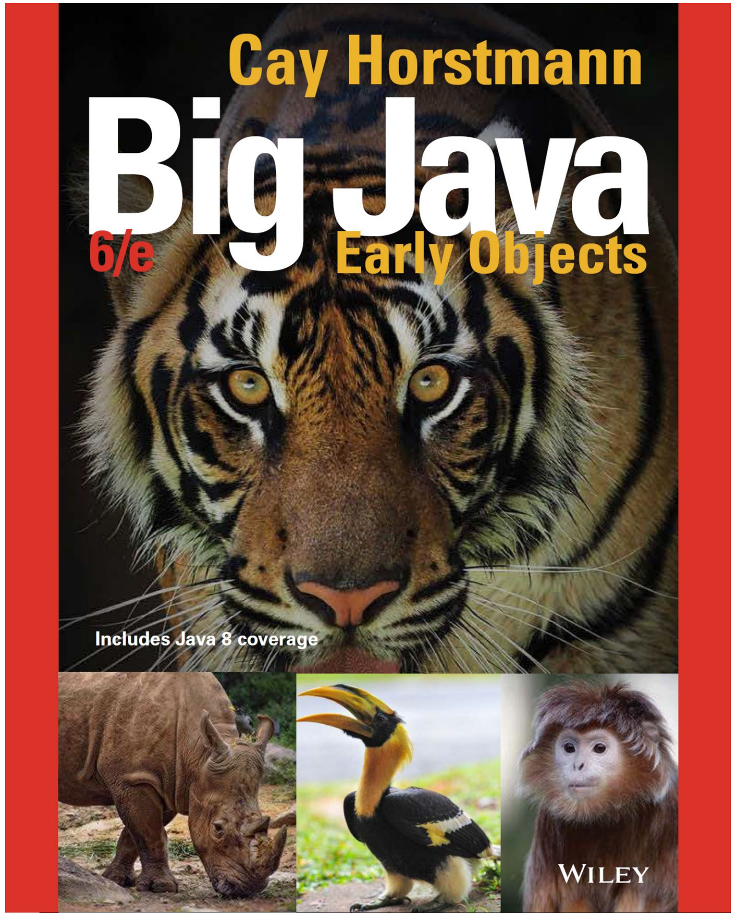

Chapter 13 : Recursion

Part II (Optional)

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

Recursive Helper Methods

- Rather than testing whether the sentence is a palindrome, check whether a substring is a palindrome:
- ```
  /**
     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:
  - ```
    public boolean isPalindrome()
    {
      return isPalindrome(0, text.length() - 1);
    }
    ```
    ```

Recursive Helper Methods: isPalindrome

```
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;
```

Continued

Recursive Helper Methods: isPalindrome (cont.)

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

Self Check 13.3

Do we have to give the same name to both `isPalindrome` methods?

Answer: No — the first one could be given a different name such as `substringIsPalindrome`.

Self Check 13.4

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

The Efficiency of Recursion

- Recursive implementation of `fib` is straightforward
- Watch the output closely as you run the test program
- First few calls to `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**

Call Tree for Computing `fib(6)`

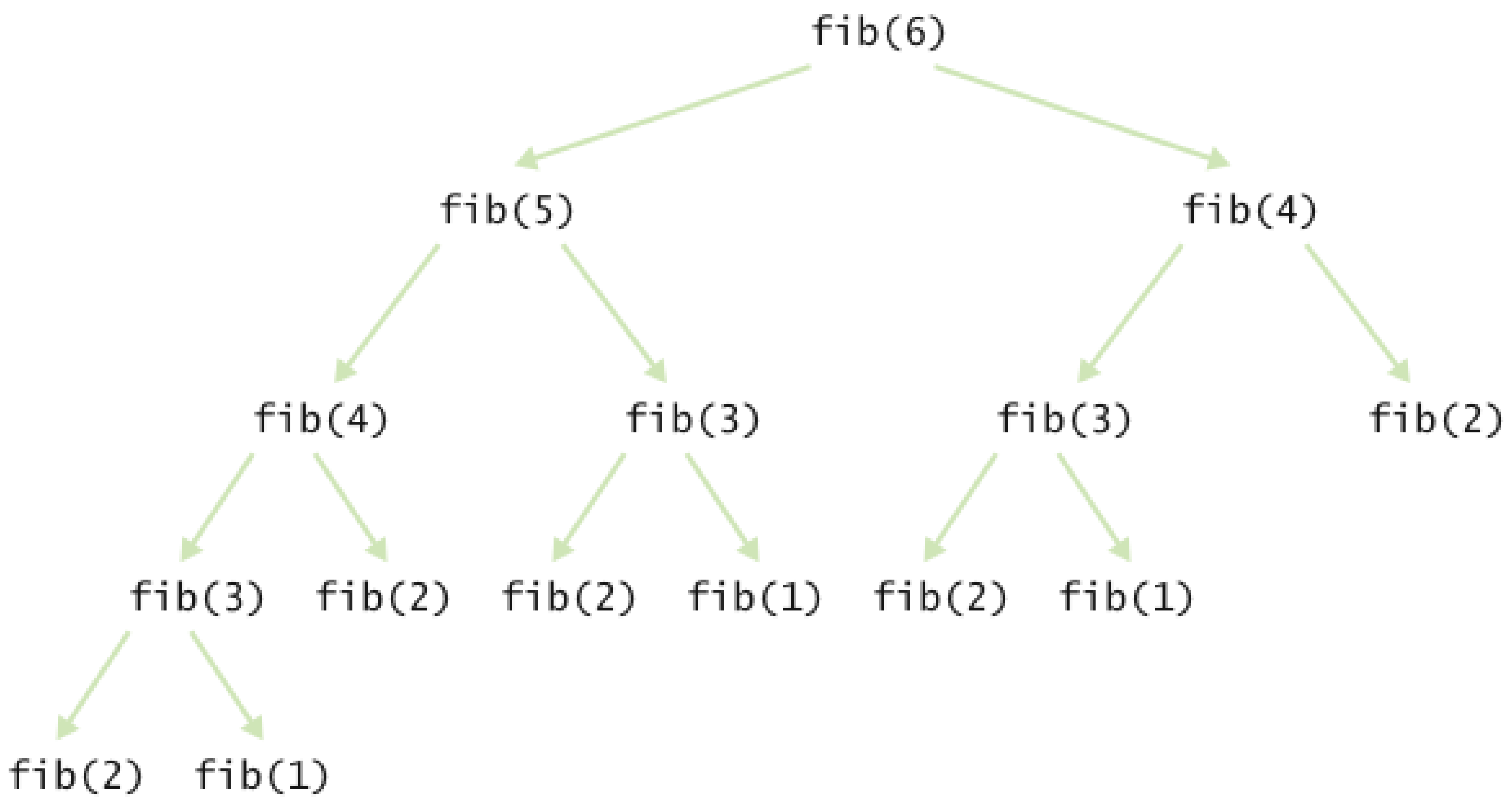

Figure 2 Call Pattern of the Recursive fib Method

The Efficiency of Recursion

- Method takes so long because it computes the same values over and over
- The computation of `fib(6)` calls `fib(3)` three times
- Imitate the pencil-and-paper process to avoid computing the values more than once

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 is `Palindrome` 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--;
         }
```

Continued

Iterative `isPalindrome` Method (cont.)

```
            else
               return false;
         }
         if (!Character.isLetter(last))
            end--;
         if (!Character.isLetter(first))
            start++;
      }
      return true;
   }
```

Self Check 13.5

Is it faster to compute the triangle numbers recursively, as shown in Section 13.1, or is it faster to use a loop that computes 1 + 2 + 3 + . . . + width?

Answer: The loop is slightly faster. Of course, it is even faster to simply compute width * (width + 1) / 2.

Self Check 13.6

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 $n!$.

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 Interface of `PermutationGenerator`

```
public class PermutationGenerator
{
   public PermutationGenerator(String aWord) { ... }
   ArrayList<String> getPermutations() { ... }
}
```

ch13/permute/PermutationGeneratorDemo.java

```
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);
      }
   }
}

```

Continued

ch13/permute/PermutationGeneratorDemo.java (cont.)

Program Run:

```
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 i^{th} letter:

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

- Construct a permutation generator to get permutations of the shorter word:

```
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:
  - ```
    for (String s : shorterWordPermutations)
    {
       result.add(word.charAt(i) + s);
    }
    ```
- Special case: Simplest possible string is the empty string; single permutation, itself
    ```

Self Check 13.7

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`.

Self Check 13.8

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.

Self Check 13.9

Why isn't it easy to develop an iterative solution for the permutation generator?

Answer: An iterative solution would have a loop whose body computes the next permutation from the previous ones. But there is no obvious mechanism for getting the next permutation. For example, if you already found permutations `eat`, `eta`, and `aet`, it is not clear how you use that information to get the next permutation. Actually, there is an ingenious mechanism for doing just that, but it is far from obvious — see Exercise P13.12.