크기를 줄여서 해결한다
        우선, 맨 처음 문자를 제외한 나머지 문자열을 역순으로 출력한다 ;
        그 다음, 미루어 두었던 맨 앞 문자를 출력한다 ;
    }
}
```

[redacted]



❑ 구체화 시켜보자: 일반적인 경우 [1]

```
public void printInReverse (char[] s, int from) {
    if (문자열 s 가 비어 있다면) {
        // (가장 단순한 경우) 할 일이 없다
    }
    else { // (일반적인 경우) 문제의 크기를 줄여서 해결한다
        우선, 맨 처음 문자를 제외한 나머지 문자열을 역순으로 출력한다 ;
        그 다음, 미루어 두었던 맨 앞 문자를 출력한다 ;
    }
}
```



❑ 구체화 시켜보자: 일반적인 경우 [2]

```
public void printInReverse (char[] s, int from) {
    if (문자열 s 가 비어 있다면) {
        // (가장 단순한 경우) 할 일이 없다
    }
    else { // (일반적인 경우) 문제의 크기를 줄여서 해결한다
        우선, 맨 처음 문자를 제외한 나머지 문자열을 역순으로 출력한다 ;
        그 다음, 미루어 두었던 맨 앞 문자를 출력한다 ;
    }
}
```

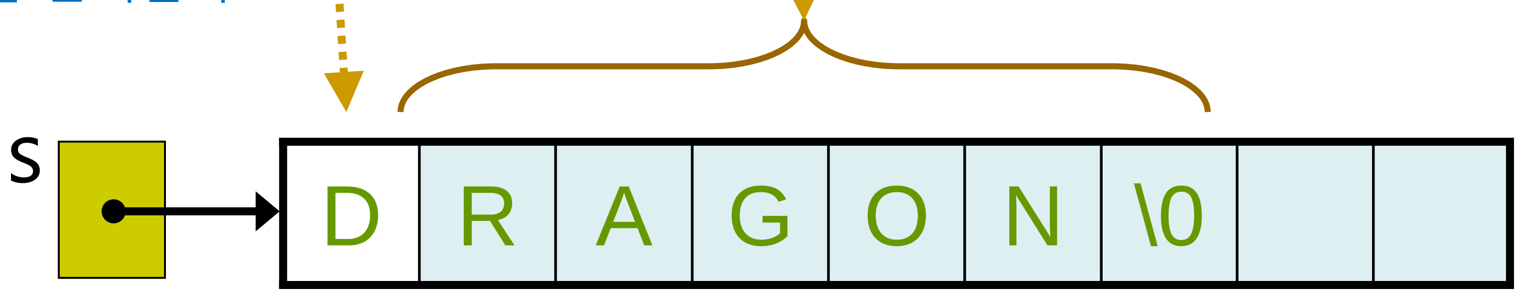
NOGARD

Recursive Function "printInReverse()"

```
public void printInReverse (char[] s, int from)
{
    if ( s[from] != '\0' ) {
        printInReverse (s, from+1) ;   // 문제의 크기를 줄여서 해결하였다
        System.out.print(s[from]) ;
    }
}
```

우선, 맨 처음 문자를 제외한 나머지 문자열을 역순으로 출력한다.

그 다음, 미루어 두었던 맨 앞 문자를 출력한다



재귀에서
문제의 크기는
어떻게 줄어드는가?

각 문제에서 줄어든 크기는?

문제	문제의 크기	
	원래 크기	줄어든 크기
Factorials	N	N-1
Fibonacci Numbers	N	(N-1) 과 (N-2)
Binomial Coefficients	N	(2 개의) N-1
배열에서 최대값 찾기	N	N-1
문자열 역순 인쇄하기	N	N-1

문제의 크기가 줄어드는 양상은 다양하다!

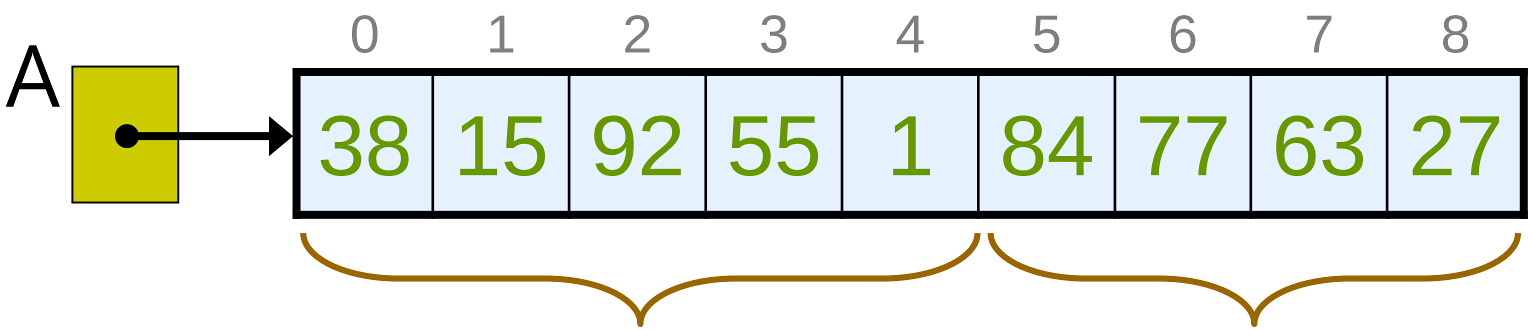
- 이렇게도...

```
P(N)
{
    if ( N 이 재귀의 끝 크기이면 ) {
        // 구체적인 답을 알고 있다. 그 답을  s0 라고 하자.
        return   s0 ;
    }
    else {
        문제를 n/2 크기의 두 개의 문제로 나눈다 ; // DIVIDE
        두 개의 문제로 나누어진 P(n/2) 를 각각 풀어서, // CONQUER
        그 답을 얻는다. 각각의 답을 s1, s2 라고 하자.
        s1 과 s2 를 문제에 맞게 합하여, 최종 답 s 를 얻는다. // MERGE
        return   s ;
    }
}
```

- DIVIDE – CONQUER – MERGE
  - **Divide-and-Conquer** 방법: 문제를 재귀적으로 해결함

❑ 배열을 반으로 나누어 최대값 찾기 [1]

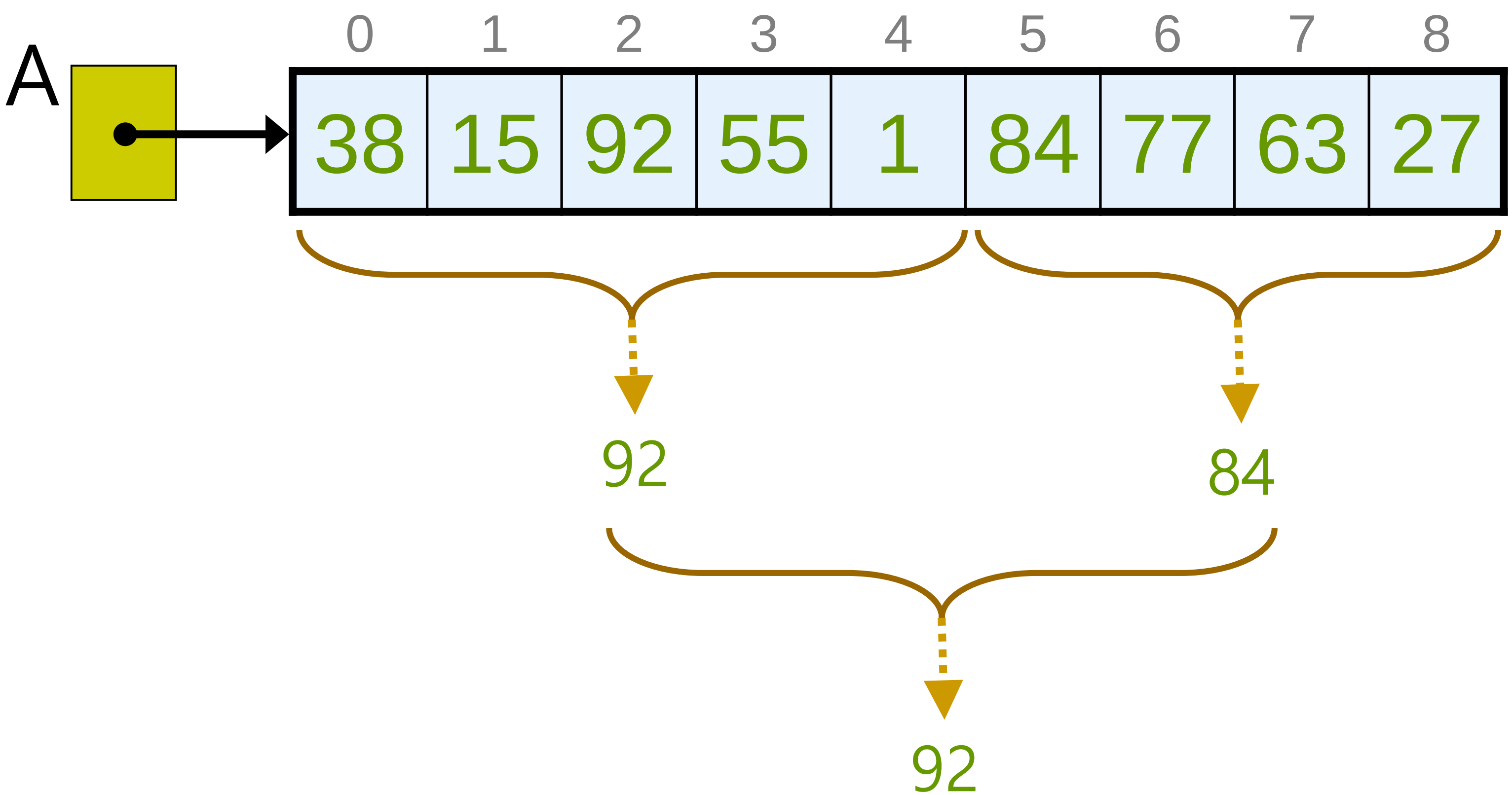
- A[0] 부터 A[8] 까지 중에서 최대값을 찾으려면?



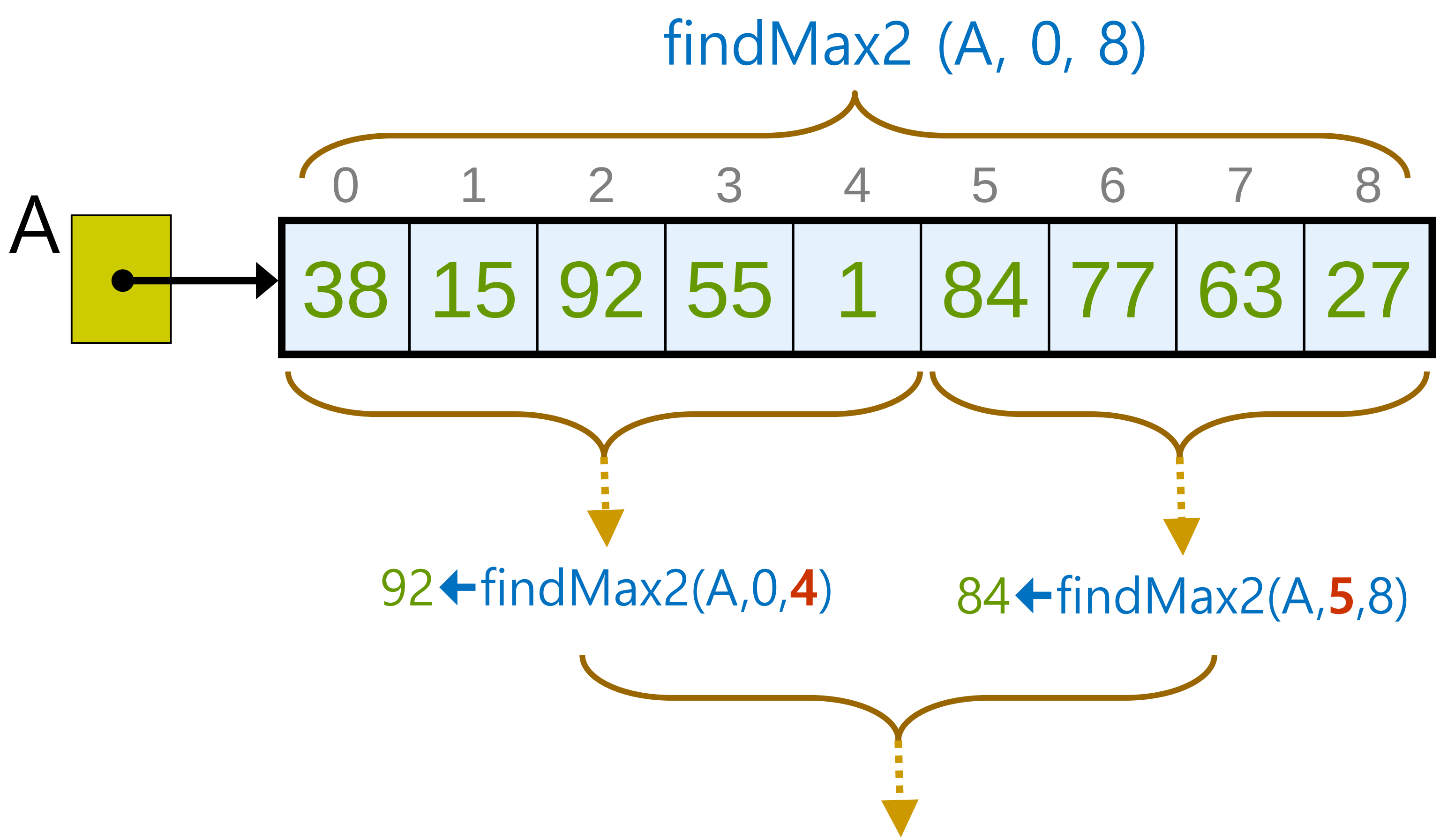
크기가 반으로 작아진 이 각각의 배열에서의 최대값을 알 수 있다고 하자.
그러면, 원래 배열에서의 최대값은?

배열을 반으로 나누어 최대값 찾기 [2]

- 반으로 나누어진 각각의 배열의 최대값을 안다면?



배열을 반으로 나누어 최대값 찾기 [3]



배열을 반으로 나누어 최대값 찾기 [4]

```
public static void  main() {
      int [] A = {32, 15, 99, ... , 27} ;
      int max ;

      max = findMax2(A, 0, 8) ;
      ......
```

```
public static int  findMax2 (int[] A, int left, int right)
{
      if ( left == right ) {
            return  A[left] ;
      }
      else {
            int  mid = (left+right) / 2 ;  // 가운데 위치
            int  maxOfLeftPart   = findMax2 (A, left, mid) ;
            int  maxOfRightPart = findMax2 (A, mid+1, right) ;
            if ( maxOfLeftPart >= maxOfRightPart ) {
                  return  maxOfLeftPart ;
            }
            else {
                  return  maxOfRightPart ;
            }
      }
}
```

☐ 배열을 반으로 나누어 최대값 찾기 [5]

```
public static int  findMax2 (int[] A, int left, int right)
{
    if ( left == right ) {
        return  A[left] ;
    }
    else {
        int  mid = (left+right) / 2 ;  // 가운데 위치                DIVIDE
        int  maxOfLeftPart   = findMax2 (A, left, mid) ;
        int  maxOfRightPart = findMax2 (A, mid+1, right) ;         CONQUER
        if ( maxOfLeftPart >= maxOfRightPart ) {
            return  maxOfLeftPart ;
        }
        else {                                                      MERGE
            return  maxOfRightPart ;
        }
    }
}
```

Divide-and-Conquer Method

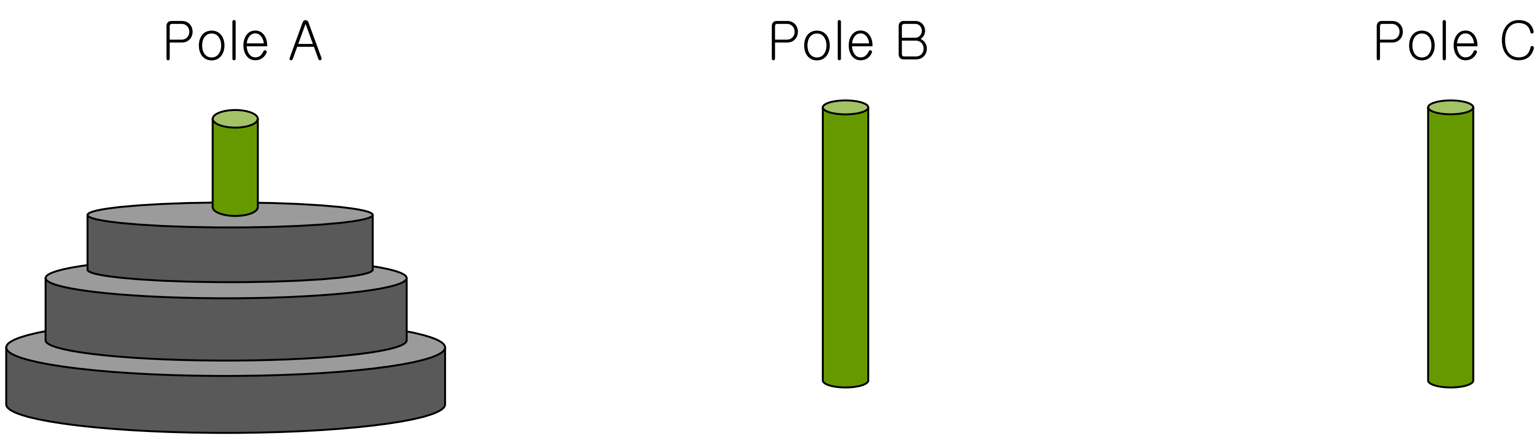
- 문제를 나누어 재귀적으로 해결하는 방법

단계	findMax (N-1) 크기로	findMax2 N/2 크기로	QuickSort	비고
DIVIDE	없음	단순	복잡	문제에 따라 단순할 수도 복잡할 수도 있다
CONQUER	N-1 1 개로	N/2 2 개로	K 와 (N-K-1) (데이터에 따라) 2 개로	줄어든 크기나, 풀어야 할 문제의 개수는, 문제에 따라 달라진다
MERGE	보통	보통	없음	문제에 따라 단순할 수도 복잡할 수도 있다

하노이 탑
(Tower of Hanoi)

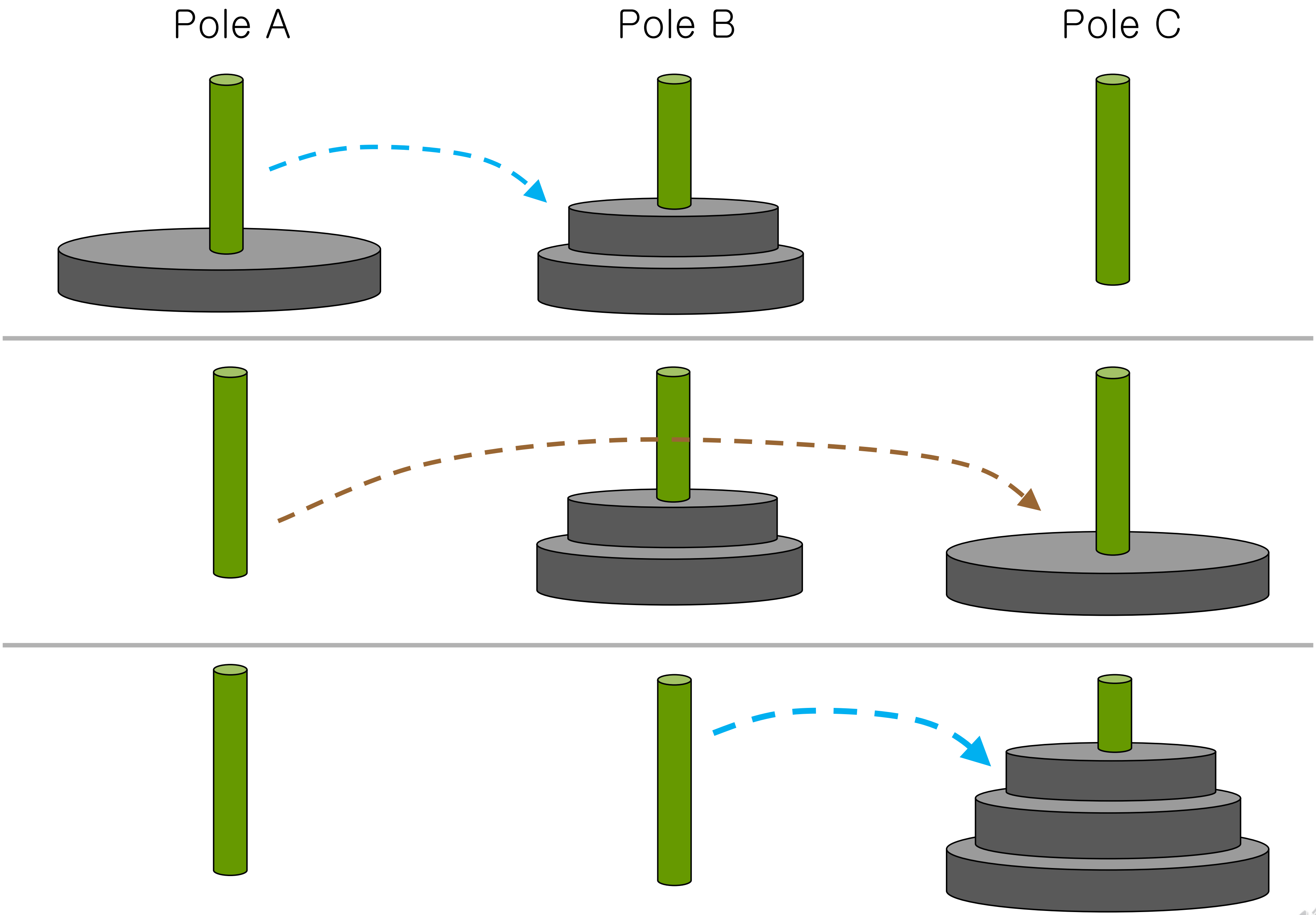
❑ 하노이 탑 (Tower of Hanoi)

- 세개의 기둥(pole) A, B, C 가 있다. 그 중 A에 밑에서부터 크기 순서로 원판 세 개가 꽂혀있다. 이제 A의 세 개의 원판을 모두 기둥 C로 옮기려고 한다. 단, 한 번에 하나의 원판만 이동시켜야 하며, 어떠한 경우에도 작은 원판이 큰 원판 밑에 있게 되어서는 아니 된다. 그러나 옮기는 과정에서 모든 기둥을 사용할 수 있다.



생각하는 방법

- 윗 쪽 두 개의 원판을 원하는 pole 로 옮길 수 있는 방법이 있다면?
  - 그렇다면, pole A 의 윗 쪽 **2** 개의 원판을 (pole C 를 활용하여) 우선 pole B 로 옮겨 놓는다.
  - 그러면, pole A 에는 가장 큰 원판 **1** 개만 남아 있을 것이므로 이 원판을 pole C 로 옮긴다.
  - 마지막으로, pole B 의 **2** 개의 원판을 (pole A 를 활용하여) pole C 로 옮긴다.


Pole A
Pole B
Pole C

문제의 답 (이동 방법을 출력)

- Move 2 disks from pole A to pole B using pole C:
  - Move from pole A to pole C.
  - Move from pole A to pole B.
  - Move from pole C to pole B.

- Move 1 disk:
  - Move from pole A to pole C.

- Move 2 disks from pole B to pole C using pole A:
  - Move from pole B to pole A.
  - Move from pole B to pole C.
  - Move from pole A to pole C.

일반적으로 n개의 원판을 옮기는 방법은?

- 윗쪽 (n-1) 개의 원판을 원하는 pole로 옮길 수 있는 방법이 있다면?
  - 그렇다면, pole A 의 윗 쪽 **(n-1)** 개의 원판을 (pole C 를 활용하여) 우선 pole B 로 옮겨 놓는다.
  - 그러면, pole A 에는 가장 큰 원판 **1** 개만 남아 있을 것이므로 이 원판을 pole C 로 옮긴다.
  - 마지막으로, pole B 의 **(n-1)** 개의 원판을 (pole A 를 활용하여) pole C 로 옮긴다.

- 그러므로, 다음과 같은 함수를 정의할 수 있다.

  void moveDisk (int n, char poleX, char poleY, char poleZ)

  - 즉, "moveDisk()" 는 pole X 에 있는 n 개의 디스크를 pole Y 를 활용하여 pole Z 로 옮기는 과정을 찾아주는 함수이다.

하노이 탑 (Tower of Hanoi) 알고리즘

- Recursive Function “moveDisk()”

```
public void moveDisk (int n, char poleX, char poleY, char poleZ)
{
    if ( n == 1 )
        System.out.println ("Move from "+ poleX + " to " + poleZ) ;
    else {
        moveDisk (n-1, poleX, poleZ, poleY) ;
        System.out.println ("Move from "+ poleX + " to " + poleZ) ;
        moveDisk (n-1, poleY, poleX, poleZ) ;
    }
}
```

moveDisk() 의 약간 수정된 형태

- 탈출 경우: 돌판이 0 개

```
public void moveDisk (int n, char poleX, char poleY, char poleZ)
{
    if ( n > 0 ) {
        moveDisk (n-1, poleX, poleZ, poleY) ;
        System.out.println ("Move from "+ poleX + " to " + poleZ) ;
        moveDisk (n-1, poleY, poleX, poleZ) ;
    }
}
```

생각해 볼 점

재귀의 특성

- Divide-And-Conquer
- 생각보다 많은 문제를 해결할 수 있다.
  - 모든 문제를 재귀적으로 해결할 수 있는 것은 아니다.
- 복잡한 문제를 쉽게 해결할 수 있다.
  - 재귀적으로 접근하지 않으면 오히려 풀기 어려운 경우도 있다.
- 재귀적으로 해결할 수 있다고 해서 반드시 효율적이지는 않다.
  - 예: Fibonacci numbers
- 문제에 따라 다양한 양상을 보인다.
  - 문제를 나누는 방법
  - 나누어진 문제의 개수
  - 부분 결과를 합하는 방법
- 데이터의 특성이 재귀적이면, 그와 관련된 문제도 재귀적으로 풀릴 가능성이 있다.
  - 배열, 리스트, 트리, 그래프, 등등

재귀적 사고

- 일반적인 경우:
  - 문제의 크기를 줄인 작아진 문제의 답을 안다고 했을 때, 원래 문제의 답을 알 수 있는지?
    - (N-1)! 을 알면 N! 의 값을 얻을 수 있는지?

- 재귀의 탈출:
  - 문제의 크기가 아주 작은 경우에, 문제의 답을 직접적으로 쉽게 얻을 수 있는지?
    - 0! 의 값은 얼마인지?

재귀적 문제 풀이는 항상 좋은가?

- Fibonacci Number 문제는?
  - N 을 증가시키면서 언제까지 프로그램이 죽지 않고 답을 계산하는지 알아보자.
  - 비재귀적으로 어떻게 할 수 있는지 생각해 보자.

재귀적 문제 풀이는 항상 좋은가?

- 하노이 탑 문제
  - N 을 증가시키면서 언제까지 프로그램이 죽지 않고 답을 계산하는지 알아보자.
    - 여러분 각자는 자신의 프로그램이 죽었는지, 아니면 컴퓨터가 계산을 계속 하고 있는지 판단할 수 있는가?
  - 이 문제는 비재귀적으로 해결이 쉽게 될까?
  - N 을 증가시키면서 각 N 에 대해 풀이 시간을 측정해 보는 것도 좋다.
    - Class “TIMER” 를 이용

재귀 함수의 성능 분석

재귀 함수의 연산 스텝 세기

$$f(n) = \begin{cases} 1 & if\ n = 1 \\ f\left(\frac{n}{2}\right) + n + 1 & if\ n > 1 \end{cases}$$

```
public int  f (int n)
{
    if (n == 1) {
        return 1 ;
    }
    else {
        return f(n/2) + n + 1 ;
    }
}
```

재귀함수 f() 의 시간 복잡도 [1]

- 모든 연산은 동일한 시간이 걸린다고 가정
  - 4 칙 연산 (+, /)
  - 비교 연산 (==)
  - 함수 call / return

```
public int  f (int n)
{
     if (n == 1) {          // +1
          return 1 ;        // +1
     }
     else {
          return f(n/2) + n + 1 ;   // T(n/2) + 5
     }
}
```

n/2	: +1
f()	: +1
+n	: +1
+1	: +1
return	: +1
f(n/2)	: T(n/2)

➡ 그러므로 f() 의 시간 복잡도 T(n) 은 다음과 같이 나타낼 수 있다:

$$T(n) = \begin{cases} 2 & if\ n = 1 \\ T(n/2) + 6 & if\ n > 1 \end{cases}$$

재귀함수 f() 의 시간 복잡도 [2]

$$T(n) = \begin{cases} 2 & if\ n = 1 \\ T(n/2) + 6 & if\ n > 1 \end{cases}$$

```
public int  f (int n)
{
        if (n == 1) {     // +1
                return 1 ;  // +1
        }
        else {
                return f(n/2) + n + 1 ; // T(n/2) + 5
        }
}
```

- 재귀함수 f() 의 시간복잡도를 나타내는 점화식 T(n) 을 풀어보자:

만일 n 이 2 의 승수 형태의 정수라면?
그러면 n = 2^k (k ≥ 0) 의 형태로 나타낼 수 있다.

$$\begin{aligned} T(n) = T(2^k) &= T(2^k/2) + 6 = T(2^{k-1}) + 6 \\ &= (T(2^{k-2}) + 6) + 6 = T(2^{k-2}) + 6 \cdot 2 \\ &= (T(2^{k-3}) + 6) + 6 \cdot 2 = T(2^{k-3}) + 6 \cdot 3 \\ &= T(2^0) + 6 \cdot k = 2 + 6 \cdot k \end{aligned}$$

즉, $T(n) = T(2^k) = 6 \cdot k + 2$
그런데, $n = 2^k$ 이므로, $k = log_2(n)$ 이다.

그러므로,
$T(n) = T(2^k) = 6 \cdot k + 2 = 6 \cdot log_2(n) + 2$

- Big-Oh 로 나타낸다면,

$T(n) = O(log(n))$

재귀함수 f() 의 시간 복잡도 [3]

- 일반적으로 T(n) 의 big-Oh 는 ?

$2^k \le n < 2^{k+1}$ 를 만족하는 k 에 대해,

$$T(n) < T(2^{k+1})$$

$T(2^k) = 6 \cdot k + 2$ 이므로, (앞 쪽의 결과)

$$T(n) < T(2^{k+1}) = 6 \cdot (k+1) + 2$$

$2^k \le n$ 이므로, $k \le log_2\ (n)$

따라서,

$$T(n) < 6 \cdot (log_2(n) + 1) + 2 = 6 \cdot log_2(n) + 8$$

그러므로,

$$T(n) = O(log(n))$$

End of

"Recursion"