Ch. 2 Recursion: The Mirrors

- Recursive algorithm
  - An algorithm that calls itself for subproblems
  - The subproblems are of exactly the same type as the original problem – mirror images
  - It provides a simple look for a complicated problem
  - E.g., search, sorting, traversals, etc.
    - Medicine when properly used, “poisonous” when badly used.

Asymptotic Analysis

- 입력의 크기가 충분히 큰 경우에 대한 분석
- 이미 알고 있는 점근적 개념의 예

$$\lim_{n \to \infty} f(n)$$

- O, Ω, Θ 표기법

Asymptotic Notation

- $O(f(n))$: Big-Oh
  - 기껏해야 $f(n)$의 비율로 증가하는 함수들의 집합
  - e.g., $O(n)$, $O(n \log n)$, $O(n^2)$, $O(2^n)$, …
  - $g(n) \in O(f(n))$을 관행적으로 $g(n) = O(f(n))$이라고 쓴다
  - 상수 비율의 차이는 무시
  - Upper bound in running time

- 예, $O(n^2)$
  - $3n^2 + 2n$
  - $7n^2 - 100n$
  - $n\log n + 5n$
  - $3n$
- 알 수 있는 한 최대한 tight 하게
  - $n\log n + 5n = O(n\log n)$ 인데 굳이 $O(n^2)$으로 쓸 필요없다
  - 엄밀하지 않은 만큼 정보의 손실이 일어난다

$\Omega(f(n))$

- 적어도 $f(n)$의 비율로 증가하는 함수
- $O(f(n))$과 대칭적
- Lower bound in running time

$\Theta(f(n))$

- $f(n)$의 비율로 증가하는 함수
- $\Theta(f(n)) = O(f(n)) \cap \Omega(f(n))$

Search in an Array

- Want to find an element with value x in an array of size n
  - Unsorted array
    - Have to check all of the n elements
    - $O(n)$
  - Sorted array
    - Don’t have to check all of the elements
    - Binary search: $O(\log n)$

Binary Search, Non-Recursive

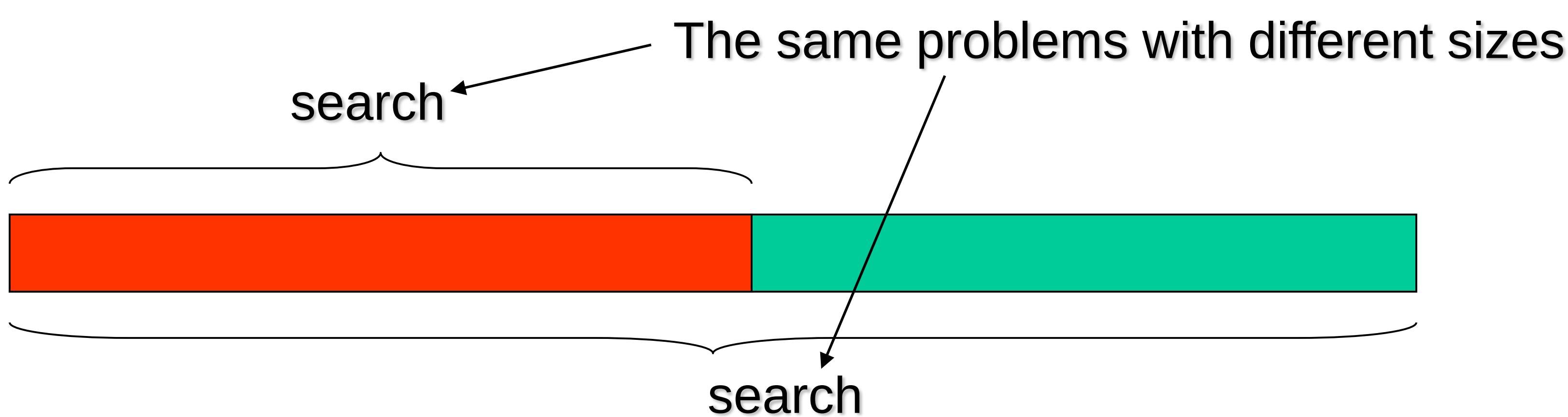
```
BinarySearch (A[ ], n, x)
        // A: array
        // n: # of elements
        // x: search key
{
        low = 0;
        high = n-1;
        while (low ≤ high) {
                mid = (low + high)/2;
                if (A[mid] < x) low = mid + 1;
                else if (A[mid] > x) high = mid – 1;
                else return mid;
        }
        return "Not found";
}
```

✓ Time: $O(\log n)$

Binary Search, Recursive

```
BinarySearch (A[ ], x, low, high)
        // A: array
        // x: search key
        // low, high: array bounds
{
        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;
}
```

✓ Time: $O(\log n)$



Factorial

- $n! = 1 \cdot 2 \cdot 3 \cdots n$
- $n! = n \cdot (n\text{-}1) \cdot (n\text{-}2) \cdots 1$

  $= n \cdot (n\text{-}1)!$

✓ Recursive structure of factorial

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

```
fact (n)
{
        tmp = 1;
        for (i=1; i ≤ n; i++) {
                tmp = i * tmp;
        }
        return tmp;
}
```

Non-recursive

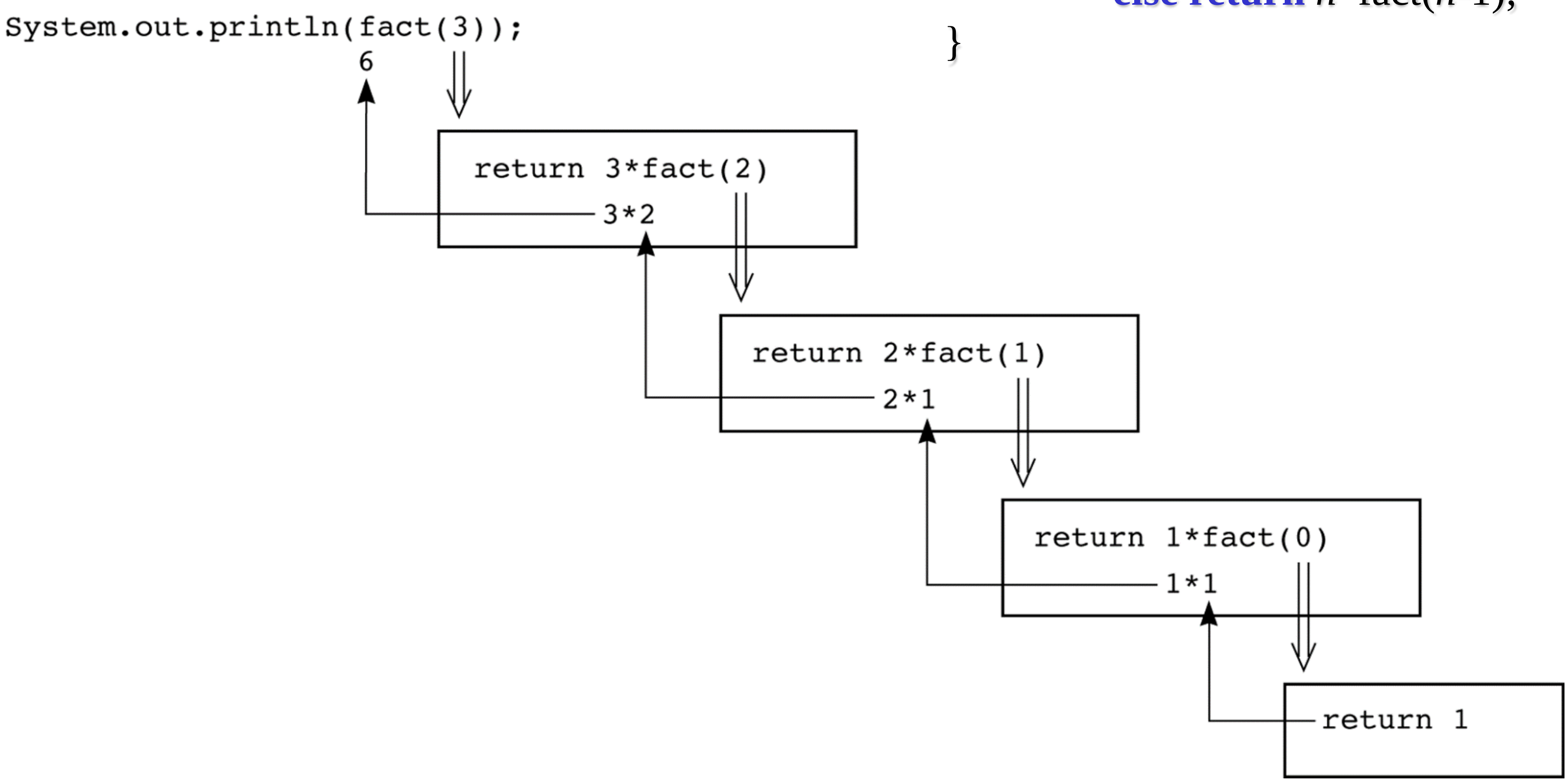
$\theta(n)$

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

Recursive

fact(3)

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



Fibonacci Sequence

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

```
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);
}
```

← Recursive

Simple, but terrible!

```
fib (n)
{
        t[1] = t[2] = 1;
        for (i=3; i ≤ n; i++) {
                t[i] = t[i-1]+t[i-2];
        }
        return t[n];
}
```

$\theta(n)$

← Non-recursive

Writing a String Backward

- Print a string in reverse order
- E.g., ACTGCC → CCGTCA

```
writeBackward (s)
// s: input string
{
        if (s is empty) { do nothing }
        else {
                write the last character of s;
                writeBackward (s minus its last character);
        }
}
```

CCGTCA

✓ Another Method for the Same Result

```
writeBackward (s)
// s: input string
{
    if (s is empty) { do nothing }
    else {
        write the last character of s;
        writeBackward (s minus its last character);
    }
}
```

```
writeBackward2 (s)
// s: input string
{
    if (s is empty) { do nothing }
    else {
        writeBackward2 (s minus its first character);
        write the first character of s;
    }
}
```

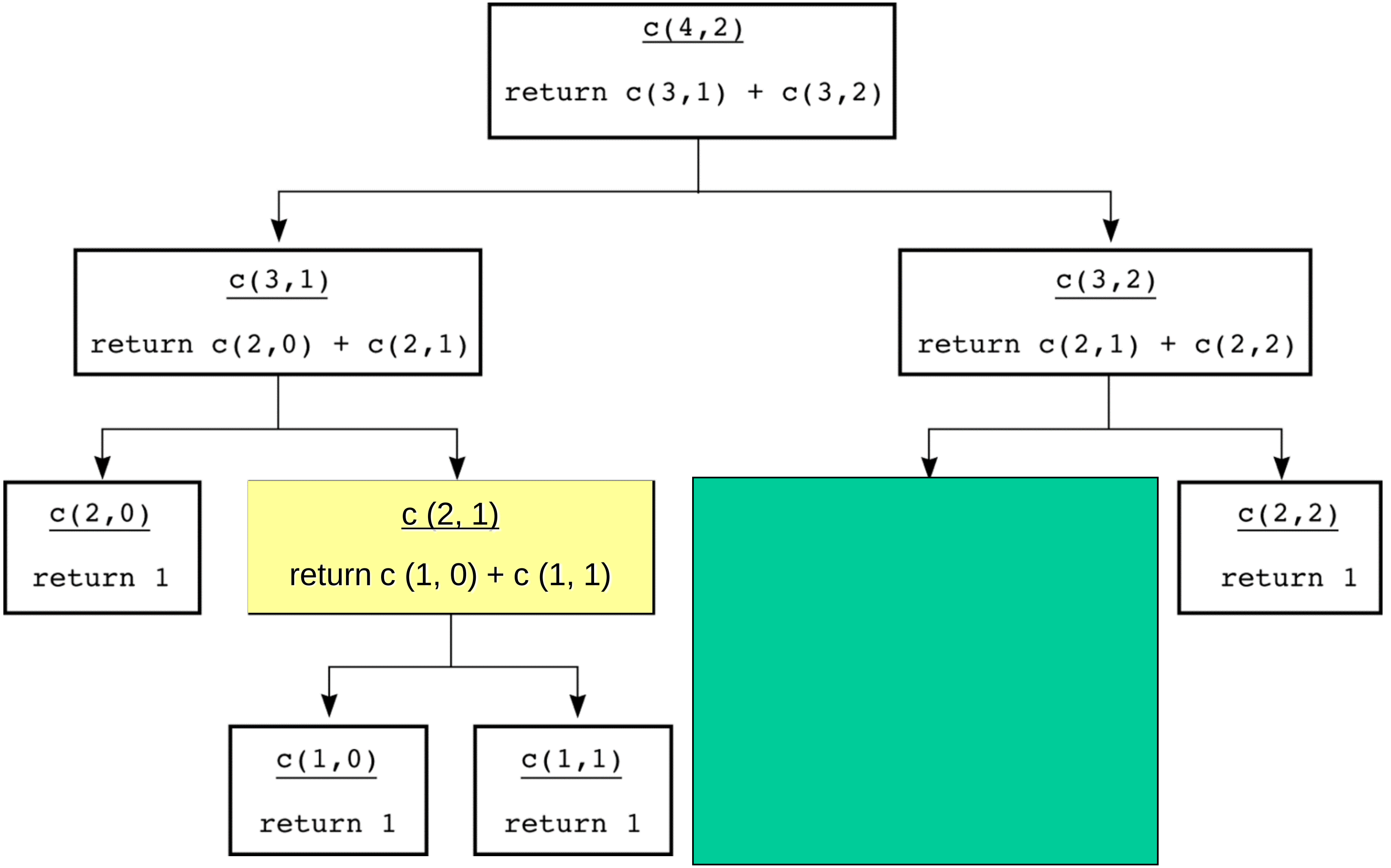
C(*n*, *k*)

$$C(n, k) = \begin{cases} 1 & \text{if } k = 0 \\ 1 & \text{if } k = n \\ 0 & \text{if } k > n \\ C(n-1, k-1)+C(n-1, k) & \text{if } 0 < k < n \end{cases}$$

```
C (n, k)
{
        if (k = 0 or k = n) return 1;
        else if (k > n) return 0;
        else return (C(n-1, k-1)+C(n-1, k));
}
```

Bad example of using recursion

The recursive calls that *c(4, 2)* generates



k^{th} Smallest Element in an Array

- Want to find the k^{th} smallest element in an unsorted array
- E.g., 7^{th} element in {10, 24, 3, 49, 55, 22, 4, 19, 24, 98, 42} ?

Elementary-School Version

```
KthSmallest (A[ ], k, n)
// find the kth smallest in A[1…n]
{
        Find the smallest item min in A[1…n];
        Remove min from the array and compact the remaining
                                        items in A[1…n-1];
        if (k = 1) return min;
        else return KthSmallest(A, k-1, n-1);
}
```

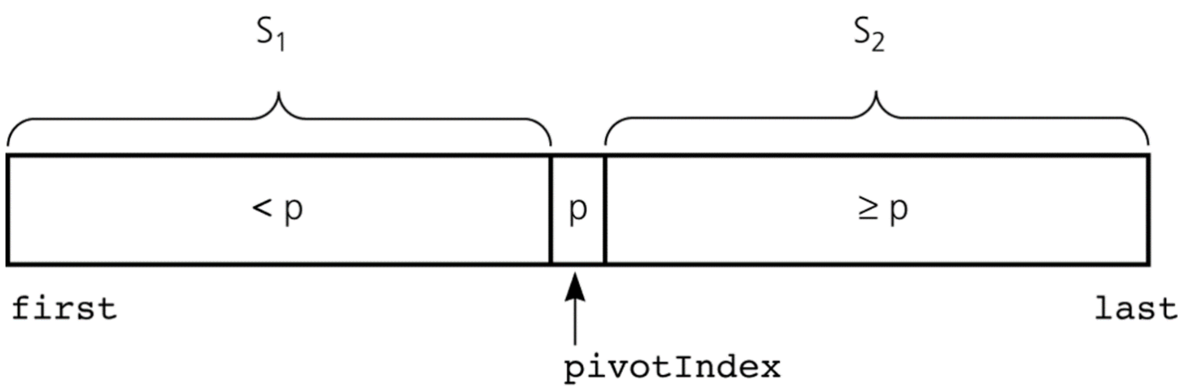
✓ Time: $\Theta(kn)$

Better Version

```
KthSmallest (A[ ], k, first, last)
// find the kth smallest in A[first…last]
{
        Select a pivot item p;  // p ∈ {A[first], … , A[last]}
        pivotIndex = partition (A, p, first, last); // pivot item이 위치한 자리
        if (k < pivotIndex – first + 1) // # of items not greater than p
                return KthSmallest(A, k, first, pivotIndex –1);
        else if ( k = pivotIndex – first + 1) return p;
        else return KthSmallest(A, k-(pivotIndex–first+1), pivotIndex + 1, last);
}
```

partition (A[ ], *p*, *first*, *last*)
{
In the array A[*first*…*last*],
put the items less than *p* in the left side of *p*,
put the items greater than *p* in the right side of *p*,
and return the index of *p*.
}

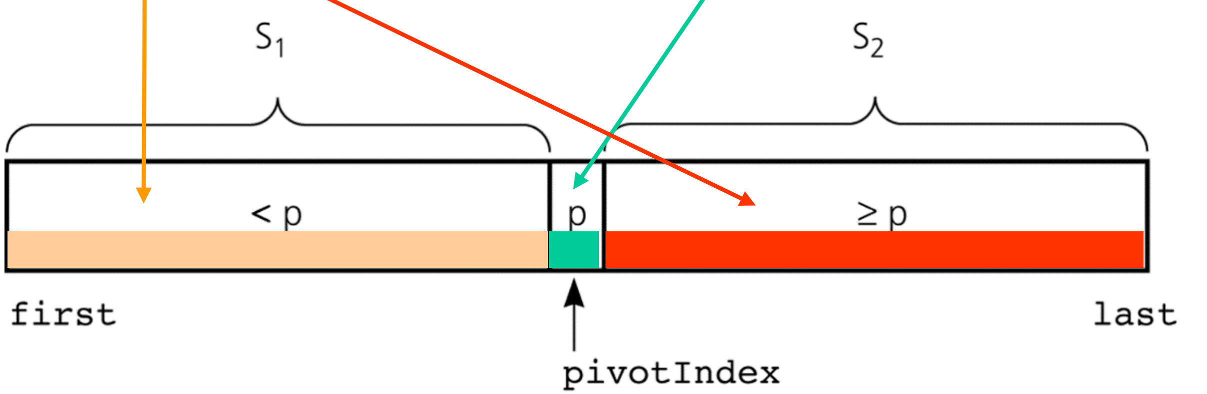
A[*first*…*last*] :



✓ Time

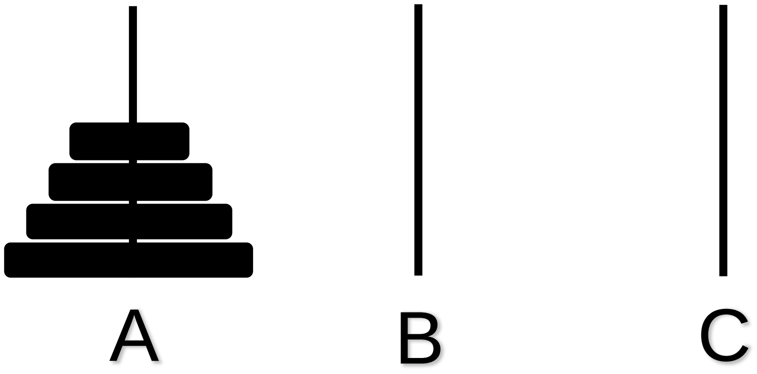
- Average case: $\Theta(n)$
- Worst case: $\Theta(n^2)$

KthSmallest (A[ ], *k*, *first*, *last*)
// find the k^{th} smallest in A[*first*…*last*]
{
  Select a pivot item *p*;
  pivotIndex = partition (A, *p*, *first*, *last*);
  if ($k <$ *pivotIndex* – *first* + 1) // # items not greater than *p*
    return KthSmallest(A, *k*, *first*, *pivotIndex* –1);
  else if (*k* = *pivotIndex* – *first* + 1) **return** *p*;
  else return KthSmallest(A, *k*-(*pivotIndex*–*first*+1), *pivotIndex* + 1, *last*);
}



Hanoi Tower

- *n* disks, 3 poles (A, B, C)
- Move only one disk at a time.
- A bigger disk cannot lie over a smaller one.
- Objective
  - Move *n* disks in pole A to pole B

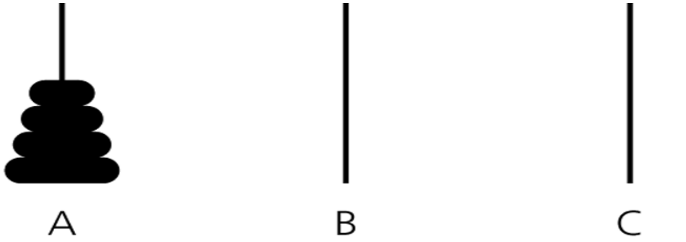


- Function definition

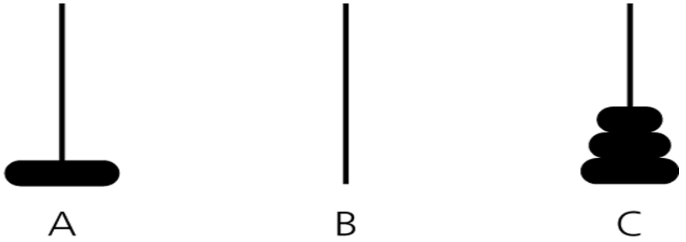
  move(n, *source, destination*, *spare*)

  // move n disks from pole *source* to pole *destination*

- Objective: move(4, A, B, C)



move(3, A, C, B)
move(1, A, B, C)
move(3, C, B, A)



move(n, A, B, C)

{

move(n-1, A, C, B);

move(1, A, B, C);

move(n-1, C, B, A);

}

어떤 문제가 발생하는가?

Keys of Recursive Solution

- Define the problem in terms of smaller problems of the same type (recursive structure)
- Base case (for termination)

fib (n)
{
base case
return fib(n-1)+fib(n-2);
}

move(n, A, B, C)

{

  if (n=1) **then** move the disk from A to B;

  else {

    move(n-1, A, C, B);

    move(1, A, B, C);

    move(n-1, C, B, A);

  }

}