Chapter Thirteen: Recursion
Chapter Goals

• To learn about the method of recursion
• To understand the relationship between recursion and iteration
• To analyze problems that are much easier to solve by recursion than by iteration
• To learn to "think recursively"
• To be able to use recursive helper methods
• To understand when the use of recursion affects the efficiency of an algorithm
Triangle Numbers

• Compute the area of a triangle of width $n$
• Assume each [] square has an area of 1
• Also called the $n$th triangle number
• The third triangle number is 6
public class Triangle
{
    public Triangle(int aWidth)
    {
        width = aWidth;
    }
    public int getArea()
    {
        ...
    }
    private int width;
}
Handling Triangle of Width 1

• The triangle consists of a single square
• Its area is 1
• Add the code to \texttt{getArea} method for width 1

\begin{verbatim}
public int getArea()
{
    if (width == 1) return 1;
    
    
}
\end{verbatim}
Handling the General Case

• Assume we know the area of the smaller, colored triangle

```
[]
[] []
[] [] []
[] [] [] []
[] [] [] [] []
```

• Area of larger triangle can be calculated as:
  smallerArea + width

• To get the area of the smaller triangle
  • *Make a smaller triangle and ask it for its area*

  ```java
  Triangle smallerTriangle = new Triangle(width - 1);
  int smallerArea = smallerTriangle.getArea();
  ```
Completed `getArea` Method

```java
public int getArea()
{
    if (width == 1) return 1;
    Triangle smallerTriangle = new Triangle(width - 1);
    int smallerArea = smallerTriangle.getArea();
    return smallerArea + width;
}
```
Computing the area of a triangle with width 4

- **getArea** method makes a smaller triangle of width 3
  - It calls **getArea** on that triangle
    - That method makes a smaller triangle of width 2
      - It calls **getArea** on that triangle
        - That method makes a smaller triangle of width 1
          - It calls **getArea** on that triangle
            - That method returns 1
          - The method returns \( \text{smallerArea} + \text{width} = 1 + 2 = 3 \)
          - The method returns \( \text{smallerArea} + \text{width} = 3 + 3 = 6 \)
          - The method returns \( \text{smallerArea} + \text{width} = 6 + 4 = 10 \)
Recursion

• A recursive computation solves a problem by using the solution of the same problem with simpler values.

• For recursion to terminate, there must be special cases for the simplest inputs.

• To complete our Triangle example, we must handle width <= 0
  ```java
  if (width <= 0) return 0;
  ```

• Two key requirements for recursion success:
  • Every recursive call must simplify the computation in some way
  • There must be special cases to handle the simplest computations directly
Other Ways to Compute Triangle Numbers

• The area of a triangle equals the sum
  \[ 1 + 2 + 3 + \ldots + \text{width} \]

• Using a simple loop:
  ```java
double area = 0;
for (int i = 1; i <= width; i++)
    area = area + i;
```

• Using math:
  \[ 1 + 2 + \ldots + n = n \times (n + 1)/2 \]
  \[ \Rightarrow \text{width} \times (\text{width} + 1) / 2 \]
A triangular shape composed of stacked unit squares like this:

. . .

public class Triangle

    /**
     * Constructs a triangular shape.
     * @param aWidth the width (and height) of the triangle
     */
    public Triangle(int aWidth)
    {
        width = aWidth;
    }

    /**
     * Computes the area of the triangle.
     * @return the area
     */
    public int area()
ch13/triangle/TriangleTester.java

23:   public int getArea()
24:   {
25:     if (width <= 0) return 0;
26:     if (width == 1) return 1;
27:     Triangle smallerTriangle = new Triangle(width - 1);
28:     int smallerArea = smallerTriangle.getArea();
29:     return smallerArea + width;
30:   }
31:
32:   private int width;
33: }

Output:
Enter width: 10
Area: 55
Expected: 55
Permutations

• Design a class that will list all permutations of a string
• A permutation is a rearrangement of the letters
• The string "eat" has six permutations:

"eat"
"eta"
"aet"
"tea"
"tae"
public class PermutationGenerator
{
    public PermutationGenerator(String aWord) { . . . }
    ArrayList<String> getPermutations() { . . . }
}
import java.util.ArrayList;

/**
 * This program demonstrates the permutation generator.
 */

public class PermutationGeneratorDemo {
    public static void main(String[] args) {
        PermutationGenerator generator = new PermutationGenerator("eat");
        ArrayList<String> permutations = generator.getPermutations();
        for (String s : permutations) {
            System.out.println(s);
        }
    }
}
Output:
eat
eta
aet
ate
tea
tae
To Generate All Permutations

• Generate all permutations that start with 'e', then 'a' then 't'

• To generate permutations starting with 'e', we need to find all permutations of "at"

• This is the same problem with simpler inputs

• Use recursion
To Generate All Permutations

• **getPermutations**: loop through all positions in the word to be permuted

• For each position, compute the shorter word obtained by removing ith letter:

  ```java
  String shorterWord = word.substring(0, i) +
  word.substring(i + 1);
  ```

• Construct a permutation generator to get permutations of the shorter word

  ```java
  PermutationGenerator shorterPermutationGenerator = new PermutationGenerator(shorterWord);
  ArrayList<String> shorterWordPermutations = shorterPermutationGenerator.getPermutations();
  ```
To Generate All Permutations

• Finally, add the removed letter to front of all permutations of the shorter word

```java
for (String s : shorterWordPermutations)
{
    result.add(word.charAt(i) + s);
}
```

• Special case: simplest possible string is the empty string; single permutation, itself
import java.util.ArrayList;

/**
 * This class generates permutations of a word.
 */
public class PermutationGenerator {
    /**
     * Constructs a permutation generator.
     * @param aWord the word to permute
     */
    public PermutationGenerator(String aWord) {
        word = aWord;
    }

    /**
     * Gets all permutations of a given word.
     */
    public ArrayList<String> getPermutations() {
        
        // Continued
22:   ArrayList<String> result = new ArrayList<String>();
23:
24:   // The empty string has a single permutation: itself
25:   if (word.length() == 0)
26:   {
27:       result.add(word);
28:       return result;
29:   }
30:
31:   // Loop through all character positions
32:   for (int i = 0; i < word.length(); i++)
33:   {
34:       // Form a simpler word by removing the i-th character
35:       String shorterWord = word.substring(0, i)
36:           + word.substring(i + 1);
37:
38:       // Generate all permutations of the simpler word
39:       PermutationGenerator shorterPermutationGenerator
40:           = new PermutationGenerator(shorterWord);
41:       ArrayList<String> shorterWordPermutations
42:           = shorterPermutationGenerator.getPermutations();
43:

Continued
44:      // Add the removed character to the front of
45:      // each permutation of the simpler word,
46:      for (String s : shorterWordPermutations)
47:      {
48:          result.add(word.charAt(i) + s);
49:      }
50:  }
51: // Return all permutations
52: return result;
53: }
54: }
55: private String word;
56: }
Self Check 13.3

What are all permutations of the four-letter word beat?

**Answer:** They are b followed by the six permutations of eat, e followed by the six permutations of bat, a followed by the six permutations of bet, and t followed by the six permutations of bea.
Our recursion for the permutation generator stops at the empty string. What simple modification would make the recursion stop at strings of length 0 or 1?

**Answer:** Simply change `if (word.length() == 0)` to `if (word.length() <= 1)`, because a word with a single letter is also its sole permutation.
Thinking Recursively

• Problem: test whether a sentence is a palindrome
• Palindrome: a string that is equal to itself when you reverse all characters
  • *A man, a plan, a canal – Panama!*
  • *Go hang a salami, I'm a lasagna hog*
  • *Madam, I'm Adam*
Implement isPalindrome Method

```java
public class Sentence {
    /**
     * Constructs a sentence.
     * @param aText a string containing all characters of the sentence
     */
    public Sentence(String aText) {
        text = aText;
    }

    /**
     * Tests whether this sentence is a palindrome.
     * @return true if this sentence is a palindrome, false otherwise
     */
    Continued
```
Implement isPalindrome Method (cont.)

```java
public boolean isPalindrome()
{
    // ...
}

private String text;
```
Thinking Recursively: Step-by-Step

1. Consider various ways to simplify inputs

- Here are several possibilities:
  - Remove the first character
  - Remove the last character
  - Remove both the first and last characters
  - Remove a character from the middle
  - Cut the string into two halves
Thinking Recursively: Step-by-Step

2. Combine solutions with simpler inputs into a solution of the original problem

• Most promising simplification: remove first and last characters "adam, I'm Ada", is a palindrome too!

• Thus, a word is a palindrome if
  • The first and last letters match, and
  • Word obtained by removing the first and last letters is a palindrome

• What if first or last character is not a letter? Ignore it
  • If the first and last characters are letters, check whether they match; if so, remove both and test shorter string
  • If last character isn't a letter, remove it and test shorter string
  • If first character isn't a letter, remove it and test shorter string
3. Find solutions to the simplest inputs

- Strings with two characters
  - *No special case required; step two still applies*

- Strings with a single character
  - *They are palindromes*

- The empty string
  - *It is a palindrome*
4. Implement the solution by combining the simple cases and the reduction step

```java
public boolean isPalindrome()
{
    int length = text.length();
    // Separate case for shortest strings.
    if (length <= 1) return true;
    // Get first and last characters, converted to lowercase.
    char first = Character.toLowerCase(text.charAt(0));
    char last = Character.toLowerCase(text.charAt(length - 1));
    if (Character.isLetter(first) && Character.isLetter(last))
    {
        // Both are letters.
        if (first == last)
        {
            Continued
```

Continued
Thinking Recursively: Step-by-Step (cont.)

    // Remove both first and last character.
    Sentence shorter = new Sentence(text.substring(1,
        length - 1));
    return shorter.isPalindrome();

} else
    return false;
} else if (!Character.isLetter(last))
{
    // Remove last character.
    Sentence shorter = new Sentence(text.substring(0,
        length - 1));
    return shorter.isPalindrome();
}
else
{

Continued
// Remove first character.
Sentence shorter = new Sentence(text.substring(1));
return shorter.isPalindrome();
Recursive Helper Methods

• Sometimes it is easier to find a recursive solution if you make a slight change to the original problem

• Consider the palindrome test of previous slide
  It is a bit inefficient to construct new `Sentence` objects in every step

Continued
• Rather than testing whether the sentence is a palindrome, check whether a substring is a palindrome:

```java
/**
   Tests whether a substring of the sentence is a palindrome.
   @param start the index of the first character of the substring
   @param end the index of the last character of the substring
   @return true if the substring is a palindrome
*/
public boolean isPalindrome(int start, int end)
```
Recursive Helper Methods

• Then, simply call the helper method with positions that test the entire string:

```java
public boolean isPalindrome()
{
    return isPalindrome(0, text.length() - 1);
}
```
Recursive Helper Methods: isPalindrome

```java
public boolean isPalindrome(int start, int end) {
    // Separate case for substrings of length 0 and 1.
    if (start >= end) return true;
    // Get first and last characters, converted to lowercase.
    char first = Character.toLowerCase(text.charAt(start));
    char last = Character.toLowerCase(text.charAt(end));
    if (Character.isLetter(first) &&
        Character.isLetter(last))
    {
        if (first == last)
        {
            // Test substring that doesn't contain the matching letters.
            return isPalindrome(start + 1, end - 1);
        }
        else return false;
    }
    else return false;
}
```
Recursive Helper Methods: isPalindrom (cont.)

```java
} else if (!Character.isLetter(last)) {
    // Test substring that doesn't contain the last character.
    return isPalindrom(start, end - 1);
}
else {
    // Test substring that doesn't contain the first character.
    return isPalindrom(start + 1, end);
}
```
Self Check 13.6

When does the recursive isPalindrome method stop calling itself?

**Answer:** When `start >= end`, that is, when the investigated string is either empty or has length 1.
Fibonacci Sequence

• Fibonacci sequence is a sequence of numbers defined by
  \[ f_1 = 1 \]
  \[ f_2 = 1 \]
  \[ f_n = f_{n-1} + f_{n-2} \]

• First ten terms:
  1, 1, 2, 3, 5, 8, 13, 21, 34, 55
This program computes Fibonacci numbers using a recursive method.
/**
 * Computes a Fibonacci number.
 * @param n an integer
 * @return the nth Fibonacci number
 */

public static long fib(int n)
{
    if (n <= 2) return 1;
    else return fib(n - 1) + fib(n - 2);
}
Output:
Enter n: 50
fib(1) = 1
fib(2) = 1
fib(3) = 2
fib(4) = 3
fib(5) = 5
fib(6) = 8
fib(7) = 13

... fib(50) = 12586269025
The Efficiency of Recursion

- Recursive implementation of \texttt{fib} is straightforward
- Watch the output closely as you run the test program
- First few calls to \texttt{fib} are quite fast
- For larger values, the program pauses an amazingly long time between outputs
- To find out the problem, let's insert trace messages
ch13/fib/RecursiveFibTracer.java

01: import java.util.Scanner;
02:
03: /**
04:     * This program prints trace messages that show how often the
05:     * recursive method for computing Fibonacci numbers calls itself.
06:     */
07: public class RecursiveFibTracer
08: {
09:     public static void main(String[] args)
10:     {
11:         Scanner in = new Scanner(System.in);
12:         System.out.print("Enter n: ");
13:         int n = in.nextInt();
14:         long f = fib(n);
15:         System.out.println("fib(" + n + ") = " + f);
16:     }
17: }
18: 
19: 

Continued
/**
 * Computes a Fibonacci number.
 * @param n an integer
 * @return the nth Fibonacci number
 */

public static long fib(int n)
{
    System.out.println("Entering fib: n = " + n);
    long f;
    if (n <= 2) f = 1;
    else f = fib(n - 1) + fib(n - 2);
    System.out.println("Exiting fib: n = " + n + " return value = " + f);
    return f;
}
Call Tree for Computing $\text{fib}(6)$

Figure 2 Call Pattern of the Recursive $\text{fib}$ Method
The Efficiency of Recursion

• Method takes so long because it computes the same values over and over

• The computation of $\text{fib}(6)$ calls $\text{fib}(3)$ three times

• Imitate the pencil-and-paper process to avoid computing the values more than once
import java.util.Scanner;

/**
 * This program computes Fibonacci numbers using an iterative method.
 */

class LoopFib {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        System.out.print("Enter n: ");
        int n = in.nextInt();

        for (int i = 1; i <= n; i++) {
            long f = fib(i);
            System.out.println("fib(" + i + ") = " + f);
        }
    }

    static long fib(int i) {
        if (i <= 1) return i;
        return fib(i - 1) + fib(i - 2);
    }
}

Continued
/**
   * Computes a Fibonacci number.
   * @param n an integer
   * @return the nth Fibonacci number
   */

public static long fib(int n)
{
    if (n <= 2) return 1;
    long fold = 1;
    long fold2 = 1;
    long fnew = 1;
    for (int i = 3; i <= n; i++)
    {
        fnew = fold + fold2;
        fold2 = fold;
        fold = fnew;
    }
    return fnew;
}
Output:
Enter n: 50
fib(1) = 1
fib(2) = 1
fib(3) = 2
fib(4) = 3
fib(5) = 5
fib(6) = 8
fib(7) = 13

... fib(50) = 1,258,626,9025
The Efficiency of Recursion

• Occasionally, a recursive solution runs much slower than its iterative counterpart
• In most cases, the recursive solution is only slightly slower
• The iterative `isPalindrome` performs only slightly better than recursive solution
  • *Each recursive method call takes a certain amount of processor time*
• Smart compilers can avoid recursive method calls if they follow simple patterns
• Most compilers don't do that
• In many cases, a recursive solution is easier to understand and implement correctly than an iterative solution
• "To iterate is human, to recurse divine.", L. Peter Deutsch
Iterative isPalindrome Method

public boolean isPalindrome()
{
    int start = 0;
    int end = text.length() - 1;
    while (start < end)
    {
        char first =
            Character.toLowerCase(text.charAt(start));
        char last = Character.toLowerCase(text.charAt(end));
        if (Character.isLetter(first) &&
            Character.isLetter(last))
        {
            // Both are letters.
            if (first == last)
            {
                start++;
                end--;
            }
        }
    }
}
Iterative isPalindrome Method (cont.)

else
    return false;
}

if (!Character.isLetter(last))
    end--;
if (!Character.isLetter(first))
    start++;

return true;
You can compute the factorial function either with a loop, using the definition that $n! = 1 \times 2 \times \ldots \times n$, or recursively, using the definition that $0! = 1$ and $n! = (n - 1)! \times n$. Is the recursive approach inefficient in this case?

**Answer:** No, the recursive solution is about as efficient as the iterative approach. Both require $n - 1$ multiplications to compute $n!$. 