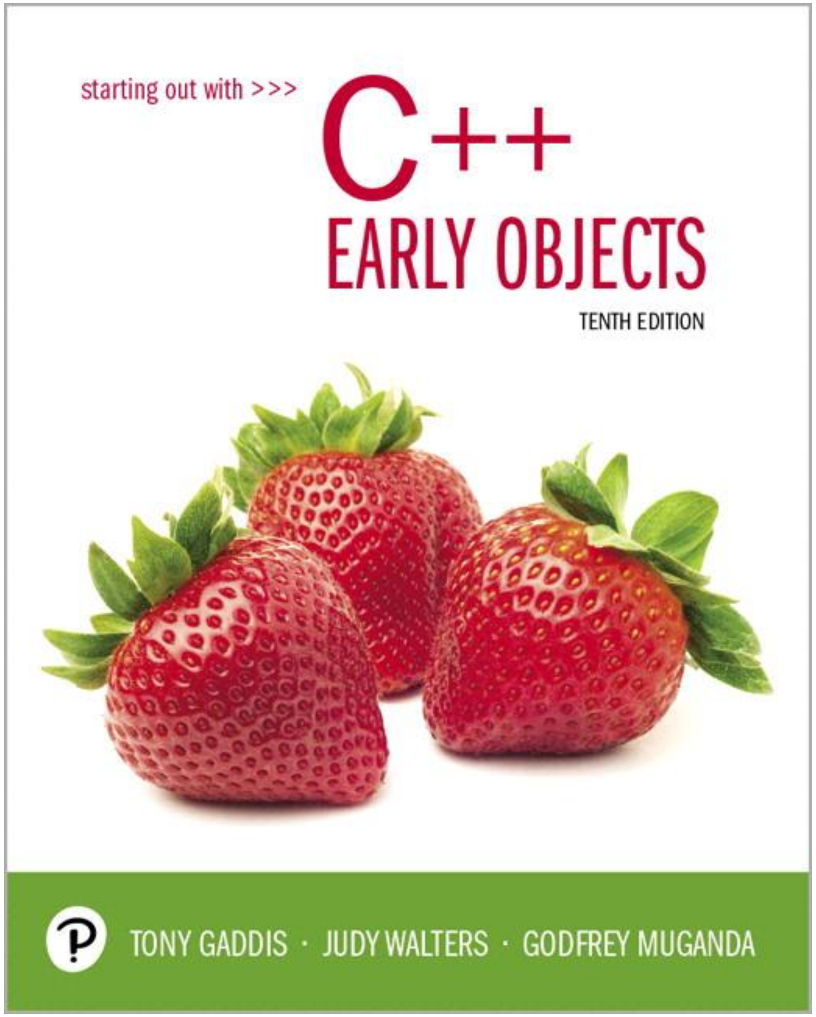
Starting Out with C++ Early Objects

Tenth Edition



Chapter 14

Recursion

Topics

14.1 Introduction to Recursion

14.2 The Recursive Factorial Function

14.3 The Recursive gcd Function

14.4 Solving Recursively Defined Problems

14.5 A Recursive Binary Search Function

14.6 The QuickSort Algorithm

14.7 The Towers of Hanoi

14.8 Exhaustive and Enumeration Algorithms

14.9 Recursion Versus Iteration

14.1 Introduction to Recursion

- A recursive function is a function that calls itself.
- Recursive functions can be useful in solving problems that can be broken down into smaller or simpler subproblems of the same type. A base case should eventually be reached, at which time the breaking down (recursion) will stop.

Recursive Functions 1 of 2

Consider a function for solving the count-down problem from some number **`num`** down to `0`:

- The base case is when **`num`** is already `0`: the problem is solved and we "blast off!"
- If **`num`** is greater than `0`, we count off **`num`** and then recursively count down from **`num-1`**

Recursive Functions 2 of 2

A recursive function for counting down to 0:

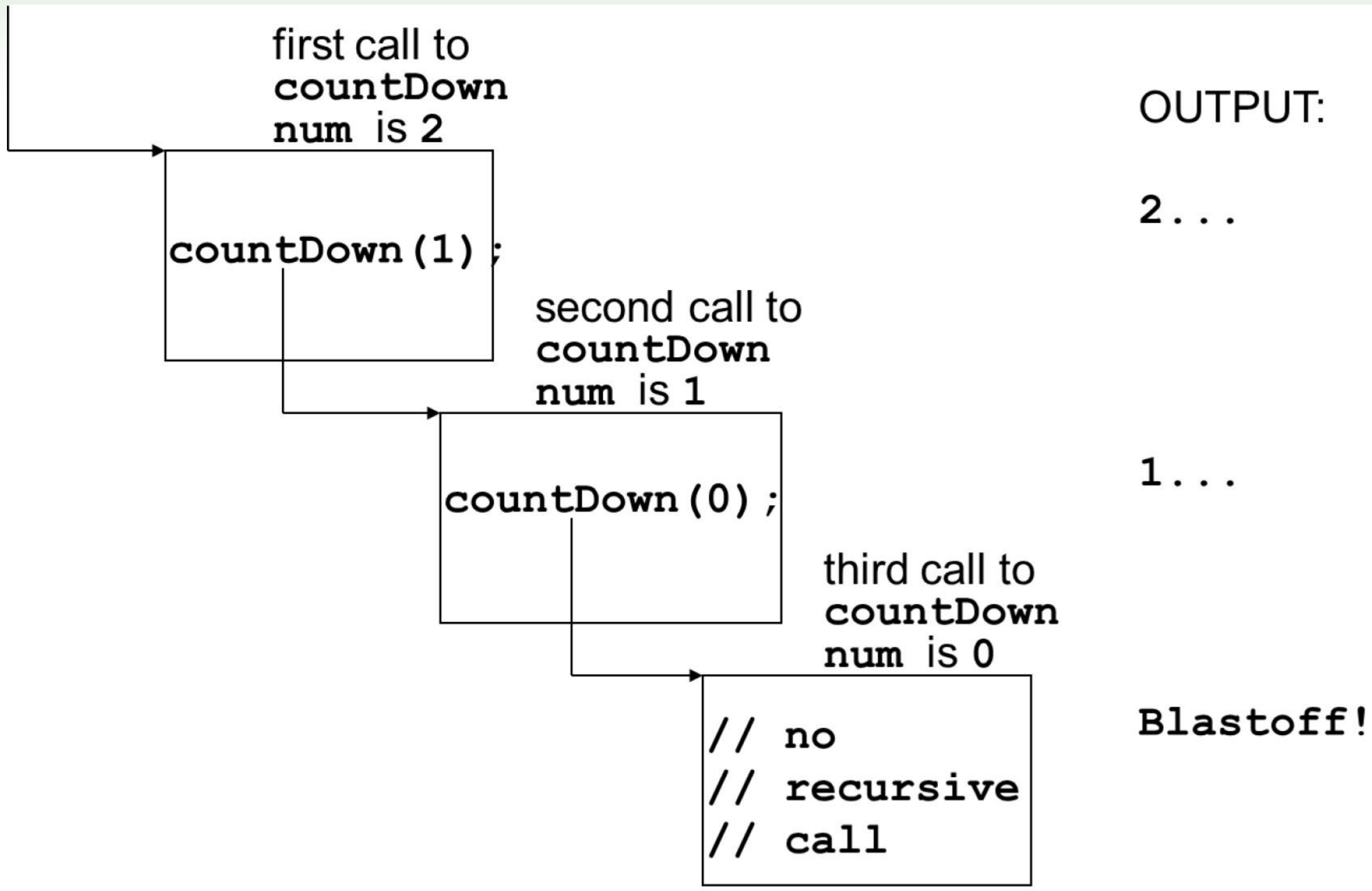
```
void countDown(int num)
{
  if (num == 0)
        cout << "Blast off!";
    else
    {
        cout << num << ". . .";
        countDown(num-1); // recursive
    }                     // call
}
```

What Happens When Called? 1 of 2

If a program contains a line like **`countDown(2);`**

1. **`countDown(2)`** generates the output **`2...`**, then it calls **`countDown(1)`**
2. **`countDown(1)`** generates the output **`1...`**, then it calls **`countDown(0)`**
3. **`countDown(0)`** generates the output **`Blast off!`**, then returns to **`countDown(1)`**
4. **`countDown(1)`** returns to **`countDown(2)`**
5. **`countDown(2)`** returns to the calling function

What Happens When Called? 2 of 2



Stopping the Recursion 1 of 4

- A recursive function should include a test for the base cases
- In the sample program, the test is:

```
if (num == 0)
```

Stopping the Recursion 2 of 4

```
void countDown(int num)
{
  if (num == 0) // test
     cout << "Blast off!";
  else
  {
     cout << num << "...\n";
     countDown(num-1); // recursive
  }                    // call
}
```

Stopping the Recursion 3 of 4

- With each recursive call, the parameter controlling the recursion should move closer to the base case
- Eventually, the parameter reaches the base case and the chain of recursive calls terminates

Stopping the Recursion 4 of 4

```
void countDown(int num)
{
    if (num == 0)       // base case
        cout << "Blast off!";
    else
    {   cout << num << "...\n";
        countDown(num-1); // decrement
        // parameter to approach base case
    }
}
```

Pearson

What Happens When Called?

- Each time a recursive function is called, a new copy of the function runs, with new instances of parameters and local variables being created
- As each copy finishes executing, it returns to the copy of the function that called it
- When the initial copy finishes executing, it returns to the part of the program that made the initial call to the function

Types of Recursion

- Direct recursion
  - a function calls itself
- Indirect recursion
  - function A calls function B, and function B calls function A. Or,
  - function A calls function B, which calls …, which then calls function A

14.2 The Recursive Factorial Function 1 of 2

- The factorial of a nonnegative integer ***n*** is the product of all positive integers less than or equal to ***n***
- The factorial of ***n*** is denoted by ***n*!**
- The factorial of 0 is 1

$$0\,! = 1$$
$$n\,! = n \times (n-1) \times \ldots \times 2 \times 1 \text{ if } n > 0$$

Recursive Factorial Function 2 of 2

- The factorial of ***n*** can be expressed in terms of the factorial of ***n*-1**

$$0\,! = 1$$
$$n\,! = n \times (n-1)\,!$$

- Recursive function:

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

Pearson

14.3 The Recursive gcd Function 1 of 2

- Greatest common divisor (gcd) of two integers x and y is the largest number that divides both x and y with no remainder.
- The Greek mathematician Euclid discovered that
  - If y divides x, then gcd(x, y) is just y
  - Otherwise, the gcd(x, y) is the gcd of y and the remainder of dividing x by y

The Recursive gcd Function 2 of 2

```
int gcd(int x, int y)
{
    if (x % y == 0) //base case
        return y;
    else
        return gcd(y, x % y);
}
```

14.4 Solving Recursively Defined Problems

- The natural definition of some problems leads to a recursive solution
- Example: Fibonacci numbers:

  `0, 1, 1, 2, 3, 5, 8, 13, 21, ...`
- After the initial **`0`** and **`1`**, each term is the sum of the two preceding terms
- Recursive calculation of the nth Fibonacci number:

  `fib(n) = fib(n – 1) + fib(n – 2);`
- Base cases: **`n == 0, n == 1`**

Pearson

Recursive Fibonacci Function

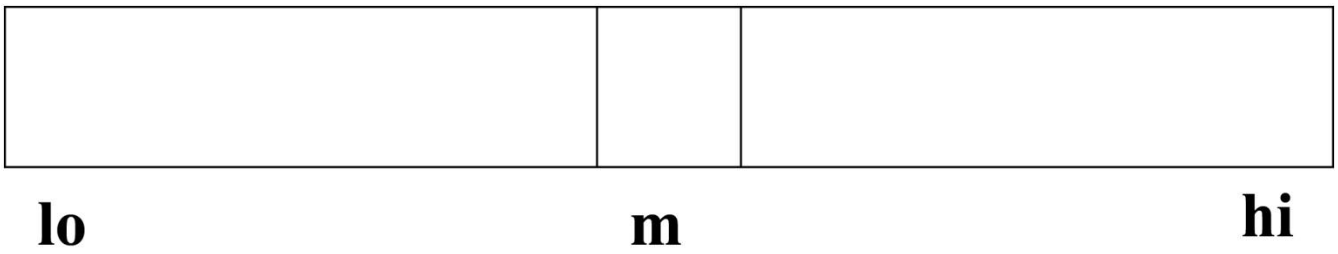
```
int fib(int n)
{
  if (n <= 0)           // base case
      return 0;
  else if (n == 1)      // base case
      return 1;
  else
       return fib(n – 1) + fib(n – 2);
}
```

14.5 A Recursive Binary Search Function

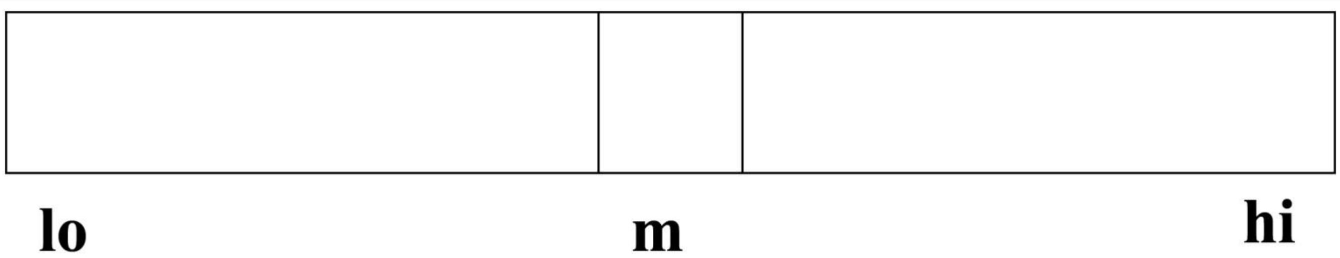
- Assume an array **`a`** that is sorted in ascending order, and an item **`x`** to search for
- We want to write a function that searches for **`x`** within the array **`a`**, returning the index of **`x`** if it is found, and returning **`-1`** if **`x`** is not in the array

Recursive Binary Search 1 of 3

A recursive strategy for searching a portion of the array from index `lo` to index `hi` is to set `m` to the index of the middle element of the array:



Recursive Binary Search 2 of 3



If `a[m] == x`, we found `x`, so return `m`

If `a[m] > x`, recursively search `a[lo..m-1]`

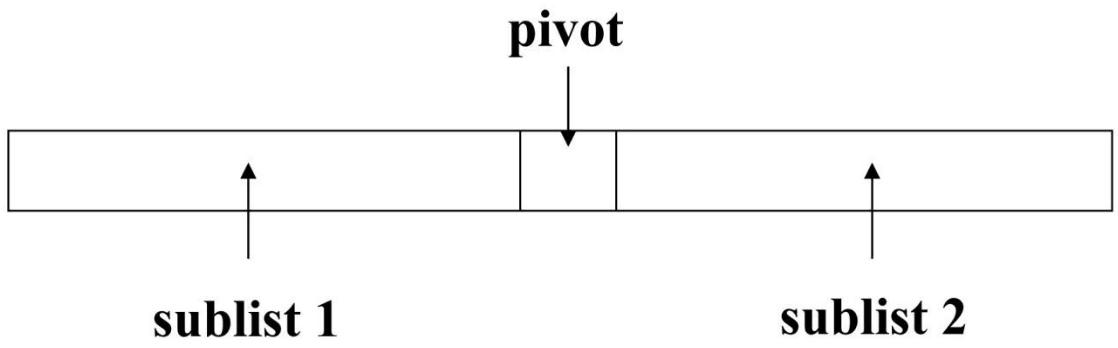
If `a[m] < x`, recursively search `a[m+1..hi]`

Recursive Binary Search 3 of 3

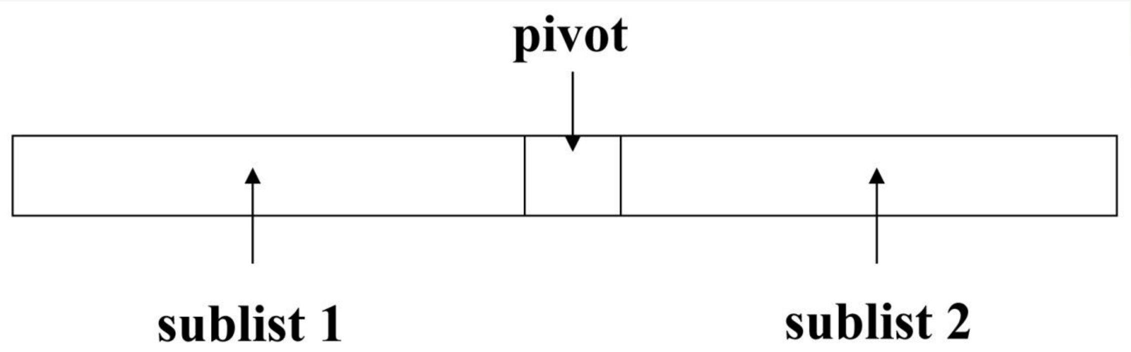
```
int bSearch(const int a[],int lo,
                          int hi,int X)
{
  int m = (lo + hi) /2;
  if(lo > hi) return -1;    // base
  if(a[m] == X) return m;   // base
  if(a[m] > X)
    return bsearch(a,lo,m-1,X);
  else
    return bsearch(a,m+1,hi,X);
}
```

14.6 The QuickSort Algorithm 1 of 2

- Recursive algorithm that can sort an array
- First, determine an element to use as pivot value:



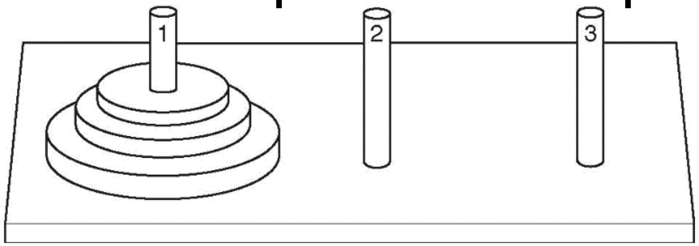
The QuickSort Algorithm 2 of 2



- Then, values are shifted so that elements in sublist1 are < pivot and elements in sublist2 are >= pivot
- Algorithm then recursively sorts sublist1 and sublist2
- Base case: a sublist has size <=1

14.7 The Towers of Hanoi 1 of 2

- Setup: 3 pegs, one has n disks on it, the other two pegs empty. The disks are arranged in increasing diameter, top→ bottom
- Objective: move the disks from peg 1 to peg 3, observing these rules:
  - only one disk moves at a time
  - all remain on pegs except the one being moved
  - a larger disk cannot be placed on top of a smaller disk at any time



The Towers of Hanoi 2 of 2

How it works:

Number of Disks	Sequence of Moves
n=1	Move disk from peg 1 to peg 3. Done.
n=2	Move top disk from peg 1 to peg 2. Move remaining disk from peg 1 to peg 3. Move disk from peg 2 to peg 3. Done.

Outline of the Recursive Algorithm

If n==0, do nothing (base case)

If n>0, then

a. Move the topmost n-1 disks from peg1 to peg2
b. Move the n^{th} disk from peg1 to peg3
c. Move the n-1 disks from peg2 to peg3

end if

14.8 Exhaustive and Enumeration Algorithms

- Enumeration algorithm: generate all possible combinations

  Example: all possible ways to make change for a certain amount of money

- Exhaustive algorithm: search a set of combinations to find an optimal one

  Example: change for a certain amount of money that uses the fewest coins

14.9 Recursion vs. Iteration

- Benefits (+), disadvantages(-) for recursion:
  - +Natural formulation of solution to certain problems
  - –May not execute very efficiently

- Benefits (+), disadvantages(-) for iteration:
  - +Executes more efficiently than recursion
  - –May not be as natural a method of solution as recursion for some problems

Copyright