4CCS1DST – Data Structures

Lecture 1:

Arrangements for the module

Recursion (5/e: § 3.5; 6/e: Ch. 5)

Introduction

- **Lecturer**:
  - Costas Iliopoulos

    e-mail:csi@kcl.ac.uk

- **Lecture 1**:
  - Aims and learning outcomes of the module
  - Arrangements for the module
  - Programming with recursion

Aims of the course

- To introduce the concepts of data structures and algorithms.
- To introduce a number of concrete data structures and algorithms.
- To develop implementations in Java of selected data structures.
- To highlight the importance of computational efficiency of data structures and algorithms.
- To introduce methods for analysing computational efficiency of algorithms.
- To develop further Java programming skills.

Learning Outcomes

Students successfully completing this course should:

- understand the concept of data structures and their relation to algorithms
- know a number of widely used data structures (list, stack, queue, tree, priority queue, map, hash table, dictionary, search tree) and their possible representations and implementations in Java
- be able to use the above data structures in simple applications
- know the basic searching algorithms (linear and binary search) and the main sorting algorithms (including: insertion sort, mergesort, quicksort, heap sort, bucket-sort), and the efficiency of these algorithms
- be able to develop their own simple data structures (design and implement in Java)
- be able to analyse computational efficiency of simple algorithms and data structures (in terms of asymptotic running time)

Programming in 1st Year Modules

Fundamentals of Programming (PPA I)	Building Applications (PPA II)	Data Structures and Efficient Algorithms (DST)
• **Small programs**	• **Larger Applications** • Structuring code • Debugging • Documentation	• **Some programming techniques** • **Modularization of programs by encapsulating data structures**
• **Interfaces for basic data collections (data structures)**		• Specifications of various data structures • **Implementations of data structures** • **Performance of data structures** • Where and how data structures are used
• Individual programming concepts	• Pre-existing code • Libraries and frameworks	• Interfaces for data structures
• Focus on technology	• Focus on user needs • User-friendly interfaces	• **Focus on efficient implementations** • **Analysis of the running times**

2nd year: Practical Experiences of Programming (5CCS2PEP); Software Engineering Group Project (5CCS2SEG)

- Focus on practical programming:
  - Use of programming environments; other programming language paradigms
  - Practice and Experience through lab work and larger programming projects

DST: not a "programming module" but "module with programming".

Lectures, tutorials, etc.

- Live Tutorials for lectures

  Monday 11-13

See your individual timetable for times/dates

Assessment: Coursework & Exam

- Two coursework assignments (dates already on KEATS)
- A weekly assignment
- The **coursework** (the two assignments together) contributes **0%** to the final mark for this course
- The **exam** (100% M.C.Q.) will contribute **100%** **of the final mark**

- To pass the module: the final mark at least 40.

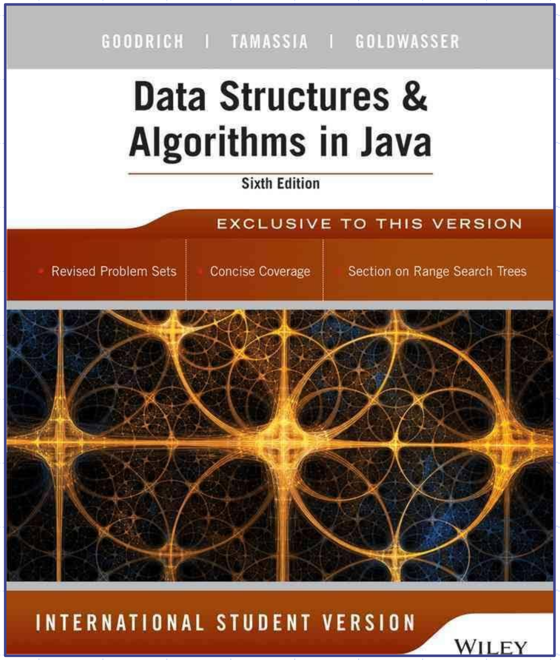
KEATS

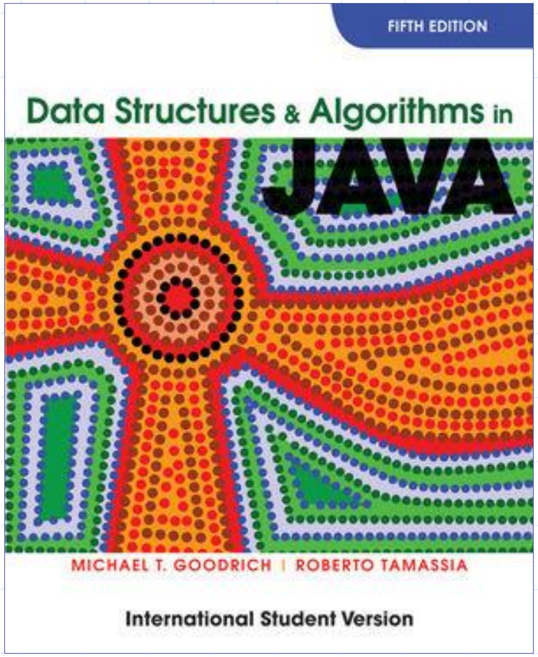
- Lecture notes (slides), coursework assignments Weekly assigments, solutions, coursework results will be on KEATS.

- Coursework /Weekly Assignments submission on KEATS

Textbook

- Michael T. Goodrich, Roberto Tamassia, Michael H. Goldwasser, ***Data structures and algorithms in Java***, (International Student Version), 6th Edition, Wiley 2014.
- Michael T. Goodrich, Roberto Tamassia, ***Data Structures and Algorithms in Java***, 5th Edition Wiley 2010.





Other useful books

Data structures and algorithms:

- M.A. Weiss, *Data Structures and Algorithm Analysis in Java*, Pearson 2012.

Java:

- H. Schildt, *Java: a beginner's guide*
- C. Horstmann, *Computing concepts with Java*.
- H.M. Deitel, P.J. Deitel, *Java: How to Program*.

Programming with Recursion

The Recursion Pattern

- Recursion: when a method calls itself
- Examples
- A classic example: the factorial function:

  $n! = 1 \cdot 2 \cdot 3 \cdot \cdots \cdot (n-1) \cdot n \qquad (\text{let } 0! = 1)$

- $0! = 1$
  $1! = 1$
  $2! = 2 \quad (= 1 \cdot 2)$
  $3! = 6 \quad (= 1 \cdot 2 \cdot 3)$
  $4! = 24 \quad (= 1 \cdot 2 \cdot 3 \cdot 4)$
  …
  $10! = 3628800$ (the number of permutations of 10 elements)

Computing factorial function

- Recursive definition:

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

- How to use it:

$$f(0) = 1; \qquad f(1) = 1 \cdot f(0) = 1 \cdot 1 = 1;$$

$$f(4) = 4 \cdot f(3) = 4 \cdot (3 \cdot f(2)) = 4 \cdot (3 \cdot (2 \cdot f(1)))$$
$$= 4 \cdot (3 \cdot (2 \cdot (1 \cdot f(0)))) = 4 \cdot (3 \cdot (2 \cdot (1 \cdot 1))) = 24$$

- As a Java method:

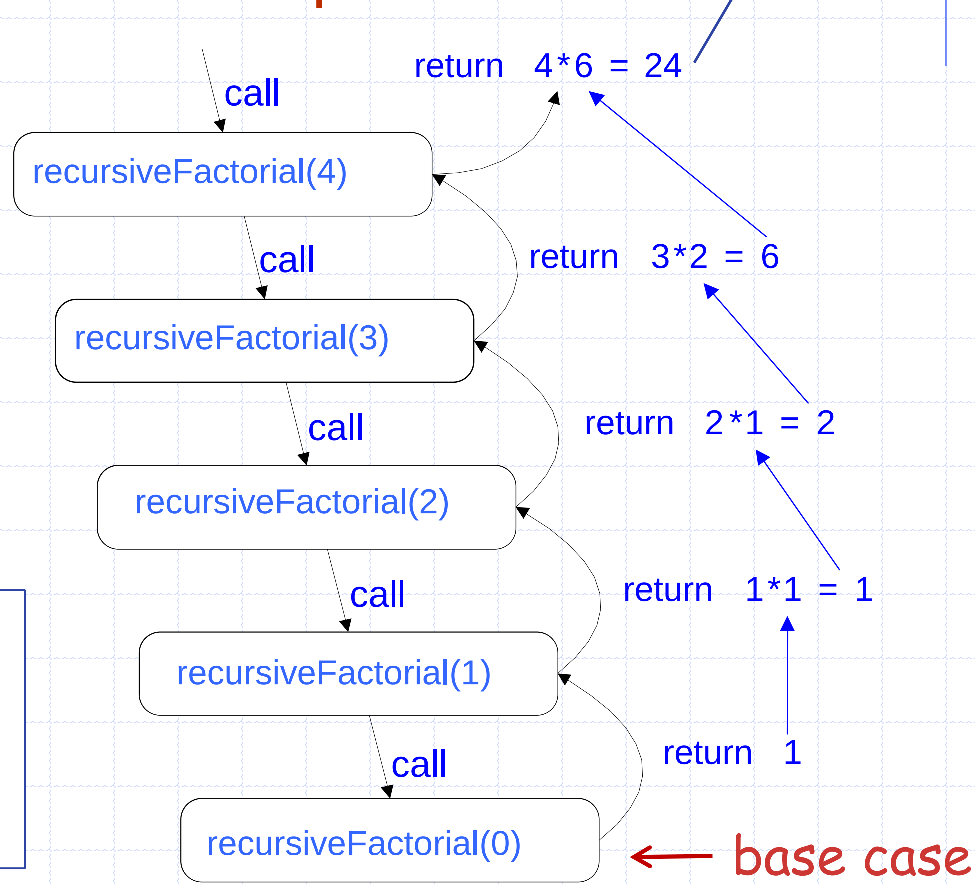
```
// recursive factorial function
public static int  recursiveFactorial(int n) {
    if  (n  <=  0)  return  1;                          // base case
    else return  n  *  recursiveFactorial(n - 1);    // recursive case
}
```

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

```
public static int  recursiveFactorial(int n)
{
   if (n <= 0) return 1;
   else return n * recursiveFactorial(n - 1);
}
```

- Example



Factorial: an iterative solution

```
public static int iterativeFactorial (int n) {
    int result = 1;

    for ( int i = 1; i <= n ; i ++ ) {
            result = result * i ;
    }

    return result ;
}
```

- We could do without Recursion, but it is a useful tool in problem solving and in programming languages.

Content of a Recursive Method

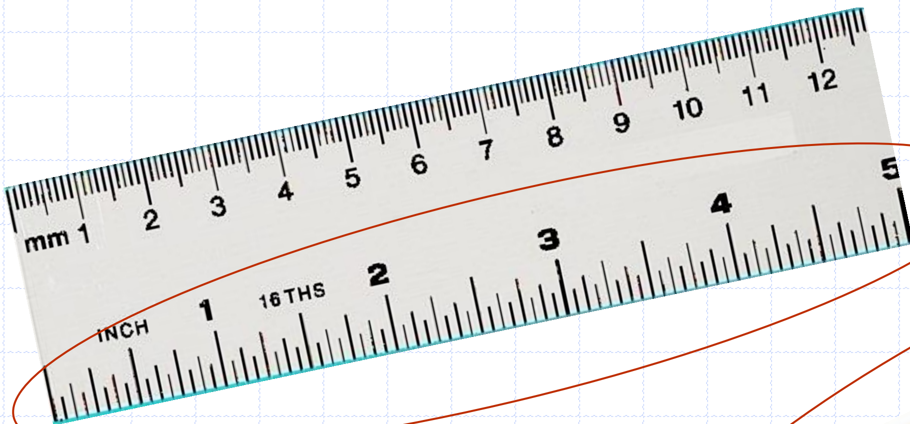
- Base case(s)
  - Values of the input variables for which we perform no recursive calls are called base cases (there should be at least one base case).
  - Every possible chain of recursive calls must eventually reach a base case.
- Recursive calls
  - Calls to the current method.
  - Each recursive call should be defined in such a way that it makes progress towards a base case.

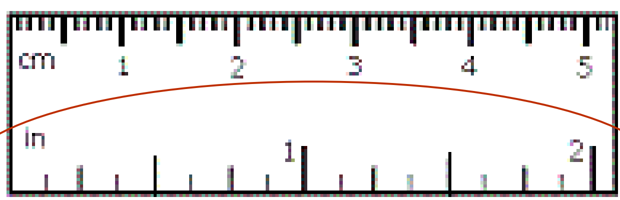
Is anything wrong with the following recursive methods?

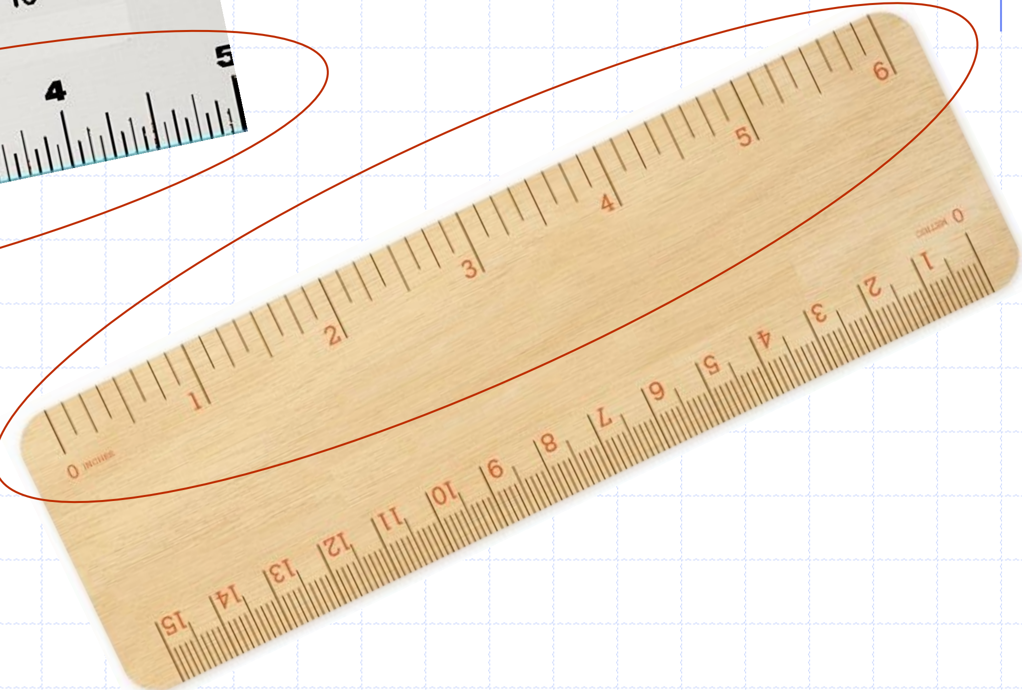
```
public static int  recursiveFactorial2 (int n) {
    if  (n  ==  0)  return  1;
    else return  n  *  recursiveFactorial2 (n - 1);
}
```

```
public static int  recursiveFactorial3 (int n) {
    if  (n  ==  0)  return  1;
    else if  (n >= 1)  return  n  *  recursiveFactorial3 (n - 1);
}
```

Example: English Ruler





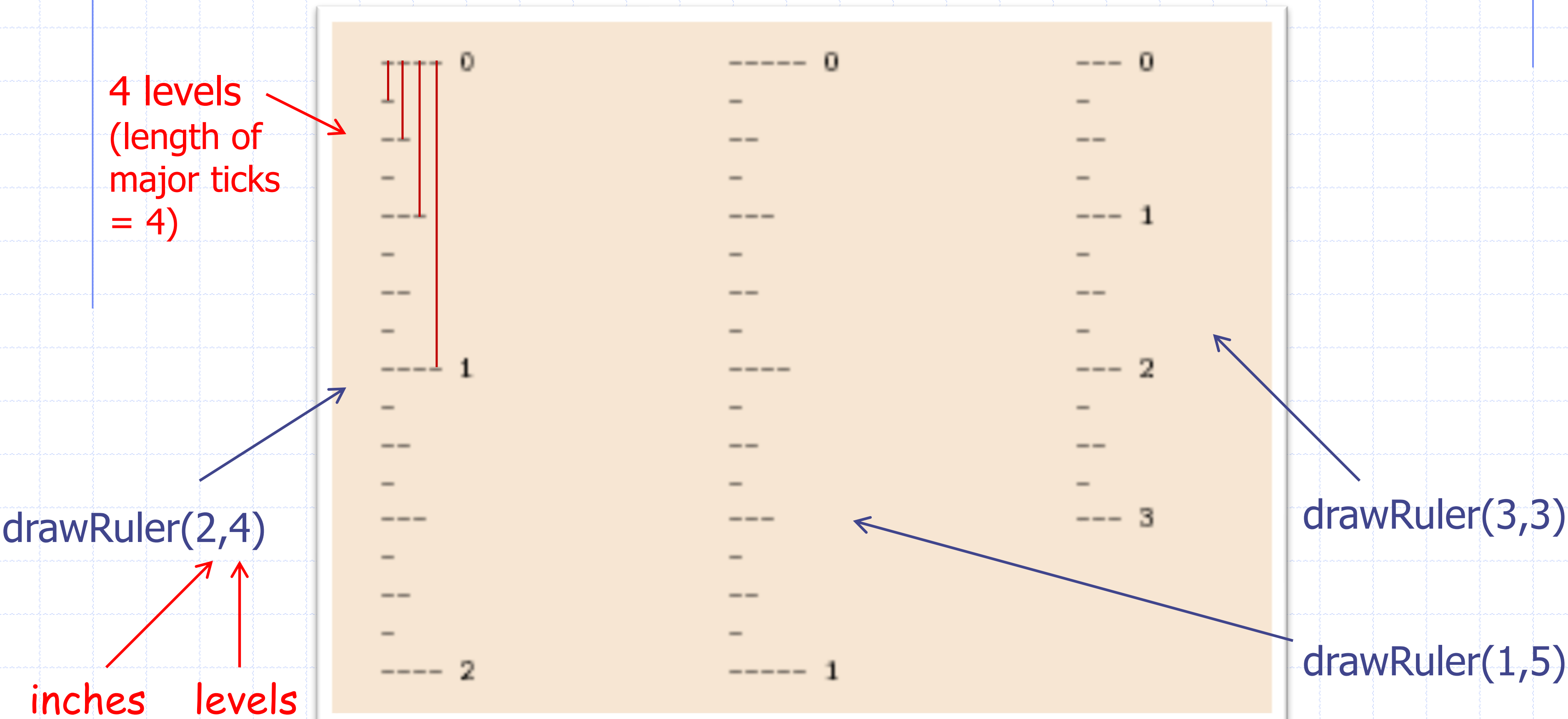


- Print the ticks and numbers like on an English ruler

major tick every inch,
minor ticks at ½ inch, ¼ inch, etc.

Example: English Ruler

- Print the ticks and numbers like on an English ruler:



Slide by Matt Stallmann included with permission.

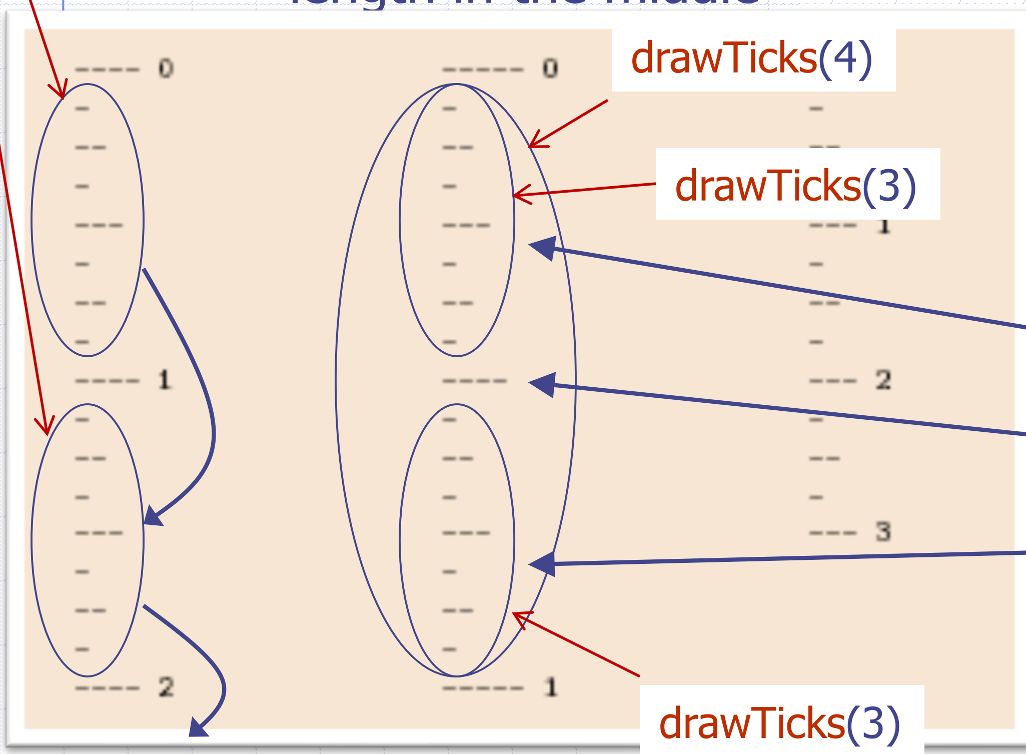
Using Recursion

drawTicks(length)

Input: length of the middle 'tick'

Output: the pattern with a tick of the given length in the middle

drawTicks(3)



drawTicks(length):

if(length > 0) then

drawTicks(length – 1)

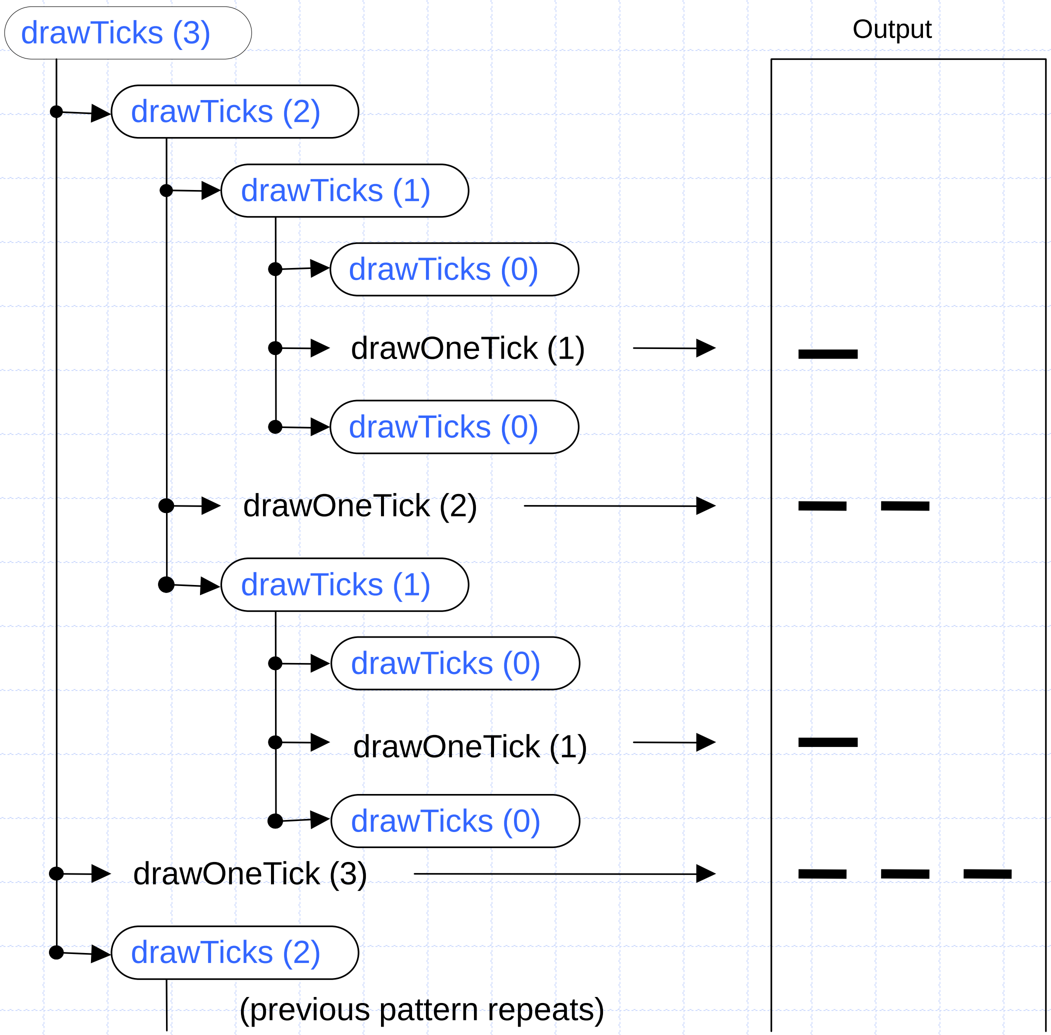
drawOneTick(length)

drawTicks(length – 1)

Recursive Drawing Method

- The drawing method is based on the following recursive definition:
- Interval with a central tick length L ≥1 consists of:
  - Interval with a central tick length L–1
  - Single tick of length L
  - Interval with a central tick length L–1

```
drawTicks(L):
  if( L > 0 ) then
    drawTicks( L – 1 )
    drawOneTick( L )
    drawTicks( L – 1 )
```



Java Implementation (1)

```
package  lecture1;
public class Ruler {

    // draw ruler  (one loop; each iteration draws one inch)
    public static void  drawRuler(int  nInches,  int  majorLength)  { … }

    // draw inner ticks for one unit at level tickLength+1 (recursive method)
    public static void  drawTicks(int  tickLength)  { … }

    // draw one tick
    public static void  drawOneTick(int  tickLength)  { … }

    // draw one tick with a label
    public static void  drawOneTick(int  tickLength,  int  tickLabel)  { … }

    // test
    public static void main(String args[ ]) {  Ruler.drawRuler(2, 4);  }
}
```

Java Implementation (2)

```
// draw ruler  (one loop; each iteration draws one inch)
public static void  drawRuler(int nIns, int majorL)  {
    drawOneTick(majorL,  0);           // draw tick 0 and its label
    for  (int i =  1; i <=  nIns;  i++)  {
        drawTicks(majorL - 1);         // draw ticks for this inch
        drawOneTick(majorL,  i);       // draw tick i and its label
    }
}
// draw ticks of given length for one inch (recursive method)
public static void  drawTicks(int  tickLength)  {
    if  (tickLength  >  0)  {          // stop when length drops to
        drawTicks(tickLength - 1);     // recursively draw left ticks
        drawOneTick(tickLength);       // draw center tick
        drawTicks(tickLength - 1);     // recursively draw right ticks
    }
}
```

drawRuler(3,4):

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

---- 2

---- 3
```

Java Implementation (3)

```
// draw one tick with 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");
}

// draw a tick with no label
public static void  drawOneTick(int  tickLength)  {
    drawOneTick(tickLength,  - 1);
}
```

drawRuler(3,4):

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

---- 2

---- 3
```

Linear Recursion

- Test for base cases
  - Begin by testing for a set of base cases (there should be at least one).
  - Every possible chain of recursive calls must eventually reach a base case, and the handling of each base case should not use recursion.
- Recur once
  - Perform a single recursive call
  - This step may have a test that decides which of several possible recursive calls to make, but it should ultimately make just one of these calls
  - Define each possible recursive call so that it makes progress towards a base case.

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

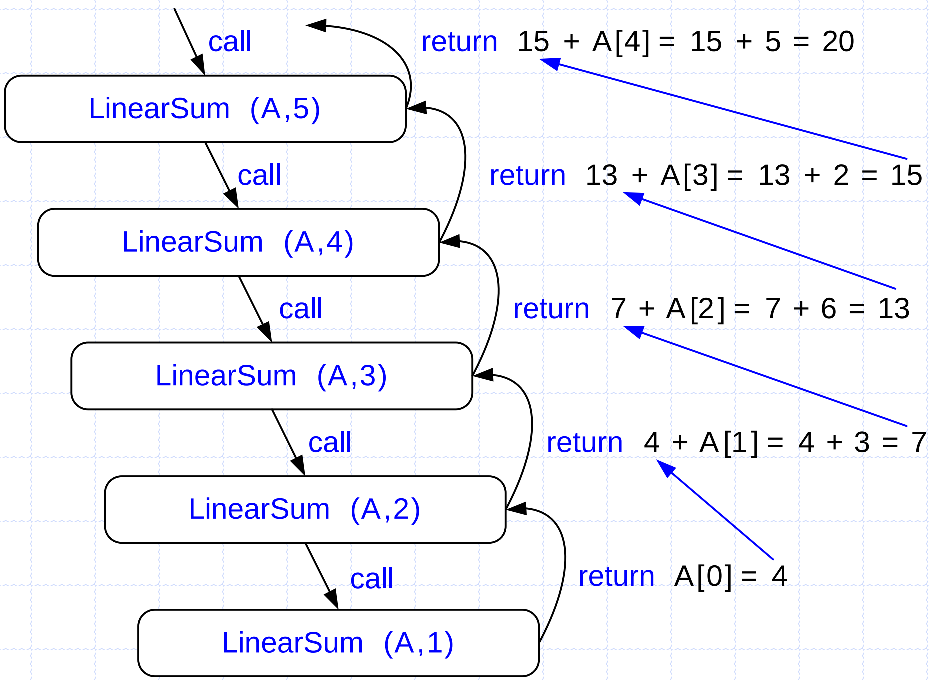
Output:

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:

A = { 4,3,6,2,5}, n = 5



Reversing an Array

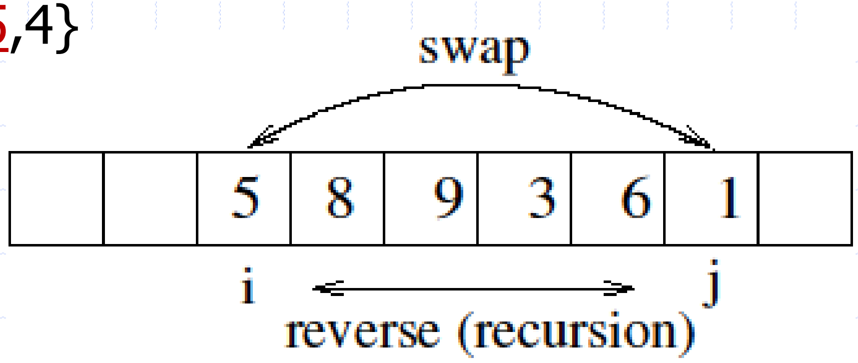
Algorithm ReverseArray(*A*, *i*, *j*):

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*

Example: *A* = {7,2,5,8,9,3,6,1,4}; ReverseArray(*A*,2,7);

A = {7,2,1,6,3,9,8,5,4}



Method:

if *i* < *j* **then**
  Swap *A*[*i*] and *A*[*j*]
  ReverseArray(*A*, *i* + 1, *j* - 1)
return

Defining Arguments for Recursion

- In specifying a recursive method, it is important to define the method in a way that facilitates recursion.
- This sometimes requires we define additional parameters that are passed to the method.
- For example, even if we only want to reverse whole arrays, we still define the array reversal method as ReverseArray(*A*, *i*, *j*), not ReverseArray(*A*).
- Method for reversing the whole array:

```
public static void reverseArray(Object [] A) {
        reverseArray(A, 0, A.length - 1);
}
```

Computing Powers

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

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

- p(5,3) = 5·p(5,2) = 5·(5·p(5,1)) = 5·(5·(5·p(5,0)))
  = 5·5·5·1 = 125
- This leads to a power function that runs in time *linear* in n (since we make n recursive calls; n multiplications).
- We can do better (faster) than this.

Recursive Squaring

- Example,

$2^4 = 2\cdot2\cdot2\cdot2 = 16$ (3 multiplications)

$2^4 = (2^2)^2 = 4^2 = 16$ (2 squarings)

$2^{32} = 2\cdot2\cdot2\cdot \ldots \cdot2 = 4{,}294{,}967{,}296$ (31 multiplications)

$2^{32} = (2^{16})^2 = ((((2^2)^2)^2)^2)^2 = \ldots$ (5 squarings)

$2^{14} = 2\cdot2\cdot2\cdot \ldots \cdot2 = 16{,}384$ (13 multiplications)

$2^{14} = (2^7)^2 = (2\cdot2^6)^2 = (2\cdot(2^3)^2)^2 = (2\cdot(2\cdot2^2)^2)^2 = \ldots$

(5 multiplications/squarings)

Recursive Squaring

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

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

- Example:

$$x^{14} = p(x, 14) = [\,p(x, 7)\,]^2 =$$
$$= [x \cdot [p(x, 3)]^2\,]^2 = [x \cdot [x \cdot [p(x, 1)]^2\,]^2\,]^2$$
$$= [x \cdot [x \cdot [x \cdot [p(x, 0)]^2\,]^2\,]^2\,]^2$$
$$= [x \cdot [x \cdot [x \cdot 1]^2\,]^2\,]^2$$

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 even **then**

    y = Power(x, n/2)

    return $y \cdot y$

else

    y = Power(x, (n-1) / 2)

    return $x \cdot y \cdot y$

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 even **then**
  y = Power(x , $n/2$)
  return $y \cdot y$
else
  y = Power(x , $(n - 1)/2$)
  return $x \cdot y \cdot y$

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 time proportional to **$\log_2 n$**

Ex.: 789103^{972183}

iterative multiplications: ~ 1M

via squaring: < $2\log_2(1M) \approx 40$

Who needs such computation?

It is important that we use a variable y here rather than call the method twice. That is, $y \cdot y$, not Power(x,n/2) · Power(x,n/2)

Tail Recursion

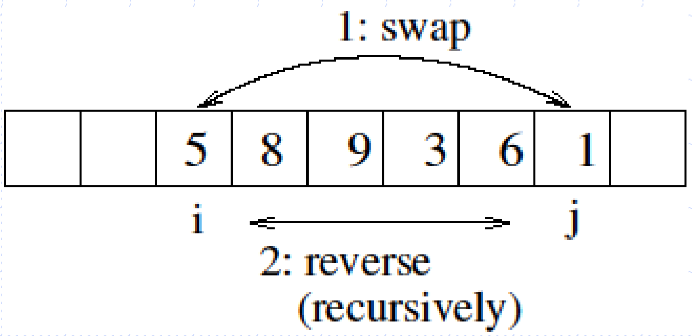
- Tail recursion occurs when a linearly recursive method makes its recursive call as its last step.
- The array-reversal method is an example.
- Such methods can be easily converted to non-recursive methods (saving on some resources).
- Example:

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



Binary Recursion

- Binary recursion occurs whenever there are **two** recursive calls for each non-base case.
- Example: the drawTicks method for drawing ticks on an English ruler.

```
public static void drawTicks(int tickLength) { // draw ticks of given length
  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
  }
}
```

two recursive calls

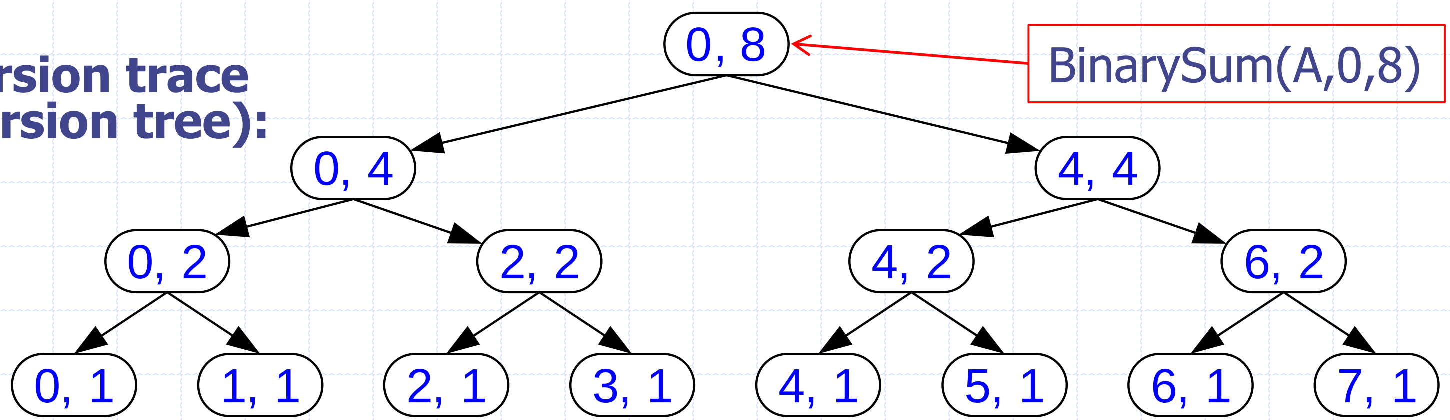
Another Binary Recusive Method

- Problem: add all numbers in an integer array A:

Algorithm BinarySum(A, i, n):
  Input: An array A and integers i >= 0 and n >= 1
  Output: The sum of the n integers in A starting at index i

if $n = 1$ **then**
  return $A[i]$
return BinarySum(A, i, $\lceil n/2 \rceil$) + BinarySum(A, $i + \lceil n/2 \rceil$, $\lfloor n/2 \rfloor$)

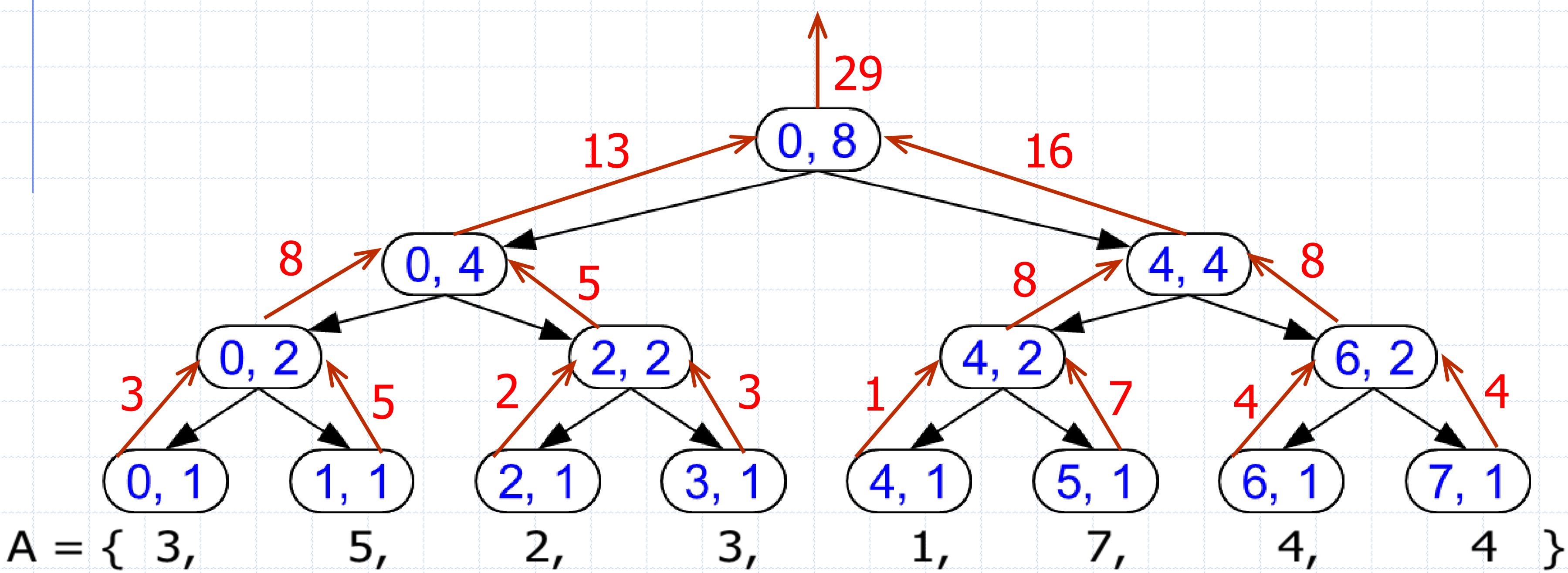
- **Recursion trace (recursion tree):**



A = { 3, 5, 2, 3, 1, 7, 4, 4 }

Recursion tree with return values

recursive calls: →
return values: →



Computing Fibonacci Numbers

- Fibonacci numbers are defined recursively:

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

- Recursive algorithm (first attempt):

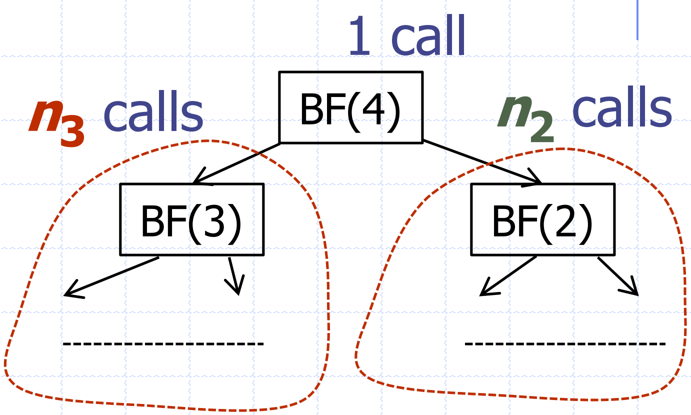
Algorithm BinaryFib(k):
  Input: Nonnegative integer k
  Output: The k-th Fibonacci number F_k

```
if k <= 1 then
    return k
else
    return BinaryFib(k - 1) + BinaryFib(k - 2)
```

$F_0 = 0$
$F_1 = 1$
$F_2 = 1$
$F_3 = 2$
$F_4 = 3$
$F_5 = 5$
$F_6 = 8$
$F_7 = 13$

Analysis

- Let n_k be the number of method calls by BinaryFib(k)
  - $n_0 = 1; \quad n_1 = 1$
  - $n_2 = n_1 + n_0 + 1 = 1 + 1 + 1 = 3$
  - $n_3 = n_2 + n_1 + 1 = 3 + 1 + 1 = 5$
  - $n_4 = \mathbf{n_3} + \mathbf{n_2} + 1 = 5 + 3 + 1 = 9$
  - $n_5 = n_4 + n_3 + 1 = 9 + 5 + 1 = 15$
  - $n_6 = n_5 + n_4 + 1 = 15 + 9 + 1 = 25$
  - $n_7 = n_6 + n_5 + 1 = 25 + 15 + 1 = 41 \qquad (n_7 \geq 2 \cdot n_5)$
  - $n_8 = n_7 + n_6 + 1 = 41 + 25 + 1 = 67 \qquad (n_8 \geq 2 \cdot n_6)$

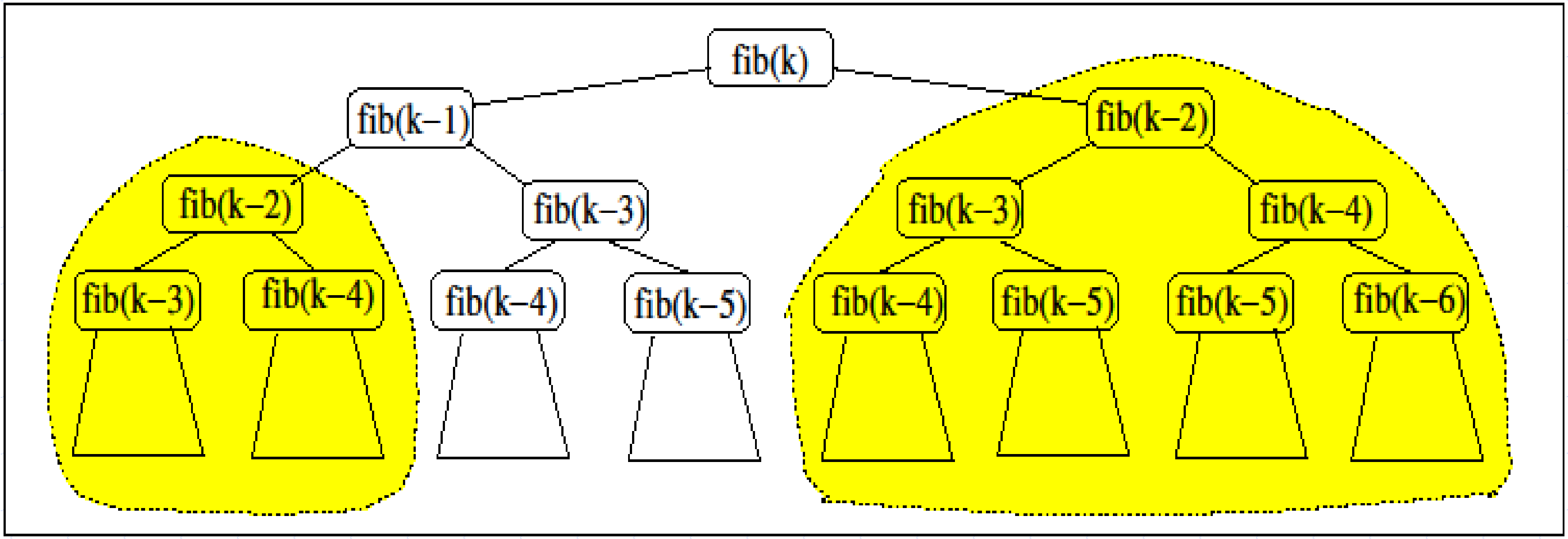


- Note that n_k at least doubles every other step.
  - $n_8 \geq 2 \cdot n_6 \geq 2 \cdot 2 \cdot n_4 \geq 2 \cdot 2 \cdot 2 \cdot n_2 \geq 2 \cdot 2 \cdot 2 \cdot 2 \cdot n_0 = 2^4 = 2^{8/2}$
- That is, $n_k > 2^{k/2}$, so exponential! E.g. $n_{40} > 2^{20} \approx 1M$

Inefficiency of this recursion

- The recursive method BinaryFib(*k*) is very inefficient: many repetitions of the same computation.

Recursion tree for BinaryFib(*k*): (BinaryFib(*k*) abbr. to fib(k))



A Better Fibonacci Algorithm

- Use linear recursion instead

Algorithm LinearFib(k):
  Input: An integer k ≥ 1
  Output: Pair of Fibonacci numbers (F_k , F_{k-1})

if k <= 1 **then**
  return (k, 0) F_1 F_0 (if k = 1)
else
  (i, j) = LinearFib(k – 1) F_{k-1} F_{k-2}
  return (i + j, i) (F_k , F_{k-1})

- LinearFib(k) makes k–1 recursive calls
  (BinaryFib(k) makes at least $2^{k/2}$ recursive calls)

Fibonacci: linear recursion

```
public static int[] linearFib(int n) {
    // method in class RecursiveFibonacci
    if (n <= 1) {
            return  ( new int[ ] { n, 0 } );
    }
    else {
            int[  ]  F = linearFib(n-1);
            return ( new int[ ] { F[0]+F[1], F[0] } );
    }
}
```

Fibonacci: iterative solution

```
public static int iterFib(int n) {        // method in class IterativeFibonacci
    if (n <= 1) return n;
    else {
        int previous = 0;                 // previous Fib. number (init. fib(0))
        int current = 1;                  // current Fib. number  (init. fib(1))
        int next;                         // next Fib. number
        for (int i = 2; i <= n; i++) {
            // current is fib(i-1); previous is fib(i-2)
            next = current + previous;    // next is fib(i)
            previous = current;
            current = next;               // current is fib(i); previous is fib(i-1)
        }
        return current;
    }
}
```

Fibonacci: comparing performance (1)

>java FibonacciTest 20

```
class FibonacciTest {
  public static void main(String args[]) {
    long startTime = System.currentTimeMillis();
    long f = IterativeFibonacci.iterFib( Integer.parseInt(args[0]) );
    long elapseTime = System.currentTimeMillis() - startTime;

    // print computed value and time
    // e.g. " fib(42) = 267914296 [computed iteratively in 0 ms] "
    System.out.println
        ( "fib(" + args[0] + ") = " + f
          + "[computed iteratively in " + elapseTime + " ms] " );

    ... // same for linearFib and binaryFib
}
```

Fibonacci: comparing performance (2)

```
> java  FibonacciTest  42
fib(42) = 267914296 [computed iteratively in 0 ms]
fib(42) = 267914296 [computed by linear recursion in 0 ms]
fib(42) = 267914296 [computed by binary recursion in 2785 ms]
>
```

Multiple Recursion (1)

- Motivating example:
  - summation puzzles – replace letters with digits (same letter – same digit) to make the equation true:
    - *pot* + *pan* = *bib*

      possible solution: $p = 4$, $o = 7$, $t = 3$, $a = 2$, $b = 8$, $i = 9$
      gives 473 + 425 = 898
    - *dog* + *cats* = *foes*
    - …
- Multiple recursion:
  - potentially many recursive calls, not just one or two

Multiple Recursion (2)

- Replace letters with unique digits to make equation true:
  pot + *pan* = *bib*
- Find a solution (*a,b,i,n,o,p,t*), for *pot* + *pan* = *bib* using unique digits in {0,1,2,3,4,5,6,7,8,9}
- Fix *a* = 0;
  Find a solution (*a*=0,*b,i,n,o,p,t*) for *pot + p0n = bib* using unique digits {1,2,3,4,5,6,7,8,9}
- Fix *a* = 1;
  Find a solution (*a*=1,*b,i,n,o,p,t*) for *pot + p1n = bib* using unique digits {0,2,3,4,5,6,7,8,9}

.....

Multiple recursion calls

- Fix *b* =1; find solution (a=0,*b*=1,*i,n,o,p,t*) using digits {2,3,4,5,6,7,8,9};
- Fix *b* =2; find solution (a=0,*b*=2,*i,n,o,p,t*) using digits {1,3,4,5,6,7,8,9}; .....

Algorithm for Multiple Recursion

and the puzzle, e.g. (a,b,i,n,o,p,t); pot+pan = bib

Algorithm PuzzleSolve(*k*,*S*,*U*):

Input: Sequence *S*: digits fixed for the initial letters;
Integer *k*: the number of remaining letters,
Sequence *U* (the digits to test for the remaining letters)

E.g. find solution (a=7,b=3,i,n,o,p,t): PuzzleSolve(5,(7,3),{0,1,2,4,5,6,8,9})

Compute: Generate and test *k*-length extensions of *S* using digits in *U* without repetitions, until a solution to the puzzle is found

if *k* = 0 **then**
  Test whether *S* is a sequence that solves the puzzle
  if *S* solves the puzzle **then** **print**("Solution found: ", *S*) **and terminate**
else
  for all *e* in *U* **do**
    Remove *e* from *U* {e is now used, so won't be available}
    Add *e* to the end of *S*
    PuzzleSolve(*k* - 1, *S*, *U*)
    Remove *e* from the end of *S*
    Add *e* back to *U* {e is again available}

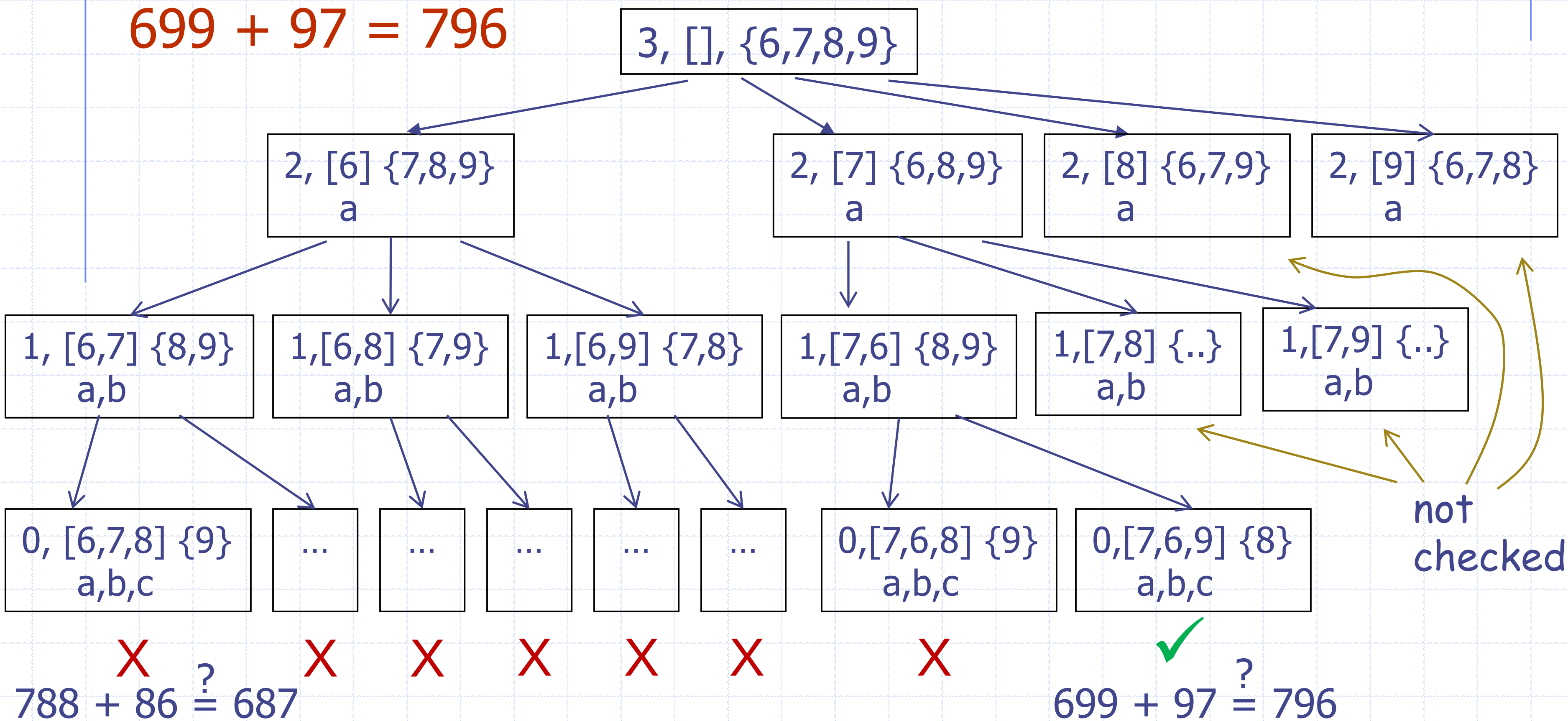
Slide by Matt Stallmann included with permission.

Example

bcc + ca = acb

699 + 97 = 796

a,b,c are from {6,7,8,9}



Implementation

```
public static boolean puzzleSolve( int k, ArrayList<Object> S,
                                   ArrayList<Object> U)  {
    if ( k == 0 ) {  return  checkSequence(S);  }
    else {
        for (int i = 0;  i < U.size();  i++)  {
            S.add( U.remove( i ) );
            if ( puzzleSolve( k – 1, S, U ) )  {
                // sequence S has been extended to a correct solution
                return true;
            }
            else    {  U.add( i, S.remove( S.size() – 1 ) );  }
        }
        // sequence S cannot be extended to a correct solution
        return false;
    } }
```

Exercises

To be done for the next tutorial

Exercise 1

- Draw the recursion trace of the call BinaryFib(6).
- Draw the recursion trace of the call LinearFib(6).

Show the values returned from each recursive call.

Exercise 2

- Implement the recursive and non-recursive (iterative) methods for reversing an array.

  That is, write Java methods which implement the algorithms:

  ReverseArray(*A, i, j*)

  IterativeReverseArray(*A, i, j*)

Exercise 3

- Consider the sequence of numbers P_0, P_1, P_2, P_3 ... defined recursively:

$$P_0 = P_1 = P_2 = 1,$$

$$P_n = (P_{n-3} * P_{n-1}) + 1, \text{ for } n \geq 3.$$

- Calculate P_6
- Write a straight recursive Java method

```
public static int recP(int n)
```

which computes the number P_n.

Exercise 3 (cont.)

- Consider the computation of recP(8), including all calls at all levels of recursion. How many calls recP(4) are there during this computation?

  Justify your answer by drawing the relevant part of the recursion tree.

- Write an iterative (non-recursive) Java method

```
public  static  int  iterP(int n)
```

  which computes the number P_n.

Exercise 4

- Consider the following Java method:

```java
public static int  tlum(int n, int m) {
    // assume both n and m are at least 1
    if ( m == 1 ) return n;
    else return (n + tlum(n, m-1));
}
```

- What are the values returned by the calls tlum(5,3) and tlum(6,15)?
- What does tlum(n, m) return?

Exercise 5

- Show the outcome of the following calls to method drawTicks (read the relevants slides, not presented):

drawTicks(0), drawTicks(1), drawTicks(2), drawTicks(3), drawTicks(4)