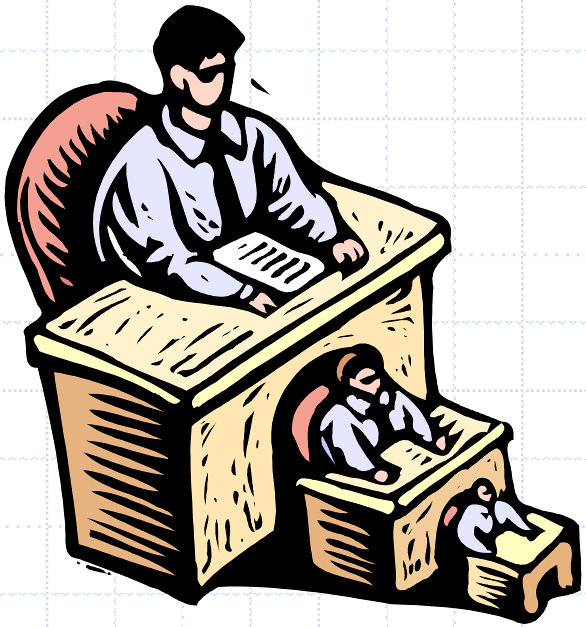
Presentation for use with the textbook Data Structures and Algorithms in Java, 6th edition, by M. T. Goodrich, R. Tamassia, and M. H. Goldwasser, Wiley, 2014

Recursion



The Recursion Pattern

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

  n! = 1· 2· 3· ··· · (n-1)· n
- Recursive definition:

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

$$\begin{cases} \text{se } n \le 1 \Rightarrow n! = 1 \\ \text{se } n > 1 \Rightarrow n! = n * (n-1)! \end{cases}$$

```
int fatt(int n)
{
  if (n<=1)
    return 1;  → Caso di base
  else
    return n * fatt(n-1); → Caso ricorsivo
}
```

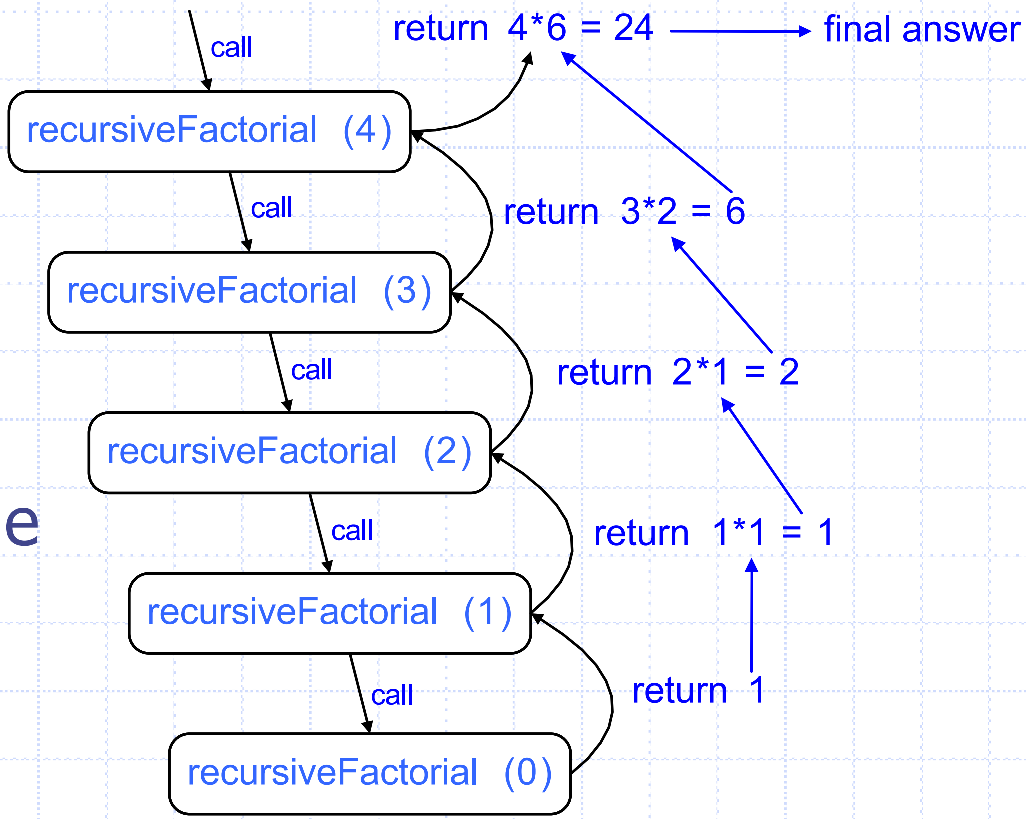
Semplificazione dei dati del problema

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



Example: English Ruler

- Print the ticks and numbers like an English ruler:

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

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

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

drawInterval(length)

if(length > 0) then

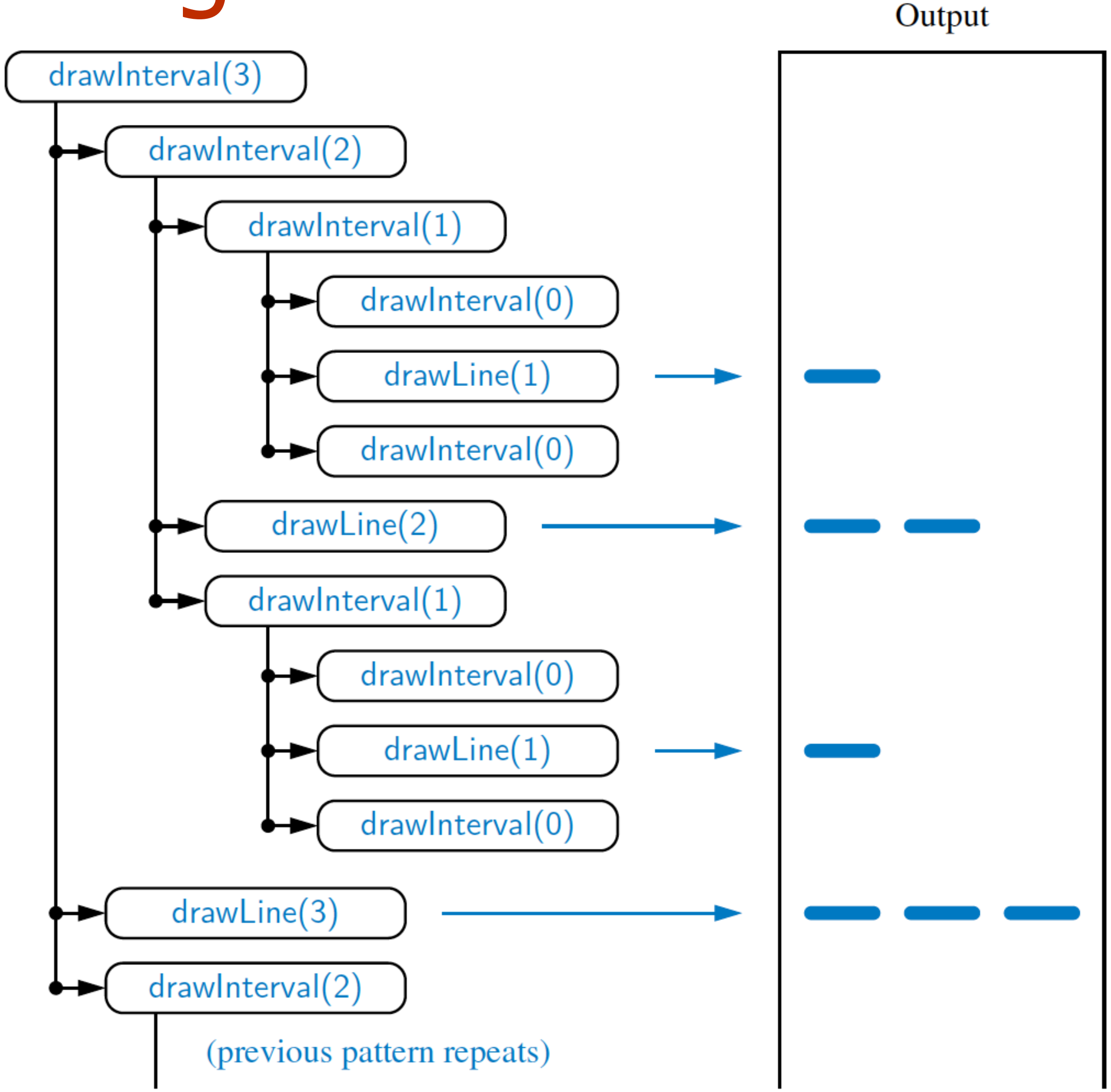
drawInterval (length – 1)

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 $\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



A Recursive Method for Drawing Ticks on an English Ruler

```
/** 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

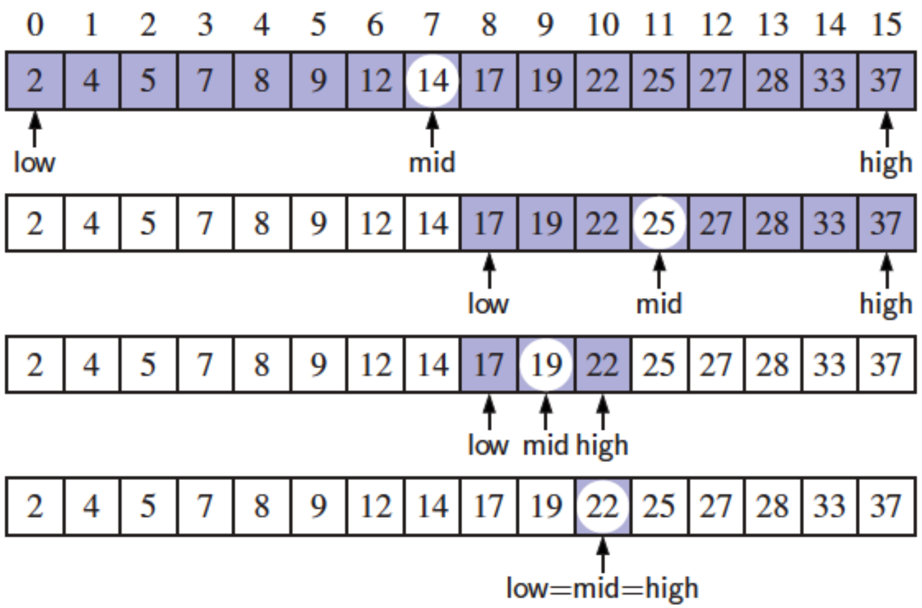
Binary Search

Search for an integer 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.
  - The remaining portion of the list is of size high – low + 1
  - After one comparison, this becomes one of the following:

$$(\text{mid}-1)-\text{low}+1=\left\lfloor\frac{\text{low}+\text{high}}{2}\right\rfloor-\text{low}\leq\frac{\text{high}-\text{low}+1}{2}$$

$$\text{high}-(\text{mid}+1)+1=\text{high}-\left\lfloor\frac{\text{low}+\text{high}}{2}\right\rfloor\leq\frac{\text{high}-\text{low}+1}{2}.$$

  - Thus, each recursive call divides the search region in half; hence, there can be at most log n levels

Linear Recursion

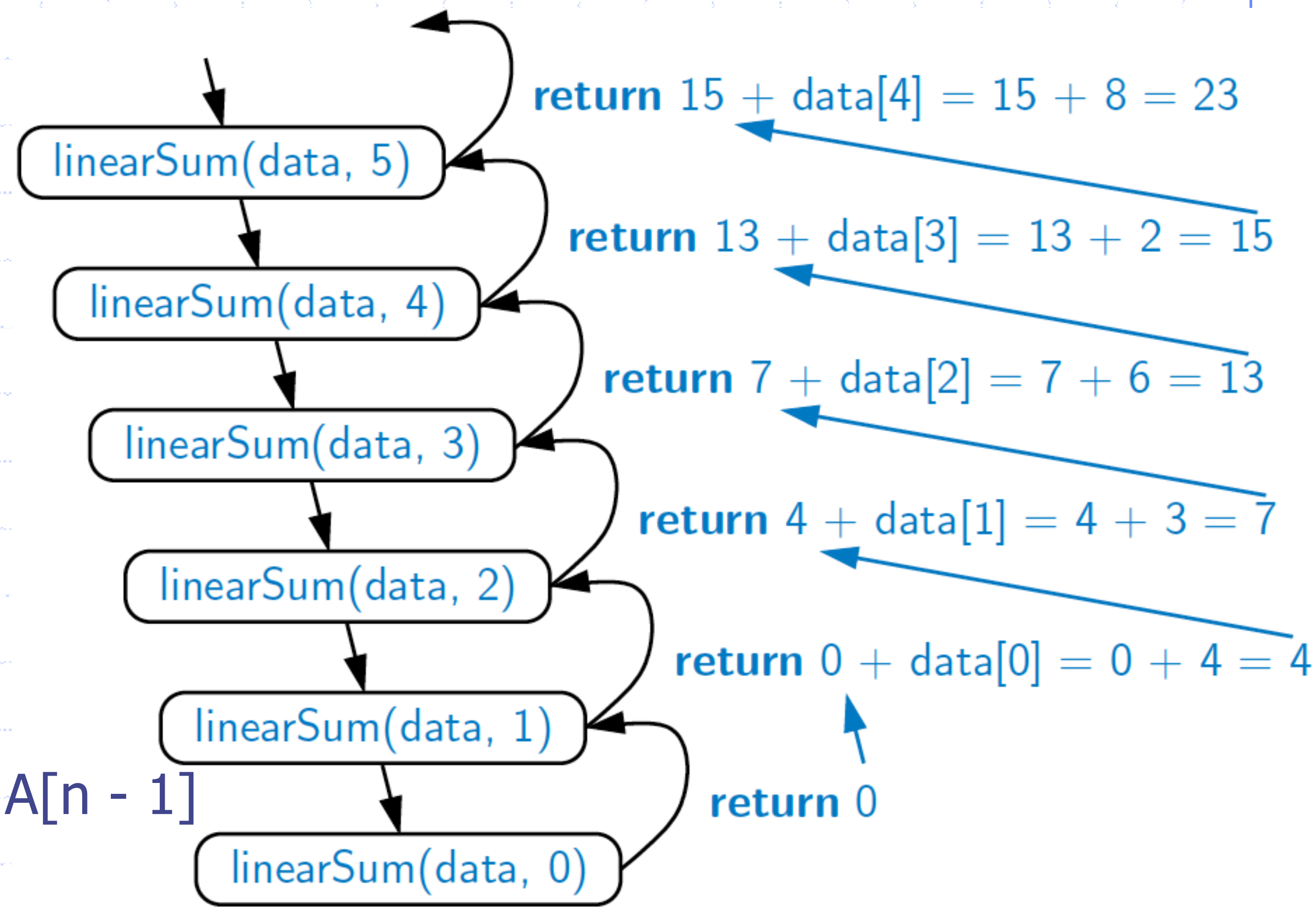
- Test for base cases
  - Begin by testing for a set of base cases (there should be at least one).
  - Every possible chain of recursive calls must eventually reach a base case, and the handling of each base case should not use recursion.
- Recur once
  - Perform a single recursive call
  - This step 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:
  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]

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

```
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 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:

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

- This leads to an 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.

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.

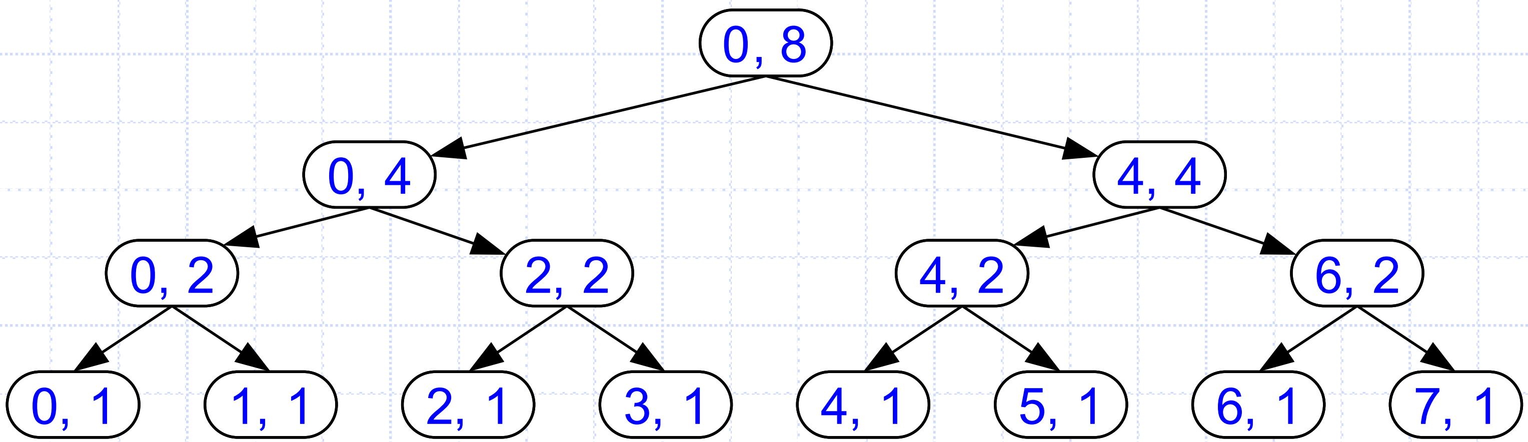
```
---- 0        ----- 0       --- 0
-             -             -
--            --            --
-             -             -
---           ---           --- 1
-             -             -
--            --            --
-             -             -
---- 1        ----          --- 2
-           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

Another Binary Recusive Method

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

  Algorithm BinarySum(A, i, n):
  Input: An array A and integers i and n
  Output: The sum of the n integers in A starting at index i
  if n = 1 **then**
  return A[i]
  return BinarySum(A, i, n/ 2) + BinarySum(A, i + n/ 2, n/ 2)

- **Example trace:**



Computing Fibonacci Numbers

- Fibonacci numbers are defined recursively:

  $F_0 = 0$

  $F_1 = 1$

  $F_i = F_{i-1} + F_{i-2}$ for $i > 1$.

- Recursive algorithm (first attempt):

Algorithm BinaryFib(k):

Input: Nonnegative integer k

Output: The kth Fibonacci number F_k

if $k <= 1$ **then**

return k

else

return BinaryFib($k - 1$) + BinaryFib($k - 2$)

Analysis

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

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)

- LinearFibonacci makes k–1 recursive calls

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
```

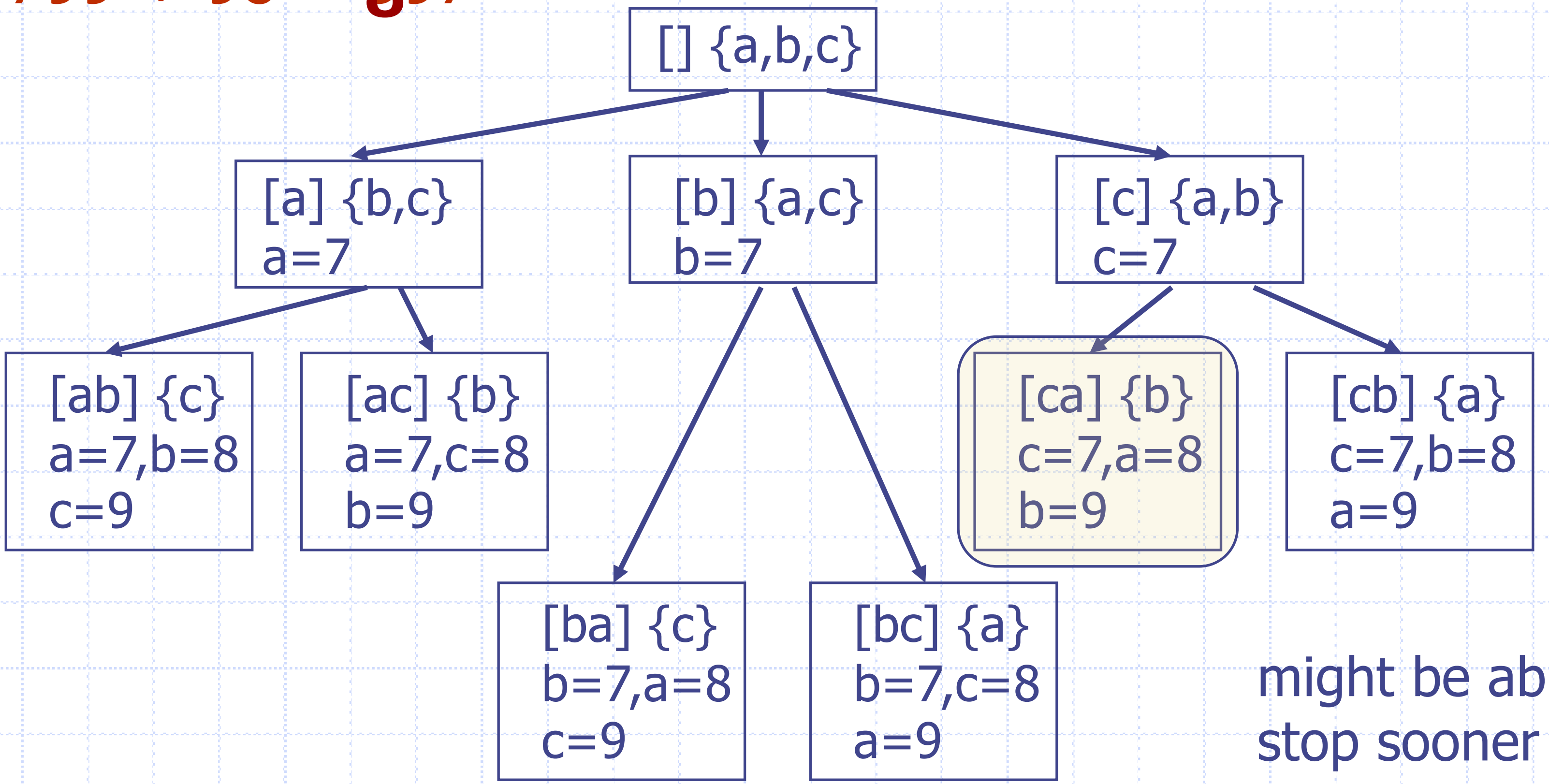
Slide by Matt Stallmann included with permission.

Example

cbb + ba = abc

799 + 98 = 897

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



might be able to stop sooner

Visualizing PuzzleSolve