2. Recursion

자료구조

(Please turn off your mobile phone !)

Agenda

- Introduction
- Recursion Examples
  - Greatest common divisor
  - Fibonacci numbers
  - Reverse keyboard input
- Designing Recursive Algorithms
- Recursion Design Examples
  - Prefix to postfix conversion
  - Towers of Hanoi
- Recursion and Divide-and-Conquer

Introduction

- Approach to repetitive algorithms
  - Iteration (loop)
    - Intuitive
  - Recursion (function call to itself)
    - Less intuitive
    - Suitable for problem breaking-down
      - Divide-and-conquer
    - Many algorithms are drastically simplified by recursion

What is Recursion?

- Recursion: a method (function) calling itself.

- Recursive definition
  - Recursion:

    see Recursion

Factorial – A Case Study

- Iterative definition of factorial

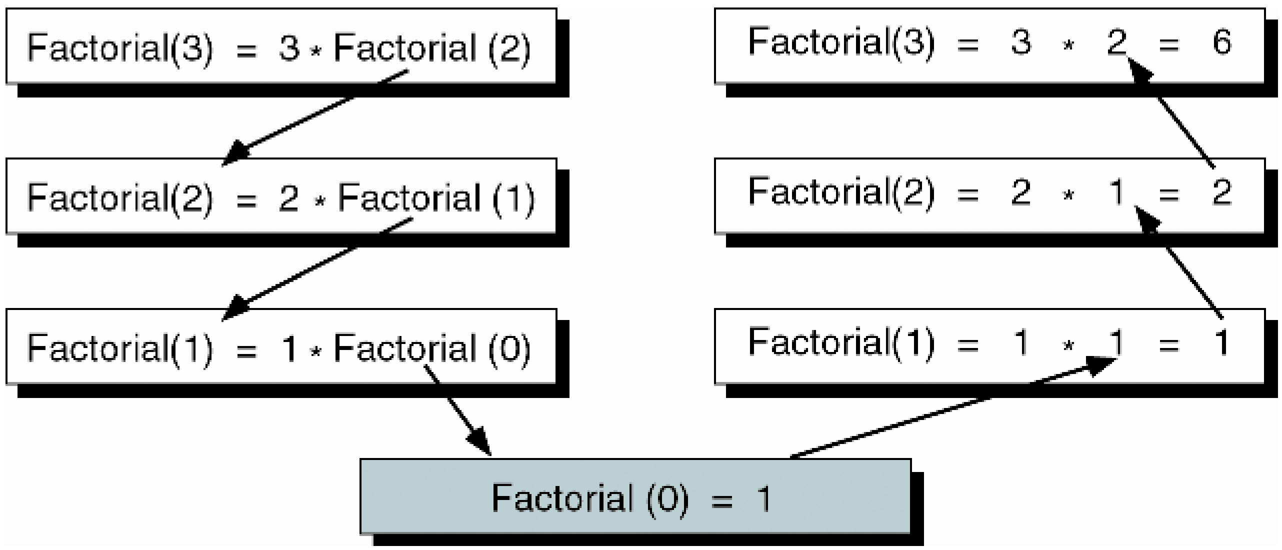
$$\text{Factorial}(n) = \begin{bmatrix} 1 & \text{if } n = 0 \\ n \times (n-1) \times (n-2) \times \ldots \times 3 \times 2 \times 1 & \text{if } n > 0 \end{bmatrix}$$

- Recursive definition of factorial

$$\text{Factorial}(n) = \begin{bmatrix} 1 & \text{if } n = 0 \\ n \times (\text{Factorial}(n-1)) & \text{if } n > 0 \end{bmatrix}$$

Factorial – A Case Study

- Factorial of 3 (by recursion)
  - Factorial(3) = 3 * factorial(2)
    = 3 * 2 * factorial(1)
    = 3 * 2 * 1 * factorial(0)
    = 3 * 2 * 1 * 1 = 6



Iterative Algorithm of Factorial

$$\text{Factorial}(n) = \left[\begin{array}{ll} 1 & \text{if } n = 0 \\ n \times (n-1) \times (n-2) \times \ldots \times 3 \times 2 \times 1 & \text{if } n > 0 \end{array}\right]$$

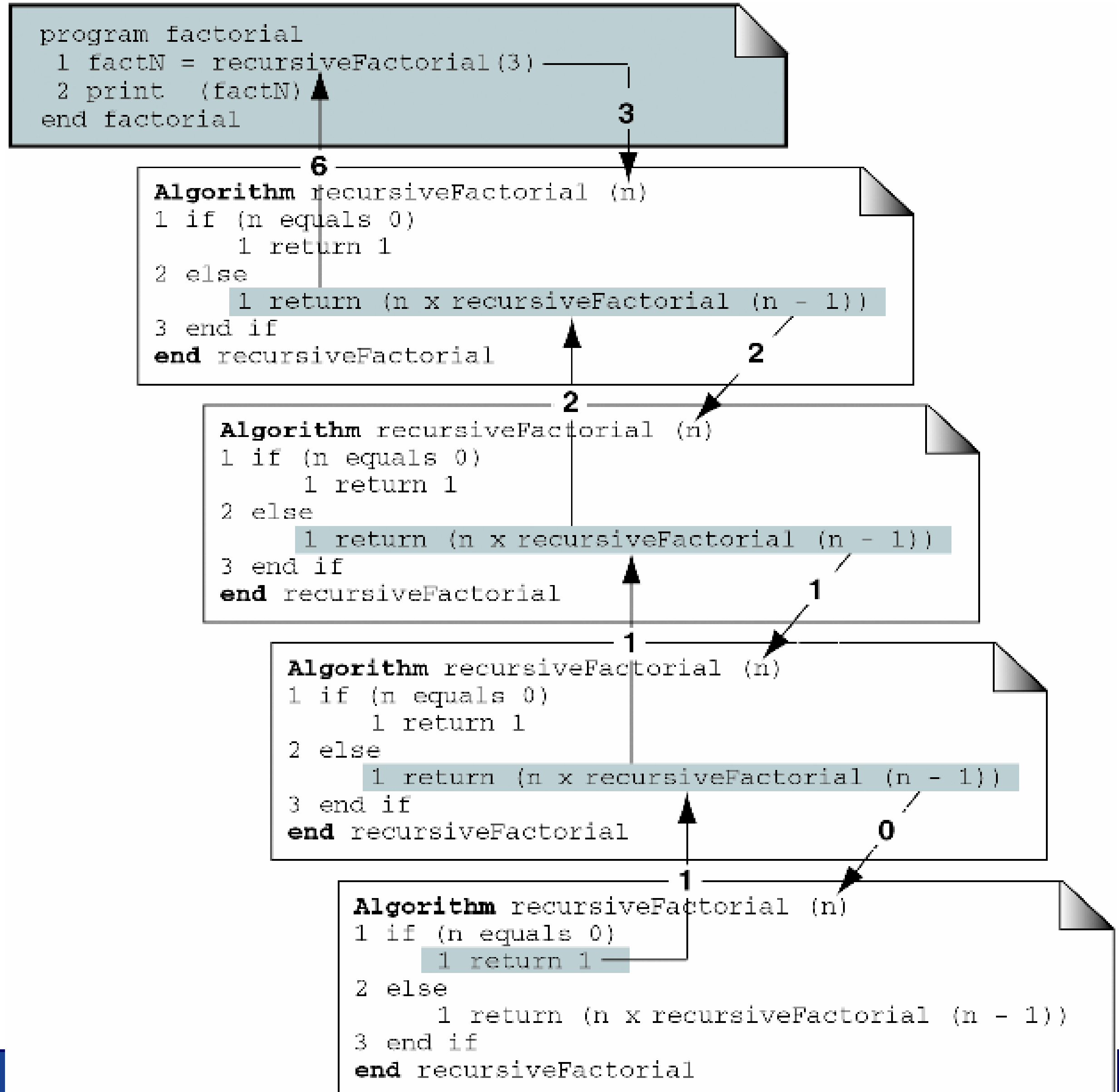
```
Algorithm iterativeFactorial (n)
Calculates the factorial of a number using a loop.
   Pre  n  is the number to be raised factorially
   Post n! is returned
1  set i to 1
2  set factN to 1
3  loop (i <= n)
   1   set factN to factN * i
   2   increment i
4  end loop
5  return factN
end iterativeFactorial
```

Recursive Algorithm of Factorial

$$\text{Factorial}(n) = \begin{bmatrix} 1 & \text{if } n = 0 \\ n \times (\text{Factorial}(n-1)) & \text{if } n > 0 \end{bmatrix}$$

```
Algorithm recursiveFactorial (n)
Calculates factorial of a number using recursion.
   Pre    n  is the number being raised factorially
   Post   n! is returned
1 if (n equals 0)
   1  return 1
2 else
   1  return (n * recursiveFactorial (n - 1))
3 end if
end recursiveFactorial
```

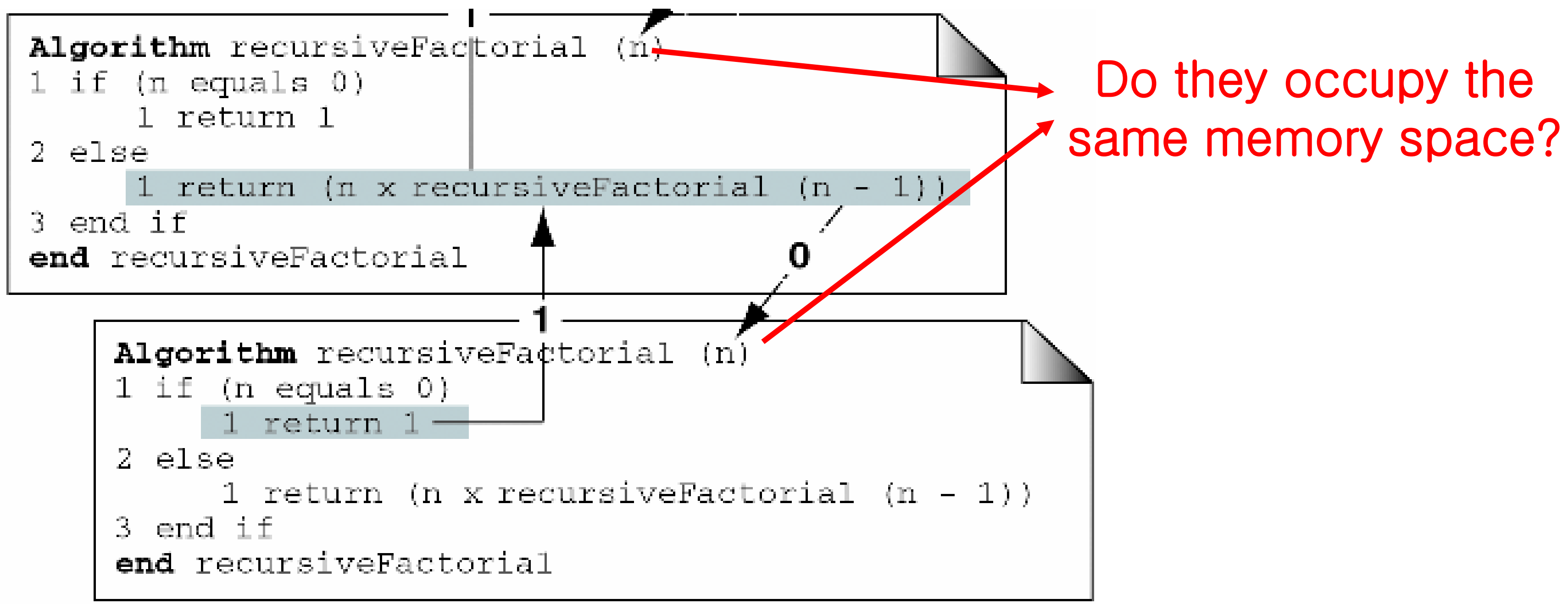
Trace of Recursion



Local Variables in Recursion

- When recursiveFactorial(3) calls recursiveFactorial(2), are local variables of the former occupy the same memory with the local variables of the latter?

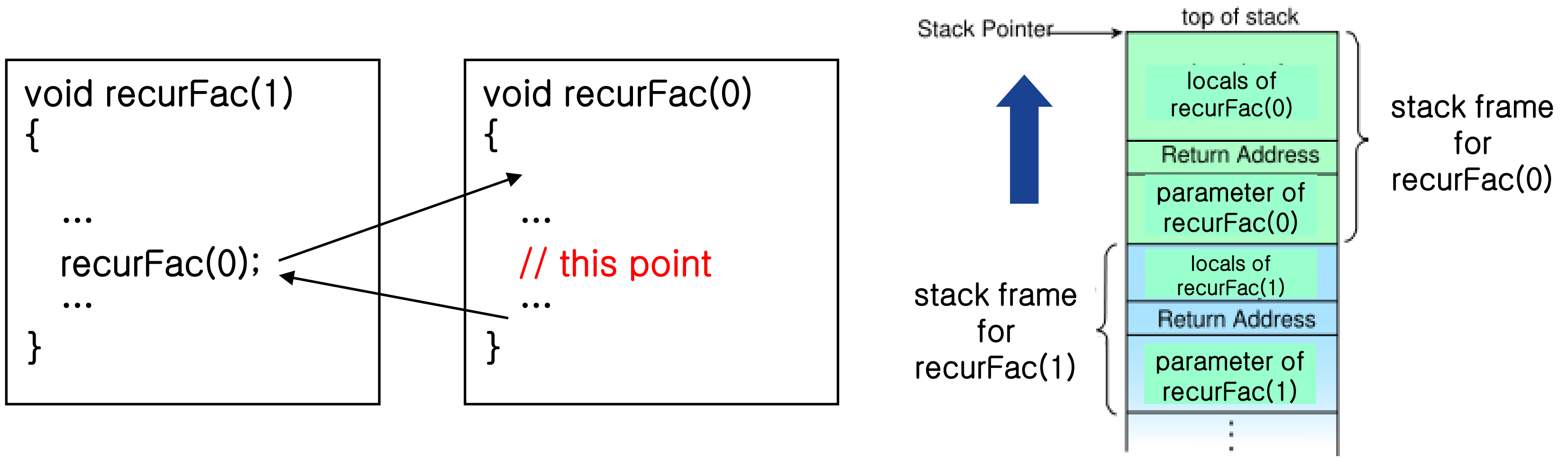
  Note! parameter ⊂ local variable



Stack Frame of Recursion

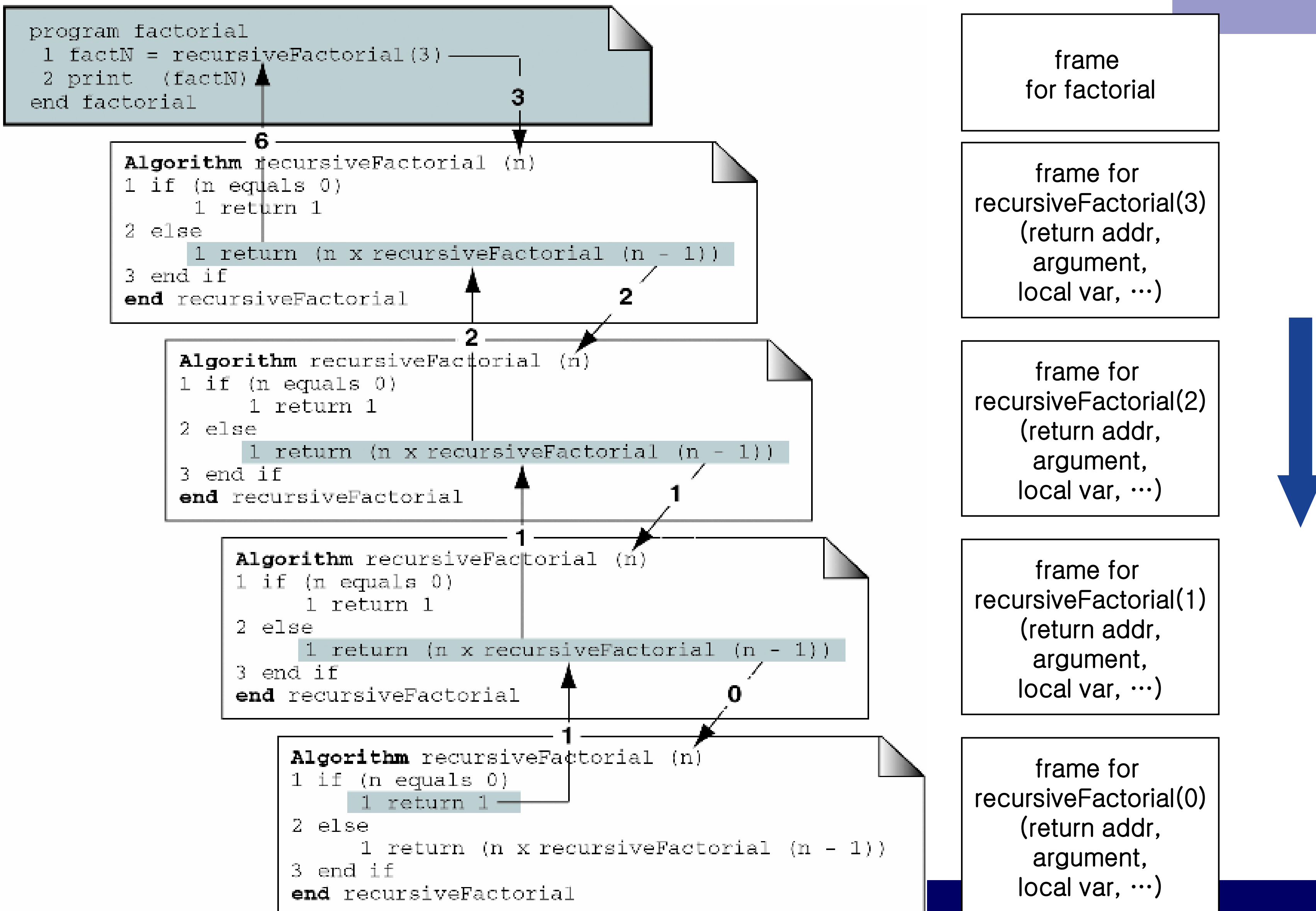
- Function call is implemented using a special type of memory, called stack
  - Stack stores parameters, return address, local variables, …

  Ex) recursiveFactorial(1) calls recursiveFactorial(0)



- An area of stack for a function call is called stack frame or activation record

Stack Frame of Recursion



Properties of Recursion

- Recursion is effective for
  - Problems that are naturally recursive
    - Binary search (chap. 13)
  - Algorithms that use a data structure naturally recursive
    - Tree (chap, 6)

- Problems of recursion
  - Function call overhead
    - Time
    - Stack memory

Limitations of Recursion

You should not use recursion if the answer to any of the following question is no:

- Is the algorithm or data structure naturally suited to recursion?
- Is the recursive solution shorter and more understandable?
- Does the recursive solution run within acceptable time and space limits?

Agenda

- Introduction
- <u>Recursion Examples</u>
  - Greatest common divisor
  - Fibonacci numbers
  - Reverse keyboard input
- Designing Recursive Algorithms
- Recursion Design Examples
  - Prefix to postfix conversion
  - Towers of Hanoi
- Recursion and Divide-and-Conquer

Greatest Common Divisor

- GCD design (Euclidean algorithm)

$$
\gcd(a, b) = \begin{cases} a & \text{if } b = 0 \\ b & \text{if } a = 0 \\ \gcd(b, a \% b) & \text{otherwise} \end{cases}
$$

- 78696과 19332의 최대공약수를 구하면,

$$
\begin{aligned}
78696 &= 19332 \times 4 + 1368 \\
19332 &= 1368 \times 14 + 180 \\
1368 &= 180 \times 7 + 108 \\
180 &= 108 \times 1 + 72 \\
108 &= 72 \times 1 + 36 \\
72 &= 36 \times 2
\end{aligned}
$$

ALGORITHM 2-4 Euclidean Algorithm for Greatest Common Divisor

```
Algorithm gcd (a, b)
Calculates greatest common divisor using the Euclidean algo-
rithm.
   Pre  a and b are positive integers greater than 0
   Post greatest common divisor returned
1  if (b equals 0)
   1  return a
2  end if
3  if (a equals 0)
   2  return b
4  end if
5  return gcd (b, a mod b)
end gcd
```

PROGRAM 2-1 GCD Driver

```
/* This program determines the greatest common divisor
   of two numbers.
      Written by:
      Date:
*/
#include <stdio.h>
```

continued

PROGRAM 2-1 GCD Driver *(continued)*

```c
#include <ctype.h>

// Prototype Statements
int gcd (int a, int b);

int main (void)
{
// Local Declarations
   int  gcdResult;

// Statements
   printf("Test GCD Algorithm\n");

   gcdResult = gcd (10, 25);
   printf("GCD of 10 & 25 is %d", gcdResult);
   printf("\nEnd of Test\n");
   return 0;
} // main
```

PROGRAM 2-1 GCD Driver (continued)

```
/* ================== gcd ==================
   Calculates greatest common divisor using the
   Euclidean algorithm.
      Pre  a and b are positive integers greater than 0
      Post greatest common divisor returned
*/
int gcd (int a, int b)
{
   // Statements
   if (b == 0)
      return a;
   if (a == 0)
      return b;
   return gcd (b, a % b);
}  // gcd
```

```
Results:
Test GCD Algorithm
GCD of 10 & 25 is 5
End of Test
```

Fibonacci Numbers

- Fibonacci numbers: each number is the sum of previous two numbers

  0, 1, 1, 2, 3, 5, 8, 13, 21, 34, …

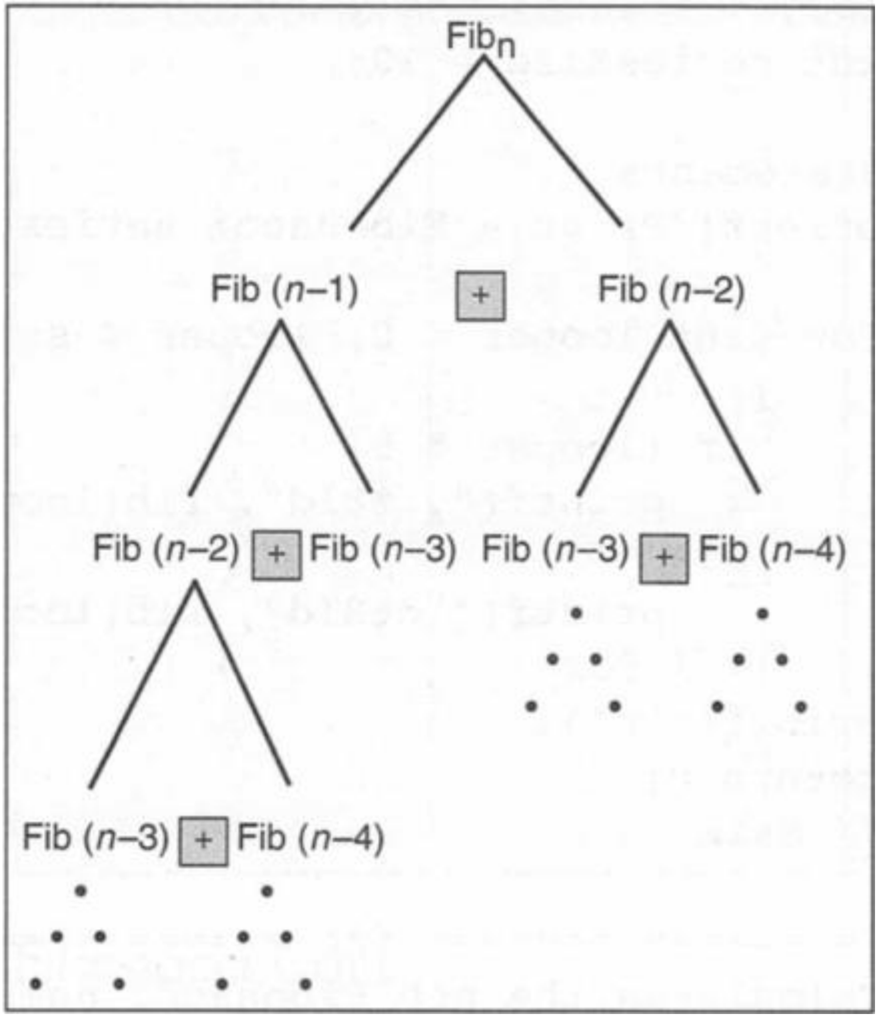
- Recursive design
  - Fibonacci(n) = 0 if n = 0
    = 1 if n = 1
    = Fibonacci(n−1) + Fibonacci(n−2)

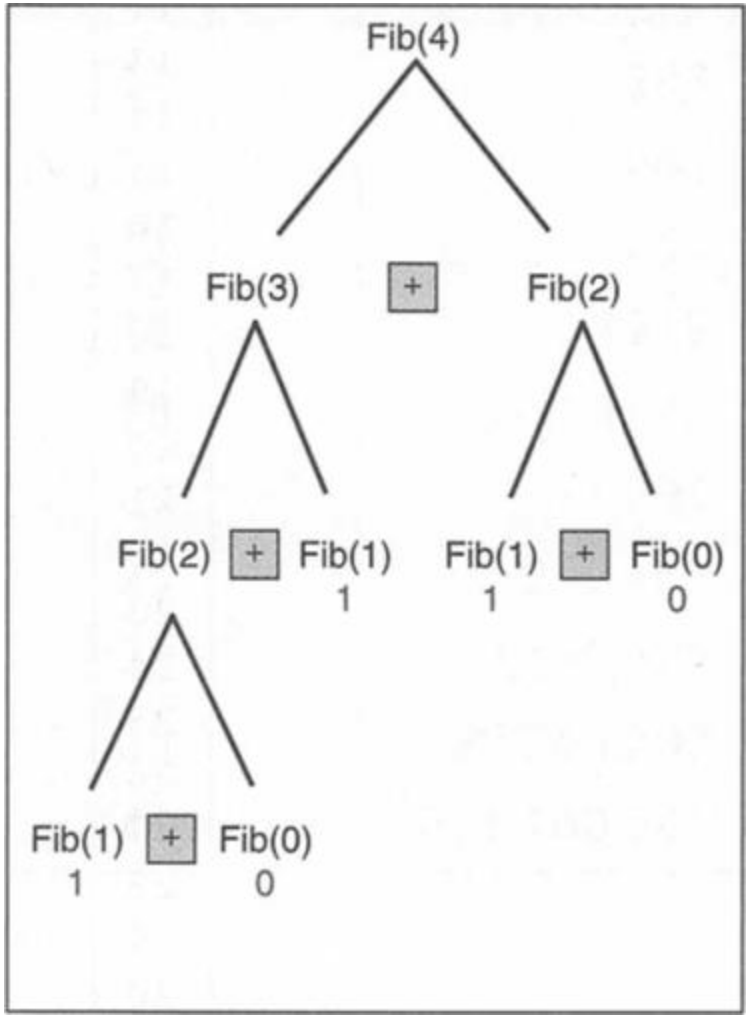
Fibonacci Numbers

- Recursive algorithm

```
long fib (long num)
{
   // Base Case
   if (num == 0 || num == 1)
      return num;

   // General Case
   return (fib (num - 1) + fib (num - 2));
}  // fib
```

Fibonacci Numbers





Fibonacci Numbers

- # of function calls to calculate Fibonacci numbers

fib(*n*)	Calls	fib(n)	Calls
1	1	11	287
2	3	12	465
3	5	13	753
4	9	14	1219
5	15	15	1973
6	25	20	21,891
7	41	25	242,785
8	67	30	2,692,573
9	109	35	29,860,703
10	177	40	331,160,281

PROGRAM 2-2 Recursive Fibonacci Series

```c
/* This program prints out a Fibonacci series.
      Written by:
      Date:
*/
#include <stdio.h>

// Prototype Statements
   long fib (long num);

int main (void)
{
// Local Declarations
   int seriesSize = 10;

// Statements
   printf("Print a Fibonacci series.\n");

```

PROGRAM 2-2 Recursive Fibonacci Series **(Continued)**

```
    for (int looper = 0; looper < seriesSize; looper++)
        {
         if (looper % 5)
            printf(", %8ld", fib(looper));
         else
            printf("\n%8ld", fib(looper));
        } // for
    printf("\n");
    return 0;
}   // main

/* ================= fib ==================
   Calculates the nth Fibonacci number
      Pre   num identifies Fibonacci number
      Post returns nth Fibonacci number
*/
long fib (long num)
{
// Statements
    if (num == 0 || num == 1)
```

continued

PROGRAM 2-2 Recursive Fibonacci Series *(continued)*

```
        // Base Case
        return num;
    return (fib (num - 1) + fib (num - 2));
} // fib
```

```
Results:
Print a Fibonacci series.

      0,       1,       1,       2,       3
      5,       8,      13,      21,      34
```

Reverse Keyboard Input

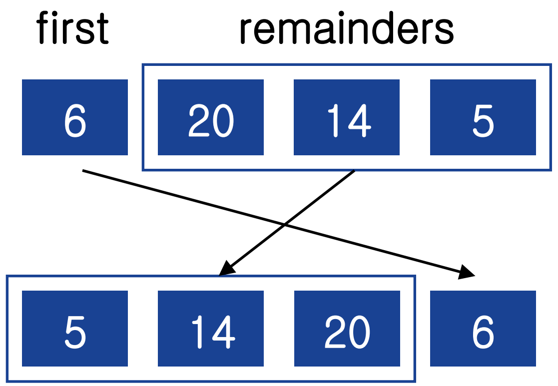
- Goal
  - Input: sequence of input (terminates by EOF)

    Ex) 6 20 14 5
  - Output: print in reverse

    Ex) 5 14 20 6

- Algorithm design
  - General case

    1. Read a data
    2. Reverse remainders (sub-problem)
    3. Print a data
  - Base case: last input? EOF
    - If input is EOF, do nothing



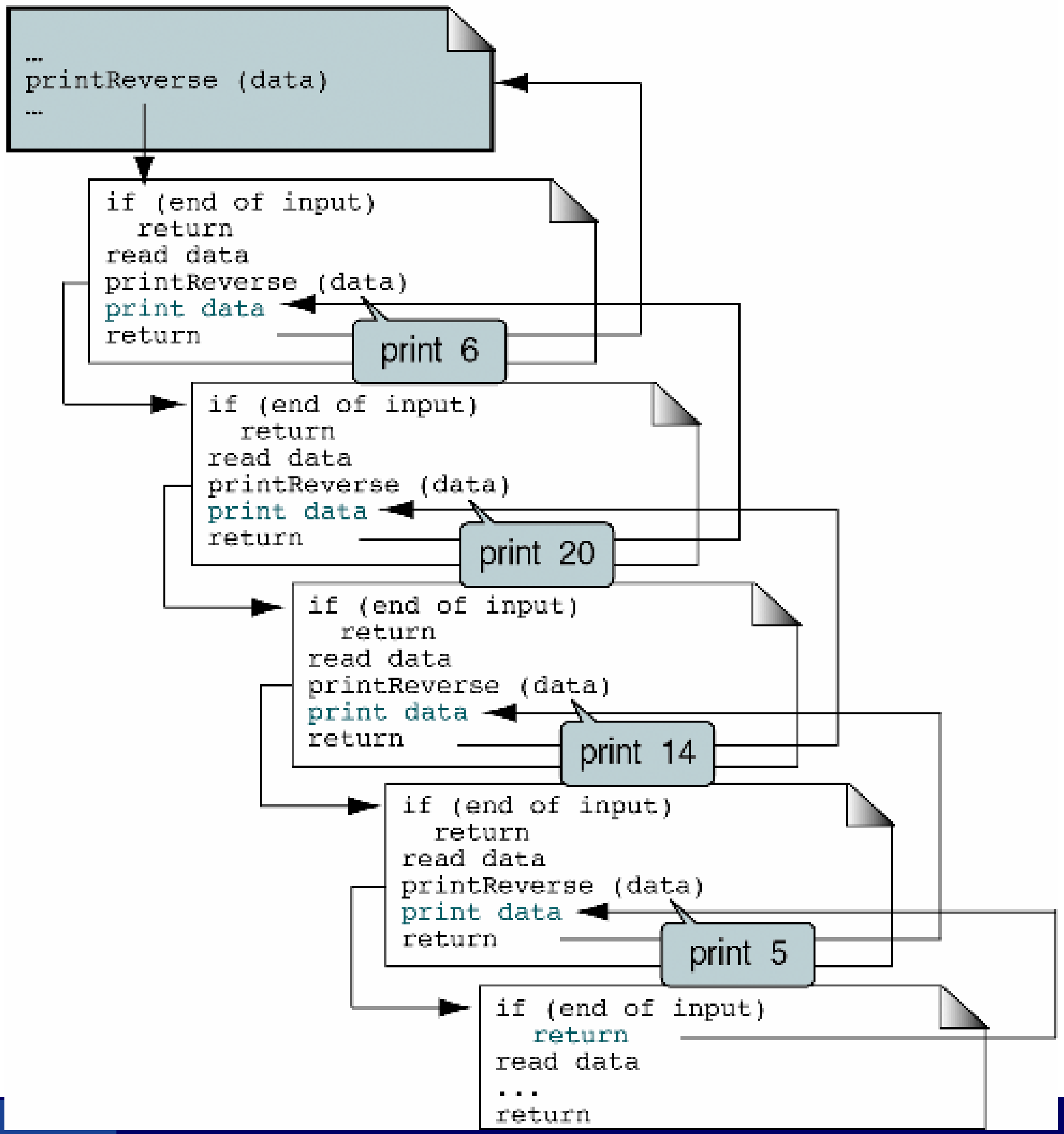
Reverse Keyboard Input

- Algorithm

```
Algorithm printReverse (data)
Print keyboard data in reverse.
   Pre  nothing
   Post data printed in reverse
 1 if (end of input)
   1  return
 2 end if
 3 read data
 4 printReverse (data)
Have reached end of input: print nodes
 5 print data
 6 return
end printReverse
```

```
void printReverse()
{
   int v = 0, ret = 0;

   ret = scanf(" %d", &v);
   if(ret != 1){
      // base case
      return;
   } else {
      // general case
      printReverse();
      printf("%d ", v);
   }
}
```

Reverse Keyboard Input



Analysis of Reverse Keyboard Input

- Is the algorithm or data structure naturally suited to recursion?
  - List is not a naturally recursive structure
  - Its is not logarithmic algorithm
- Is the recursive solution shorter and more understandable?
- Does the recursive solution run within acceptable time and space limits?
  - O(n)

Agenda

- Introduction
- Recursion Examples
  - Greatest common divisor
  - Fibonacci numbers
  - Reverse keyboard input
- Designing Recursive Algorithms
- Recursion Design Examples
  - Prefix to postfix conversion
  - Towers of Hanoi
- Recursion and Divide-and-Conquer

Designing Recursive Algorithms

- Every recursive algorithm has two elements
  - Solve a primitive problem ➔ non-recursive solution
    - Base case
  - Reduce the size of problem ➔ recursion
    - General case

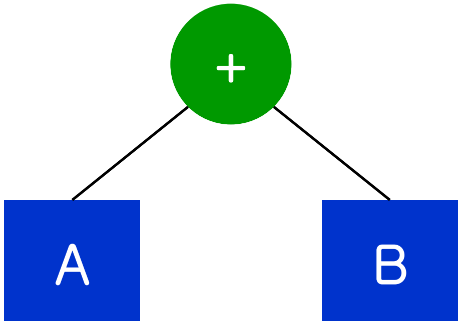
```
Algorithm recursiveFactorial (n)
Calculates factorial of a number using recursion.
   Pre    n  is the number being raised factorially
   Post   n! is returned
1  if (n equals 0)
   1  return 1
2  else
   1  return (n * recursiveFactorial (n - 1))
3  end if
end recursiveFactorial
```

Design Methodology

- Rules for designing a recursive algorithm
  1. Determine base case
     - Non-recursive solution of primitive case such as n = 0, 1
  2. Determine general case
     - Break down the problem into sub-problems which are the same, but smaller than original
     - Assume sub-problems are already solved
       Ex) To calculate factorial(n), assume factorial(n-1), factorial(n-2), …, factorial(0) are solved
  3. Combine base and general cases

Prefix to Postfix Conversion

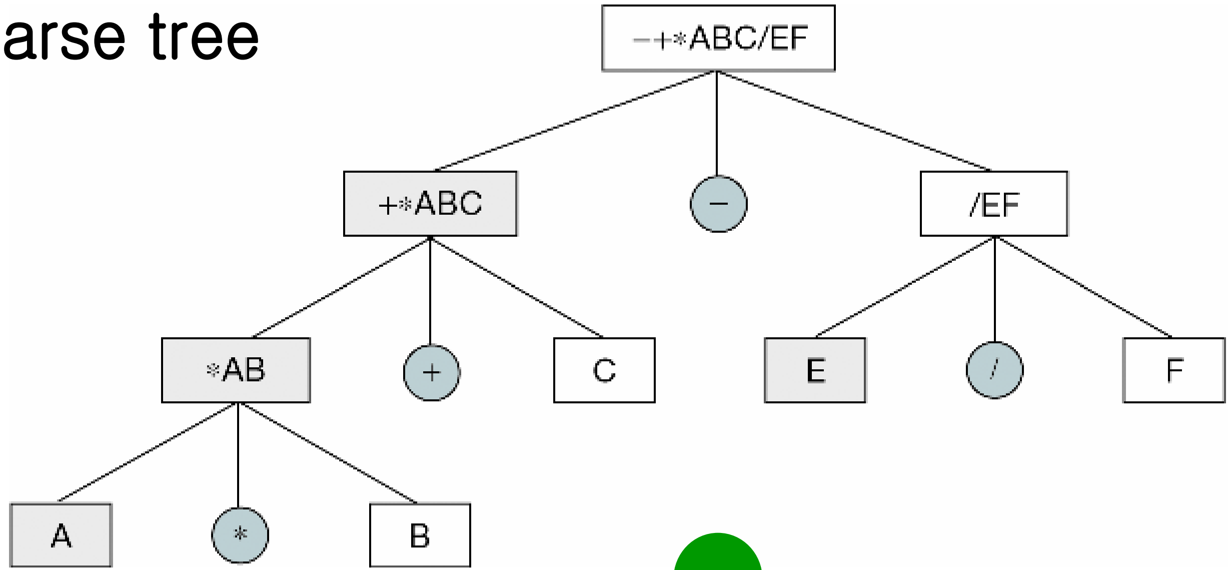
- Arithmetic expressions
  - Infix: operator comes **between** operands
    Ex) A+B
  - Prefix: operator comes **before** operands
    Ex) +AB
  - Postfix: operator comes **after** operands
    Ex) AB+

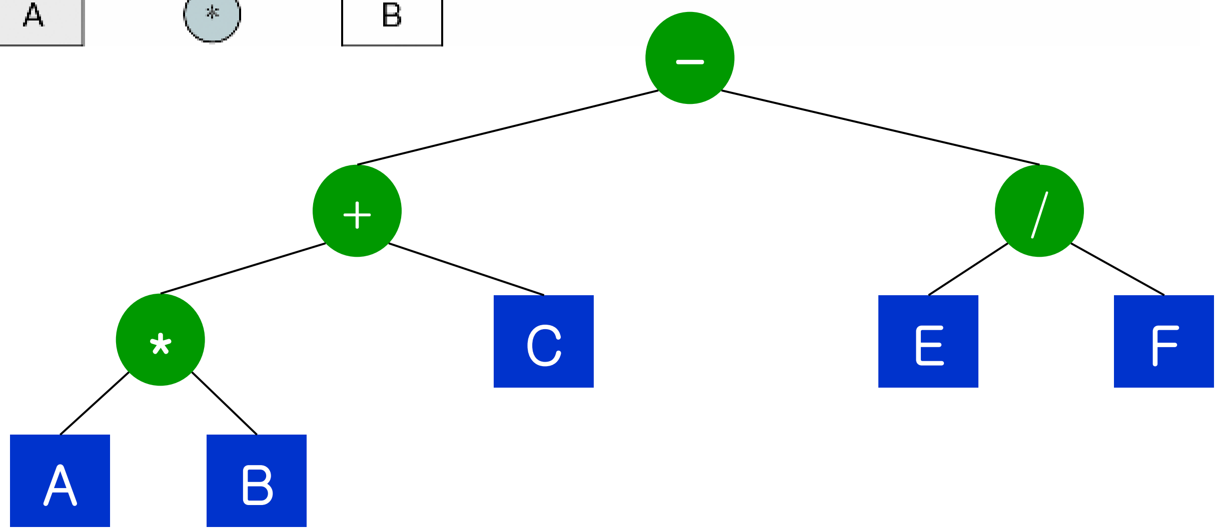


- Prefix to postfix conversion
  - +AB ➔ AB+
  - −+*ABC/EF ➔ ?

Tree Representation of −+∗ABC/EF

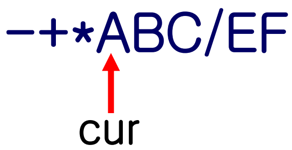
- Parse tree

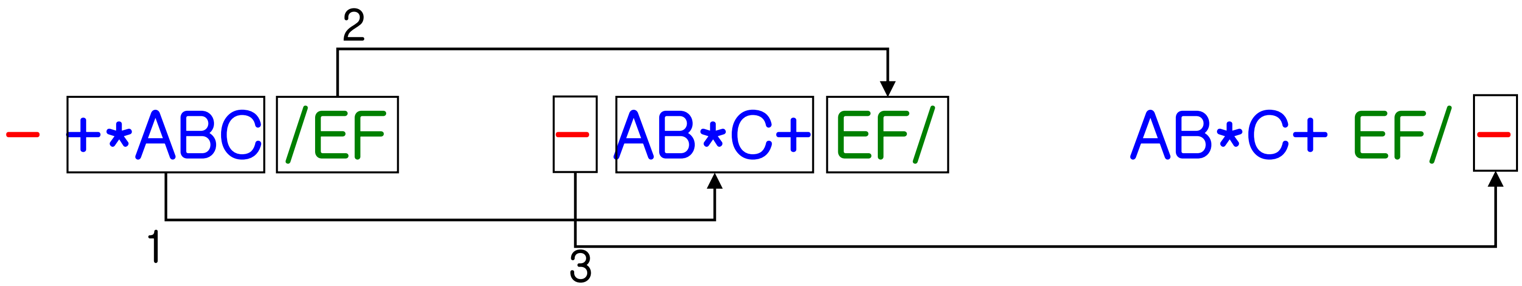




Prefix to Postfix Conversion

- Algorithm design
  - Base case: terminal node
    - Current element is not an operator
    - ➔ Just return current token
  - General case: non-terminal node (operator)
    - Current element is an operator
    - ➔ 1. Convert left operand
      2. Convert right operand
      3. Concatenate converted operands and operator





ALGORITHM 2-5 Convert Prefix Expression to Postfix

```
Algorithm preToPostFix (preFixIn, postFix)
Convert a preFix string to a postFix string.
   Pre  preFix is a valid preFixIn expression
        postFix is reference for converted expression
   Post postFix contains converted expression
 1 if (length of preFixIn is 1)
      Base case: one character string is an operand
   1  set postFix to preFixIn
   2  return
 2 end if
   If not an operand, must be an operator
 3 set operator to first character of preFixIn
   Find first expression
 4 set lengthOfExpr to findExprLen (preFixIn less first char)
 5 set temp to substring(preFixIn[2, lengthOfExpr])
 6 preToPostFix (temp, postFix1)
   Find second postFix expression
 7 set temp to prefixIn[lengthOfExpr + 1, end of string]
 8 preToPostFix (temp, postFix2)
   Concatenate postfix expressions and operator
 9 set postFix to postFix1 + postFix2 + operator
10 return
end preToPostFix
```

ALGORITHM 2-6 Find Length of Prefix Expression

```
Algorithm findExprLen (exprIn)
Recursively determine the length of a prefix expression.
   Pre  exprIn is a valid prefix expression
   Post length of expression returned
1 if (first character is operator)
  General Case: First character is operator
  Find length of first prefix expression
  1  set len1 to findExprLen (exprIn + 1)
  2  set len2 to findExprLen (exprIn + 1 + len2)
2 else
  Base case--first char is operand
  1  set len1 and len2 to 0
3 end if
4 return len1 + len2 + 1
end findExprLen
```

PROGRAM 2-3 Prefix to Postfix

```
/* Convert prefix to postfix expression.
      Written by:
      Date:
*/
#include <stdio.h>
#include <string.h>

#define OPERATORS "+-*/"

// Prototype Declarations
void preToPostFix  (char* preFixIn, char* exprOut);
int  findExprLen   (char* exprIn);

int main (void)
{
```

continued

PROGRAM 2-3 Prefix to Postfix *(continued)*

```
// Local Definitions
   char  preFixExpr[256]  = "-+*ABC/EF";
   char  postFixExpr[256] = "";

// Statements
   printf("Begin prefix to postfix conversion\n\n");

   preToPostFix (preFixExpr, postFixExpr);
   printf("Prefix expr:  %-s\n", preFixExpr);
   printf("Postfix expr: %-s\n", postFixExpr);

   printf("\nEnd prefix to postfix conversion\n");
   return 0;
}  // main
```

PROGRAM 2-3 Prefix to Postfix *(continued)*

```
/* ==================== preToPostFix =====================
   Convert prefix expression to postfix format.
      Pre  preFixIn is string prefix expression
           expression can contain no errors/spaces
           postFix is string variable for postfix
      Post expression has been converted
*/
void preToPostFix (char* preFixIn, char* postFix)
{
// Local Definitions
   char  operator [2];
   char  postFix1[256];
   char  postFix2[256];
   char  temp    [256];
   int   lenPreFix;

```

```
// Statements
    if (strlen(preFixIn) == 1)
       {
        *postFix         = *preFixIn;
        *(postFix + 1) = '\0';
        return;
       } // if only operand

    *operator         = *preFixIn;
    *(operator + 1) = '\0';

    // Find first expression
    lenPreFix = findExprLen (preFixIn + 1);
    strncpy (temp, preFixIn + 1, lenPreFix);
    *(temp + lenPreFix) = '\0';
    preToPostFix (temp, postFix1);
```

continued

PROGRAM 2-3 Prefix to Postfix *(continued)*

```

   // Find second expression
   strcpy (temp, preFixIn + 1 + lenPreFix);
   preToPostFix (temp, postFix2);

   // Concatenate to postFix
   strcpy (postFix, postFix1);
   strcat (postFix, postFix2);
   strcat (postFix, operator);

   return;
}  // preToPostFix
```

PROGRAM 2-3 Prefix to Postfix *(continued)*

```
/* ==================== findExprLen ====================
   Determine size of first substring in an expression.
      Pre  exprIn contains prefix expression
      Post size of expression is returned
*/
int findExprLen (char* exprIn)
{
// Local Definitions
   int  len1;
   int  len2;

```

PROGRAM 2-3 Prefix to Postfix (continued)

```
// Statements
    if (strcspn (exprIn, OPERATORS) == 0)
          // General Case: First character is operator
          // Find length of first expression
        {
          len1 = findExprLen(exprIn + 1);

          // Find length of second expression
          len2 = findExprLen(exprIn + 1 + len1);
        } // if
    else
          // Base case--first char is operand
          len1 = len2 = 0;
    return len1 + len2 + 1;
}   // findExprLen
```

```
Results:
Begin prefix to postfix conversion

Prefix expr:  -+*ABC/EF
Postfix expr: AB*C+EF/-

End prefix to postfix conversion
```

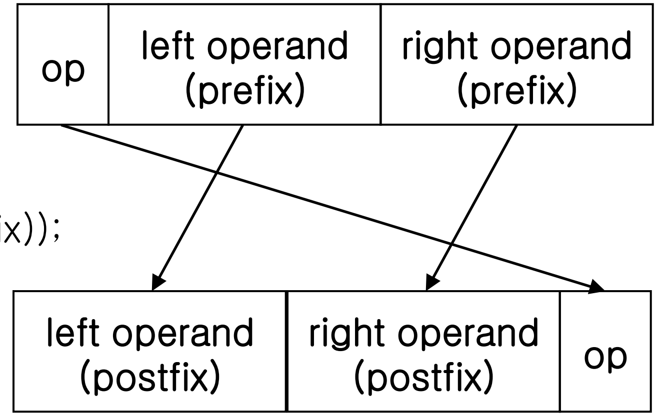
참고

- unsigned int strcspn(const char* string, const char* strCharSet)
  - string은 검색될 문자열, strCharSet은 검색할 문자들의 집합(문자열?)
  - 문자집합 중 하나의 문자라도 일치하면 위치를 반환하고, 없으면 문자열의 길이를 반환한다
- char * strncpy (char * destination, const char * source, size_t n);
  - Destination은 문자열을 복사할 버퍼, source는 복사할 원본 문자열, n은 복사할 문자 개수
  - 부분 문자열을 복사하는 함수
  - 반환 값: destionation

Prefix to Postfix Conversion

- Algorithm in C code

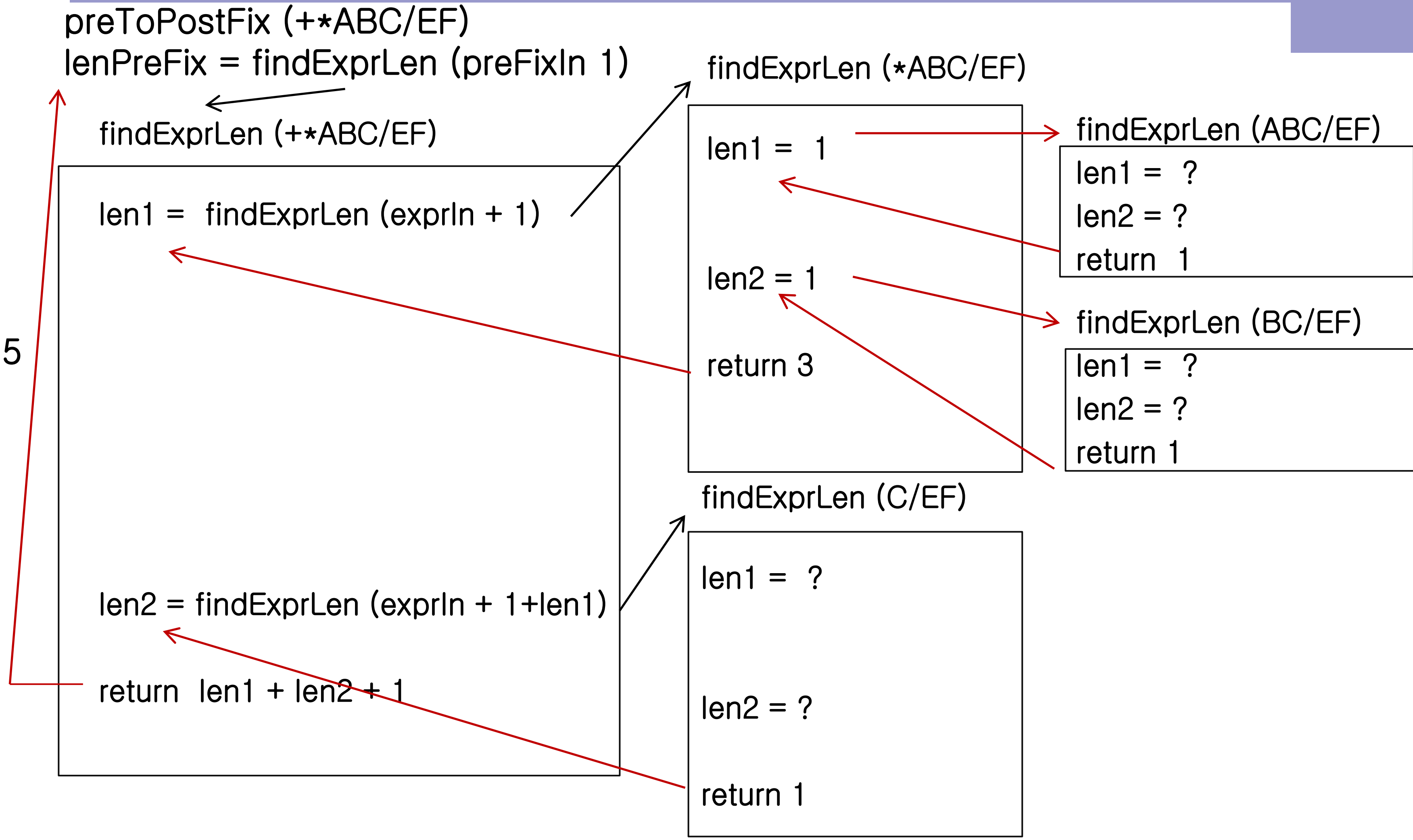
```
int preToPostFix (char* preFixIn, char* postFix)
// return value: length of converted expression
{
    char  op[2];              // operator
    int   p = 0;

    // base case
    if (strchr("+-*/", preFixIn[0]) == NULL){
        postFix[0] = preFixIn[0];
        postFix[1] = '₩0';
        return strlen(postFix);
    }

    // general case
    op[0] = preFixIn[p++];
    op[1] = '₩0';
    p += preToPostFix(preFixIn + p, postFix);
    p += preToPostFix(preFixIn + p, postFix + strlen(postFix));
    strcat(postFix, op);

    // return length of current sub-expression
    return strlen(postFix);
}   // preToPostFix
```



참고

- strchr(대상문자열, 검색할문자);
- char *strchr(char * const _String, int const _Ch)
  - 문자를 찾으면 문자로 시작하는 문자열의 포인터를 반환하고, 문자가 없으면 NULL을 반환
  - char* ptr = strchr(s1, ‘a’); 와 같이 문자열(s1)과 검색할 문자(‘a’)를 넣어주면 해당 문자로 시작하는 문자열의 위치(포인터)를 반환

Recursive Call Sequence

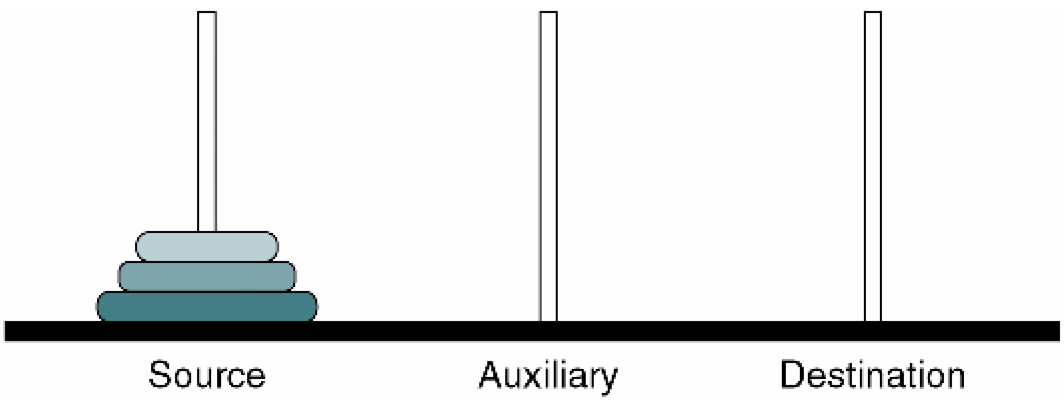


Prefix to Postfix Conversion

- Exercises
  - Draw call graph of *preToPostFix* by yourself

Towers of Hanoi

- Input
  - Three towers (source, destination, auxiliary)
  - *n* disks of different diameters placed on source tower in decreasing diameter
- Problem
  - Move all disks from source tower to destination tower
- Rules
  - Only one disk can be moved at any time
  - No disk can be placed on top of a disk with a smaller diameter



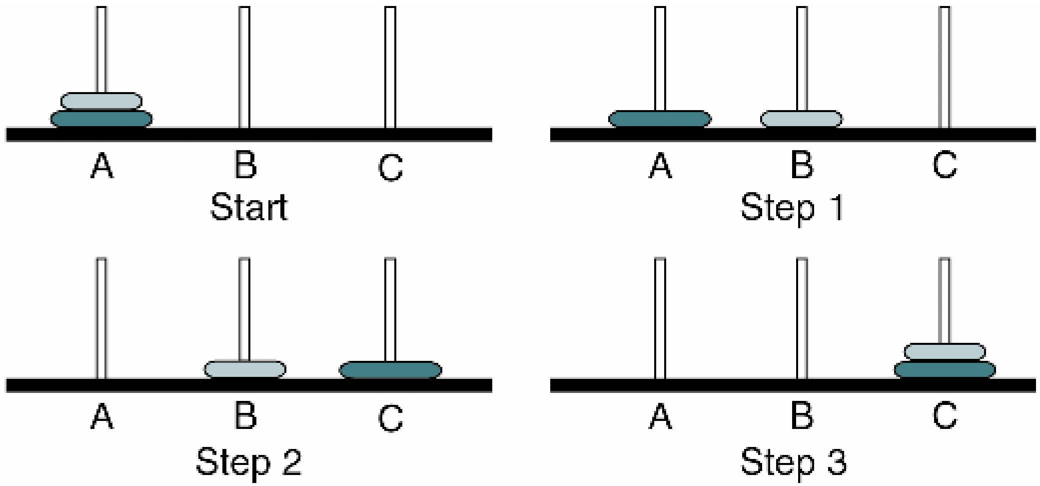
Towers of Hanoi

- Algorithm design
  - Base case: only one disk to move
    - Just move it
  - General case: # of disk > 1

    ➔ How to break down the problem ?

Towers of Hanoi

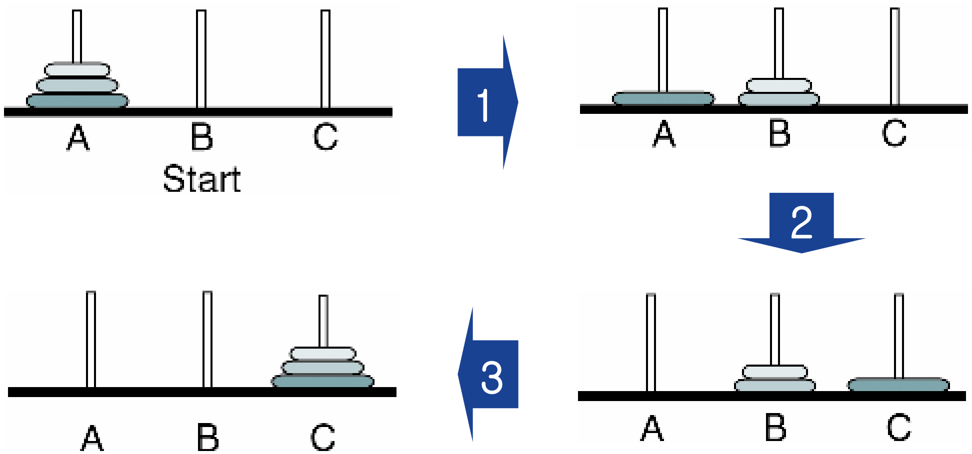
- Moving 2 disks
  1. Move one disk from source to auxiliary
  2. Move one disk from source to destination
  3. Move one disk from auxiliary to destination



Towers of Hanoi

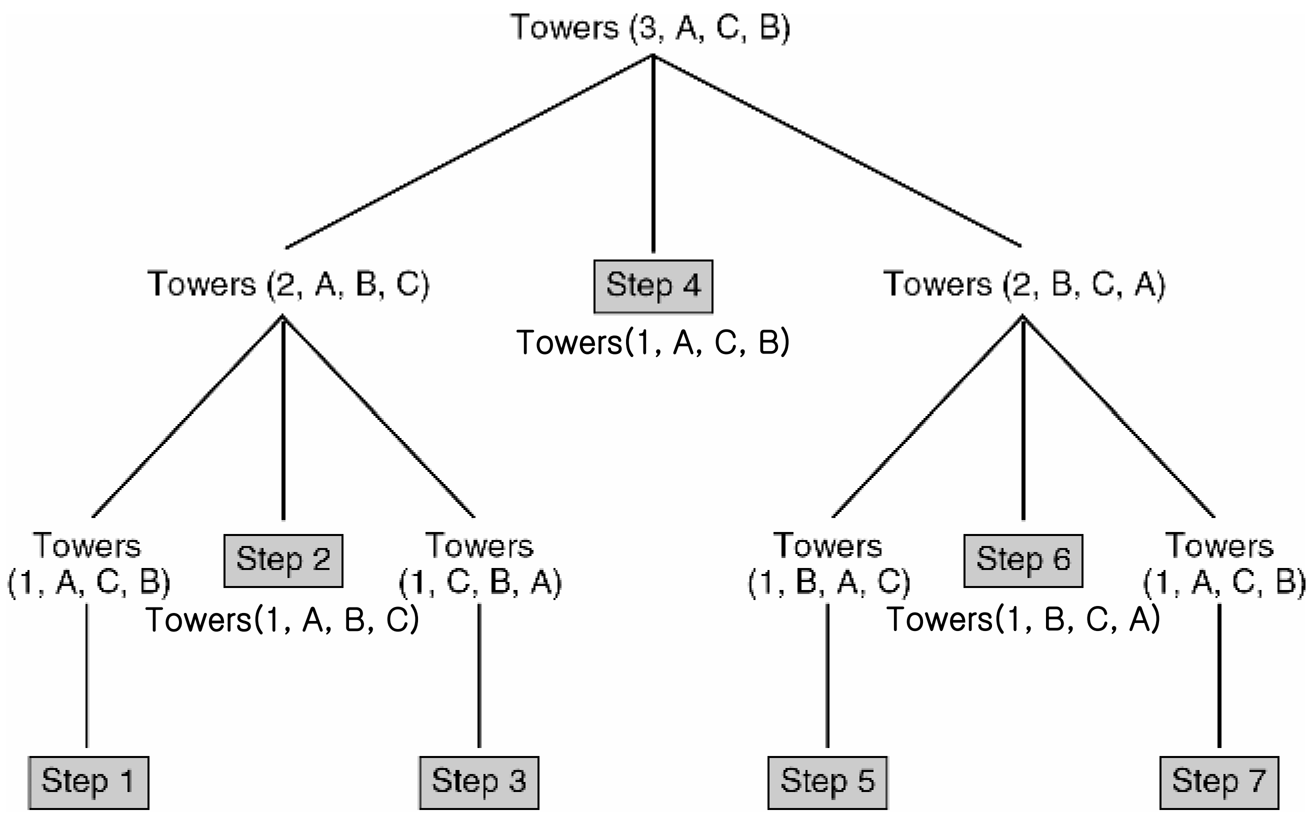
- Moving n disks
  1. move n –1 disks from source to auxiliary
  2. move 1 disk from source to destination ➔ base case
  3. move n –1 disks from auxiliary to destination

Ex) n = 3



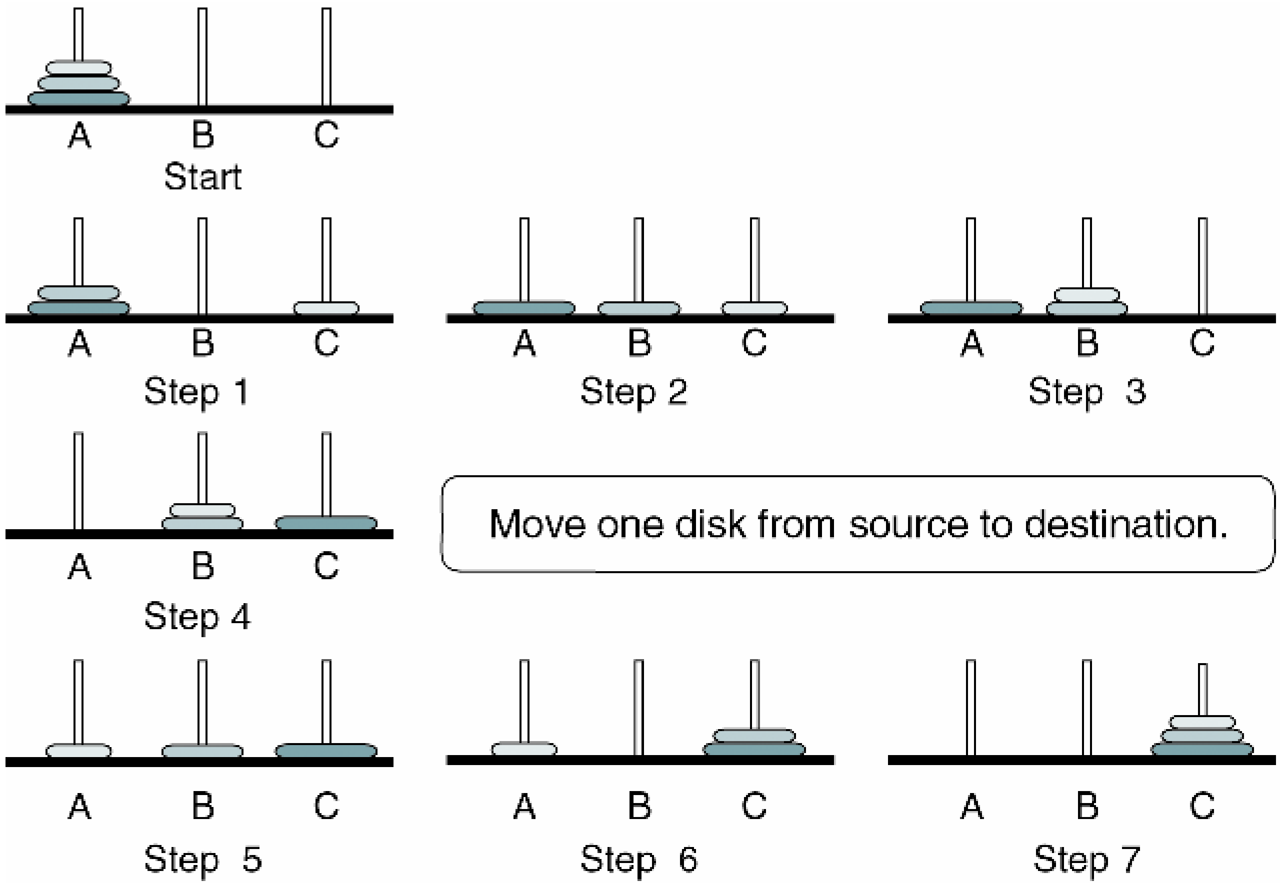
Towers of Hanoi

- Moving 3 disks



Towers of Hanoi

- Moving 3 disks



Towers of Hanoi

- Algorithm

```
Algorithm towers (numDisks, source, dest, auxiliary)
Recursively move disks from source to destination.
   Pre  numDisks is number of disks to be moved
        source, destination, and auxiliary towers given
   Post steps for moves printed
 1 print("Towers: ", numDisks, source, dest, auxiliary)
 2 if (numDisks is 1)
   1  print ("Move from ", source, " to ", dest)
 3 else
   1  towers (numDisks - 1, source, auxiliary, dest, step)
   2  print ("Move from " source " to " dest)
   3  towers (numDisks - 1, auxiliary, dest, source, step)
 4 end if
end towers
```

Towers of Hanoi

- Algorithm in C code

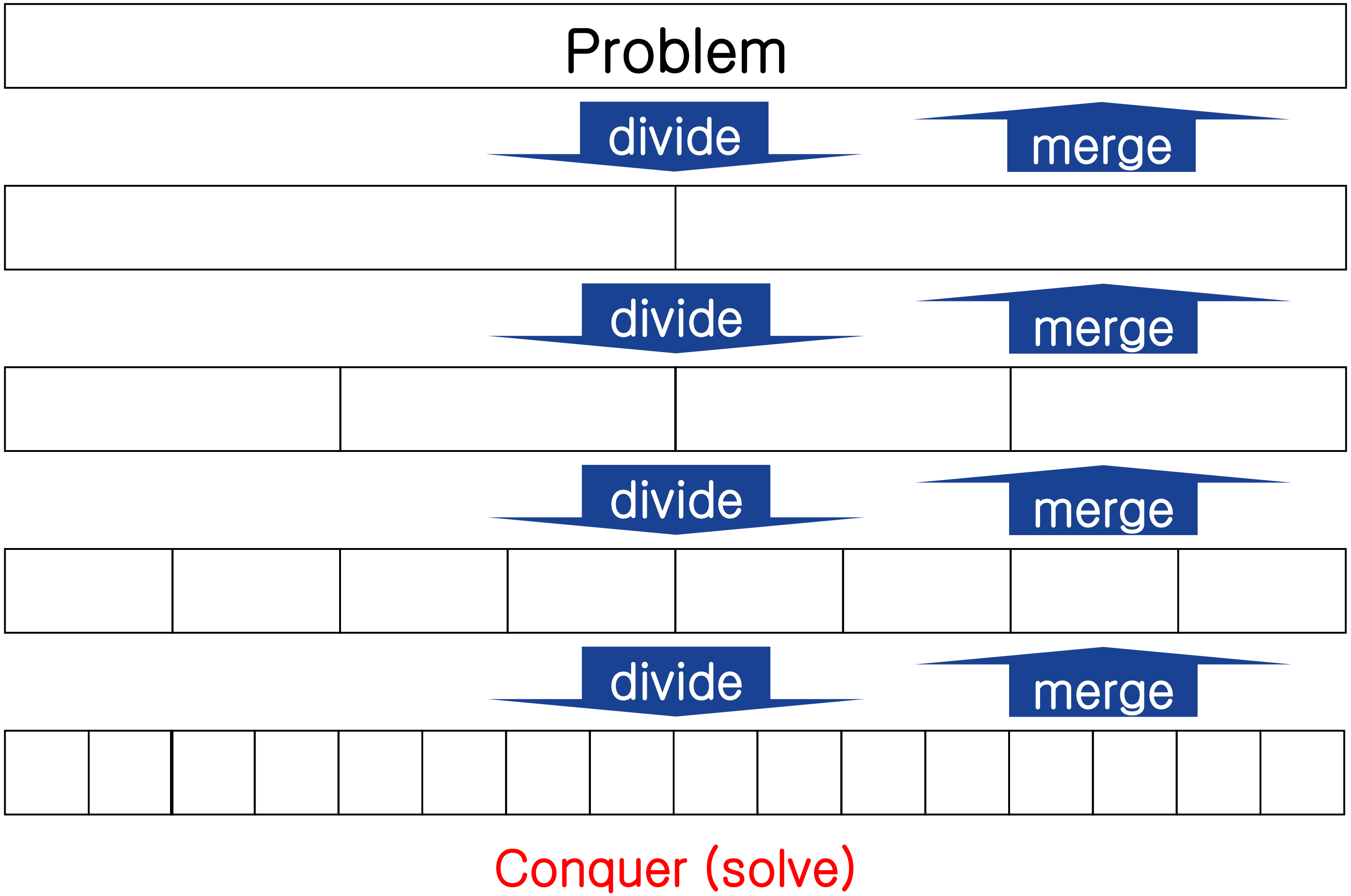
```
void towers (int n, char source, char dest, char auxiliary)
{
   if (n == 1)          // base case
      printf("Move from %c to %c₩n", source, dest);
   else {               // general case
      towers(n – 1, source, auxiliary, dest);
      towers(1, source, dest, auxilary);
      towers(n – 1, auxiliary, dest, source);
    } // if else
}  // towers
```

Agenda

- Introduction
- Recursion Examples
  - Greatest common divisor
  - Fibonacci numbers
  - Reverse keyboard input
- Designing Recursive Algorithms
- Recursion Design Examples
  - Prefix to postfix conversion
  - Towers of Hanoi
- <u>Recursion and Divide-and-Conquer</u>

Recursion and Divide-and-Conquer

- Divide-and-Conquer: a strategy to solve a problem



Divide and Conquer

- Procedure

  1. Divide problem into sub-problems with smaller size

  2. If problem is small enough to solve directly, solve it without recursion

  3. Merge sub-problems

Frame of Recursion

- Divide and Conquer
  1. Divide problem
  2. Solve elementary problems through a trivial method
  3. Merge solutions of sub-problems

- Typical recursive function

```
RecurFunc(problem)
{
  if(termination_condition){
    // base case
    return;
  }

  // general case
  sub_problems = Divide(problem);
  RecurFunc(sub_problem1);
  RecurFunc(sub_problem2);
  …
  RecurFunc(sub_problemN);

  solution = Merge(sub_solutions);
}
```

Example: Tower of Hanoi

- Typical recursive function

```
RecurFunc(problem)
{
   if(termination_condition){
     // base case
     return;
   }

   // general case
   sub_problems = Divide(problem);
   RecurFunc(sub_problem1);
   RecurFunc(sub_problem2);
   ...
   RecurFunc(sub_problemN);

   solution = Merge(sub_solutions);
}
```

- Tower of Hanoi

```
void towers(int n, char source, char dest,
     char auxiliary)
{
     if (n == 1)          // base case
        printf("Move from %c to %c₩n",
                           source, dest);
     else {               // general case
        towers(n – 1, source, auxiliary, dest);
        towers(1, source, dest, auxilary);
        towers(n – 1, auxiliary, dest, source);
    } // if else
}    // towers
```