King Fahd University of Petroleum & Minerals

College of Computer Science & Engineering

Information & Computer Science Department

Unit 5

Recursion

Reading Assignment

- "Data Structures and Algorithms in Java", 3rd Edition, Adam Drozdek, Cengage Learning, ISBN 978-9814239233
  - Chapter 5 Sections 1-8, 10.
  - Backtracking and the Case Study in Sections 9 and 11 are optional reading.

Objectives

Discuss the following topics:

- Recursive Definitions
- Method Calls and Recursion Implementation
- Tracing of Recursive Methods
- Tail and Non-Tail Recursive Methods
- Direct and Indirect Recursive Methods
- Nested and non-Nested Recursive Methods
- Excessive Recursion
- Final Remarks on Recursion

Recursive Definitions

- **Recursive definitions** are programming concepts that define themselves
- A recursive definition consists of two parts:
  - The **anchor** or **ground case**, the basic elements that are the building blocks of all other elements of the set
  - Rules that allow for the construction of new objects out of basic elements or objects that have already been constructed

Recursive Definitions (continued)

- Recursive definitions serve two purposes:
  - **Generating** new elements
  - **Testing** whether an element belongs to a set
- Recursive definitions are frequently used to define functions and sequences of numbers

What is a Recursive Method?

- A method is recursive if it calls itself either directly or indirectly.
- Recursion is a technique that allows us to break down a problem into one or more simpler sub-problems that are similar in form to the original problem.
- Example 1: A recursive method for computing x!

```
long factorial (int x) {
   if (x == 0)
      return 1;     //base case
   else
      return x * factorial (x – 1);  //recursive case
}
```

- This method illustrates all the important ideas of recursion:
  - A base (or stopping) case
    - Code first tests for stopping condition (is x = = 0 ?)
    - Provides a direct (non-recursive) solution for the base case (0! = 1).
  - The recursive case
    - Expresses solution to problem in 1, 2 or more smaller parts
    - Invokes itself to compute the smaller parts, eventually reaching the base case

What is a Recursive Method?

- Example 2: count zeros in an array

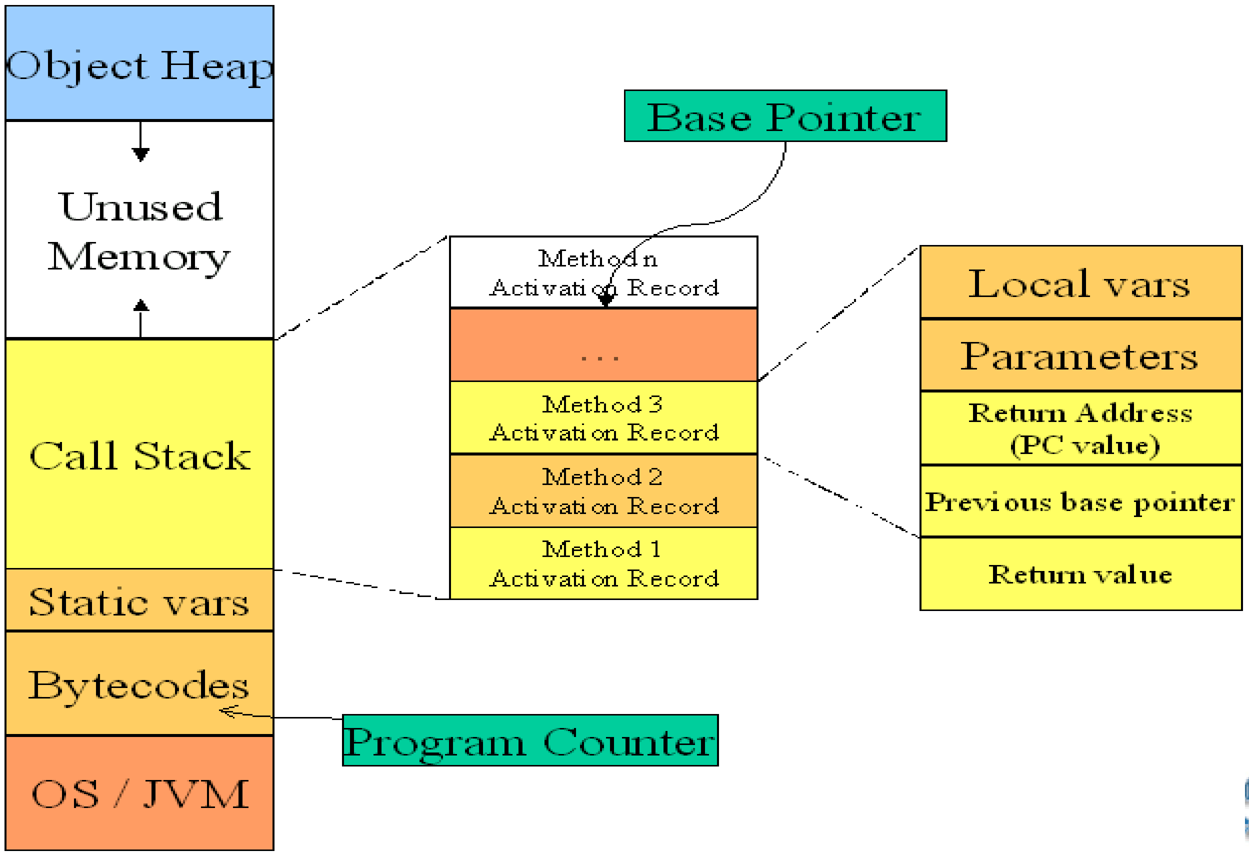
```
int countZeros(int[] x, int index) {
   if (index == 0)
         return x[0] == 0 ? 1: 0;
   else if (x[index] == 0)
      return 1 + countZeros(x, index - 1);
   else
      return countZeros(x, index - 1);
}
```

Method Calls and Recursion Implementation

- When a method is called an Activation Record is created. It contains:
  - The values of the parameters.
  - The values of the local variables.
  - The return address (The address of the statement after the call statement).
  - The previous activation record address.
  - A location for the return value of the activation record.
- When a method returns:
  - The return value of its activation record is passed to the previous activation record or it is passed to the calling statement if there is no previous activation record.
  - The Activation Record is popped entirely from the stack.
- Recursion is handled in a similar way. Each recursive call creates a separate Activation Record. As each recursive call completes, its Activation Record is popped from the stack. Ultimately control passes back to the calling statement.

Method Calls and Recursion Implementation

- Modern computers use a stack as the primary memory management model for a running program.
- Each running program has its own memory allocation containing the typical layout as shown below.



Method Calls and Recursion Implementation

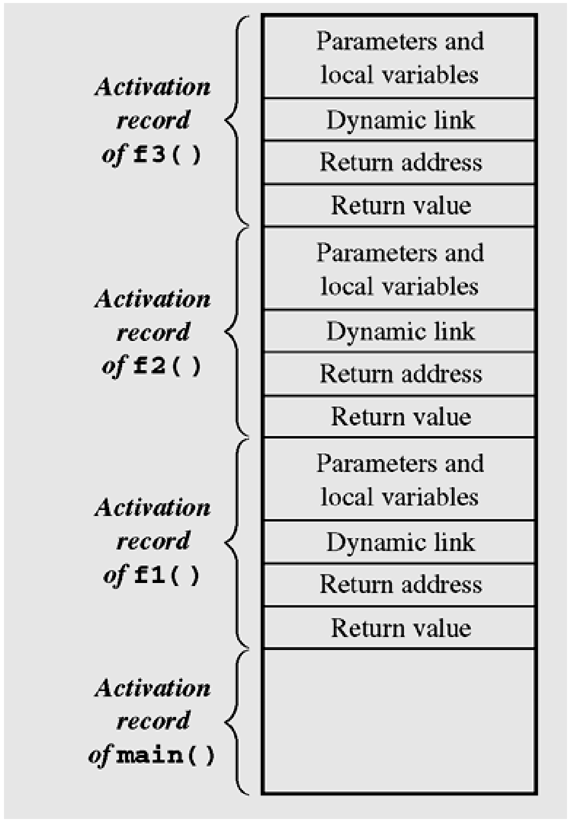

Activation record of f3( )
Parameters and local variables
Dynamic link
Return address
Return value
Activation record of f2( )
Parameters and local variables
Dynamic link
Return address
Return value
Activation record of f1( )
Parameters and local variables
Dynamic link
Return address
Return value
Activation record of main( )

Figure 5-1 Contents of the run-time stack when `main()` calls method `f1()`, `f1()` calls `f2()`, and `f2()` calls `f3()`

Tracing of Recursive Methods

- A recursive method may be traced using the recursion tree it generates.
- Example1: Consider the recursive method f defined below. Draw the recursive tree generated by the call f("KFU", 2) and hence determine the number of activation records generated by the call and the output of the following program:

```
public class MyRecursion3 {
   public static void main(String[] args){
      f("KFU", 2);
   }

   public static void f(String s, int index){
      if (index >= 0) {
         System.out.print(s.charAt(index));
         f(s, index - 1);
         System.out.print(s.charAt(index));
          f(s, index - 1);
        }
   }
}
```

Tracing of Recursive Methods

```
f("KFU", 2)
```

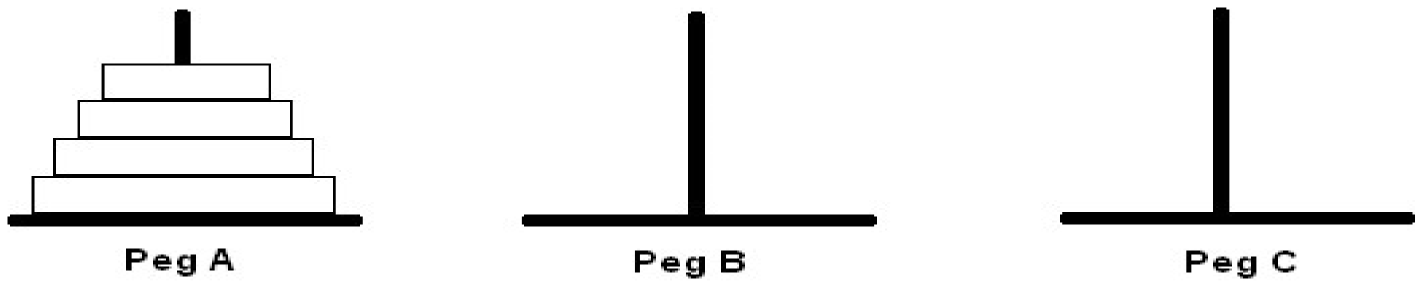
Tracing of Recursive Methods

- Example2: The Towers of Hanoi problem:
  - A total of n disks are arranged on a peg A from the largest to the smallest; such that the smallest is at the top. Two empty pegs B and C are provided.
  - It is required to move the **n** disks from peg A to peg C under the following restrictions:
    - Only one disk may be moved at a time.
    - A larger disk must not be placed on a smaller disk.
    - In the process, any of the three pegs may be used as temporary storage.
  - Suppose we can solve the problem for **n – 1** disks. Then to solve for **n** disks use the following algorithm:

    Move **n – 1** disks from peg A to peg B

    Move the nth disk from peg A to peg C

    Move **n – 1** disks from peg B to peg C



Tracing of Recursive Methods

- This translates to the Java method hanoi given below:

```
import java.io.*;
public class TowersOfHanoi{
   public static void main(String[] args) throws IOException {
      BufferedReader stdin =
           new BufferedReader(new InputStreamReader(System.in));
      System.out.print("Enter the value of n: " );
      int n = Integer.parseInt(stdin.readLine());
      hanoi(n, 'A', 'C', 'B');
   }

   public static void hanoi(int n, char from, char to, char temp){
      if (n == 1)
         System.out.println(from + " --------> " + to);
      else{
         hanoi(n - 1, from, temp, to);
         System.out.println(from + " --------> " + to);
         hanoi(n - 1, temp, to, from);
      }
   }
}
```

Tracing of Recursive Methods

- Draw the recursion tree of the method hanoi for n = = 3 and hence determine the output of the above program.

```
hanoi(3, 'A', 'C', 'B')
```

Tail and Non-Tail Recursive Methods

- A method is tail recursive if in each of its recursive cases it executes one recursive call and if there are no pending operations after that call.
- Example 1:

```
public static void f1(int n){
    System.out.print(n + "  ");
    if(n > 0)
        f1(n - 1);
}
```

- Example 2:

```
public static void f3(int n){
    if(n > 6){
        System.out.print(2*n + "  ");
        f3(n - 2);
    } else if(n > 0){
        System.out.print(n + "  ");
        f3(n - 1);
    }
}
```

Tail and Non-Tail Recursive Methods

- Example of non-tail recursive methods:
- Example 1:

```
public static void f4(int n){
   if (n > 0)
      f4(n - 1);
   System.out.print(n + "  ");
}
```

  - After each recursive call there is a pending System.out.print(n + " ") operation.

- Example 2:

```
long factorial(int x) {
    if (x == 0)
      return 1;
    else
      return x * factorial(x - 1);
}
```

  - After each recursive call there is a pending * operation.

Non-Tail Recursion Example

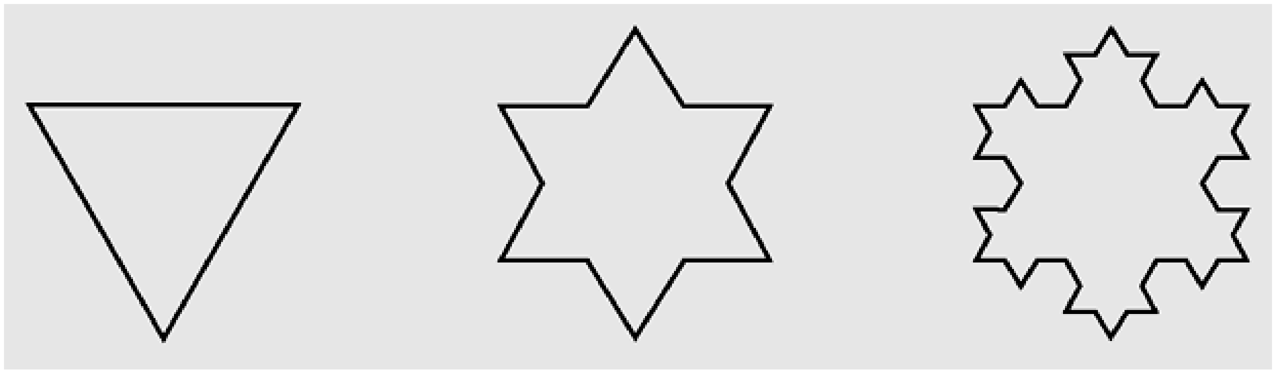


Figure 5-4 Examples of von Koch snowflakes

Non-Tail Recursion Example

1. Divide an interval *side* into three even parts
2. Move one-third of *side* in the direction specified by *angle*
3. Turn to the right 60° (i.e., turn –60°) and go forward one-third of *side*
4. Turn to the left 120° and proceed forward one-third of *side*
5. Turn right 60° and again draw a line one-third of *side* long

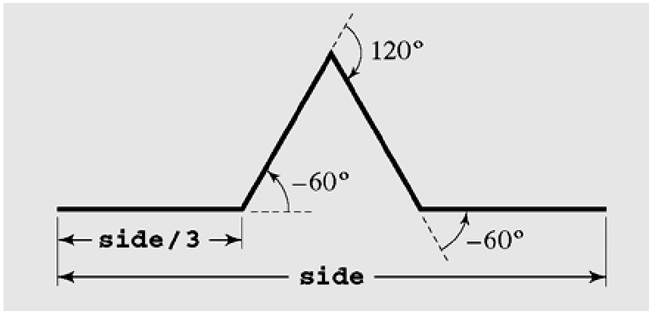


Figure 5-5 The process of drawing four sides of one segment of the von Koch snowflake

Non-Tail Recursion Example

```
drawFourLines (side, level)
    if (level = 0)
       draw a line;
    else
       drawFourLines(side/3, level-1);
       turn left 60°;
       drawFourLines(side/3, level-1);
       turn right 120°;
       drawFourLines(side/3, level-1);
       turn left 60°;
       drawFourLines(side/3, level-1);
```

Non-Tail Recursion Example

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class vonKoch extends JFrame implements ActionListener {
    public vonKoch() {
        super("von Koch snowflake");
        JButton draw  = new JButton("draw");
        lvl = new TextField("4",3);
        len = new TextField("200",3);
        lvl.addActionListener(this);
        len.addActionListener(this);
        draw.addActionListener(this);
```

Figure 5-6 Recursive implementation of the von Koch snowflake

Non-Tail Recursion Example

```
        Container cp = getContentPane();
        cp.setLayout(new FlowLayout());
        cp.add(new JLabel("level"));
        cp.add(lvl);
        cp.add(new JLabel("side"));
        cp.add(len);
        cp.add(draw);
        panel.setBackground(Color.pink);
        panel.setForeground(Color.white);
        panel.setPreferredSize(new Dimension(600,400));
        cp.add(panel);
        setSize(700,500);
        cp.setBackground(Color.red);
        setVisible(true);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }
```

Recursive implementation of the von Koch snowflake(cont.)

Non-Tail Recursion Example

```
static final long serialVersionUID = 123;
private TextField lvl, len;
private MyPanel panel = new MyPanel();
private double angle;
private Point currPt, pt = new Point();
private void right(double x) {
    angle += x;
}
private void left (double x) {
    angle -= x;
}
```

Recursive implementation of the von Koch snowflake(cont.)

Non-Tail Recursion Example

```
private void drawFourLines(double side, int level, Graphics g) {
    if (level == 0) {
         // arguments to sin() and cos() must be angles given in radians,
         // thus, the angles given in degrees must be multiplied by PI/180;
         pt.x = ((int)(Math.cos(angle*Math.PI/180)*side)) + currPt.x;
         pt.y = ((int)(Math.sin(angle*Math.PI/180)*side)) + currPt.y;
         g.drawLine(currPt.x, currPt.y, pt.x, pt.y);
         currPt.x = pt.x;
         currPt.y = pt.y;
    }
   else {
         drawFourLines(side/3.0,level-1,g);
         left (60);
         drawFourLines(side/3.0,level-1,g);
         right(120);
         drawFourLines(side/3.0,level-1,g);
         left (60);
         drawFourLines(side/3.0,level-1,g);
    }
}
```

Recursive implementation of the von Koch snowflake(cont.)

Non-Tail Recursion Example

```
 public void actionPerformed(ActionEvent e) {
        panel.repaint();
    }
    class MyPanel extends JPanel {
        static final long serialVersionUID = 124;
        public void paintComponent(Graphics g) {
                 super.paintComponent(g);
                 int level = Integer.parseInt(lvl.getText().trim());
                 double side = Double.parseDouble(len.getText().trim());
                 currPt = new Point(200,150);
                 angle = 0;
                 for (int i = 1; i <= 3; i++) {
                 drawFourLines(side,level,g);
                 right(120);
            }
        }
    }
    static public void main(String[] a) {
        new vonKoch();
    }
}
```

Recursive implementation of the von Koch snowflake(cont.)

Why tail recursion?

- It is desirable to have tail-recursive methods, because:
  a. The amount of information that gets stored during computation is independent of the number of recursive calls.
  b. Some compilers can produce optimized code that replaces tail recursion by iteration
     i. In general, an iterative version of a method will execute more efficiently in terms of time and space than a recursive version.
     ii. This is because the overhead involved in entering and exiting a function in terms of stack I/O is avoided in iterative version.
     iii. Sometimes we are forced to use iteration because stack cannot handle enough activation records - Example: power(2, 5000))
  c. Tail recursion is important in languages like Prolog and Functional languages like Clean, Haskell, Miranda, and SML that do not have explicit loop constructs (loops are simulated by recursion).

Direct and Indirect Recursive Methods

- A method is *directly* recursive if it contains an explicit call to itself.

```
long factorial (int x) {
   if (x == 0)
      return 1;
   else
      return x * factorial (x – 1);
}
```

- A method x is *indirectly* recursive if it contains a call to another method which in turn calls x. They are also known as *mutually recursive* methods:

```
public static boolean isEven(int n) {
   if (n==0)
       return true;
    else
        return(isOdd(n-1));
}

public static boolean isOdd(int n) {
  return (! isEven(n));
}
```

Direct and Indirect Recursive Methods

```
receive(buffer)
    while buffer is not filled up
        if information is still incoming
            get a character and store it in buffer;
        else exit();
        decode(buffer);

decode(buffer)
    decode information in buffer;
    store(buffer);

store(buffer)
    transfer information from buffer to file;
    receive(buffer);
```

Direct and Indirect Recursive Methods

- **Another example of mutually recursive methods:**

$$\sin(x) = \sin(\frac{x}{3})\frac{(3-\tan^2(\frac{x}{3}))}{(1+\tan^2(\frac{x}{3}))}$$

$$\sin(x) \approx x - \frac{x^3}{6} \qquad \text{for small values of x}$$

$$\tan(x) = \frac{\sin(x)}{\cos(x)}$$

$$\cos(x) = 1 - \sin\left(\frac{x}{2}\right)$$

Direct and Indirect Recursive Methods

```
public static double sin(double x){
    if(x < 0.0000001)
        return x - (x*x*x)/6;
    else{
        double y = tan(x/3);
        return sin(x/3)*((3 - y*y)/(1 + y*y));
    }
}

public static double tan(double x){
    return sin(x)/cos(x);
}

public static double cos(double x){
    return 1 - sin(x/2);
}
```

Direct and Indirect Recursive Methods

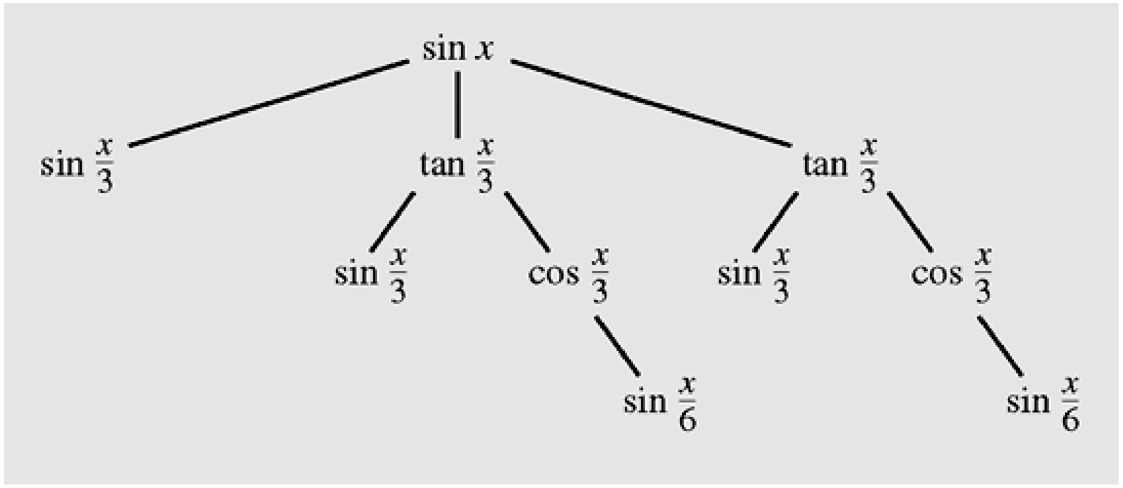


Figure 5-7 A tree of recursive calls for sin (*x*)

Nested and Non-Nested Recursive Methods

- Nested recursion occurs when a method is not only defined in terms of itself; but it is also used as one of the parameters:
- Example: The Ackerman function

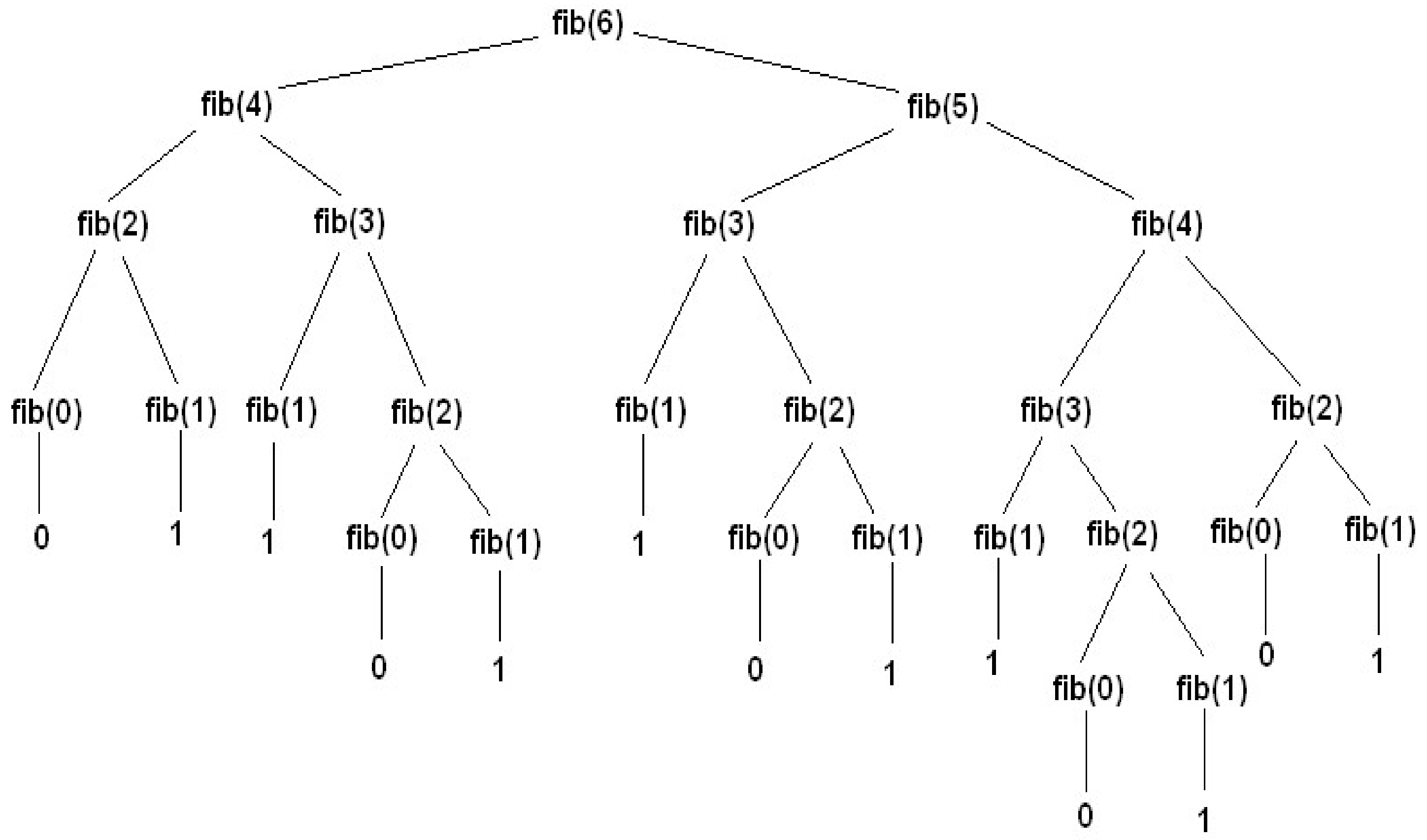
$$A(n, m) = \begin{cases} m + 1 & \text{if } n = 0 \\ A(n - 1, 1) & \text{if } n > 0,\ m = 0 \\ A(n - 1, A(n, m - 1)) & \text{otherwise} \end{cases}$$

```
public static long Ackmn(long n, long m){
    if (n == 0)
          return m + 1;
    else if (n > 0 && m == 0)
           return Ackmn(n - 1, 1);
     else
          return Ackmn(n - 1, Ackmn(n, m - 1));
}
```

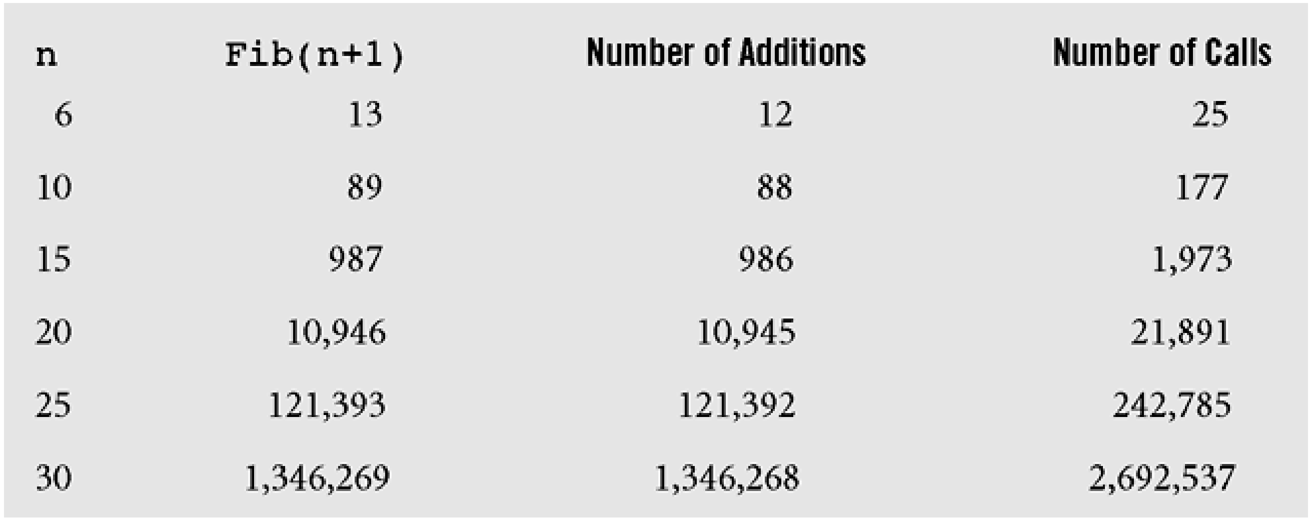
- The Ackermann function grows faster than a multiple exponential function.

Excessive Recursion

- A recursive method is excessively recursive if it repeats computations for some parameter values.
- Example: The call fib(6) results in two repetitions of f(4). This in turn results in repetitions of fib(3), fib(2), fib(1) and fib(0):



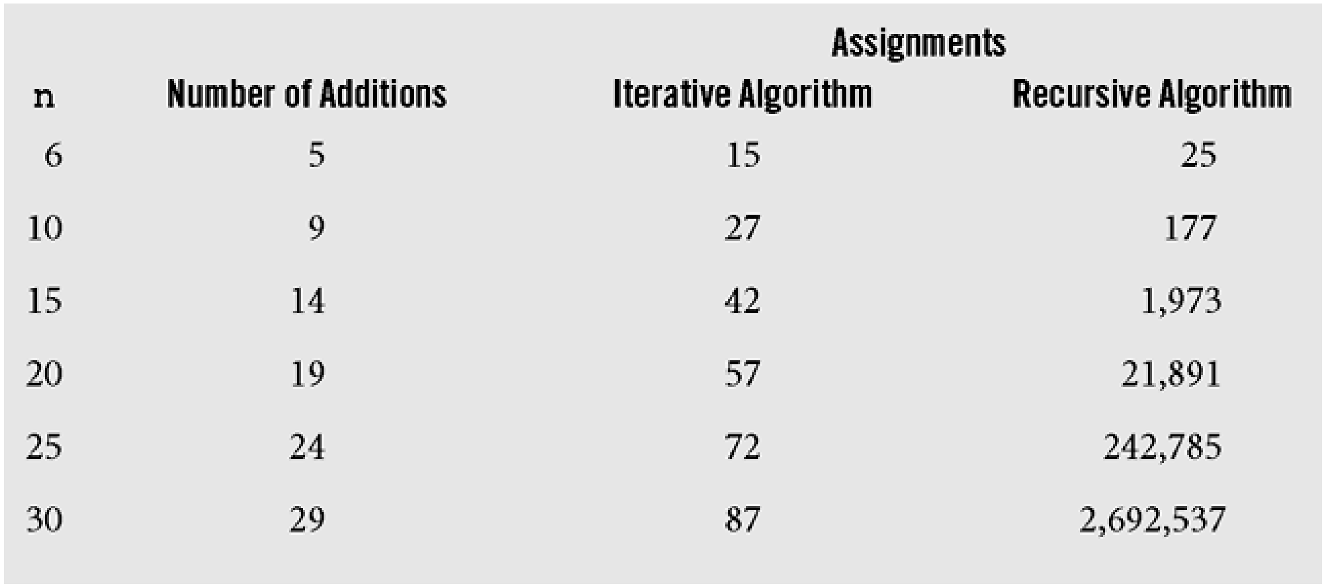
Excessive Recursion



n	Fib(n+1)	Number of Additions	Number of Calls
6	13	12	25
10	89	88	177
15	987	986	1,973
20	10,946	10,945	21,891
25	121,393	121,392	242,785
30	1,346,269	1,346,268	2,692,537

Figure 5-9 Number of addition operations and number of recursive calls to calculate Fibonacci numbers

Excessive Recursion



		Assignments	
n	Number of Additions	Iterative Algorithm	Recursive Algorithm
6	5	15	25
10	9	27	177
15	14	42	1,973
20	19	57	21,891
25	24	72	242,785
30	29	87	2,692,537

Figure 5-10 Comparison of iterative and recursive algorithms for calculating Fibonacci numbers

Final Remarks on Recursion

- Why Recursion?
- The need for Auxiliary (Helper) Methods
- Common Errors in Writing Recursive Methods:

Why Recursion?

- Usually recursive algorithms have less code, therefore algorithms can be easier to write and understand - e.g. Towers of Hanoi. However, avoid using excessively recursive algorithms even if the code is simple.

- Sometimes recursion provides a much simpler solution. Obtaining the same result using iteration requires complicated coding - e.g. Quicksort, von Koch snowflakes, Towers of Hanoi, etc.

- Recursive methods provide a very natural mechanism for processing recursive data structures. A recursive data structure is a data structure that is defined recursively – e.g. Tree.

- Functional programming languages such as Clean, FP, Haskell, Miranda, and SML do not have explicit loop constructs. In these languages looping is achieved by recursion.

Why Recursion?

- Some recursive algorithms are more efficient than equivalent iterative algorithms.
- Example:

```
public static long power1 (int x, int n) {
   long product = 1;
   for (int i = 1; i <= n; i++)
      product *= x;
   return product;
}
```

```
public static long power2 (int x, int n) {
   if (n == 1) return x;
   else if (n == 0)return 1;
   else {
      long t = power2(x , n / 2);
      if ((n % 2) == 0) return t * t;
      else return x * t * t;
   }
}
```

The need for Auxiliary (or Helper) Methods

- Auxiliary or helper methods are used for one or more of the following reasons:
  - To make recursive methods more efficient.
  - To make the user interface to a method simpler by hiding the method's initializations.
- Example 1: Consider the method:

```
public long factorial (int x){
  if (x < 0)
       throw new IllegalArgumentException("Negative argument");
  else if (x == 0)
        return 1;
  else
       return x * factorial(x - 1);
}
```

- The condition x < 0, which should be executed only once, is being executed in each recursive call. We can use a private auxiliary method to avoid this.

The need for Auxiliary (or Helper) Methods

```
public long factorial(int x){
   if (x < 0)
      throw new IllegalArgumentException("Negative argument");
   else
      return factorialAuxiliary(x);
}

private long factorialAuxiliary(int x){
   if (x == 0)
      return 1;
   else
      return x * factorialAuxiliary(x - 1);
}
```

The need for Auxiliary (or Helper) Methods

- Example 2: Consider the method:

```
public int binarySearch(int target, int[] array, int low, int high) {
   if(low > high)
      return -1;
   else {
      int middle = (low + high)/2;
      if(array[middle] == target)
         return middle;
      else if(array[middle] < target)
         return binarySearch(target, array, middle + 1, high);
      else
         return binarySearch(target, array, low, middle - 1);
   }
}
```

- The first time the method is called, the parameter low and high must be set to 0 and array.length – 1 respectively. Example:

```
int result = binarySearch (target, array, 0, array.length -1);
```

- From a user's perspective, the parameters *low* and *high* introduce an unnecessary complexity that can be avoided by using an auxiliary method:

The need for Auxiliary (or Helper) Methods

```
public int binarySearch(int target, int[] array){
   return binarySearch(target, array, 0, array.length - 1);
}

private int binarySearch(int target, int[] array, int low, int high){
   if(low > high)
      return -1;
   else{
      int middle = (low + high)/2;
      if(array[middle] == target)
         return middle;
      else if(array[middle] < target)
         return binarySearch(target, array, middle + 1, high);
      else
         return binarySearch(target, array, low, middle - 1);
   }
}
```

- A call to the method becomes simple:

```
int result = binarySearch(target, array);
```

Common Errors in Writing Recursive Methods

- The method does not call itself directly or indirectly.
- Non-terminating Recursive Methods (Infinite recursion):
  a) No base case.

```
int badFactorial(int x) {
   return x * badFactorial(x-1);
}
```

  b) The base case is never reached for some parameter values.

```
int anotherBadFactorial(int x) {
   if(x == 0)
      return 1;
   else
      return x*(x-1)*anotherBadFactorial(x -2);
      // When x is odd, we never reach the base case!!
}
```

Common Errors in Writing Recursive Methods

- Post increment and decrement operators must not be used since the update will not occur until AFTER the method call - infinite recursion.

```
public static int sumArray (int[ ] x, int index) {
   if (index == x.length)return 0;
   else
      return x[index] + sumArray (x, index++);
}
```

- Local variables must not be used to accumulate the result of a recursive method. Each recursive call has its own copy of local variables.

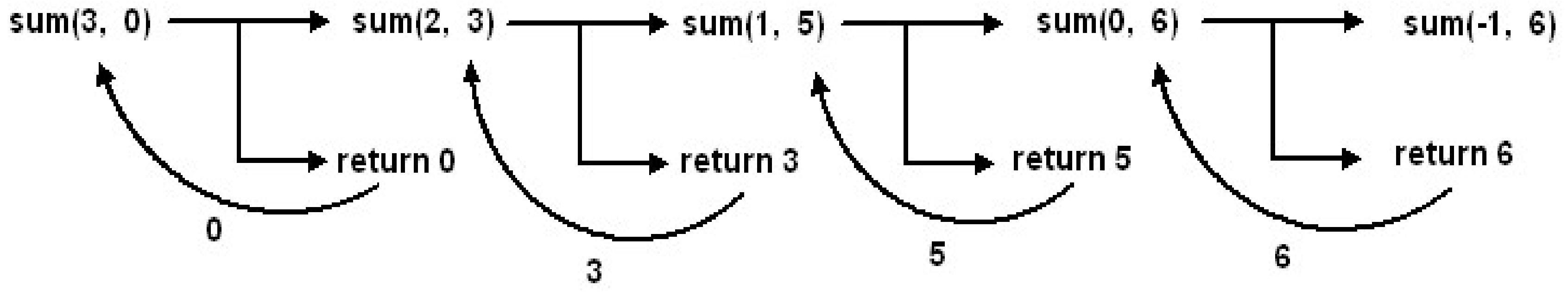
```
public static int sumArray (int[ ] x, int index) {
   int sum = 0;
   if (index == x.length)return sum;
   else {
      sum += x[index];
      return sumArray(x,index + 1);
   }
}
```

Common Errors in Writing Recursive Methods

- Wrong placement of **return** statement.
- Consider the following method that is supposed to calculate the sum of the first **n** integers:

```
public static int sum (int n, int result) {
    if (n >= 0)
        sum(n - 1, n + result);
    return result;
}
```

- When **result** is initialized to **0**, the method returns **0** for whatever value of the parameter **n**. The result returned is that of the final **return** statement to be executed. Example: A trace of the call sum(3, 0) is:

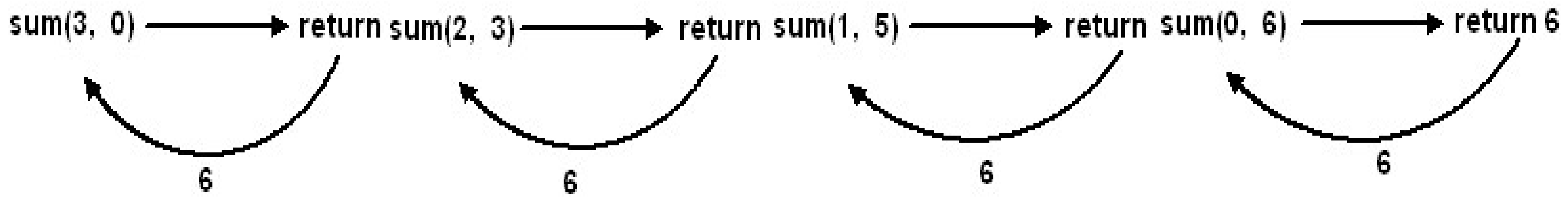


Common Errors in Writing Recursive Methods

- A correct version of the method is:

```
public static int sum(int n, int result){
   if (n == 0)
      return result;
   else
      return sum(n-1, n + result);
}
```

- Example: A trace of the call sum(3, 0) is:



Common Errors in Writing Recursive Methods

- The use of instance or static variables in recursive methods should be avoided.
- Although it is not an error, it is bad programming practice. These variables may be modified by code outside the method and cause the recursive method to return wrong result.

```
public class Sum{
   private int sum;

   public int sumArray(int[ ] x, int index){
      if(index == x.length)
         return sum;
      else {
         sum += x[index];
         return sumArray(x,index + 1);
      }
   }
}
```