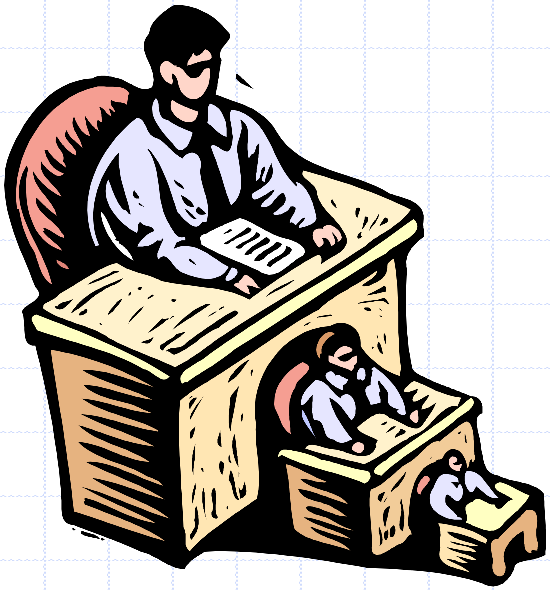
Presentation for use with the textbook Data Structures and Algorithms in Java, 6th edition, by M. T. Goodrich, R. Tamassia, and M. H. Goldwasser, Wiley, 2014

Recursion



Analysis of Algorithms - Review

- **Running time** – focus on worst case scenario
- **Experimental studies**:
  - Write the algorithm and **measure running times by varying input size**, plot the results
  - **Disadvantages**:
    - algorithm implementation is required
    - the same hardware and software environment should be used

Analysis of Algorithms - Review

- **Theoretical Analysis**
  - **Running time** as function of input size, n
  - Evaluate the performance **independent** of hardware/software environment
  - **Random Access Machine Model** (RAM)
    - CPU, numbered memory cells whose access takes unit time
    - Primitive operations take a **constant amount of time** in the RAM model
  - **Determine** the maximum **number of primitive operations f(n)** executed by an algorithm, **as a function of the input size**

Analysis of Algorithms - Review

- Examples of **primitive operations**:
  - Evaluating an expression
  - Assigning a value to a variable
  - Indexing into an array
  - Calling a method
  - Returning from a method
- Evaluate f(n) in terms of **Big-Oh Notation**:

$f(n)$ is $O(g(n))$ if there are positive constants c and n_0 such that

$$f(n) \leq cg(n) \text{ for } n \geq n_0$$

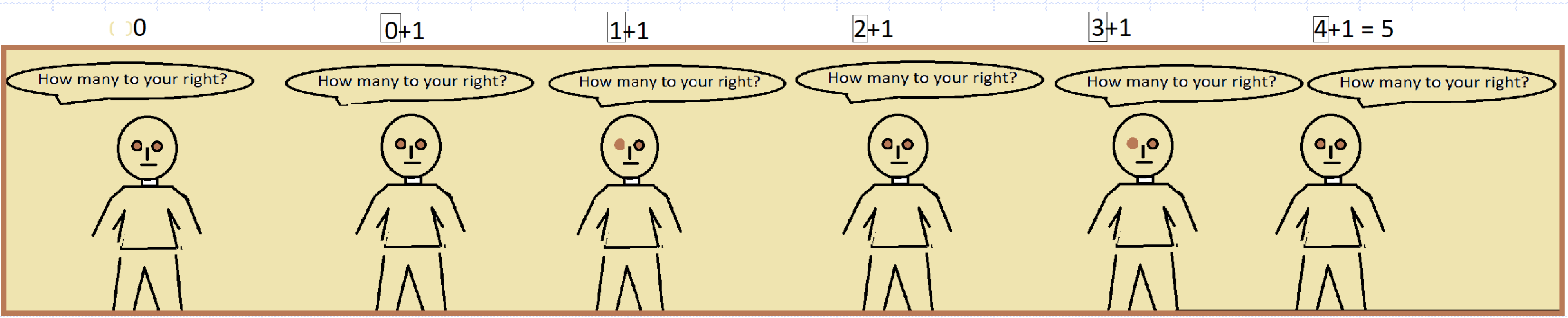
- **Asymptotic analysis**:
  - find the worst-case number of primitive operations executed as a function of the input size f(n)
  - express this function with big-Oh notation

Objectives

- Define recursion pattern
- Illustrate recursion using factorial function, ruler drawing, binary search, and file systems
- Analyze recursive algorithms

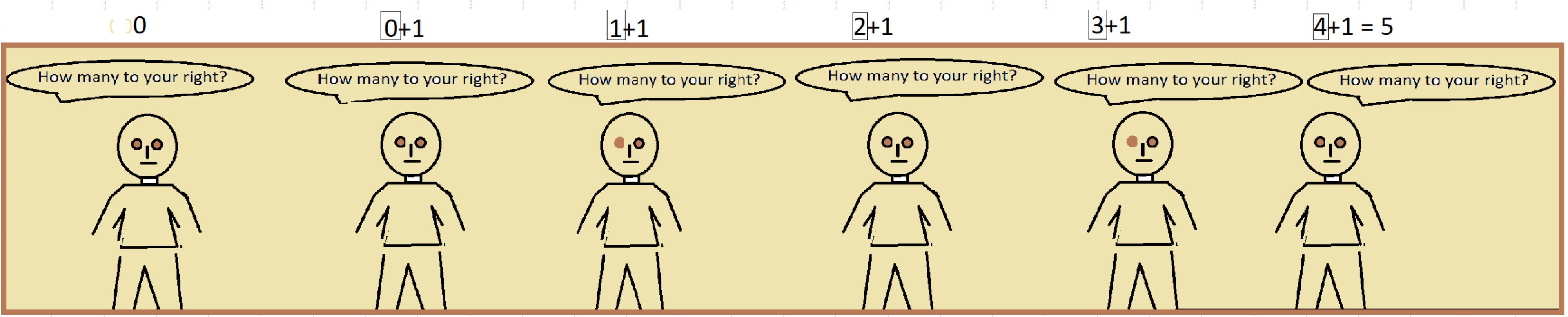
Exercise

- In the picture below:
  - The rightmost person wants to know how many people are to the right of his/her position.
  - How can he/she solve this problem? (*recursively*)



Recursive algorithm

- Recursion is all about **breaking a big problem into smaller occurrences** of that same problem.
  - Each person can **solve a small part of the problem** by asking the person to the right:
  - If there is someone to the right, ask him/her **how many people are to his/her right**.
    - When he/she respond with a value **N**, then answer **N + 1**.
    - If there is nobody to the right, the answer should be **0**.



Recursion Pattern

- Allows to solve large problems by solving a smaller occurrence of the same problem.
- A recursive method must contain:
  1. *One or more stopping conditions*: under certain conditions, it would stop the method from calling itself again
     - This is known as **base case**
  2. *One or more recursive calls*: this is when **a method calls itself**
     - These are known as **recursive cases**
  - The recursive cases must eventually **lead to a base case**.

The Recursion Pattern

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

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

- Recursive definition:

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

- As a Java method:

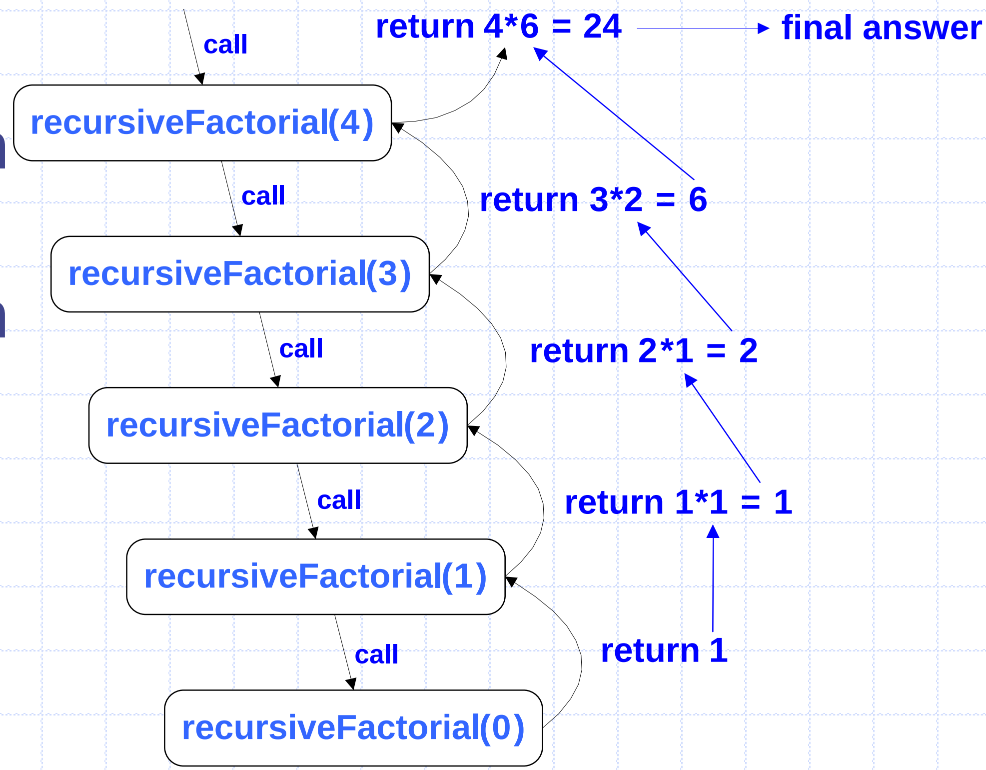
```
public static int factorial(int n) throws IllegalArgumentException {
  if (n < 0)
    throw new IllegalArgumentException( );    // argument must be nonnegative
  else if (n == 0)
    return 1;                                 // base case
  else
    return n * factorial(n−1);                // recursive case
}
```

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 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**
- Example



Recursion Steps

```java
public static void main(String[] args) {

      int n=5;
      System.out.println("factorial("+n+") = " + factorial(n));

  }
```

```java
/** Computes the factorial of the given (nonnegative) integer) */

  public static int factorial(int n) throws IllegalArgumentException {
    if (n < 0)
      throw new IllegalArgumentException();     // argument must be nonnegative
    else if (n == 0)
      return 1;                                 // base case
    else
      return n * factorial(n-1);                // recursive case
  }
```

returns 5*24

```java
/** Computes the factorial of the given (nonnegative) integer) */

  public static int factorial(int n) throws IllegalArgumentException {
    if (n < 0)
      throw new IllegalArgumentException();     // argument must be nonnegative
    else if (n == 0)
      return 1;                                 // base case
    else
      return n * factorial(n-1);                // recursive case
  }
```

returns 4*6

```java
/** Computes the factorial of the given (nonnegative) integer) */
  public static int factorial(int n) throws IllegalArgumentException {
    if (n < 0)
      throw new IllegalArgumentException();     // argument must be nonnegative
    else if (n == 0)
      return 1;                                 // base case
    else
      return n * factorial(n-1);                // recursive case
```

returns 3*2

Recursion Steps

```
}
/** Computes the factorial of the given (nonnegative) integer */

public static int factorial(int n) throws IllegalArgumentException {
  if (n < 0)
    throw new IllegalArgumentException();     // argument must be nonnegative
  else if (n == 0)
    return 1;                                  // base case
  else
    return n * factorial(n-1);                 // recursive case
}
```

returns 2*1

```
/** Computes the factorial of the given (nonnegative) integer */

public static int factorial(int n) throws IllegalArgumentException {
  if (n < 0)
    throw new IllegalArgumentException();     // argument must be nonnegative
  else if (n == 0)
    return 1;                                  // base case
  else
    return n * factorial(n-1);
                                             // recursive case
}
```

returns 1*1

```
/** Computes the factorial of the given (nonnegative) integer */

public static int factorial(int n) throws IllegalArgumentException {
  if (n < 0)
    throw new IllegalArgumentException();     // argument must be nonnegative
  else if (n == 0)
    return 1;                                  // base case
  else
    return n * factorial(n-1);
                                             // recursive case
}
```

returns 1

Verifying The Recursion

```
// precondition: n >= 0
if (n == 0)        //base case
return (1)
else
//recursive case: the parameter is n and the recursive call passes
// the argument n – 1
return (n * factorial (n – 1))
```

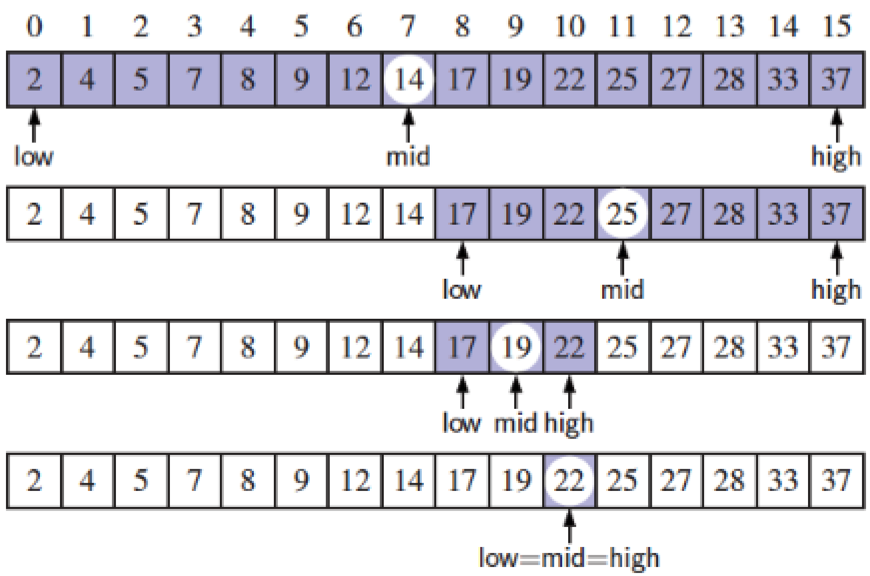
Binary Search

Search for an integer target = 22 in an **ordered list,**

```
/**
 * Returns true if the target value is found in the indicated portion of the data array.
 * This search only considers the array portion from data[low] to data[high] inclusive.
 */
public static boolean binarySearch(int[ ] data, int target, int low, int high) {
  if (low > high)
    return false;                                        // interval empty; no match
  else {
    int mid = (low + high) / 2;
    if (target == data[mid])
      return true;                                       // found a match
    else if (target < data[mid])
      return binarySearch(data, target, low, mid − 1);   // recur left of the middle
    else
      return binarySearch(data, target, mid + 1, high);  // recur right of the middle
  }
}
```

Visualizing Binary Search

- We consider three cases:
  - If the target equals data[mid], then we have found the target.
  - If target < data[mid], then we recur on the first half of the sequence.
  - If target > data[mid], then we recur on the second half of the sequence.



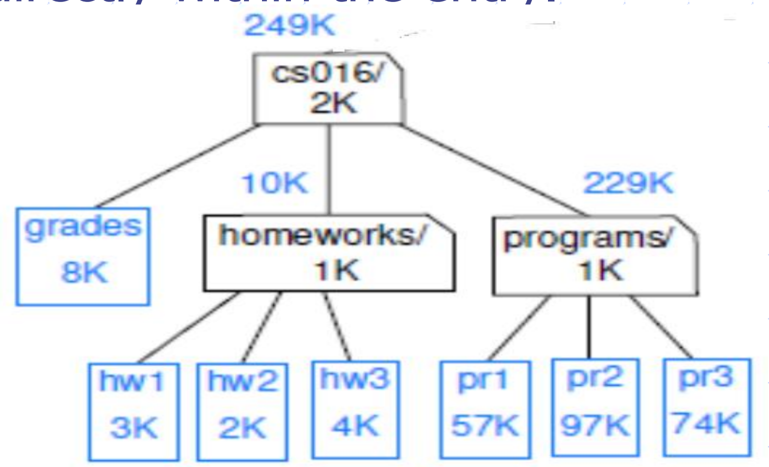
Analyzing Binary Search

- Runs in **O(log n)** time.
  - In the worst case scenario, low = 0, high = n-1
  - **At each step, divide the search region by 2**
  - Let **k** be the number of steps or levels:
  - mid = (low+high)/2 = $(n-1)/2^1$
  - mid = $(n-1)/2^2$
  - mid = $(n-1)/2^3$
  - ……
  - mid = $(n-1)/2^k \geq \mathbf{1} \rightarrow \mathbf{(n-1)} \geq 2^k \rightarrow \mathbf{n} \geq 2^k \rightarrow$ $\mathbf{log(n)} \geq \mathbf{k\ log(2)} \rightarrow \mathbf{log(n)} \geq \mathbf{k}$

- Hence, there can be **at most log n levels**

File Systems

- The operating system allows **directories** to be nested arbitrarily deeply
- The cumulative disk space for an entry can be computed with a simple recursive algorithm.
  - It is equal to the immediate disk space used by the entry plus the sum of the cumulative disk space usage of any entries that are stored directly within the entry.



Pseudocode for calculating disk usage of a file system

Algorithm DiskUsage(*path*):

Input: A string designating a path to a file-system entry

Output: The cumulative disk space used by that entry and any nested entries

total = size(*path*) {immediate disk space used by the entry}

if *path* represents a directory **then**

for each *child* entry stored within directory *path* **do**

total = *total* + DiskUsage(*child*) {recursive call}

return *total*

A recursive method for calculating disk usage of a file system

```
/**
 * Calculates the total disk usage (in bytes) of the portion of the file system rooted
 * at the given path, while printing a summary akin to the standard 'du' Unix tool.
 */
public static long diskUsage(File root) {
  long total = root.length();                      // start with direct disk usage
  if (root.isDirectory()) {                         // and if this is a directory,
    for (String childname : root.list()) {          // then for each child
      File child = new File(root, childname);      // compose full path to child
      total += diskUsage(child);                    // add child's usage to total
    }
  }
  System.out.println(total + "\t" + root);          // descriptive output
  return total;                                     // return the grand total
}
```

Code Fragment 5.5: A recursive method for reporting disk usage of a file system.

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.

Recursive Problems - Recap

- Factorial and Disk Usage problems lend themselves naturally into recursive definitions:
  - factorial **function definition is recursive**.
  - In file systems the **data structure is recursive**, folders contain other folders, and then there are files.
- Each **recursive step** reduces the problem into a smaller instance and then the **base case** lies at the bottom, guaranteeing the convergence.
- Recursion leads to more readable algorithms.
- Let's illustrate these with more examples.

Example of Linear Recursion

Algorithm linearSum(A, n):
Input:
  Array, A, of integers
  Integer n such that
    $0 \le n \le |A|$
Output:
    Sum of the first **n** integers in A

```
if n = 0 then
  return 0
else
  return
linearSum(A, n - 1) + A[n - 1]
```

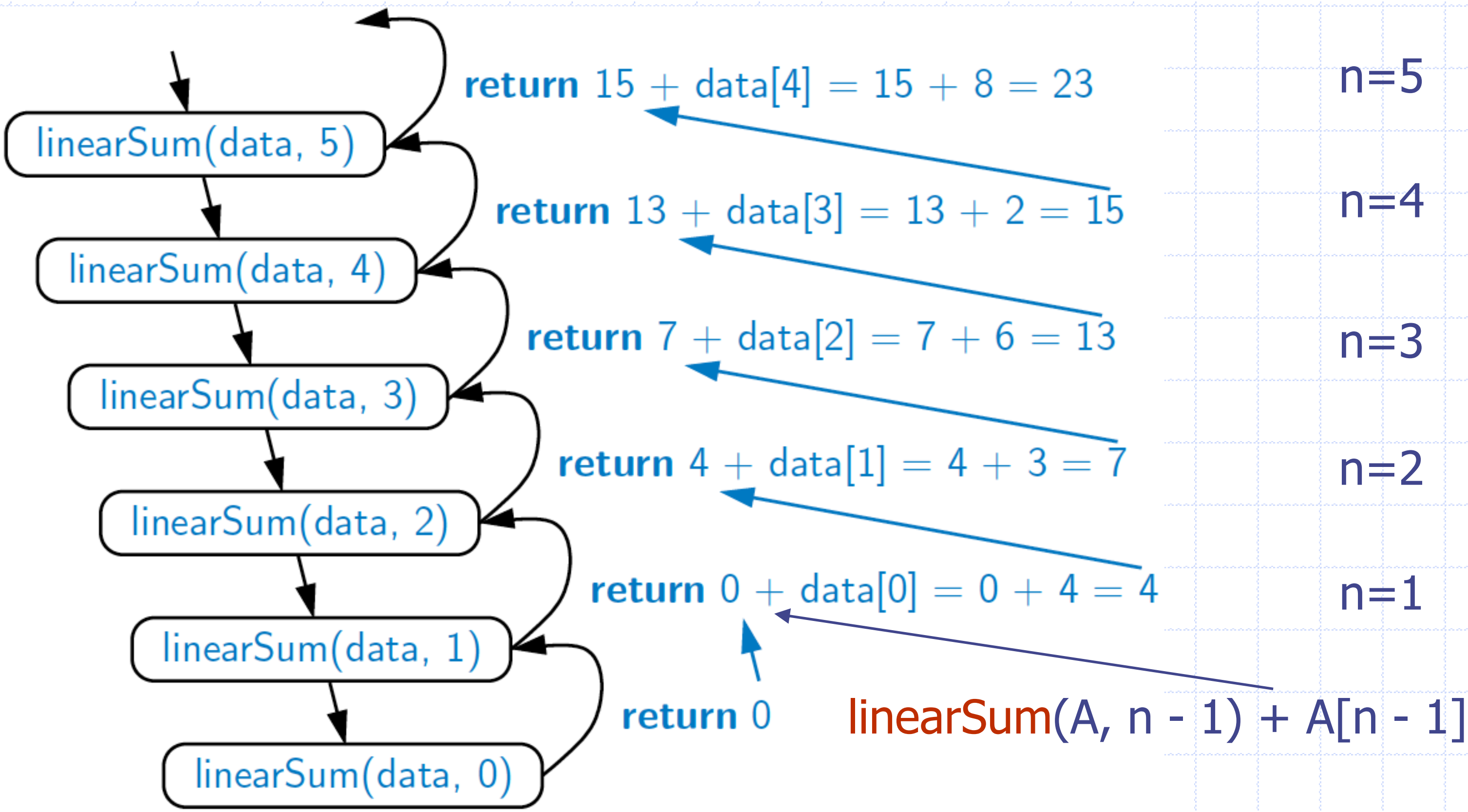
Example: data = [4, 3, 6, 2, 8], n=5

The **sum can be expressed recursively** as the sum of n-1 elements plus the last one.

$$\text{data} = [\overbrace{\underbrace{4, 3, 6, 2,}_{n-1}\ 8}^{n}]$$

Example of Linear Recursion

- Recursion trace of linearSum(data, 5) called on array data = [4, 3, 6, 2, 8]



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 index j

if $i < j$ then

    Swap A[i] and A[j]

    reverseArray(A, i + 1, j – 1)

return

data = [4, 3, 2, 6, 8]

Defining Arguments for Recursion

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

```
/** Reverses the contents of subarray data[low] through data[high] inclusive. */
public static void reverseArray(int[ ] data, int low, int high) {
  if (low < high) {                          // if at least two elements in subarray
    int temp = data[low];                    // swap data[low] and data[high]
    data[low] = data[high];
    data[high] = temp;
    reverseArray(data, low + 1, high − 1);   // recur on the rest
  }
}
```

Computing Powers

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

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

- For example: $p(x,3) = x^3 = x \cdot x^2$
- This leads to a power function that runs in O(n) time (for we make **n** recursive calls)
- We can do better than this, however

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,

$$2^4 = 2^{(4/2)2} = (2^{4/2})^2 = (2^2)^2 = 4^2 = 16$$
$$2^5 = 2^{1+(4/2)2} = 2(2^{4/2})^2 = 2(2^2)^2 = 2(4^2) = 32$$
$$2^6 = 2^{(6/2)2} = (2^{6/2})^2 = (2^3)^2 = 8^2 = 64$$
$$2^7 = 2^{1+(6/2)2} = 2(2^{6/2})^2 = 2(2^3)^2 = 2(8^2) = 128$$

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  = Power(x, (n - 1)/ 2)
    return x · y ·y
else
    y = Power(x, n/ 2)
    return y · 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.

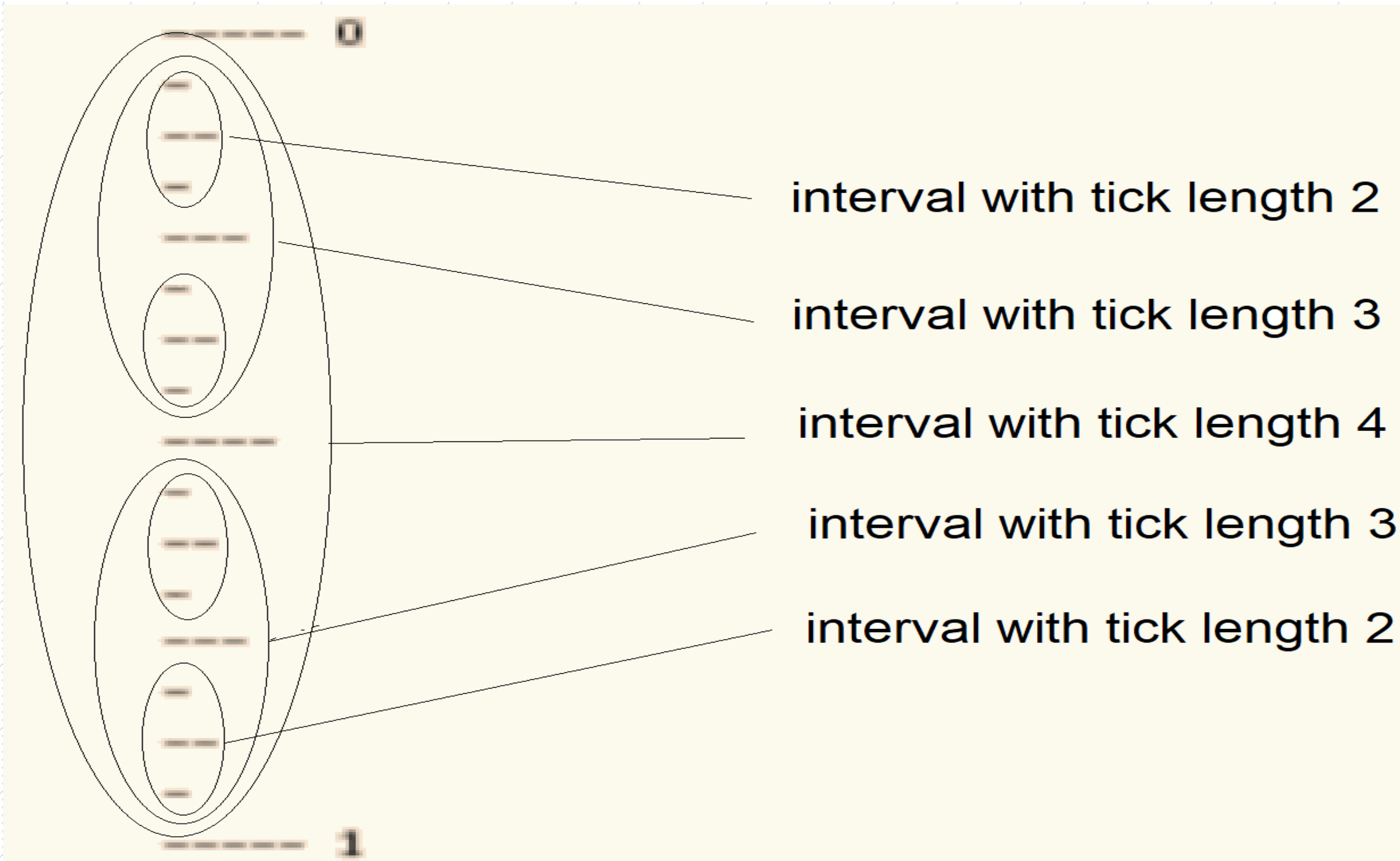
Example: English Ruler

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

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

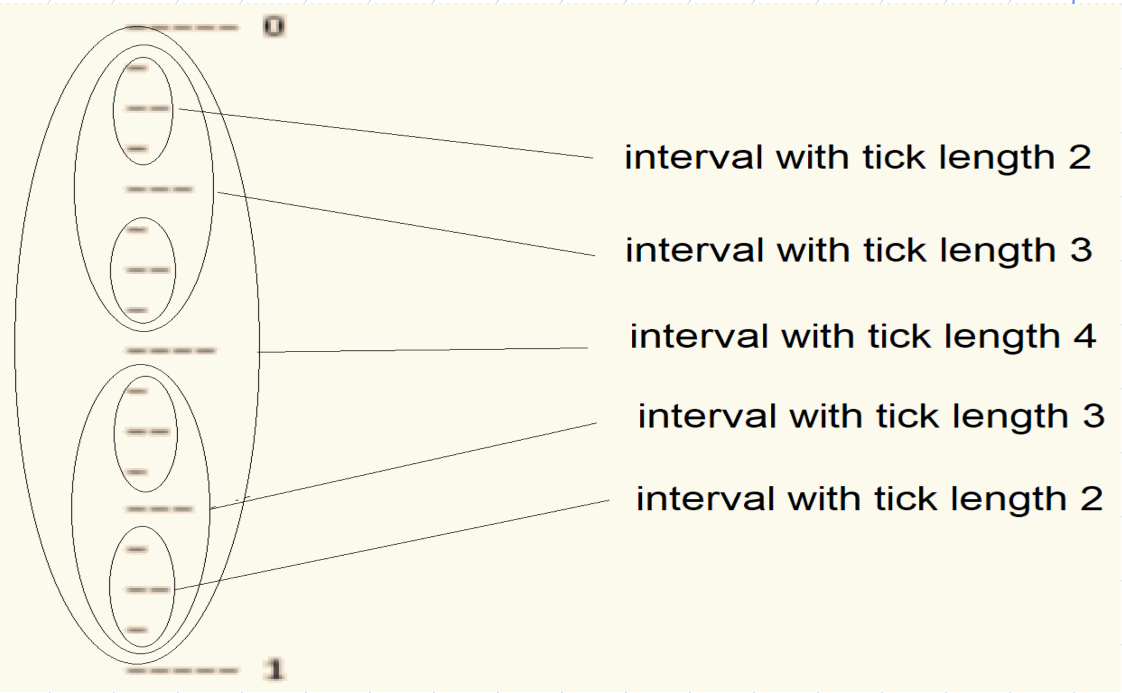
Recursive Decomposition

- Define the intervals of different central tick length:



Recursive Definition

- An interval with a central tick length $L \geq 1$ consists of:
  - An interval with a central tick length $L-1$
  - A single tick of length L
  - An interval with a central tick length $L-1$
- Base case is L=0



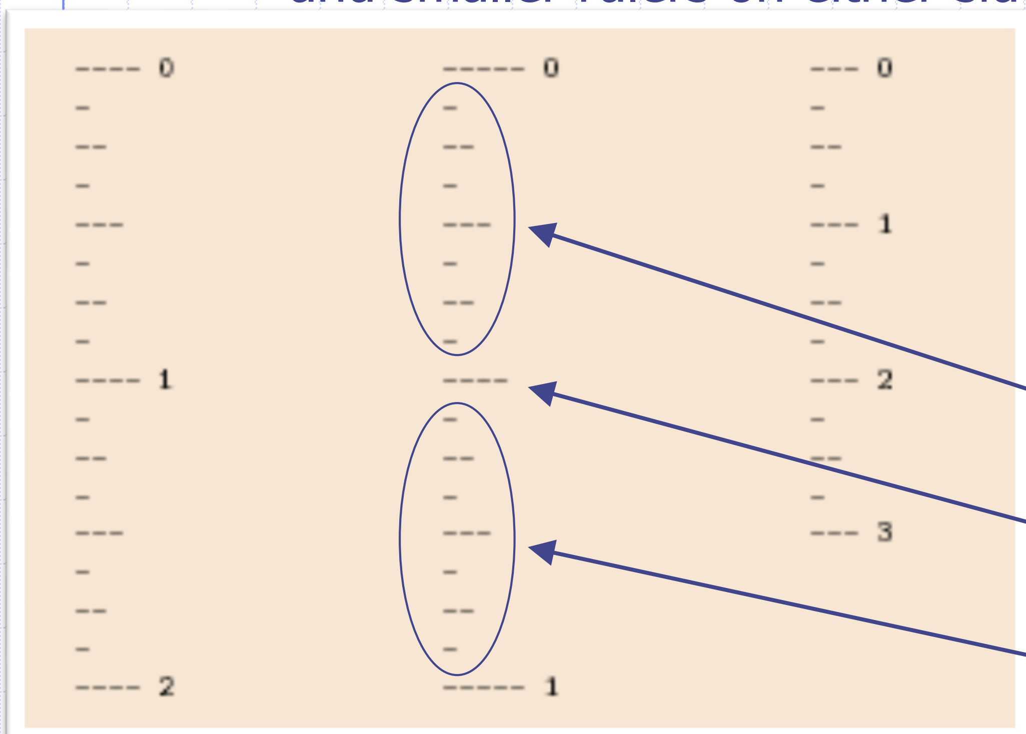
Slide by Matt Stallmann included with permission.

Using Recursion

drawInterval(length)

Input: length of a 'tick'

Output: ruler with tick of the given length in the middle and smaller rulers on either side



drawInterval(length)

if(length > 0) then

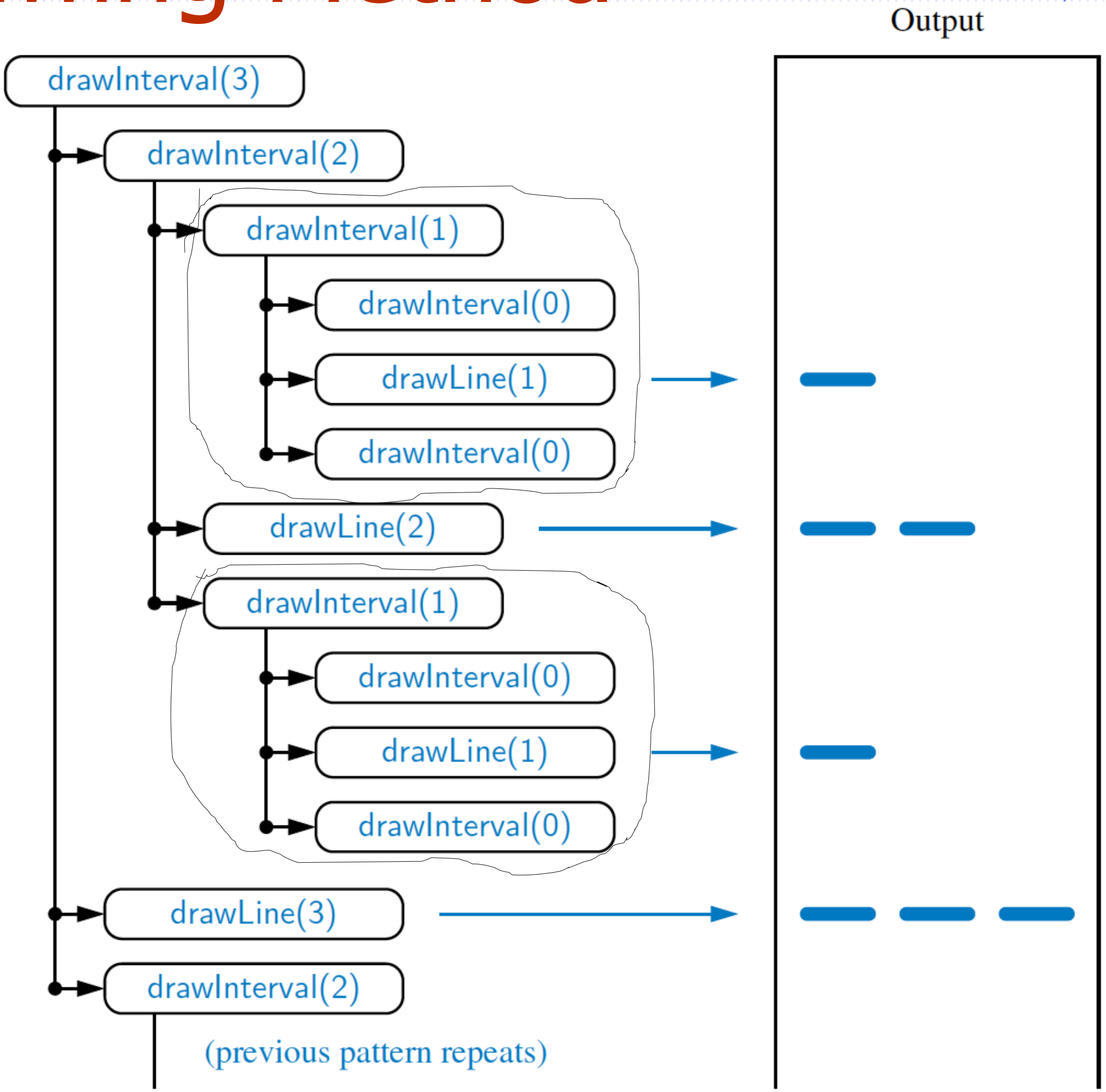
drawInterval (length – 1)

draw line of the given length

drawInterval (length – 1)

Recursive Drawing Method

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



A Recursive Method for Drawing Ticks on an English Ruler

```java
/** Draws an English ruler for the given number of inches and major tick length. */
public static void drawRuler(int nInches, int majorLength) {
  drawLine(majorLength, 0);                      // draw inch 0 line and label
  for (int j = 1; j <= nInches; j++) {
    drawInterval(majorLength - 1);               // draw interior ticks for inch
    drawLine(majorLength, j);                    // draw inch j line and label
  }
}
private static void drawInterval(int centralLength) {
  if (centralLength >= 1) {                      // otherwise, do nothing
    drawInterval(centralLength - 1);             // recursively draw top interval
    drawLine(centralLength);                     // draw center tick line (without label)
    drawInterval(centralLength - 1);             // recursively draw bottom interval
  }
}
private static void drawLine(int tickLength, int tickLabel) {
  for (int j = 0; j < tickLength; j++)
    System.out.print("-");
  if (tickLabel >= 0)
    System.out.print(" " + tickLabel);
  System.out.print("\n");
}
/** Draws a line with the given tick length (but no label). */
private static void drawLine(int tickLength) {
  drawLine(tickLength, -1);
}
```

Note the two recursive calls

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 (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

Binary Recursion

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

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

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

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

```
private static void drawInterval(int centralLength) {
  if (centralLength >= 1) {                 // otherwise, do nothing
    drawInterval(centralLength − 1);        // recursively draw top interval
    drawLine(centralLength);                // draw center tick line (without label)
    drawInterval(centralLength − 1);        // recursively draw bottom interval
  }
}
```

Another Binary Recursive Method

- Problem: add all the numbers in an integer array A:
  - we can recursively compute the sum of the first half, and the sum of the second half, and add those sums together

```
/** Returns the sum of subarray data[low] through data[high] inclusive. */
public static int binarySum(int[ ] data, int low, int high) {
  if (low > high)                                    // zero elements in subarray
    return 0;
  else if (low == high)                              // one element in subarray
    return data[low];
  else {
    int mid = (low + high) / 2;
    return binarySum(data, low, mid) + binarySum(data, mid+1, high);
  }
}
```

Code Fragment 5.10: Summing the elements of a sequence using binary recursion.

Binary Recursive Method

- **Example trace of** binarySum(data, 0, 7)**:**

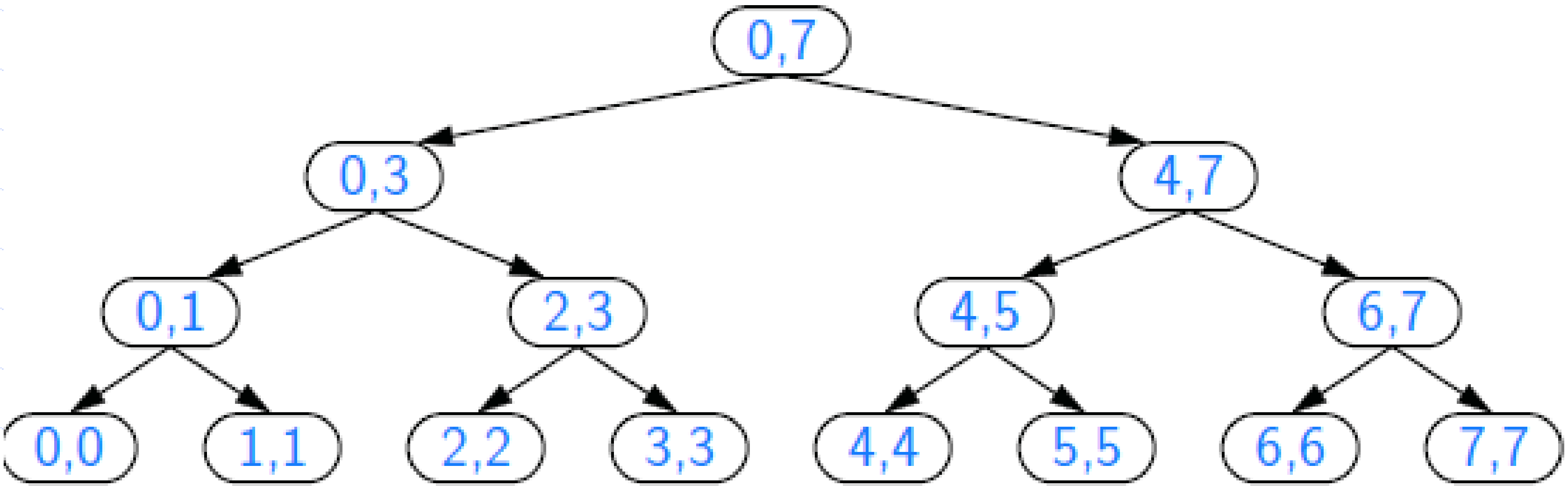


Figure 5.13: Recursion trace for the execution of binarySum(data, 0, 7).

- The running time of binarySum is $O(n)$, however binarySum uses $O(\log n)$ amount of additional space, whereas linearSum uses $O(n)$

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 kth Fibonacci number F_k
  if $k \leq 1$ **then**
    return k
  else
    return BinaryFib($k - 1$) + BinaryFib($k - 2$)

Analysis

- Let n_k be the number of calls performed in the execution of 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
- That is, $n_k > 2^{k/2}$. It is exponential!

A Better Fibonacci Algorithm

- Use linear recursion instead by defining a recursive method that returns an array with two consecutive Fibonacci numbers (F_k , F_{k-1}):

  Algorithm LinearFibonacci(k):

  Input: A nonnegative integer k

  Output: Pair of Fibonacci numbers (F_k , F_{k-1})

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

  return (k, 0)

  else

  (i, j) = LinearFibonacci(k – 1) //returns $\{F_{k-1}, F_{k-2}\}$

  return (i +j, i) // we want $\{F_k, F_{k-1}\}$

LinearFibonacci makes k–1 recursive calls – no need to recompute the second value already known.

Multiple Recursion

- Motivating example:
  - summation puzzles
    - *pot* + *pan* = *bib*
    - *dog* + *cat* = *pig*
    - *boy* + *girl* = *baby*
- Multiple recursion:
  - makes potentially many recursive calls
  - not just one or two

Algorithm for 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
```

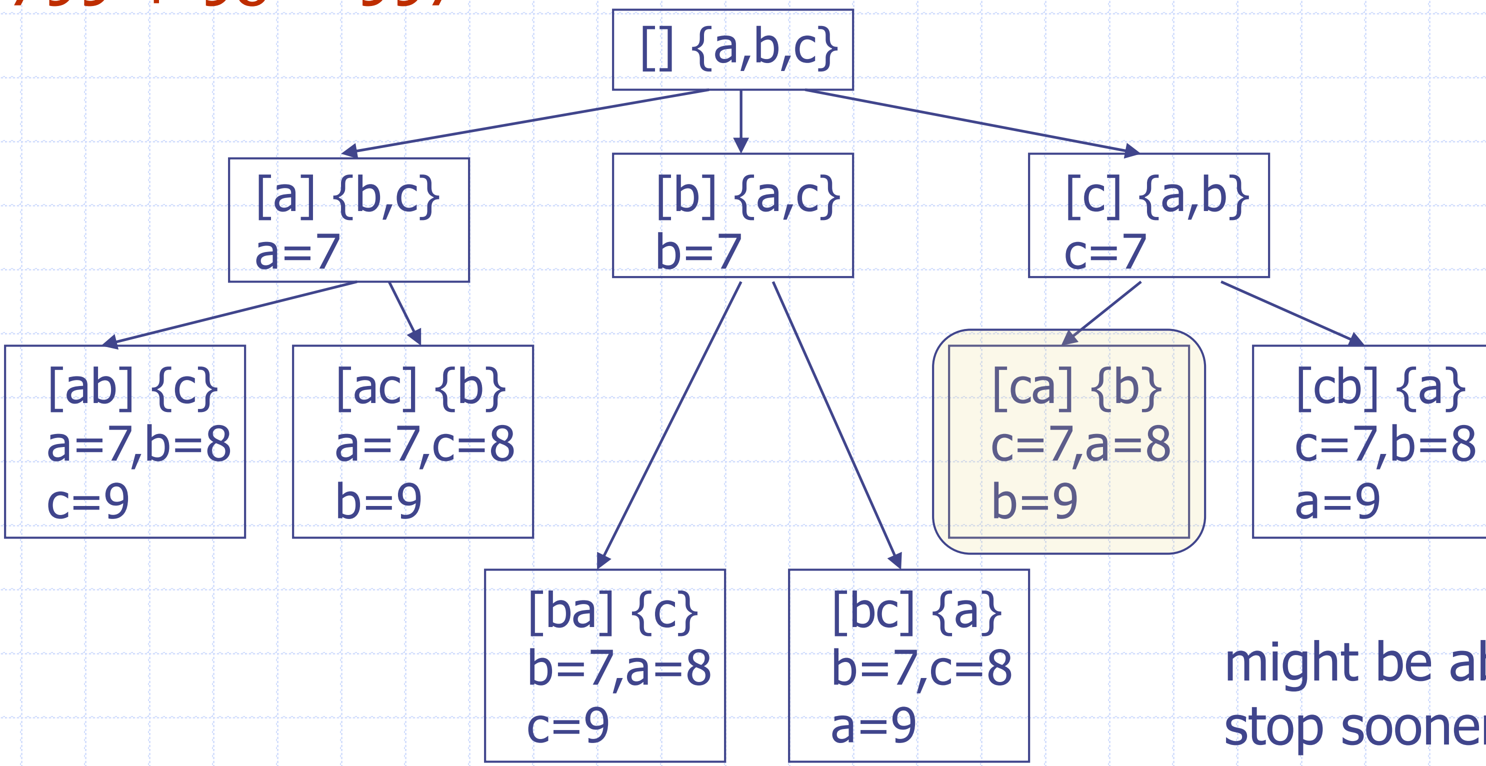
Slide by Matt Stallmann included with permission.

Example

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