Chapter 6: Recursion

Learning Outcomes

At the end of this lecture, you should be able to

- Describe the concept of recursion
- Solve a problem using recursion
- Trace a recursive method call
- Analyze the efficiency of a recursive solution as compared to other alternative solutions

What Is Recursion?

- It is a problem-solving process that breaks a problem into identical but smaller problems
- Eventually you reach a smallest problem where there is a direct solution
- Using that solution enables you to solve the previous problems
- Eventually the original problem is solved

What Is Recursion? (cont’d)

- Recursion is an alternative to iteration
- It is a very powerful way to solve certain problems for which the solution would otherwise be very complicated

Terminology

Recursion	The process of solving a problem by reducing it to smaller versions of itself.
Recursive definition	A definition in which something is defined in terms of a smaller version of itself.
Stopping case	The case for which the solution is obtained directly.
Recursive case	The case in a recursive algorithm in which the problem is specified as a smaller version of the original problem

Terminology (cont'd)

Recursive algorithm	An algorithm that finds the solution to a given problem by reducing the problem to smaller versions of itself
Recursive method	A method that calls itself. The body of the recursive method contains a statement that causes the same method to execute before completing the current call. Recursive algorithms are implemented using recursive methods.

A recursive story

A mother told a story of a to send her child to sleep

in which a bear told a story of a to send her cub to sleep

in which a cat told a story of a to send her kitten to sleep

kitten went to sleep…zzzz

cub went to sleep…zzzz

child went to sleep….zzz

tellStory(mother)

Recursive definition

tellStory( )

tellStory( )

sleep (child)

TellStory( )

sleep (cub)

tellStory( )

sleep (kitten)

Recursive method

Recursive case

Stopping case

Example: Factorial

- factorial(4) = 4!
- Iterative solution:

  4! = 4 x 3 x 2 x 1 = 24

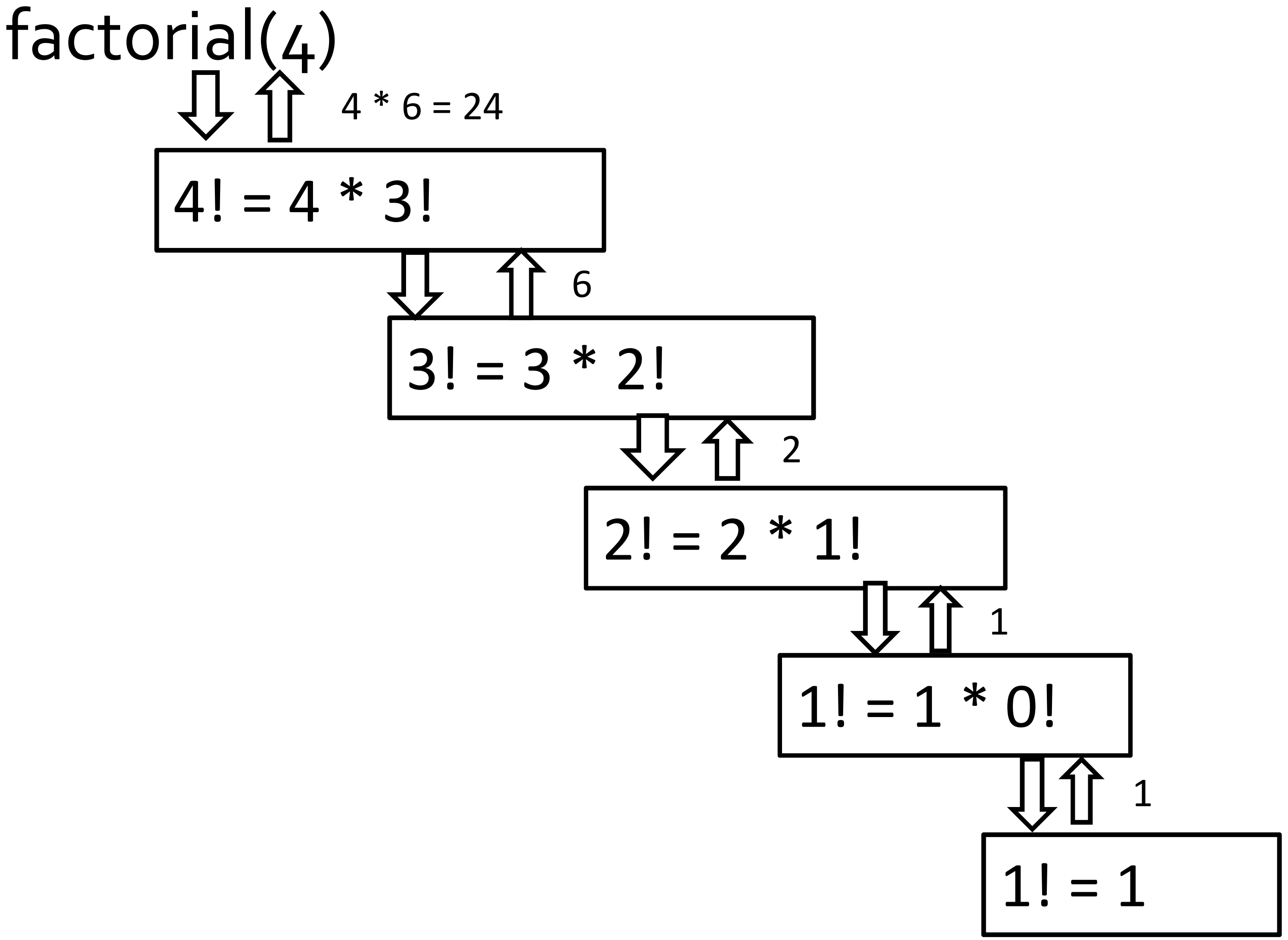
- i.e., in general:
  - 0! = 1
  - n! = n x (n-1) x (n-2) x ..x 2 x 1 for n > 0

Factorial: Recursive Definition

- Recursive solution:

$4! = 4 \times 3!$

Factorial: Recursion Trace



Principles of Recursion

1. Every recursive definition must have one or more stopping cases.
2. The recursive case must eventually be reduced to a stopping case
3. The stopping case stops the recursion

Implementation of method `factorial()`

```
public int factorial(int n) {
  if (n==0 || n==1) // Stopping cases
    return 1;
  else
    return n * factorial(n-1);
}
```

Note: all sample code in this chapter is found in

- The **`Chapter6\samplecode\`** folder

Exercise 1

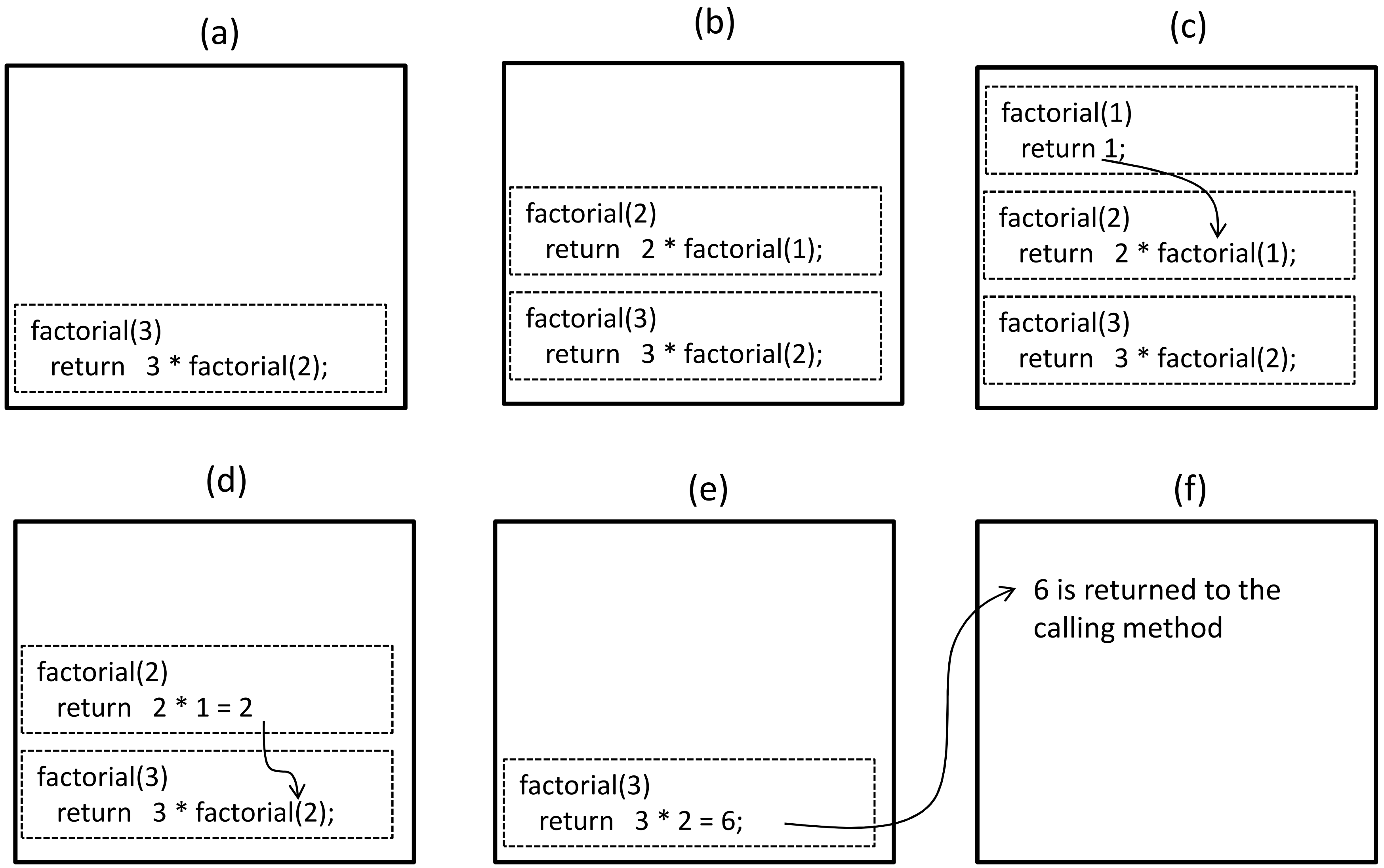
Write a recursive method to compute the following series:

`sumSeries(n)= 1 + 1/2 + 1/3 + ⋯ + 1/n,`

where n is a positive integer value.

For example: if n is 2, the method performs the computation `1 + 1/2` and it returns 1.5.

Recursive Calls in Program Stack



(a)

```
factorial(3)
  return   3 * factorial(2);
```

(b)

```
factorial(2)
  return   2 * factorial(1);
```

```
factorial(3)
  return   3 * factorial(2);
```

(c)

```
factorial(1)
  return 1;
```

```
factorial(2)
  return   2 * factorial(1);
```

```
factorial(3)
  return   3 * factorial(2);
```

(d)

```
factorial(2)
  return   2 * 1 = 2
```

```
factorial(3)
  return   3 * factorial(2);
```

(e)

```
factorial(3)
  return   3 * 2 = 6;
```

(f)

6 is returned to the calling method

About the Recursion Trace (1)

- We can illustrate the execution of a recursive method by doing a recursion trace or box trace.
- Each box corresponds to a recursive call. In each box, indicate:
  - The values of arguments for the current method invocation
  - The statements that were executed
- Each new recursive method call is indicated by a down arrow (↓)to the newly called method.
- When the method returns, an upward arrow (↑) is drawn and the return value is indicated.

Exercise 2

Given the following recursive method:

```
public void isChar(char val) {
  if (val > 'A') {
     isChar(--val);
     System.out.printf("%c ", val);
  }
}
```

Perform a **box trace** for the method call `isChar('D')`. For each method call, indicate the argument value and the statement(s) executed for each box

```
Display A B C
```

About the Recursion Trace (2)

- Logically, you can think of a recursive method as having unlimited copies of itself.
- Every recursive call has its own code and its own set of parameters and local variables.
- After completing a particular recursive call, the control goes back to the calling environment, which is the previous call. The current (recursive) call must execute completely before the control goes back to the previous call. The execution in the previous call begins from the point immediately following the recursive call.

When Designing Recursive Solution

- Method definition must provide parameter
  - Leads to different cases
  - Typically includes an **`if`** or a **`switch`** statement
- One or more of these cases should provide a non recursive solution: the stopping (base) case
- One or more cases includes recursive invocation: takes a step towards the stopping case

Implementing Recursive Methods

General structures:

(a) // Stopping case and recursive case have different actions

```
If the stopping case is reached
    Solve the problem directly
Else
    Recursively solve smaller version of
    the problem
```

(b) // Stopping case and recursive case have common action

```
Perform common step
If recursive case
  Recursively solve smaller version of
  the problem
```

(c) // Stopping case has no actions to be performed

```
If recursive case
  Recursively solve smaller version
```

`void` Recursive Methods

- Methods that do not return a value
- *E.g.*, to display a count down from the given number

`countDown` - Implementation 1

```
public void countDown(int n) {
  if (n == 1)
      System.out.println(n);
  else {
    System.out.println(n);
    countDown(n - 1);
  }
}
```

- Observations:
  - The stopping case is considered first.
  - Redundant **`println`** statement occurs in both cases.
- Sample code: **`CountDown.java`**

countDown - Implementation 2

```
public void countDown(int n) {
  if (n >= 1) {
    System.out.println(n);
    countDown(n - 1);
  }
}
```

- Sample code: **CountDown2.java**
- Observations:
  - Redundant **println** statement has been removed.
  - The recursive case is checked.
    - When **n** is **1**, this method will invoke the recursive call **countDown(0)**. This is the stopping case and no action is performed.

countDown - Implementation 3

```
public void countDown(int n) {
  System.out.println(n);
  if(n > 1)
    countDown(n - 1);
}
```

- Sample code: **CountDown3.java**
- Observations: when the method is invoked,
  - The current **n** value is first displayed.
  - The recursive case is checked (note: this version uses **n > 1** instead of **n >= 1**)

    ➔ Will have 1 less recursive call compared to the previous implementation (**CountDown2.java**):

Exercise 3

- When you design a recursive solution to a problem, what are the sequence of steps to be carried out?
  - ❑ List the stopping (base) cases
  - ❑ List the recursive cases
    - ○ Ensure that they take a step towards the stopping case
  - ❑ Arrange the cases in the correct sequence

Exercise 4

The mathematical function *C(n, k)* computes the number of po
combinations for selecting *k* objects out of *n* and is defined as:

```
            ⎧ 1                                if  k = 0
            ⎪ 1                                if  k = n
C(n,k) =   <  0                                if  k > n
            ⎪ C(n - 1, k - 1) + C(n - 1, k)    if  0 < k < n
            ⎩
```

Write a <u>recursive method</u> which computes *C(n, k)*.

Box trace for method **`countDown()`**

```
public static void countDown(int n) {

  System.out.println(n);
  if(n > 1)
    countDown(n - 1);
}
```

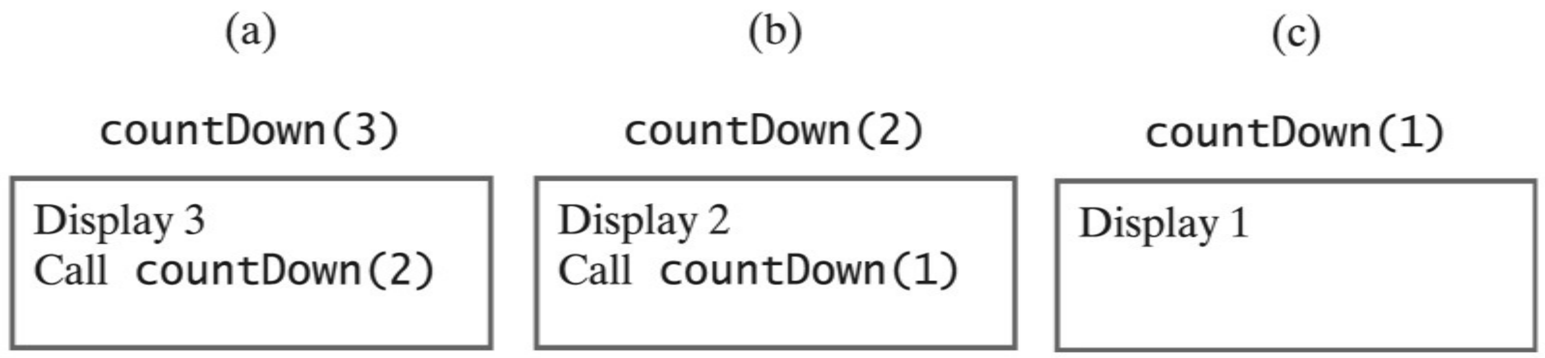
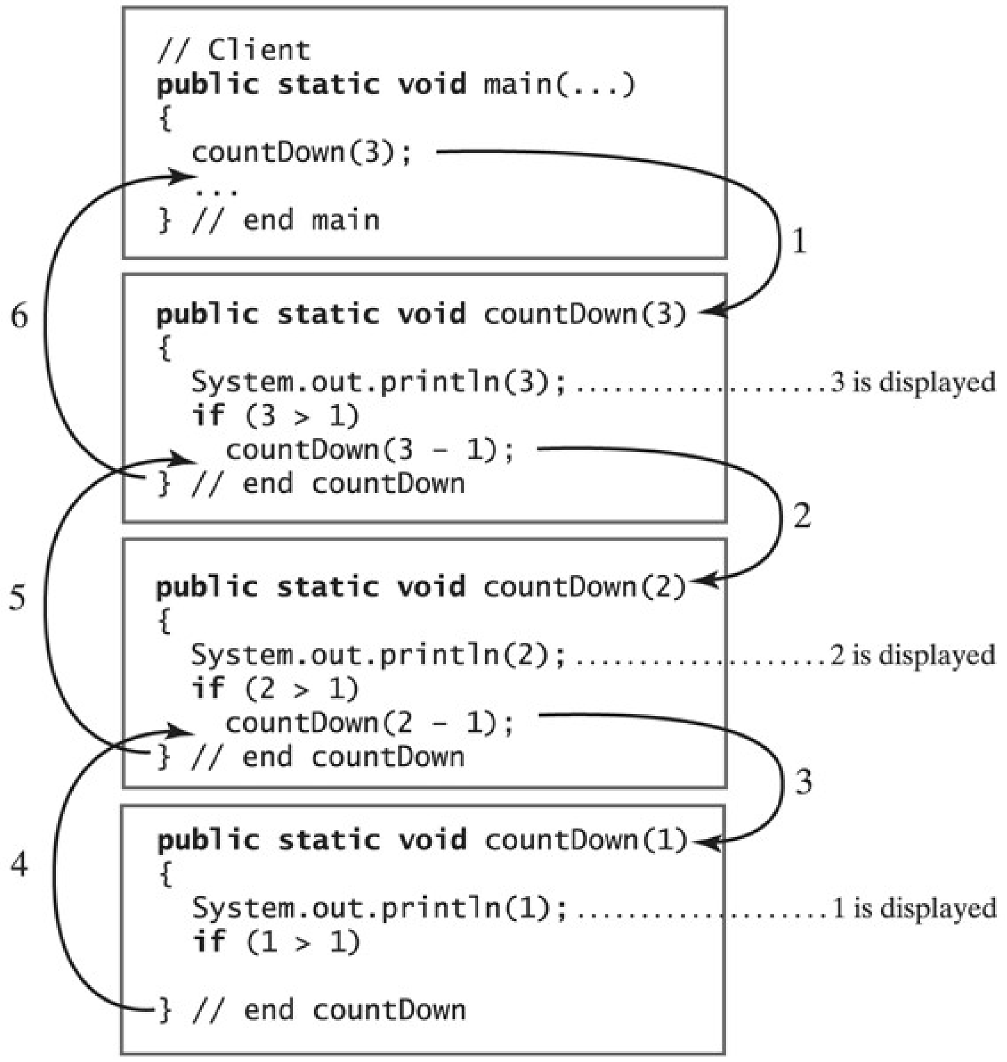


Fig. 6.1: The effect of method call **`countDown(3)`**

Tracing a Recursive Method



```
// Client
public static void main(...)
{
  countDown(3);
  ...
} // end main

public static void countDown(3)
{
  System.out.println(3);
  if (3 > 1)
    countDown(3 - 1);
} // end countDown

public static void countDown(2)
{
  System.out.println(2);
  if (2 > 1)
    countDown(2 - 1);
} // end countDown

public static void countDown(1)
{
  System.out.println(1);
  if (1 > 1)

} // end countDown
```

Fig. 6.2: Tracing the recursive call **`countDown(3)`**

Tracing a Recursive Method

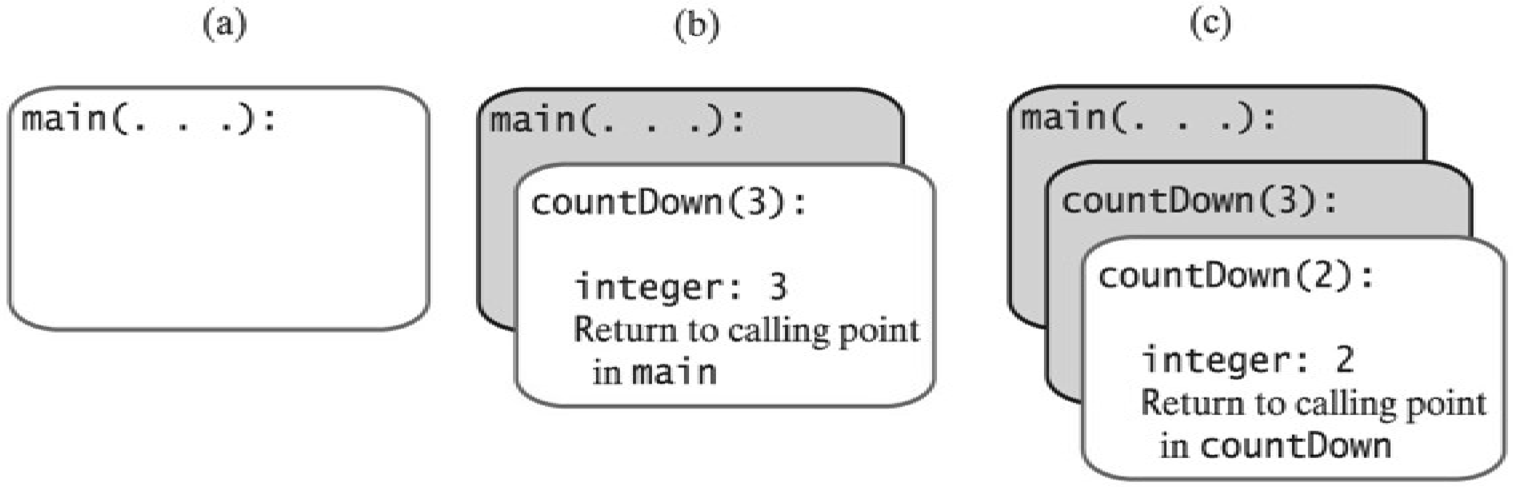


Fig. 6.3: The stack of activation records during the execution of a call to **`countDown(3)`** (continued →)

Tracing a Recursive Method

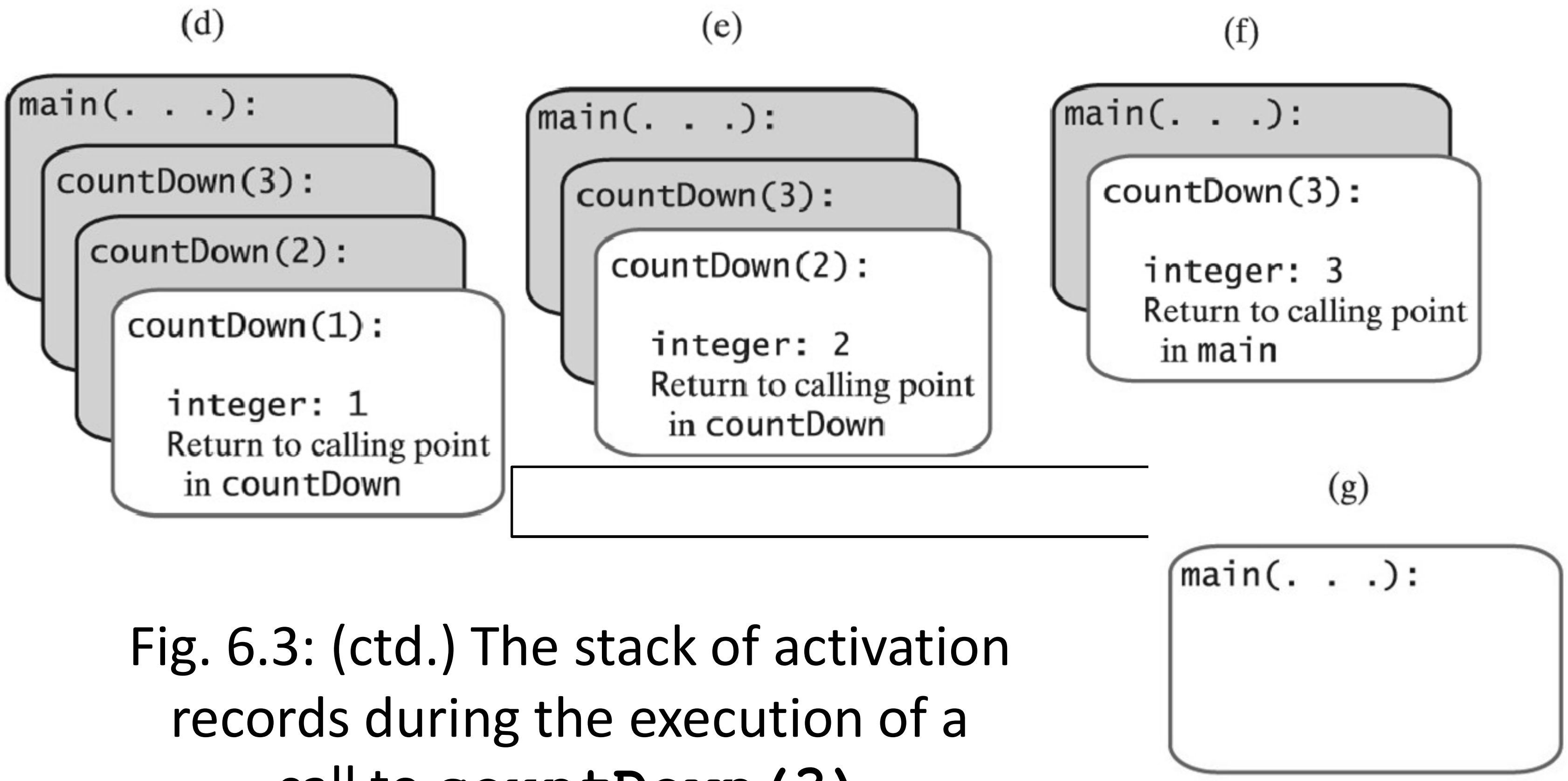


Fig. 6.3: (ctd.) The stack of activation records during the execution of a call to **`countDown(3)`**

Recursively Processing an Array

- When processing array recursively, divide it into two pieces
  a) First element one piece, rest of array another
  b) Last element one piece, rest of array another
  c) Divide array into two halves

- *A recursive method part of an implementation of an ADT is often* **`private`**
  – *Its necessary parameters make it unsuitable as an ADT operation*

Recursively Processing an Array

- Sample code: **`RecursiveDisplayArray.java`**

  Note recursive **`private`** methods:

  `(a)displayArray1()`

  - displays the first array element and then recursively displays the rest.

  (b) **`displayArray2()`**

  - displays the last array element after recursively displaying the all the preceding elements.

  (c) **`displayArray3()`**

  - Divides the array into 2 halves and then recursively displays the left subarray and the right subarray elements

Recursively Processing an Array

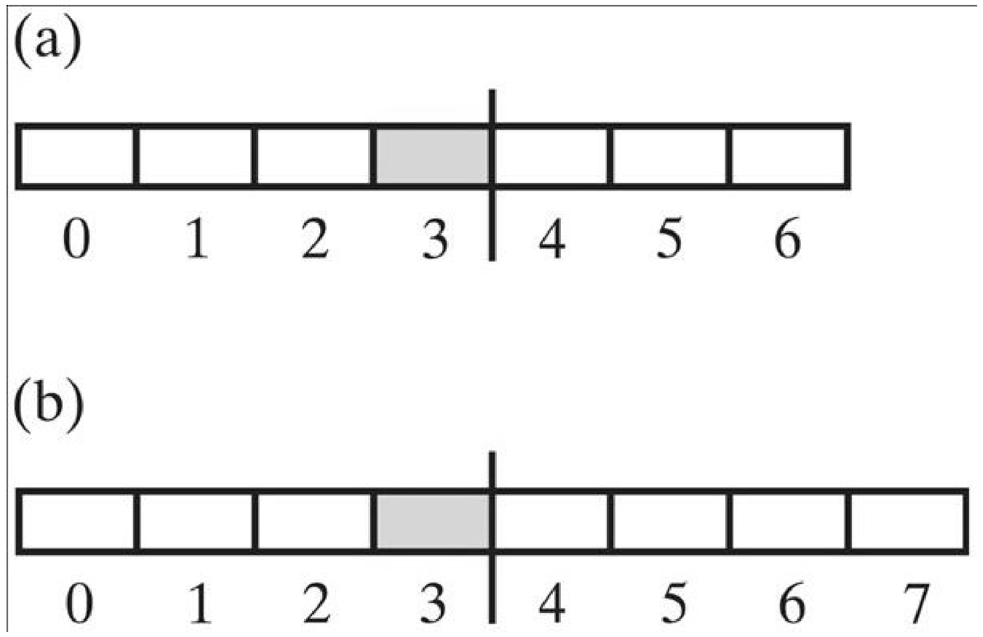


Fig. 6.4: Two arrays with middle elements within left halves

Recursively Processing a Linked Chain

- Sample code: **`SimpleList.java`**
- Consider the private **`toString`** method
  - Processes a chain of linked nodes recursively
    - Use a reference to the chain's first node as the method's parameter
    - Then process the first node
    - Followed by the rest of the chain (*note recursive call*)

Time Efficiency of Recursive Methods

- For the **`countDown`** method

```
   public static void countDown(int n) {
     System.out.println(n);
     if(n > 1)
       countDown(n - 1);
   }
```

  - The efficiency is **`O(n)`**

A Simple Solution to a Difficult Problem: **`Towers of Hanoi`**

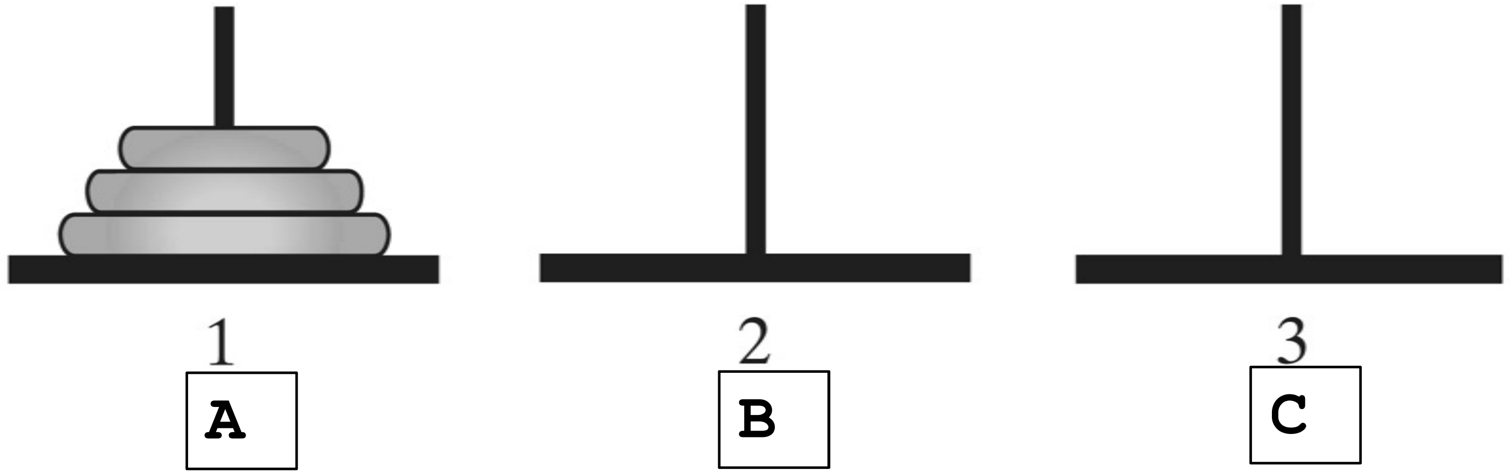


Fig. 6.5: The initial configuration of the **`Towers of Hanoi`** for three disks

A Simple Solution to a Difficult Problem: **`Towers of Hanoi`**

Rules for the **`Towers of Hanoi`** game

1. Move one disk at a time. Each disk you move must be a topmost disk.
2. No disk may rest on top of a disk smaller than itself.
3. You can store disks on the second pole temporarily, as long as you observe the previous two rules.

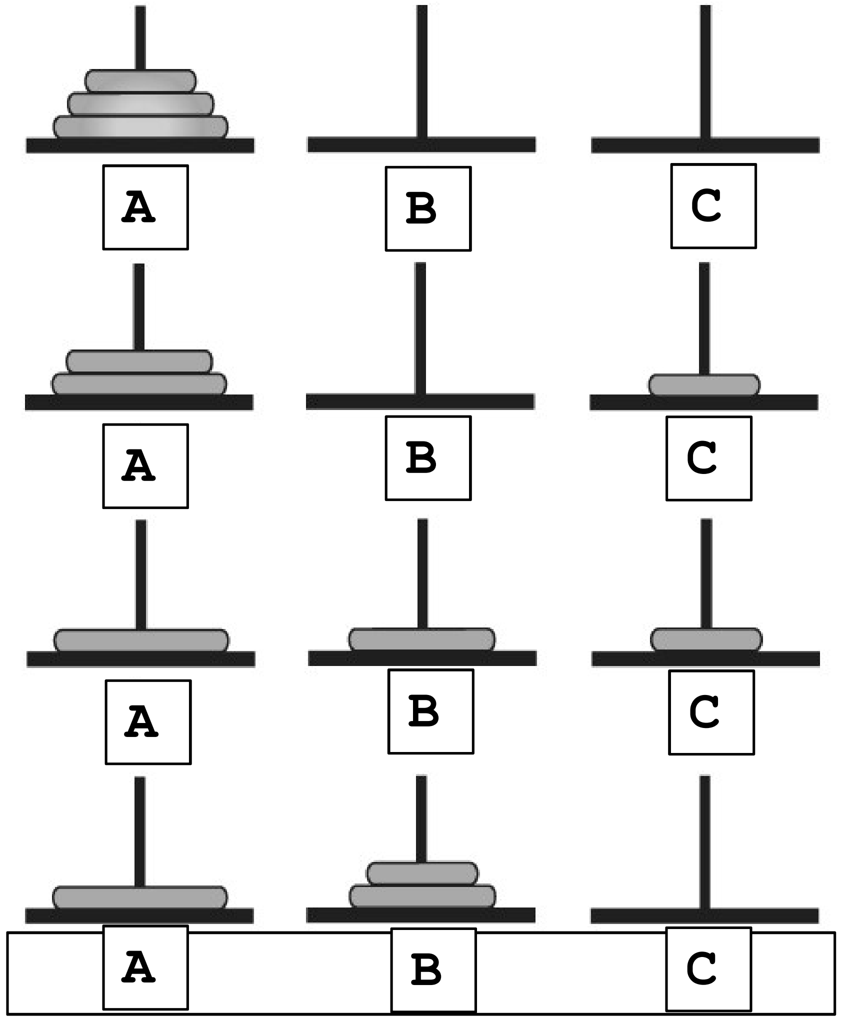
A Simple Solution to a Difficult Problem: **`Towers of Hanoi`**

- Problem 1: Move a disk from peg A to peg C
  - Solution: Directly move 1 disk from peg A to peg ***C***.
- Problem 2: Move 2 disks from pegs A to C
  - Solution: Move the smaller disk from peg A to peg ***B*** (temporarily), then move larger disk to peg ***C*** and finally move smaller disk from peg ***B*** to peg ***C***
- Problem 3: Move 3 disks from pegs A to C
  - Try to solve problem 3.

A Simple Solution to a Difficult Problem: **`Towers of Hanoi`**

Fig. 6.6: The sequence of moves for solving the Towers of Hanoi problem with three disks.

(Continued →)



A Simple Solution to a Difficult Problem: **`Towers of Hanoi`**

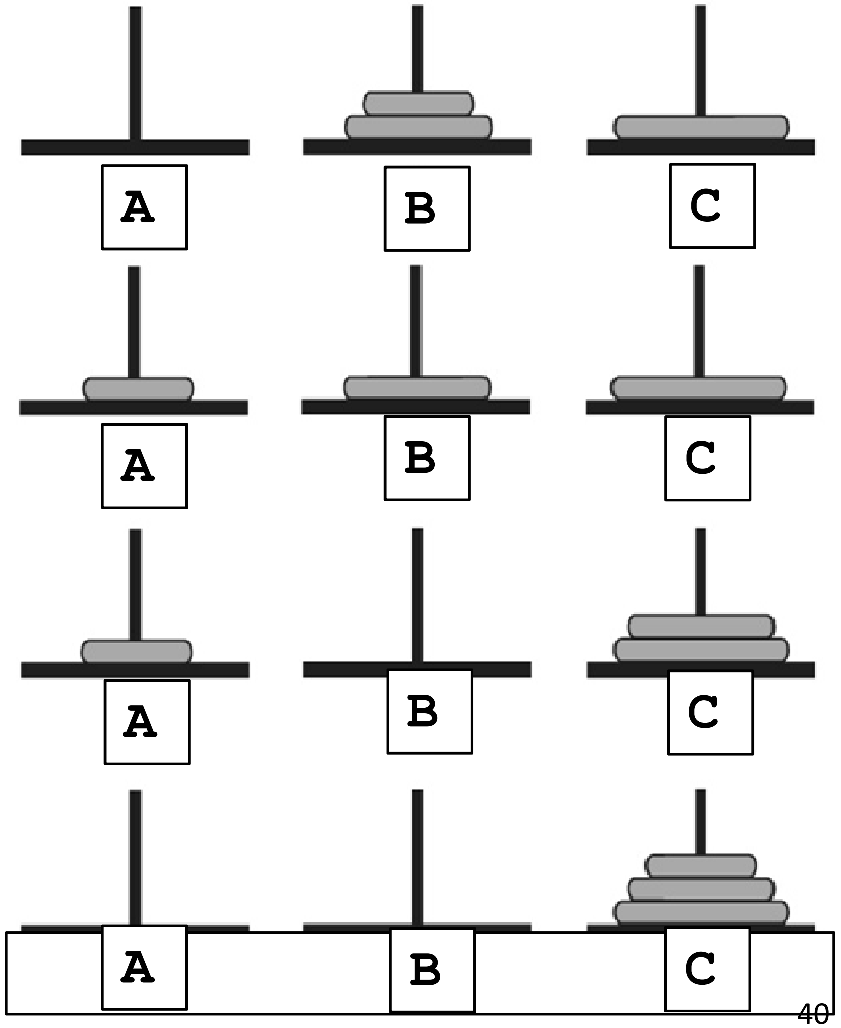


Fig. 6.6: (ctd) The sequence of moves for solving the Towers of Hanoi problem with 3 disks

A Simple Solution to a Difficult Problem: **`Towers of Hanoi`**

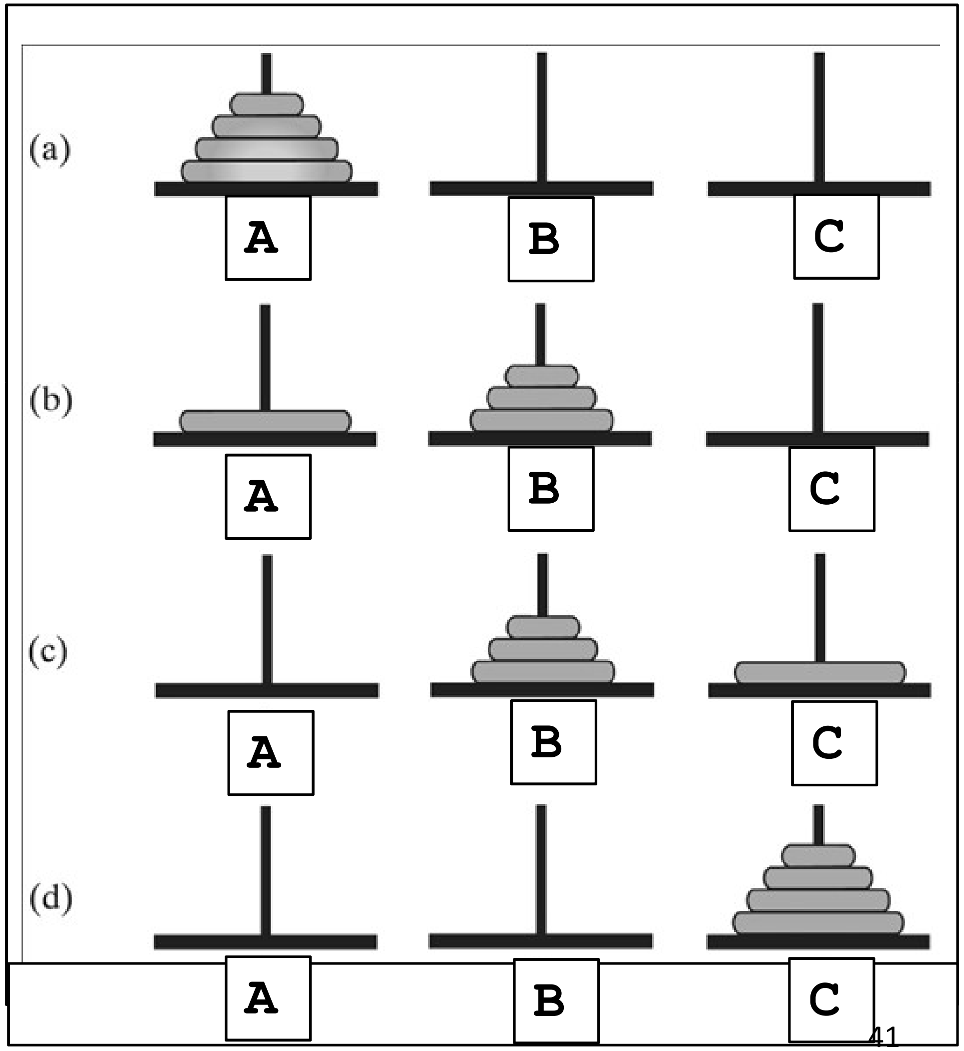


Fig. 6.7: The smaller problems in a recursive solution for four disks

A Simple Solution to a Difficult Problem: **Towers of Hanoi**

Algorithm:

```
Algorithm solveTowers (numberOfDisks, startPole, tempPole,
                       endPole)
  if (numberOfDisks == 1)
    Move disk from startPole to endPole
  else {
    solveTowers (numberOfDisks - 1, startPole, endPole,
                 tempPole)
    Move disk from startPole to endPole
    solveTowers (numberOfDisks - 1, tempPole, startPole,
                 endPole)
}
```

Listing 6.8: The algorithm to solve **Towers of Hanoi**

Advantages of Recursion

- For some problems, a recursive implementation can be significantly simpler and easier to understand than an iterative implementation.
- A recursive approach to algorithm allows us to take advantage of the repetitive structure present in many problems (e.g., folders have subfolders, etc). By making our algorithm description exploit this repetitive structure in a recursive way, we can often avoid complex case analyses and nested loops.

A Poor Solution to a Simple Problem

- Fibonacci numbers
  - First two numbers of sequence are 1 and 1
  - Successive numbers are the sum of the previous two
  - 1, 1, 2, 3, 5, 8, 13, …
- This has a natural looking recursive solution
  - Turns out to be a poor (inefficient) solution

A Poor Solution to a Simple Problem

- The recursive algorithm

```
Algorithm Fibonacci(n)
   if (n <= 1)
      return 1
   else
      return Fibonacci(n-1) + Fibonacci(n-2)
```

Analysis of the recursive Fibonacci solution (1)

Let f_n denote the number of calls performed in the execution of `Fibonacci(n)`. Then, we have the following values for the f_n's:

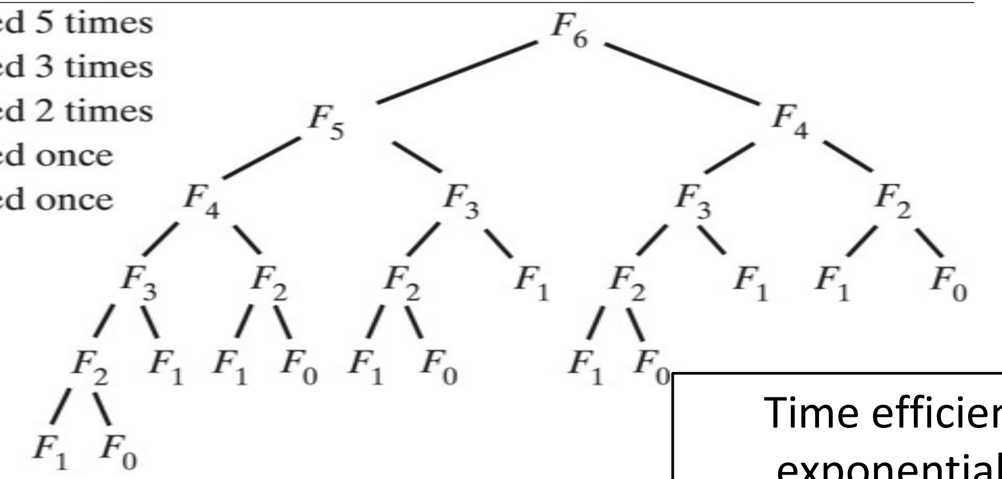
$$f_0 = 1$$
$$f_1 = 1$$
$$f_2 = f_1 + f_0 + 1 = 1 + 1 + 1 = 3$$
$$f_3 = f_2 + f_1 + 1 = 3 + 1 + 1 = 5$$
$$f_4 = f_3 + f_2 + 1 = 5 + 3 + 1 = 9$$
$$f_5 = f_4 + f_3 + 1 = 9 + 5 + 1 = 15$$
$$f_6 = f_5 + f_4 + 1 = 15 + 9 + 1 = 25$$
$$f_7 = f_6 + f_5 + 1 = 25 + 15 + 1 = 41$$

Analysis of the recursive Fibonacci solution (2)

- If we follow the pattern forward, we see that the number of calls more than doubles for each <u>two consecutive indices</u>. I.e., f_4 is more than twice f_2, f_5 is more than twice f_3, f_6 is more than twice f_4, and so on.
- This means $f_n > 2^{n/2}$, which means that `Fibonacci(n)` makes a number of calls that are exponential in n.
- The algorithm is $O(2^n)$, which is very inefficient.

A Poor Solution to a Simple Problem

(a) F_2 is computed 5 times
F_3 is computed 3 times
F_4 is computed 2 times
F_5 is computed once
F_6 is computed once



Time efficiency grows exponentially with n

(b)
$F_0 = 1$
$F_1 = 1$
$F_2 = F_1 + F_0 = 2$
$F_3 = F_2 + F_1 = 3$
$F_4 = F_3 + F_2 = 5$
$F_5 = F_4 + F_3 = 8$
$F_6 = F_5 + F_4 = 13$

Iterative solution is O(n)

Fig. 6.9: The computation of the Fibonacci number F_6
(a) recursively; (b) iteratively

Tail Recursion

- Occurs when the last action performed by a recursive method is a recursive call

```
public static void countDown(int n) {
  if (n >= 1) {
    System.out.println(n);
    countDown(n - 1);
  }
} // end countDown
```

- This performs a method is usually straightforward repetition that can be done more efficiently with iteration
- Conversion to iterative

Tail Recursion (cont'd)

- The tail recursion simply repeats the method's logic with changes to the parameters.

  ➔Can perform the same repetition by using iteration.
  - Replace the **`if`** statement with a **`while`** statement.
  - Replace the recursive call with a statement to update the parameter
  - Sample code: **`CountDown4.java`**

```
public static void countDown(int n) {
  while (n >= 1) {
    System.out.println(n);
    n--;
  }
} // end countDown
```

Mutual Recursion

- Another name for indirect recursion
- Happens when Method A calls Method B
  - Which calls Method B
  - Which calls Method A
  - etc.
- Difficult to understand and trace
  - Happens naturally in certain situations

Mutual Recursion (cont'd)

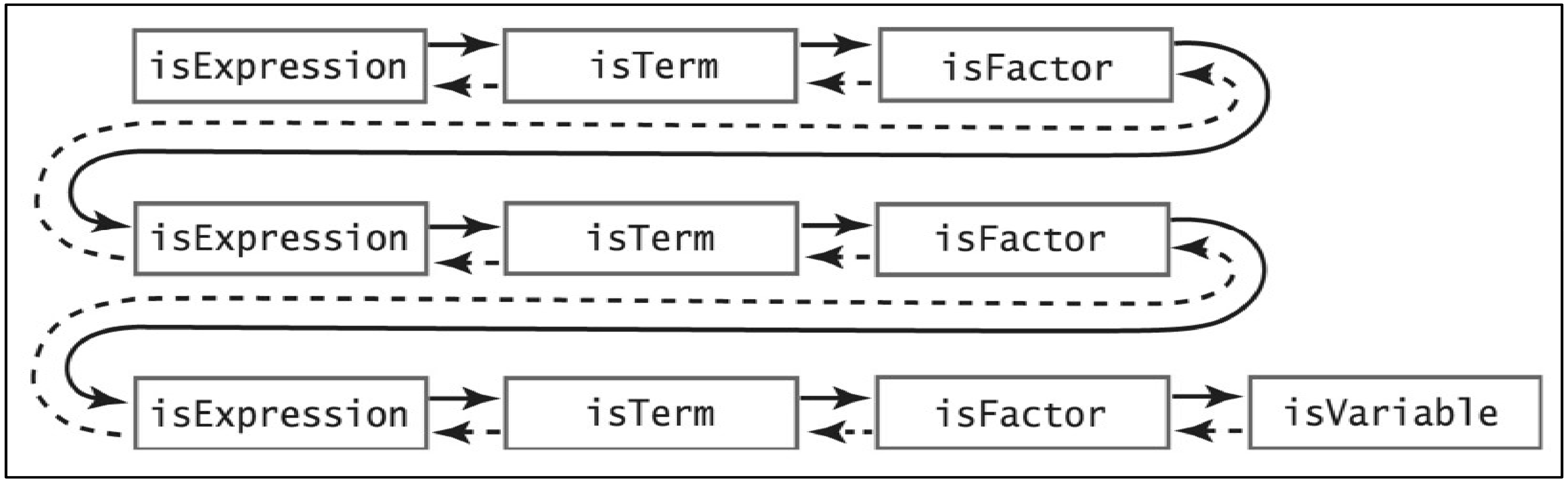


Fig. 6.10: An example of mutual recursion.

Iteration or Recursion?

- There are usually 2 ways to solve a problem – iteration and recursion.
- There's no simple answer to which way is better.
- Factors to consider:
  - The nature of the problem
  - Efficiency

Discussion on Recursive Solutions (1)

- When a method is called, memory space for its formal parameters and local variables is allocated. When the method terminates, the memory space is then deallocated.
- This means that every recursive call requires the system to allocate memory space for its formal parameters and local variables, and then deallocate the memory space when the method exits.
- Thus, there is overhead associated with executing a (recursive) method both in terms of memory space and computer time. Therefore, a recursive method executes more slowly than its iterative counterpart.

Discussion on Recursive Solutions (2)

- Today’s computers, however, are fast and have inexpensive memory. Therefore, the execution of a recursive method is not noticeable.
- Keeping the power of today’s computer in mind, the choice between 2 alternatives – recursion or iteration – depends on the nature of the problem.
- Of course, for problems such as mission control systems, efficiency is absolutely critical and therefore, the efficiency factor would dictate the solution method.

Discussion on Recursive Solutions (3)

- As general rule, if you think that an iterative solution is more obvious and easier to understand than a recursive solution, use the iterative solution, which would be more efficient.
- On the other hand, problems exist for which the recursive solution is more obvious or easier to construct. Keeping the power of recursion in mind, if the definition of a problem is inherently recursive, then you should consider a recursive solution.

References

- Carrano, F. M., 2019, Data Structures and Abstractions with Java, 5th edn, Pearson
- Liang, Y.D., 2018. Introduction to Java Programming and Data Structures.11th ed.United Kingdom:Pearson

Learning Outcomes

You should now be able to

- Describe the concept of recursion
- Solve a problem using recursion
- Trace a recursive method call
- Analyze the efficiency of a recursive solution as compared to other alternative solutions