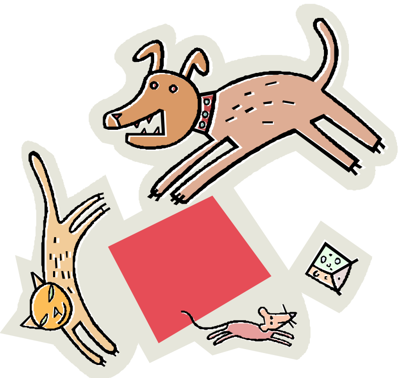
Data Structures

Lecture 2: Recursion

Dongbo Min
Department of Computer Science and Engineering
Ewha Womans University, Korea
E-mail: dbmin@ewha.ac.kr

Recursion

- Method that solves the problem by calling the algorithm (or function) back
- A suitable method for circular definition
- Examples



Factorial computation

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

Fibonacci series

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

Binomial coefficient

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

Factorial Programming

- Definition of Factorial

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

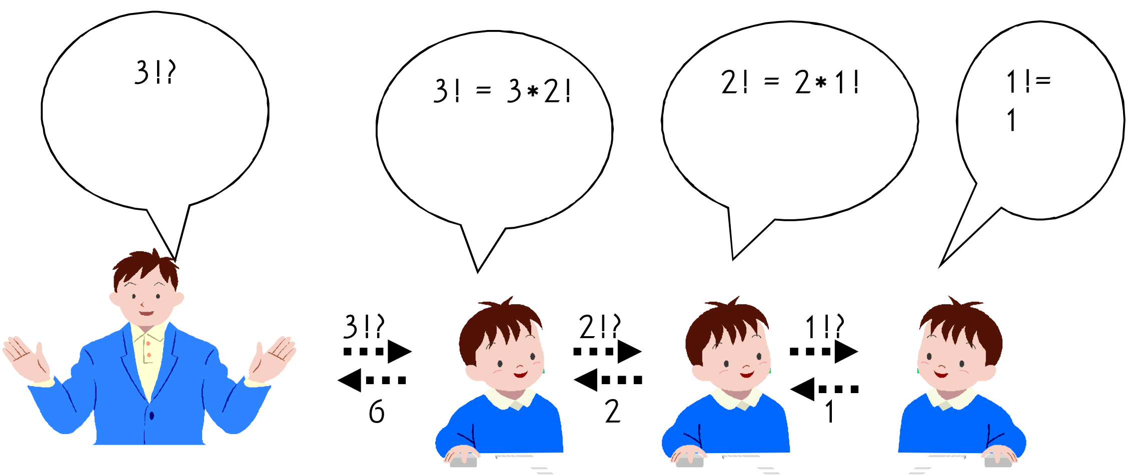
- Implementation 1
  - Use ‘factorial_n()’, ‘factorial_n_1()’, ‘factorial_n_2()’…

```
int factorial_n(int n)
{
    if( n<= 1 ) return(1);
    else return (n * factorial_n_1(n-1) );
}
```

Factorial Programming

- Implementation 2
  - Using recursion with a single function ‘factorial()’

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



Call in Recursion

- Call order in factorial

```
factorial(3)
{
    if( 3 <= 1 ) return 1;
    else return (3 * factorial(3-1) );
}
```

④ ①

```
factorial(2)
{
    if( 2 <= 1 ) return 1;
    else return (2 * factorial(2-1) );
}
```

③ ②

```
factorial(1)
{
    if( 1 <= 1 ) return 1;
            .....
}
```

factorial(3) = 3 * factorial(2)
= 3 * 2 * factorial(1)
= 3 * 2 * 1
= 6

Operation in ①, ②

1. Save the return address at system stack
2. Allocate parameters and local variables from system stack
3. Jump to the address of the called function

Operation in ③, ④

1. Call the return address from system stack
2. Go back to the call function

Recursion Structure

- The recursive algorithm includes the following parts.
  - The part that makes the recursive call
  - The part that stops the recursive call

```
int factorial(int n)
{
    if( n <= 1 )  return 1                  ← Stop the recursion

    else return n * factorial(n-1);         ← Call the recursion
}
```

Q: What if there is no part that stops the circular call?
A: Until a system error occurs, it is called indefinitely.

Recursion Principle

- Divide-and-conquer
  - Divide the problem into a set of sub-problems
  - The number of sub-problems can be >=1.

1) Divide problem

2) Divide problem

3) Divide problem

Max in [2, 5, 3, 1]? Max in [5, 3, 1]? Max in [3, 1]? Max in [1]?

7) Compare 2 and 5, then return 5

6) Compare 5 and 3, then return 5

5) Compare 3 and 1, then return 3

4) Return 1

Recursion vs. Iteration

Recursion: using recursive calls   Iteration: using ‘for’ or ‘while’ loop

- Most recursion can be implemented in a form of iteration.

- Recursion
  - \+ Good choice for recursive problems (*easy* to implement)
  - \- Overhead of function calls → Usually *slower* execution time

- Iteration
  - \+ *Fast* execution time
  - \- Programming can be often very *difficult* for recursive problems.

Recursion vs. Iteration

- What is the best strategy?
  - It depends on problems.

Explain the algorithm using the recursion.
Then, implement the algorithm using the iteration.

Ex 1) Factorial computation

Time complexity:	recursion = iteration
Memory and call overhead:	recursion > iteration
Total complexity:	recursion > iteration

Ex 2) Power computation

Time complexity:	recursion $O(logn)$ < iteration $O(n)$
Memory and call overhead:	recursion > iteration
Total complexity:	recursion < iteration

Ex 3) Fibonacci series

Time complexity:	recursion > iteration
Memory and call overhead:	recursion > iteration
Total complexity:	recursion > iteration

Iterative Implementation of Factorial

$$n! = \begin{cases} 1 & n = 1 \\ n*(n-1)*(n-2)*\cdots*1 & n \geq 2 \end{cases}$$

```
int factorial_iter(int n)
{
    int k, v=1;
    for(k=n; k>0; k--)
        v = v*k;
    return(v);
}
```

Factorial: Time Complexity Analysis

- $T(n)$: Complexity with n inputs

```
int factorial_iter(int n)
{
    int k, v=1;
    for(k=n; k>0; k--)
        v = v*k;
    return(v);
}
```

Iteration

$$\Rightarrow T(n) = O(n)$$

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

Recursion

$$T(n) = T(n-1) + 1 \Rightarrow T(n) = O(n)$$

Power Computation

- The problem of finding the n-squared value of x: x^n
- Example that recursion is more efficient than the iteration

- Iterative method

```
double slow_power(double x, int n)
{
   int i;
   double r = 1.0;
   for(i=0; i<n; i++)
       r = r * x;
   return(r);
}
```

Power Computation

- Recursion

```
power(x, n)

if n=0
        then return 1;
else if n is even
        then return power(x², n/2);
else if n is odd
        then return x*power(x², (n-1)/2);
```

When n is even

$$power(x,n) = power(x^2, \frac{n}{2})$$

When n is odd

$$power(x,n) = x \cdot power\left(x^2, \frac{n-1}{2}\right) = x \cdot x^{n-1} = x^n$$

Power Computation

- Recursion

```
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);
}
```

- Time complexity
  - When n is the square of 2, the problem is reduced as follows.

$$2^n \rightarrow 2^{n/2} \rightarrow \cdots 2^2 \rightarrow 2^1 \rightarrow 2^0$$

	slow_power (iteration)	power (recursion)
Time complexity	O(n)	$O(\log_2 n)$
Execution time	7.17 sec	0.47 sec

Power Computation: Time Complexity Analysis

- $T(n)$: Complexity with n inputs

Iteration

```
double slow_power(double x, int n)
{
   int i;
   double r = 1.0;
   for(i=0; i<n; i++)
       r = r * x;
   return(r);
}
```

$$\Rightarrow \quad T(n) = O(n)$$

Recursion

```
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);
}
```

c: constant

$$T(n) = T\left(\frac{n}{2}\right) + c$$

$$\Rightarrow \quad T(n) = O(log_2 n)$$

Fibonacci Series

- Recursion is not a good choice.

- Fibonacci Series

Ex) 0,1,1,2,3,5,8,13,21,...

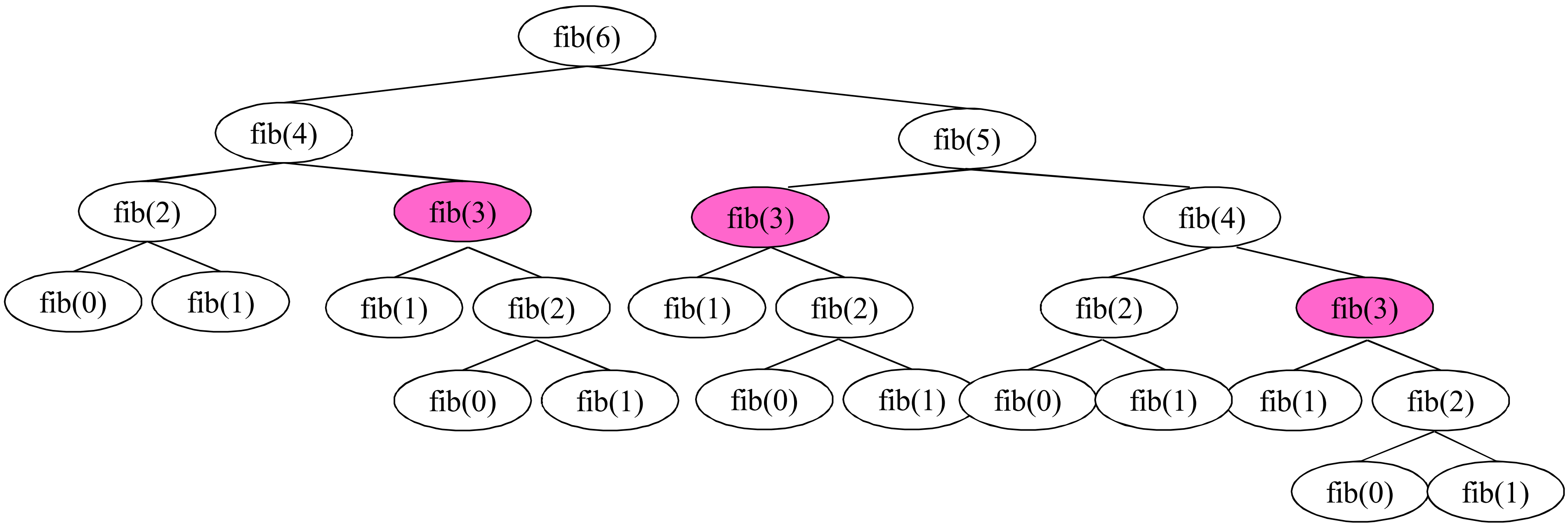
$$fib(n) = \begin{cases} 0 & n = 0 \\ 1 & n = 1 \\ fib(n-2) + fib(n-1) & otherwise \end{cases}$$

- Recursive implementation

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

Fibonacci Series

- Why is the recursion is inefficient for Fibonacci Series?
  - The same terms are computed in duplicate.
    Ex) When calling $fib(6)$, $fib(3)$ will be computed 3 times.
  - It becomes worse when n becomes larger.



For fib(6), maximum depth of tree = 5
→ $T(n) < 2^{n-1} = O(2^n)$

Fibonacci Series

- Iteration

```
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;
        }
}
```

Fibonacci Series: Time Complexity Analysis

- $T(n)$: Complexity with n inputs

Iteration

```
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;
        }
}
```

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

Recursion

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

$$T(n) = T(n-1) + T(n-2) + 1$$

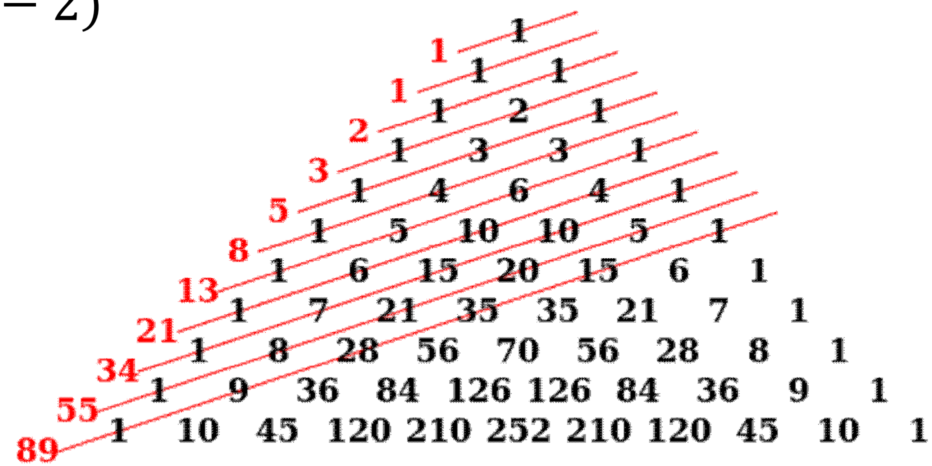
$$T(n) = \sum_{k=0}^{\left\lfloor \frac{n-1}{2} \right\rfloor} \binom{n-k-1}{k} = O(2^n)$$

Fibonacci Series: Time Complexity Analysis

- The Fibonacci numbers
  - occur in the sums of "shallow" diagonals in Pascal's triangle.

$$Fib(n) = Fib(n-1) + Fib(n-2)$$

$$\Rightarrow \quad Fib(n) = \sum_{k=0}^{\left\lfloor \frac{n-1}{2} \right\rfloor} \binom{n-k-1}{k}$$

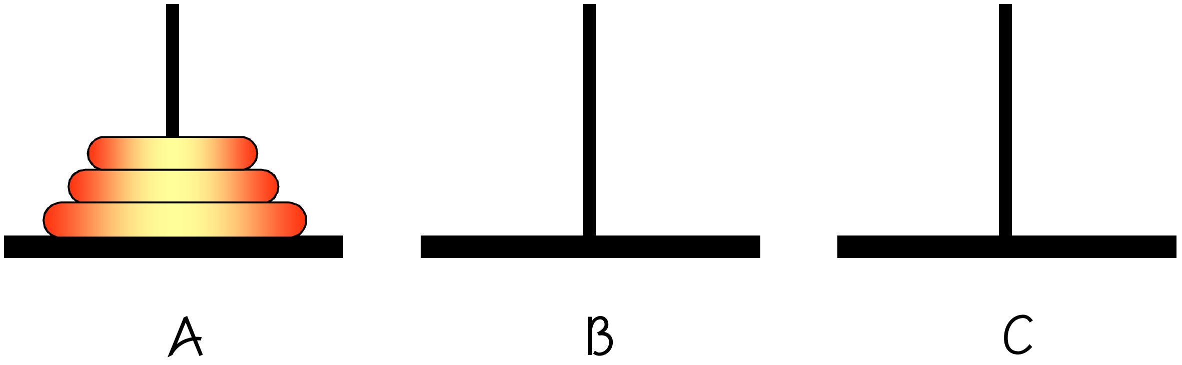


binomial coefficient

for n^{th} polynomial, k^{th} coefficient: $\binom{n}{k}$

Hanoi Tower

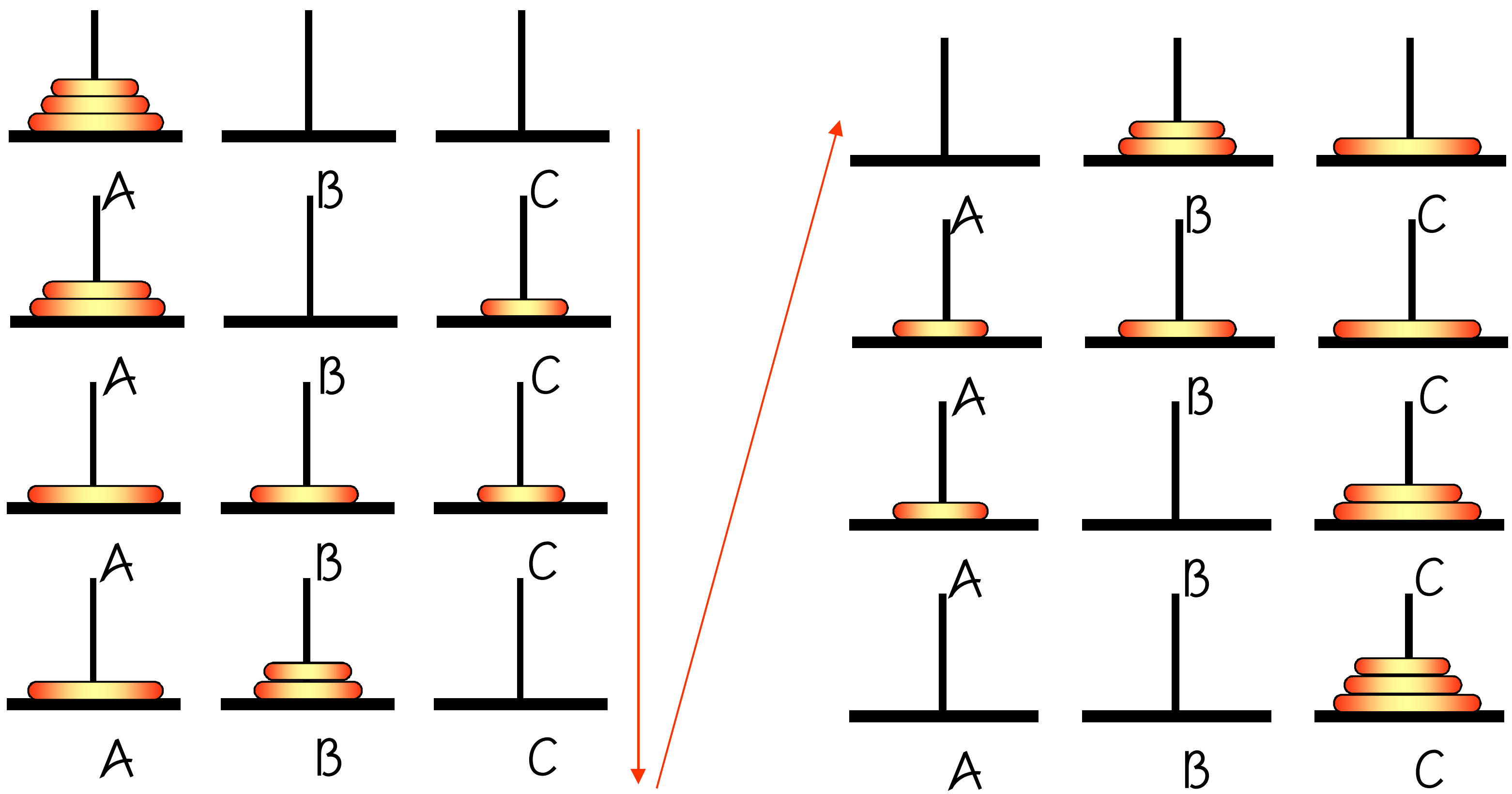
- The problem is to move n discs stacked on rod A to rod C, with the following conditions.
  - Only one disc can be moved at a time
  - Only the top disc can be moved
  - A large disc can not be stacked on a small disc.
  - The middle bar may be used temporarily, but the preceding conditions must be kept.



Hanoi Tower

For 3 discs

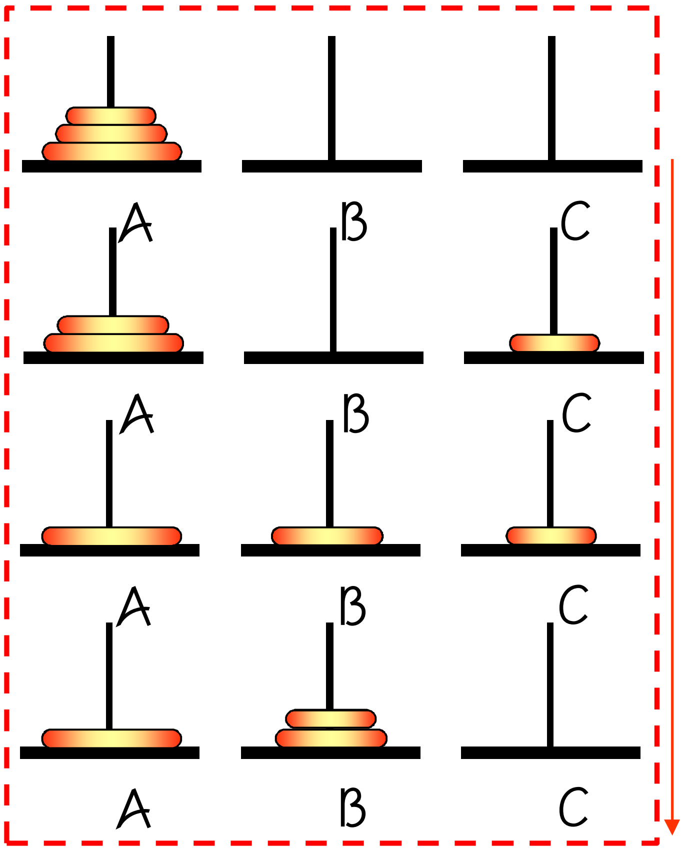
Original problem: A → C for 3 discs



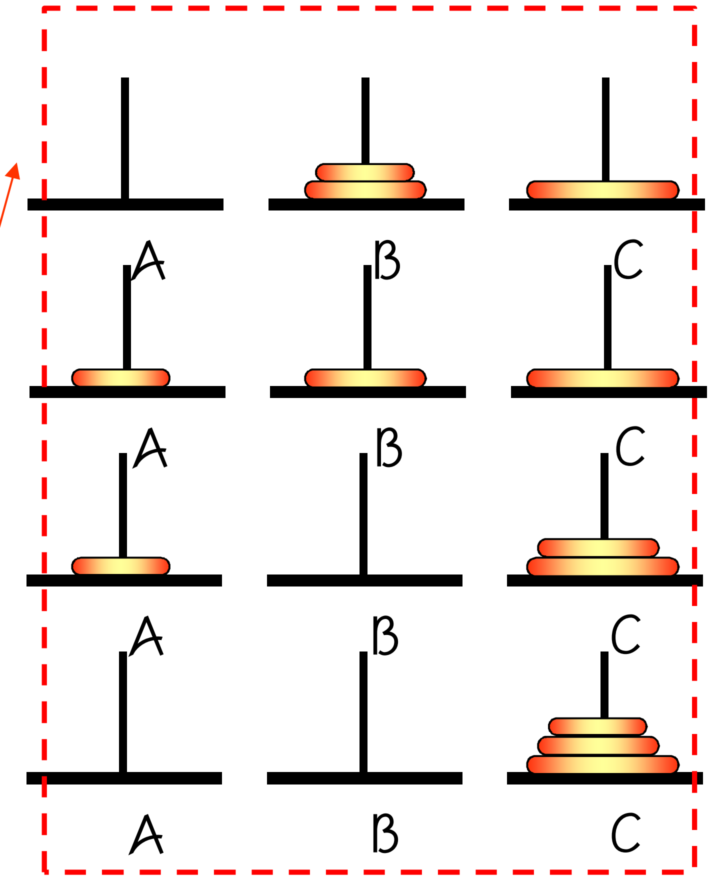
Hanoi Tower

For 3 discs

Original problem: A → C for 3 discs



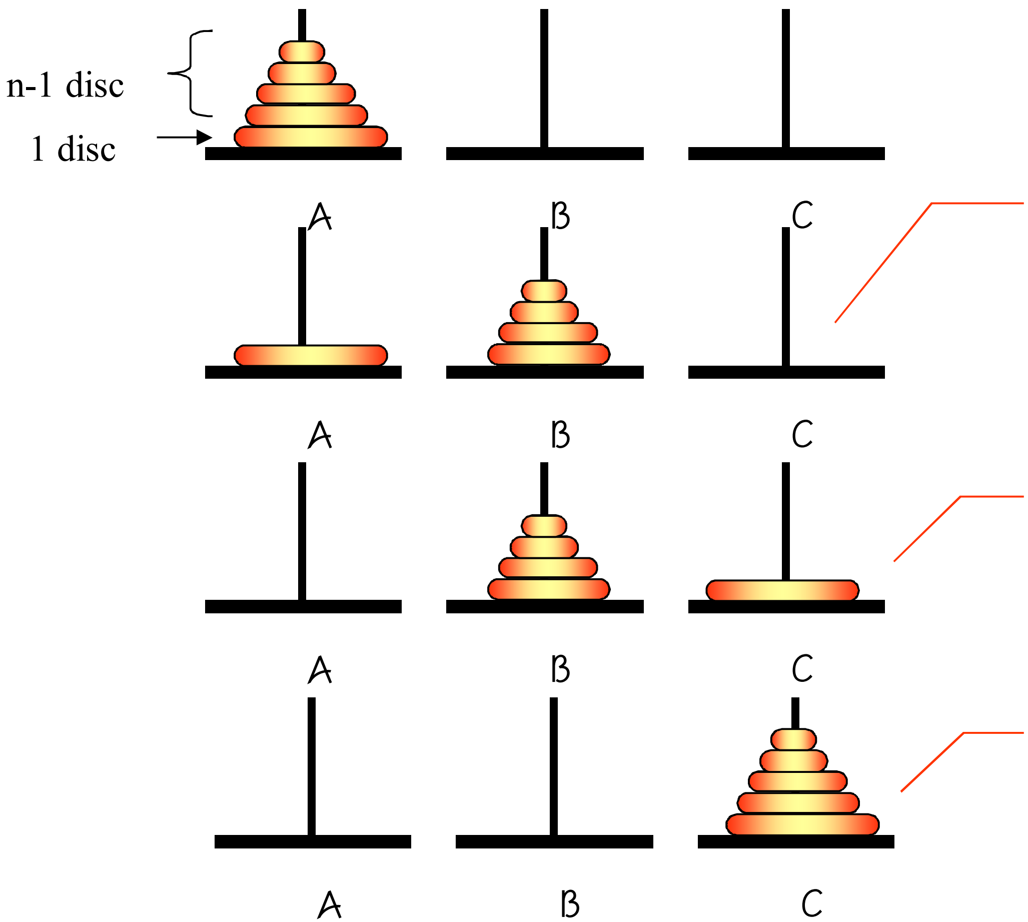
Sub-problem: A → B for 2 discs



Sub-problem: B → C for 2 discs

Hanoi Tower

For n discs



Using C as a temporary buffer
Move n-1 discs stacked in A to B.

Move the largest disc of A to C.

Using A as a temporary buffer,
move n-1 discs in B to C.

Hanoi Tower

- How do you move n-1 discs from A to B and from B to C?

  Note) Our original problem is to move n discs from A to C

→ It is necessary to recursively call the function with n-1 disc as an input.

```
//Move n discs stacked on the bar 'from' to the bar 'to' using the bar 'tmp'.
void hanoi_tower(int n, char from, char tmp, char to)
{
   if (n==1){
      Move a disc 'from' → 'to'
   }
   else{
      hanoi_tower(n-1, from, to, tmp);
      Move a disc at the bar 'from' to the bar 'to'.
      hanoi_tower(n-1, tmp, from, to);
   }
}
```

Hanoi Tower

- Procedure
  1. Move n-1 discs from A to B.
  2. Move n-th disc from A to C.
  3. Move n-1 discs from B to C.

```
#include <stdio.h>
void hanoi_tower(int n, char from, char tmp, char to)
{
  if( n==1 ) printf("Move 1 disc from %c to %c.\n", from, to);
    else {
              hanoi_tower(n-1, from, to, tmp);
              printf("Move disc %d from %c to %c.\n", n, from, to);
              hanoi_tower(n-1, tmp, from, to);
  }
}
main()
{
    hanoi_tower(4, 'A', 'B', 'C');
}
```

Hanoi Tower: Time Complexity Analysis

- $T(n)$: Complexity with n inputs

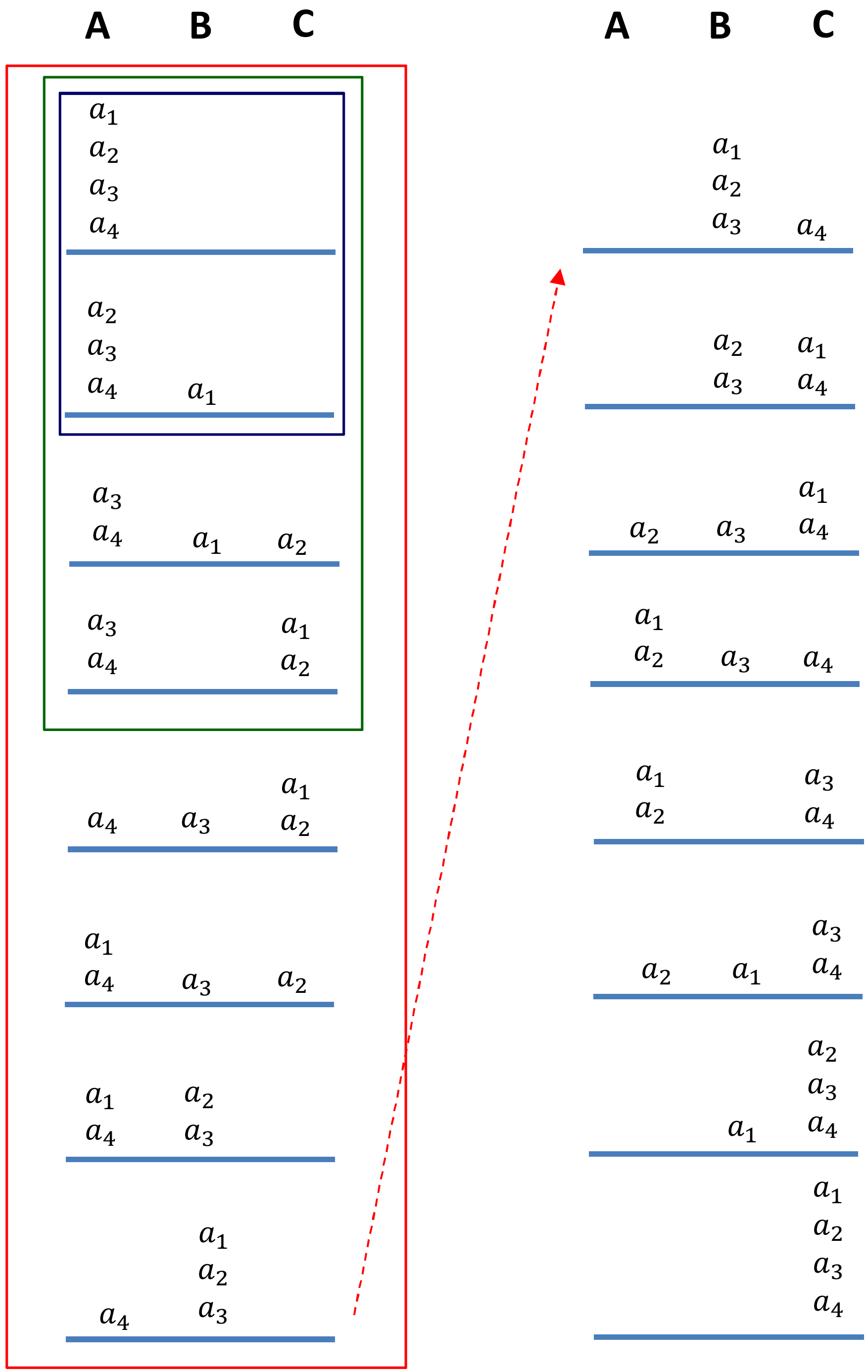
Recursion

```
#include <stdio.h>
void hanoi_tower(int n, char from, char tmp, char to)
{
  if( n==1 ) printf("Move 1 disc from %c to %c.\n", from, to);
    else {
          hanoi_tower(n-1, from, to, tmp);
          printf("Move disc %d from %c to %c.\n", n, from, to);
          hanoi_tower(n-1, tmp, from, to);
  }
}
main()
{
    hanoi_tower(4, 'A', 'B', 'C');
}
```

$$T(n) = 2T(n-1) + 1$$

$$\Rightarrow \quad T(n) = 2^n - 1 = O(2^n)$$

Hierarchical Structure of Recursion
(Divide-and-Conquer)



Original problem
Move 4 discs a_1, a_2, a_3, a_4
from A to C

: Move 3 discs a_1, a_2, a_3
from A to B

: Move 2 discs a_1, a_2
from A to C

: Move 1 discs a_1
from A to B

Note) $T(n) = 2^n - 1$

Hanoi Tower

Homework

- Q: Explain the process when 5 discs are used.

- Q: Implement Hanoi Tower in an iterative manner, and explain what is the complexity, and how many bars are needed.
  (Note that the recursion of Hanoi Tower requires 3 bars only.)

Binary Search

- Problem
  - Input: a set of ordered numbers $\{a_1, \ldots, a_n\}$
  - Goal: query b
  - Output: an index k where $a_k = b$

- Iterative implementation

```
int search_iter(A, b)
    for i=1 to n
        if(A[i] == b)
            k=i;
    return k
```

- Recursive implementation

```
int search_recur(A, b, start, end)
    if(start>end)       return -1;
    int median = (start+end)/2;
    if(A[median]<b)
        search_recur(A, b, median, end);
    else if(A[median]>b)
        search_recur(A, b, start, median);
    else
        return median;
```

Binary Search: Time Complexity Analysis

- $T(n)$: Complexity with n inputs

```
int search_iter(A, b)
    for i=1 to n
        if(A[i] == b)
            k=i;
    return k
```

Iteration

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

```
int search_recur(A, b, start, end)
    if(start>end)    return -1;
    int median = (start+end)/2;
    if(A[median]<b)
        search_recur(A, b, median, end);
    else if(A[median]>b)
        search_recur(A, b, start, median);
    else
        return median;
```

Recursion

c: constant

$$T(n) = T\left(\frac{n}{2}\right) + c$$

$$T(n) = O(log_2 n)$$

Recursion Types

- Tail recursion: can be easily implemented using iteration

```
return n * factorial(n-1);
```

- Head recursion: is difficult to implement using iteration

```
return factorial(n-1) * n;
```

- Multi-recursion: is difficult to implement using iteration

```
function(A, n)
{
    function(A, n-1)
    function(A, n-1)
}
```

```
function(A, n, p)
{
    if(...)  function(A, n-1, p)
    else     function(A, n-1, q)
}
```

This is NOT a multi-recursion