Recursion

Dr. Aiman Hanna
Department of Computer Science & Software Engineering
Concordia University, Montreal, Canada

These slides have been extracted, modified and updated from original slides of :
Data Structures and Algorithms in Java, 5th edition. John Wiley& Sons, 2010. ISBN 978-0-470-38326-1.
Data Structures and the Java Collections Framework by William J. Collins, 3rdedition, ISBN 978-0-470-48267-4.
Both books are published by Wiley.

The Recursion Pattern

- **Recursion**: when a method calls itself
- Classic example: the factorial function:

  $$n! = 1 * 2 * 3 * \cdots * (n-1) * n$$

- Recursive definition:

$$f(n) = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot f(n-1) & \textit{else} \end{cases}$$

- As a Java method:

```
// recursive factorial function
public static int  recursiveFactorial(int n) {
    if  (n  ==  0)  return  1;                    // base case
    else return  n  *  recursiveFactorial(n- 1);  // recursive case
}
```

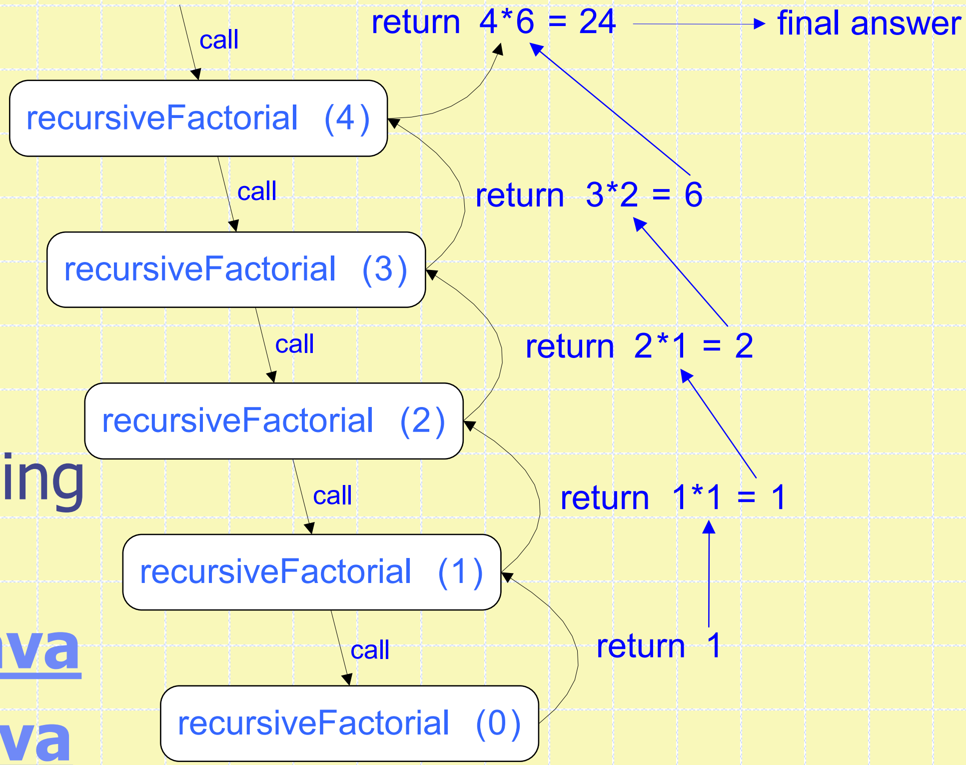
Content of a Recursive Method

- Base case(s)
  - Also referred to as stopping cases. These are the cases where the method performs NO more recursive calls.
  - There should be at least one base case.
  - Every possible chain of recursive calls must eventually reach a base case.
- Recursive calls
  - Calls to the method itself.
  - Each recursive call should be defined so that it makes progress towards a base case.

Visualizing Recursion

- Recursion trace
  - A box for each recursive call
  - An arrow from each caller to callee
  - An arrow from each callee to caller showing return value
  - See Recursion1.java
    Recursion2.java

- Example



Recursion & The Stack

- A running Java program maintains a private memory area called the ***stack***, which is used to keep track of the methods as they are invoked.

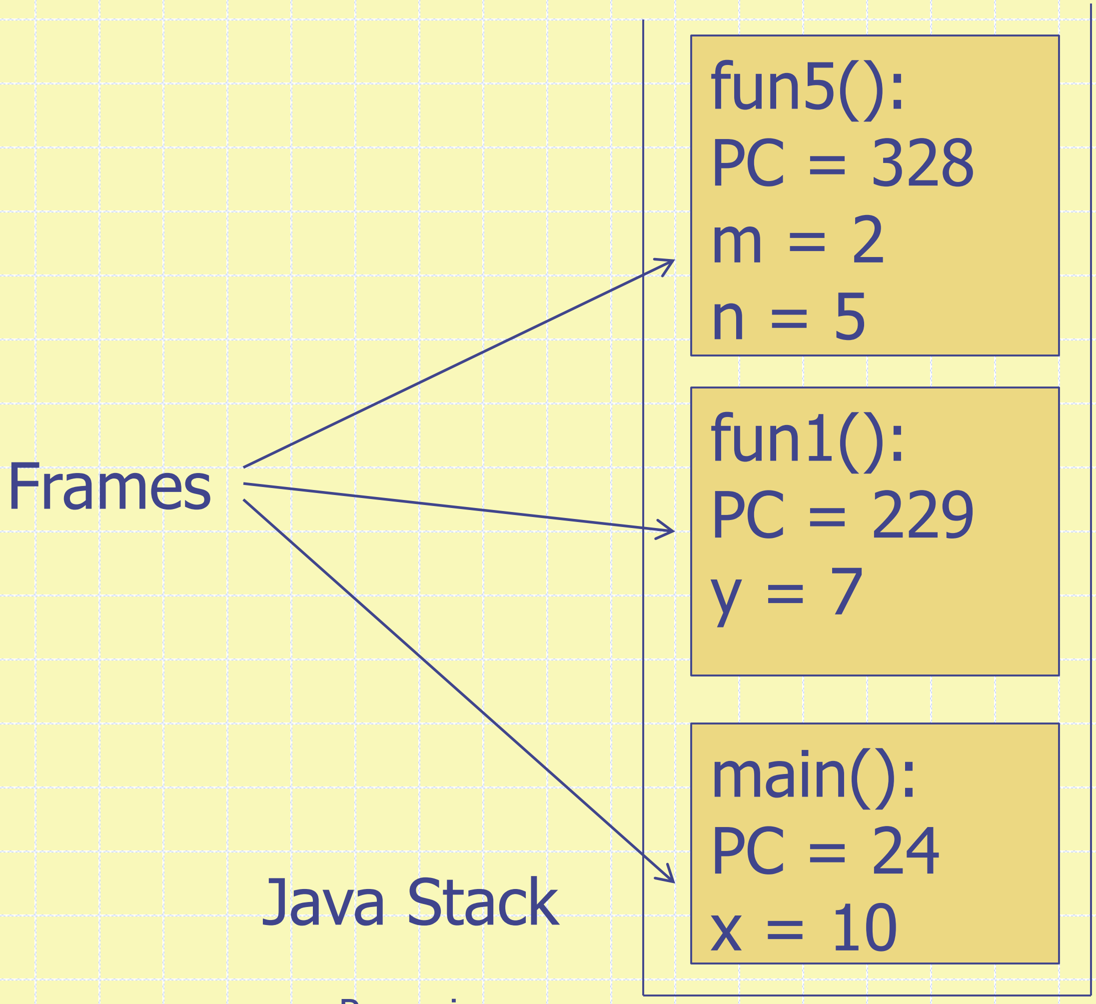
- Whenever a method is invoked, its information (parameters, local variables, Program Counter (PC), …) is placed as one ***frame*** into the stack.

- The frame is removed from the stack once the method returns.

Recursion & The Stack

```
main()
{ …
   fun1();
   …
}

fun1()
{ …
   fun5();
   …
}
```



Recursion & The Stack

- The *heap* is another memory area that is maintained for a running program.
- The heap is used for dynamic allocation of memory at runtime (i.e. when **new** is called to create an object).
- Usually the stack and the heap grow against each other in the memory.
- Recursion has hence the potential of overflowing the stack by quickly consuming all available space.
- ➔ **See Recursion3.java**

Linear Recursion

- Simplest form of recursion, where the method makes at most one recursive call each time it is invoked.

- Very useful when the problem is viewed in terms of first or last element, plus a remaining set that has the same structure as the original set.

- For instance, obtaining the summation of *n* values in an array can be viewed as:
  - Obtaining the sum of the first *n-1* elements plus the value of the last element;
  - If the array has only one element, then the summation is that single value, *A[0]* .

Example of Linear Recursion

Algorithm LinearSum(*A*, *n*):

Input:

An integer array *A* and an integer *n* >= 1, such that *A* has at least *n* elements

Output:

The sum of the first *n* integers in *A*

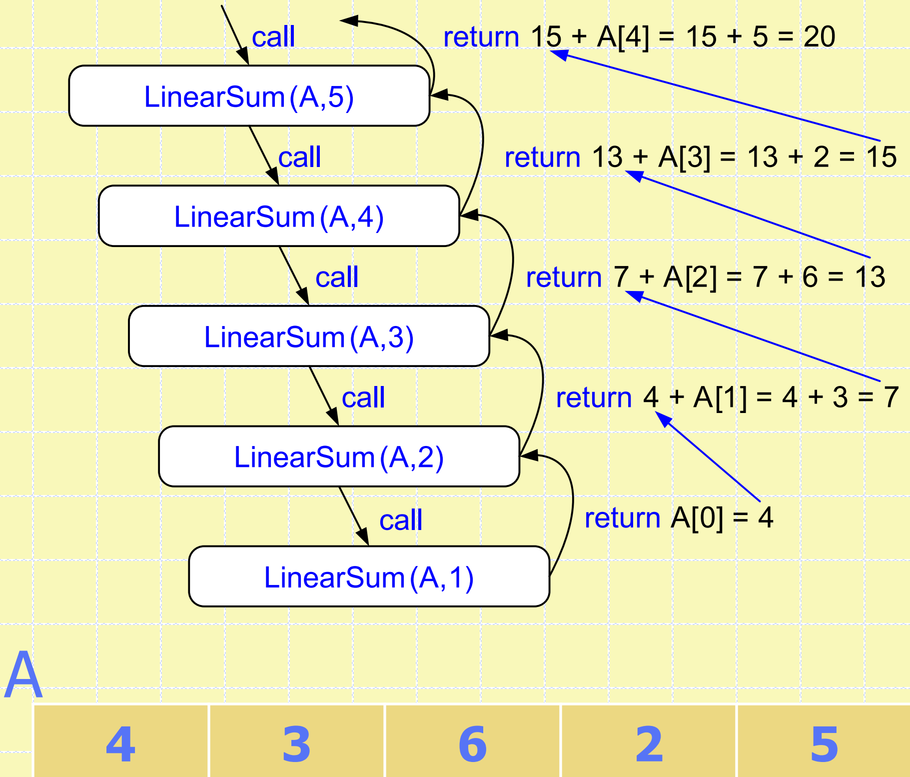
if *n* = 1 **then**

return *A*[0]

else

return LinearSum(*A*, *n* - 1) + *A*[*n* - 1]

Example recursion trace:



Example: Reversing an Array

Algorithm ReverseArray(A, i, j):

Input: An array A and nonnegative integer indices i and j

Output: The reversal of the elements in A starting at index i and ending at j

if $i < j$ **then**

Swap $A[i]$ and $A[j]$

ReverseArray(A, $i + 1$, $j - 1$)

return

Defining Arguments for Recursion

- In creating recursive methods, it is important to define the methods in ways that facilitate recursion.
- This sometimes requires additional parameters to be passed to the method.
- For example, we defined the array reversal method as ReverseArray(A, i, j), not ReverseArray(A).

Example: Computing Powers

- The power function, $p(x,n)=x^n$, can be defined recursively:

$$p(x,n) = \begin{cases} 1 & \text{if } n = 0 \\ x \cdot p(x, n-1) & \text{else} \end{cases}$$

- This leads to a power function that runs in $O(n)$ time (for we make n recursive calls).
- However, can we do better than this?

Recursive Squaring

- We can derive a more efficient linearly recursive algorithm by using repeated squaring:

$$p(x,n)=\begin{cases} 1 & \text{if } x=0 \\ x\cdot p(x,(n-1)/2)^2 & \text{if } x>0 \text{ is odd} \\ p(x,n/2)^2 & \text{if } x>0 \text{ is even} \end{cases}$$

- For example,

$2^4 = 2^{(4/2)2} = (2^{4/2})^2 = \mathbf{(2^2)^2} = 4^2 = 16$

$2^5 = 2^{1+(4/2)2} = 2(2^{4/2})^2 = \mathbf{2(2^2)^2} = 2(4^2) = 32$

$2^6 = 2^{(6/2)2} = (2^{6/2})^2 = \mathbf{(2^3)^2} = 8^2 = 64$

$2^7 = 2^{1+(6/2)2} = 2(2^{6/2})^2 = \mathbf{2(2^3)^2} = 2(8^2) = 128.$

Recursive Squaring Method

Algorithm Power(x, n):
  Input: A number x and integer $n = 0$
  Output: The value x^n
 if $n = 0$ **then**
  return 1
 if n is odd **then**
  $y = \text{Power}(x, (n - 1)/2)$
  return $x \cdot y \cdot y$
 else
  $y = \text{Power}(x, n/2)$
  return $y \cdot y$

Analysis

Algorithm Power(x, n):
Input: A number x and integer $n = 0$
Output: The value x^n
if $n = 0$ **then**
return 1
if n is odd **then**
$y = $ Power(x, $(n - 1)/2$)
return $x \cdot y \cdot y$
else
$y = $ Power(x, $n/2$)
return $y \cdot y$

Each time we make a recursive call we halve the value of n; hence, we make log n recursive calls. That is, this method runs in O(log n) time.

It is important that we use a variable twice here rather than calling the method twice.

Tail Recursion

- Tail recursion occurs when a linearly recursive method makes its recursive call as its last step; as in the array reversal method.

- Such methods can be easily converted to non-recursive methods (which saves on some resources).
- Example:

Algorithm IterativeReverseArray(A, i, j):

Input: An array A and nonnegative integer indices i and j

Output: The reversal of the elements in A starting at index i and ending at j

while $i < j$ **do**

Swap $A[i]$ and $A[j]$

$i = i + 1$

$j = j - 1$

return

➔See Factorial.java

Binary Recursion

- Binary recursion occurs whenever there are **two**, and exactly two, recursive calls for each non-base case.
- Applicable, for instance, when attempting to solve two different halves of some problem.
- Example: Calculating the summation of an n array elements, can be done by:
  - Recursively summing the elements in the first half;
  - Recursively summing the elements in the second half;
  - Adding the two values.

Example: Summing Array Elements

- Example: Summing n consecutive elements of an array, starting from a given index i, using binary recursion

Algorithm *binarySum*(A, i, n)
  Input An array A and integers i and n
  Output The sum of the n element of A, starting at index i

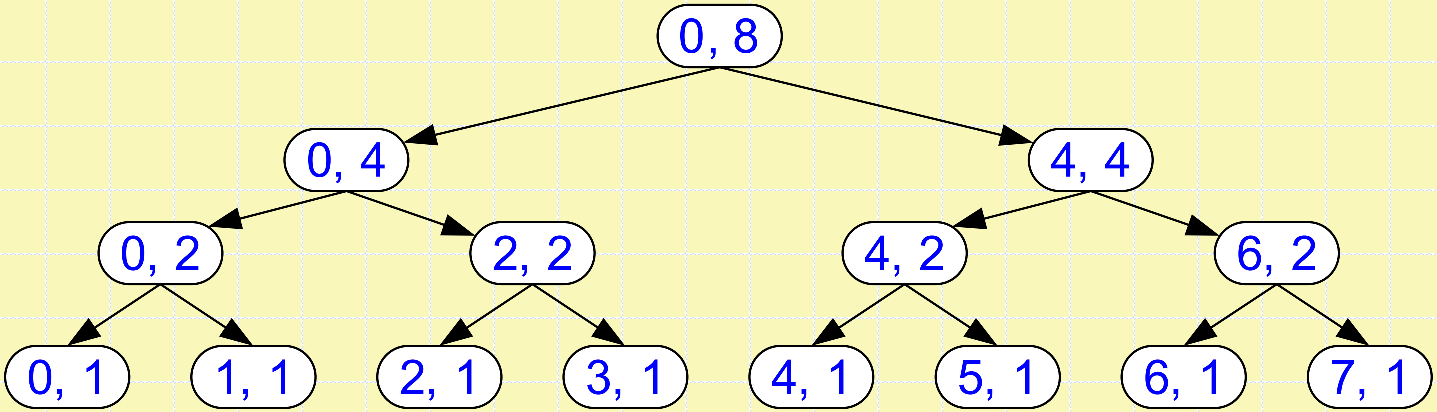
  if $n = 1$ **then**
    return $A[i]$

  return *binarySum*(A, i, $\lceil n/2 \rceil$) + *binarySum*(A, $i + \lceil n/2 \rceil$, $\lfloor n/2 \rfloor$)

Example: Summing Array Elements

- The following provides an example of a *binarySum(0, 8)* trace, where each box indicates the starting index and the number of elements to sum.

- **Analysis:** In every half, the call will be made $n-1$ times, resulting in a total of $2n - 1$ calls ➔ $O(n)$.
- However, it should be noted that there is a maximum of $1 + \log_2 n$ active calls at any point of time, which improves space utilization as we discuss later.



Example: Fibonacci Numbers

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

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, \ldots$$

- By definition, the first two Fibonacci numbers are 0 and 1, and each subsequent number is the sum of the previous two.

- Fibonacci numbers can be defined recursively as:

$$F_0 = 0$$
$$F_1 = 1$$
$$F_i = F_{i-1} + F_{i-2} \quad \text{for } i > 1.$$

Example: Fibonacci Numbers

- Recursive algorithm (first attempt):

Algorithm ***binaryFib*(*k*)**:

Input: Nonnegative integer k

Output: The k^{th} Fibonacci number F_k

if $k \leq 1$ **then**

return k

else

return ***binaryFib(k - 1) + binaryFib(k - 2)***

Example: Fibonacci Numbers

- **Analysis:** Let n_k be the number of recursive calls (notice that this is not the value) by ***binaryFib***(***k***)
  - $n_0 = 1$
  - $n_1 = 1$
  - $n_2 = n_1 + n_0 + 1 = 1 + 1 + 1 = 3$
  - $n_3 = n_2 + n_1 + 1 = 3 + 1 + 1 = 5$
  - $n_4 = n_3 + n_2 + 1 = 5 + 3 + 1 = 9$
  - $n_5 = n_4 + n_3 + 1 = 9 + 5 + 1 = 15$
  - $n_6 = n_5 + n_4 + 1 = 15 + 9 + 1 = 25$
  - $n_7 = n_6 + n_5 + 1 = 25 + 15 + 1 = 41$
  - $n_8 = n_7 + n_6 + 1 = 41 + 25 + 1 = 67$.
- Note that n_k at least doubles every other time.
- In fact, $n_k > 2^{k/2}$. It is exponential!

Example: Fibonacci Numbers

- The main problem with ***binaryFib***(***k***) approach is that the computation of Fibonacci numbers is really a linearly recursive problem, in spite of its look where F_k depends on F_{k-1} and F_{k-2}.

- The problem is hence not a good candidate for binary recursion.

- We should use linear recursion instead.

A Better Fibonacci Algorithm

- Use linear recursion instead

```
Algorithm linearFibonacci(k):
  Input: A nonnegative integer k
  Output: Pair of Fibonacci numbers (F_k , F_k-1)
 if k = 1 then
        return (k, 0)
 else
        (i, j) = linearFibonacci(k - 1)
  return (i +j, i)
```

// notice that the values are retuned (however, not both are displayed)

- ***linearFibonacci*** makes ***k-1*** recursive calls, so total calls is ***k***.

➔ **See LinearFib.java & BinaryFibStack.java**

A Better Fibonacci Algorithm

For instance (note: Fib is short for ***linearFibonacci***),

- Fib(2) ➔ (i+j, i) is (1,1) ➔ Will be displaying 1
- Fib(3) ➔ (i+j, i) is (2,1) ➔ Will be displaying 2
- Fib(4) ➔ (i+j, i) is (3,2) ➔ Will be displaying 3
- Fib(5) ➔ (i+j, i) is (5,3) ➔ Will be displaying 5
- Fib(6) ➔ (i+j, i) is (8,5) ➔ Will be displaying 8
- Fib(7) ➔ (i+j, i) is (13,8) ➔ Will be displaying 13
- Fib(8) ➔ (i+j, i) is (21,13) ➔ Will be displaying 21
- Fib(9) ➔ (i+j, i) is (34,21) ➔ Will be displaying 34
- :
- Fib(12) ➔ (i+j, i) is (144,89) ➔ Will be displaying 144

Binary Recursion Another Example

- The English Ruler:
  - Print the ticks and numbers like an English ruler

```
---- 0
-
--
-
---
-
--
-
---- 1
-
--
-
---
-
--
-
---- 2
```

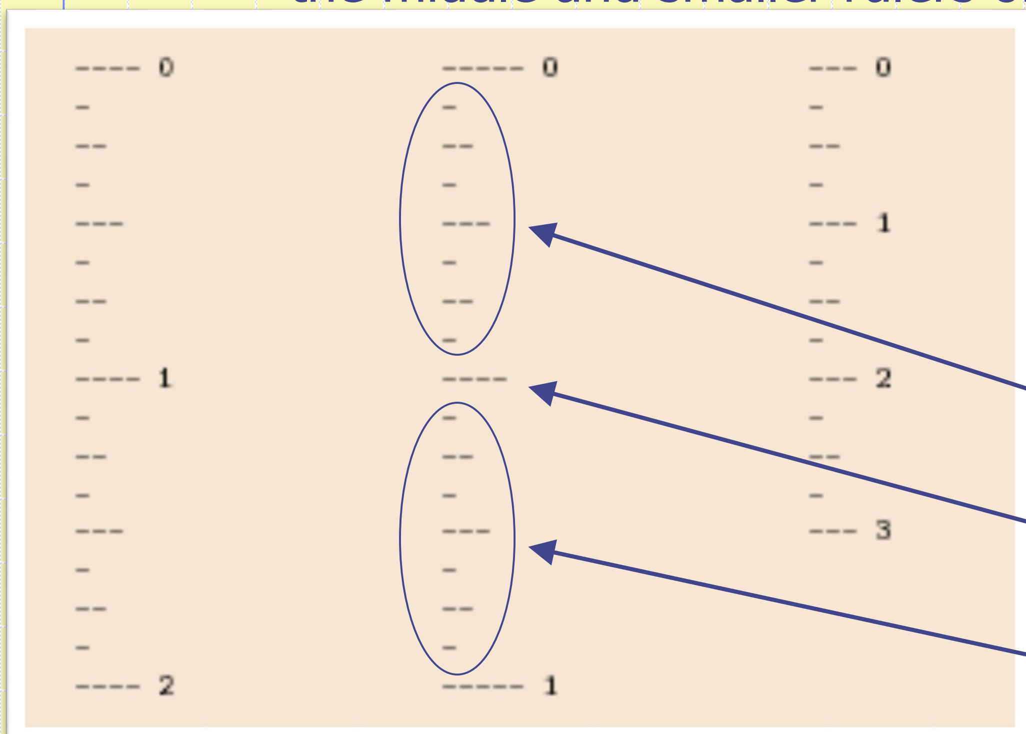
```
----- 0
-
--
-
---
-
--
-
----
-
--
-
---
-
--
-
----- 1
```

```
--- 0
-
--
-
--- 1
-
--
-
--- 2
-
--
-
--- 3
```

Slide by Matt Stallmann included with permission.

Example: The English Ruler

drawTicks(length)
Input: length of a 'tick'
Output: ruler with tick of the given length in the middle and smaller rulers on either side



drawTicks(length)

if(length > 0) then

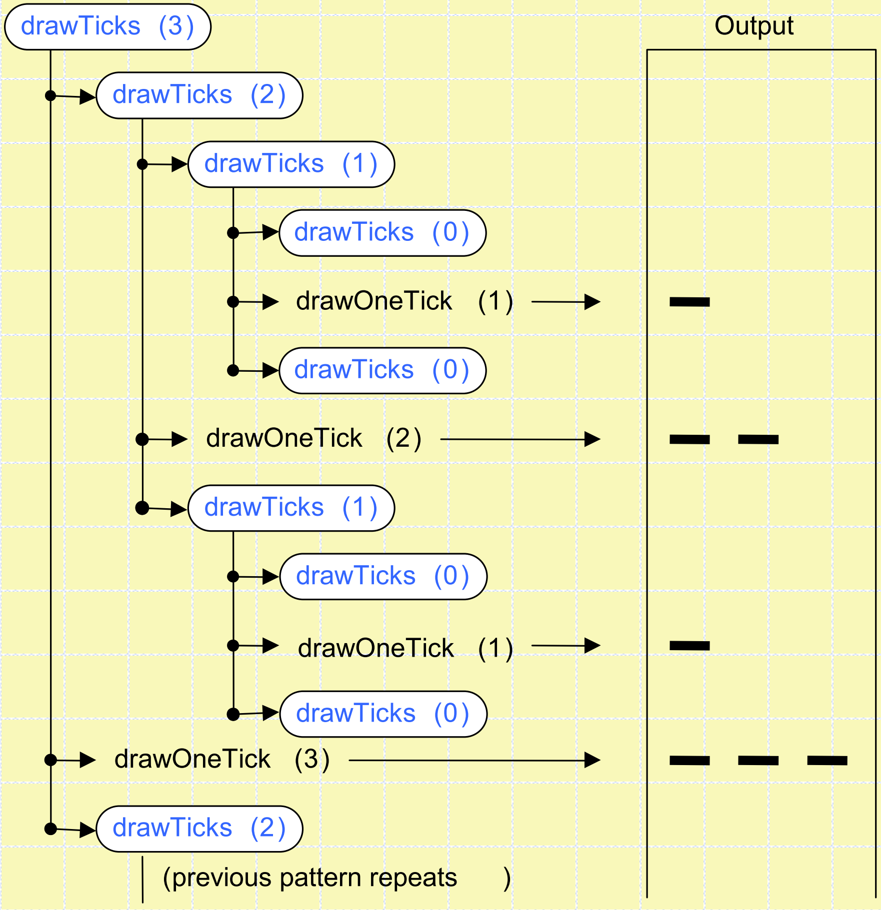
drawTicks(length – 1)

draw tick of the given length

drawTicks(length – 1)

Recursive Drawing Method

- The drawing method is based on the following recursive definition
- An interval with a central tick length L $\geq$1 consists of:
  - An interval with a central tick length L–1
  - An single tick of length L
  - An interval with a central tick length L–1



Java Implementation (1)

```
// draw ruler
public static void drawRuler(int nInches, int majorLength) {
    drawOneTick(majorLength, 0);            // draw tick 0 and its label
    for (int i = 1; i <= nInches; i++){
        drawTicks(majorLength- 1);          // draw ticks for this inch
        drawOneTick(majorLength, i);        // draw tick i and its label
    }
}

// draw ticks of given length
public static void drawTicks(int tickLength) {
    if (tickLength > 0) {                   // stop when length drops to 0
        drawTicks(tickLength- 1);           // recursively draw left ticks
        drawOneTick(tickLength);            // draw center tick
        drawTicks(tickLength- 1);           // recursively draw right ticks
    }
}
```

Note the two recursive calls

Java Implementation (2)

```
// draw one tick;
// passing last parameter as -1 will draw ticks without a label
public static void drawOneTick(int tickLength, int tickLabel) {
    for (int i = 0; i < tickLength; i++)
        System.out.print("-");
    if (tickLabel >= 0) System.out.print(" " + tickLabel);
    System.out.print("\n");
}
```

Multiple Recursion

- Multiple recursion:
  - Makes potentially many recursive calls
  - Not just one or two
- Motivating example:
  - Coping folders (directories)
  - Finding enumerations of sequence
    - *{a,b,c} : abc, acb, bac, bca, cab, cba*

Example of Multiple Recursion

Algorithm CopyFolder(folder):
Input: A directory folder, which possibly includes files and subfolders
Output: A copy of the given folder with all its files and subfolders

for all files in folder **do**
    copy file
for all subfolder in folder **do**
    copyfolder(subfolder) // this line is where recursion happens

Example of Multiple Recursion

```
Algorithm PuzzleSolve(k,S,U):
 Input: Integer k, sequence S, and set U (universe of elements to
    test)
 Output:  Enumeration of all k-length extensions to S using elements
    in U without repetitions
 for all e  in U do
    Remove e from U       {e is now being used}
    Add e to the end of S
    if k = 1 then
           Test whether S is a configuration that solves the puzzle
           if S solves the puzzle then
                  return "Solution found: " S
    else
           PuzzleSolve(k - 1, S,U)
    Add e back to U        {e is now unused}
    Remove e from the end of S
```

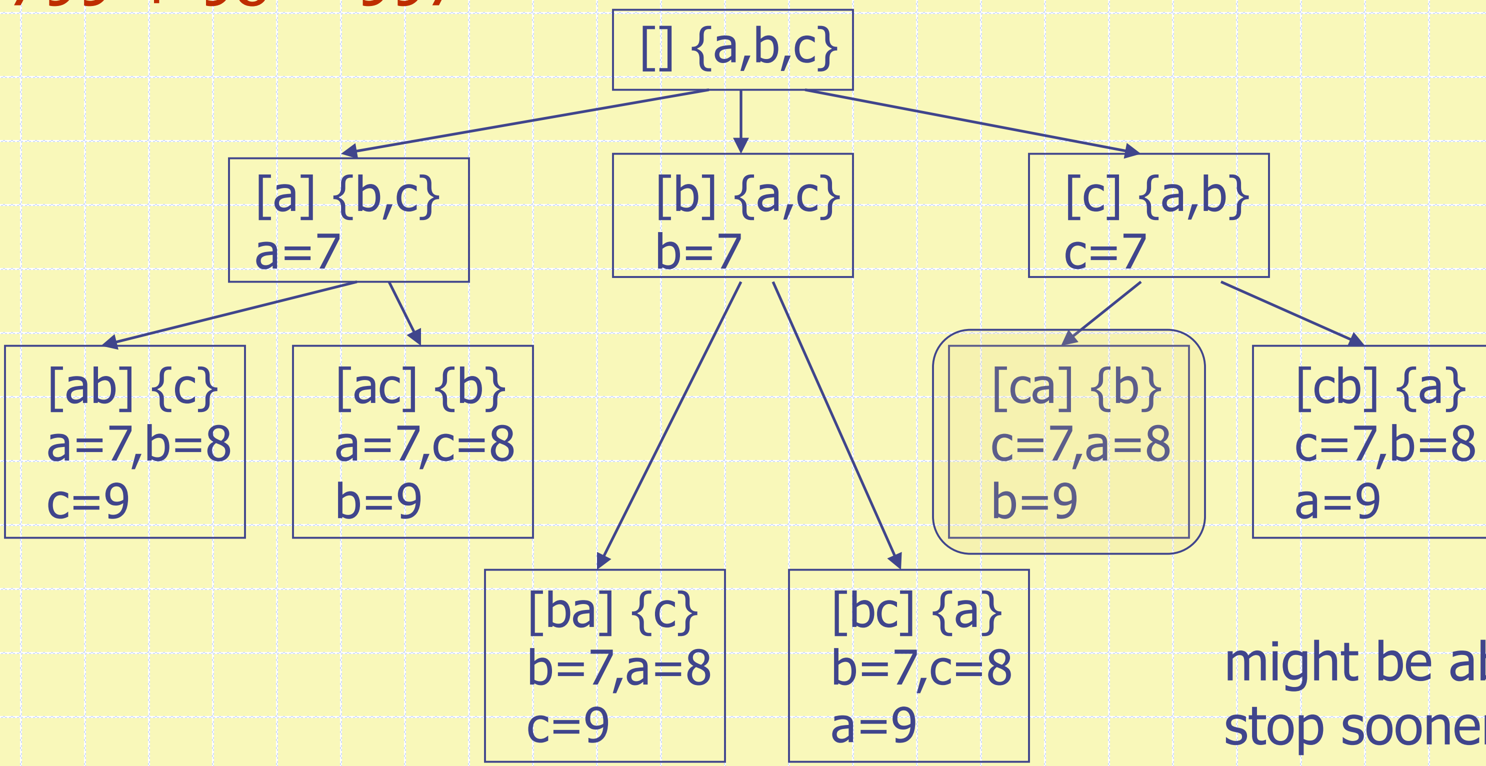
Slide by Matt Stallmann included with permission.

Example of Multiple Recursion

cbb + ba = abc

799 + 98 = 997

a,b,c stand for 7,8,9; not necessarily in that order



might be able to stop sooner

Visualizing PuzzleSolve

- Notice that the number of concurrently active calls can still be limited with multiple recursion.
- For instance, the number of active calls of CopyFolder depends on how many nested subfolders may exist at a time and not on the total number of subfolders in the directory.