

CMPT 225

Recursion

Objectives

- Understand how the Fibonacci series is generated
 - Recursive Algorithms
 - Write simple recursive algorithms
 - Analyze simple recursive algorithms
 - Understand the drawbacks of recursion
 - Name other recursive algorithms and data structures
-

Recursive Solutions

- Recursion

- An extremely powerful problem-solving technique
- Breaks a problem in smaller identical problems
- An alternative to iteration
 - An iterative solution involves loops



Example: writing a string backwards

Iterative Solution

s	t	r	i	n	g	
---	---	---	---	---	---	--

g n i r t s

Recursive Solution

s	t	r	i	n	g	
---	---	---	---	---	---	--

g	n	i	r	t
---	---	---	---	---

 s

```
printReverseItr( String s){
```

```
    for (i=s.size()-1; i>=0; i--){  
        print s[i];  
    }
```

```
}
```

```
printReverseRec(String s){
```

```
    if (s.size() == 0)  
        return ; //if the string is empty return
```

```
    printReverseRec(suffix(s, 1)); //suffix(s, i) returns the ith suffix of s  
    print s[0];
```

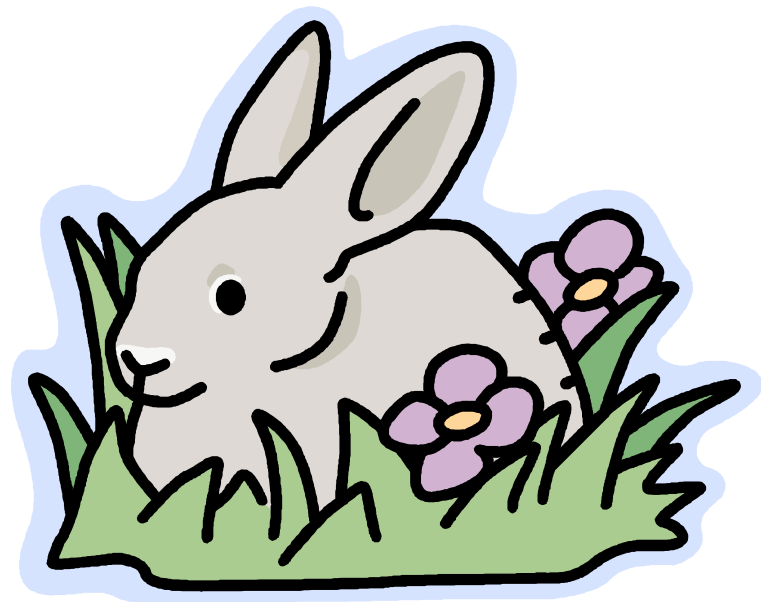
```
}
```

Recursive Solutions

- Facts about a recursive solution
 - A recursive method calls itself
 - Each recursive call solves an identical, but smaller, problem
 - A test for the base case enables the recursive calls to stop
 - Base case: a known case in a recursive definition
 - Eventually, one of the smaller problems must be the base case
-

Example: Rabbits

- What happens if you put a pair of rabbits in a field?
 - You get more rabbits!
- Assume that rabbits take one month to reach maturity and that
- Each pair of rabbits produces another pair of rabbits one month after mating.
- **That is:** each pair will produce a new pair 2 months after it was born (and every month after that)



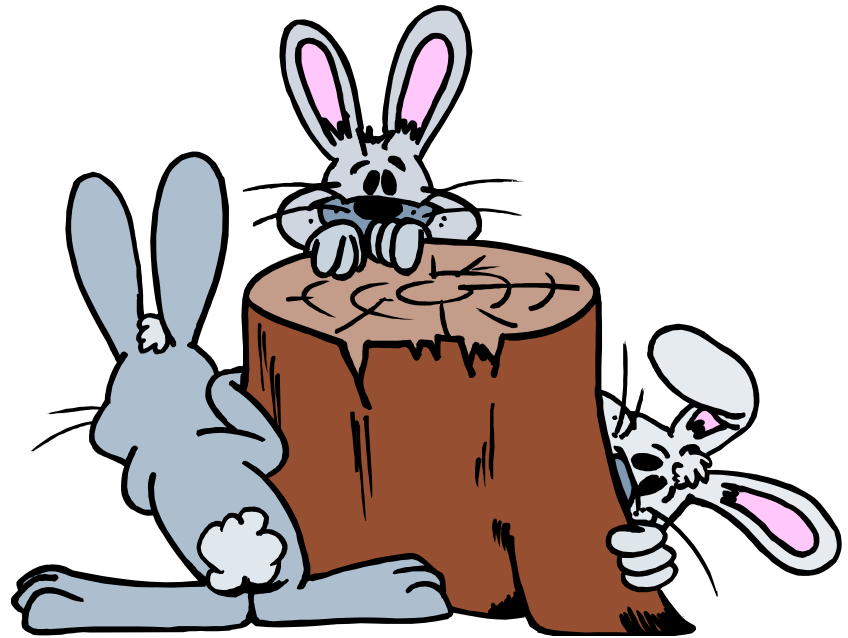
Example: Rabbits

- How many pairs of rabbits are there after five months?
- Month 1: 1 pair
- Month 2: 1 pair
 - after one month the rabbits are mature and can mate
- Month 3: 2 pairs
 - the first pair gives birth to a new pair of rabbits
- Month 4: 3 pairs
 - the first pair gives birth to a new pair, her first children are now mature
- Month 5: 5 pairs



Example: Rabbits

- After 5 months there are 5 pairs of rabbits
 - i.e. the number of pairs at 4 months (3) plus the number of pairs at 3 months (2)
 - Why?
- We have count existing pairs of rabbits (the same number as previous month) + the new pairs (the same number as 2 months ago, as only those rabbits can now produce children)
- This series of numbers is called the Fibonacci series



$$\text{rabbit}(n) = \text{rabbit}(n-1) + \text{rabbit}(n-2)$$

Multiplying Rabbits (The Fibonacci Sequence)

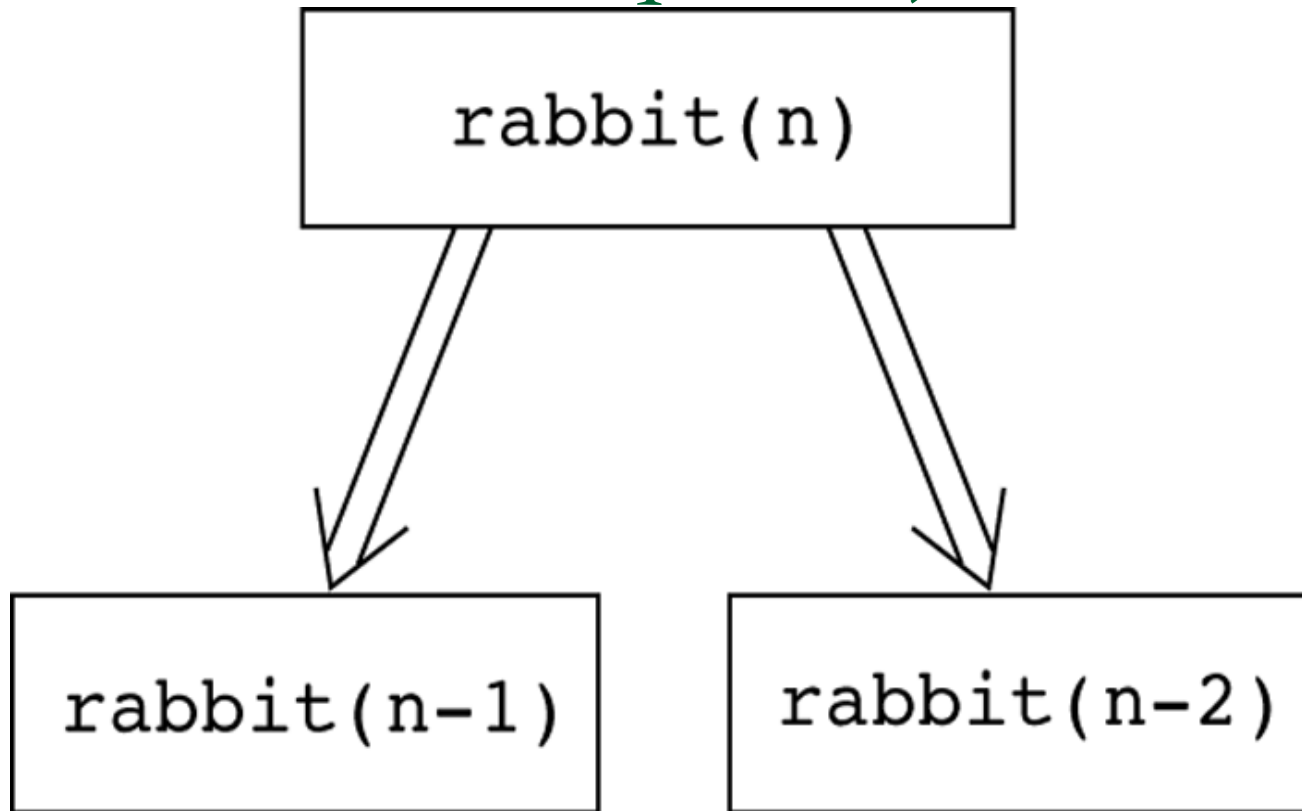


Figure 3-10

Recursive solution to the rabbit problem

Fibonacci Sequence

- The n^{th} number in the Fibonacci sequence, $fib(n)$, is:
 - 0 if $n = 0$, and 1 if $n = 1$
 - $fib(n - 1) + fib(n - 2)$ for $n > 1$
 - So what, say, is $fib(23)$, or how many pairs of rabbits would there be after 23 months?
 - This would be easy if only we knew $fib(22)$ and $fib(21)$
 - If we did then the answer is just $fib(22) + fib(21)$
 - What happens if we write a function to calculate Fibonacci numbers in this way?
-

Calculating the Fibonacci Sequence

- Here is a function to return nth number in the Fibonacci sequence

□ e.g. $\text{fib}(1) = 1$, $\text{fib}(3) = 2$, $\text{fib}(5) = 5$, $\text{fib}(8) = 21$, ...

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

The function
calls itself!!!

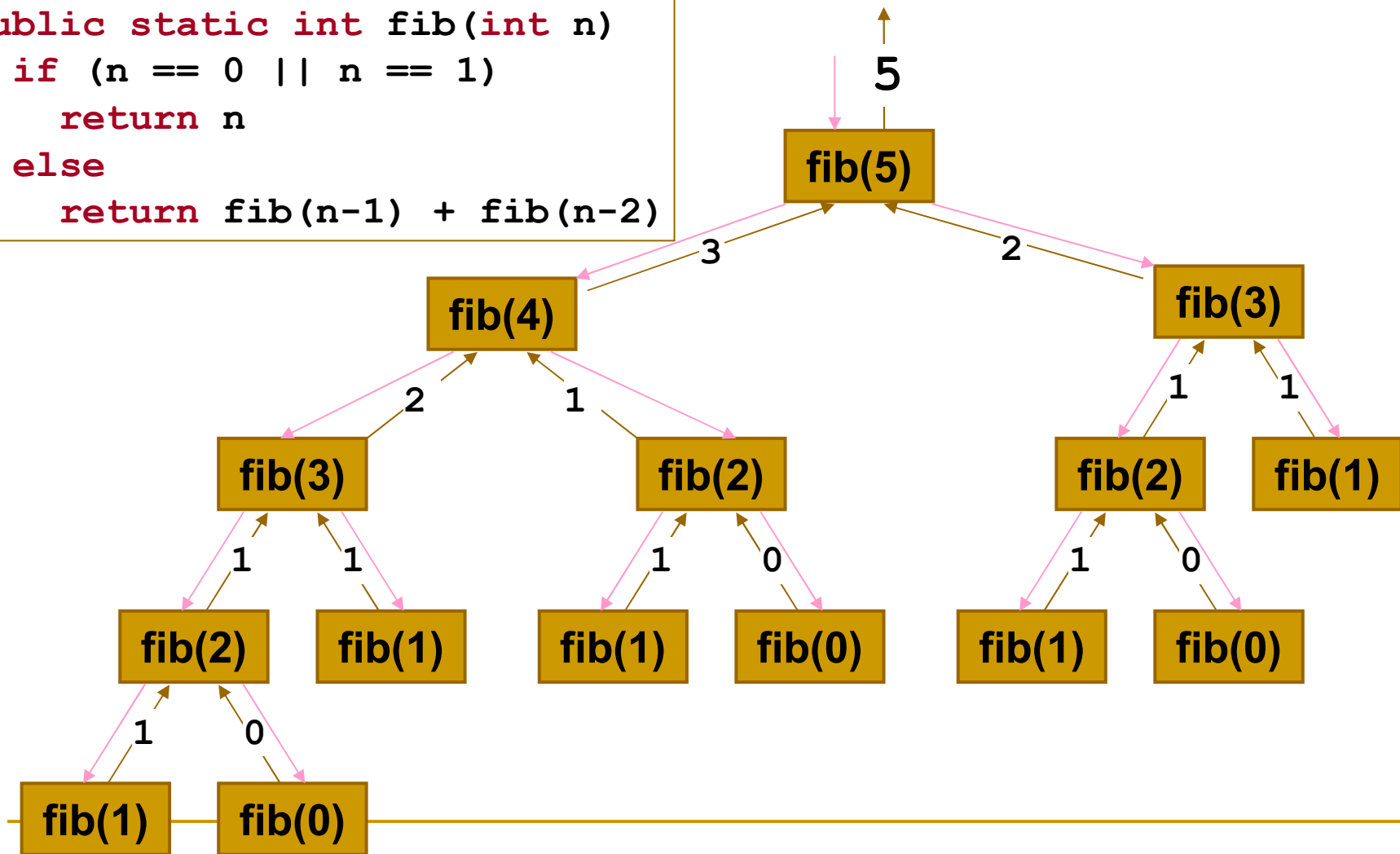
- Notice this looks just like the description of the Fibonacci sequence given previously, but does it work!?

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)
```



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
-

STACK

```
main() {  
  
    printReverseRec("cats");  
0)    ...  
}  
  
printReverseRec(String s){  
  
1)    if (s.size() == 0)  
2)        return ;  
3)    printReverseRec(suffix(s, 1));  
4)    print s[0];  
  
    return;  
}
```

s t a c

S="" Line=4
S="s" Line=4
S="ts" Line=4
S="ats" Line=4
S="cats" Line=0

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.
-

A recursive definition of factorial $n!$

$$\text{fact}(n) = n * (n-1) * (n-2) * \dots * 1$$

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

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

Figure 2.2

fact(3)

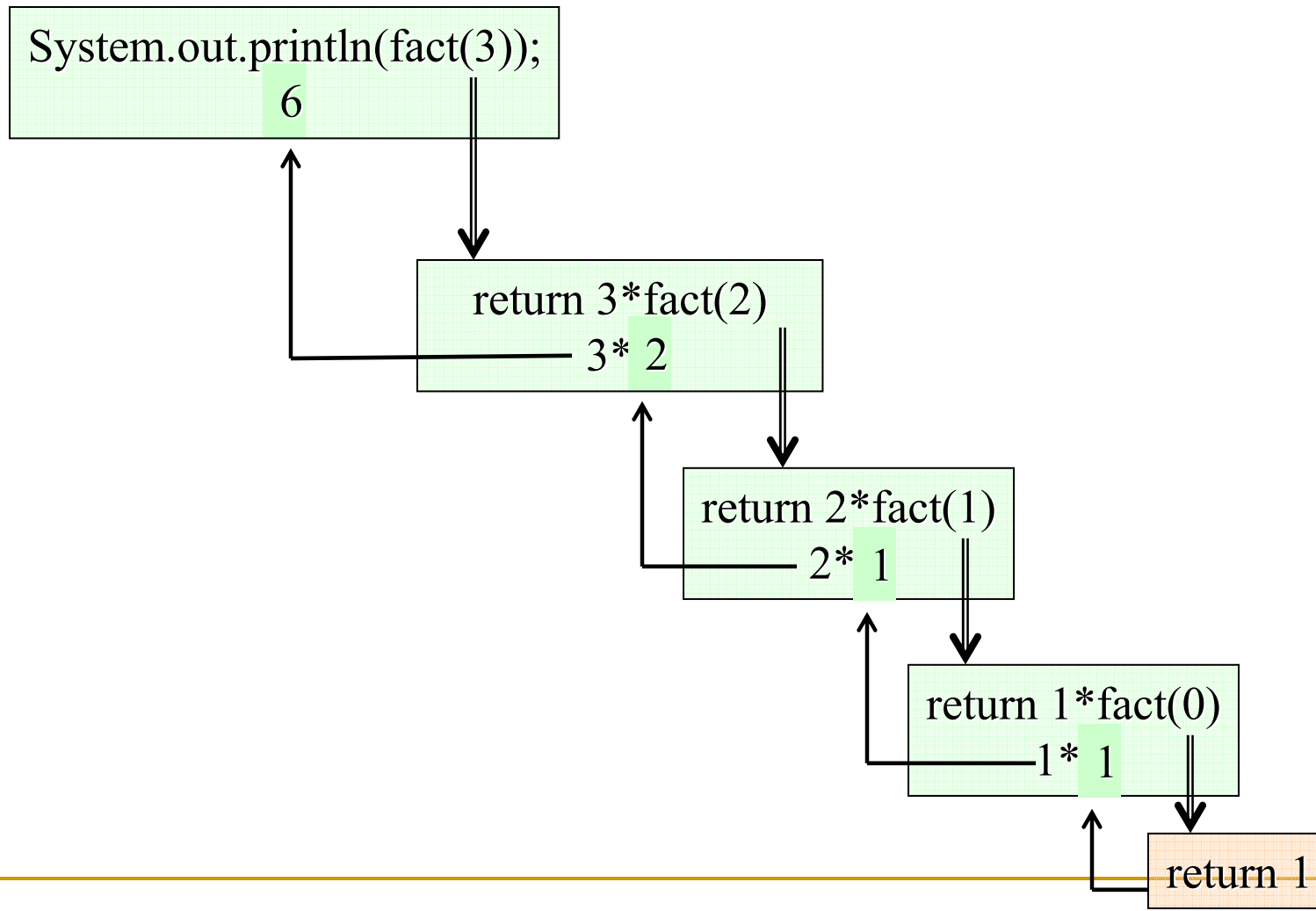


Figure 2.3

A box

```
n = 3
```

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

```
return ?
```

Figure 2.4

The beginning of the box trace

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

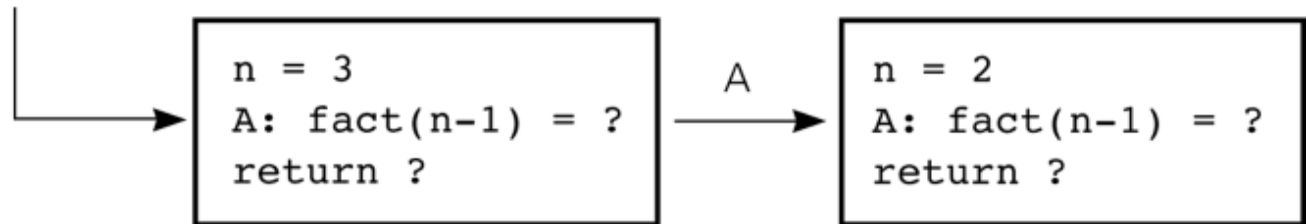


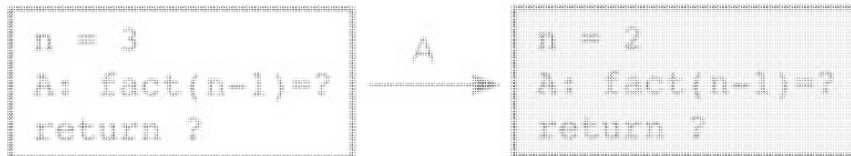
Figure 2.5a

Box trace of *fact* (3)

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

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

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:

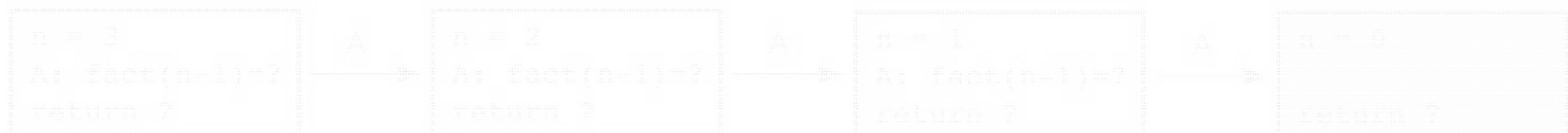
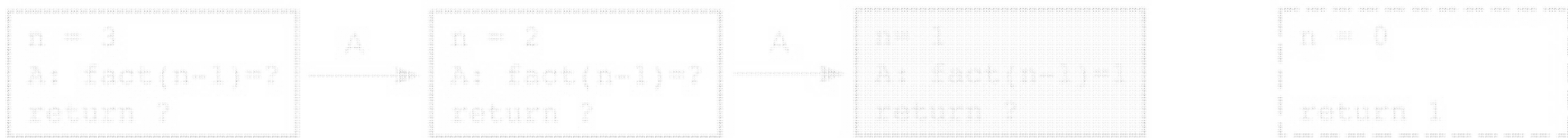


Figure 2.5b

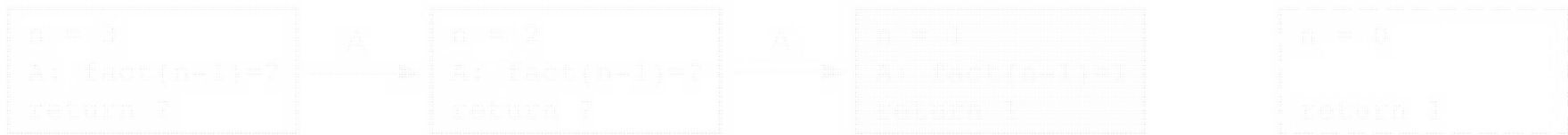
Box trace of *fact* (3)



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



The current invocation of *fact* completes:



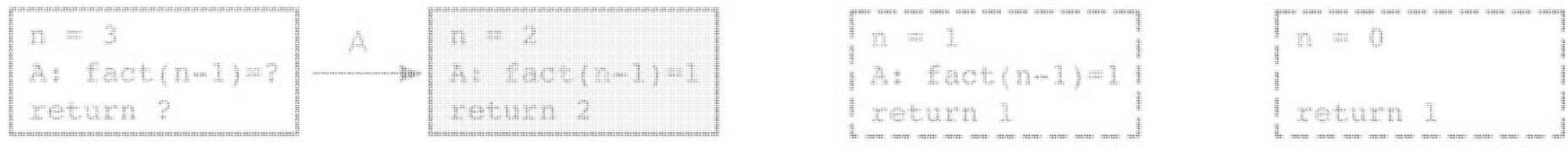
The method value is returned to the calling box, which continues execution:



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:



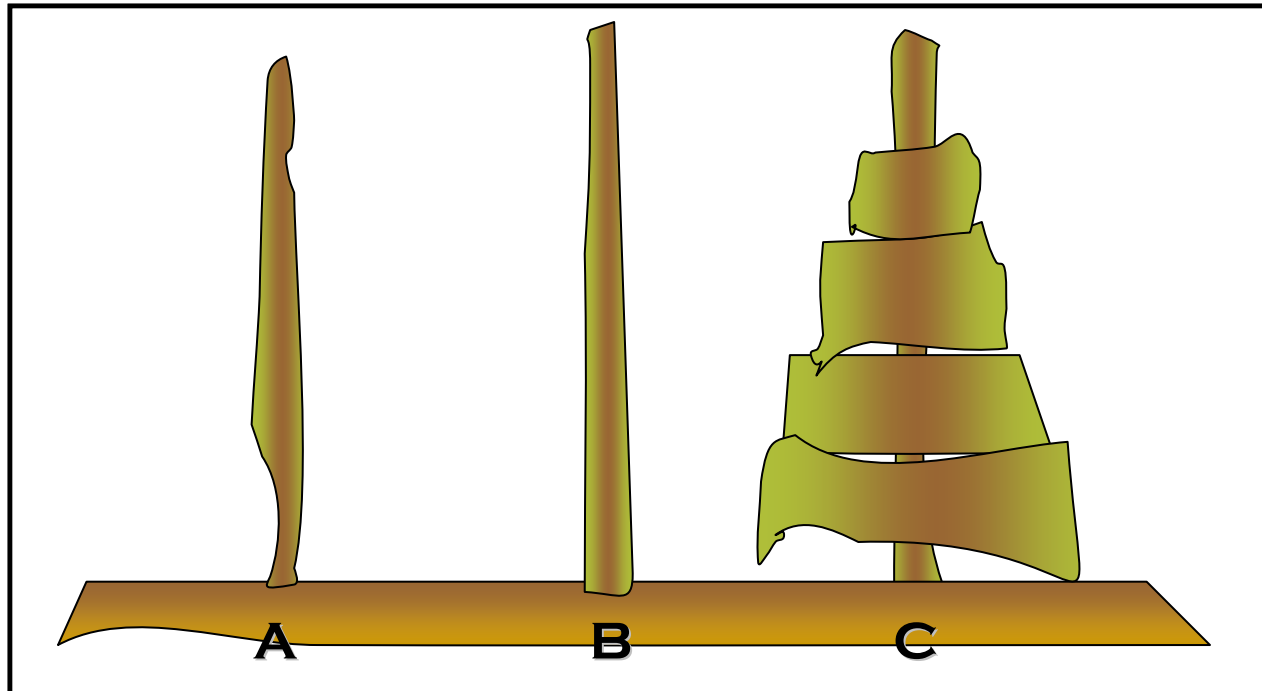
The current invocation *fact* completes:



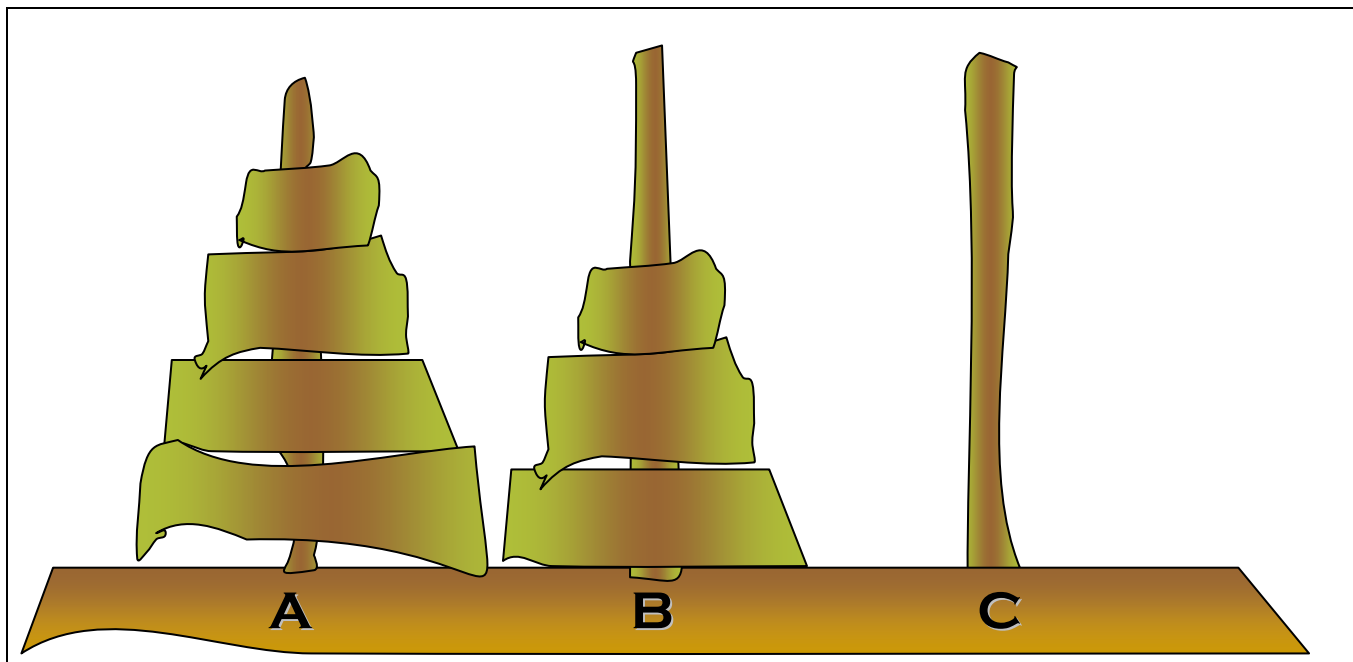
The value 6 is returned to the initial call.

Towers of Hanoi

- Move n (4) disks from pole A to pole C
- such that a disk is never put on a smaller disk



- Move n (4) disks from A to C
 - Move $n-1$ (3) disks from A to B
 - Move 1 disk from A to C
 - Move $n-1$ (3) disks from B to C



Hanoi towers

```
public static void solveTowers(int count, char source,
                                char destination, char spare) {
    if (count == 1) {
        System.out.println("Move top disk from pole " + source +
                            " to pole " + destination);
    }
    else {
        solveTowers(count-1, source, spare, destination); // X
        solveTowers(1, source, destination, spare);       // Y
        solveTowers(count-1, spare, destination, source); // Z
    } // end if
} // end solveTowers
```

Recursion tree:

The order of recursive calls that results from *solveTowers*(3,A,B,C)

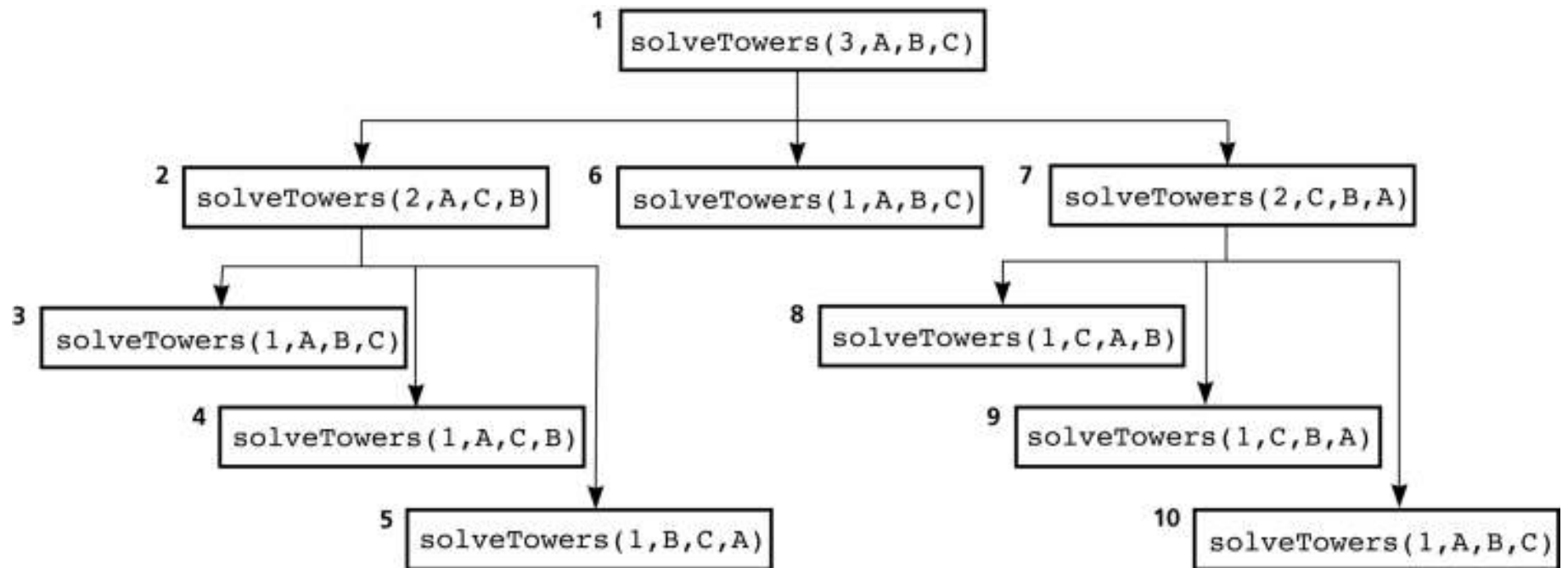
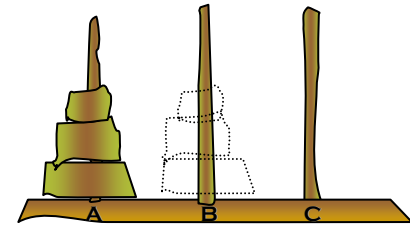


Figure 2.21a

Box trace of *solveTowers*(3, 'A', 'B', 'C')

The initial call 1 is made, and *solveTowers* begins execution:

```
count = 3
source = A
dest = B
spare = C
```

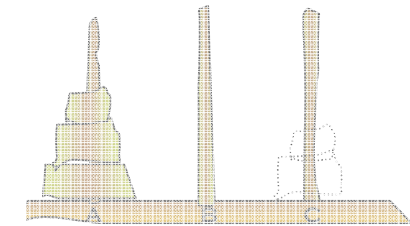


At point X, recursive call 2 is made, and the new invocation of the method begins execution:

```
count = 3
source = A
dest = B
spare = C
```

X

```
count = 2
source = A
dest = C
spare = B
```



At point X, recursive call 3 is made, and the new invocation of the method begins execution:

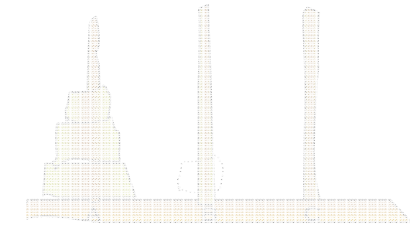
```
count = 3
source = A
dest = B
spare = C
```

X

```
count = 2
source = A
dest = C
spare = B
```

X

```
count = 1
source = A
dest = B
spare = C
```



This is the base case, so a disk is moved, the return is made, and the method continues execution:

```
count = 3
source = A
dest = B
spare = C
```

X

```
count = 2
source = A
dest = C
spare = B
```

X

```
count = 1
source = A
dest = B
spare = C
```

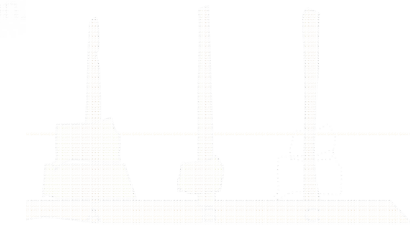
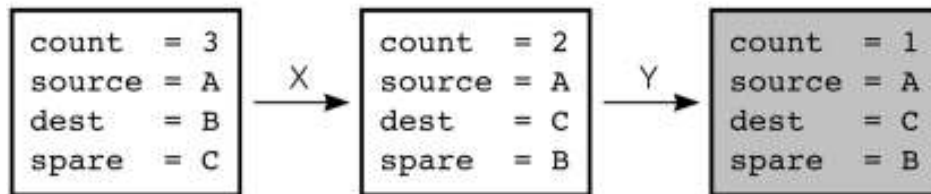
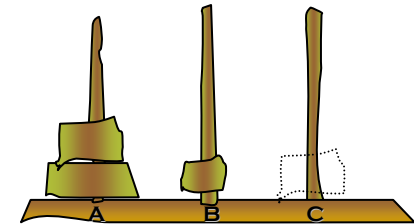


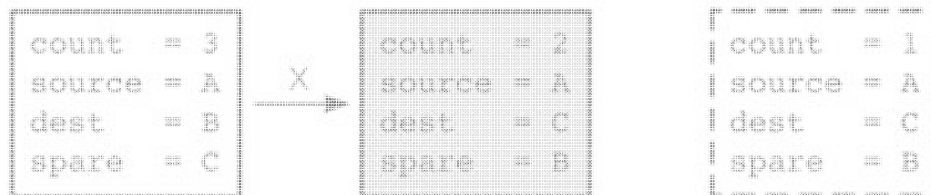
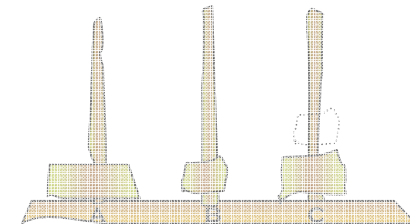
Figure 2.21b

Box trace of *solveTowers*(3, 'A', 'B', 'C')

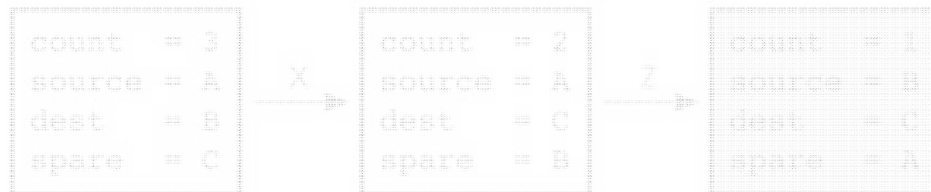
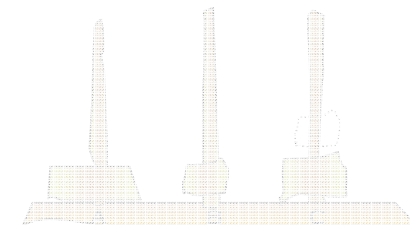
At point Y, recursive call 4 is made, and the new invocation of the method begins execution:



This is the base case, so a disk is moved, the return is made, and the method continues execution.



At point Z, recursive call 5 is made, and the new invocation of the method begins execution:



This is the base case, so a disk is moved, the return is made, and the method continues execution.

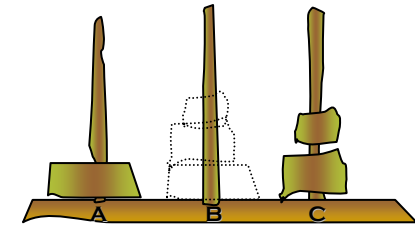


Figure 2.21c

Box trace of *solveTowers*(3, 'A', 'B', 'C')

This invocation completes, the return is made, and the method continues execution.

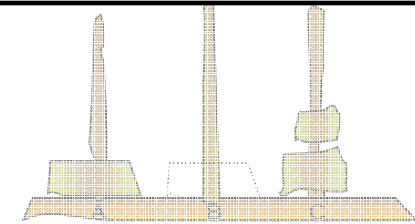
count = 3	count = 2	count = 1
source = A	source = A	source = B
dest = B	dest = C	dest = C
spare = C	spare = B	spare = A



At point Y, recursive call 6 is made, and the new invocation of the method begins execution:

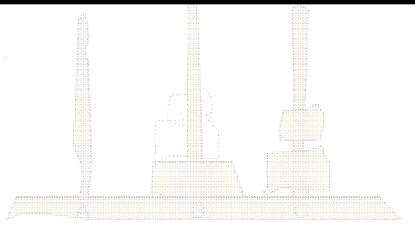
count = 3	count = 1
source = A	source = A
dest = B	dest = B
spare = C	spare = C

Y



This is the base case, so a disk is moved, the return is made, and the method continues execution.

count = 3	count = 1
source = A	source = A
dest = B	dest = B
spare = C	spare = C



At point Z, recursive call 7 is made, and the new invocation of the method begins execution:

count = 3	count = 2
source = A	source = C
dest = B	dest = B
spare = C	spare = A

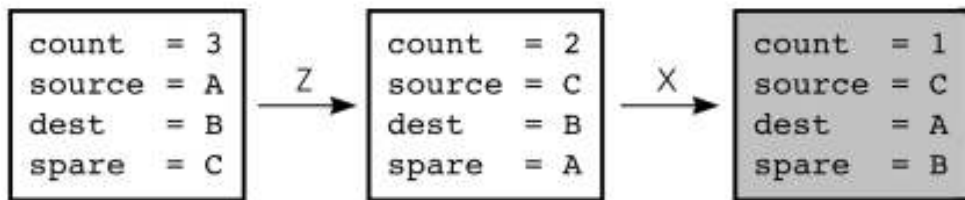
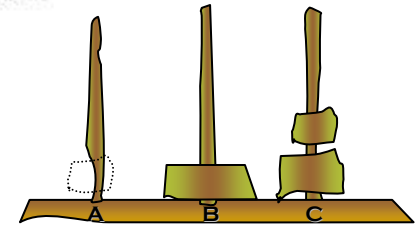
Z



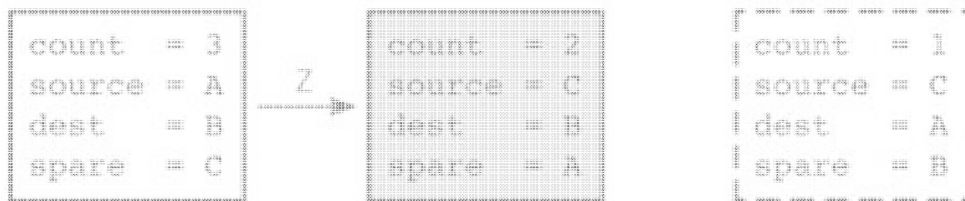
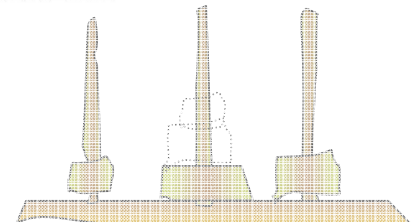
Figure 2.21d

Box trace of *solveTowers*(3, 'A', 'B', 'C')

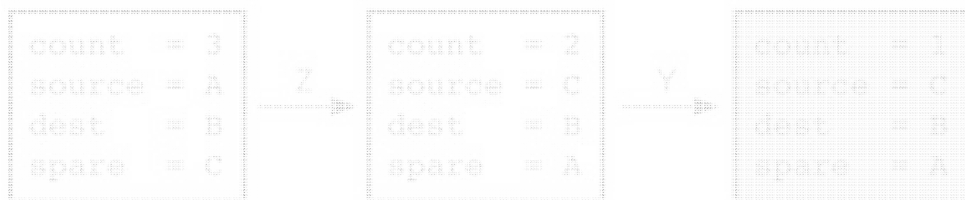
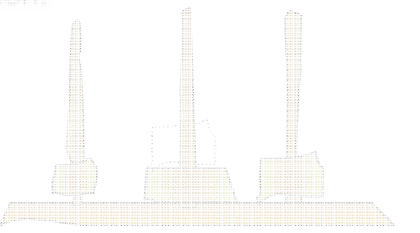
At point X, recursive call 8 is made, and the new invocation of the method begins execution:



This is the base case, so a disk is moved, the return is made, and the method continues execution



At point Y, recursive call 9 is made, and the new invocation of the method begins execution



This is the base case, so a disk is moved, the return is made, and the method continues execution

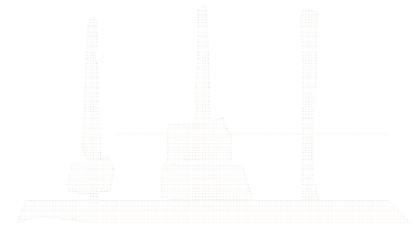
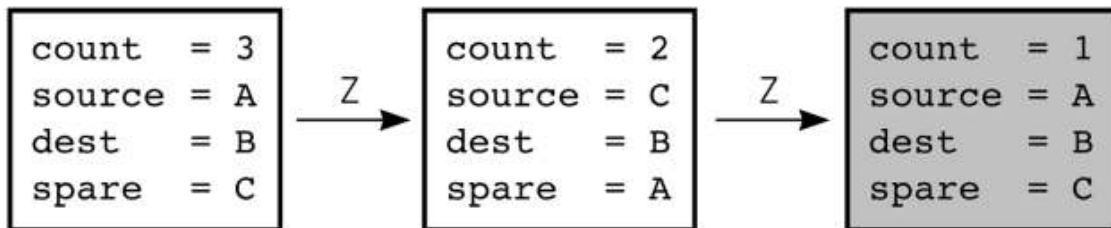
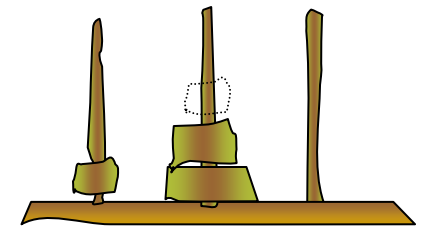


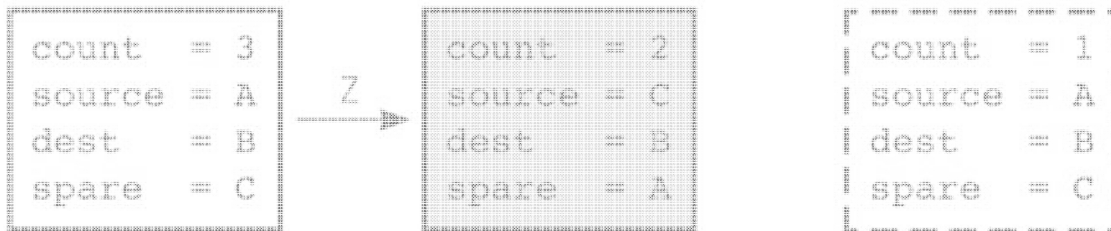
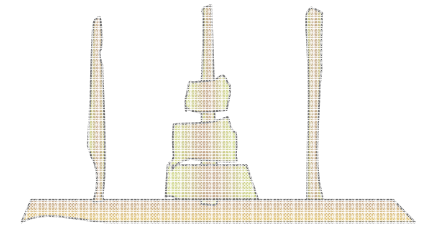
Figure 2.21e

Box trace of *solveTowers*(3, 'A', 'B', 'C')

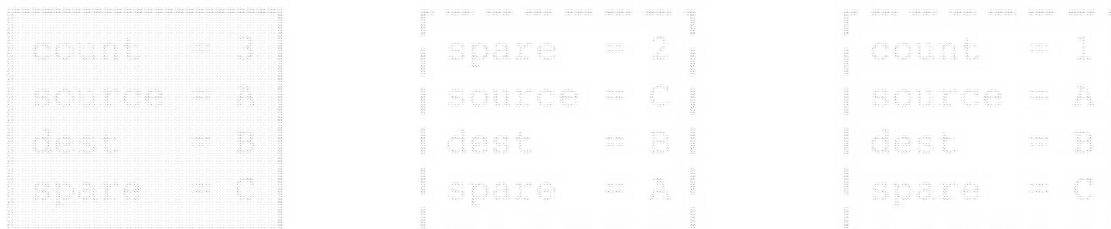
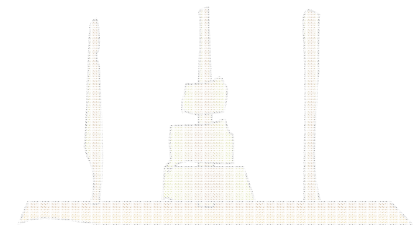
At point Z, recursive call 10 is made, and the new invocation of the method begins execution:



This is the base case, so a disk is moved, the return is made, and the method continues execution.



This invocation completes, the return is made, and the method continues execution.



Cost of Hanoi Towers

- How many moves is necessary to solve Hanoi Towers problem for N disks?
 - $\text{moves}(1) = 1$
 - $\text{moves}(N) = \text{moves}(N-1) + \text{moves}(1) + \text{moves}(N-1)$
 - i.e.
 $\text{moves}(N) = 2 * \text{moves}(N-1) + 1$
 - Guess solution and show it's correct with Mathematical Induction!
-

Recursive Searching

- Linear Search
 - Binary Search
 - Find an element in an array, return its position (index) if found, or -1 if not found.
-

Linear Search Algorithm (Java)

```
public int linSearch(int[] arr,  
                    int target)  
{  
    for (int i=0; i<arr.size; i++) {  
        if (target == arr[i]) {  
            return i;  
        }  
    } //for  
    return -1; //target not found  
}
```

Linear Search

- Iterate through an array of n items searching for the target item
 - The crucial instruction is equality checking (or “comparisons” for short)
 - `x.equals(arr[i]);` //for objects or
 - `x == arr[i];` //for a primitive type
 - Linear search performs at most n comparisons
 - We can write linear search recursively
-

Recursive Linear Search Algorithm

- Base case
 - Found the target or
 - Reached the end of the array
- Recursive case
 - Call linear search on array from the next item to the end

```
public int recLinSearch(int[] arr,int low,int x) {  
    if (low >= arr.length) { // reach the end  
        return -1;  
    } else if (x == arr[low]){  
        return low;  
    } else  
        return recLinSearch(arr, low + 1, x);  
}
```

Binary Search Sketch

- Linear search runs in $O(n)$ (linear) time (it requires n comparisons in the worst case)
 - *If the array to be searched is sorted (from lowest to highest), we can do better:*
 - Check the midpoint of the array to see if it is the item we are searching for
 - Presumably there is only a $1/n$ chance that it is!
(assuming that the target is in the array)
 - If the value of the item at the midpoint is less than the target then the target must be in the upper half of the array
 - So perform binary search on that half
 - and so on
-

Thinking About Binary Search

- Each sub-problem searches an array slice (or subarray)
 - So differs only in the upper and lower array indices that define the array slice
 - Each sub-problem is smaller than the previous problem
 - In the case of binary search, half the size
 - The final problem is so small that it is trivial
 - Binary search terminates after the problem space consists of one item or
 - When the target item is found
 - Be careful when writing the terminating condition
 - When exactly do we want to stop?
 - When the search space consists of one element but
 - Only **after** that one element has been tested
-

Recursive Binary Search Algorithm

```
public int binSearch(  
    int[] arr, int lower, int upper, int x)  
{  
    int mid = (lower + upper) / 2;  
    if (lower > upper) { // empty interval  
        return - 1; // base case  
    } else if (arr[mid] == x) {  
        return mid; // second base case  
    } else if (arr[mid] < x) {  
        return binSearch(arr, mid + 1, upper, x);  
    } else { // arr[mid] > target  
        return binSearch(arr, lower, mid - 1, x);  
    }  
}
```

Analyzing Binary Search

- **Best case:** 1 comparison
 - **Worst case:** target is not in the array, or is the last item to be compared
 - Each recursive call halves the input size
 - Assume that $n = 2^k$ (e.g. if $n = 128$, $k = 7$)
 - After the **first** iteration there are $n/2$ candidates
 - After the **second** iteration there are $n/4$ (or $n/2^2$) candidates
 - After the **third** iteration there are $n/8$ (or $n/2^3$) candidates
 - After the **k-th** iteration there is one candidate because $n/2^k = 1$
 - Because $n = 2^k$, $k = \log_2 n$
 - Thus, at most $k = \log_2 n$ recursive calls are made in the worst case!
-

Binary Search vs Linear Search

	<u>Linear</u>	<u>Binary</u>
<u>N</u>	<u>N</u>	<u>$\log_2(N)$</u>
10	10	4
100	100	7
1,000	1000	10
10,000	10,000	14
100,000	100,000	17
1,000,000	1,000,000	20
10,000,000	10,000,000	24

Iterative Binary Search

- Use a **while** loop instead of recursive calls
 - The initial values of lower and upper do not need to be passed to the method but
 - Can be initialized before entering the loop with lower set to 0 and upper to the length of the array-1
 - Change the lower and upper indices in each iteration
 - Use the (negation of the) base case condition as the condition for the loop in the iterative version.
 - Return a negative result if the while loop terminates without finding the target
-

Binary Search Algorithm (Java)

```
public int binSearch(int[] arr, int target){
    int lower = 0;
    int upper = arr.length - 1;

    while (lower <= upper){
        int mid = (lower + upper) / 2;
        if (target == arr[mid]) {
            return mid;
        } else if (target > arr[mid]) {
            lower = mid + 1;
        } else { //target < arr[mid]
            upper = mid - 1;
        }
    } //while
    return -1; //target not found
}
```

Index of the first and last elements in the array

Recursion Disadvantage 1

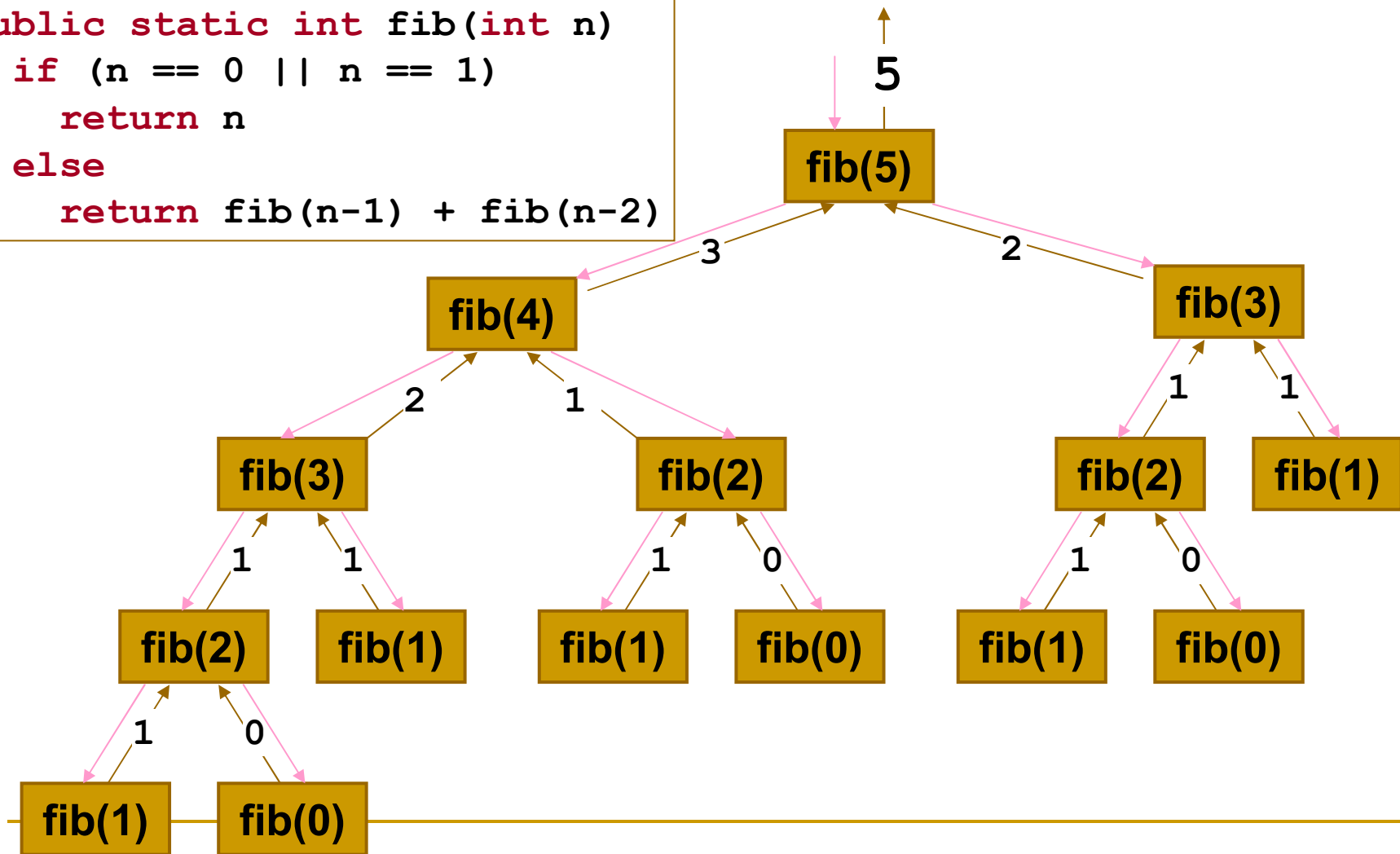
- Recursive algorithms have more overhead than similar iterative algorithms
 - Because of the repeated method calls (storing and removing data from call stack)
 - This may also cause a “stack overflow” when the call stack gets full
 - It is often useful to derive a solution using recursion and implement it iteratively
 - Sometimes this can be quite challenging!
(Especially, when computation continues after the recursive call -> we often need to remember value of some local variable -> stacks can be often used to store that information.)
-

Recursion Disadvantage 2

- Some recursive algorithms are inherently inefficient
 - An example of this is the recursive Fibonacci algorithm which repeats the same calculation again and again
 - Look at the number of times fib(2) is called
 - Even if the solution was determined using recursion such algorithms should be implemented iteratively
 - To make recursive algorithm efficient:
 - Generic method (used in AI): store all results in some data structure, and before making the recursive call, check whether the problem has been solved.
 - Make iterative version.
-

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)
```





Example

- Example of recursive algorithm which is more efficient than straightforward non-recursive version (pow).



Analyzing Recursive Functions

- Recursive functions can be tricky to analyze
 - It is useful to trace through the sequence of recursive calls
 - This can be done using a recursion tree
 - As shown for the Fibonacci function
 - Recursion trees can also be used to determine the running time (in number of operations) of algorithms
 - Annotate the tree to indicate how much work is performed at each level of the tree
 - Determine how many levels of the tree there are
-

Recursion and Induction

- Recursion is similar to mathematical induction as recursion solves a problem by
 - Specifying a solution for the base case and
 - Using the recursive case to derive solutions of any size from the solutions to smaller problems
 - Induction proves a property by
 - Proving it is true for a base case (which is often true by definition) and
 - Proving that it is true for some number, n , if it is true for all numbers less than n
-

Recursive Factorial Algorithm

```
public int fact (int x){  
    // Should check for negative values of x  
    if (x == 0){  
        return 1;  
    } else  
        return x * fact(x - 1);  
}
```

- Prove, using induction, that the algorithm returns the values:
 - $\text{fact}(0) = 0! = 1$
 - $\text{fact}(n) = n! = n * (n - 1) * (n - 2) * \dots * 1$ if $n > 0$

Proof by Induction of **fact** Method

- **Basis:** Show that the property is true for $n = 0$, i.e. that `fact(0)` returns 1
 - This is true by definition as `fact(0)` is the base case of the algorithm and returns 1
 - Now establish that the property is true for an arbitrary k implies that it is true for $k + 1$
 - **Inductive hypothesis:** Assume that the property is true for $n = k$, that is assume that
 - $\text{fact}(k) = k * (k - 1) * (k - 2) * \dots * 2 * 1$
-

Proof by Induction of **fact** Method

- **Inductive conclusion:** Show that the property is true for $n = k + 1$, i.e., that `fact(k + 1)` returns
 - $(k + 1) * k * (k - 1) * (k - 2) * \dots * 2 * 1$
 - By definition of the function `fact(k + 1)` returns
 - $(k + 1) * \text{fact}(k)$ – the recursive case
 - And by the inductive hypothesis `fact(k)` returns
 - $k * (k - 1) * (k - 2) * \dots * 2 * 1$
 - Therefore `fact(k + 1)` must return
 - $(k + 1) * k * (k - 1) * (k - 2) * \dots * 2 * 1$
 - Which completes the inductive proof
-

More Recursive Algorithms

- Recursive counting (Chapter 3)
 - *Backtracking* - Eight Queens problem (see Chapter 6 in the textbook)
 - Sorting (later in the course)
 - Mergesort
 - Quicksort
-

Recursive Data Structures

- Linked Lists are recursive data structures
 - They are defined in terms of themselves
 - There are recursive solutions to many list methods
 - List traversal can be performed recursively
 - Recursion allow for easy and elegant solutions of problems that would be difficult to implement iteratively, such as printing a list backwards
 - See the recursive version of the LinkedList code developed in class
-