LESSON 4
RECURSION

Wholeness of the Lesson

Computation of a function by recursion involves repeated self-calls of the function. Recursion is implicit also at the design level when a reflexive association is present. Recursion mirrors the self-referral dynamics of consciousness, on the basis of which all creation emerges.

Recursion

- Recursion is a problem-solving approach that can be used to generate simple solutions to certain kinds of problems that are difficult to solve by other means.
- In the field of artificial intelligence, recursion often is used to write programs that exhibit intelligent behavior.
- Recursion reduces a problem into one or more simpler versions of itself.
- A Java method is recursive, or exhibits recursion, if in its body it calls itself.

Steps to Design a Recursive Algorithm

- There must be at least one case (the base case), for a small value of *n*, that can be solved directly
- A problem of a given size *n* can be reduced to one or more smaller versions of the same problem (recursive case(s))
- Identify the base case(s) and solve it directly
- Devise a strategy to reduce the problem to smaller versions of itself while making progress toward the base case
- Combine the solutions to the smaller problems to solve the larger problem

Simple Countdown using recursion

```
public class RecursiveCountdown{
public static void main(String[] args){
countDown(3);
}
public static void countDown(int num){
if (num <=0){
System.out.println();
}
else{
System.out.print(num);
countDown(num - 1);
}
}
}
```

General Examples

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

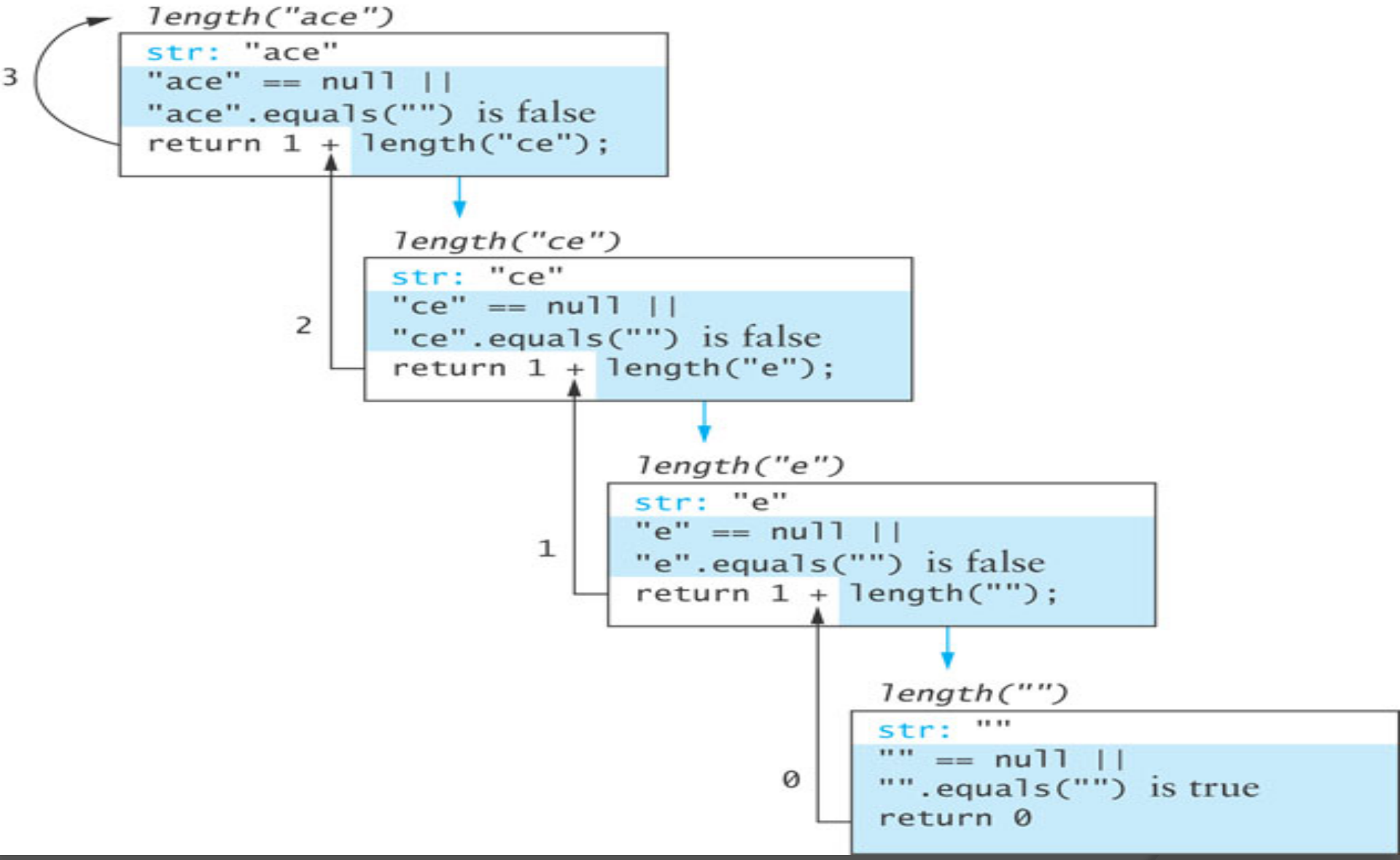
Recursive Algorithm for Finding the Length of a String (cont.)

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

Run-Time Stack and Activation Frames

- Java maintains a run-time stack on which it saves new information in the form of an *activation frame*
- The activation frame contains storage for
  - method arguments
  - local variables (if any)
  - the return address of the instruction that called the method
- Whenever a new method is called (recursive or not), Java pushes a new activation frame onto the run-time stack and return will remove the activation frame from the stack and return to the previous call.

Stack Trace


length("ace")
str: "ace"
"ace" == null ||
"ace".equals("") is false
return 1 + length("ce");
3
length("ce")
str: "ce"
"ce" == null ||
"ce".equals("") is false
return 1 + length("e");
2
length("e")
str: "e"
"e" == null ||
"e".equals("") is false
return 1 + length("");
1
length("")
str: ""
"" == null ||
"".equals("") is true
return 0
0

Find the sum of 1 to n numbers

Method : Sum(n) = 1 + 2 + 3 + …+n

Sum(1) = 1

Sum(2) = 1 + 2 => Sum(2) = Sum(1) + 2

Sum(2) = 1 + 2

Sum(3) = 1 + 2 + 3 => Sum(3) = Sum(2) + 3

Sum(n-1) = 1 + 2 + 3 + … + (n-2) + (n-1)

Sum(n) = 1 + 2 + 3 + … + (n-2) + (n-1) + n

Sum(n) = Sum(n-1) + n

Finding the sum of n integer

Sum(n) = Sum(n-1) + n

Recursive relationship.

This relationship establishes the general case or recursive case.

Solution is expressed in terms of solutions to smaller versions of the same problem

All other cases are base cases.

Solution is obtained directly.

Recursive case :

Sum(n) = Sum(n-1) + n (if n > 1)

Base case

Sum(1) = 1

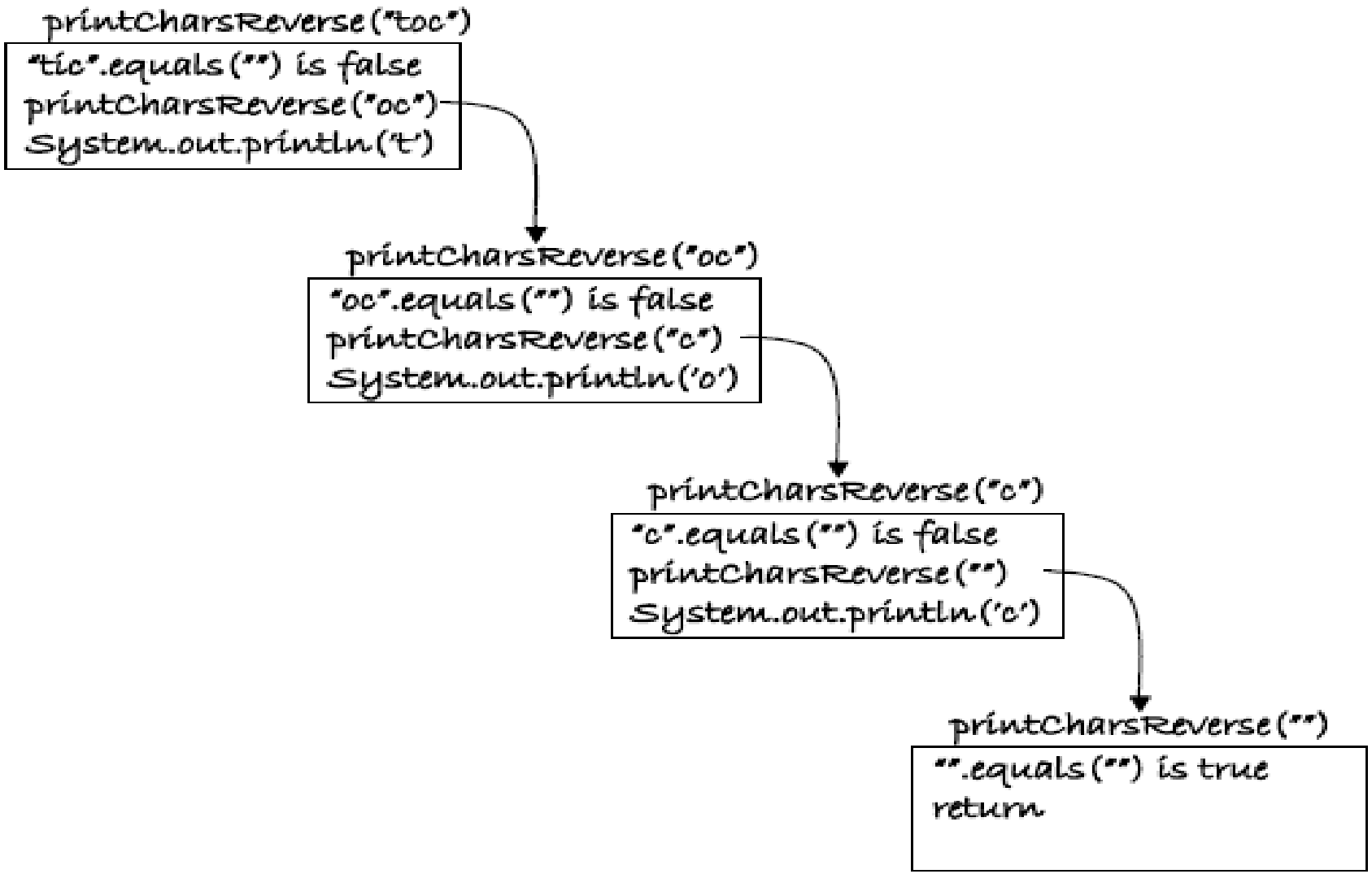
```
public static int Sum(int n)
{
   if (n == 1)                  //base case
     return 1;
   else
     return Sum(n-1) + n; //general case
}
```

Write a Recursive Algorithm for Printing String Characters in Reverse

```
/** Recursive method printCharsReverse
    Problem : The argument string is displayed in
              reverse, one character per line
    @param str :The input string
*/
public static void printCharsReverse(String str) {
    if (str == null || str.equals(""))
        return;
    else {
        printCharsReverse(str.substring(1));
        System.out.println(str.charAt(0));
    }
}
```

Draw the Stack diagram

Trace the execution of `printCharsReverse("toc")` using activation frames.



Main Point

When a recursion involves many redundant computations, one tries to write an iterative version of the method (using loops). Likewise, though all healing can in principle be done on the level of consciousness, when consciousness is not yet sufficiently established in its home, many steps of healing may be required to obtain the desired result.

Mathematical Examples

- Factorial
- Fibonacci
- Power Function
- Binomial Coefficient

Factorial Function

The factorial function on input n computes the product of the positive integers less than or equal to n.

For example, 5! = 5*4*3*2*1 = 120

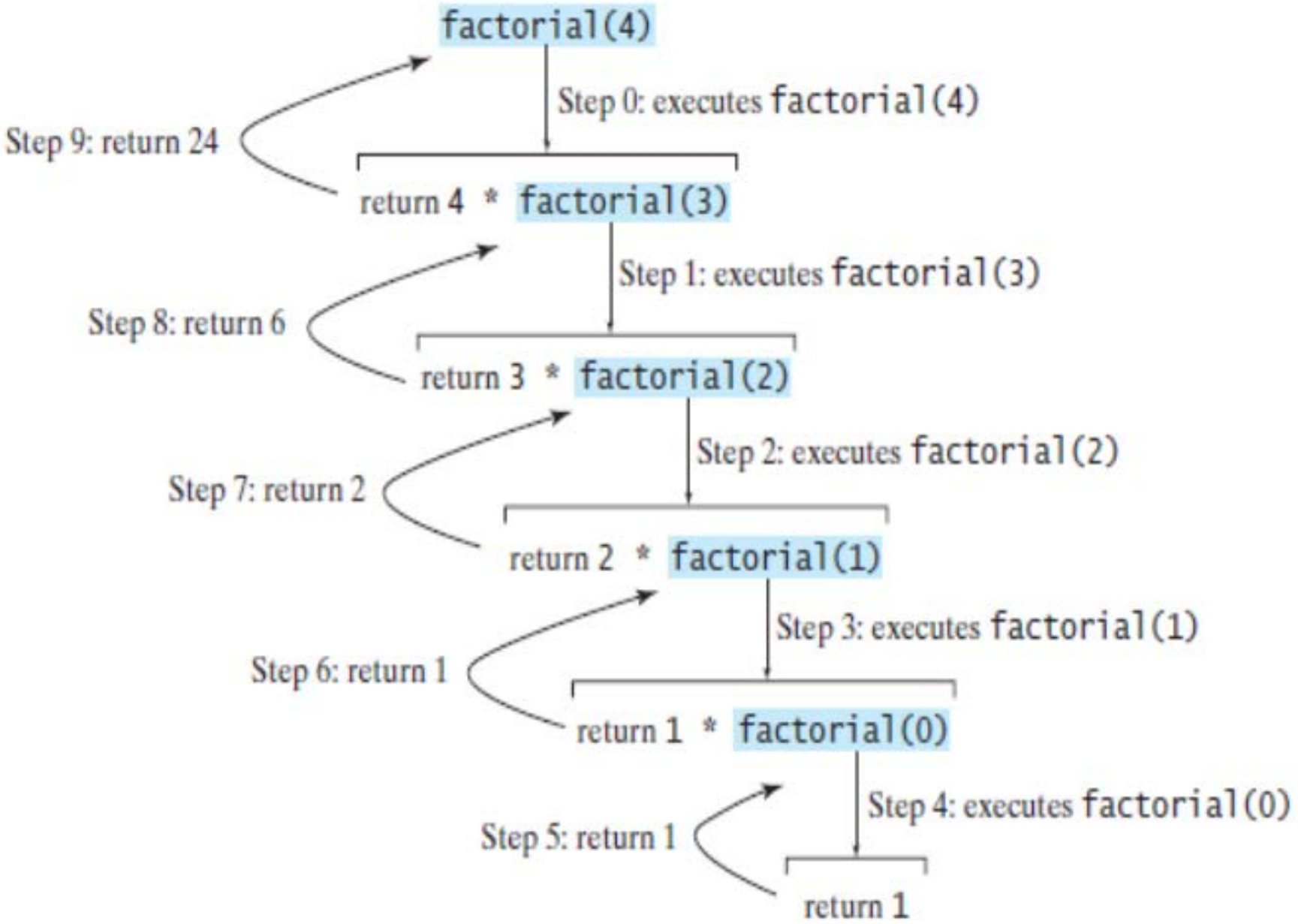
```
public static int fact(int num) {
        if(num == 0 || num == 1) {     //base case
                return 1;
        }
        return num * fact(num-1);
}
```

Factorial Class

```
public class factorial{
        public static void main(String[] args) {
                System.out.println("Factorial of 6 = " + fact(6));
                System.out.println("Factorial of 10 = " +
fact(10));
        }

        public static int fact(int num) {
                if(num == 0 || num == 1)
                        return 1;
                else
                        return num * fact(num - 1);
        }
}
```

Stack Trace of Factorial



What is the Output ?

```
public class test {
public static void main(String[] args) {
xMethod(1234567);
}
public static void xMethod(int n) {
if (n > 0) {
System.out.print(n % 10);
xMethod(n / 10);
} }
}
// Output : 7654321
```

Fibonacci Numbers

$F_0 = 0, F_1 = 1, F_2 = 1, F_3 = 2, F_4 = 3, F_5 = 5, \ldots,$
OR

$F_0 = 0, F_1 = 1, F_n = F_{n-1} + F_{n-2}$ for $n > 1$

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

Binomial Co-efficient-C(n, k)

C(n, 0) = C(n, n) = 1
C(n, k) = C(n-1, k) + C(n-1, k - 1)

```
int comb(int n, int k) {
        if(k == 0 || k == n) {
                return 1;
        }
        return comb(n-1, k) + comb(n-1, k-1);
}
```

Data Structure Problems

- Towers of Hanoi
- Linear Search
- Binary Search
- Merge sort
- Binary Search Trees

Linear Search

- Searching an array can be accomplished using recursion
- Base cases for recursive search:
  - Empty array, target can not be found;
    result is -1
  - First element of the array being searched = target;
    result is the subscript of first element
- The recursive step searches the rest of the array, excluding the first element

Linear Search

```
private static int linearSearch(Object[] items, Object target, int posFirst)
{
        if (posFirst == items.length) {
            return -1;
        } else if (target.equals(items[posFirst])) {
            return posFirst;
        } else {
            return linearSearch(items, target, posFirst + 1);
        }
    }
```

Towers of Hanoi

- Problem that consists of a number of disks placed on three columns.
- The disks have different diameters and holes in the middle so they will fit over the columns.
- All the disks start out on the first column.
- The object of the problem is to transfer all the disks from first column to last.
- Only one disk can be moved at a time and no disk can be placed on a disk that’s smaller than itself.

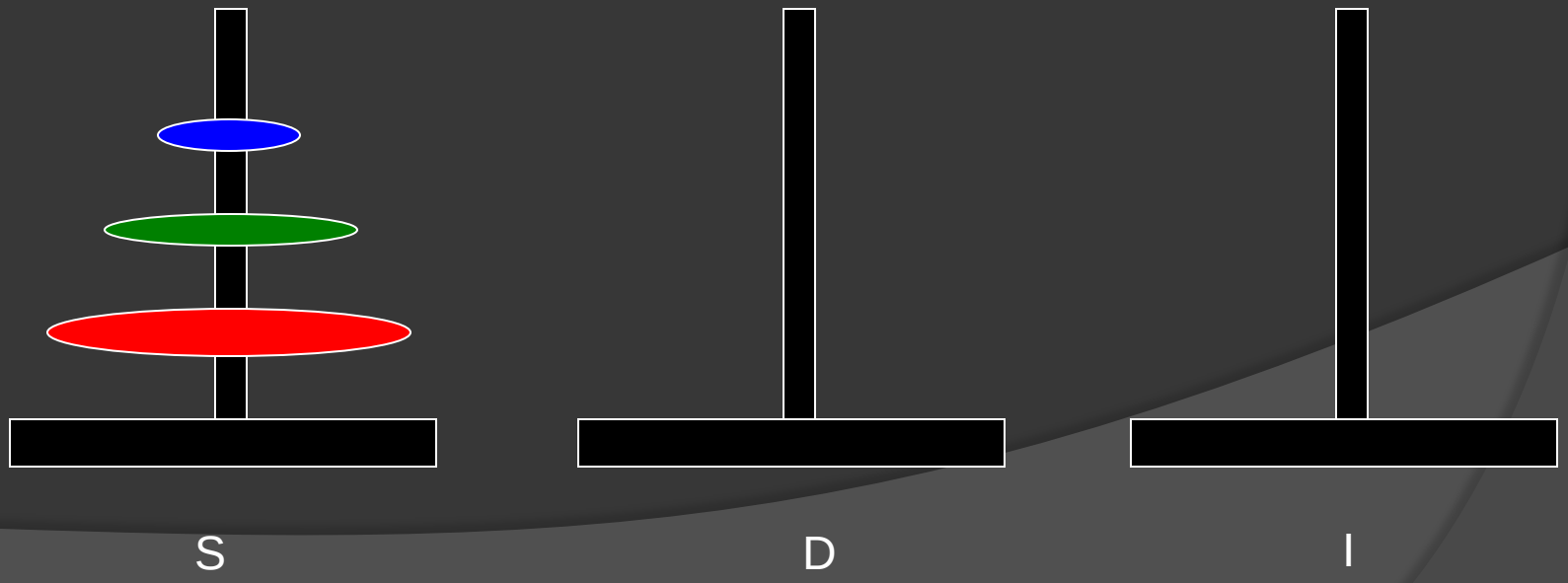
Solution

- Assume that you want to move all the disks from a source tower S to a destination tower D.
- Assume an intermediate tower, I.
- Assume n disks on S.

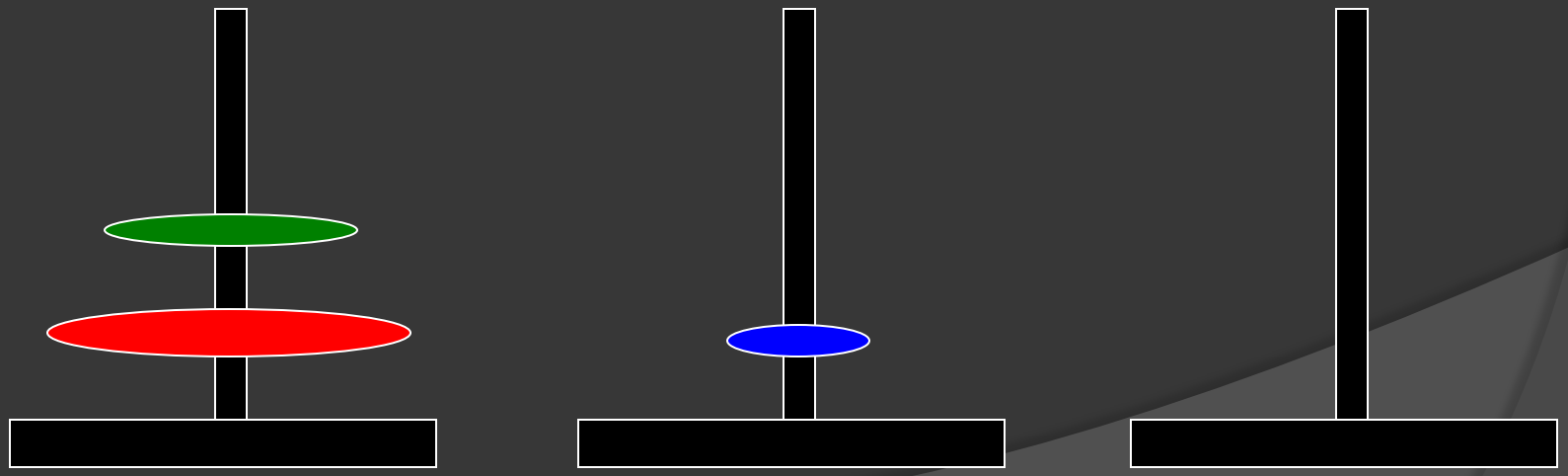
Procedure:

1. Move the subtree consisting of the top n-1 disks from S to I.
2. Move the remaining (largest) disk from S to D.
3. Move the subtree from I to D.

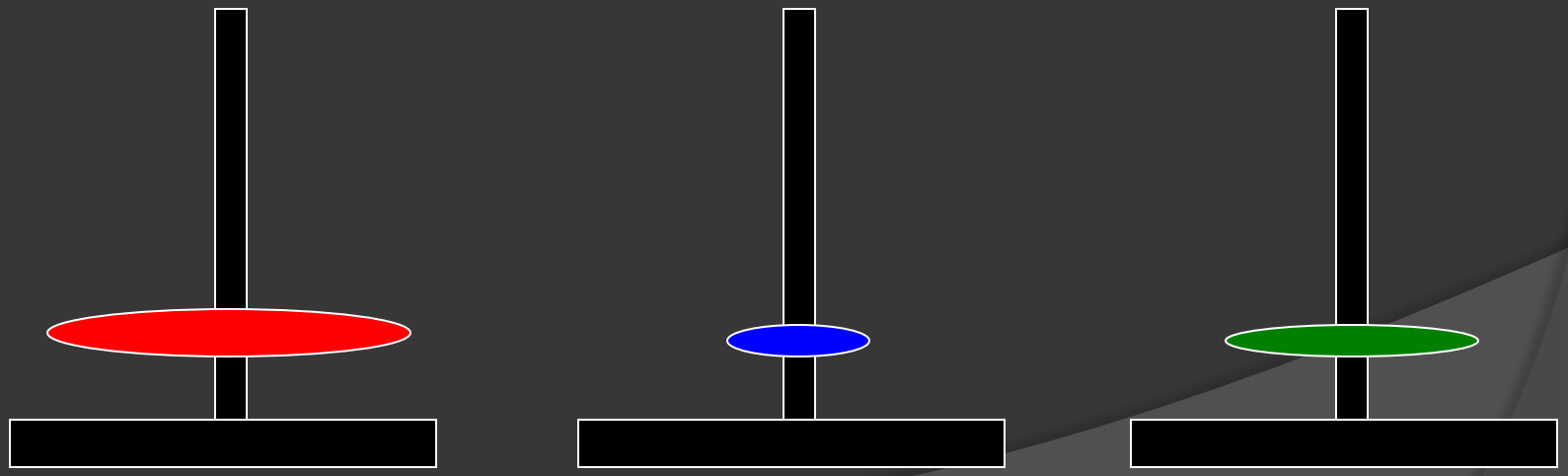
Towers of hanoi



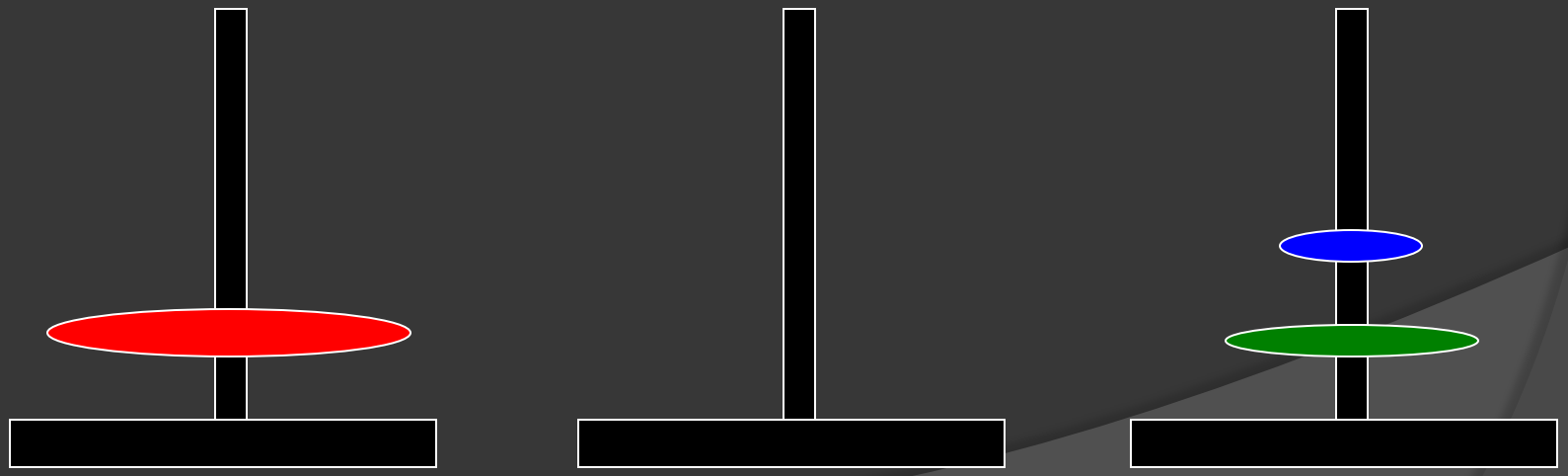
Tower of Hanoi



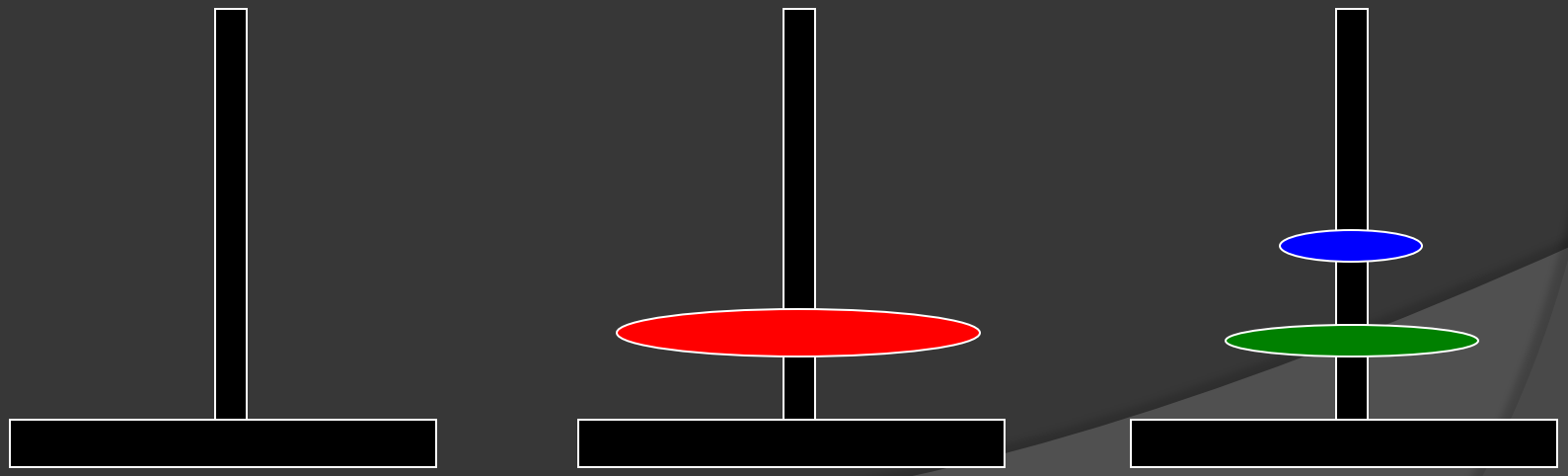
Tower of Hanoi



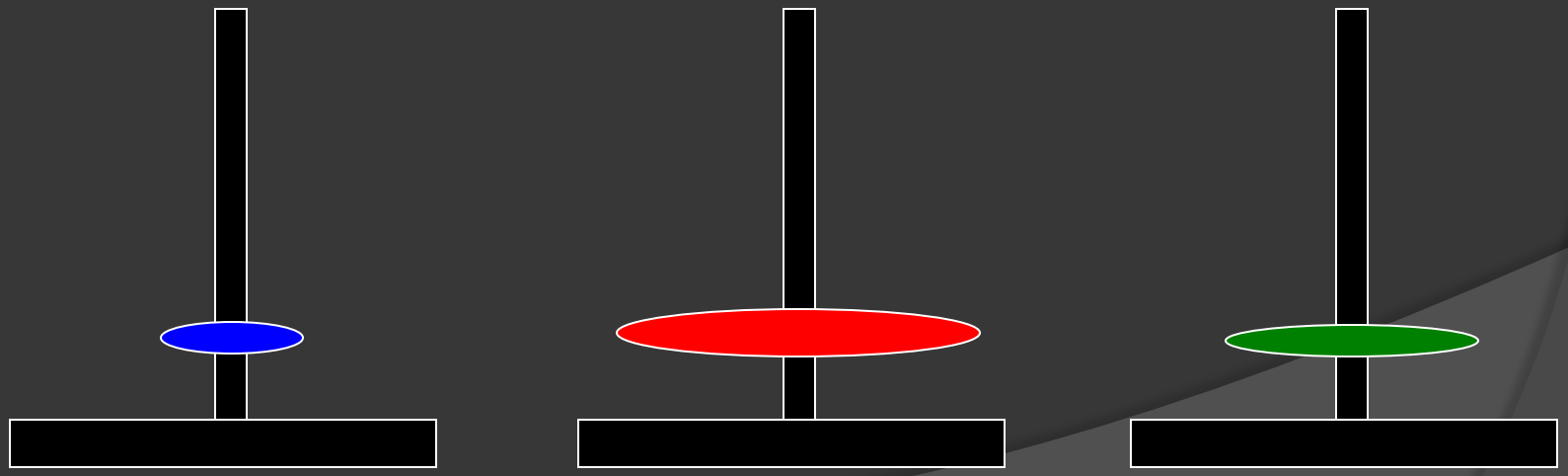
Tower of Hanoi



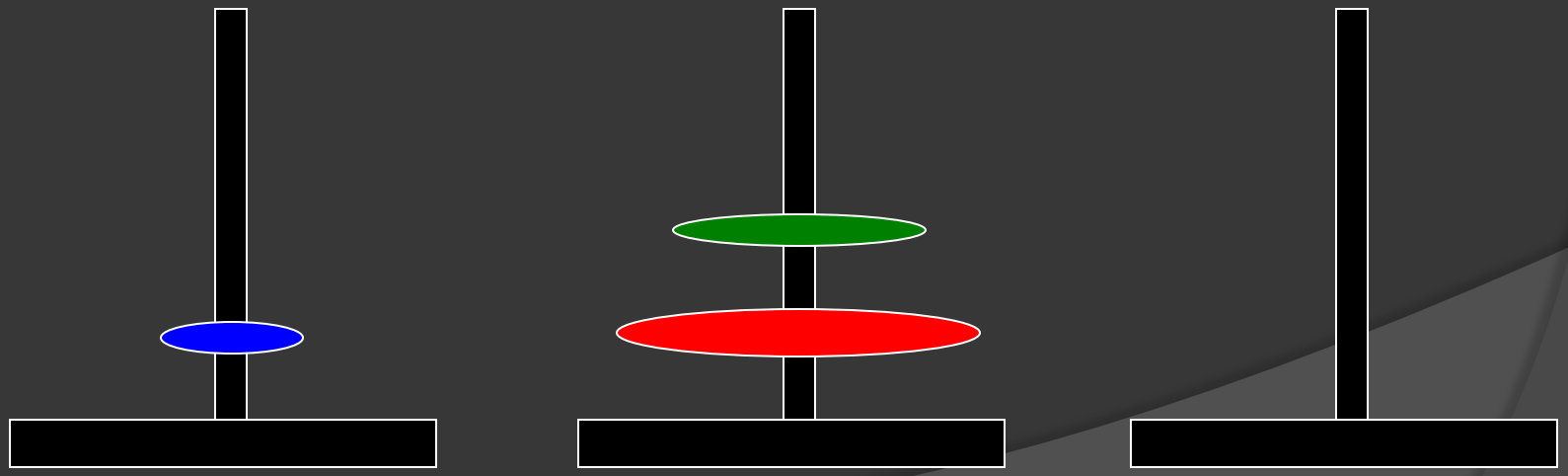
Tower of Hanoi



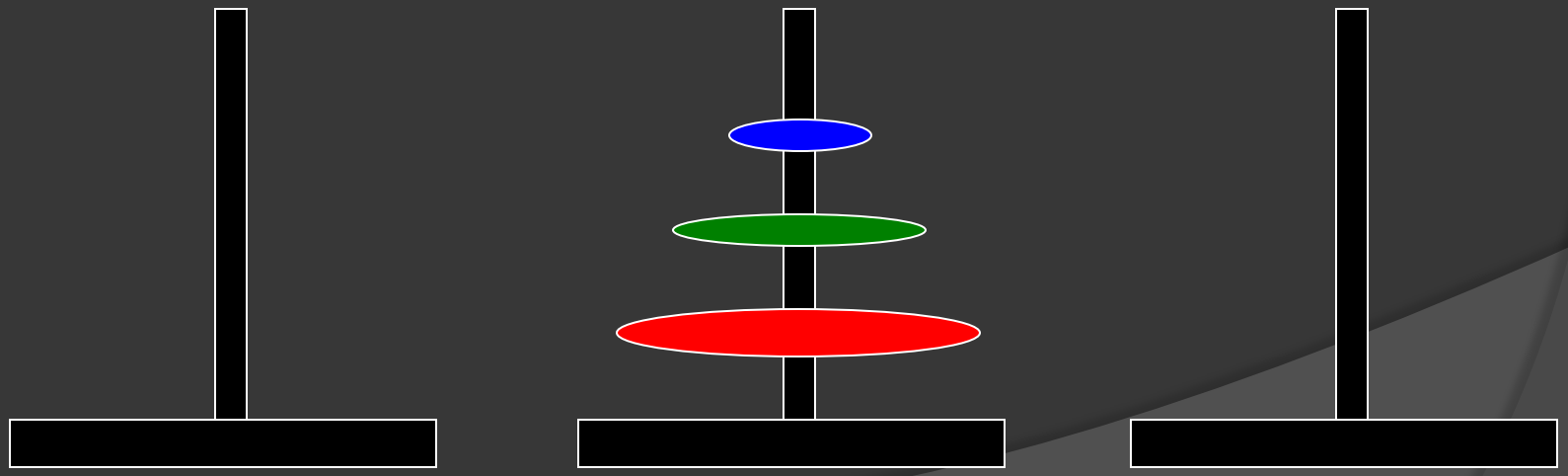
Tower of Hanoi



Tower of Hanoi



Tower of Hanoi



Tower of Hanoi

```
public static void moveDisks(int n, char fromTower,char toTower, char auxTower)
{
 if (n == 1) // Stopping condition
 System.out.println("Move disk " + n + " from " +
 fromTower + " to " + toTower);
 else { // Recursive Part
moveDisks(n - 1, fromTower, auxTower, toTower);
System.out.println("Move disk " + n + " from " +
 fromTower + " to " + toTower);
moveDisks(n - 1, auxTower, toTower, fromTower);
}
}
```

Recursion Versus Iteration

- There are similarities between recursion and iteration
- In iteration, a loop repetition condition determines whether to repeat the loop body or exit from the loop
- In recursion, the condition usually tests for a base case
- You can always write an iterative solution to a problem that is solvable by recursion
- A recursive algorithm may be simpler than an iterative algorithm and thus easier to write, code, debug, and read

PRACTICE

Palindrome

A string is a palindrome if it reads the same from left to right or right to left.

Create a recursive definition

Identify base case(s)

Write a recursive program and test it

Power Function

```
pow(x, 0) = 1  if y = 0
pow(x, y) = x * pow(x, y-1),
  y > 0
```

CONNECTING THE PARTS OF KNOWLEDGE WITH THE WHOLENESS OF KNOWLEDGE

Recursion creates from self-referral activity

- In Java, it is possible for a method to call itself.
- For a self-calling method to be a legitimate recursion, it must have a base case, and whenever the method is called, the sequence of self-calls must converge to the base case.

Transcendental Consciousness: TC is the self-referral field of existence, at the basis of all manifest existence.

Wholeness moving within Itself: In Unity Consciousness, one sees that all activity in the universe springs from the self-referral dynamics of wholeness. The "base case" – the reference point – is always the Self, realized as Brahman.