Lecture 3:
Recursion

Definition: See last slide……

Lecture Outline

- Recursion
  - Basic idea
  - Execution phases of recursion
  - Iteration versus Recursion

  - Examples on recursion
    - Factorial
    - Print Digits
    - Cumulative Sum
    - Fibonacci
    - Tower of Hanoi
    - N choose K

Recursion: **The Basic Idea**

- The process of solving a problem with a function that **calls itself** directly or indirectly
  - The solution can be derived from solution of smaller problem of the same type
- Example:
  - `Factorial(4) = 4 * Factorial (3)`
- This process can be repeated
  - E.g. `Factorial(3)` can be solved in term of `Factorial(2)`
- Eventually, the problem is so simple that it can solve immediately
  - E.g. `Factorial(0) = 1`
- The solution to the larger problem **can then be derived from this …**

Recursion: **The Main Ingredients**

- To formulate a recursive solution:
  a. Identify the **simplest** instance (**base case**)

     ➔ can solve the problem **without recursion**

  b. Identify **simpler** instances of the **same problem** (**recursive case**)

     ➔can make recursive calls to solve them

  c. Identify how the solution from the simpler problem can help to **construct the final result**

  - Ensure we are able to reach base case
    - So that we will not get an **infinite recursion**

Example : **Factorial**

- Lets write a recursive function ***`factorial`*** **`(k)`** that finds **`k!`**

  - **Base Case:**
    - Return `1` when **`k = 0`**
    - C code:

```
if( k == 0 )
   return 1;
```

  - **Recursive Case:**
    - Return **`k * (k-1)!`**

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

Example : Factorial (code)

- Full code for `factorial`:

```
int factorial( int k )
{
    if ( k == 0 ){
        return 1;
    } else {
        return k * factorial( k-1 );
    }
}
```

Base Case:
factorial(0) = 1

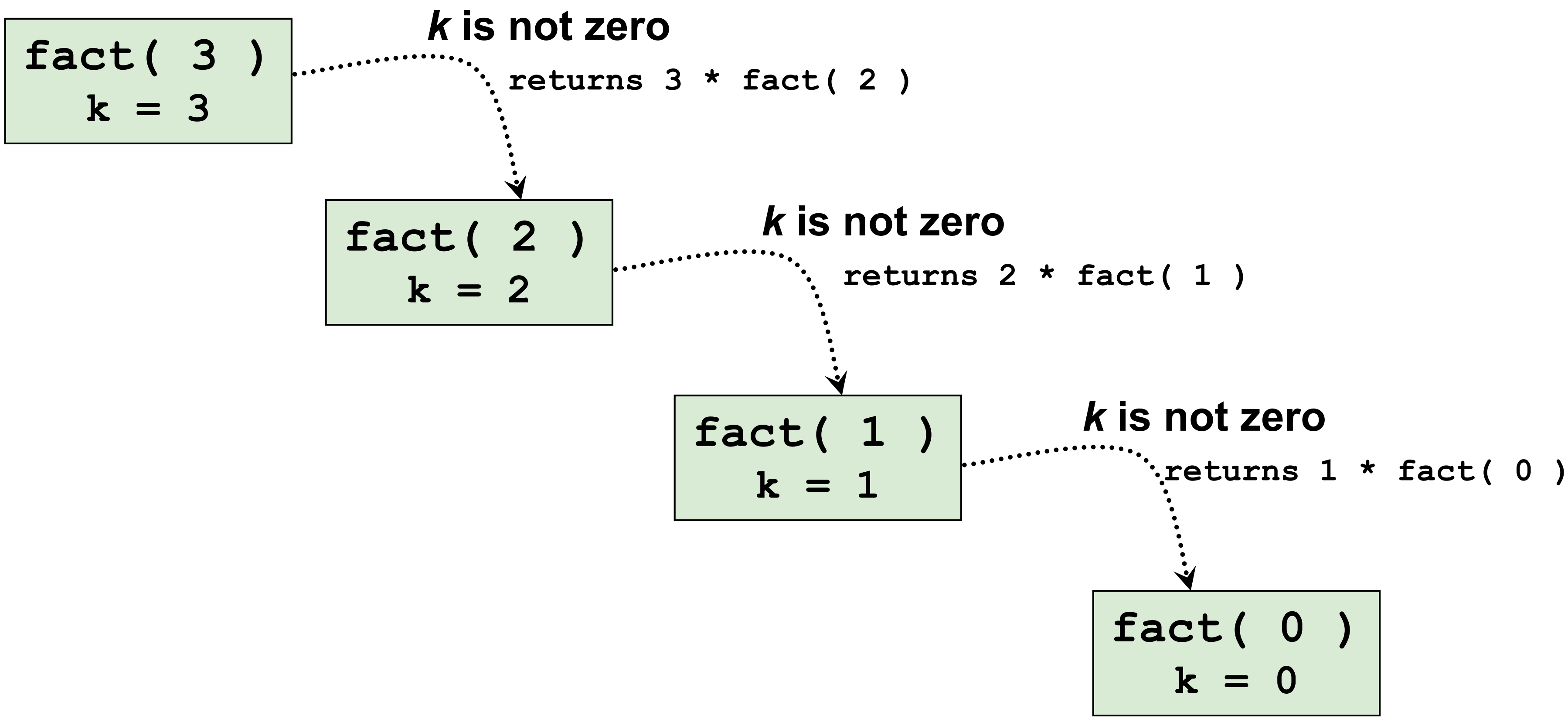
Recursive Case:
factorial(k) =
k * factorial(k – 1)

Understanding Recursion

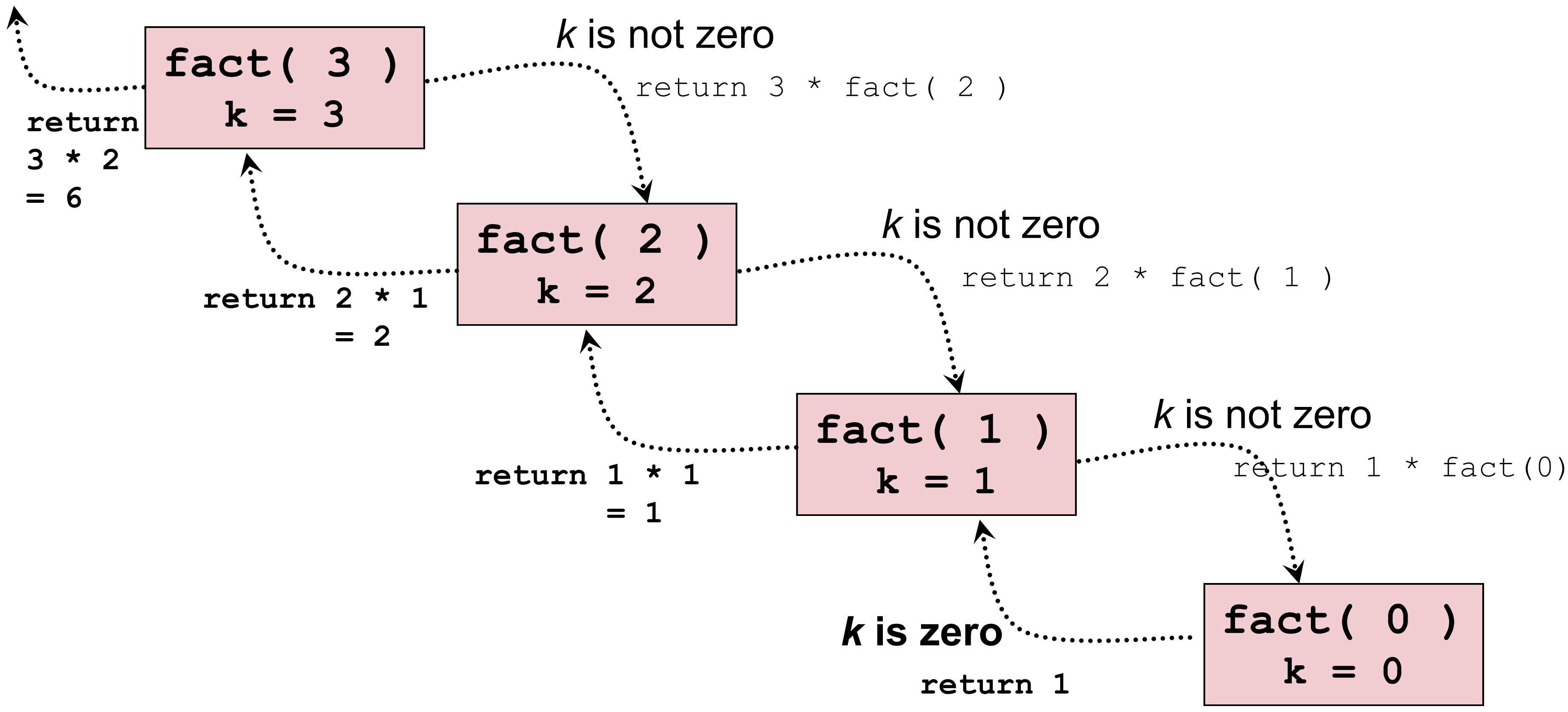
- A recursion always goes through two phases:
  - A **wind-up phase**:
    - When the **base case** is *not* satisfied i.e. the function calls itself
    - This phase carries on **until** we reach the **base case**
  - An **unwind phase**:
    - The recursively called functions return their values to previous “instances” of the function call
    - i.e. the last function returns to its parent (the 2nd last function), then the 2nd last function returns to the 3rd last function etc…
    - Eventually reaches the very first function, which computes the final value.

The Factorial : **Windup Phase**

- Lets trace the execution of `factorial(3)` (`factorial` abbreviated as `fact`)



The Factorial : **Unwind Phase**



Recursion vs. Loops

- Most recursions essentially accomplishes a loop (iterations)

  + Recursions are usually much **more elegant** than its iterative equivalent

  + Recursions are **conceptually simpler and easier to implement**

  - Iterative version using loops is **usually faster**

- **Common practice:**
  - Figure out the solution using recursion
  - Convert to iterative version if possible

Recursive vs. Iterative Versions

```
int factorial( int k )
{
   int j, term;

   term = 1;
   for ( j=2; j<=k; j++ ){
        term *= j;
   }

   return term;
}
```

Iterative Version

```
int factorial( int k )
{
   if ( k == 0 ){
        return 1;
   } else {
        return k * factorial( k-1 );
    }
}
```

Recursive Version

Problem: Print Digits

- Given:
  - A positive integer **N**
- Your task:
  - Write a ***recursive function*** to print out the digits from right (less significant) to left (most significant)

Sample Run:

```
Enter N (positive integer): 12345
5
4
3
2
1
```

Problem: Print Digits (Reverse)

- Given:
  - A positive integer **N**
- Your task:
  - Write a ***recursive function*** to print out the digits from left (most significant) to right (less significant)

Sample Run:

```
Enter N (positive integer): 12345
1
2
3
4
5
```

Problem: Cumulative Sum (Again!)

- Write a **recursive** function to return the cumulative sum:
  - **`Result[i] = sum of A[i] to A[size-1]`**

```
void cumulative( int a[], int result[], int size );
```

a	1	2	3	4	5
result	15	14	12	9	5

- Apply the same principle:
  a) Simplest case?
  b) How to break it into smaller and simpler problem?
  c) How does the answer from (b) form the larger answer?

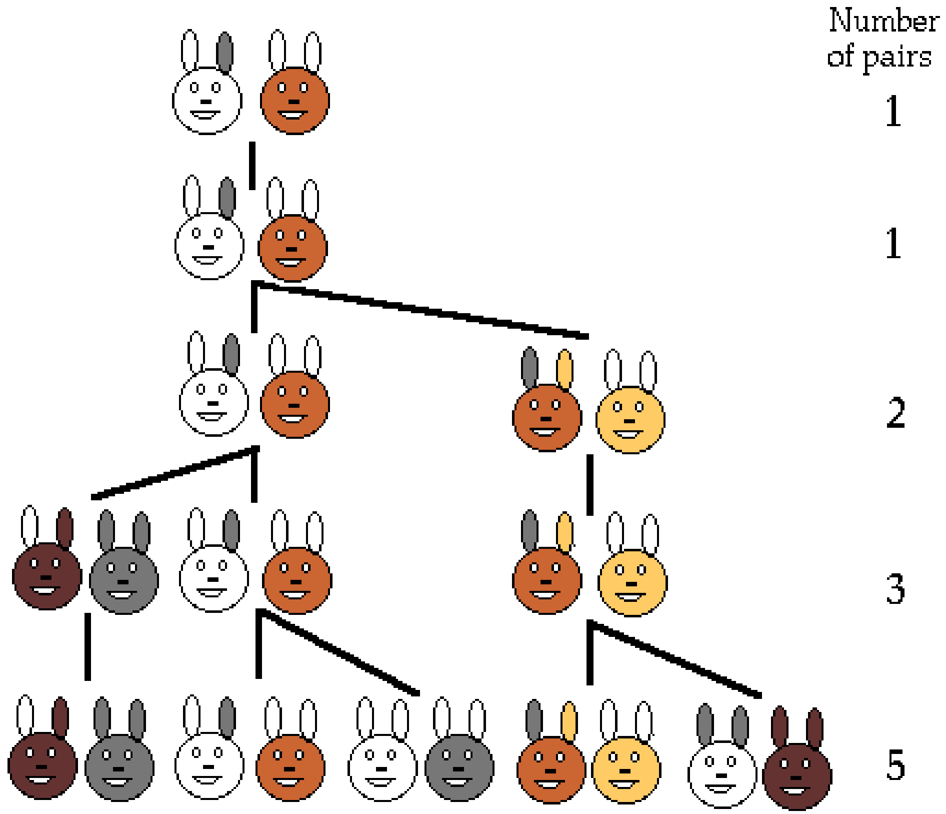
More examples to convince your mind

EXAMPLES AND EXAMPLES

Example : **Fibonacci Number**

Rabbits give birth monthly once they are **3 months old** and they always conceive a **single male/female pair**.

Given a pair of male/female rabbits, assuming rabbits **never die**, **how many pairs of rabbits are there after n months?**



The Fibonacci Series

- `Rabit(N)` = # pairs of rabbit at Nth month
  1. Equal to the total number of rabbit pairs in the previous month (N - 1)th month
     - Rabbits never die
  2. Additionally, new rabbit pairs = the total rabbit pairs two months ago (N - 2)th month
     - Rabbits give birth from the 3rd month onwards

➔ `Rabbit(N) = Rabbit(N – 1) + Rabbit(N – 2)`

- Special cases:
  - `Rabbit( 1 ) = 1` One pair in the 1st month
  - `Rabbit( 2 ) = 1` Still one pair in the 2nd month
- **`Rabbit(N)`** is the famous **`Fibonacci(N)`**

Fibonacci Number: **Implementation**

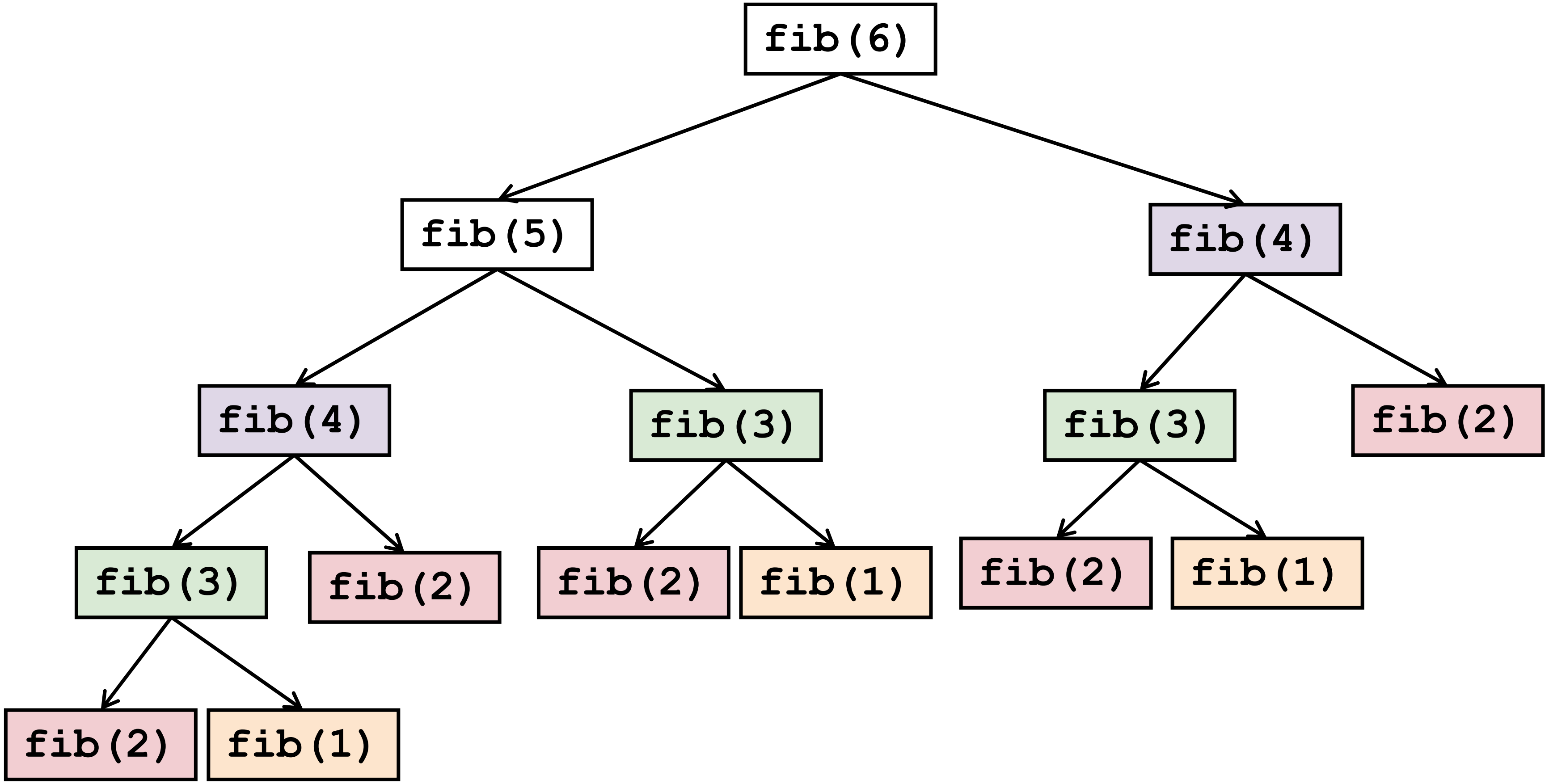
```
int fibo(int n)
{
   if (n <= 2)
       return 1;
   else
       return fibo( n-1 ) + fibo( n-2 );
}
```

Base Cases:
fibo(1) = 1
fibo(2) = 1

Recursive Case:
fibo(n)

= fibo(n – 1) + fibo(n – 2)

Execution Trace: **Fibonacci**



- Many duplicated calls:
  - The **same computations** are done over and over again!

Fibonacci Number : **Iterative Solution**

```
int fibo( int n )
{
   int cur, prev1 = 1, prev2 = 1, j;

   if( n<=2 )
        return 1;
   else
        for ( j=3; j<=n; j++ ) {
            cur = prev1 + prev2;
            prev2 = prev1;
            prev1 = cur;
        }
   return cur;
}
```

Iterative Version

- How many time do we calculate a particular *fibonacci* number?

Observation: **Excessive Recursion**

- The Fibonacci example highlight the following:
  - Recursive solution may looks very simple but can cause a **huge amount** of function calls
  - Should be mindful of such possibilities:
    - When the recursive case makes **more than one recursive calls**
  - Find out an iterative equivalent if possible

Problem: Greatest Common Divisor

- **Given:**
  - Two integers X and Y, find the **largest divisor that divides both of them with no remainder**
- **Your task:**
  - Write a recursive version

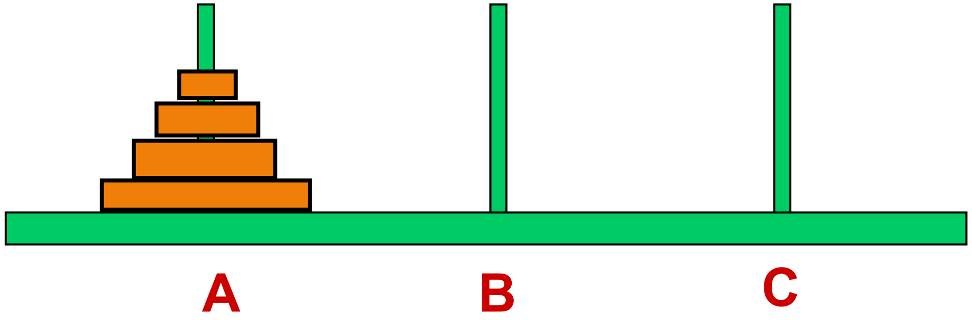
- Remember the ***Euclid Algorithm***:
  - If Y can divide X evenly, then answer is Y
  - If not, the answer is to find GCD of Y and (X % Y)

A bit tough, but **really interesting**

ADVANCED EXAMPLE

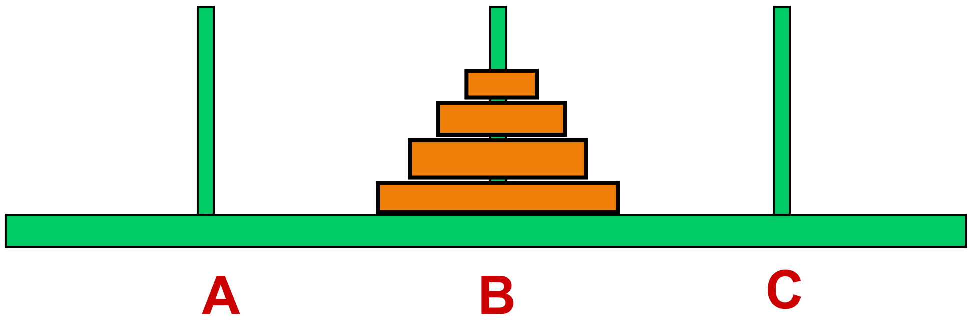
Example: **Tower of Hanoi**

initial state



- How do we move all the disks from pole "**A**" to pole "**B**", using pole "**C**" as temporary storage
  - One disk at a time
  - Disk must rest on top of larger disk

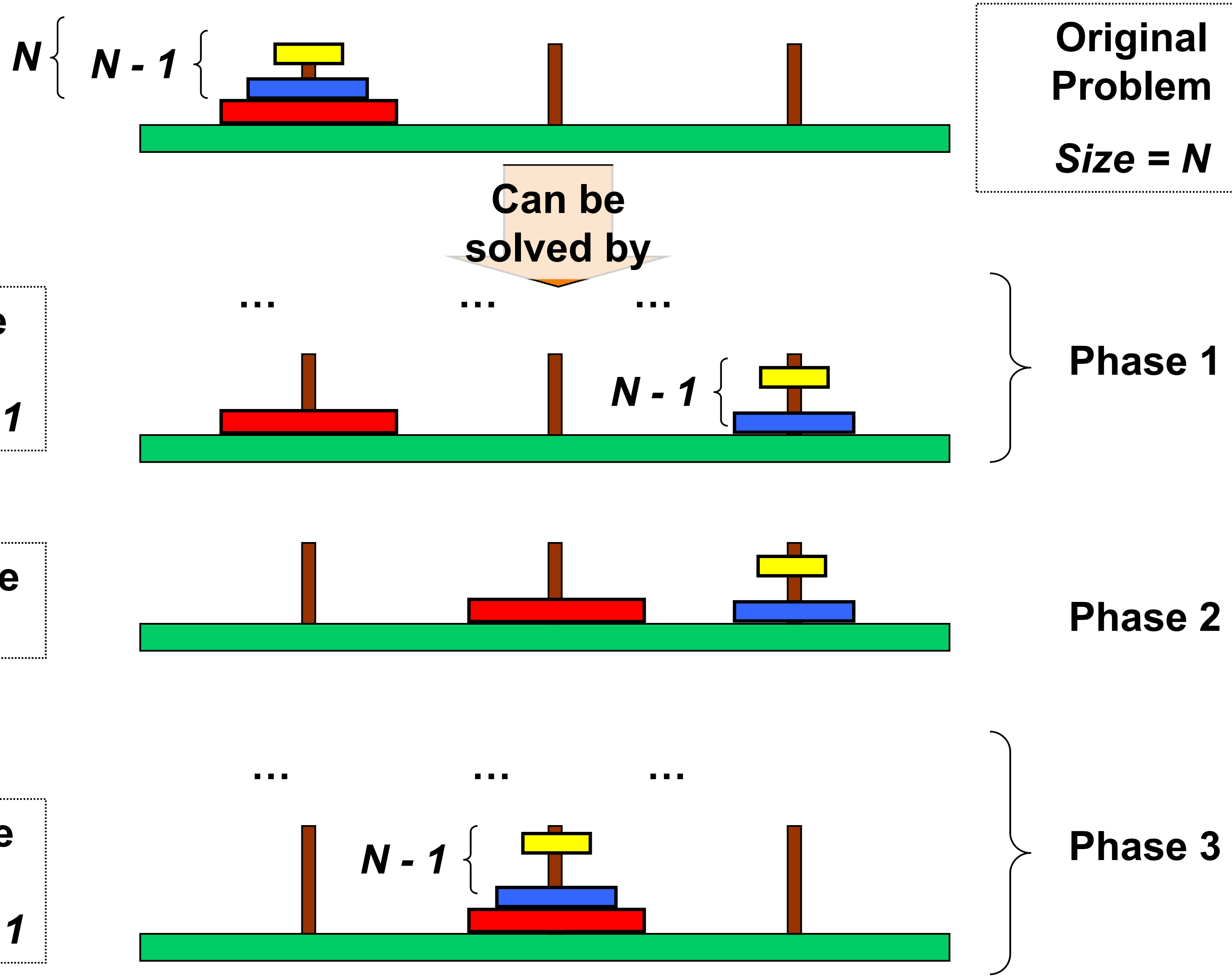
final state



Tower of Hanoi: **Recursive Solution**

Original Problem

Size = N



Recursive Case
Size = N – 1

Base Case
Size = 1

Recursive Case
Size = N - 1

Tower of Hanoi : **Solution**

```
void tower(int N, char A, char B, char C)
{
  if ( N == 1 )
      move( A, B);
  else {
      tower( N-1 , A, C, B);
      move( A, B);
      tower( N-1 , C, B, A);
  }
}
```

Perform the "move"
Many implementations
Below is one possibility

```
void move(char s, char d)
{
    printf( "move from %d to %d\n", s, d );

}
```

Number of moves needed?

Num of discs, n	Num of moves, f(n)	Time (1 sec per move)
1	1	1 sec
2	3	3 sec
3	3+1+3 = 7	7 sec
4	7+1+7 = 15	15 sec
5	15+1+15 = 31	31 sec
6	31+1+31 = 63	1 min
…	…	…
16	65,536	18 hours
32	4.295 billion	136 years
64	1.8 * 10^10 billion	584 billion years

Note the pattern

$f(n) = 2^n - 1$

Slightly more than required for this course [For Exploration]

ADDITIONAL EXAMPLE

Example: **Combinatorial**

- How many **ways** can we choose **k** items out of **n** items?

choose k out of n

X selected → choose k-1 out of n-1

or

X not selected → choose k out of n-1

Recursive Cases

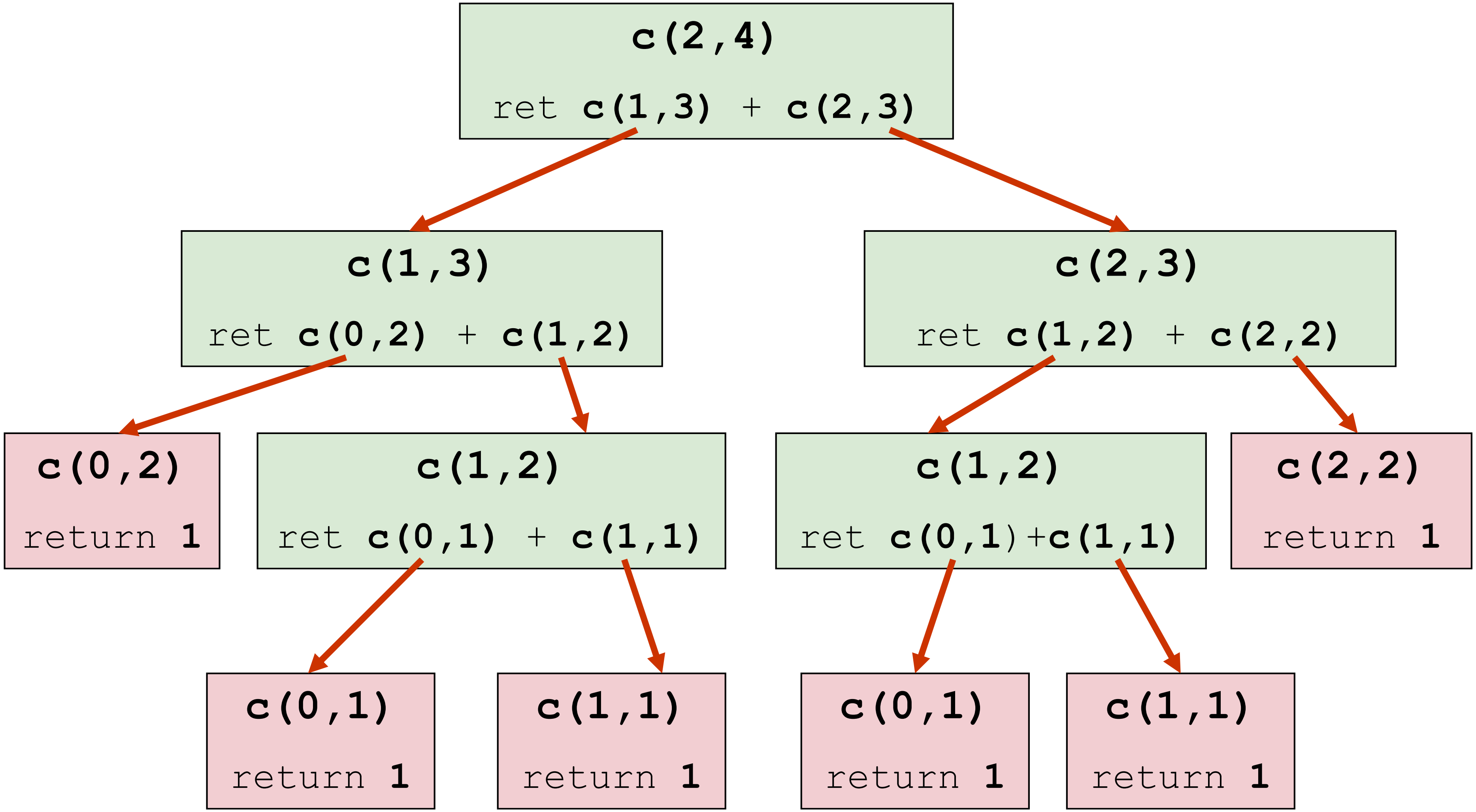
k==n → 1 way

k==0 → 1 way

Base Cases

```
int choose( int k, int n )
{
    if (k>n) return 0;
    if (k==n || k==0) return 1;
    return choose( k-1, n-1)
         + choose( k, n-1);
}
```

Execution Trace : Choose(2, 4)



The final answer is the sum of those base cases

Summary

- Recursive Function
  - Definition
  - How-to
  - Examples
    - Factorial
    - Fibonacci Numbers
    - Print Digits (and in Reverse!)
    - Cumulative Sum
    - Greatest Common Divisor
    - Tower of Hanoi
    - Choose k out of n things

Recursion Definition: See first slide