# Recursion

# Learning Objectives

- Recursive void Functions
  - Tracing recursive calls
  - Infinite recursion, overflows
- Recursive Functions that Return a Value
  - Powers function
- Thinking Recursively
  - Recursive design techniques
  - Binary search

# Introduction to Recursion

- A function that "calls itself"
  - Said to be *recursive*
  - In function definition, call to same function
- Java allows recursion
  - As do most high-level languages
  - Can be useful programming technique
  - Has limitations

# Recursive void Functions

- Divide and Conquer
  - Basic design technique
  - Break large task into subtasks
- Subtasks could be smaller versions of the original task!
  - When they are → recursion may be used in solution

# Recursive void Function Example

- Consider task:
- Search list for a value
  - Subtask 1: search 1st half of list
  - Subtask 2: search 2nd half of list
- Subtasks are smaller versions of original task!
- When this occurs, recursive function can be used.
  - Usually results in "elegant" solution

# Recursive void Function: Vertical Numbers

- Task: display digits of number vertically, one per line
- Example call:
  writeVertical(1234);
  Produces output:
  1
  2
  3
  4

# Vertical Numbers: Recursive Definition

- Break problem into two cases
- Simple/base case: if n<10
  - Simply write number n to screen
- Recursive case: if n>=10, two subtasks:

  1- Output all digits except last digit

  2- Output last digit
- Example: argument 1234:
  - $1^{st}$ subtask displays 1, 2, 3 vertically
  - $2^{nd}$ subtask displays 4

# writeVertical Function Definition

- Given previous cases:

```
public static void writeVertical(int n)
 {
    if(n<10) System.out.println(n); // base case
    else // recursive step
    {
        writeVertical(n/10);
        System.out.println(n%10);
    }
 }
```

# writeVertical Trace

If (123<10) { …} writeVertical(123)

else {

If (12<10) {. . . } writeVertical(123/10)

else {

If (1<10) writeVertical(12/10)

{

System.out.println(1); output 1

} else {

writeVertical(1/10);

System.out.println(1%10);}

System.out.println(12%10) ; } output 2

System.out.println(123%10); } output 3

# Recursion—A Closer Look

- Computer tracks recursive calls
  - Stops current function
  - Must know results of new recursive call before proceeding
  - Saves all information needed for current call
    - To be used later
  - Proceeds with evaluation of new recursive call
  - When THAT call is complete, returns to "outer" computation

# Recursion Big Picture

- Outline of successful recursive function:
  - One or more cases where function accomplishes it's task by:
    - Making one or more recursive calls to solve smaller versions of original task
    - Called "recursive case(s)"
  - One or more cases where function accomplishes it's task without recursive calls
    - Called "base case(s)" or stopping case(s)

# Infinite Recursion

- Base case MUST eventually be entered
- If it doesn't → infinite recursion
  - Recursive calls never end!
- Recall writeVertical example:
  - Base case happened when down to 1-digit number
  - That's when recursion stopped

# Infinite Recursion Example

- Consider alternate function definition:

```
void newWriteVertical(int n)
{
        newWriteVertical(n/10);
        System.out.println(n%10);
}
```

- Seems "reasonable" enough
- Missing "base case"!
- Recursion never stops

# Stacks for Recursion

- A stack
  - Specialized memory structure
  - Like stack of paper
    - Place new on top
    - Remove when needed from top
  - Called "last-in/first-out" memory structure
- Recursion uses stacks
  - Each recursive call placed on stack
  - When one completes, last call is removed from stack

# Stack Overflow

- Size of stack limited
  - Memory is finite
- Long chain of recursive calls continually adds to stack
  - All are added before base case causes removals
- If stack attempts to grow beyond limit:
  - Stack overflow error
- Infinite recursion always causes this

# Recursion Versus Iteration

- Recursion not always "necessary"
- Not even allowed in some languages
- Any task accomplished with recursion can also be done without it
  - Nonrecursive: called iterative, using loops
  - Additional memory for storage of intermediate results may be needed
- Recursive:
  - Runs slower, uses more storage
  - Elegant solution; less coding

# Recursive Functions that Return a Value

- Recursion not limited to void functions
- Can return value of any type
- Same technique, outline:
  1. One or more cases where value returned is computed by recursive calls
     - Should be "smaller" sub-problems
  2. One or more cases where value returned computed without recursive calls
     - Base case

# Return a Value Recursion Example: Powers

- Recall predefined function pow():
  result = pow(2, 3);
  - Returns 2 raised to power 3 (8)
  - Takes two int arguments
  - Returns int value
- Let's write recursively
  - For simple example

# Function Definition for power()

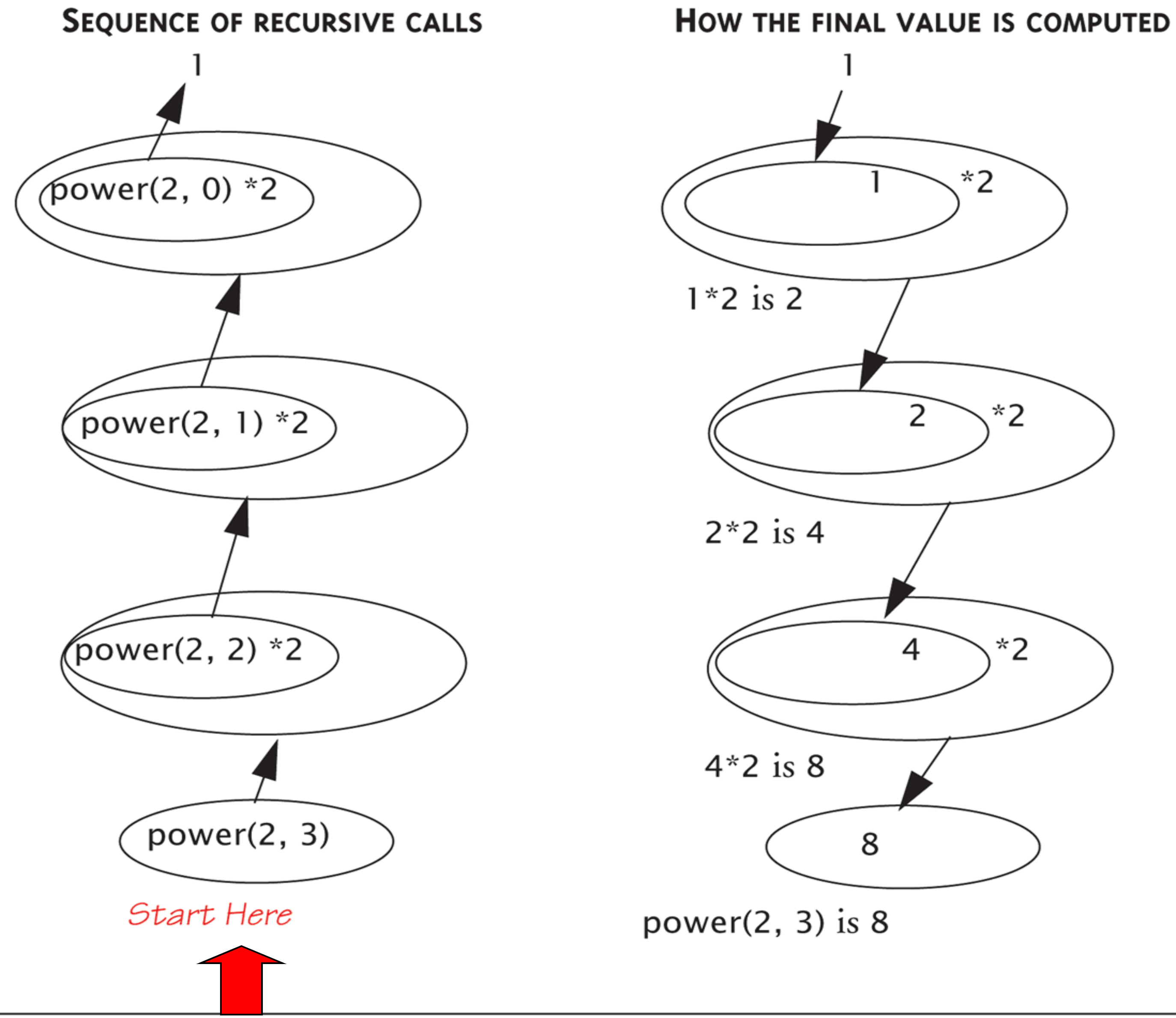
```
public static int power(int x, int n){
        if (n<0)
        {  System.out.println("Illigal argument");
            System.exit(0);
        }
        if(n>0)
        {       return x*power(x,(n-1));   }
        else return 1;
}
```

# Calling Function power()

- Example calls:
- power(2, 0);
  → returns 1
- power(2, 1);
  → returns (power(2, 0) * 2);
  → returns 1
  - Value 1 multiplied by 2 & returned to original call

# Tracing Function power():

**Display 13.4 Evaluating the Recursive Function Call** `power(2,3)`



# Thinking Recursively

- Ignore details
  - Forget how stack works
  - Forget the suspended computations
  - Yes, this is an "abstraction" principle!
  - And encapsulation principle!
- Let computer do "bookkeeping"
  - Programmer just think "big picture"

# Thinking Recursively: power

- Consider power() again
- Recursive definition of power:
  power(x, n)

  returns:

  power(x, n – 1) * x
  - Just ensure "formula" correct
  - And ensure base case will be met

# Recursive Design Techniques

- Don't trace entire recursive sequence!
- Just check 3 properties:
  1. No infinite recursion
  2. Stopping cases return correct values
  3. Recursive cases return correct values

# Recursive Algorithms

- Recursive function calls itself
- Programs evaluating mathematical expressions with recurrence – the most natural ones to implement with recursion.

# Recurrence in Math

- Factorial:
  - $5!=5*4*3*2*1$
  - $0!=1$
- The recurrence relation :
  - $N! = N * (N - 1)!$
  - For $N>=1$ with $0!=1$

$5! = 5*4! = 5*4*3! = \ldots$

# Recursive Factorial Implementation

```
public ststic int factorial (int N)
  {
    if (N == 0) return 1;
    return N * factorial(N-1);
  }
```

$N! = N * (N - 1)!$

# Non-recursive Factorial Solution

```
public static int factorial (int N)
  {
    int t=1, i;
    for (i=1; i<=N; i++)   t*=i;
     return t;
  }
```

# Recursion vs. Loops

- Always possible to transform a recursive program into non-recursive.
- Recursion gives compact, clean solutions without sacrificing efficiency.
- Function call stack overhead is minimal. Systems make sure function calls are done fast and efficiently.
- In loop implementation – need local variables.

# Basic Features of Recursion

- Recursive case – recursive call that involves *smaller values of arguments*

```
N * factorial(N-1);
```

- Stopping case – returns value (number)

```
if (N == 0) return 1;
```

```
int factorial (int N)
  {
    if (N == 0) return 1;
    return N * factorial(N-1);
  }
```

# Questionable Recursive Algorithm

```
public ststic int puzzle(int N)
  {
    if (N == 1) return 1;
    if (N % 2 == 0)   // argument is even – call itself with N/2
         return puzzle(N/2);
    else return puzzle(3*N+1); // argument odd – call itself with 3*N+1
  }
```

- Have 2 recursive cases
  - return puzzle(N/2); -- good one
  - return puzzle(3*N+1);-- bad one, recursive call with bigger argument

# Illustration

- Difficult to predict how deep recursion is going to be.
- Can not estimate efficiency. May be will never reach the end?

```
puzzle(3)
  puzzle(10)
    puzzle(5)
      puzzle(16)
        puzzle(8)
          puzzle(4)
            puzzle(2)
              puzzle(1)
```

# Euclid's Algorithm

- Find greatest common divisor (GCD) of two integers.
- GCD of x and y with $x > y$ is the same as GCD of y and (x % y )
- $x = ky + x\%y$

# Euclid's Algorithm Implementation

```
int gcd(int m, int n)
  {
    if (n == 0) return m;
    return gcd(n, m % n);
  }
```

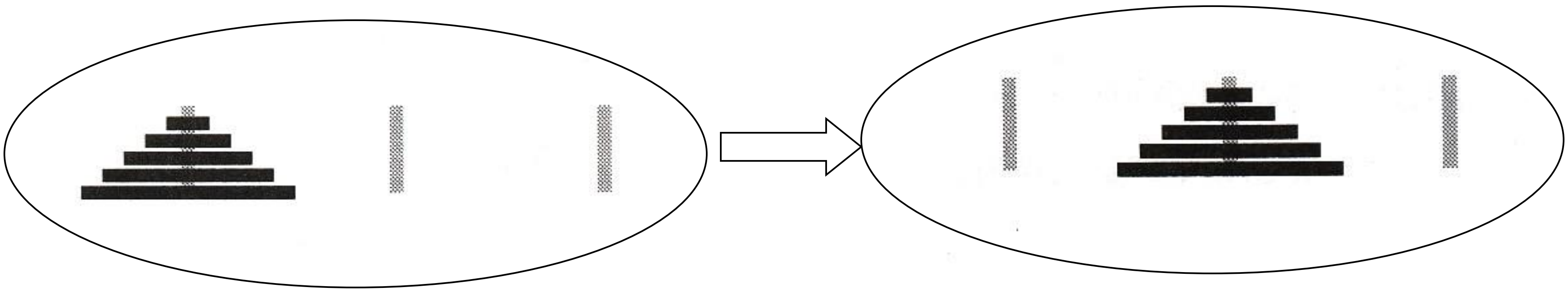
```
gcd(314159, 271828)
  gcd(271828, 42331)
    gcd(42331, 17842)
      gcd(17842, 6647)
        gcd(6647, 4458)
          gcd(4458, 2099)
            gcd(2099, 350)
              gcd(350, 349)
                gcd(349, 1)
                  gcd(1, 0)
```

# Recursion and System Stack

- Depth of recursion – how many recursion calls are waiting their completion on the stack.
- The depth of recursion depends on input
- For huge numbers – the recursion may be too deep for the system to handle. System stack overflow will result.

# Towers of Hanoi –Ancient Problem

- 3 pegs and N disks
- The disks differ in size and originally arranged by size on one of the pegs
- The final goal is to move all N disks from the first peg to the next one.



# Rules of the Game

- Only one disk may be moved at a time
- A disk can not be placed on top of a smaller disk
- All disks must be stored on a peg except while being moved.

# The End of the World Prediction

- One legend says that a certain group of monks in a temple is working on the problem of moving 40 golden disks on 3 diamond pegs.
- The legend says the world will end when the monks will be done with their work of moving the tower from one peg to the next.
- How much time do we have left?

# The Solution

- By finding of the programming solution to the Towers of Hanoi we will accomplish two tasks:
  - Find the universal solution for any number of disks N
  - Estimate the complexity of the algorithm and find how long may it take for the monks to finish their job.

# Solution Details

- Need to specify which disk is moved where on each step.
- Each disk has its number (from 1 to N)
- We will use "direction" marker
  - \+ means move one peg to the right, cycling to the leftmost peg when on the rightmost peg;
  - \- means move one peg left, cycling to the rightmost peg when on the leftmost peg.

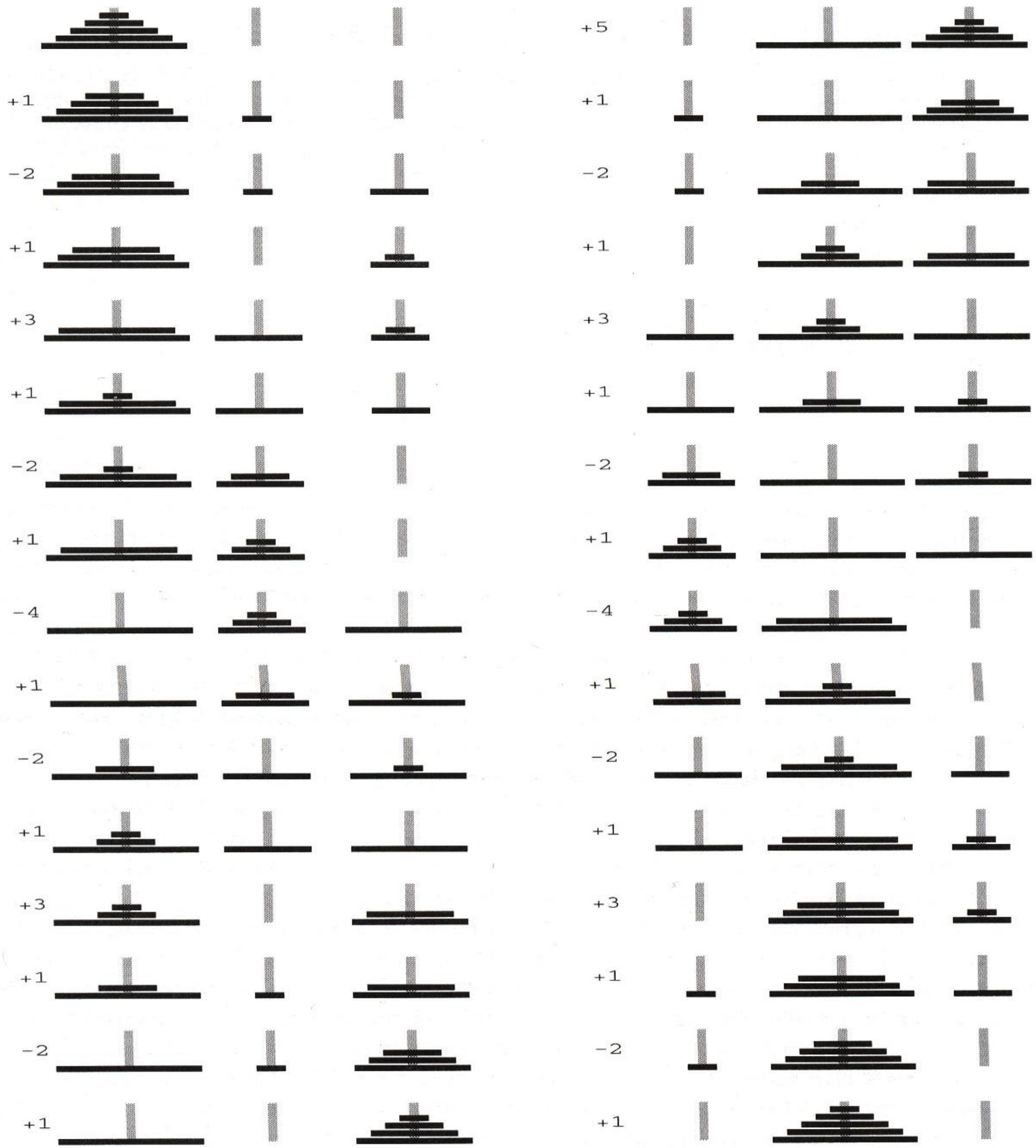
# Solution Idea

- To move N disks one peg to right, we first move the top N-1 disks one peg to the left, then shift disk N one peg to the right, then move the N-1 disks one more peg to the left (onto disk N)

# Towers of Hanoi

```
public ststic void hanoi(int N, int d)
  {
    if (N == 0) return;

    hanoi(N-1, -d); // move N-1 disks left
    shift(N, d);    // shift Nth disk to the right
    hanoi(N-1, -d); // move  all N-1 disks back one peg left
                    // on top of the Nth disk
  }
```



+1
−2
+1
+3
+1
−2
+1
−4
+1
−2
+1
+3
+1
−2
+1
+5
+1
−2
+1
+3
+1
−2
+1
−4
+1
−2
+1
+3
+1
−2
+1

# The Complexity of the Solution

- Looking at the code gives us immediate recursive formula of time:
- $T_N = 2T_{N-1} + 1,\ \ N >= 2\ T_1 = 1$

```
void hanoi(int N, int d)
 {
   if (N == 0) return;

   hanoi(N-1, -d);
   shift(N, d);
   hanoi(N-1, -d); }
```

- By induction:
  - $T(1)=2^1-1=1$; $T(k)=2^k-1$, for $k<N$
  - $\mathbf{T(N)=}2(2^{N-1}-1)+1= \mathbf{2^N-1}$

# Time Estimate

- If the monks are moving disks at the rate of one disk per second, it will take at least 348 centuries for them to finish.

| *seconds* | |
|---|---|
| $10^2$ | 1.7 minutes |
| $10^4$ | 2.8 hours |
| $10^5$ | 1.1 days |
| $10^6$ | 1.6 weeks |
| $10^7$ | 3.8 months |
| $10^8$ | 3.1 years |
| $10^9$ | 3.1 decades |
| $10^{10}$ | 3.1 centuries |
| $10^{11}$ | *never* |