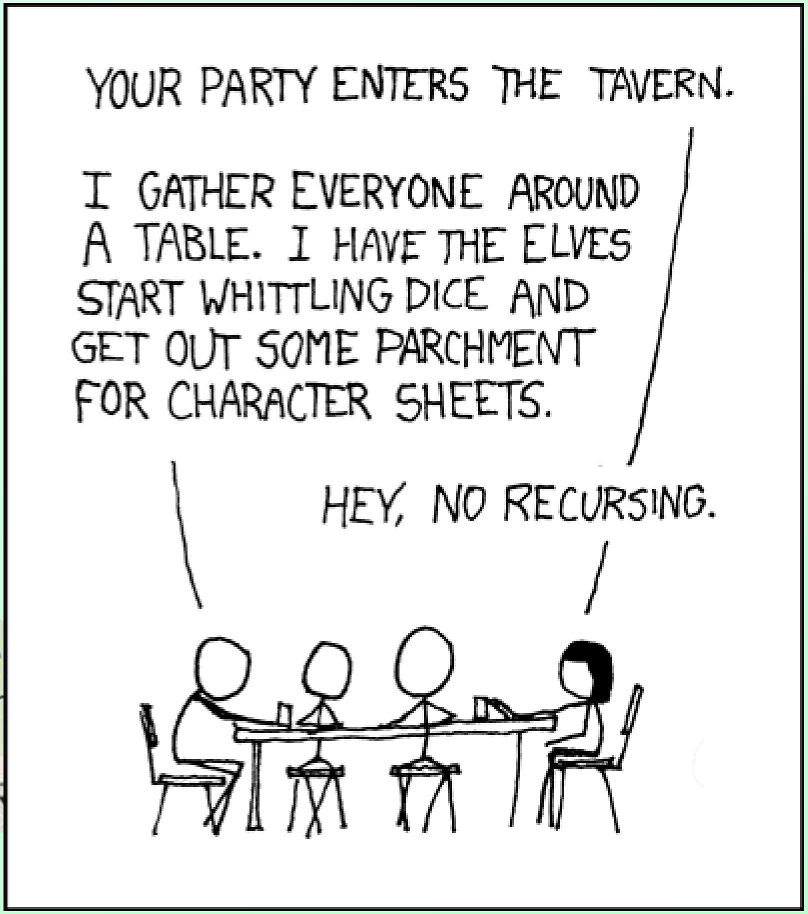
Recursion

Chapter 7



Contents

Contents

Objectives

- Decide whether given recursive method will end successfully in finite amount of time
- Write recursive method
- Estimate time efficiency of recursive method
- Identify tail recursion and replace it with iteration

What Is Recursion?

- We often solve a problem by breaking it into smaller problems
- When the smaller problems are identical (except for size)
  - This is called “recursion”
- Repeated smaller problems
  - Until problem with known solution is reached
- Example: Counting down from 10

Figure 7-1 Counting down from 10

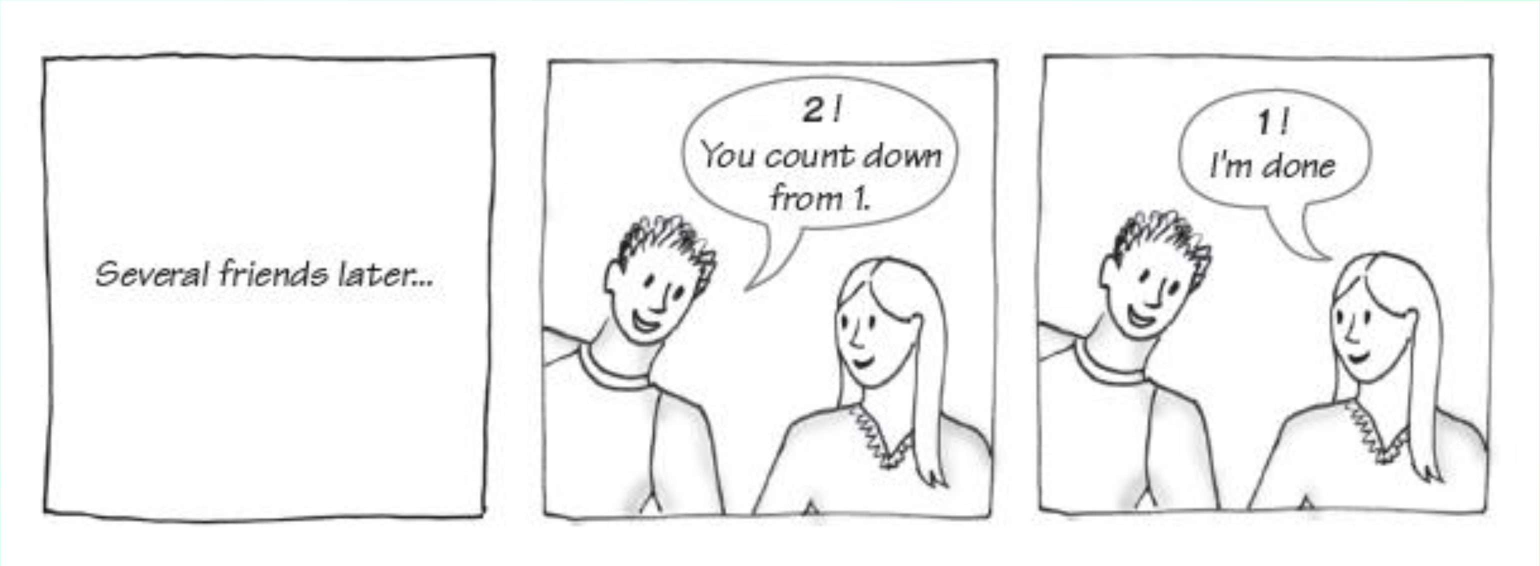


Figure 7-1 Counting down from 10

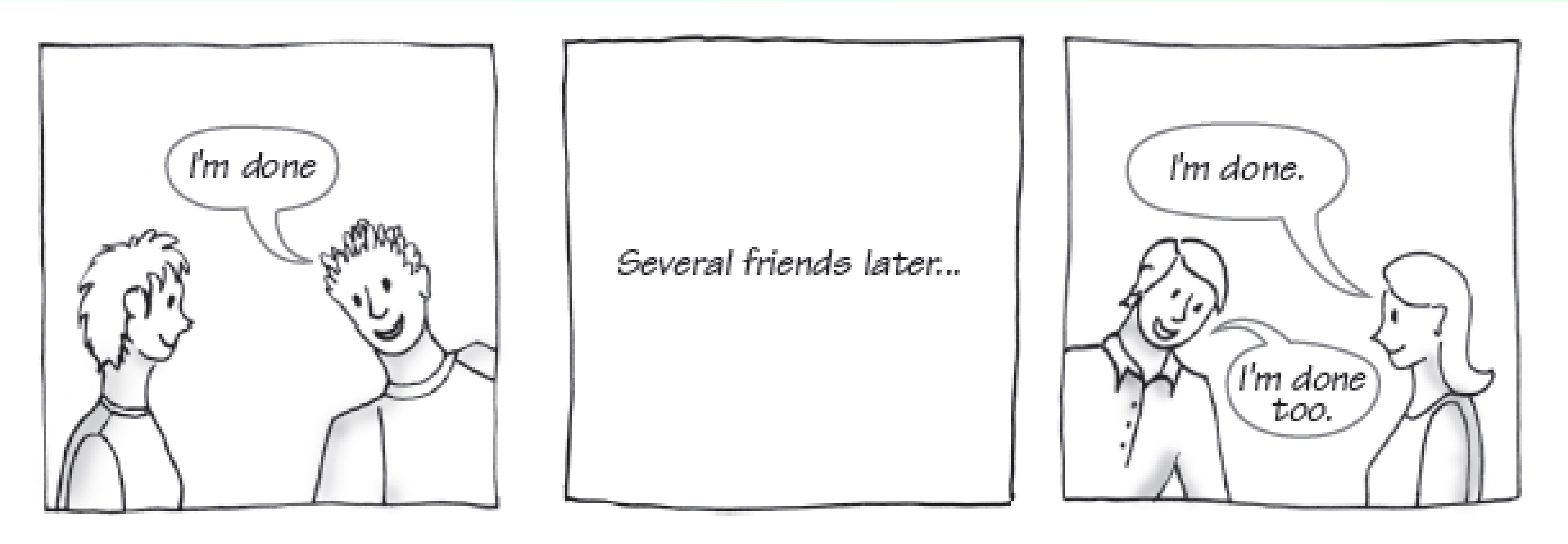


Figure 7-1 Counting down from 10

What Is Recursion

- A method that calls itself is
  - A recursive method.
- The invocation is
  - A recursive call or
  - Recursive invocation
- Example:
  - Countdown

```java
/** Counts down from a given positive integer.
    @param integer  an integer > 0 */
public static void countDown(int integer)
{
   System.out.println(integer);
   if (integer > 1)
      countDown(integer - 1);
} // end countDown
```

Design of Recursive Solution

- What part of solution can contribute directly?
- What smaller (identical) problem has solution that …
  - When taken with your contribution
  - Provides the solution to the original problem
- When does process end?
  - What smaller but identical problem has known solution
  - Have you reached this problem, or base case?

Design Guidelines

- Method must receive input value
- Must contain logic that involves this input value and leads to different cases
- One or more cases should provide solution that does not require recursion
  - Base case or stopping case
- One or more cases must include recursive invocation of method

Question 1 Write a recursive void method that skips n lines of output, where n is a positive integer. Use `System.out.println()` to skip one line.

Question 2 Describe a recursive algorithm that draws a given number of concentric circles. The innermost circle should have a given diameter. The diameter of each of the other circles should be four-thirds the diameter of the circle just inside it.

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

2.
```
Algorithm drawConcentricCircles(givenNumber, givenDiameter, givenPoint)
if (givenNumber >= 1)
{
   Draw a circle whose diameter is givenDiameter and whose center is at givenPoint
   givenDiameter = 4 * givenDiameter / 3
   drawConcentricCircles(givenNumber - 1, givenDiameter, givenPoint)
}
```

Tracing Recursive Method

- Given recursive **`countDown`** method

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

- Figure 7-2 The effect of the method call **`countDown(3)`**

(a)	(b)	(c)
`countDown(3)`	`countDown(2)`	`countDown(1)`
Display 3 Call `countDown(2)`	Display 2 Call `countDown(1)`	Display 1

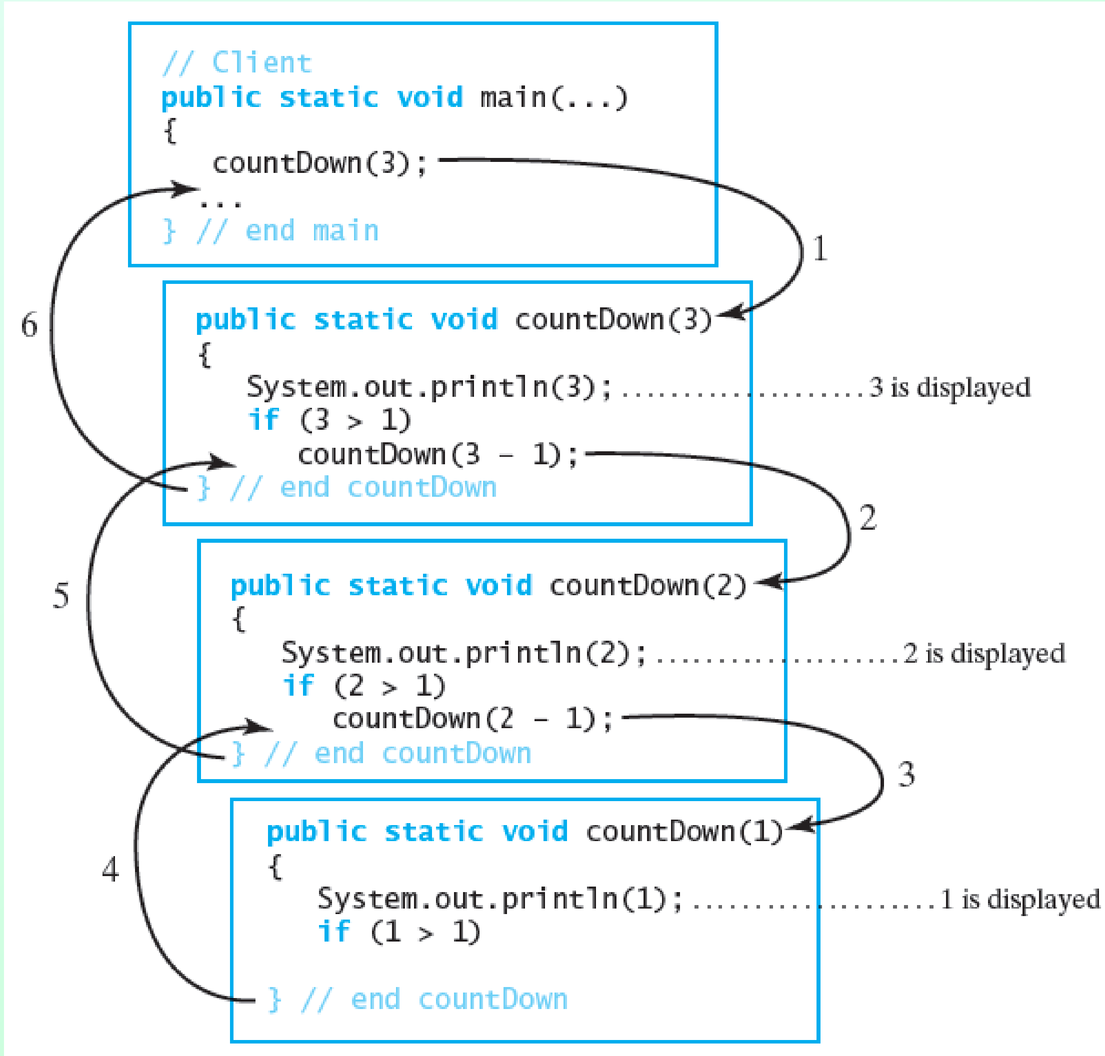


Figure 7-3 Tracing the recursive call **`countDown(3)`**

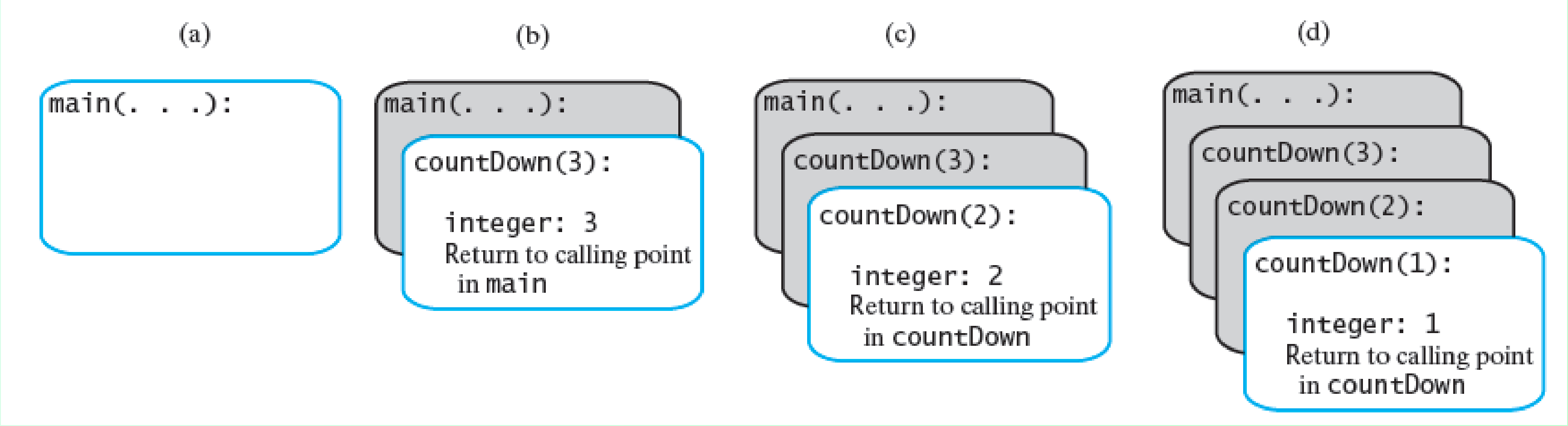


FIGURE 7-4 The stack of activation records during the execution of the call **`countDown(3)`**

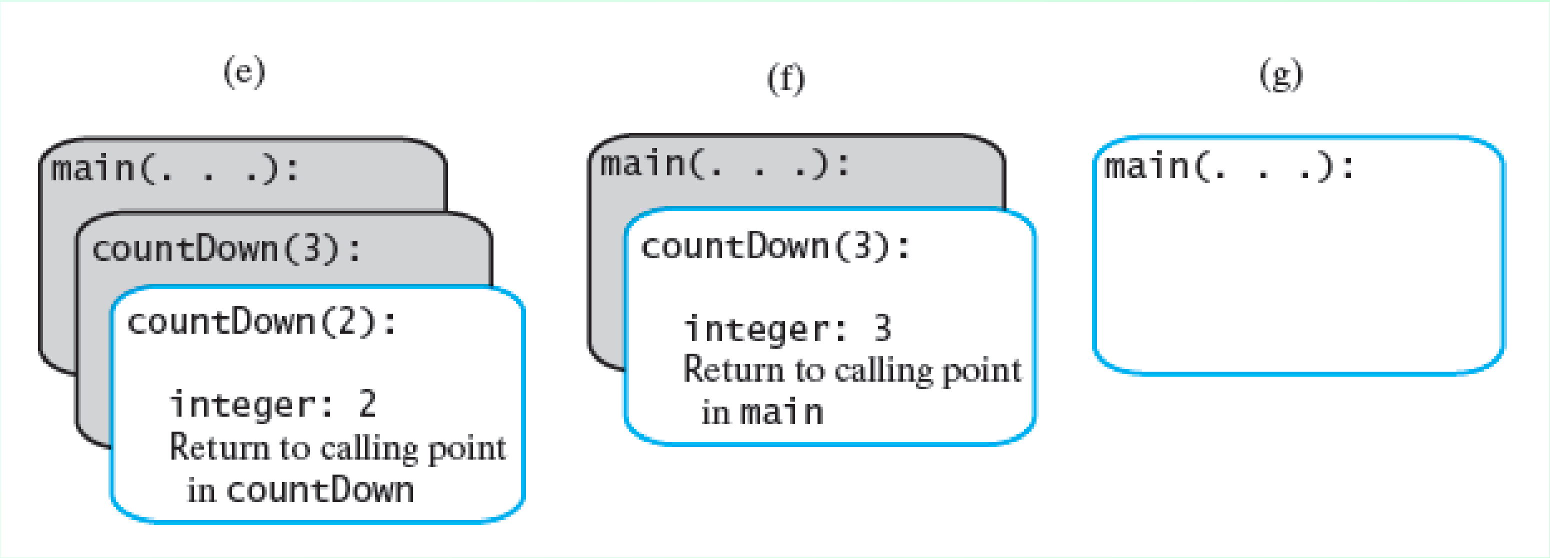


FIGURE 7-4 The stack of activation records during the execution of the call **`countDown(3)`**

Question 3 Write a recursive void method `countUp(n)` that counts up from 1 to n, where n is a positive integer. *Hint*: A recursive call will occur before you display anything.

3.

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

Recursive Methods That Return a Value

- Example:
  - Compute the sum 1 + 2 + . . . + n
  - For any integer n > 0.

```
/** @param n  an integer > 0
    @return the sum 1 + 2 + ... + n */
public static int sumOf(int n)
{
   int sum;
   if (n == 1)
      sum = 1;                    // base case
   else
      sum = sumOf(n - 1) + n; // recursive call

   return sum;
} // end sumOf
```

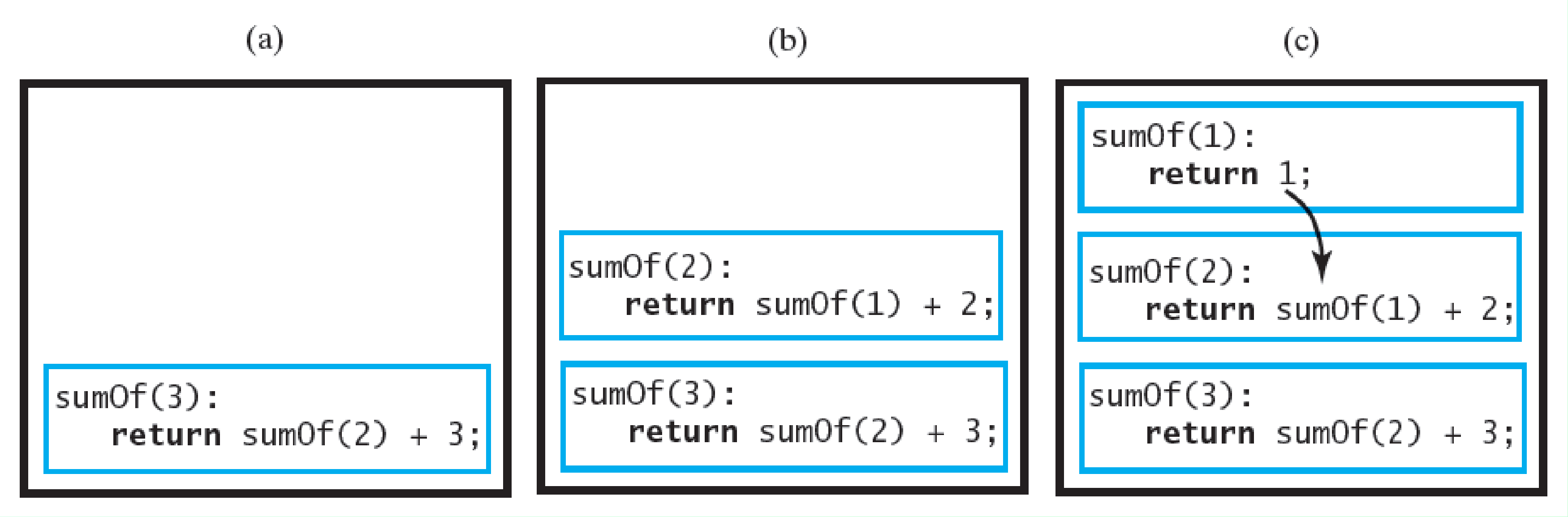


Figure 7-5 Tracing the execution of **`sumOf(3)`**

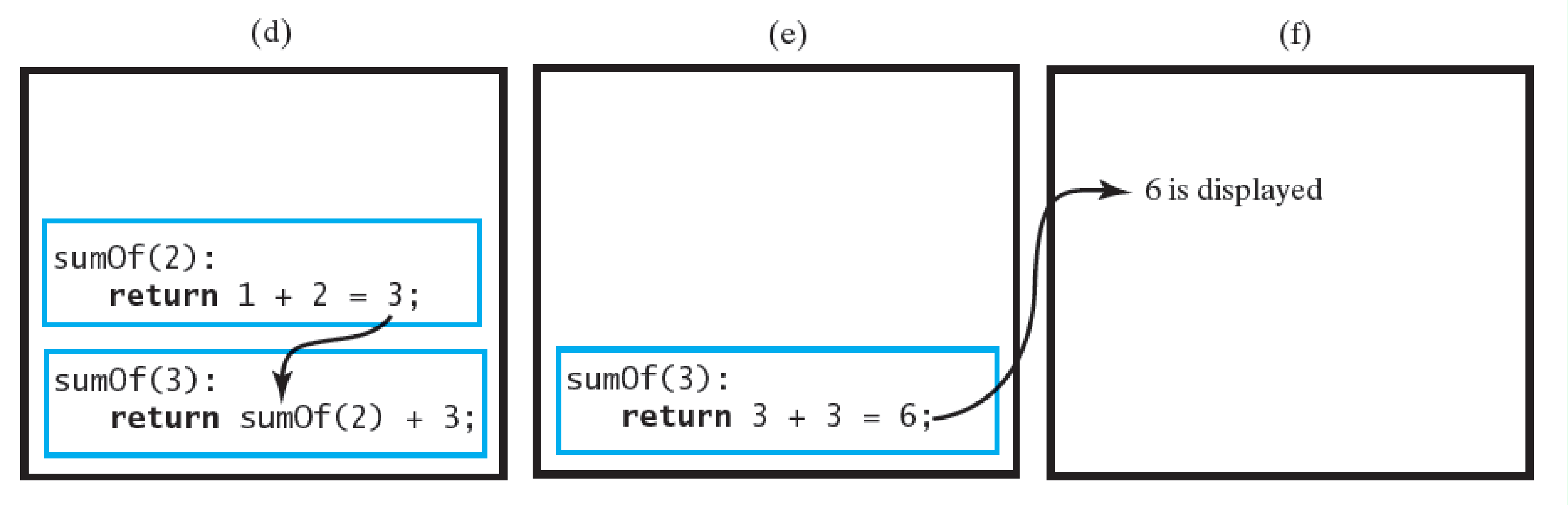


Figure 7-5 Tracing the execution of **`sumOf(3)`**

Question 4 Write a recursive valued method that computes the product of the integers from 1 to n, where $n > 0$.

4.
```java
public static int productOf(int n)
{
    int result = 1;
    if (n > 1)
        result = n * productOf(n - 1);
    return result;
} // end productOf
```

Recursively Processing an Array

- Consider an array of integers
- We seek a method to display all or part
- Declaration

```
public static void displayArray
(int[] array, int first, int last)
```

- Solution could be
  - Iterative
  - Recursive

Recursively Processing an Array

- Recursive solution starting with `array[first]`

```
public static void displayArray(int array[], int first, int last)
{
   System.out.print(array[first] + " ");
   if (first < last)
      displayArray(array, first + 1, last);
} // end displayArray
```

- Recursive solution starting with `array[last]`

```
public static void displayArray(int array[], int first, int last)
{
   if (first <= last)
   {
      displayArray(array, first, last - 1);
      System.out.print (array[last] + " ");
   } // end if
} // end displayArray
```

Dividing an Array in Half

- Common way to process arrays recursively
  - Divide array into two portions
  - Process each portion separately
- Must find element at or near middle

```
int mid = (first + last) / 2;
```

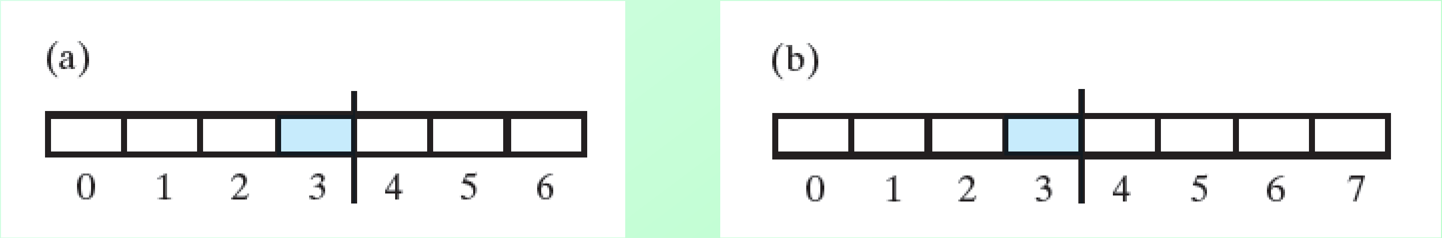


FIGURE 7-6 Two arrays with their middle elements within their left halves

Dividing an Array in Half

- Recursive method to display array
  - Divides array into two portions

```java
public static void displayArray(int array[], int first, int last)
{
   if (first == last)
      System.out.print(array[first] + " ");
   else
   {
      int mid = (first + last) / 2;
      displayArray(array, first, mid);
      displayArray(array, mid + 1, last);
   } // end if
} // end displayArray
```

Binary Search of a Sorted Array

- Algorithm for binary search

```
Algorithm binarySearch(a, first, last, desiredItem)
mid = (first + last) / 2 // approximate midpoint
if (first > last)
   return false
else if (desiredItem equals a[mid])
   return true
else if (desiredItem < a[mid])
   return binarySearch(a, first, mid - 1, desiredItem)
else // desiredItem > a[mid]
   return binarySearch(a, mid + 1, last, desiredItem)
```

(a) A search for 8

Look at the middle entry, 10:

2	4	5	7	8	**10**	12	15	18	21	24	26
0	1	2	3	4	5	6	7	8	9	10	11

$8 < 10$, so search the left half of the array.

Look at the middle entry, 5:

2	4	**5**	7	8
0	1	2	3	4

$8 > 5$, so search the right half of the array.

Look at the middle entry, 7:

7	8
3	4

$8 > 7$, so search the right half of the array.

Look at the middle entry, 8:

8
4

$8 = 8$, so the search ends. 8 is in the array.

Figure 18-6 A recursive binary search of a sorted array that (a) finds its target;

(b) A search for 16

Look at the middle entry, 10:

2	4	5	7	8	**10**	12	15	18	21	24	26
0	1	2	3	4	5	6	7	8	9	10	11

16 > 10, so search the right half of the array.

Look at the middle entry, 18:

12	15	**18**	21	24	26
6	7	8	9	10	11

16 < 18, so search the left half of the array.

Look at the middle entry, 12:

12	15
6	7

16 > 12, so search the right half of the array.

Look at the middle entry, 15:

15
7

16 > 15, so search the right half of the array.

The next subarray is empty, so the search ends. 16 is not in the array.

Figure 18-6 A recursive binary search of a sorted array that (b) does not find its target

Efficiency of a Binary Search of an Array

- Given *n* elements to be searched
- Number of recursive calls is of order

$$\log_2 n$$

The time efficiency of a binary search	
Best case:	$O(1)$
Worst case:	$O(\log n)$
Average case:	$O(\log n)$

- How many compares to search for an item in an array of 1 million items?

Java Class Library: The Method **binarySearch**

- Class Arrays contains versions of static method
  - Note specification

```
/** Searches an entire array for a given item.
    @param array        an array sorted in ascending order
    @param desiredItem  the item to be found in the array
    @return index of the array entry that equals desiredItem;
            otherwise returns -belongsAt - 1, where belongsAt is
            the index of the array element that should contain
            desiredItem */
public static int binarySearch(type[] array, type desiredItem);
```

Merge Sort

- Divide array into two halves
  - Sort the two halves
  - Merge them into one sorted array
- Uses strategy of “divide and conquer”
  - Divide problem up into two or more distinct, smaller tasks
- Good application for recursion

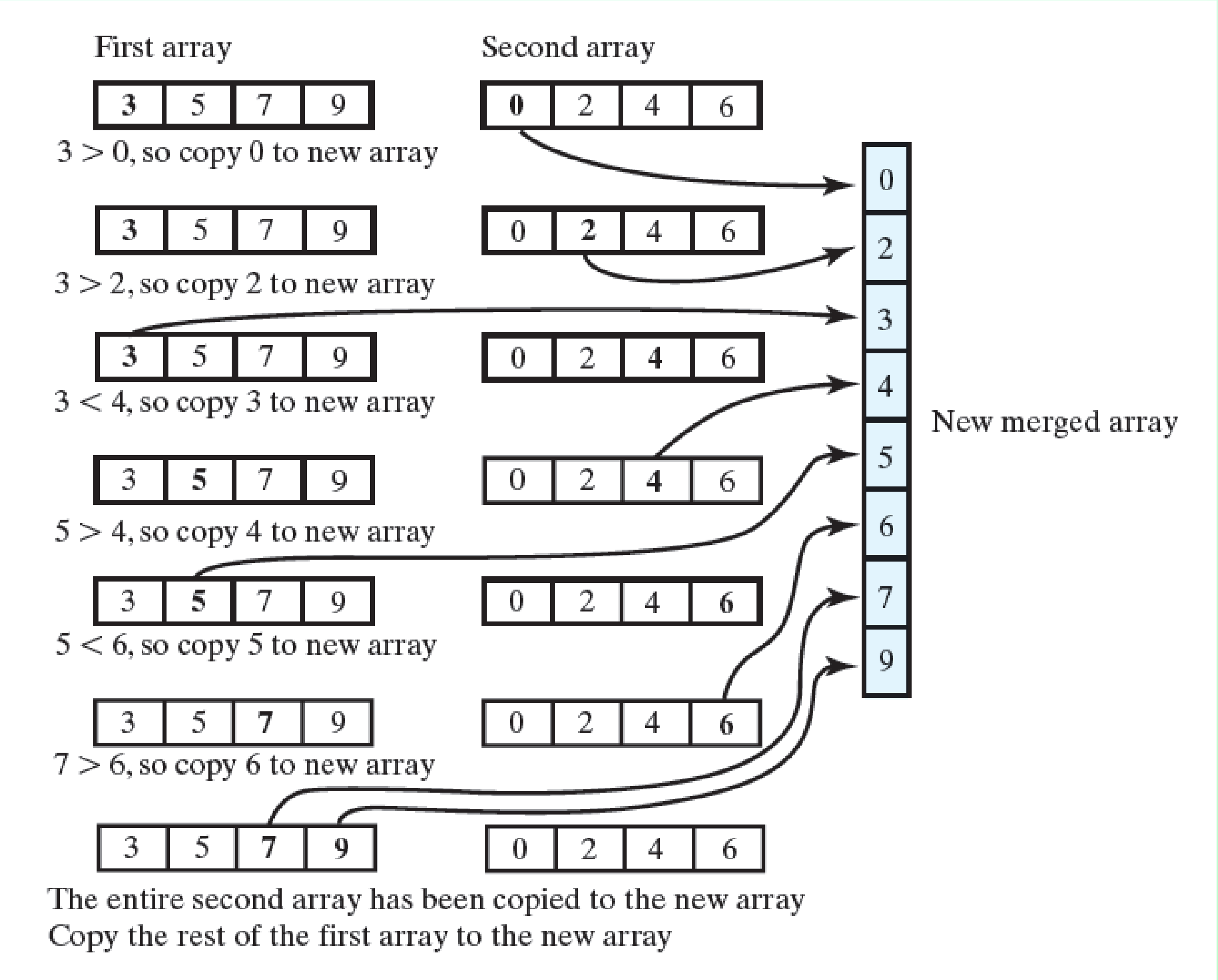


Figure 9-1 Merging two sorted arrays into one sorted array

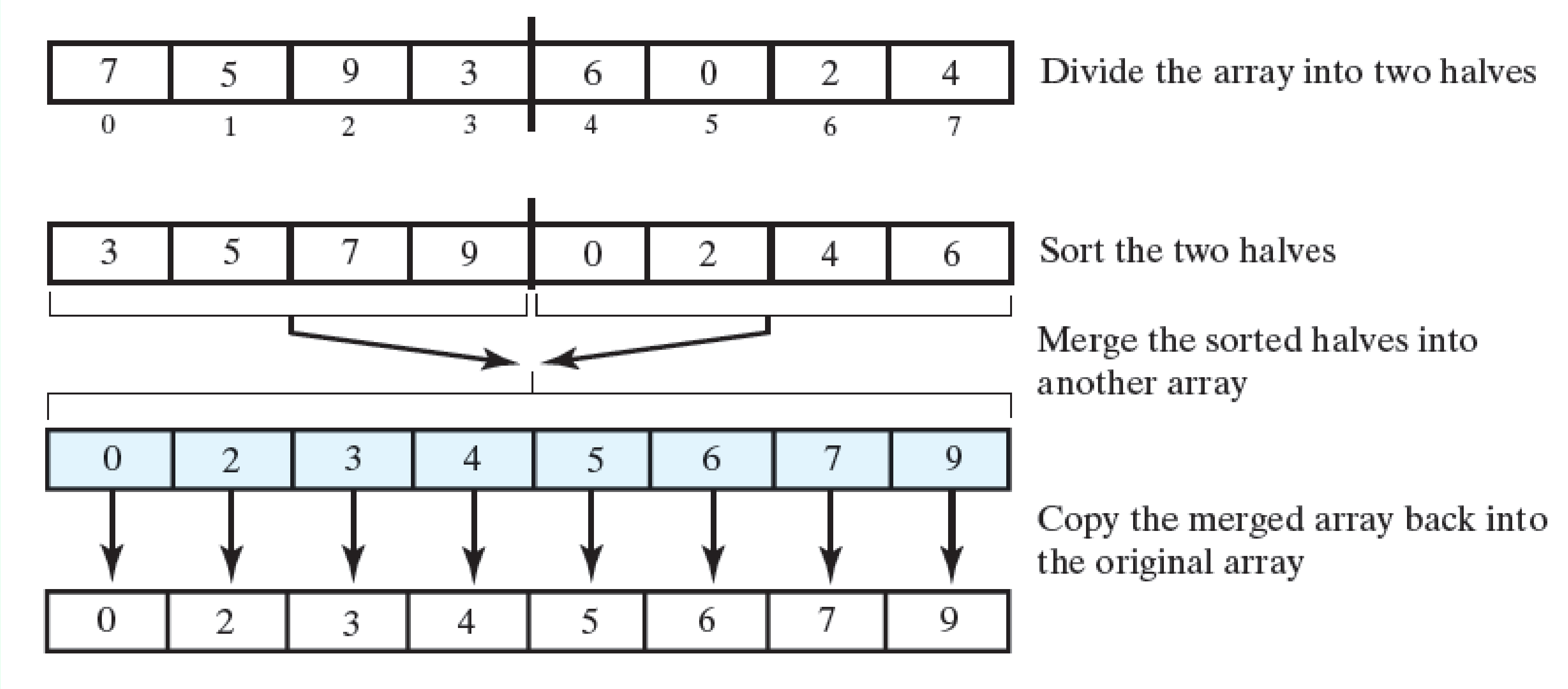


Figure 9-2 The major steps in a merge sort

```
Algorithm mergeSort(a, tempArray, first, last)
// Sorts the array entries a[first] through a[last] recursively.

if (first < last)
{
   mid = (first + last) / 2
   mergeSort(a, tempArray, first, mid)
   mergeSort(a, tempArray, mid + 1, last)
   Merge the sorted halves a[first..mid] and a[mid + 1..last] using the array tempArray
}
```

Merge Sort Algorithm

Algorithm to Merge

```
Algorithm merge(a, tempArray, first, mid, last)
// Merges the adjacent subarrays a[first..mid] and a[mid + 1..last].

beginHalf1 = first
endHalf1 = mid
beginHalf2 = mid + 1
endHalf2 = last

// While both subarrays are not empty, compare an entry in one subarray with
// an entry in the other; then copy the smaller item into the temporary array
index = 0 // next available location in tempArray
while ( (beginHalf1 <= endHalf1) and (beginHalf2 <= endHalf2) )
{
   if (a[beginHalf1] <= a[beginHalf2])
   {
      tempArray[index] = a[beginHalf1]
      beginHalf1++
   }
   else
   {
      tempArray[index] = a[beginHalf2]
      beginHalf2++
   }
   index++
}
// Assertion: One subarray has been completely copied to tempArray.

Copy remaining entries from other subarray to tempArray
Copy entries from tempArray to array a
```

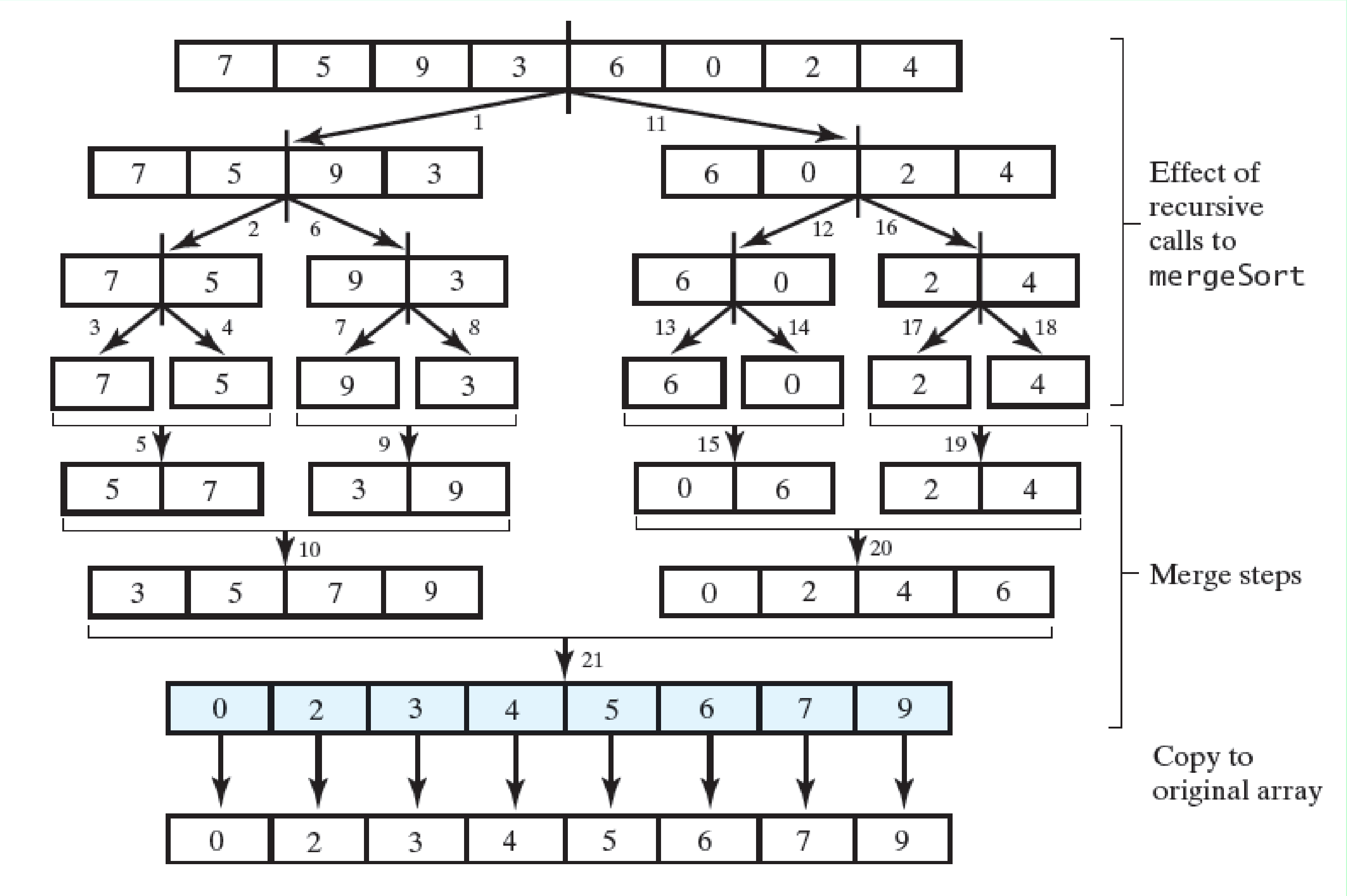


Figure 9-3 The effect of the recursive calls and the merges during a merge sort

Time Efficiency of Recursive Methods

- Consider method **`countdown`**

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

- Recurrence relation $t(n) = 1 + t(n - 1)$ for $n > 1$
- Proof by induction shows $t(n) = n$
- Thus method is O(n)

Question 7 What is the Big Oh of the method `sumOf` given in Segment 7.12?

Question 8 Computing x^n for some real number x and an integral power $n \geq 0$ has a simple recursive solution:

$$x^n = x\, x^{n-1}$$
$$x^0 = 1$$

a. What recurrence relation describes this algorithm's time requirement?

b. By solving this recurrence relation, find the Big Oh of this algorithm.

7. $O(n)$. You can use the same recurrence relation that was shown in Segments 7.22 and 7.23 for the method `countDown`.

8. **a.** $t(n) = 1 + t(n - 1)$ for $n > 0$, $t(0) = 1$.
b. Since $t(n) = n + 1$, the algorithm is $O(n)$.

Efficiency of Merge Sort

- For $n = 2^k$ entries
  - In general k levels of recursive calls are made
- Each merge requires at most $3n - 1$ comparisons
- Calls to merge do at most $3n - 2^2$ operations
- Can be shown that efficiency is O(n log n)

Simple Solution to a Difficult Problem

- Consider Towers of Hanoi puzzle

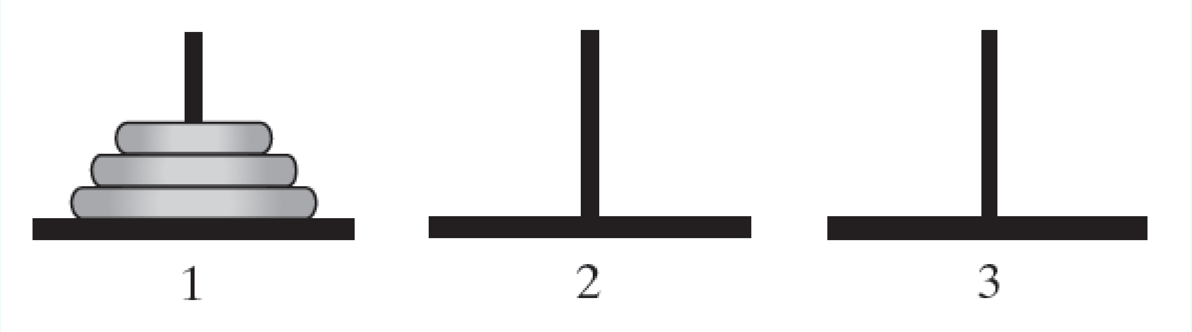


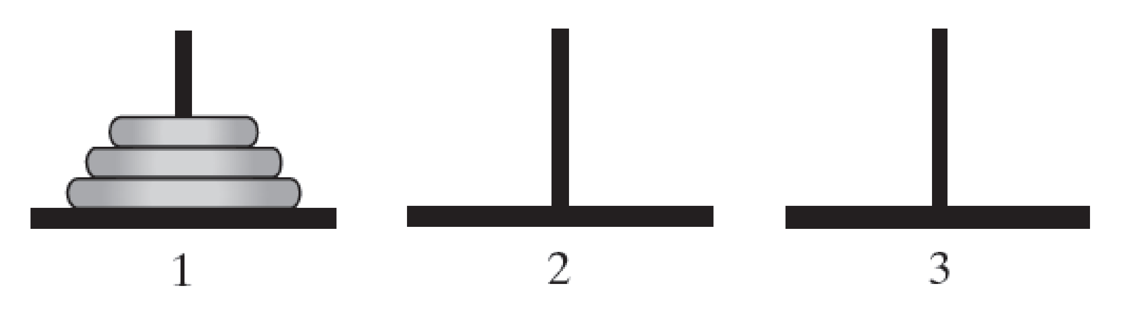
Figure 7-7 The initial configuration of the Towers of Hanoi for three disks.

Towers of Hanoi

- Rules
  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.

Solution

- Move a disk from pole 1 to pole 3
- Move a disk from pole 1 to pole 2
- Move a disk from pole 3 to pole 2
- Move a disk from pole 1 to pole 3
- Move a disk from pole 2 to pole 1
- Move a disk from pole 2 to pole 3
- Move a disk from pole 1 to pole 3



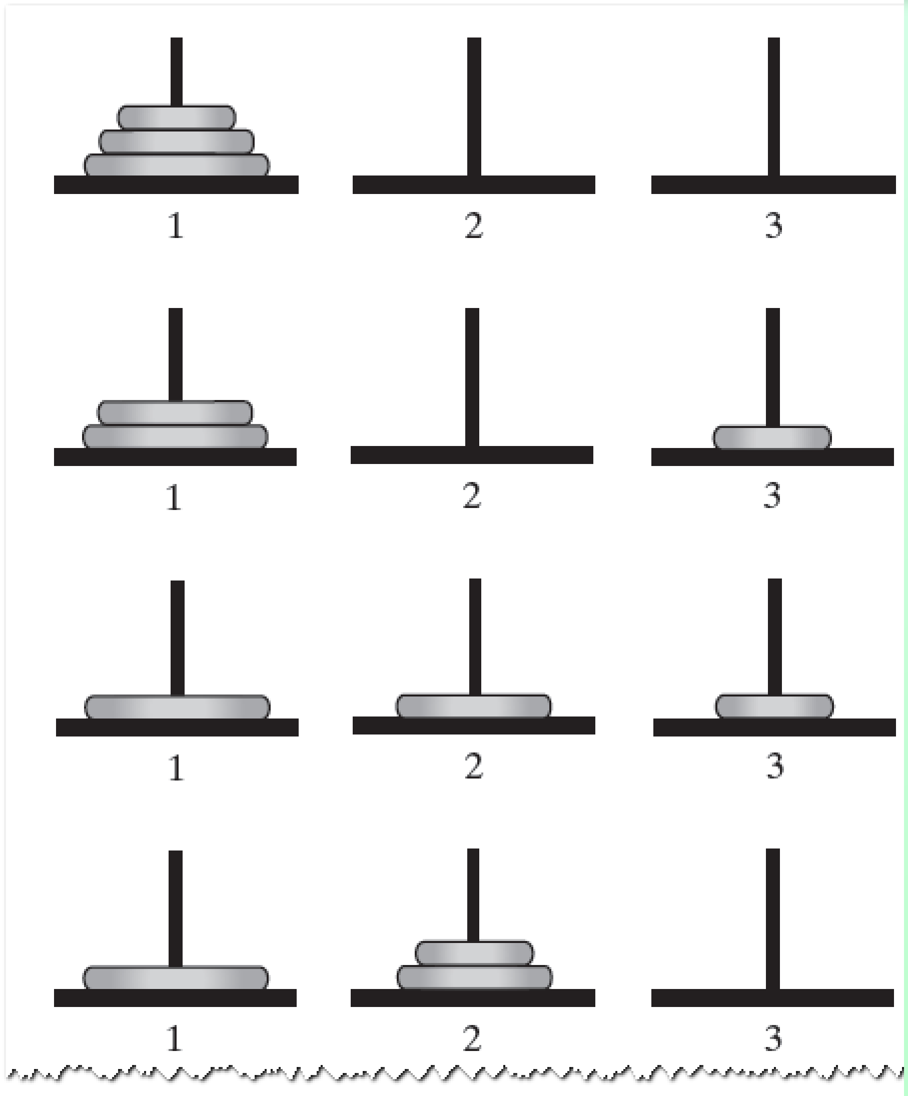


Figure 7-8 The sequence of moves for solving the Towers of Hanoi problem with three disks

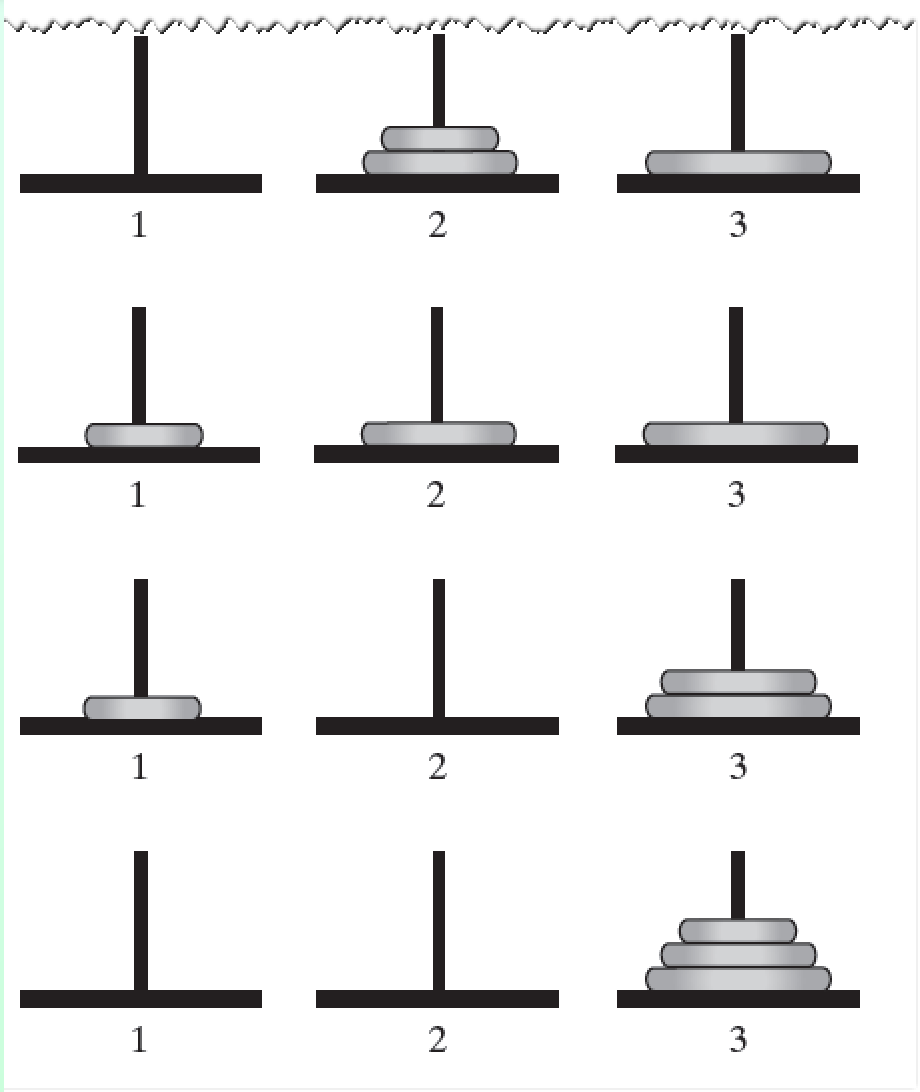


Figure 7-8 The sequence of moves for solving the Towers of Hanoi problem with three disks

Question 9 We discovered the previous solution for three disks by trial and error. Using the same approach, find a sequence of moves that solves the problem for four disks.

9. Move a disk from pole 1 to pole 2
Move a disk from pole 1 to pole 3
Move a disk from pole 2 to pole 3
Move a disk from pole 1 to pole 2
Move a disk from pole 3 to pole 1
Move a disk from pole 3 to pole 2
Move a disk from pole 1 to pole 2
Move a disk from pole 1 to pole 3
Move a disk from pole 2 to pole 3
Move a disk from pole 2 to pole 1
Move a disk from pole 3 to pole 1
Move a disk from pole 2 to pole 3
Move a disk from pole 1 to pole 2
Move a disk from pole 1 to pole 3
Move a disk from pole 2 to pole 3

Recursive Solution

- To solve for *n* disks …
  - Ask friend to solve for *n – 1* disks
  - He in turn asks another friend to solve for *n – 2*
  - Etc.
  - Each one lets previous friend know when their simpler task is finished

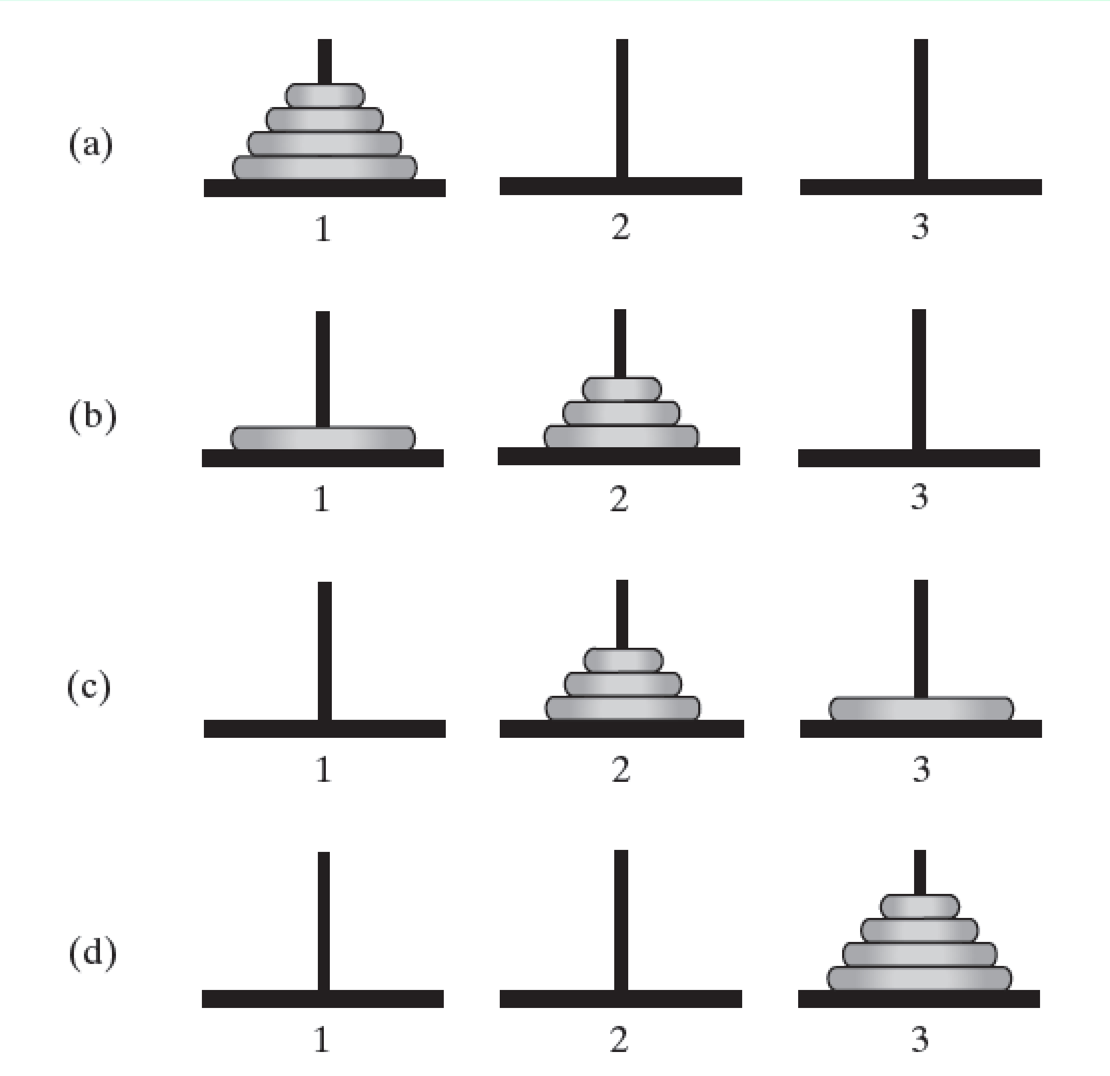


Figure 7-9 The smaller problems in a recursive solution for four disks

Recursive Algorithm, VER1

```
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)
}
```

Recursive Algorithm, VER2

```
Algorithm solveTowers(numberOfDisks, startPole, tempPole, endPole)
// Version 2
if (numberOfDisks > 0)
{
   solveTowers(numberOfDisks - 1, startPole, endPole, tempPole)
   Move disk from startPole to endPole
   solveTowers(numberOfDisks - 1, tempPole, startPole, endPole)
}
```

Question 10 For two disks, how many recursive calls are made by each version of the algorithm just given?

10. 2 and 6, respectively.

Algorithm Efficiency

- Moves required for *n* disks

$$m(n) = m(n-1) + 1 + m(n-1)$$
$$= 2\,m(n-1) + 1$$

- We note $\left\{\begin{array}{l} m(1) = 1 \\ m(2) = 3 \\ m(3) = 7 \\ m(4) = 15 \\ m(5) = 31 \\ m(6) = 63 \end{array}\right.$ and conjecture

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

(proved by induction)

Poor Solution to a Simple Problem

- Fibonacci sequence
  1, 1, 2, 3, 5, 8, 13, …

$$F_0 = 1$$
$$F_1 = 1$$
$$F_n = F_{n-1} + F_{n-2} \text{ when } n \geq 2$$

- Suggests a recursive solution

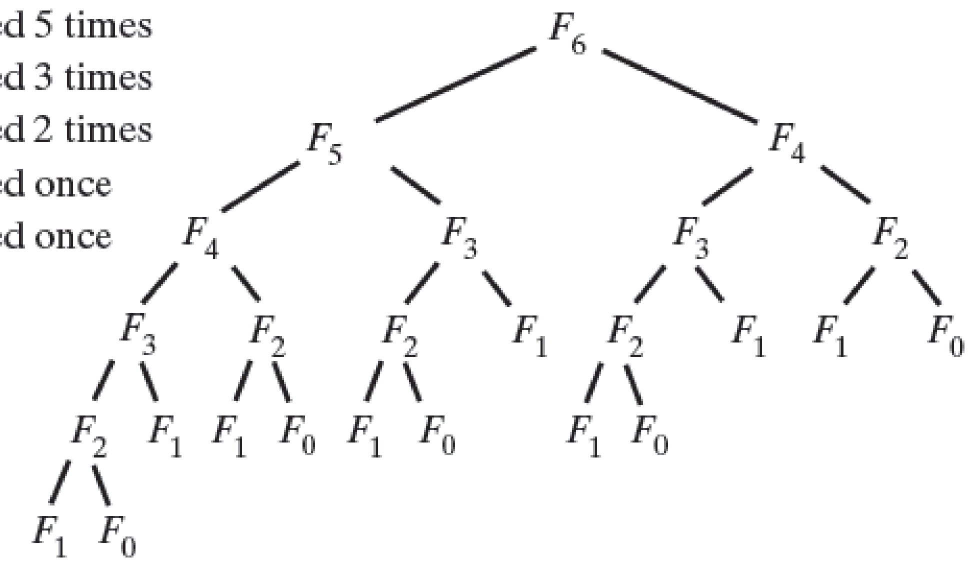
Note the *two* recursive calls

Algorithm Fibonacci(n)

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

(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



(b)

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

FIGURE 7-10 The computation of the Fibonacci number F6 using (a) recursion; (b) iteration

Time Efficiency of Algorithm

- Looking for relationship

$$t(2) = 1 + t(1) + t(0) = 1 + F_1 + F_0 = 1 + F_2 > F_2$$
$$t(3) = 1 + t(2) + t(1) > 1 + F_2 + F_1 = 1 + F_3 > F_3$$
$$t(4) = 1 + t(3) + t(2) > 1 + F_3 + F_2 = 1 + F_4 > F_4$$

- Can be shown that $t(n) > F_n$ for $n \geq 2$
- Conclusion: Do not use recursive solution that repeatedly solves same problem in its recursive calls.

Question 11 If you compute the Fibonnaci number F_6 recursively, how many recursive calls are made, and how many additions are performed?

Question 12 If you compute the Fibonnaci number F_6 iteratively, how many additions are performed?

11. 24 recursive calls and 12 additions.

12. 5 additions.

Tail Recursion

- When the last action performed by a recursive method is a recursive call
- Example:

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

- Repeats call with change in parameter or variable

Tail Recursion

- Consider this simple change to make an iterative version

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

  - Replace **`if`** with **`while`**
  - Instead of recursive call, subtract 1 from **`integer`**
- Change of tail recursion to iterative often simple

Indirect Recursion

- Consider chain of events
  - Method A calls Method B
  - Method B calls Method C
  - and Method C calls Method A
- Mutual recursion
  - Method A calls Method B
  - Method B calls Method A

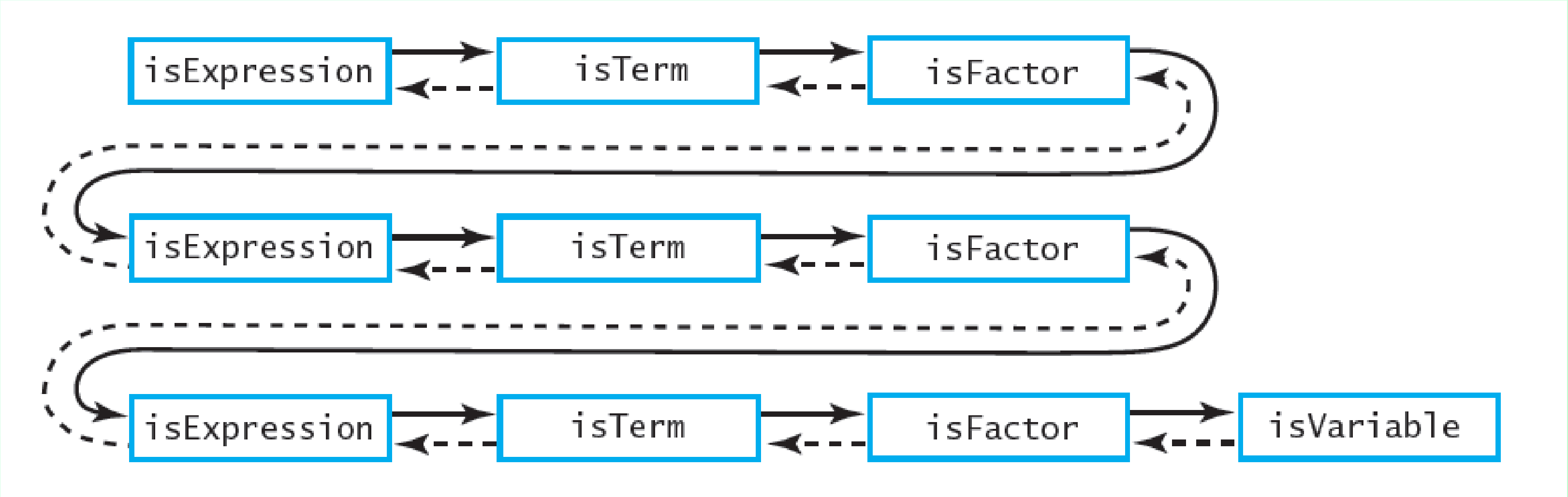


Figure 7-11 An example of indirect recursion

Using a Stack Instead of Recursion

- A way of replacing recursion with iteration
- Consider recursive **`displayArray`**

```
public void displayArray(int first, int last)
{
   if (first == last)
      System.out.println(array[first] + " ");
   else
   {
      int mid = first + (last - first) / 2; // improved calculation of
                                            // midpoint
      displayArray(first, mid);
      displayArray(mid + 1, last);
   } // end if
} // end displayArray
```

Using a Stack Instead of Recursion

- We make a stack that mimics the program stack
  - Push objects onto stack like activation records
  - Shown is example record

```
private class Record
{
   private int first, last;

   private Record(int firstIndex, int lastIndex)
   {
      first = firstIndex;
      last = lastIndex;
   } // end constructor
} // end Record
```

Using a Stack Instead of Recursion

- Iterative version of **`displayArray`**

```
private void displayArray(int first, int last)
{
   boolean done = false;
   StackInterface<Record> programStack = new LinkedStack<Record>();
   programStack.push(new Record(first, last));
   while (!done && !programStack.isEmpty())
   {
      Record topRecord = programStack.pop();
      first = topRecord.first;
      last = topRecord.last;
      if (first == last)
         System.out.println(array[first] + " ");
      else
      {
         int mid = first + (last - first) / 2;
         // Note the order of the records pushed onto the stack
         programStack.push(new Record(mid + 1, last));
         programStack.push(new Record(first, mid));
      } // end if
   } // end while
} // end displayArray
```

Recursion

Chapter 7