재귀 Recursion

재귀: 내 안의 나를 찾는다

재귀 알고리즘 = 자기호출 알고리즘

- 자신과 성격은 똑같지만 크기만 작은 알고리즘(들)을 호출하는 알고리즘
- 복잡한 문제도 간명하게 볼 수 있게 한다
- 예: 탐색, 정렬, 수열, …
- 잘 쓰면 보약
  - 정렬, 탐색, …
- 잘못 쓰면 독약
  - 피보나치수 구하기, 최적 행렬곱 경로, …

수열

등차 수열

$a_n = a_{n-1} + 3,\ a_1 = 1$

n번째 원소는 자신과 성격이 똑같지만
순서가 하나 작은(n-1번째) 원소에 3을 더한 것

재귀 알고리즘으로

```
seq(n) :
    if (n = 1)
        return 1
    else
        return  (seq(n-1) + 3)
```

피보나치 수열

재귀가 치명적인 예

$$\text{fib}(n) = \begin{cases} 1 & \text{if } \ n = 1 \text{ or } 2 \\ \text{fib}(n\text{-}1) + \text{fib}(n\text{-}2) & \text{if } \ n > 2 \end{cases}$$

```
fib(n) :
    if (n ≤ 2) return 1
    else return (fib(n-1)+fib(n-2))
```

```
fib(n) :
    if (n ≤ 2) return 1
    else return fib(n-1)+fib(n-2)
```

← 재귀

Simple, but terrible!

```
fib(n) :
    t[1] ← t[2] ← 1
    for i ← 3 to n
        t[i] ← t[i-1] + t[i-2]
    return t[n]
```

n에 비례하는 시간

← 비재귀

재귀적 fib(100)은 얼마나 걸릴까?

내 데스크 탑 PC: Pentium 3GHz

fib(50) – 36초

fib(66) – 하루 정도

fib(100) – 3만5천년 정도

fib(136) – 1조년 초과

지수함수적 중복 호출로 인해 이런 치명적인 비효율이 발생한다

비재귀적 fib(100)은?

```
fib(n) :
    t[1] ← t[2] ← 1
    for i ← 3 to n
        t[i] ← t[i-1] + t[i-2]
    return t[n]
```

천만분의 1초도 안걸린다

팩토리얼

$n! = 1 \cdot 2 \cdot 3 \cdots n$

$n! = n \cdot (n-1) \cdot (n-2) \cdots 1$
$\quad = n \cdot (n-1)!$

재귀적 구조

$$\text{factorial}(n) = \begin{cases} 1 & \text{if } n = 0 \\ n * \text{factorial}(n-1) & \text{if } n > 0 \end{cases}$$

fact(n) :

    tmp ← 1

    for i ← 2 **to** n

        tmp ← i * tmp

    return tmp

← 비재귀

n에 비례하는 시간

fact(n) :

    if (n = 0) **return** 1

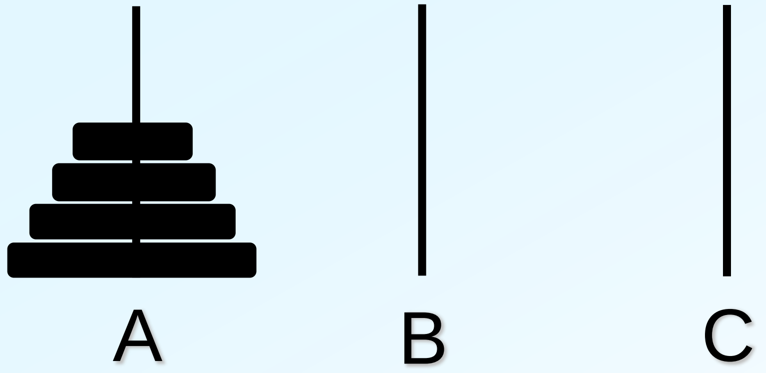
    else return n*fact(n-1)

← 재귀

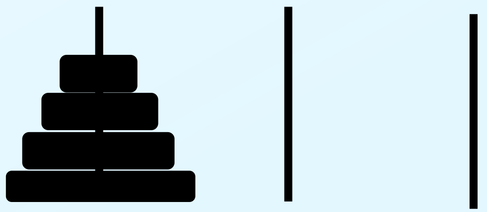
하노이 타워

- 디스크 n개, 기둥 3개 (A, B, C)
- 한 번에 하나의 디스크를 옮길 수 있다
- 큰 디스크는 작은 디스크 위에 놓일 수 없다
- 목표:

  n 개의 디스크를 기둥 A로부터 기둥 B로 옮긴다

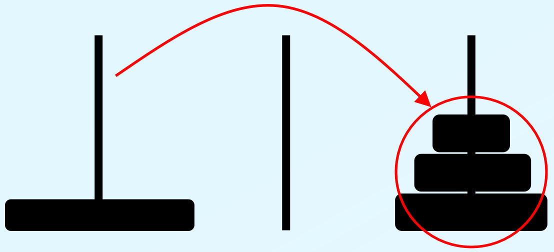


- 함수 정의: move(*n*, *source*, *destination*, *spare*)



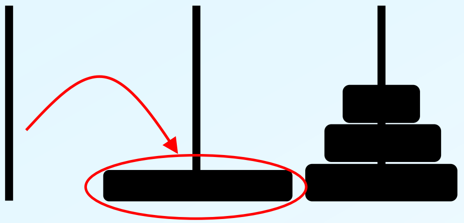
  - *n* 개의 디스크를 *source* 기둥으로부터 *destination* 기둥으로 옮긴다.
  - *Spare* : 보조 기둥.

- 예: move(4, *A*, *B*, *C*)



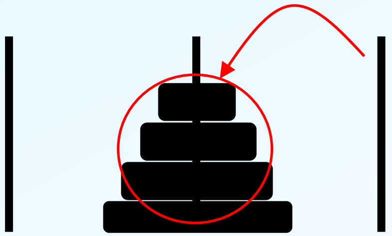
  move(3, *A*, *C*, *B*)

  A 에 있는 (유일한) 디스크를 *B*로 옮긴다



  move(3, *C*, *B*, *A*)

- 목표 : move(*n*, *A*, *B*, *C*)



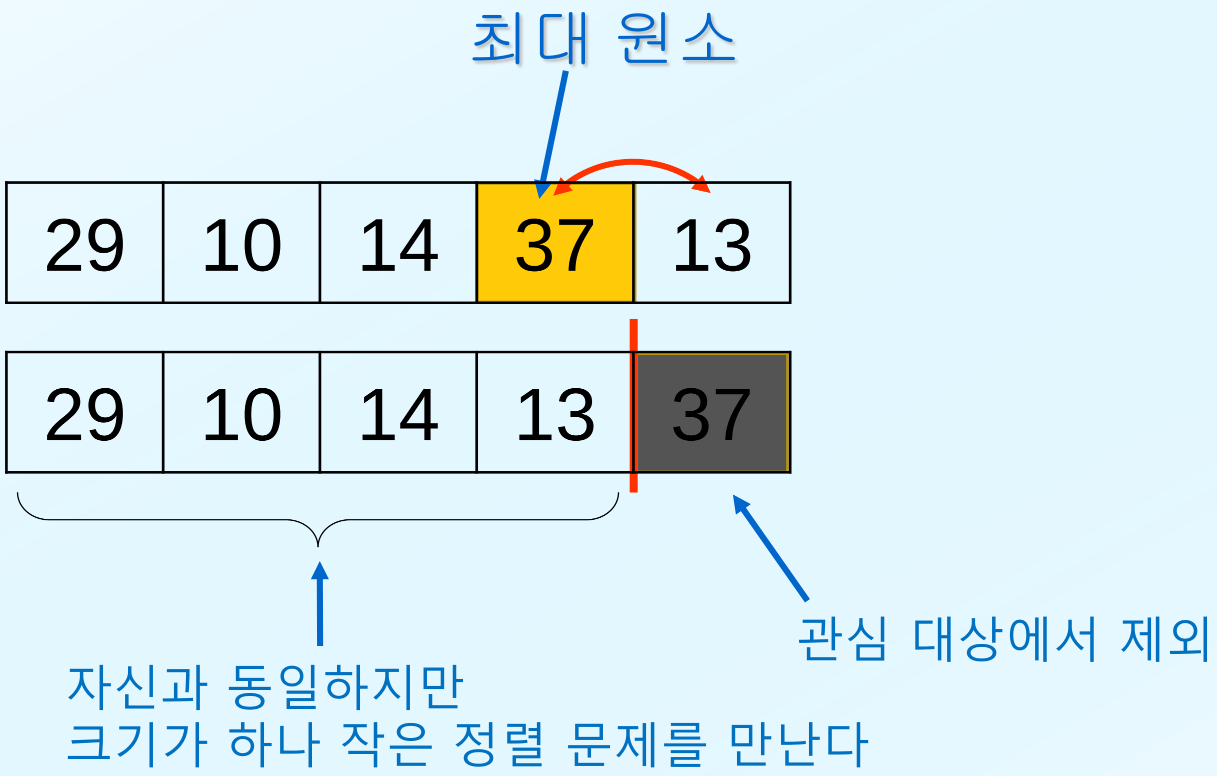
```
move(n, A, B, C):
    if (n > 0)
        move(n-1, A, C, B)
        A 에 있는 (유일한) 디스크를 B로 옮긴다
        move(n-1, C, B, A)
```

선택 정렬Selection Sort

1. 최대 원소를 찾는다
2. 최대 원소와 맨 오른쪽 원소를 자리 바꾼다
3. 맨 오른쪽 자리를 관심 대상에서 제외한다
4. 원소가 1 개 남을 때까지 위 1~3의 순환을 반복

이 과정을 한번 끝내면 자신과 동일하지만
크기가 하나 작은 문제를 만난다

Animation


최대 원소
29
10
14
37
13
29
10
14
13
37
자신과 동일하지만
크기가 하나 작은 정렬 문제를 만난다
관심 대상에서 제외

알고리즘 selectionSort()

```
selectionSort(A[], n):    ◀ 배열 A[0...n-1]을 정렬한다
    for last ← n-1  downto 1
        A[0...last] 중 가장 큰 수 A[k]를 찾는다
        A[k] ↔ A[last]    ◀ A[k]와 A[last]의 값을 교환
```

이를 재귀 알고리즘으로 표현하면

```
selectionSort(A[], n):
    if (n > 1)
        A[0...n-1] 중 가장 큰 수 A[k]를 찾는다
        A[k] ↔ A[n-1]
        selectionSort(A, n-1)
```

전위, 중위, 후위 표현법

수식을 표현할 때 연산자의 상대적 위치에 따라

전위 표현 Prefix Expression
중위 표현 Infix Expression
후위 표현 Postfix Expression

으로 나뉜다

중위 표현법		전위 표현법	후위 표현법
A + B * C − 2	→	− + A * B C 2	A B C * + 2 −
(A + B) * (C − 2)	→	* + A B − C 2	A B + C 2 − *

형식 언어로 표현하면

중위 표현

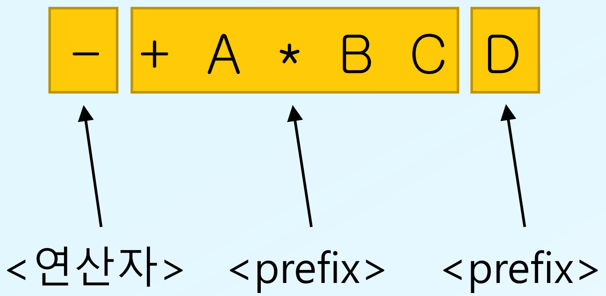
```
<infix> = <변수> ¦ <infix><연산자><infix>
<연산자> = + ¦ - ¦ * ¦ /
<변수> = A ¦ B ¦ ... ¦ Z
```

전위 표현

```
<prefix> = <변수> ¦ <연산자><prefix><prefix>
<연산자> = + ¦ - ¦ * ¦ /
<변수> = A ¦ B ¦ ... ¦ Z
```

후위 표현

```
<postfix> = <변수> ¦ <postfix><postfix><연산자>
<연산자> = + ¦ - ¦ * ¦ /
<변수> = A ¦ B ¦ ... ¦ Z
```



중위 표현법		전위 표현법	후위 표현법
A + B * C - D	→	− + A * B C D	A B C * D 2 −
(A + B) * (C - D)	→	* + A B − C D	A B + C D - *

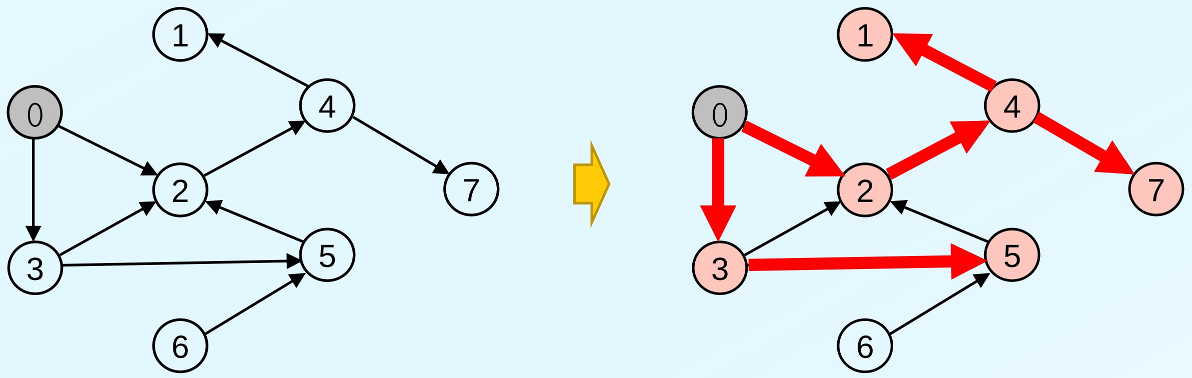
* 교재 typo: 2 → D

깊이 우선 탐색 Depth-First Search

DFS(x):

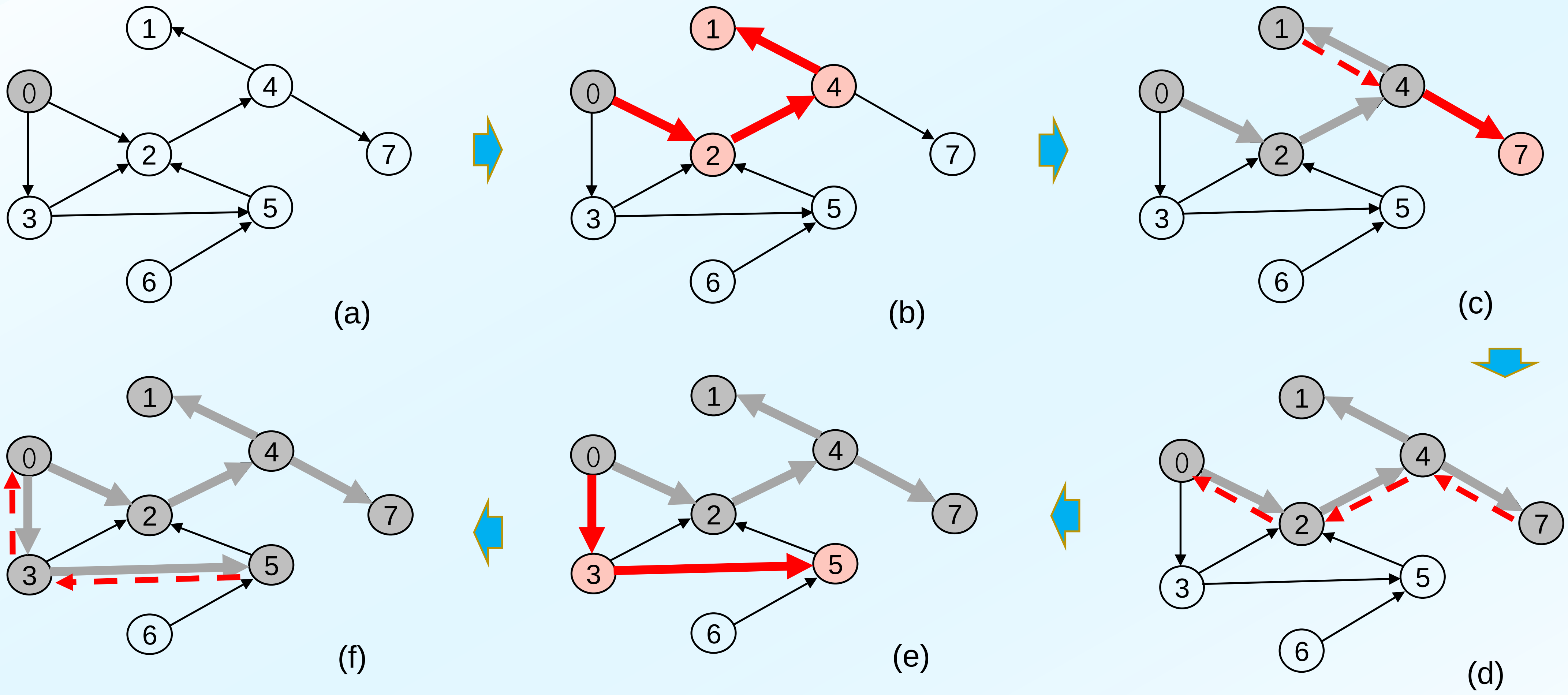
    x.visited ← true
    x에서 화살표로 연결된 노드 중 방문되지 않은 노드 y에 대하여
        DFS(y)

노드와 이들을 연결하는 간선으로 이루어진 그래프에서
시작 노드로부터 방문할 수 있는 모든 노드를 방문하는 작업



0번 노드로부터
방문할 수 있는 노드들:
1, 2, 3, 4, 5, 7

DFS의 작동 예



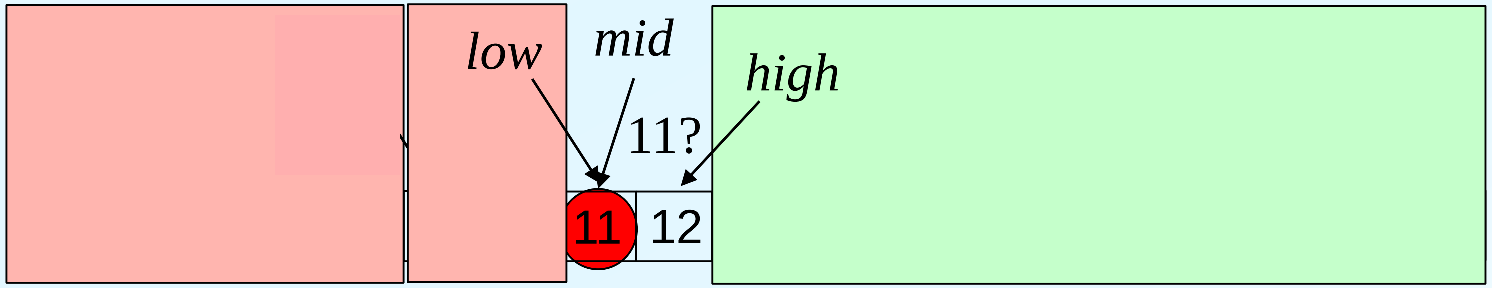
이진 탐색 Binary Search

Animation

```
binarySearch (A[], n, x) :
        ◀ 정렬된 배열 A[0…n-1]에서 원소 x를 찾는다
        low ← 0
        high ← n-1
        while (low ≤ high)
                mid ← (low + high)/2
                if (A[mid] < x) low ← mid + 1
                else if (A[mid] > x) high ← mid – 1
                else return mid
        return “Not found”
```

✓ Time: $O(\log n)$

x = 11을 찾아보자

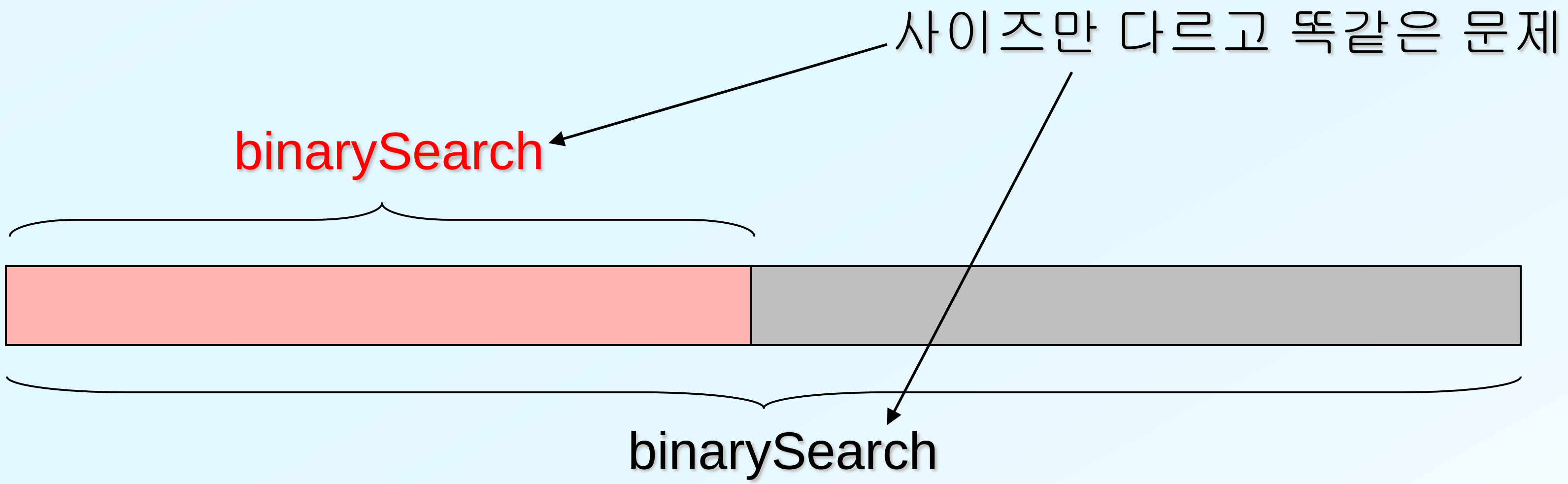


✓ 재귀적 성격이 숨어있다. 그러면 아예 재귀 알고리즘으로 만들어보자.

재귀 알고리즘: 이진 탐색

Animation

```
binarySearch(A[], x, low, high) :
        ◀ 정렬된 배열 A[low…high]에서 원소 x를 찾는다
        if (low > high) return "Not found"
        mid ← (low + high)/2
        if (A[mid] < x) return binarySearch(A, x, mid+1, high)
        else if (A[mid] > x) return binarySearch(A, x, low, mid-1)
        else return mid
```



재귀와 수학적 귀납법

재귀 알고리즘의 필수 구비 조건

> ❶경계 조건 Base Condition (또는 종료 조건): 재귀 호출이 반복되다 궁극적으로 끝나는 조건
> ❷재귀 호출
> ❸관계: 닮음꼴 작은 문제(들)와 본 문제 간의 관계를 나타내는 부분

하노이 타워의 예

```
move(n, A, B, C):
        if (n > 0)                                   ❶
            move(n-1, A, C, B)
❷           A 에 있는 (유일한) 디스크를 B로 옮긴다      ❸
            move(n-1, C, B, A)
```

종료 조건이 없으면?

Base case가 없으면?

fib(n) :

return (fib(n-1)+fib(n-2))

move(n, A, B, C) :

Base case가 없으면?

move(n-1, A, C, B)
A 에 있는 (유일한) 디스크를 B로 옮긴다
move(n-1, C, B, A)

하노이 타워 문제와 수학적 귀납법

$T(n)$: 하노이 탑 알고리즘이 n개의 원반을 옮기는 데 필요한 이동(원반 하나를 옮기는 것)의 총 횟수

Fact: $T(n) = 2^n - 1$

<증명>

경계 조건: $T(0) = 0 = 2^0 - 1$

귀납적 가정: $T(k) = 2^k - 1$라 가정하자

귀납적 전개:

$$\begin{aligned} T(k+1) &= 2 \cdot T(k) + 1 \\ &= 2(2^k - 1) + 1 \\ &= 2^{k+1} - 1 \end{aligned}$$