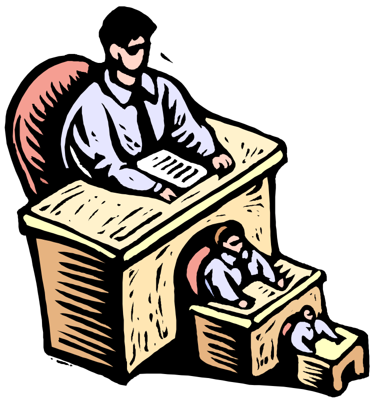
Presentation for use with the textbook Data Structures and Algorithms in Java, 6th edition, by M. T. Goodrich, R. Tamassia, and M. H. Goldwasser, Wiley, 2014

Recursion



Factorial

$$5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 = 5! = f(n)$$

$$f(n) = n! = n \cdot (n-1)! = n \cdot f(n-1)$$

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

$$f(n) = n! = n \cdot (n-1)! = n \cdot f(n-1)$$

A recursive function (or method) is a function that calls itself!

```
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
}
```


$$f(n) = n! = n \cdot (n-1)! = n \cdot f(n-1)$$

Recursion solves a problem with the solution of the smaller identical problem(s).

```
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
}
```

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

Base Case + Recursive Case

```
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
}
```


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

Each recursive call should be defined so that it makes *progress towards a base case*.

```
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*.

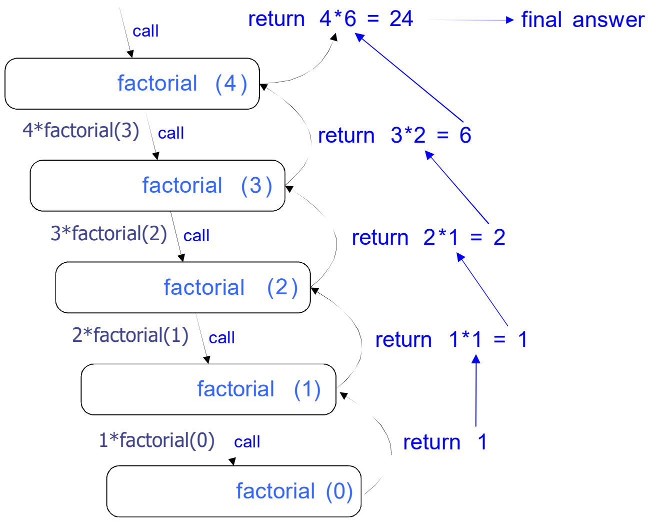
Recursion Trace

$$
\begin{aligned}
factorial(5) &= 5 \cdot factorial(4) \\
&= 5 \cdot (4 \cdot factorial(3)) \\
&= 5 \cdot (4 \cdot (3 \cdot factorial(2))) \\
&= 5 \cdot (4 \cdot (3 \cdot (2 \cdot factorial(1)))) \\
&= 5 \cdot (4 \cdot (3 \cdot (2 \cdot (1 \cdot factorial(0))))) \\
&= 5 \cdot (4 \cdot (3 \cdot (2 \cdot (1 \cdot 1)))) \\
&= 5 \cdot (4 \cdot (3 \cdot (2 \cdot 1))) \\
&= 5 \cdot (4 \cdot (3 \cdot 2)) \\
&= 5 \cdot (4 \cdot 6) \\
&= 5 \cdot 24 \\
&= 120
\end{aligned}
$$

```
public static int factorial(int n) throws Illeg
  if (n < 0)
    throw new IllegalArgumentException( );
  else if (n == 0)
    return 1;
  else
    return n * factorial(n−1);
}
```

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



Types of Recursions

- **Linear recursion**
  - A method makes at most one recursive call.
  - ***Tail recursion***: a linear recursion in which a recursive call is the method's last operation.
- **Binary recursion**
  - A method makes two recursive calls.
- **Multiple recursion**
  - A method makes multiple recursive calls (usually more than two).

Linear Recursion

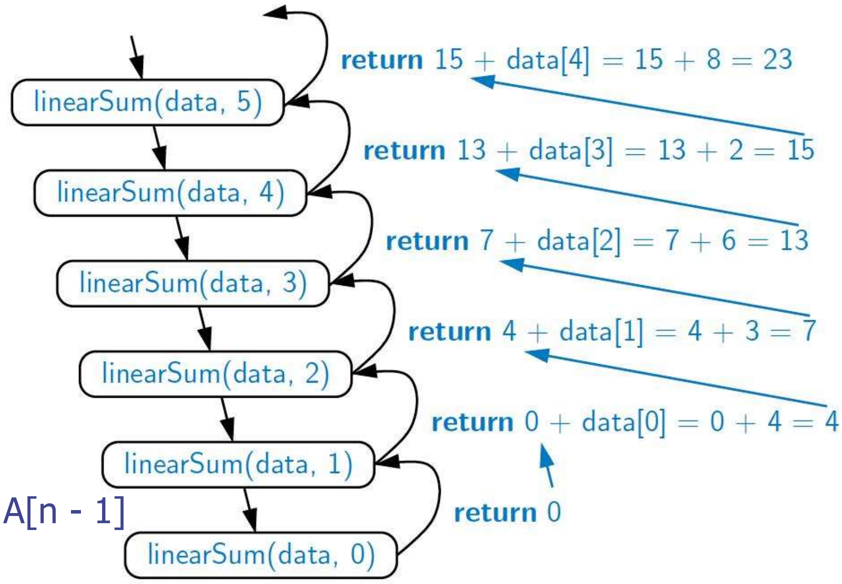
- Test for base cases
  - Begin by testing for a set of base cases (there should be at least one).
  - Every possible chain of recursive calls must eventually reach a base case, and the handling of each base case should not use recursion.
- Recur once
  - Perform a single recursive call
  - This step 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
    $0 \leq n \leq |A|$
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
  }
}
```

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

Is Factorial Tail-Recursive?

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

Tail Recursive Factorial

```
public static int factTailRec(int n, int result) {
    if (n <= 1)
        return result;
    else
        return factTailRec(n - 1, n * result);
}
```


Binary Recursion

- Binary recursion occurs whenever there are **two** recursive calls for each non-base case.

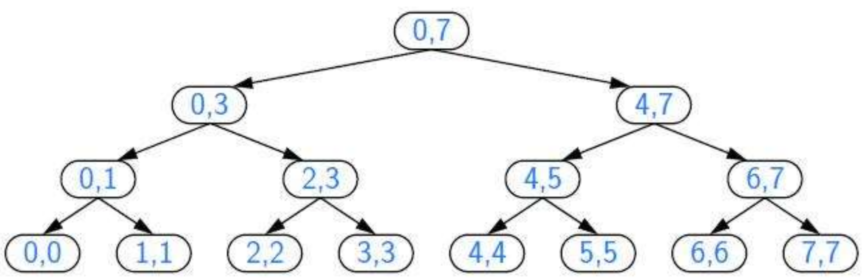


Figure 5.13: Recursion trace for the execution of binarySum(data, 0, 7).

Another Binary Recursive Method

```
/** 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);
  }
}
```

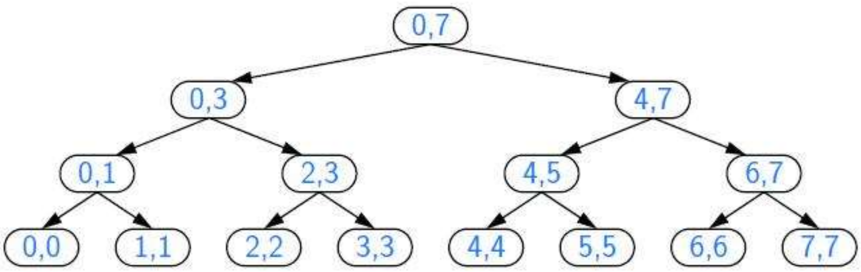


Figure 5.13: Recursion trace for the execution of binarySum(data, 0, 7).

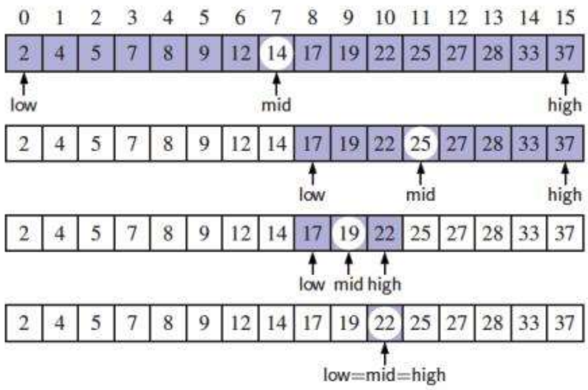
Binary Search

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

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 $k = 1$ **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

Example: English Ruler

- Print the ticks and numbers like an English ruler:

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

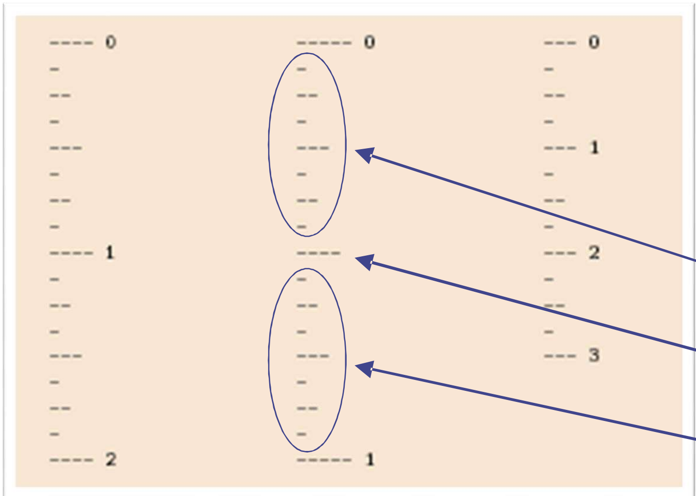
Slide by Matt Stallmann
included with permission.

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



drawInterval(length)

if(length > 0) then

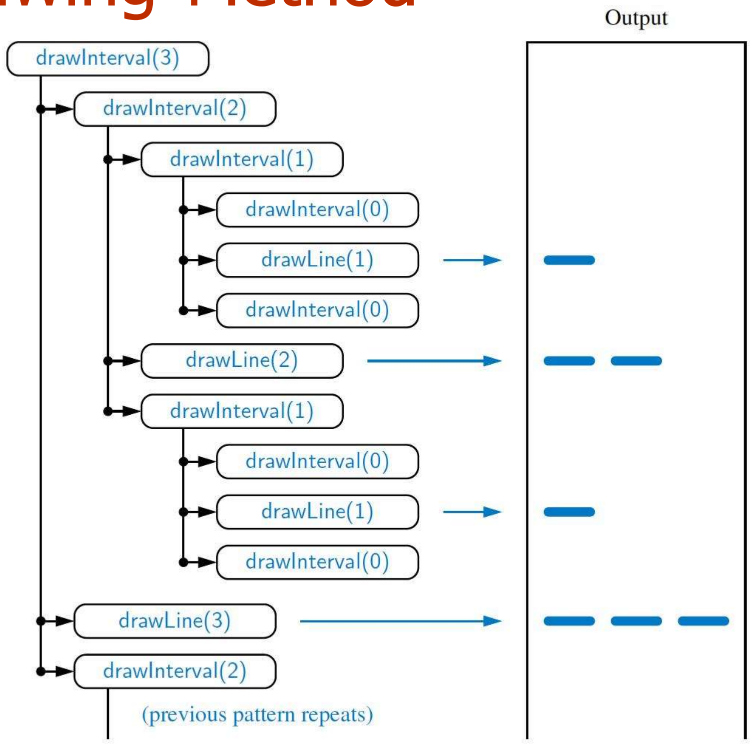
drawInterval (length – 1)

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
  - An single tick of length L
  - An interval with a central tick length L–1



Java Implementation (1)

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

// draw one tick
public static void drawLine(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");
}
```

Java Implementation (2)

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

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

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

Deciding Whether to Use a Recursive Solution

- The main issues are the efficiency and the clarity of the solution.
- For many problems, a recursive solution is simpler and more natural for the programmer to write.
- However, a recursive solution usually has more "overhead" than a non-recursive solution because of the number of method calls
  - Each call involves processing to create and dispose of the activation record, and to manage the run-time stack