LECTURE 20

Conclusion of the introduction to recursion

Over the weekend

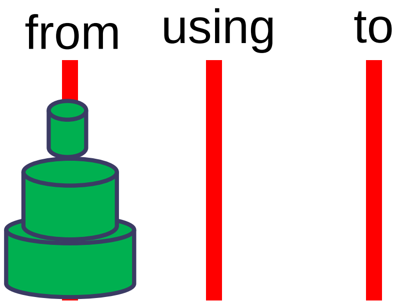
- Read Ch 5. You should just sail through! (Compare our *toh* solution with that in the book.)
- Implement our *toh* (should take 10-15 minutes)
- Find the highest number *n* for which you can solve *toh* in 8 hours

Plan

- Finishing with ToH (including the prove of the inherent exponential nature of the solution)
- A few prosaic examples in Java
  - String length
  - Arrays and lists traversing
- A few poetic examples
  - Binary search
  - GCD
  - More to come at the recitation next week!
- … Pitfalls and limitations in the application of recursion

toh


```
toh(string from, string using, string to, n)
  {
    if n > 0
        {
          toh(from, to, using, n-1)
          print(from, "->" to)
          toh(using, from, to, n-1)
         }
     }
```



toh execution (*in-order*)

from using to
toh("1", "2", "3", 3)

from using to
{ toh("1", "3", "2", 2)

1 -> 3

from using to
toh("2", "1", "3", 2) }

from using to
{ toh("1", "2", "3", 1) **1 -> 2**

from using to
toh("3", "1", "2", 1) }

from using to
{ toh("2", "3", "1", 1) **2 -> 3**

from using to
toh("1", "2", "3", 1) }

1 -> 3 **3 -> 2** **2 -> 1** **1 -> 3**

```
toh(from, to, using, n-1)
print(from, "=>" to)
toh(using, from, to, n-1)
```

toh complexity

- Our solution: $2^n - 1 = O(2^n)$ disk moves.
- The best possible solution: $2^n - 1$ disk moves.
- Pretty heavy problem!

Recursive Thinking (cont.)

Recursive Algorithm to Search an Array

`if` the array is empty

return -1 as the search result

`else if` the middle element matches the target

return the subscript of the middle element as the result

`else if` the target is less than the middle element

recursively search the array elements before the middle element and return the result

`else`

recursively search the array elements after the middle element and return the result

Steps to Design a Recursive Algorithm

- There must be at least one case (the base case), hat can be solved directly
- Find a way to reduce a problem of a given size *n* to smaller versions of the same problem
- Identify the base case and provide a solution to it
- Combine the solutions to the smaller problems to solve the larger problem

Proving that a Recursive Method is Correct

- Proof by induction
  - Prove the theorem is true for the base case
  - Show that if the theorem is assumed true for n=k, then it must be true for n=k+1
- Recursive proof is a form of the proof by induction
  - Verify the base case is recognized and solved correctly
  - Verify that each recursive case makes progress towards the base case
  - Verify that if all smaller problems are solved correctly, then the original problem also is solved correctly

Recursive algorithm for finding the length of a string

`if` the string is empty (has no characters)

the length is 0

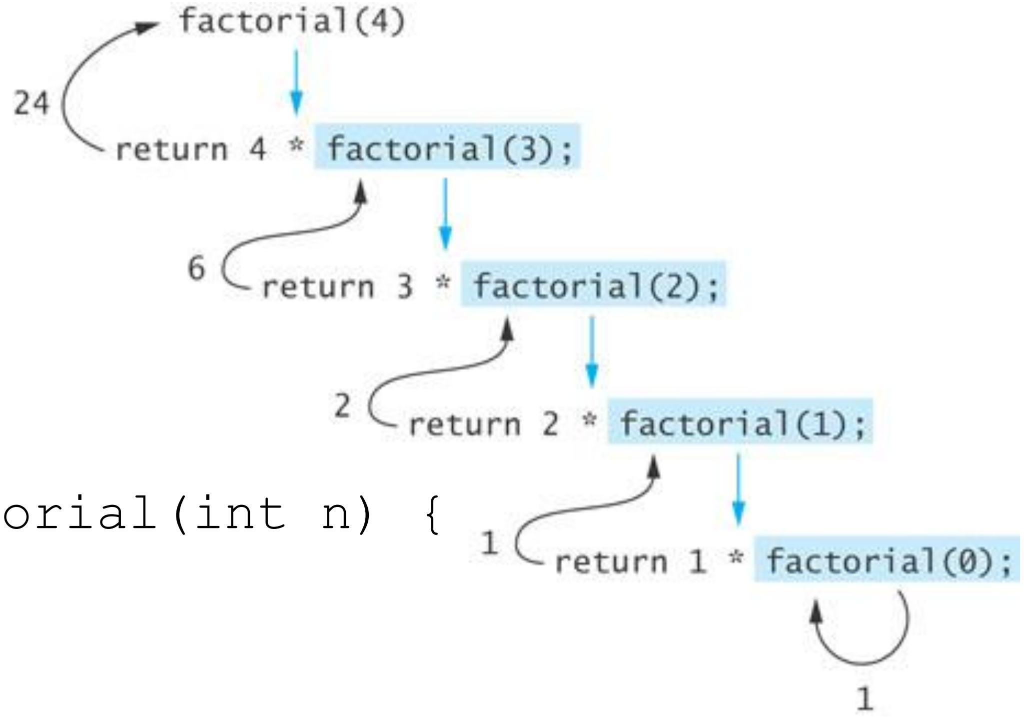
`else`

the length is 1 plus the length of the string that excludes the first character

Recursive Algorithm for Finding the Length of a String (cont.)

```
/** Recursive method length
    @param str The string
    @return The length of the string
*/
public static int length(String str) {
   if (str == null || str.equals(""))
      return 0;
   else
      return 1 + length(str.substring(1));
}
```

Factorial of *n*: *n*!




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

Iterative `factorial` Method

```
/** Iterative factorial method.
    pre: n >= 0
    @param n The integer whose factorial is being computed
    @return n!
*/
public static int factorialIter(int n) {
    int result = 1;
    for (int k = 1; k <= n; k++)
        result = result * k;
    return result;
}
```

Infinite Recursion and Stack Overflow

- If we call method `factorial` with a negative argument, the recursion will not terminate because `n` will never equal `0`
- If a program does not terminate, it will eventually throw the `StackOverflowError` exception
- **Make sure your recursive methods are constructed so that a stopping case is always reached**

Recursive Algorithm for Calculating the Greatest Common Divisor

- The greatest common divisor (gcd) of two numbers is the largest integer that divides both numbers
  - The gcd of 20 and 15 is 5
  - The gcd of 36 and 24 is 12
  - The gcd of 38 and 18 is 2
- If m > n then gcd(m, n) = gcd(m-n, n)

Recursive Algorithm for Calculating gcd (from the book)

- Given two positive integers m and n (m > n)

  `if` n is a divisor of m

  gcd(m, n) = n

  `else`

  gcd (m, n) = gcd (n, m % n)

Recursive Algorithm for Calculating gcd (from the book)

```
/** Recursive gcd method (in RecursiveMethods.java).
    pre: m > 0 and n > 0
    @param m The larger number (Irrelevant!)
    @param n The smaller number
    @return Greatest common divisor of m and n
*/
public static double gcd(int m, int n) {
    if (m % n == 0)
        return n;
    else if (m < n)
        return gcd(n, m); // Transpose arguments.
    else
        return gcd(n, m % n);
}
```

Recursive Algorithm for Calculating gcd (*not* from the book)

```
/** Recursive gcd method with no division required
    pre: m > 0 and n > 0
*/
public static double gcd(int m, int n)
{
    if (m < n)
        return gcd (n, m);
     else
        if (m == n)
          return n;
     else /* m > n */
       if (n == 1)
            return 1;
       else
         return gcd(m-n, n);
}
```

Please program this and tell me on Monday if it works!

Efficiency of Recursion

- Recursive methods have slower execution times relative to their iterative counterparts
- The overhead for loop repetition is smaller than the overhead for a method call and return (Why?)
- If it is easier to conceptualize an algorithm using recursion, then you should code it as a recursive method
- The reduction in efficiency does not outweigh the advantage of readable code that is easy to debug

Fibonacci Numbers

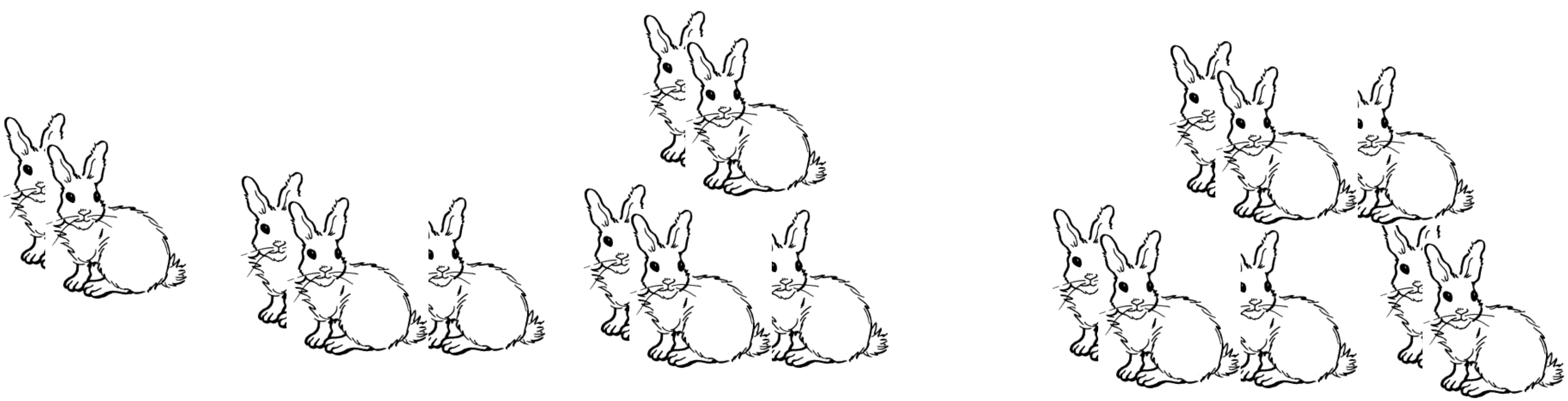
- Fibonacci numbers

  $fib_1 = 1$

  $fib_2 = 1$

  $fib_n = fib_{n-1} + fib_{n-2}$

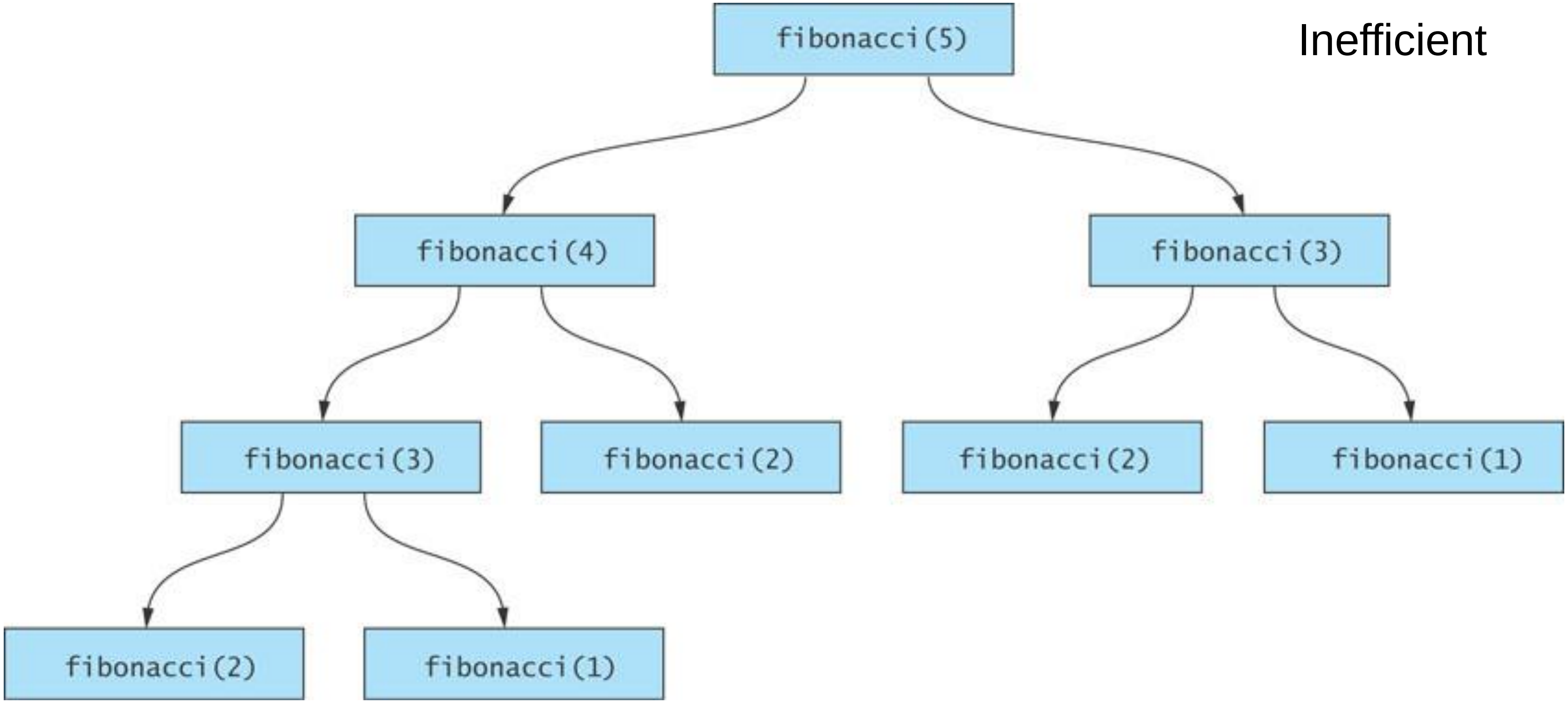
- 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, …



An Exponential Recursive `fibonacci` Method

```java
/** Recursive method to calculate Fibonacci numbers
    (in RecursiveMethods.java).
    pre: n >= 1
    @param n The position of the Fibonacci number being calculated
    @return The Fibonacci number
*/
public static int fibonacci(int n) {
    if (n <= 2)
        return 1;
    else
        return fibonacci(n - 1) + fibonacci(n - 2);
}
```

Inefficiency of Recursion: Exponential `fibonacci`



Inefficient

An O(n) Recursive `fibonacci` Method

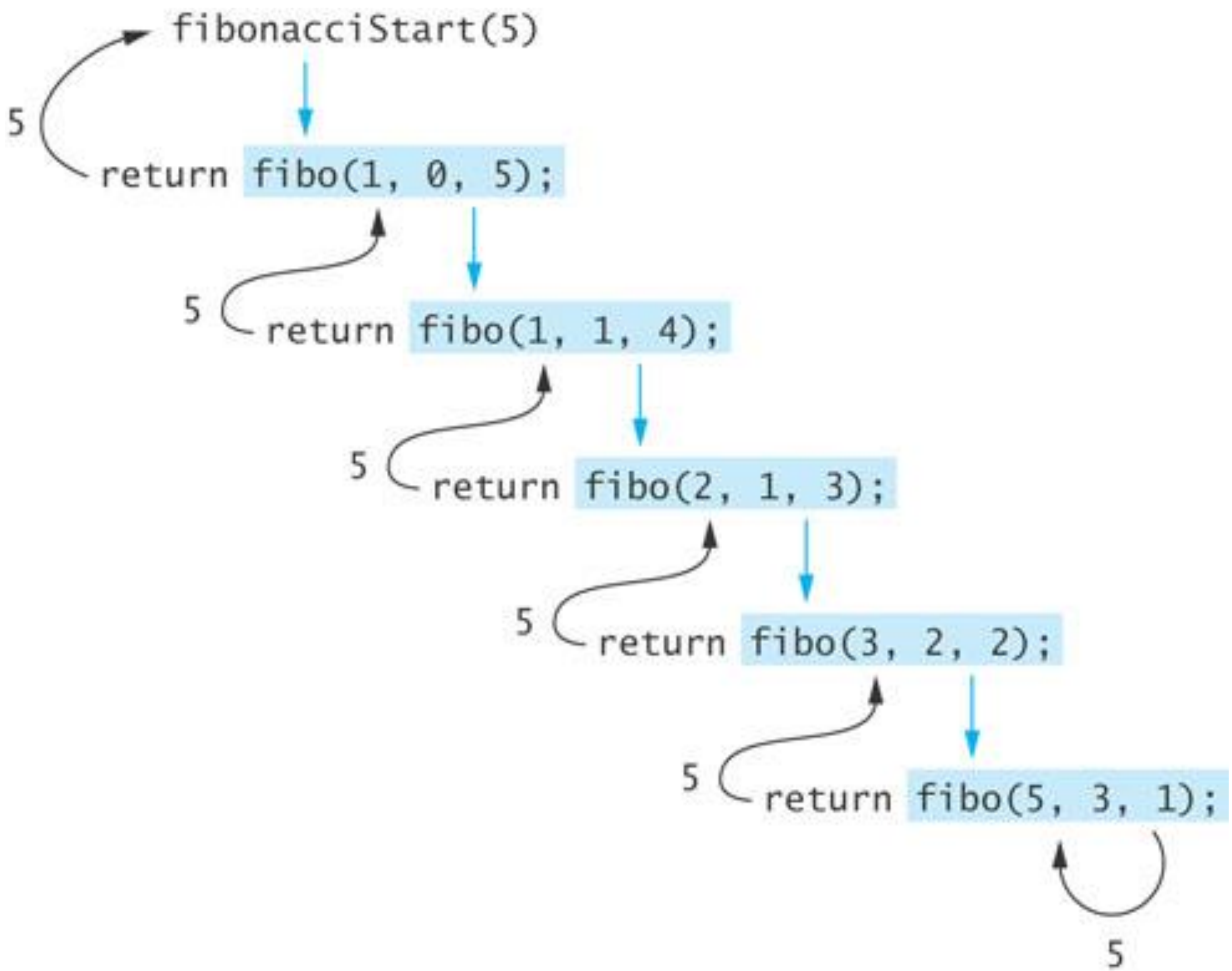
```
/** Recursive O(n) method to calculate Fibonacci numbers
    (in RecursiveMethods.java).
    pre: n >= 1
    @param fibCurrent The current Fibonacci number
    @param fibPrevious The previous Fibonacci number
    @param n The count of Fibonacci numbers left to calculate
    @return The value of the Fibonacci number calculated so far
*/
private static int fibo(int fibCurrent, int fibPrevious, int n) {
    if (n == 1)
        return fibCurrent;
    else
        return fibo(fibCurrent + fibPrevious, fibCurrent, n - 1);
}
```

An O(n) Recursive `fibonacci` Method (cont.)

- In order to start the method executing, we provide a non-recursive wrapper method:

```
/** Wrapper method for calculating Fibonacci numbers
(in RecursiveMethods.java).
    pre: n >= 1
    @param n The position of the desired Fibonacci
      number
    @return  The value of the nth Fibonacci number
*/
public static int fibonacciStart(int n) {
     return fibo(1, 0, n);
}
```

Efficiency of Recursion: O(n) fibonacci



Efficient

Efficiency of Recursion: O(n) `fibonacci`

- Method `fibo` is an example of *tail recursion* or *last-line recursion*
- When recursive call is the last line of the method, arguments and local variable do not need to be saved in the activation frame