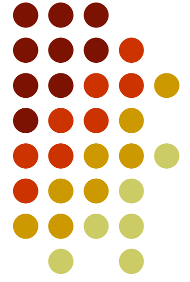
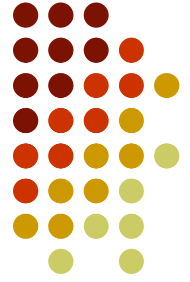
Recursive Functions



- The Fibonacci function shown previously is **recursive**, that is, it calls itself
  - Each call to a recursive method results in a *separate* instance (invocation) of the method, with its own input and own local variables

Function Analysis for call `fib(5)`



```
public static int fib(int n)
  if (n == 0 || n == 1)
    return n
  else
    return fib(n-1) + fib(n-2)
```

5

fib(5)

3 2

fib(4) fib(3)

2 1 1 1

fib(3) fib(2) fib(2) fib(1)

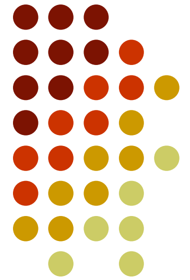
1 1 1 0 1 0

fib(2) fib(1) fib(1) fib(0) fib(1) fib(0)

1 0

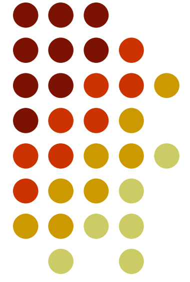
fib(1) fib(0)

Recursive Function Calls on the Stack



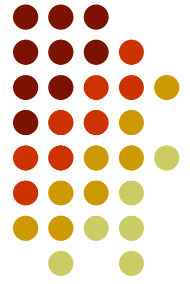
- When a method is called it is pushed onto the **call stack**
  - Subsequent recursive invocations are also pushed onto the call stack
- Whenever a recursive invocation is made execution is switched to that method invocation
  - The call stack keeps track of the line number of the previous method where the call was made from
  - Once execution of one method invocation is finished it is removed from the call stack, and execution returns to the previous invocation

Anatomy of a Recursive Function



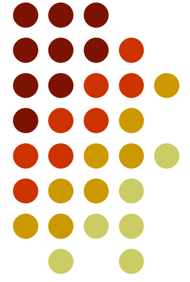
- A recursive function consists of two types of cases
  - A base case(s) and
  - A recursive case
- The base case is a small problem
  - The solution to this problem should not be recursive, so that the function is guaranteed to terminate
  - There can be more than one base case
- The recursive case defines the problem in terms of a smaller problem of the same type
  - The recursive case includes a recursive function call
  - There can be more than one recursive case

Finding Recursive Solutions



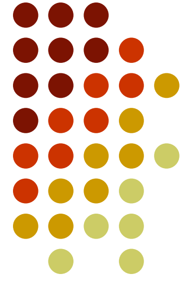
- Define the problem in terms of a smaller problem of the same type and
  - The recursive part
  - e.g. `return fib(n-1) + fib(n-2);`
- A small problem where the solution can be easily calculated
  - This solution should not be recursive
  - The "base case"
  - e.g. `if (n == 0 || n == 1) return n;`

Steps Leading to Recursive Solutions



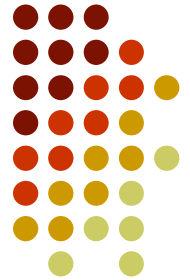
- How can the problem be defined in terms of smaller problems of the same type?
- By how much does each recursive call reduce the problem size?
- What is the base case that can be solved without recursion?
- Will the base case be reached as the problem size is reduced?

Designing a recursive solution: Writing a String Backward



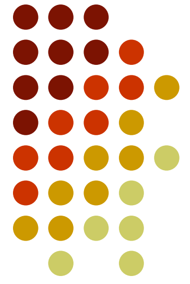
- **Problem:**
  - *Given a string of characters, write it in reverse order*
- **Recursive solution:**
  - How can the problem be defined in terms of smaller problems of the same type?
    - We could write the last character of the string and then solve the problem of writing first n-1 characters backward
  - By how much does each recursive call reduce the problem size?
    - Each recursive step of the solution diminishes by 1 the length of the string to be written backward
  - What is the base case that can be solved without recursion?
    - Base case: Write the empty string backward = Do nothing.
  - Will the base case be reached as the problem size is reduced?
    - Yes.

Designing a recursive solution: Writing a String Backward



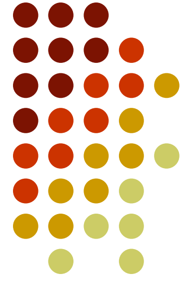
```
public static void writeBackward(String s, int size) {
// ------------------------------------------------------
// Writes a character string backward.
// Precondition: The string s contains size
// characters, where size >= 0.
// Postcondition: s is written backward, but remains
// unchanged.
// ------------------------------------------------------
  if (size > 0) {
    // write the last character
    System.out.println(s.substring(size-1, size));

    // write the rest of the string backward
    writeBackward(s, size-1);  // Point A
  }  // end if
  // size == 0 is the base case - do nothing
}  // end writeBackward
```

Designing a recursive solution: Writing a String Backward



- Execution of recursive method like `writeBackward()` can be traced using the **box trace:**
  - Useful for debugging and understanding recursive methods.
  - Simulates the work of computer.
  - A sequence of boxes: each box corresponds to an *activation record* = a record put on the call stack when a function is called containing values of parameters and local variables.
  - All recursive calls in the body of the recursive method are labeled with different letters (the label is then used to indicate which recursive call caused creation of a new box in the sequence.

Box trace of ***writeBackward("cat", 3)***



The initial call is made, and the method begins execution:

s = "cat" size = 3

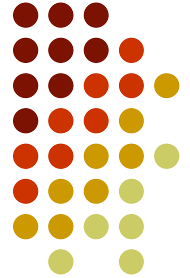
Outputline: **t**
Point A (writeBackward(s, size-1)) is reached, and the recursive call is made.
The new invocation begins execution:

s = "cat" size = 3	—A→	s = "cat" size = 2

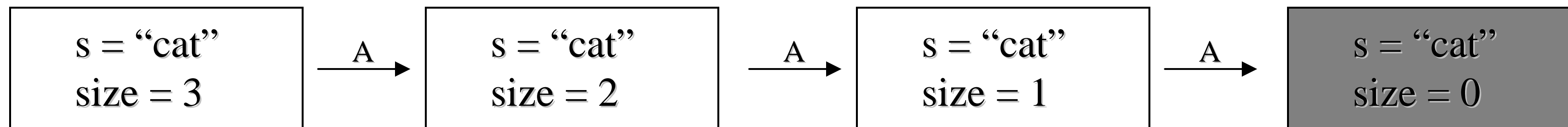
Outputline: **ta**
Point A is reached, and the recursive call is made.
The new invocation begins execution:

s = "cat" size = 3	—A→	s = "cat" size = 2	—A→	s = "cat" size = 1

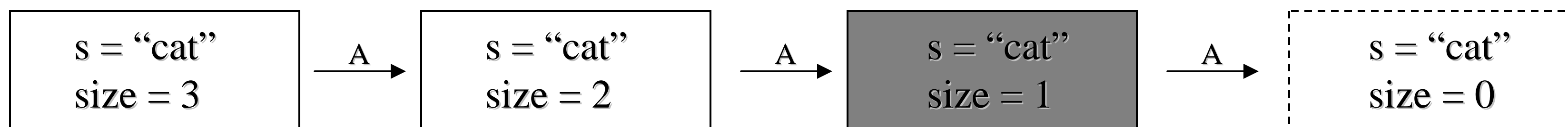
Box trace of *writeBackward("cat", 3)*



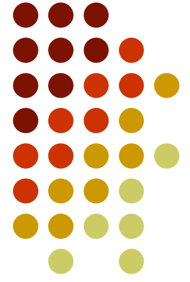
Output line: **tac**
Point A is reached, and the recursive call is made.
The new evocation begins execution:



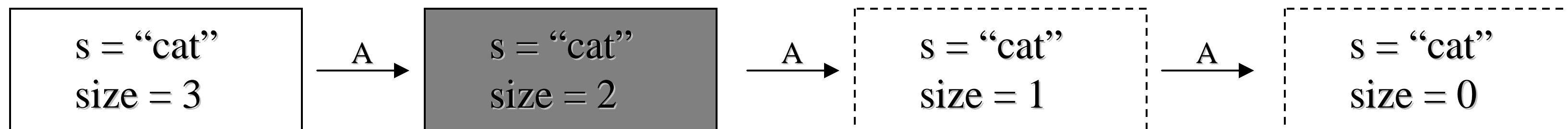
This is the base case, so this invocation completes.
Control returns to the calling box, which continues execution:



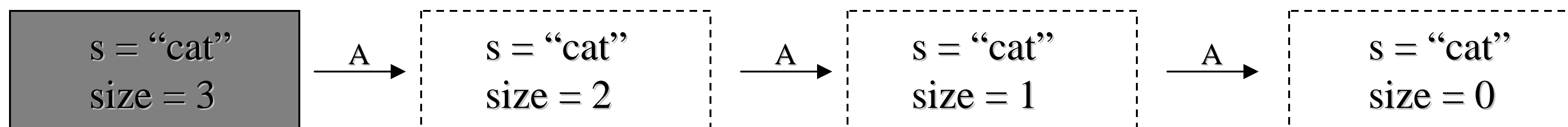
Box trace of *writeBackward("cat", 3)*



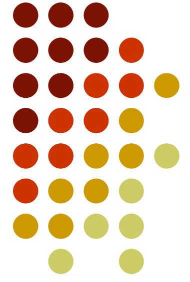
This invocation completes.
Control returns to the calling box, which continues execution:



This invocation completes.
Control returns to the calling box, which continues execution:



This invocation completes.
Control returns to the statement following the initial call.



A recursive definition of factorial *n!*

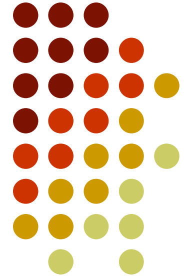
fact(n) = n*(n-1)*(n-2)* … *1

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

```
public static int fact (int n) {
   if (n==0) {
      return 1;
   }
   else {
      return n * fact(n-1); // Point A
   }
}
```

Figure 2.2

fact(3)



System.out.println(fact(3));
6

return 3*fact(2)
3* 2

return 2*fact(1)
2* 1

return 1*fact(0)
1* 1

return 1

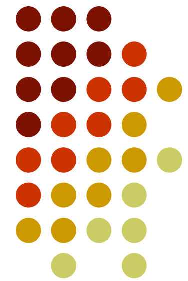


Figure 2.3

A box

```
n = 3
A: fact(n-1) = ?
return ?
```

Figure 2.4

The beginning of the box trace

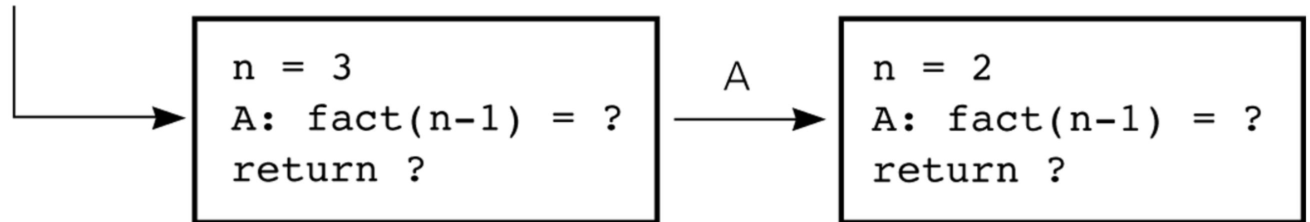
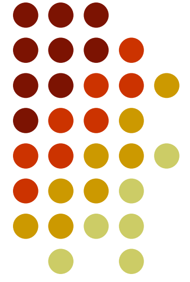
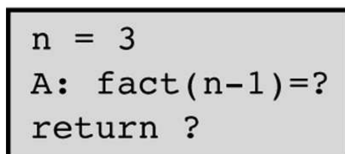


Figure 2.5a

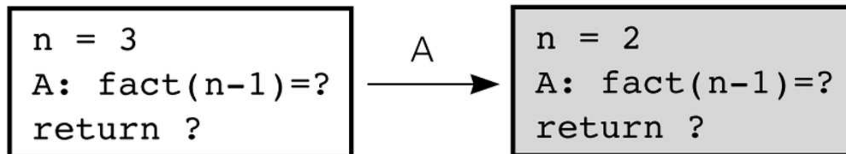
Box trace of ***fact(3)***



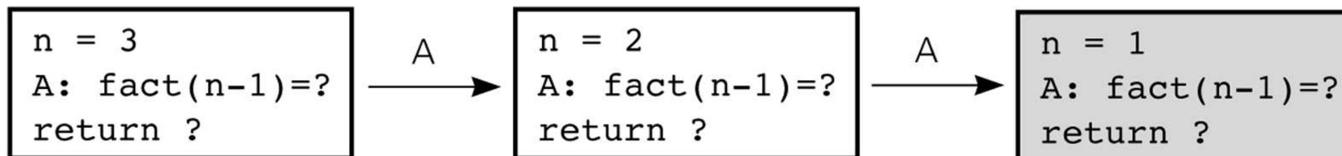
The initial call is made, and method `fact` begins execution:



At point A a recursive call is made, and the new invocation of the method `fact` begins execution:



At point A a recursive call is made, and the new invocation of the method `fact` begins execution:



At point A a recursive call is made, and the new invocation of the method `fact` begins execution:

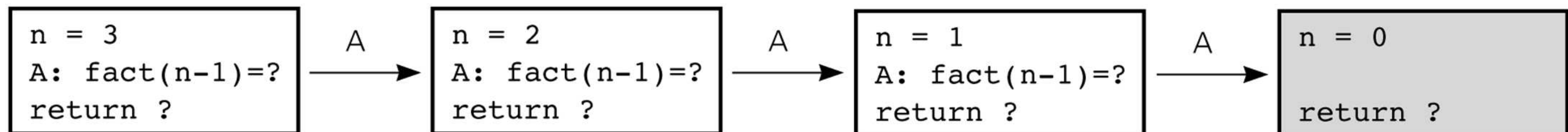
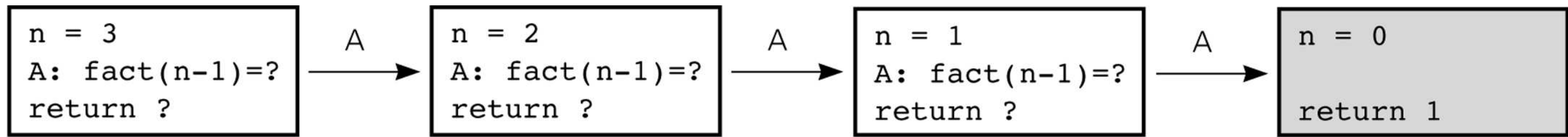
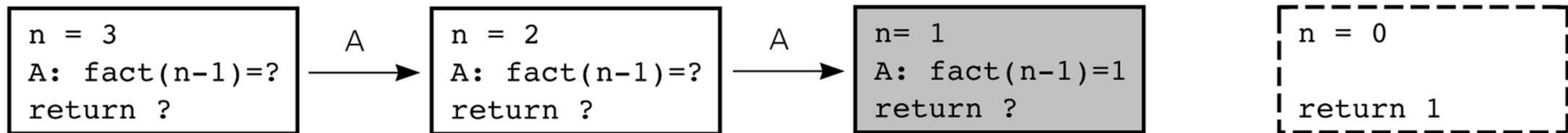


Figure 2.5b

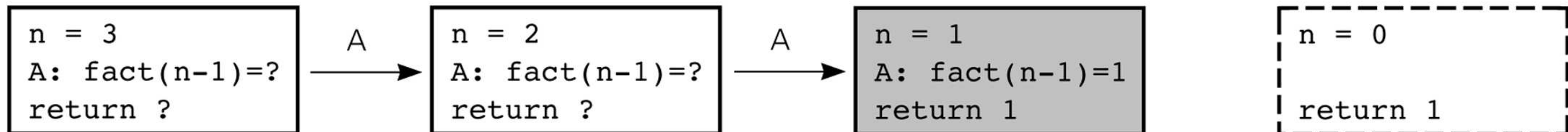
Box trace of ***fact(3)***



The method value is returned to the calling box, which continues execution:



The current invocation of `fact` completes:



The method value is returned to the calling box, which continues execution:

Figure 2.5c

Box trace of ***fact(3)***

The current invocation of `fact` completes:

The method value is returned to the calling box, which continues execution:

The current invocation `fact` completes:

The value 6 is returned to the initial call.