

CS 304 Lecture 4

Recursion

Xiwei Wang, Ph.D.

Assistant Professor

Department of Computer Science

Northeastern Illinois University

Chicago, Illinois, 60625

15 September 2015

Recursion

- Recursion is a powerful technique for breaking up complex computational problems into simpler ones, where the "simpler one" is the solution to the whole problem!
- A recursive method is a method that calls itself, reducing the problem a bit on each call:

```
void solveIt(the-Problem)
{
    . . .
    solveIt(the-Problem-a-bit-reduced) ;
}
```

- In recursion, the same computation recurs, or occurs repeatedly, as the problem is solved. But this is not looping!
- Recursion is often the most natural way of thinking about a problem, and there are some computations that are very difficult to perform without recursion.

How to think recursively – Triangles of boxes

- Write a method that prints a triangle of boxes.

```
public void print_triangle(int side_length) ;  
    []  
    [] []  
    [] [] []  
    [] [] [] []
```

- Of course it can be done by using nested `for` loops. But we want to use the recursive way.
- To think recursively, you can pretend that "someone else" has written the method that draws triangles.
- Now that that problem is solved, analyze the problem, looking for a way to reduce the problem and use that method to solve the reduced problem.

How to think recursively – Triangles of boxes

- You can call that method (that someone else wrote) to solve the reduced problem of printing a triangle of side length three.

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

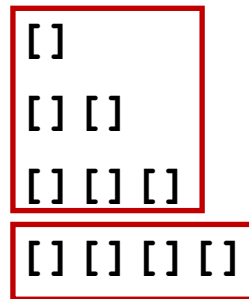
- Then you solve the much easier problem of printing a line of 4 boxes.

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

How to think recursively – Triangles of boxes

```
public static void print_triangle(int side_length)
{
    print_triangle(side_length - 1);
    for (int i = 0; i < side_length; i++)
    {
        System.out.print("[]");
    }
    System.out.println();
}
```

The problem is being reduced by making the length 1 shorter.



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

The simple for statement draws a line side_length long.

BUT, what if side_length is 0?

How to think recursively – Triangles of boxes

- When the length of the line is less than one, we want to terminate the recursive call.
- We use the "end condition" ("termination condition", "base case") to stop the call.

```
public static void print_triangle(int side_length)
{
    if (side_length < 1) { return; }

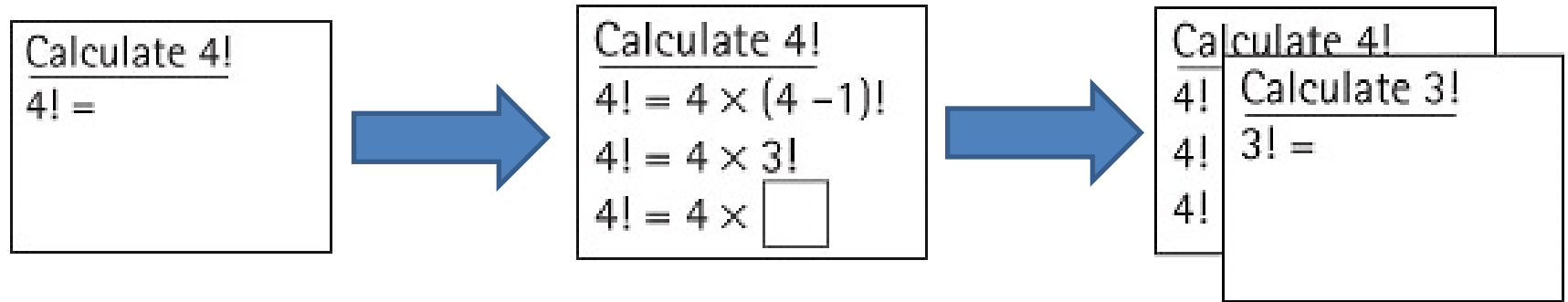
    print_triangle(side_length - 1);
    for (int i = 0; i < side_length; i++)
    {
        System.out.print("[]");
    }
    System.out.println();
}
```

If this condition is met, simply stop.

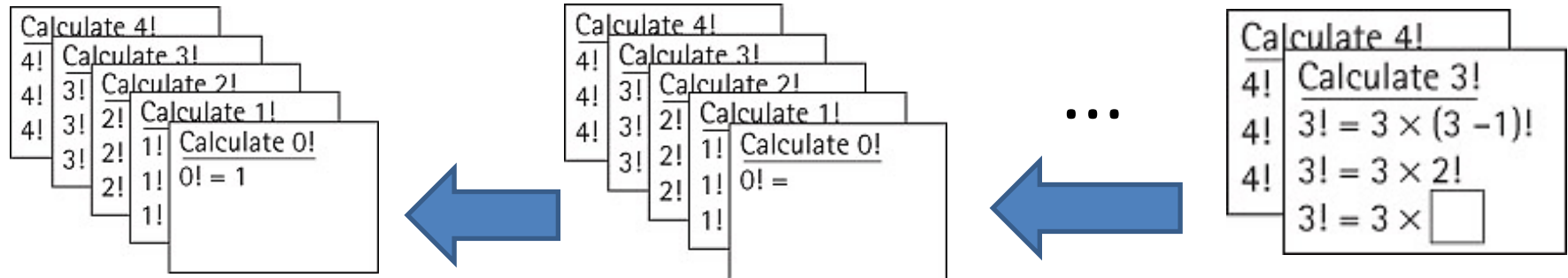
Recursive algorithms

- There are three factors for a successful recursive method:
 - Recursive algorithm A solution that is expressed in terms of smaller instances of itself and a base case.
 - Base case The case for which the solution can be stated non-recursively.
 - General (recursive) case The case for which the solution is expressed in terms of a smaller version of itself.

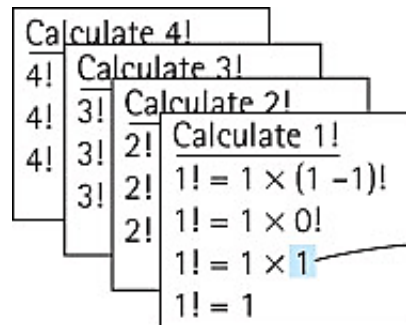
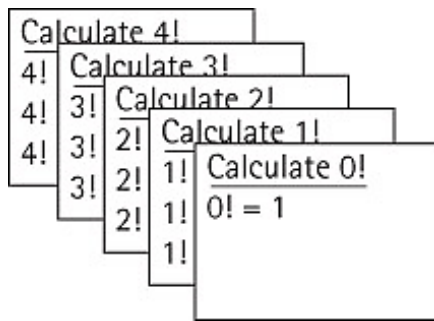
Calculating factorials with recursion



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



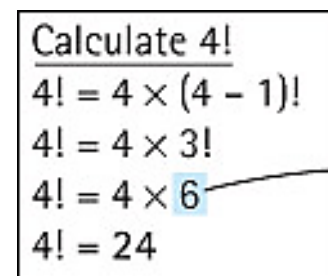
Calculating factorials with recursion



From the discarded Calculate 0! card



...



From the discarded Calculate 3! card

Calculating factorials with recursion

```
public static int factorial(int n)
// Precondition: n is non-negative
//
// Returns the value of "n!".
{
    if (n == 0)
        return 1;                // Base case
    else
        return (n * factorial(n - 1)); // General case
}
```

Recursion terms

- **Recursive call** - A method call in which the method being called is the same as the one making the call.
- Recursive calls can be divided into two subcategories:
 - **Direct recursion** - Recursion in which a method directly calls itself, like the factorial method.
 - **Indirect recursion (a.k.a mutual recursion)** - Recursion in which a chain of two or more method calls returns to the method that originated the chain, for example method **A** calls method **B** which in turn calls method **A**.

Iterative solution for factorial

- We have used the factorial algorithm to demonstrate recursion because it is familiar and easy to visualize. In practice, one would never want to solve this problem using recursion, since a straightforward, more efficient iterative solution exists:

```
public static int factorial(int n)
{
    int value = n;
    int retValue = 1;    // return value
    while (value != 0)
    {
        retValue = retValue * value;
        value = value - 1;
    }
    return retValue;
}
```

The three questions

- When we design recursive solutions, we can use the previously mentioned three questions to help us verify, design, and debug them.
- To verify that a recursive solution works, we must be able to answer "Yes" to all three of these questions:
 - The Base-Case Question: Is there a nonrecursive way out of the algorithm, and does the algorithm work correctly for this base case?
 - The Smaller-Caller Question: Does each recursive call to the algorithm involve a smaller case of the original problem, leading inescapably to the base case?
 - The General-Case Question: Assuming the recursive call(s) to the smaller case(s) works correctly, does the algorithm work correctly for the general case?

The three questions

```
Factorial (int n)
// Assume n >= 0
if (n == 0)
    return 1;
else
    return n * Factorial(n - 1);
```

- The Base-Case Question - Is there a nonrecursive way out of the algorithm, and does the algorithm work correctly for this base case?
 - The base case occurs when n is 0.
 - The factorial algorithm then returns the value of 1, which is the correct value of $0!$, and no further (recursive) calls to Factorial are made.
 - **The answer is yes.**

The three questions

```
Factorial (int n)
// Assume n >= 0
if (n == 0)
    return 1;
else
    return n * Factorial(n - 1);
```

- The Smaller-Caller Question - Does each recursive call to the algorithm involve a smaller case of the original problem, leading inescapably to the base case?
 - The parameter is n and the recursive call passes the argument $n - 1$. Therefore each subsequent recursive call sends a smaller value, until the value sent is finally 0.
 - At this point, as we verified with the base-case question, we have reached the smallest case, and no further recursive calls are made.
 - **The answer is yes.**

The three questions

```
Factorial (int n)
// Assume n >= 0
if (n == 0)
    return 1;
else
    return n * Factorial(n - 1);
```

- The General-Case Question - Assuming the recursive call(s) to the smaller case(s) works correctly, does the algorithm work correctly for the general case?
 - Assuming that the recursive call `Factorial($n - 1$)` gives us the correct value of $(n - 1)!$, the return statement computes $n * (n - 1)!$.
 - This is the definition of a factorial, so we know that the algorithm works in the general case.
 - **The answer is yes.**

Steps for designing recursive solutions

1. Get an exact definition of the problem to be solved.
2. Determine the size of the problem to be solved on this call to the method.
3. Identify and solve the base case(s) in which the problem can be expressed nonrecursively. This ensures a yes answer to the base-case question.
4. Identify and solve the general case(s) correctly in terms of a smaller case of the same problem—a recursive call. This ensures yes answers to the smaller-caller and general-case questions.

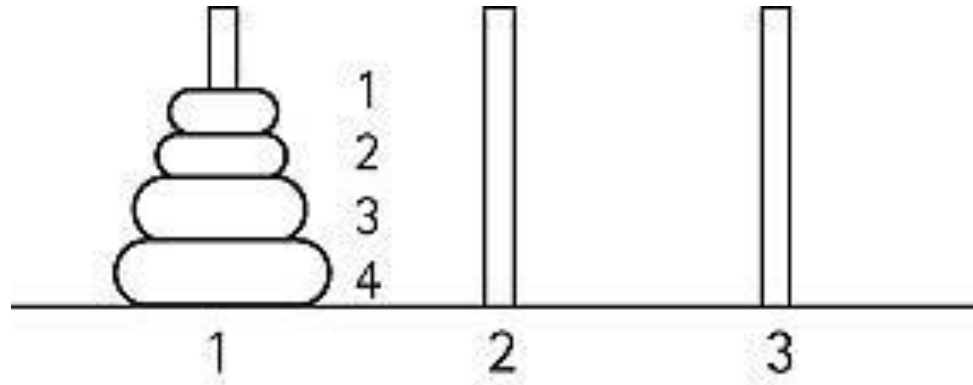
Towers of Hanoi

- Many of the recursive methods we have seen could be solved just as easily using iteration. But sometimes recursion is a far simpler way to solve the problem. One such problem is the Tower of Hanoi, also known as the Tower of Brahma:

"In the great temple of Benares, so legend has it, are 64 golden disks, all of different sizes and mounted on diamond pegs.

At the time of creation, the god Brahma placed all the disks on one peg, in order of size with the largest on the bottom. The task of the temple priests is to transfer the disks unceasingly from peg to peg, one at a time, never placing a larger disk on a smaller one. When all the disks have been transferred the universe will end. (Edouard Lucas, 1883)"

Towers of Hanoi

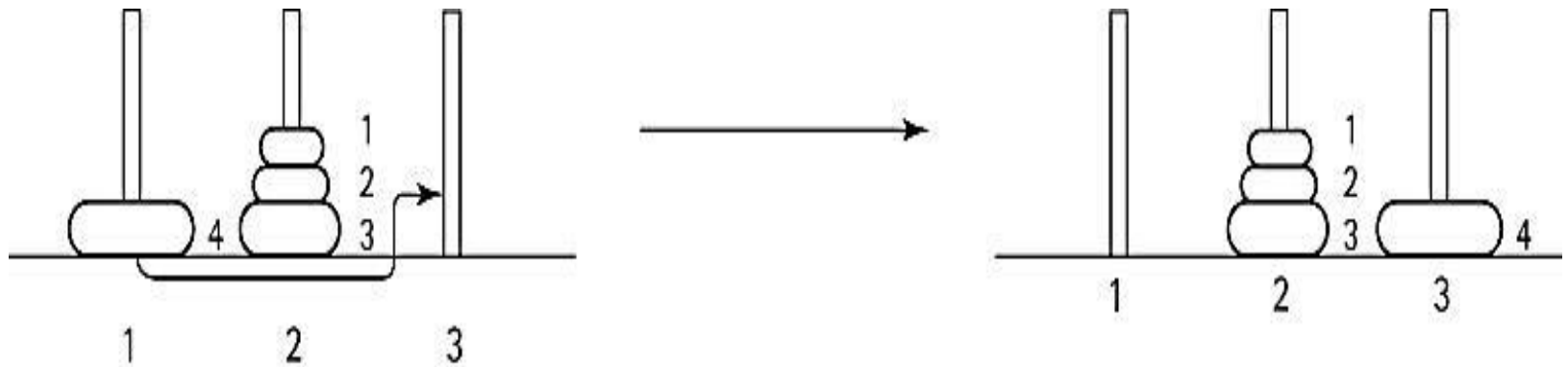


- Move the disks, one at a time, to the third peg.
- A disk cannot be placed on top of one that is smaller in diameter.
- The middle peg can be used as an auxiliary peg, but it must be empty at the beginning and at the end of the game.
- The disks can only be moved one at a time.

Towers of Hanoi

- General approaches

- To move the largest disk (disk 4) to peg 3, we must move the three smaller disks to peg 2 (this cannot be done with 1 move).
- Let's assume we can do this.
- Then disk 4 can be moved into its final place:



Towers of Hanoi

- Can you see that our assumption (that we move the three smaller disks to peg 2) involved solving a smaller version of the problem? We have solved the problem using recursion.
- The general recursive algorithm for moving n disks from the starting peg to the destination peg 3 is:
 - Move n disks from starting peg to destination peg
 - Move $n - 1$ disks from starting peg to auxiliary peg
 - Move the n th disk from starting peg to destination peg
 - Move $n - 1$ disks from auxiliary peg to destination peg
- We'll practice a few times:
<http://www.dynamicdrive.com/dynamicindex12/towerhanoi.htm>

Towers of Hanoi

```
public static void doTowers(  
    int n,                // Number of disks to move  
    int startPeg,         // Peg containing disks to move  
    int auxPeg,           // Peg holding disks temporarily  
    int endPeg            ) // Peg receiving disks being moved  
{  
    if (n > 0)  
    {  
        // Move n - 1 disks from starting peg to auxiliary peg  
        doTowers(n - 1, startPeg, endPeg, auxPeg);  
  
        System.out.println("Move disk from peg " + startPeg  
                            + " to peg " + endPeg);  
  
        // Move n - 1 disks from auxiliary peg to ending peg  
        doTowers(n - 1, auxPeg, startPeg, endPeg);  
    }  
}
```

Recursion and complexity

- How many moves does it take to solve the *Tower of Hanoi*?
 - We are asking about the computational complexity of the problem.
 - The answer depends on the number of disks N .
 - We want to know how the answer changes with N .
 - Base case: $\text{Moves}(0) = 0$ (nothing needed).
 - Recursive case: $\text{Moves}(i) = 1 + 2 \times \text{Moves}(i - 1)$
 - $\text{Moves}(1) = 1 + 2 \times 0 = 1$
 - $\text{Moves}(2) = 1 + 2 \times 1 = 3$
 - $\text{Moves}(3) = 1 + 2 \times (1 + 2 \times 1) = 7$
 - $\text{Moves}(4) = 1 + 2 \times (1 + 2 \times (1 + 2 \times 1)) = 15$
 - It looks like it's approximately doubling each time.
 - That makes sense - we have to solve the next-smaller problem twice.
 - $\text{Moves}(N) = 1 + 2^1 + 2^2 + 2^3 + \dots + 2^{N-1} = 2^N - 1$
 - So this problem has **exponential complexity**: the time taken is approximately something (2) to the power of N .

Recursion and complexity

- So what about the 64-disk Tower of Hanoi? How long will that take?
 - How many moves? $2^{64}-1 = 18,446,744,073,709,551,615$
 - So if the priests move one disk per second. . .
 - There are on average 31,556,952 seconds in a year.
 - Divide those: 584,554,049,254 years.
 - Astronomers tell us the universe is about 14,000,000,000 years old. . .
 - So we have plenty of time left.

Fibonacci numbers

- In mathematics, the Fibonacci numbers or Fibonacci sequence are the numbers in the following integer sequence:

1, 1, 2, 3, 5, 8, 13, 21, ...

$$f_0 = 1$$

$$f_1 = 1$$

$$f_n = f_{n-2} + f_{n-1}$$

- How do you calculate Fibonacci numbers by using iterative solutions?
- What about recursive solutions?

Fibonacci numbers – iterative solution

```
public static int fibIter(int n)
{
    if (n == 0 || n == 1)
        return 1;

    int[] FibNumbers = new int[n + 1];
    FibNumbers[0] = FibNumbers[1] = 1;

    for (int i = 2; i <= n; i++)
        FibNumbers[i] = FibNumbers[i - 1] +
                        FibNumbers[i - 2];

    return FibNumbers[n];
}
```

Fibonacci numbers – recursive solution

```
public static int fibRec(int n)
{
    if (n == 0 || n == 1)
        return 1;
    else
        return fibRec(n - 1) + fibRec(n - 2);
}
```

Common error: Infinite recursion

- Consider this code:

```
public static void forever_young()  
{  
    System.out.print("I am ");  
    forever_young();  
    System.out.print("forever young!");  
}
```


Common error: Infinite recursion

Guys!
How do I stop
this crazy
thing?

Common error: Infinite recursion

- You would get very, very old waiting to be "forever young!"

```
public static void forever_young()  
{  
    System.out.print("I am ");  
    forever_young();  
    System.out.print("forever young!");  
}
```

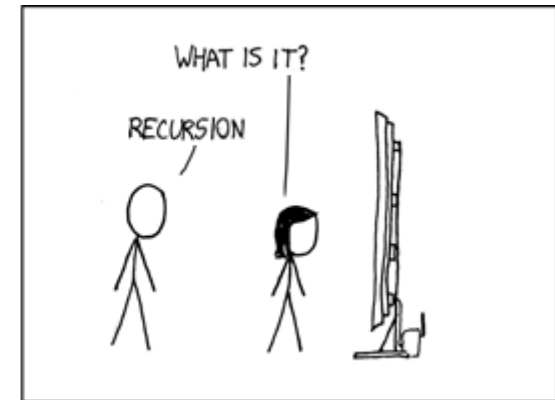
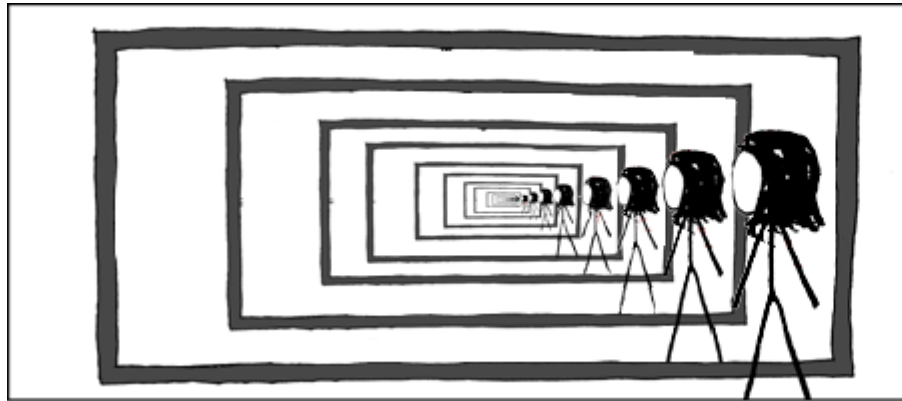


Common error: Infinite recursion

- Infinite recursion is an error. Each method call uses some system resources that only a return statement releases.
- It is very likely that your computer will hang or crash or whatever you call it when it just WON'T STOP.
 - In the previous code, the programmer forgot to write the end test. This is easy to see in that code.
 - Infinite recursion can also occur when the test to end never happens.

Common error: Infinite recursion

- Recall the key requirements for a successful recursive method:
 - Every recursive call must simplify the task in some way.
 - There must be special cases to handle the simplest tasks directly so that the method will stop calling itself.
- Failing to implement these requirements can lead to infinite recursion (and very unpleasant consequences).



Thinking recursively – Palindromes

- Palindrome: a string that is equal to itself when you reverse all characters.
 - I, madam, rotor, level, 123321....
- Write a method to test if a string is a palindrome.
 - `boolean is_palindrome(String s)`
- Think about "rotor". How do you reduce the problem?
 - Remove the first character?
 - Remove the last character?
 - Remove both the first and the last character?

Thinking recursively – Palindromes

- If the end letters are the same AND `is_palindrome` (the middle word) then the string is a palindrome!

"rotor"

(chop) (chop)

"r" "oto" "r"

Thinking recursively – Palindromes

- When to stop the recursive call?
 - There are two possible cases:
 1. string of length 0
 2. string of length 1

- The base case is:

```
if (s.length() == 0 || s.length() == 1 )  
{  
    return true;  
}
```

- If the string length is not 0 or 1, we do the recursive calls.

Thinking recursively – Palindromes

```
public static boolean is_palindrome(String s)
{
    // Separate case for shortest strings
    if (s.length() == 0 || s.length() == 1 )
    {
        return true;
    }
    // Get first and last characters
    char first = s.charAt(0);
    char last = s.charAt(s.length() - 1);
    if (first == last)
    {
        String shorter = s.substring(1, s.length() - 1);
        return is_palindrome(shorter);
    }
    else
    {
        return false;
    }
}
```

Recursive helper methods

- Sometimes it is easier to find a recursive solution if you change the original problem slightly. Then the original problem can be solved by calling a recursive helper method.
- Consider the palindrome problem. It is a bit inefficient to construct new string objects in every step.
- Now consider the following change in the problem.
 - Rather than testing whether the entire string is a palindrome, check whether a substring is a palindrome:

```
boolean substring_is_palindrome(string s, int start,  
                                int end);
```

Recursive helper methods

```
public static boolean substring_is_palindrome(String s,
                                             int start, int end)
{
    // Separate case for substrings of length 0 and 1
    if (start >= end) { return true; }
    if (s.charAt(start) == s.charAt(end))
    {
        // Test substring that doesn't contain
        // the first and last letters
        return substring_is_palindrome(s, start+1, end-1);
    }
    else
    {
        return false;
    }
}
```

In the recursive calls, simply adjust the start and end arguments to skip over matching letter pairs. There is no need to construct new string objects to represent the shorter strings.

Recursive helper methods

- Provide a method that solves the problem by calling the helper method.

```
public static boolean is_palindrome(String s)
{
    return substring_is_palindrome(s, 0, s.length() - 1);
}
```

- This method is not recursive but the helper method is.

The efficiency of recursion

- Recursion can be a powerful tool for implementing complex algorithms. However, recursion can lead to algorithms that perform poorly.

- Come back to Fibonacci numbers:

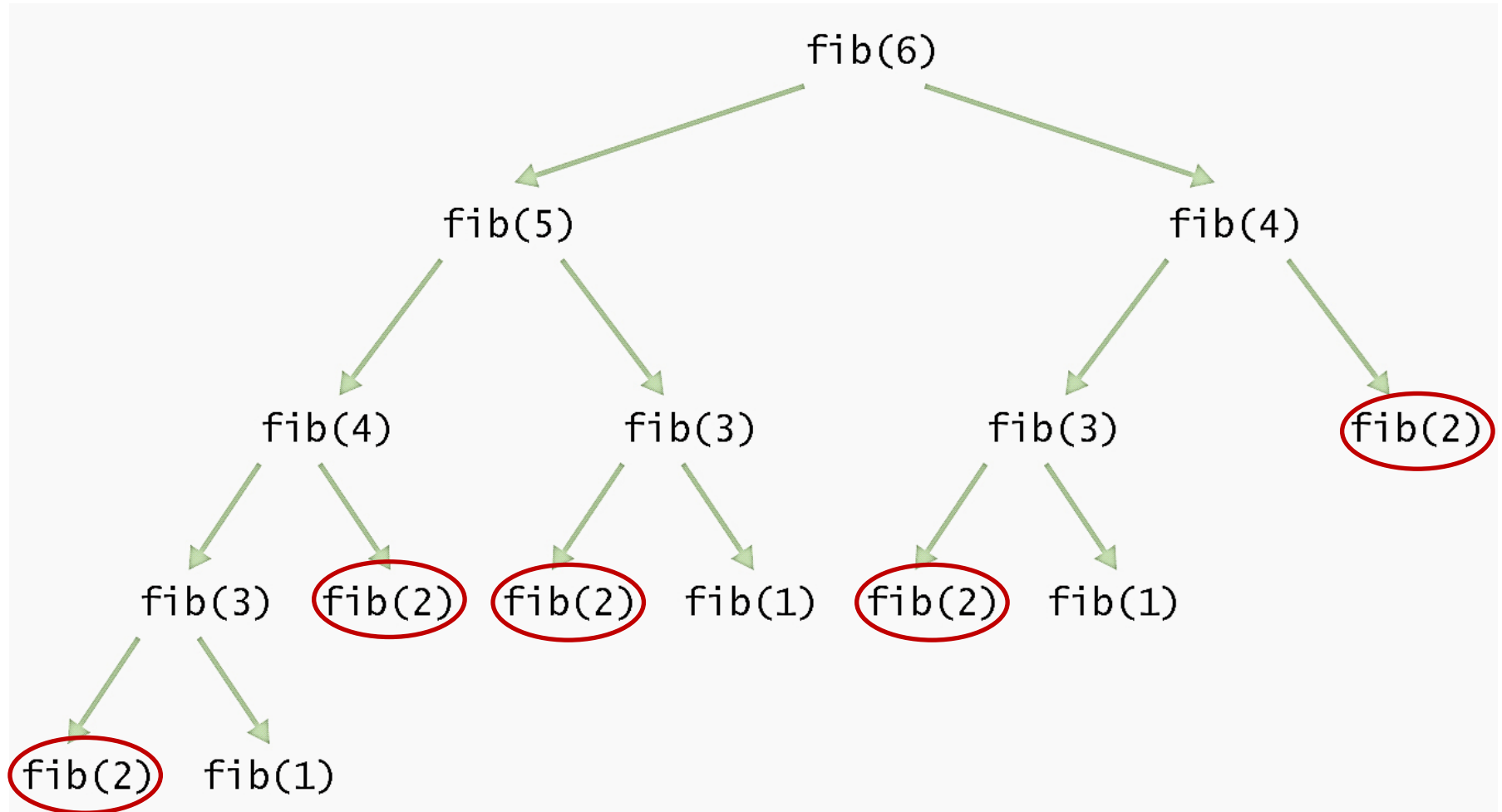
$$\begin{aligned}f_0 &= 1 \\f_1 &= 1 \\f_n &= f_{n-2} + f_{n-1}\end{aligned}$$

```
public static int fibRec(int n)
{
    if (n == 0 || n == 1)
        return 1;
    else
        return fibRec(n - 1) +
               fibRec(n - 2);
}
```

It will take quite a long time if **n** is big.

The efficiency of recursion

- This can be shown more clearly as a call tree:



The efficiency of recursion

- It is apparent that the iterative solution (using for loop) is better than the recursive solution in this case.

- So is the iterative solution always better than the recursive solution?

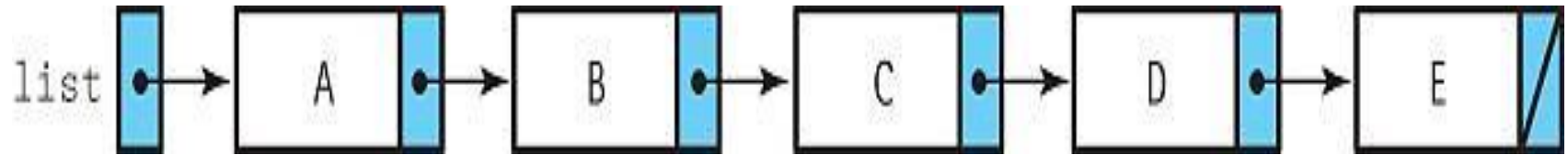
- Look at the iterative palindrome solution:

```
public static boolean is_palindrome(String s) {  
    int start = 0;  
    int end = s.length() - 1;  
    while (start < end) {  
        if (s.charAt(start) != s.charAt(end))  
            return false;  
        start++;  
        end--;  
    }  
    return true;  
}
```

The efficiency of recursion

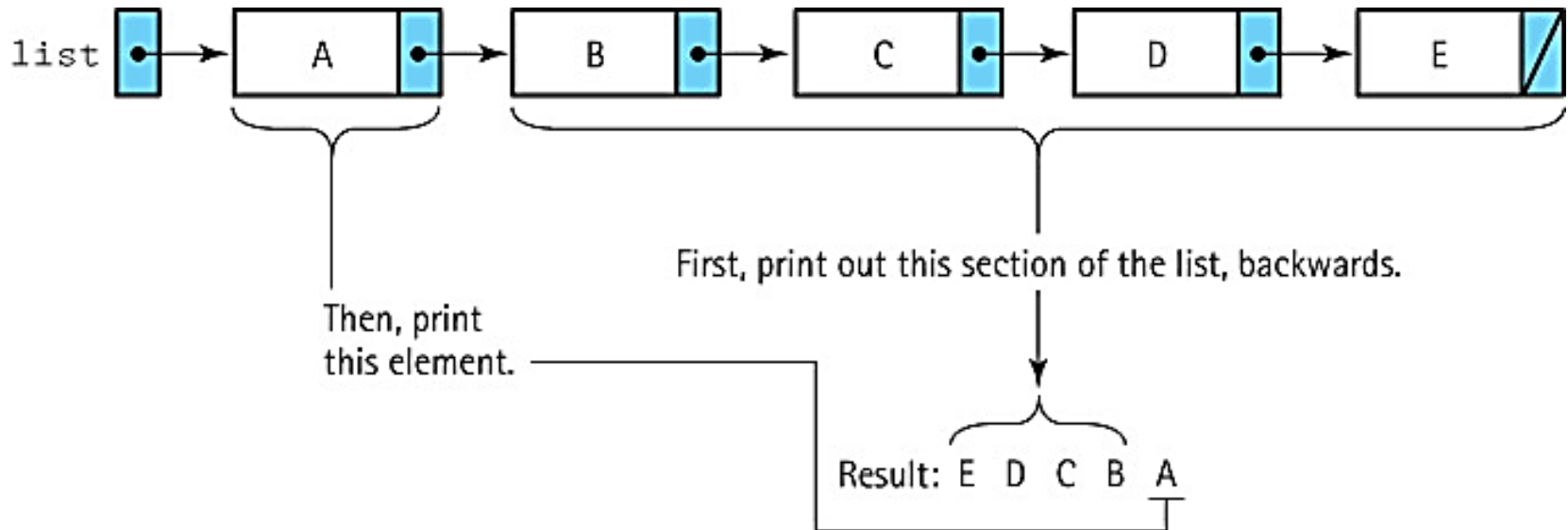
- Both the iteration and the recursion run at about the same speed.
 - If a palindrome has n characters, the iteration executes the loop $n/2$ times and the recursive solution calls itself $n/2$ times, because two characters are removed in each step.
 - In such a situation, the iterative solution tends to be a bit faster, because every method call takes a certain amount of processor time (and a recursive method call is a method call).
 - From that point of view, an iterative solution is preferable.
- However, recursive solutions are easier to understand and implement correctly than their iterative counterparts. There is a certain elegance and economy of thought to recursive solutions that makes them more appealing.

Recursive linked-list processing



- Reverse Printing: Our goal is to print the elements of a linked list in reverse order.
- This problem is much more easily and elegantly solved recursively than it is iteratively.

Recursive linked-list processing



- Recursive `revPrint(listRef)`
 - Print out the second through last elements in the list referenced by `listRef` in reverse order.
 - Then print the first element in the list referenced by `listRef`.

Recursive linked-list processing

- Extending `LinkedStack` with a *reverse print*:

```
import ch03.stacks.*;
import support.LLObjectNode;

public class LinkedStack2<T> extends LinkedStack<T>
{
    private void revPrint(LLNode<T> listRef)
    {
        if (listRef != null)
        {
            revPrint(listRef.getLink());
            System.out.println(" " + listRef.getInfo());
        }
    }
    public void printReversed()
    {
        revPrint(top);
    }
}
```

Removing recursion

- We consider two general techniques that are often substituted for recursion
 - iteration
 - stacking
- First we take a look at how recursion is implemented.
 - Understanding how recursion works helps us see how to develop nonrecursive solutions.

Removing recursion - Iteration

- Suppose the recursive call is the last action executed in a recursive method (*tail recursion*)
 - The recursive call causes an activation record to be put on the run-time stack to contain the invoked method's arguments and local variables.
 - When this recursive call finishes executing, the run-time stack is popped and the previous values of the variables are restored.
 - Execution continues where it left off before the recursive call was made.
 - But, because the recursive call is the last statement in the method, there is nothing more to execute and the method terminates without using the restored local variable values.
- In such a case the recursion can easily be replaced with iteration.

Removing recursion - Iteration

- Converting the recursive factorial to the iterative solution:

```
public static int factorial(int n) {  
    if (n == 0)  
        return 1;                // Base case  
    else  
        return (n * factorial(n - 1)); // General case  
}
```



```
public static int factorial(int n) {  
    int retValue = 1; // return value  
    while (n != 0) {  
        retValue = retValue * n;  
        n = n - 1;  
    }  
    return (retValue);  
}
```

Removing recursion - Stacking

- When the recursive call is not the last action executed in a recursive method, we cannot simply substitute a loop for the recursion.
- In such cases we can "mimic" recursion by using our own stack to save the required information when needed, as shown in the Reverse Print example on the next slide.

Removing recursion - Stacking

```
public class LinkedStack3<T> extends LinkedStack<T> {
    public void printReversed() {
        UnboundedStackInterface<T> stack = new LinkedStack<T>();
        LLNode<T> listNode;
        Object object;

        listNode = top;

        while (listNode != null) {    // Put references onto the
                                    // stack
            stack.push(listNode.getInfo());
            listNode = listNode.getLink();
        }

        // Retrieve references in reverse order and print elements
        while (!stack.isEmpty()){
            System.out.println(" " + stack.top());
            stack.pop();
        }
    }
}
```

Summary: recursive vs nonrecursive

- Recursion Overhead

- A recursive solution usually has more "overhead" than a non-recursive solution because of the number of method calls.
 - Time: each call involves processing to create and dispose of the activation record, and to manage the run-time stack.
 - Space: activation records must be stored.

- Inefficient Algorithms

- Another potential problem is that a particular recursive solution might just be inherently inefficient. This can occur if the recursive approach repeatedly solves the same sub-problem, over and over again.

Summary: recursive vs nonrecursive

- Clarity

- For many problems, a recursive solution is simpler and more natural for the programmer to write. The work provided by the system stack is "hidden" and therefore a solution may be easier to understand.
- Compare, for example, the recursive and nonrecursive approaches to printing a linked list in reverse order that were developed previously in this chapter. In the recursive version, we let the system take care of the stacking that we had to do explicitly in the nonrecursive method.

Action items

- Read book chapter 4.