Chapter 2

Recursion

Objectives

Upon completion you will be able to:

- **Explain the difference between iteration and recursion**

- **Design a recursive algorithm**

- **Determine when an recursion is an appropriate solution**

- **Write simple recursive functions**

2-1   Factorial - A Case Study

We begin the discussion of recursion with a case study and use it to define the concept.
This section also presents an iterative and a recursive solution to the factorial algorithm.

- **Recursive Defined**
- **Recursive Solution**

Iterative Factorial Algorithm

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

```
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
```
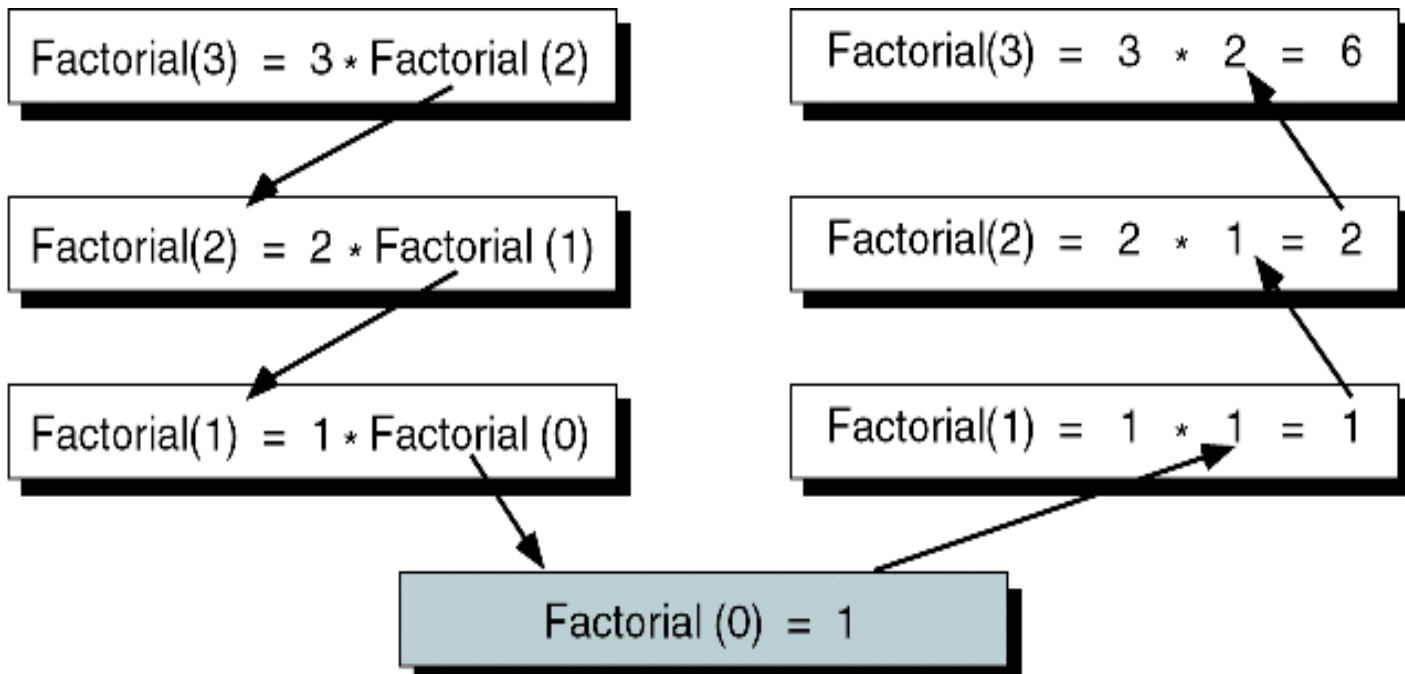
Recursive Factorial Algorithm

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

Factorial(3) = 3 * Factorial (2)

Factorial(2) = 2 * Factorial (1)

Factorial(1) = 1 * Factorial (0)

Factorial(3) = 3 * 2 = 6

Factorial(2) = 2 * 1 = 2

Factorial(1) = 1 * 1 = 1

Factorial (0) = 1

Recursive Factorial Program

```
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
```

```
program factorial
  1 factN = recursiveFactorial (3)
  2 print   (factN)
end factorial
```

3

6

```
Algorithm recursiveFactorial (n)
1 if (n equals 0)
      1 return 1
2 else
      1 return (n x recursiveFactorial (n - 1))
3 end if
end recursiveFactorial
```

2

2

```
Algorithm recursiveFactorial (n)
1 if (n equals 0)
      1 return 1
2 else
      1 return (n x recursiveFactorial (n - 1))
3 end if
end recursiveFactorial
```

1

1

```
Algorithm recursiveFactorial (n)
1 if (n equals 0)
      1 return 1
2 else
      1 return (n x recursiveFactorial (n - 1))
3 end if
end recursiveFactorial
```

0

1

```
Algorithm recursiveFactorial (n)
1 if (n equals 0)
      1 return 1
2 else
      1 return (n x recursiveFactorial (n - 1))
3 end if
end recursiveFactorial
```

2-2   Designing Recursive Algorithms

In this section we present an analytical approach to designing recursive algorithms. We also discuss algorithm designs that are not well suited to recursion.

- **The Design Methodology**
- **Limitation of Recusion**
- **Design Implemenation**

Recursive Design Methdology

- **Rules for designing a recursive algorithm**
  - **Determine the base case (e.g., factorial(0)=1)**
  - **Determine the general case (e.g., factorial (n) = n x factorial (n-1))**
  - **Combine the base case and the general cases into an algorithm**
- **Each call must reduce the size of the problem and move it toward the base case**
- **The base case, when reached, must terminate without a call to the recursive algorithm (must execute a return)**
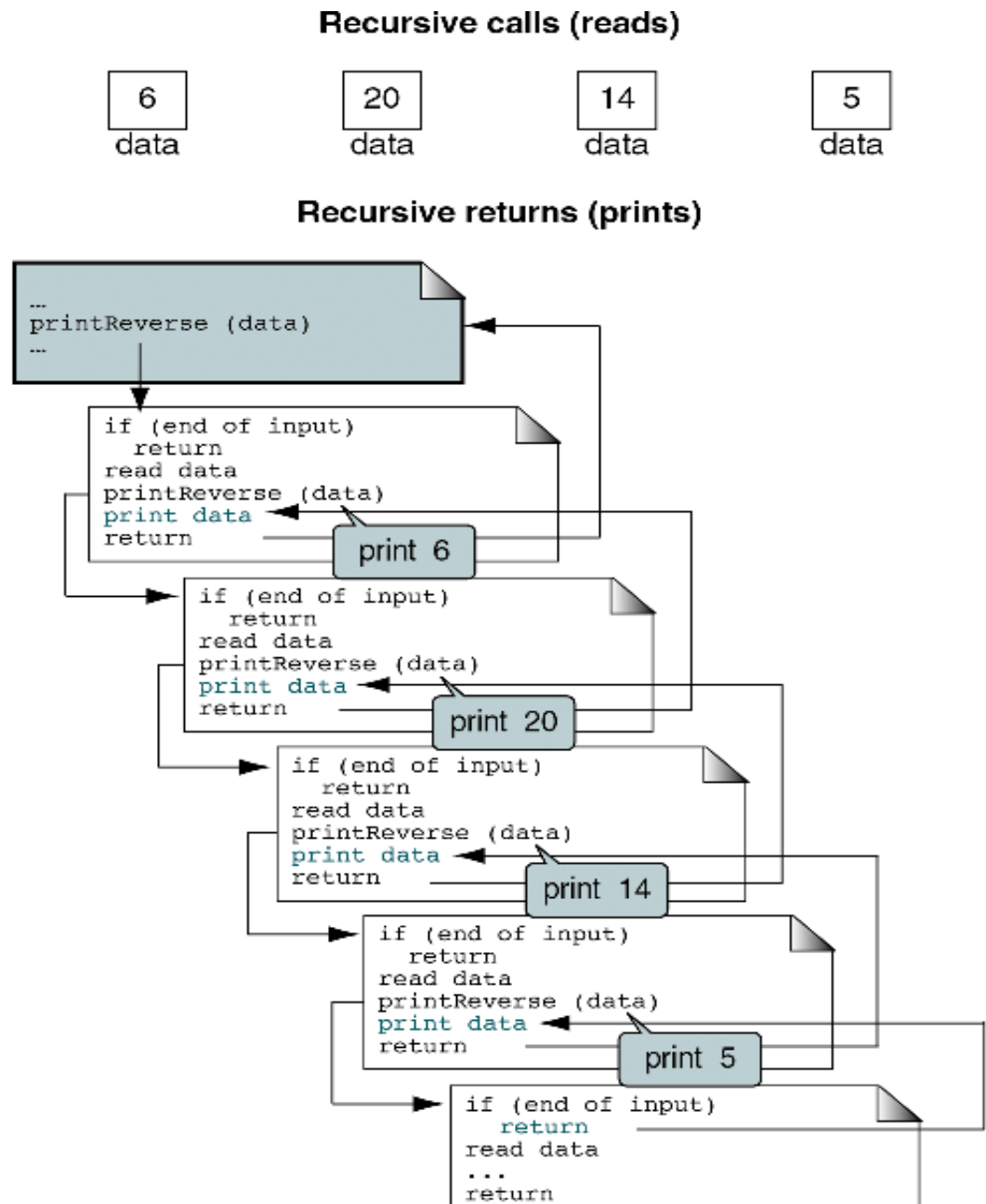
Limitations of Recursive

- **Recursive solutions involve extensive overhead (time & memory)**

  - **Each call takes time to execute !**

- **Should NOT use recursion when**

  - **The algorithm or data structure is not <u>naturally suited</u> (e.g., tree structure) to recursion** **通常是「自我相似」的結構**

  - **Recursive solution is not shorter and more understandable**

  - **Recursive solution does not run within acceptable time and space limits**

Reverse Keyboard Input

```
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
```

- **Is it naturally suited for recursion ?**
  - **No** (list structure)
- **Is it shorter and more understandable**
  - **Yes**
- **Does it run within acceptable time and space limits**
  - **No (O(n))**

Recursive calls (reads)

6	20	14	5
data	data	data	data

Recursive returns (prints)

```
...
printReverse (data)
...
```

```
if (end of input)
    return
read data
printReverse (data)
print data
return
```
print 6

```
if (end of input)
    return
read data
printReverse (data)
print data
return
```
print 20

```
if (end of input)
    return
read data
printReverse (data)
print data
return
```
print 14

```
if (end of input)
    return
read data
printReverse (data)
print data
return
```
print 5

```
if (end of input)
    return
read data
...
return
```

2-3   Recursive Examples

Four recursive programs are developed and analyzed. Only one, the Towers of Hanoi, turns out to be a good application for recursion.

- **Greatest Common Divisor**
- **Fiboncci Numbers**
- **Prefix to Postfix Conversion**
- **The Towers of Honoi**

Greatest Common Divisor Recursive

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

```
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
```

```c
 6   #include <stdio.h>
 7    #include <ctype.h>
 8
 9    // Prototype Statements
10    int gcd (int a, int b);
11
12    int main (void)
13    {
14    // Local Declarations
15       int  gcdResult;
16
17    // Statements
18       printf("Test GCD Algorithm\n");
19
20       gcdResult = gcd (10, 25);
21       printf("GCD of 10 & 25 is %d", gcdResult);
22       printf("\nEnd of Test\n");
23       return 0;
24    }   // main
```

```
25  /* ================= gcd ==================
26      Calculates greatest common divisor using the
27      Euclidean algorithm.
28          Pre   a and b are positive integers greater than 0
29          Post greatest common divisor returned
30  */
31  int gcd (int a, int b)
32  {
33      // Statements
34      if (b == 0)
35          return a;
36      if (a == 0)
37          return b;
38      return gcd (b, a % b);
39  }  // gcd
```

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

Fibonacci Number Recursion

$$\text{Fibonacci } (n) = \begin{bmatrix} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ \text{Fibonacci } (n-1) + \text{Fibonacci } (n-2) & \text{otherwise} \end{bmatrix}$$

For example: 0, 1, 1, 2, 3, 5, 8, 13,……
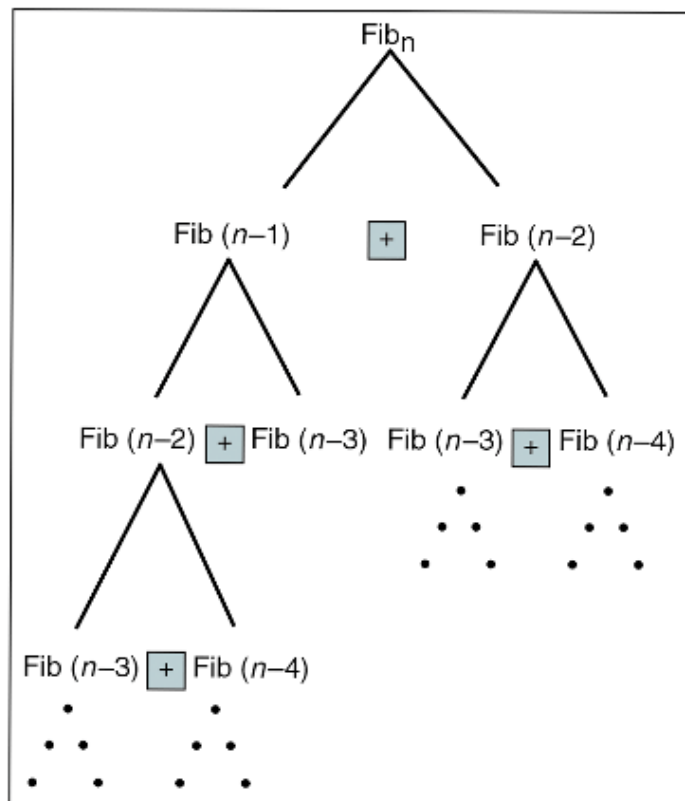
(a) Fib(n)
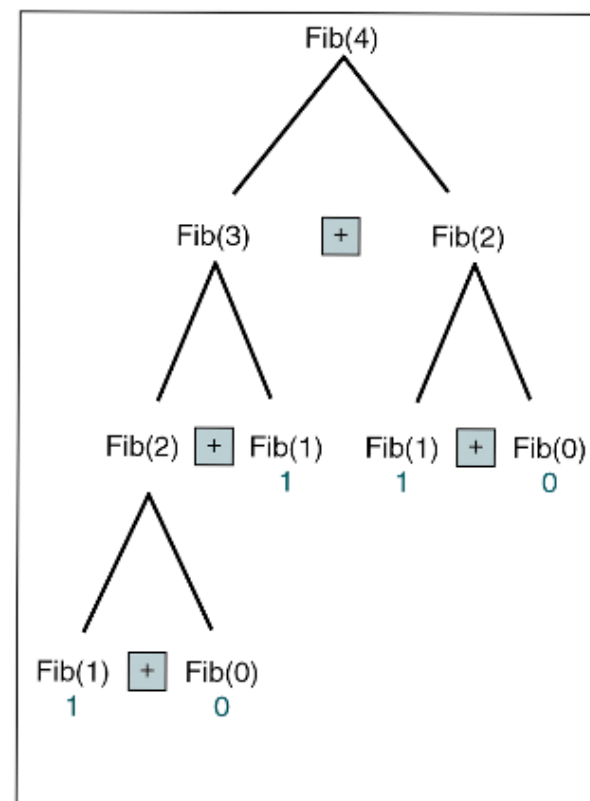
(b) Fib(4)

PROGRAM 2-2 Recursive Fibonacci Series

```
 1  /* This program prints out a Fibonacci series.
 2         Written by:
 3         Date:
 4  */
 5  #include <stdio.h>
 6
 7  // Prototype Statements
 8     long fib (long num);
 9
10  int main (void)
11  {
12  // Local Declarations
13     int seriesSize = 10;
14
15  // Statements
16     printf("Print a Fibonacci series.\n");
17
```

```
18        for (int looper = 0; looper < seriesSize; looper++)
19            {
20              if (looper % 5)
21                  printf(", %8ld", fib(looper));
22              else
23                  printf("\n%8ld", fib(looper));
24            } // for
25      printf("\n");
26      return 0;
27  }  // main
28
29  /* ================= fib =================
30     Calculates the nth Fibonacci number
31         Pre  num identifies Fibonacci number
32         Post returns nth Fibonacci number
33  */
34  long fib (long num)
35  {
36  // Statements
37      if (num == 0 || num == 1)
```

PROGRAM 2-2 Recursive Fibonacci Series (continued)

```
38          // Base Case
39        return num;
40      return (fib (num - 1) + fib (num - 2));
41  }  // fib
```

```
Results:
Print a Fibonacci series.

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

Fibonacci Calls

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

A recursive solution to calculate Fibonacci number is not efficient for large number !

Prefix to Postfix Conversion

- **Prefix:** **+ A B**

- **Infix:** **A + B**

- **Postfix: A B +**

Note in Prefix: the operator is always the first character in the prefix string
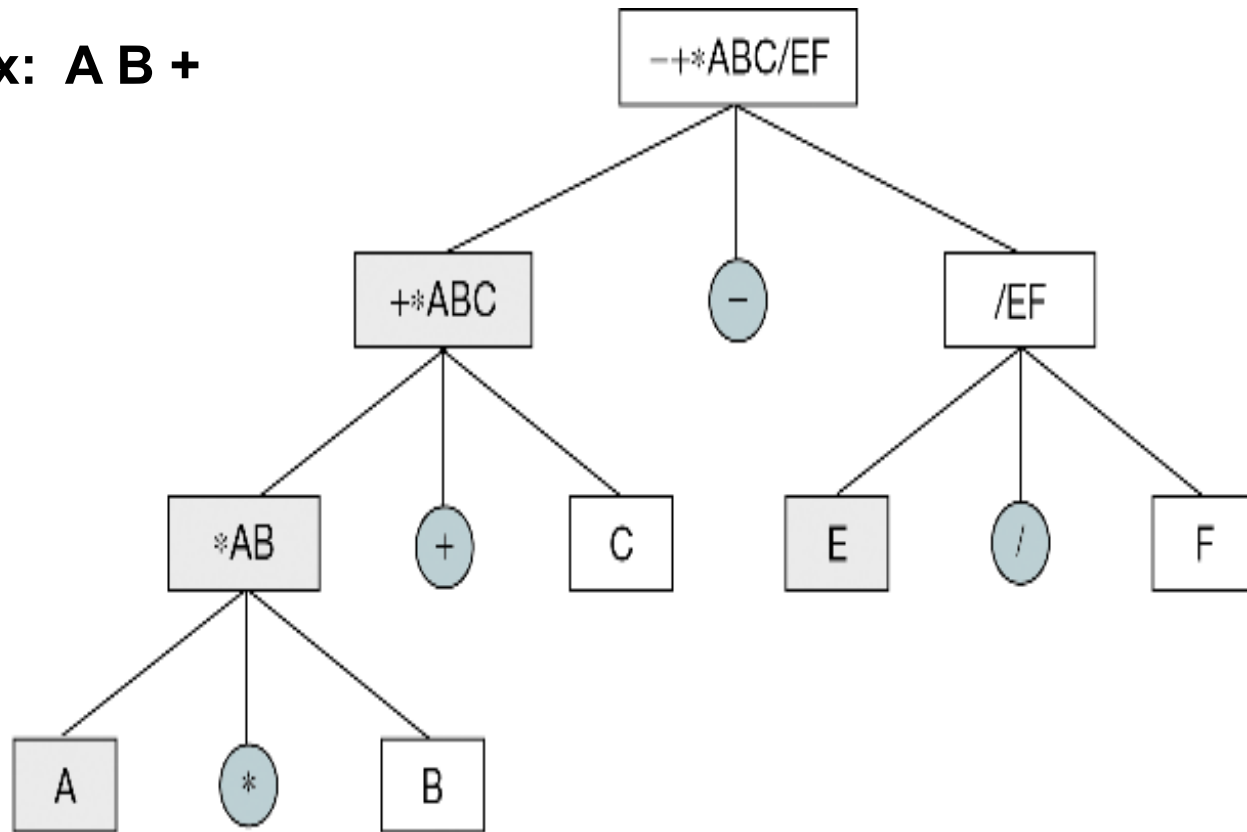
ALGORITHM 2-5 Convert Prefix Expression to Postfix

```
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

```
 1 | /* Convert prefix to postfix expression.
 2 |       Written by:
 3 |       Date:
 4 | */
 5 | #include <stdio.h>
 6 | #include <string.h>
 7 |
 8 | #define OPERATORS "+-*/"
 9 |
10 | // Prototype Declarations
11 | void preToPostFix  (char* preFixIn, char* exprOut);
12 | int  findExprLen    (char* exprIn);
13 |
14 | int main (void)
15 | {
```

```
16    // Local Definitions
17       char   preFixExpr[256]  = "-+*ABC/EF";
18       char   postFixExpr[256] = "";
19
20    // Statements
21       printf("Begin prefix to postfix conversion\n\n");
22
23       preToPostFix (preFixExpr, postFixExpr);
24       printf("Prefix expr:  %-s\n", preFixExpr);
25       printf("Postfix expr: %-s\n", postFixExpr);
26
27       printf("\nEnd prefix to postfix conversion\n");
28       return 0;
29    }  // main
30
```

PROGRAM 2-3   Prefix to Postfix (continued)

```
31   /* =================== preToPostFix ===================
32       Convert prefix expression to postfix format.
33           Pre  preFixIn is string prefix expression
34                expression can contain no errors/spaces
35                postFix is string variable for postfix
36           Post expression has been converted
37   */
38   void preToPostFix (char* preFixIn, char* postFix)
39   {
40   // Local Definitions
41       char  operator [2];
42       char  postFix1[256];
43       char  postFix2[256];
44       char  temp     [256];
45       int   lenPreFix;
46
```

```
47   // Statements
48      if (strlen(preFixIn) == 1)
49         {
50           *postFix        = *preFixIn;
51           *(postFix + 1) = '\0';
52           return;
53         } // if only operand
54
55      *operator        = *preFixIn;
56      *(operator + 1) = '\0';
57
58      // Find first expression
59      lenPreFix = findExprLen (preFixIn + 1);
60      strncpy (temp, preFixIn + 1, lenPreFix);
61      *(temp + lenPreFix) = '\0';
62      preToPostFix (temp, postFix1);
```

continued

```
63
64      // Find second expression
65      strcpy (temp, preFixIn + 1 + lenPreFix);
66      preToPostFix (temp, postFix2);
67
68      // Concatenate to postFix
69      strcpy (postFix, postFix1);
70      strcat (postFix, postFix2);
71      strcat (postFix, operator);
72
73      return;
74   }  // preToPostFix
75
```

```
76  /* ===================== findExprLen =====================
77      Determine size of first substring in an expression.
78          Pre   exprIn contains prefix expression
79          Post  size of expression is returned
80  */
81  int findExprLen (char* exprIn)
82  {
83  // Local Definitions
84      int  len1;
85      int  len2;
86
```

```
 87 | // Statements
 88 |    if (strcspn (exprIn, OPERATORS) == 0)
 89 |          // General Case: First character is operator
 90 |          // Find length of first expression
 91 |        {
 92 |          len1 = findExprLen(exprIn + 1);
 93 |
 94 |          // Find length of second expression
 95 |          len2 = findExprLen(exprIn + 1 + len1);
 96 |        } // if
 97 |    else
 98 |          // Base case--first char is operand
 99 |          len1 = len2 = 0;
100 |    return len1 + len2 + 1;
101 | }  // findExprLen
```

Results:
Begin prefix to postfix conversion

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

End prefix to postfix conversion

Towers of Hanoi

- **Only one disk could be moved at a time and a larger disk must never be stacked above a smaller one**

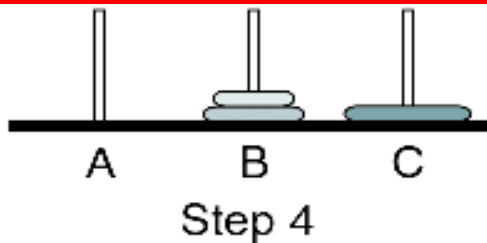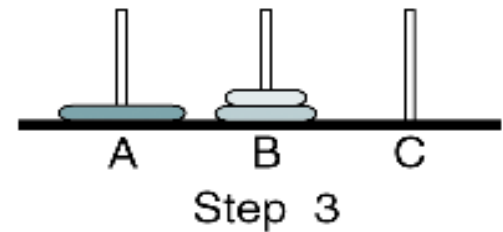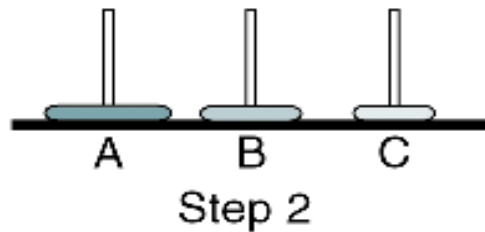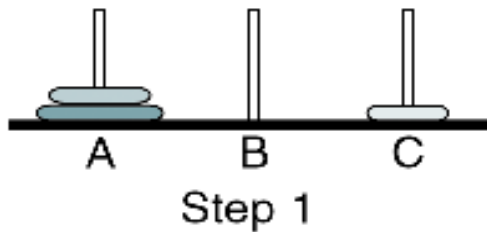- **One and only one auxiliary needle could be used for the intermediate storage of disks**

Source   Auxiliary   Destination
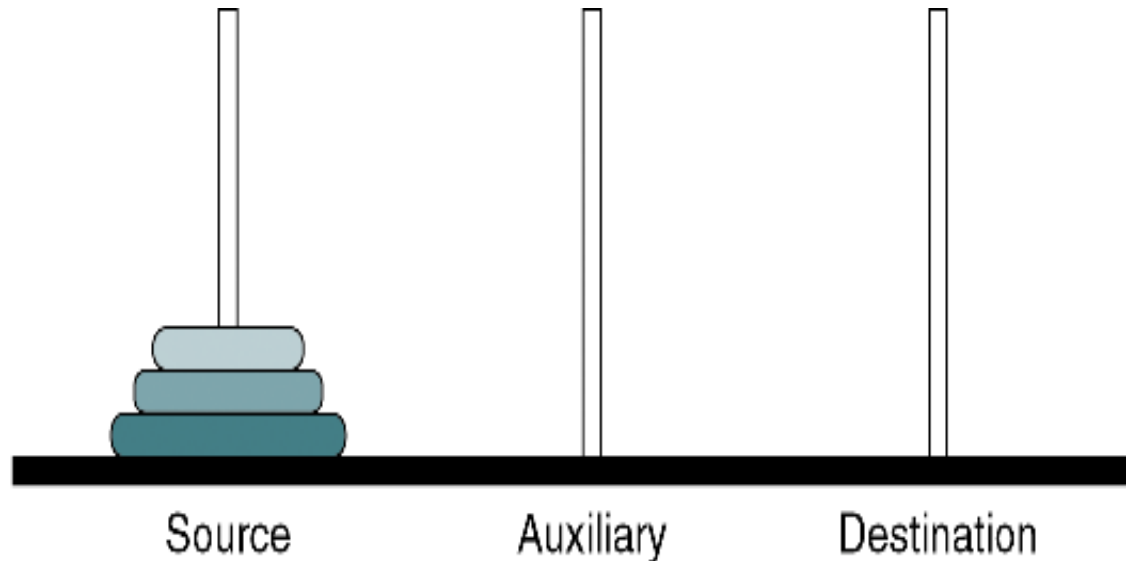
Towers Solution for Two Disks

Source A Aux. B Dest. C
Start

A   B   C
Step 1

A   B   C
Step 2

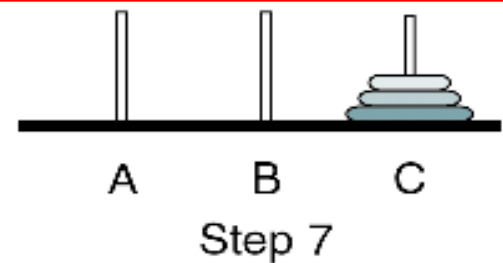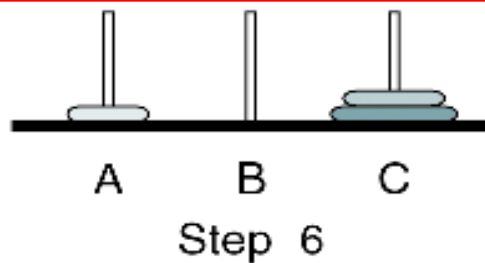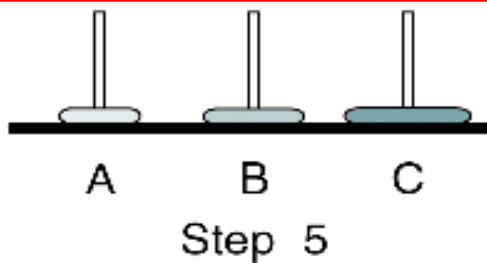A   B   C
Step 3
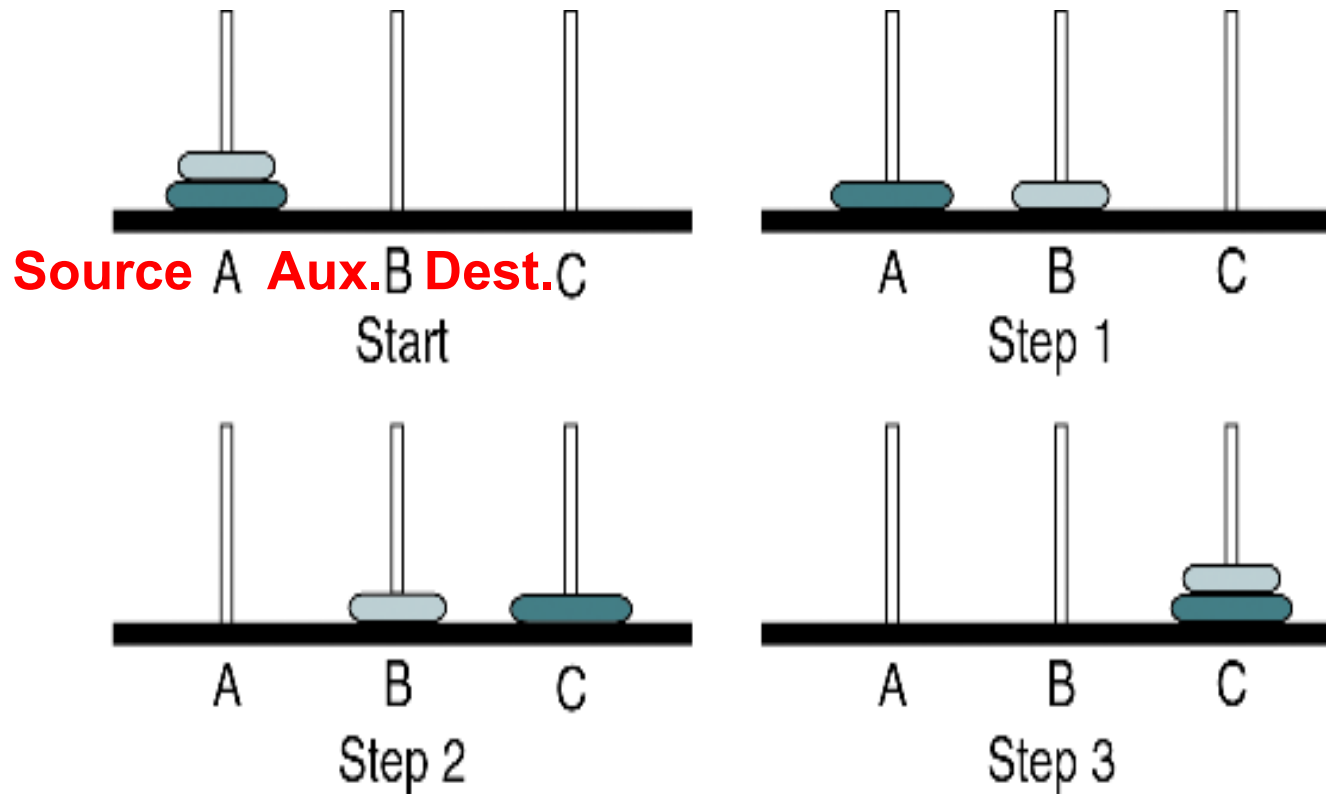
Towers Solution for Three Disks (cont.)
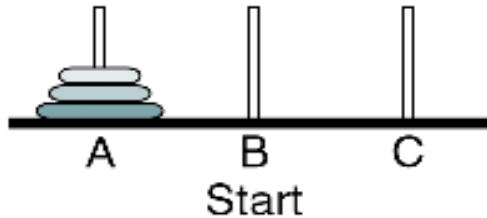
Move 2 disks from A (source) **to B** (dest.)

Move one disk from A to C

**Move 2 disks from B** (source) **to C** (dest.)

Towers Solution for N Disks

- **Generalization of the problem**
  - **General case:** Move n-1 disks from source to auxiliary
  - **Base case:** Move one disk from source to destination
  - **General case:** Move n-1 disks from aux. to destination
- **Four parameters for the algorithm "Towers"**
  - **The number of disks to be moved**
  - **The source needle**
  - **The destination needle**
  - **The auxiliary needle**

Pseudocode note:Tower (num,source, destination, auxiliary)
1. **Call Towers (n-1, source, auxiliary, destination)**
2. **Move one disk from source to destination**
3. **Call Towers (n-1, auxiliary, destination, source**

Towers Solution for Three Disks
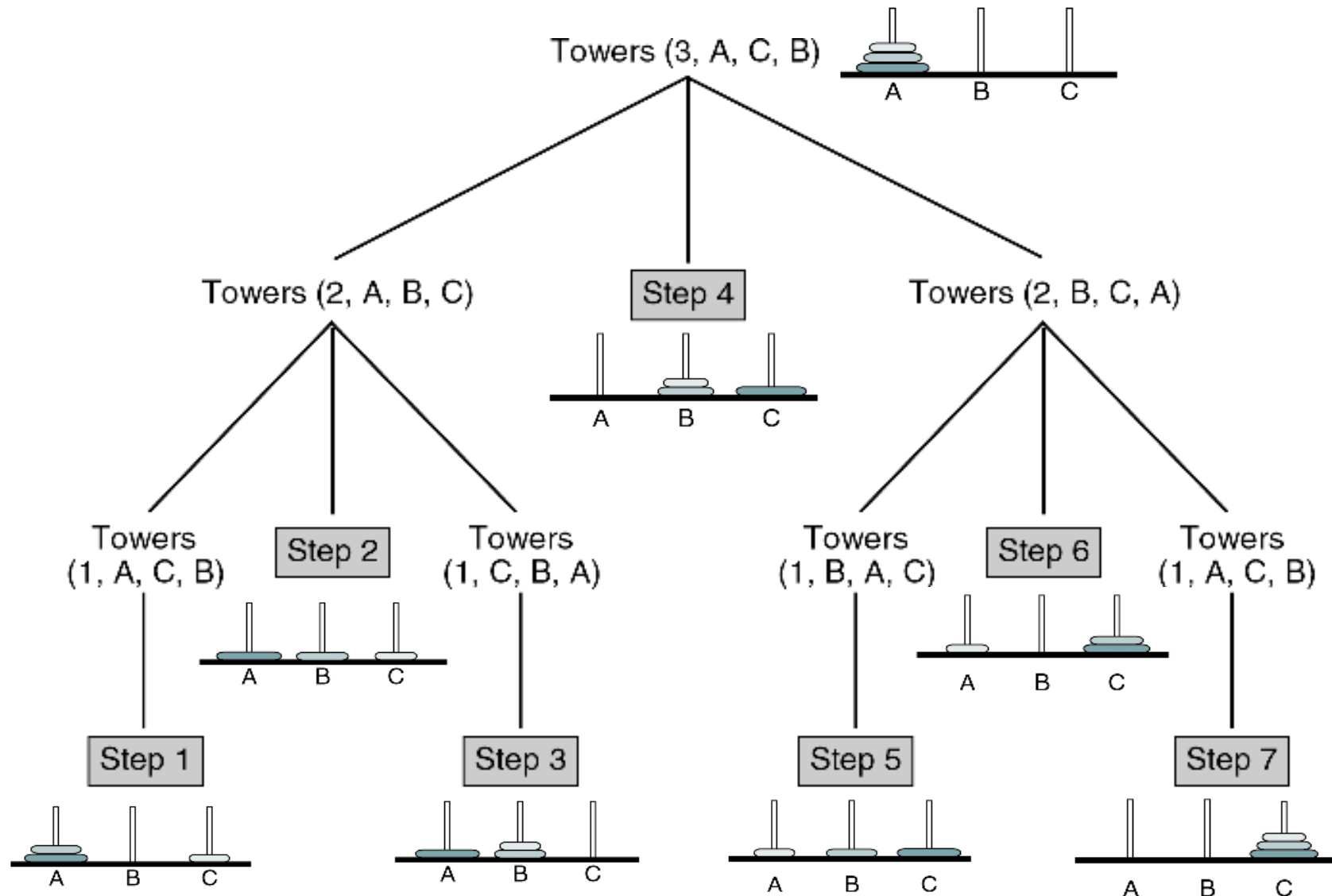
ALGORITHM 2-7 Towers of Hanoi

```
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
```
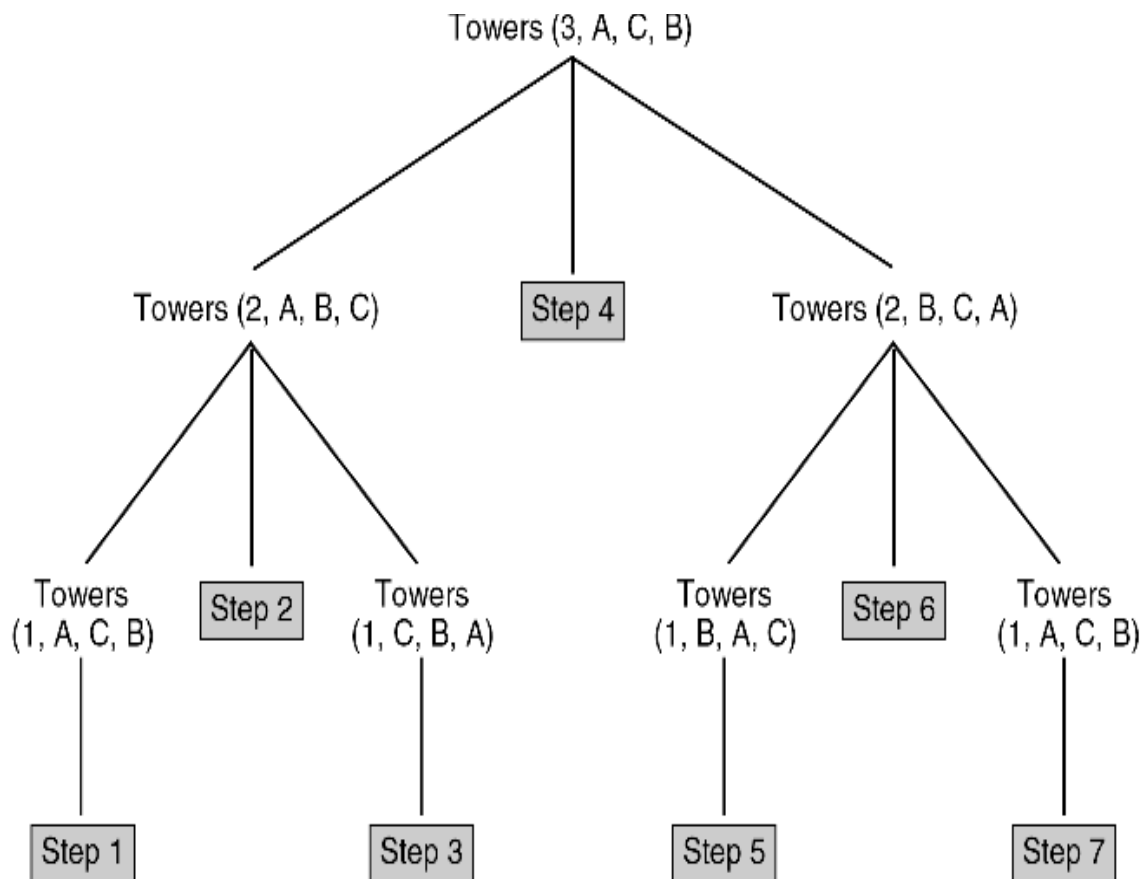
Calls:

```
Towers (3, A, C, B)
Towers (2, A, B, C)
Towers (1, A, C, B)
                    Move from A to C
                    Move from A to B
Towers (1, C, B, A)
                    Move from C to B
                    Move from A to C
Towers (2, B, C, A)
Towers (1, B, A, C)
                    Move from B to A
                    Move from B to C
Towers (1, A, C, B)
                    Move from A to C
```

PROGRAM 2-4 Towers of Hanoi

```
 1  /* Test Towers of Hanoi
 2         Written by:
 3         Date:
 4  */
 5  #include <stdio.h>
 6
 7  // Prototype Statements
 8     void towers (int  n,      char source,
 9                    char dest,  char auxiliary);
10
11  int main (void)
12  {
13  // Local Declarations
14     int numDisks;
15
```

continued

PROGRAM 2-4   Towers of Hanoi (Continued)

```
16    // Statements
17       printf("Please enter number of disks: ");
18       scanf ("%d", &numDisks);
19
20       printf("Start Towers of Hanoi.\n\n");
21
22       towers (numDisks, 'A', 'C', 'B');
23
24       printf("\nI Hope you didn't select 64 "
25               "and end the world!\n");
26       return 0;
27    }  // main
28
```

```
29    /* ====================== towers =======================
30        Move one disk from source to destination through
31        the use of recursion.
32            Pre  The tower consists of n disks
33                 Source, destination, & auxiliary towers
34            Post Steps for moves printed
35    */
36    void towers (int   n,    char  source,
37                     char  dest, char  auxiliary)
38    {
39    // Local Declarations
40        static int step = 0;
41
```

PROGRAM 2-4 Towers of Hanoi (Continued)

```
42   // Statements
43      printf("Towers (%d, %c, %c, %c)\n",
44                       n, source, dest, auxiliary);
45      if (n == 1)
46         printf("\t\t\tStep %3d: Move from %c to %c\n",
47               ++step, source, dest);
48      else
49         {
50           towers (n - 1, source, auxiliary, dest);
51           printf("\t\t\tStep %3d: Move from %c to %c\n",
52               ++step, source, dest);
53           towers (n - 1, auxiliary, dest, source);
54         } // if … else
55      return;
56   } // towers
```