# Course Survey and Final

- Course survey
  - **https://sakai.rutgers.edu/portal/site/sirs**
- Final
  - Dec 17, 2015: 4:00 PM - 7:00 PM
  - https://finalexams.rutgers.edu/

# CS111
# Introduction to Computer Science

Fall 2015

- Recursion
  - Factorial
  - Fibonacci
  - Palindromes
  - Towers of Hanoi
  - Binary Search
  - Mergesort

# A problem divided into smaller identical sub-problems

Some problems can be divided into several sub-problems that are similar to the original problem but smaller in size

Factorial

```
n! = n * n-1 * n-2 * n-3 * … * 1
4! = 4 * [3 * 2 * 1]
3! = 3 * [2 * 1]
2! = 2 * [1]
1! = 1
```

To solve for n!
n = n * (n-1)!

# Recursion

We can use a *recursive solution* to solve such a problem

- the solution to the problem depends on the solution of the sub-problems (smaller instances of the same problem)

Recursive algorithms

- to solve a given problem, they call themselves recursively

# Recursive Factorial Solution

n! = n * (n-1)!

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

base case

recursive case

Two cases:

- recursive case
- base case (stopping point)

# Recursive Factorial Solution

n! = n * (n-1)!

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

Call stack for the execution of factorial(5)

```
factorial(5)
    factorial(4)
        factorial(3)
            factorial(2)
                factorial(1)
                    return 1
                return 2*1 = 2
            return 3*2 = 6
        return 4*6 = 24
    return 5*24 = 120
```

# Recursive Algorithms Analysis

- To analyze the running time of a recursive algorithm we write the recurrence relation as a function of the input size *n*

  $T(n) = T(n-1) + 1$

- Then we repeat the recurrence to find a pattern that will give us the running time.

# Factorial Analysis

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

- Checking the if statement and multiplying by *n* is O(1)

  Time for factorial (*n*) = Time for factorial (*n*-1) + O(1)

- Dropping the Big O for the moment, we get the recurrence: $T(n) = T(n-1) + 1$

# Factorial Analysis

- Then we repeat the recurrence

$T(n) = T(n-1) + 1$

$= T(n-2) + 1 + 1 \quad = T(n-2) + 2$

$= T(n-3) + 1 + 2 \quad = T(n-3) + 3$

- The pattern is

$T(n) = T(n-k) + k$

- Let k = n

$T(n) = T(0) + k$ where $T(0) = 1$ which is the base case for factorial(0)

$T(n) = 1 + n = O(n)$

# Method call and return

- When a method is *called*, it starts up from the *beginning* with a new frame (activation record)
- When a method *calls* a method, the method doing the call *waits*, and its frame is saved
- When a call *returns* to a waiting frame, that invocation activates the waiting frame and it continues from *where it left off*

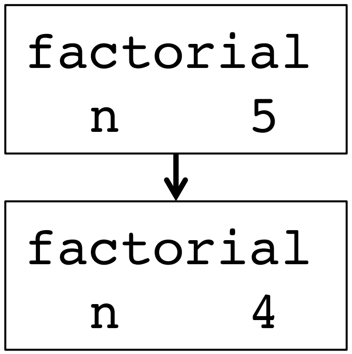
# Execution of factorial(5)

| factorial | |
|---|---|
| n | 5 |

Call Sequence

`factorial(5)`

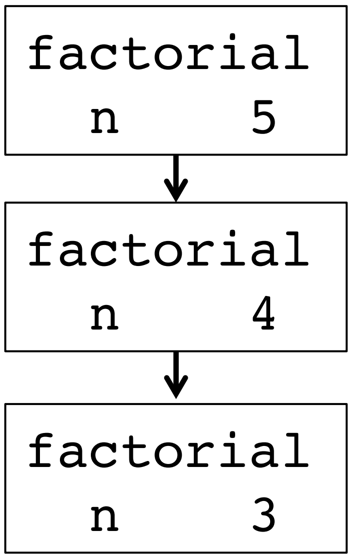
# Execution of factorial(5)




Call Sequence

`factorial(5)`
`factorial(4)`

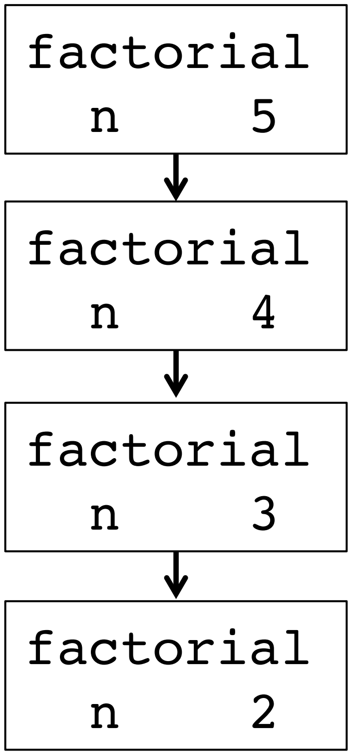
# Execution of factorial(5)




Call Sequence

```
factorial(5)
factorial(4)
factorial(3)
```

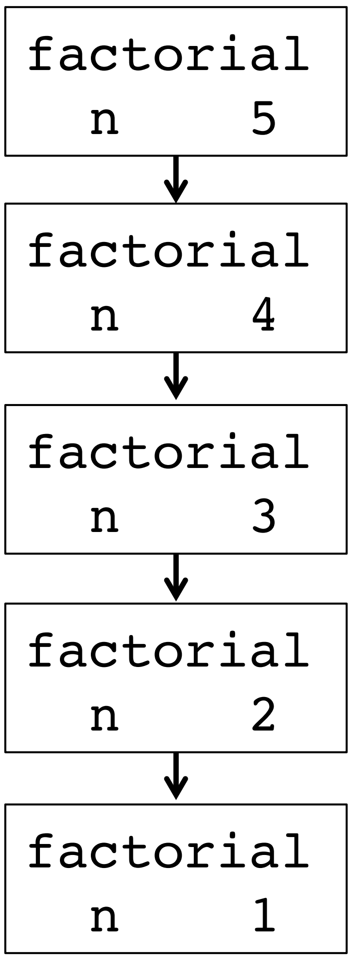
# Execution of factorial(5)




Call Sequence

```
factorial(5)
factorial(4)
factorial(3)
factorial(2)
```

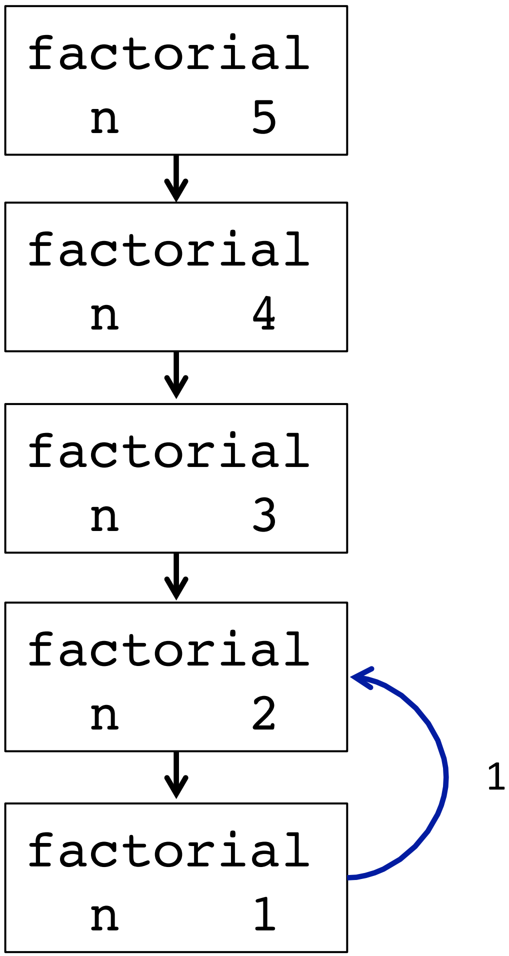
# Execution of factorial(5)




Call Sequence

```
factorial(5)
factorial(4)
factorial(3)
factorial(2)
factorial(1)
```

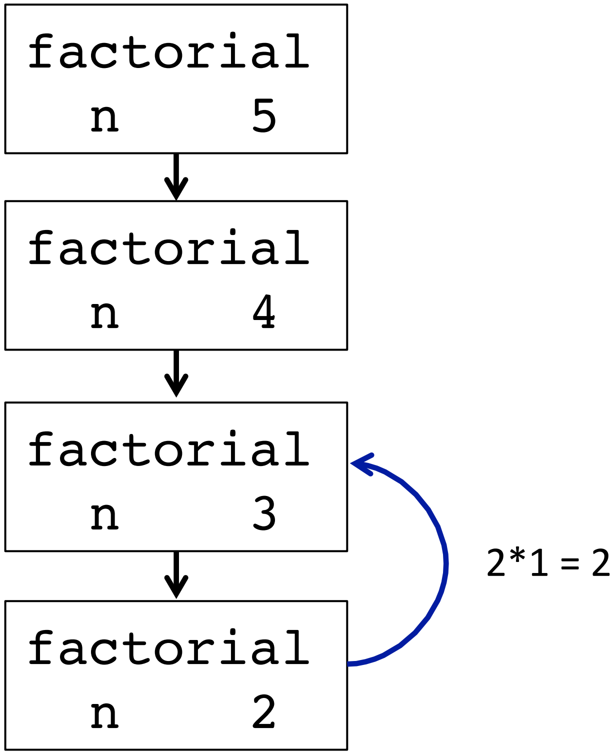
# Execution of factorial(5)




Call Sequence

```
factorial(5)
factorial(4)
factorial(3)
factorial(2)
factorial(1)
```

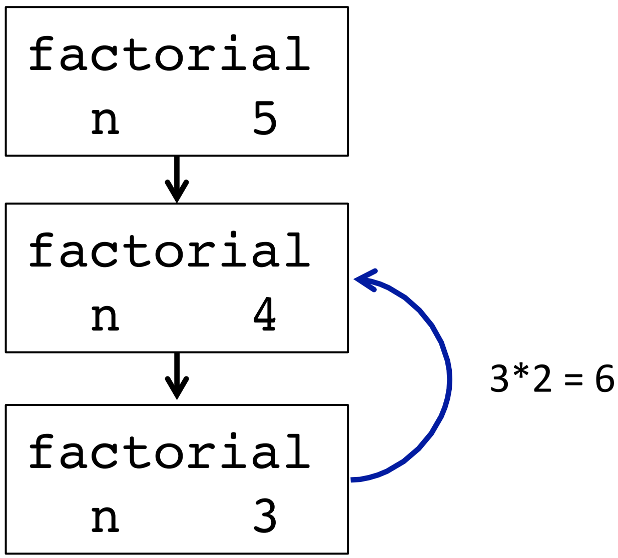
# Execution of factorial(5)




Call Sequence

```
factorial(5)
factorial(4)
factorial(3)
factorial(2)
factorial(1)
```

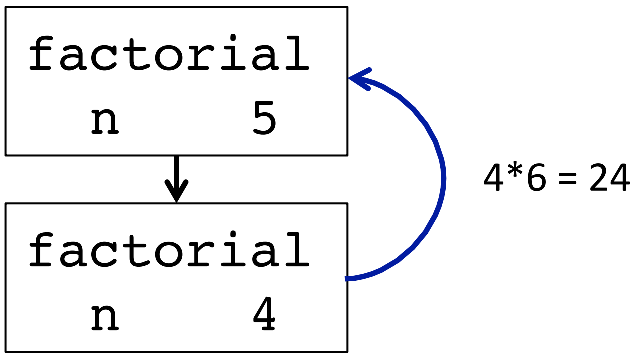
# Execution of factorial(5)




Call Sequence

```
factorial(5)
factorial(4)
factorial(3)
factorial(2)
factorial(1)
```

# Execution of factorial(5)




Call Sequence

```
factorial(5)
factorial(4)
factorial(3)
factorial(2)
factorial(1)
```

# Execution of factorial(5)

```
factorial
  n     5
```

5*24 = 120

Call Sequence

```
factorial(5)
factorial(4)
factorial(3)
factorial(2)
factorial(1)
```

Output

```
120
```

# Recursive versus Iterative

Both solve a problem one piece at a time

- Recursive
  - the solution to a problem depends on solutions to smaller instances of the same problem
- Iterative
  - the solution to a problem depends on repeating a process that after each iteration brings the result closer to the solution (without recursion)

# Iterative Factorial Solution

n! = n * n-1 * n-2 * … * 1

```
public static int factorial (int n) {
   int fact = 1;
   for (int i = n; i > 1; i--) {
      fact *= i;
   }
   return fact;
}
```

# Printing Pattern: a triangle

- Iterative view (non-recursive)

```
*
**
***
****
```

A size 4 triangle is four lines or length 1, 2,3 and 4

- Recursive view

```
*
**
***
****
```

A size 4 triangle is a size 3 triangle followed by a line of length 4

# Iterative Printing Pattern

```
public static void main(String[] args) {
    for (int i = 1; i <= 3; i++) {
        printNStars(i);
    }
}
public static void printNStars (int n) {
    System.out.println(nTimesChar(n, '*'));
}
public static String nTimesChar (int n, char c) {
    String result = "";
    for (int i = 1; i <= n; i++) {
        result = result + c;
    }
    return result;
}
```

# Recursive Printing Pattern

```
public static void triangle (int n) {
    if (n == 1) {
      printNStars(1);
    } else {
       triangle(n-1);
       printNStars(n);
    }
}
public static void printNStars (int n) {
       System.out.println(nTimesChar(n, '*'));
}
public static String nTimesChar (int n, char c) {
    String result = "";
    for (int i = 1; i <= n; i++) {
       result = result + c;
    }
    return result;
}
```

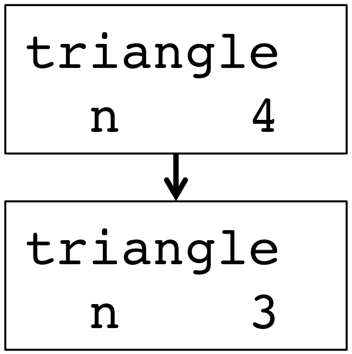
# Recursive Printing Pattern: execution

```
triangle
  n    4
```

Call Sequence

`triangle(4)`

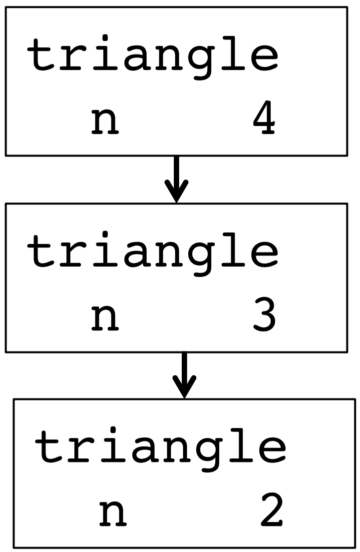
# Recursive Printing Pattern: execution




Call Sequence

```
triangle(4)
triangle(3)
```

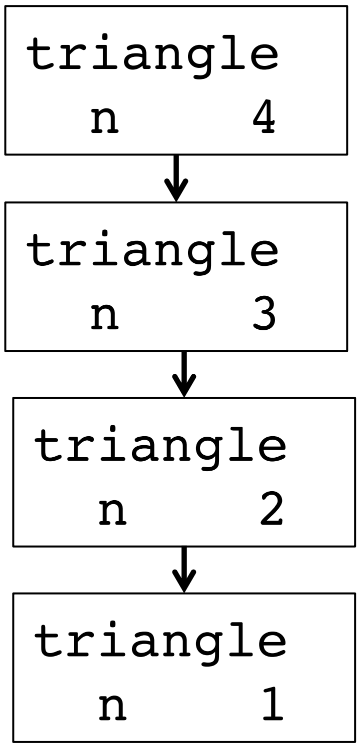
# Recursive Printing Pattern: execution




Call Sequence

```
triangle(4)
triangle(3)
triangle(2)
```

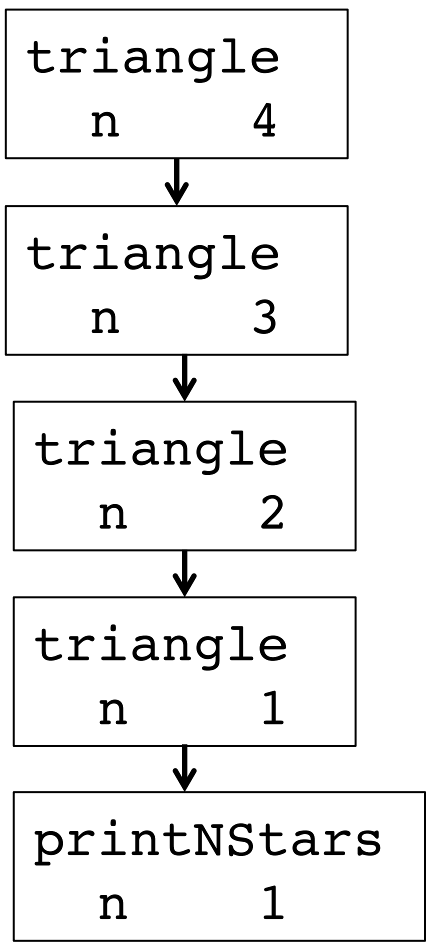
# Recursive Printing Pattern: execution




Call Sequence

```
triangle(4)
triangle(3)
triangle(2)
triangle(1)
```

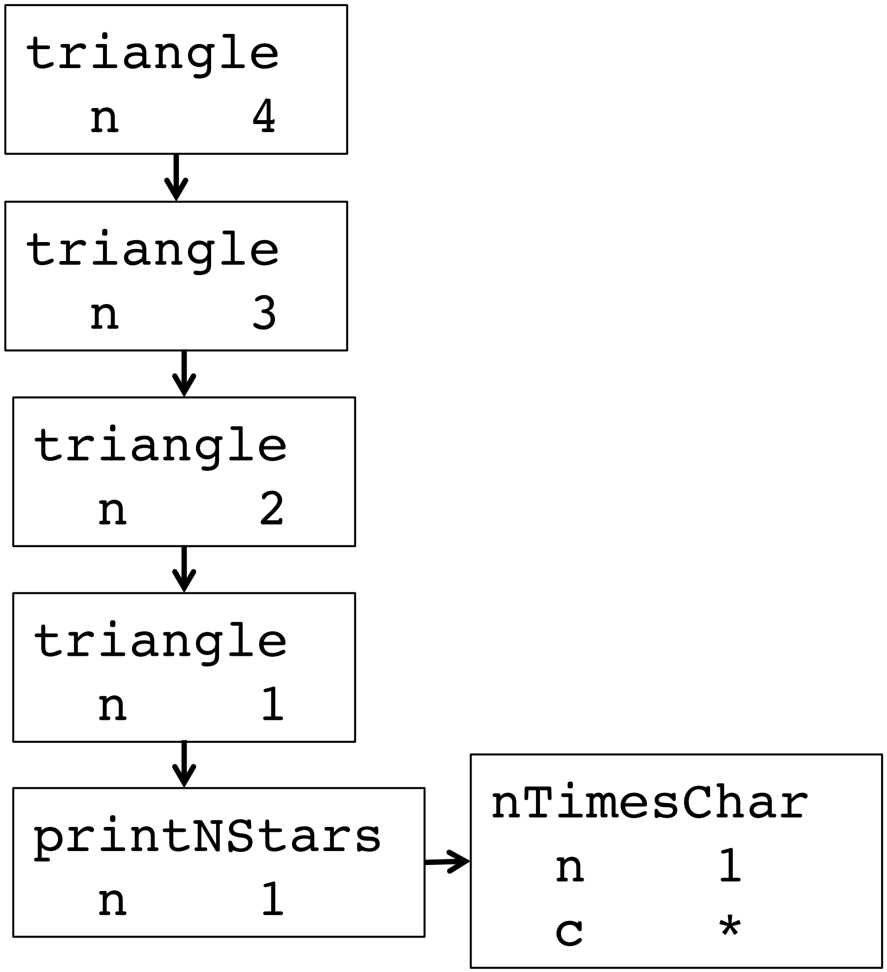
# Recursive Printing Pattern: execution




Call Sequence

```
triangle(4)
triangle(3)
triangle(2)
triangle(1)
printNStars(1)
```

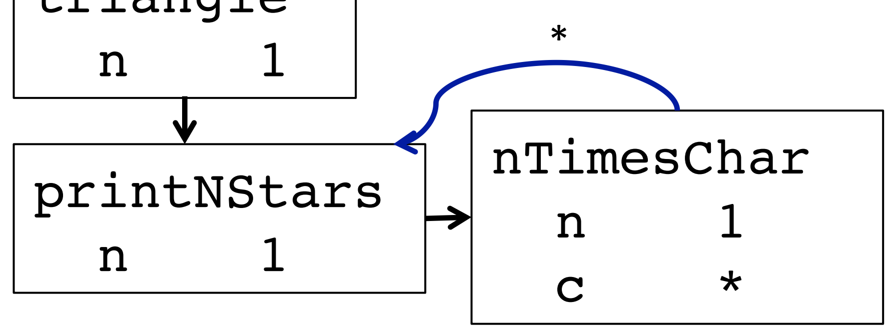
# Recursive Printing Pattern: execution




Call Sequence

```
triangle(4)
triangle(3)
triangle(2)
triangle(1)
printNStars(1)
nTimesChar(1, '*')
```

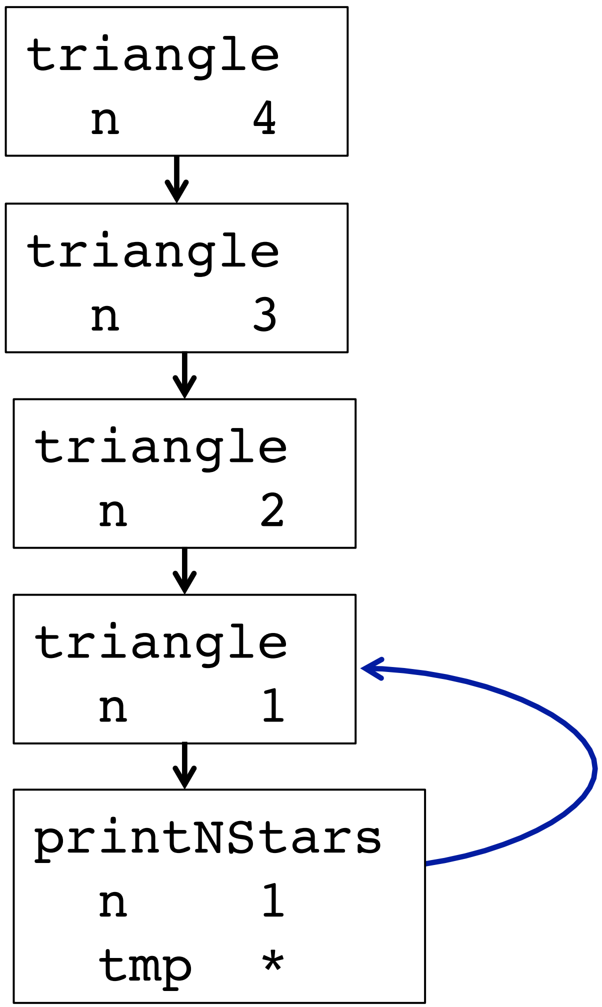
# Recursive Printing Pattern: execution




Call Sequence

```
triangle(4)
triangle(3)
triangle(2)
triangle(1)
printNStars(1)
nTimesChar(1, '*')
```

# Recursive Printing Pattern: execution



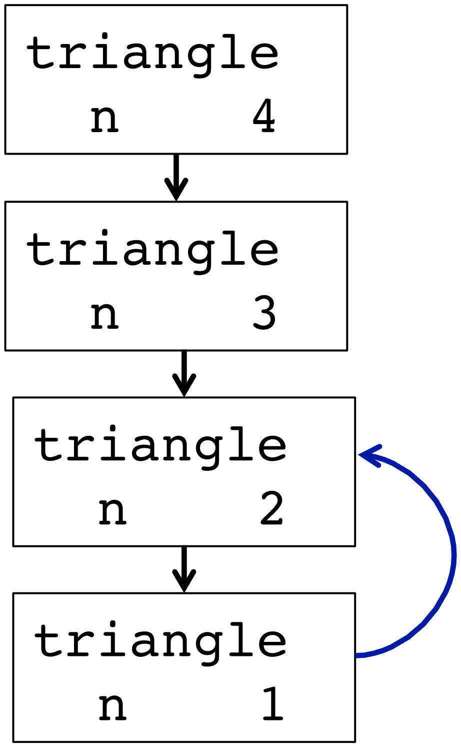

Call Sequence

```
triangle(4)
triangle(3)
triangle(2)
triangle(1)
printNStars(1)
nTimesChar(1, '*');
```

Output

```
*
```

# Recursive Printing Pattern: execution



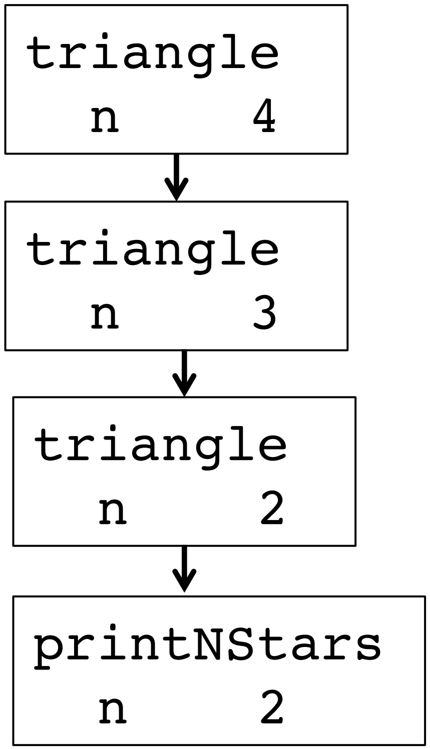

Call Sequence

```
triangle(4)
triangle(3)
triangle(2)
triangle(1)
printNStars(1)
nTimesChar(1, ‘*’);
```

Output

```
*
```

# Recursive Printing Pattern: execution



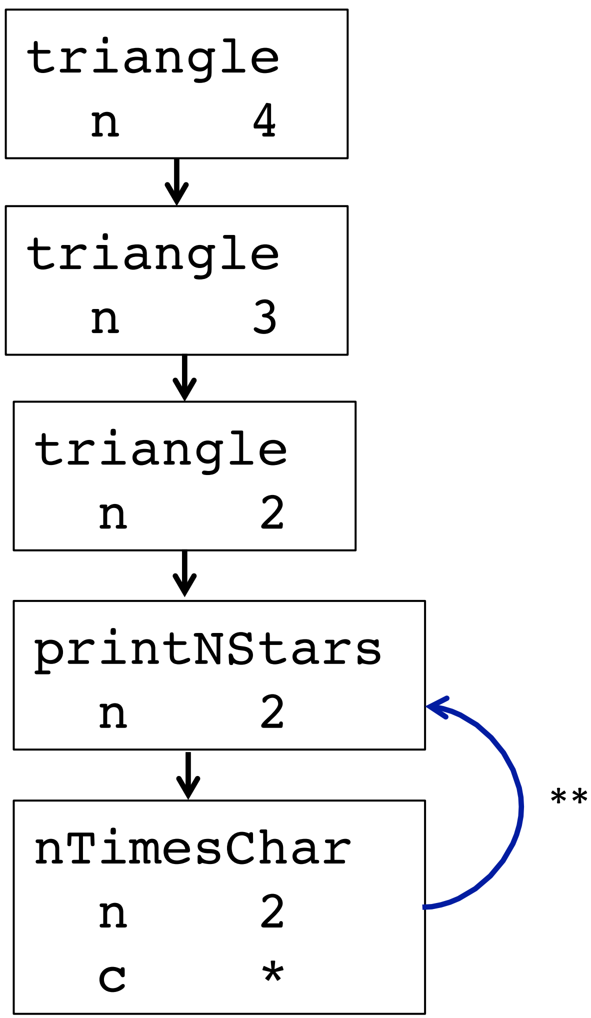

Call Sequence

```
triangle(4)
triangle(3)
triangle(2)
triangle(1)
printNStars(1)
nTimesChar(1, '*')
printNStars(2)
```

Output

```
*
```

# Recursive Printing Pattern: execution



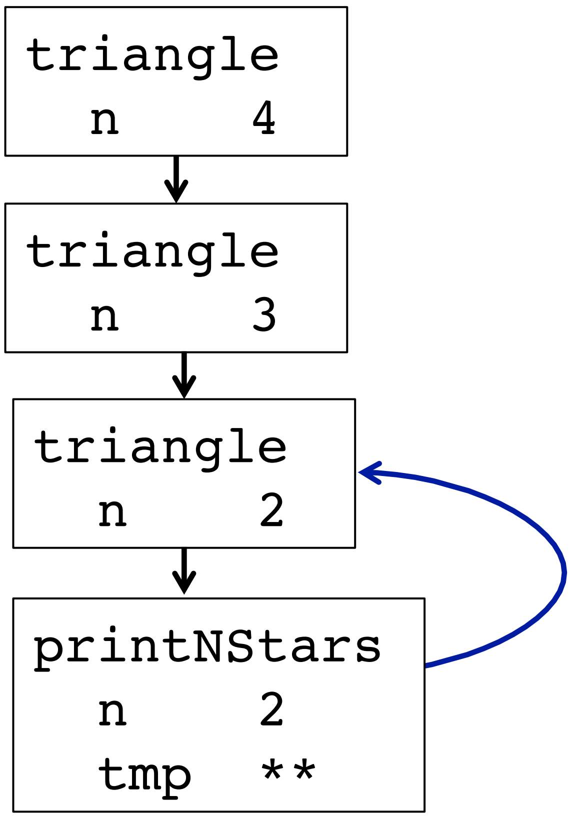

Call Sequence

```
triangle(4)
triangle(3)
triangle(2)
triangle(1)
printNStars(1)
nTimesChar(1, '*')
printNStars(2)
nTimesChar(2, '*')
```

Output

*

# Recursive Printing Pattern: execution



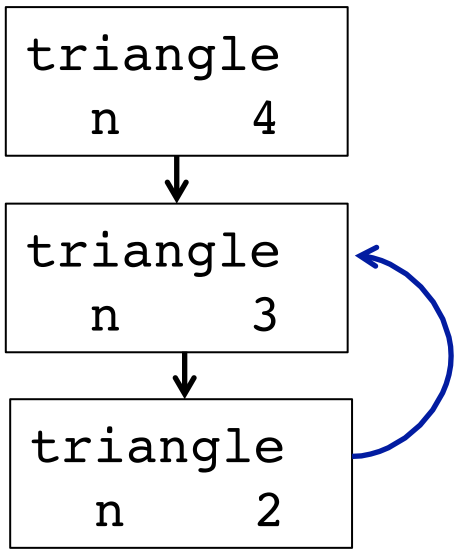

Call Sequence

```
triangle(4)
triangle(3)
triangle(2)
triangle(1)
printNStars(1)
nTimesChar(1, '*')
printNStars(2)
nTimesChar(2, '*')
```

Output

```
*
**
```

# Recursive Printing Pattern: execution




Call Sequence

```
triangle(4)
triangle(3)
triangle(2)
triangle(1)
printNStars(1)
nTimesChar(1, '*')
printNStars(2)
nTimesChar(2, '*')
```

Output

```
*
**
```

# Recursive Printing Pattern: execution

```
triangle
  n    4
```

```
triangle
  n    3
```

```
printNStars
  n    3
```

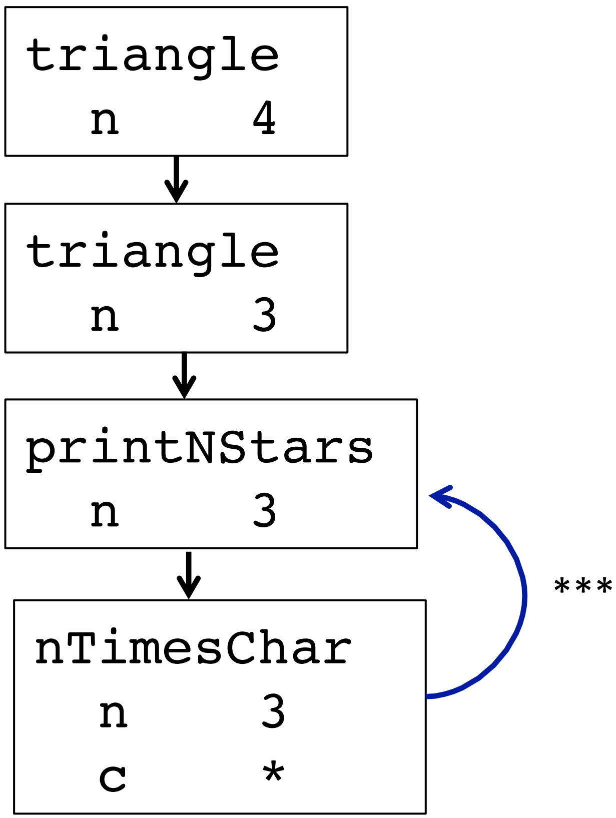
Call Sequence

```
triangle(4)
triangle(3)
triangle(2)
triangle(1)
printNStars(1)
nTimesChar(1, '*')
printNStars(2)
nTimesChar(2, '*')
printNStars(3)
```

Output

```
*
**
```

# Recursive Printing Pattern: execution



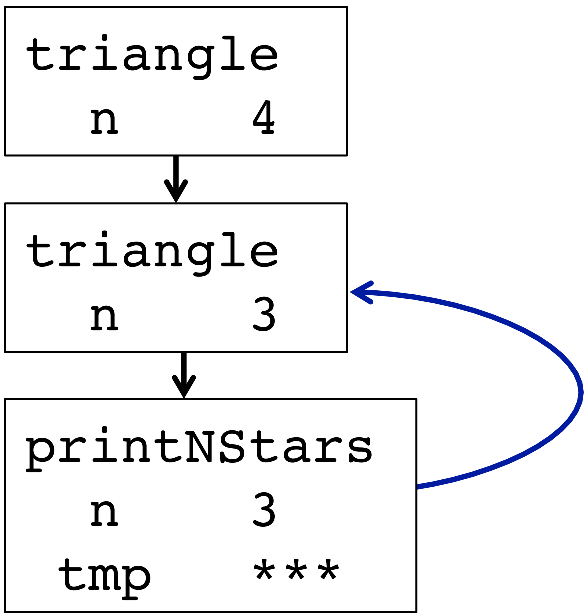

Call Sequence

```
triangle(4)
triangle(3)
triangle(2)
triangle(1)
printNStars(1)
nTimesChar(1, '*')
printNStars(2)
nTimesChar(2, '*')
printNStars(3)
nTimesChar(3, '*')
```

Output

```
*
**
```

# Recursive Printing Pattern: execution



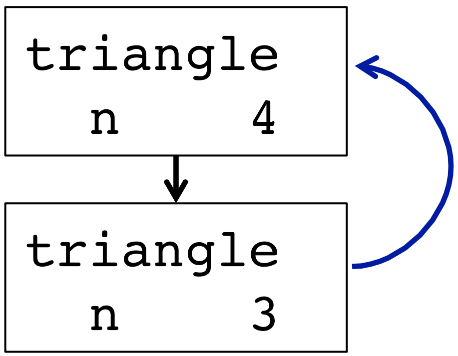

Call Sequence

```
triangle(4)
triangle(3)
triangle(2)
triangle(1)
printNStars(1)
nTimesChar(1, '*')
printNStars(2)
nTimesChar(2, '*')
printNStars(3)
nTimesChar(3, '*')
```

Output

```
*
**
***
```

# Recursive Printing Pattern: execution




Call Sequence

```
triangle(4)
triangle(3)
triangle(2)
triangle(1)
printNStars(1)
nTimesChar(1, '*')
printNStars(2)
nTimesChar(2, '*')
printNStars(3)
nTimesChar(3, '*')
```

Output

```
*
**
***
```

# Recursive Printing Pattern: execution

```
triangle
  n     4
```

```
printNStars
  n     4
```

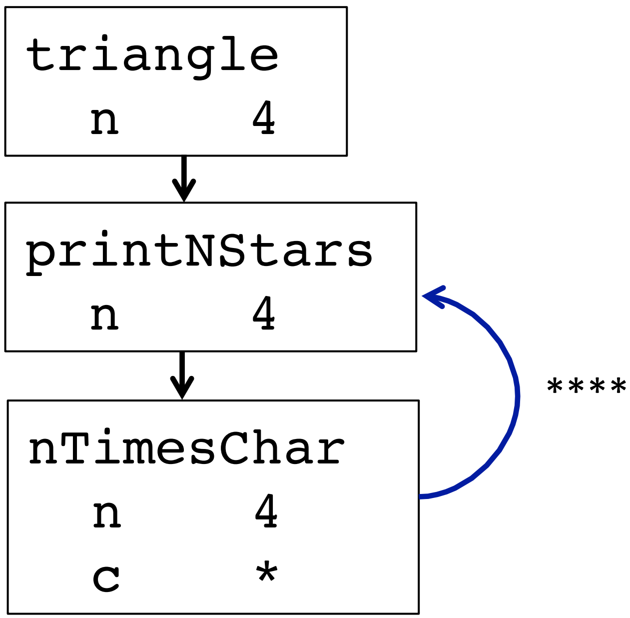
Call Sequence

```
triangle(4)
triangle(3)
triangle(2)
triangle(1)
printNStars(1)
nTimesChar(1, ‘*’)
printNStars(2)
nTimesChar(2, ‘*’)
printNStars(3)
nTimesChar(3, ‘*’)
printNStars(4)
```

Output

```
*
**
***
```

# Recursive Printing Pattern: execution



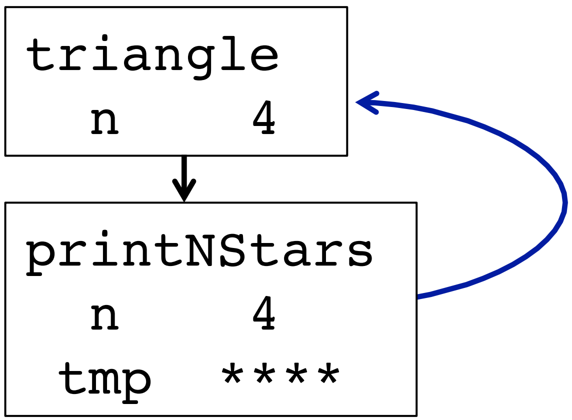

Call Sequence

```
triangle(4)
triangle(3)
triangle(2)
triangle(1)
printNStars(1)
nTimesChar(1, '*')
printNStars(2)
nTimesChar(2, '*')
printNStars(3)
nTimesChar(3, '*')
printNStars(4)
nTimesChar(4, '*')
```

Output

```
*
**
***
```

# Recursive Printing Pattern: execution




Call Sequence

```
triangle(4)
triangle(3)
triangle(2)
triangle(1)
printNStars(1)
nTimesChar(1, '*')
printNStars(2)
nTimesChar(2, '*')
printNStars(3)
nTimesChar(3, '*')
printNStars(4)
nTimesChar(4, '*')
```

Output

```
*
**
***
****
```

# Recursive Printing Pattern: execution

```
triangle
  n     4
```

Call Sequence

```
triangle(4)
triangle(3)
triangle(2)
triangle(1)
printNStars(1)
nTimesChar(1, '*')
printNStars(2)
nTimesChar(2, '*')
printNStars(3)
nTimesChar(3, '*')
printNStars(4)
nTimesChar(4, '*')
```

Output

```
*
**
***
****
```

# Iterative versus Recursive

**Iterative**

- Faster
  - no repetitive creation of activation records
- More code lines

**Recursive**

- Slower
  - each method call takes time to execute and takes memory space. If there are too many calls, could run out of memory
- Less code lines
  - code is easier to read, often more elegant

# Palindromes

- A sequence of characters that reads the *same in both directions* (forward and backward)
  - e.g. radar, madam, eye

- How to write a recursive method to test if a string is palindrome?
  - disregard punctuation and space

# Palindromes

- A string is palindrome if:
  - first and last characters are the same, and

    madam

  - rest of the string without the first and last characters is a palindrome

    ada

- A string of length 0 or 1 is a palindrome

  d

recursive case

base case

# Palindromes

```
public static boolean palindrome (String word) {
   int length = word.length();
    if (length <= 1) {
      return true;
    } else {
       return word.charAt(0)==word.charAt(length-1) &&
           palindrome(word.substring(1,length-1);
    }
}

public static void main (String[] args) {
    System.out.println(palindrome(args[0]));
}
```

base case

recursive case

gets the string from the command line

# Palindromes Analysis

- The recurrence T(n)
  - $T(n) = T(n-2) + 1$, where n is the string length
- The pattern is
  - $T(n) = T(n-2k) + k$
- Let $k = n/2$
  - $T(n) = O(n)$

# Fibonacci Sequence

The first two numbers are 1 and 1, and each subsequent number is the sum of the previous two numbers

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

- How to write the Fibonacci sequence using a recursive method?
  - base case?
  - recursive case?

# Fibonacci Sequence

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

base case

recursive case

- Call stack for the execution of fib(4):

```
fib(4)
  fib(3)
    fib(2)
      fib(1)
      fib(0)
    fib(1)
  fib(2)
    fib(1)
    fib(0)
```

- Recurrence Relation
  - $T(0) = 1$
  - $T(1) = 1$
  - $T(n) = T(n-1) + T(n-2) + 1$

# Fibonacci Analysis

- Then we repeat the recurrence

  $T(n) = T(n-1) + T(n-2) + 1$ level 1

  $= [T(n-2) + T(n-3) + 1] + [T(n-3) + T(n-4) + 1] + 1$ level 2

- Sometimes we can't write the full pattern but we can still figure out how many computation we are making
  - if we repeat the recurrence we are going to get 8 T's on level 3, 16 T's on level 4, 32, 64 and so on.
  - So, there are $2^k$ T's at level k
  - To get down to level T(n-1) to the base case T(1), we'll need to go to level k = n-1
  - We'll have $2^{n-1}$ T's there, so $T(n) = O(2^n)$

# Towers of Hanoi

The objective is to move the entire stack of disks to another peg

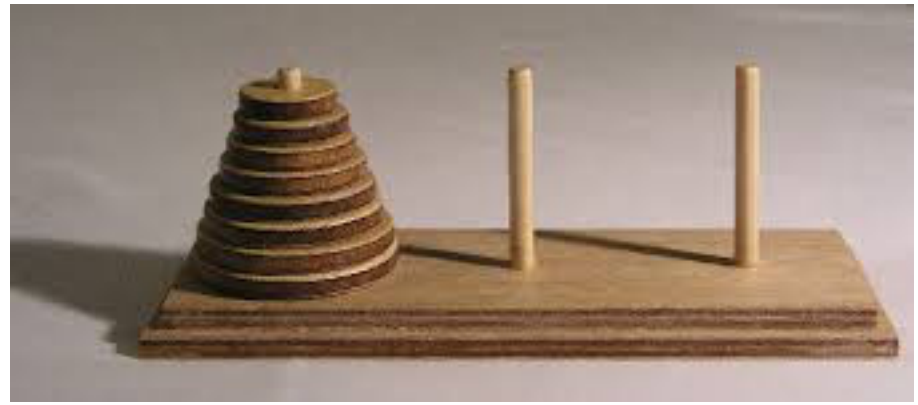


Image from Wikipedia

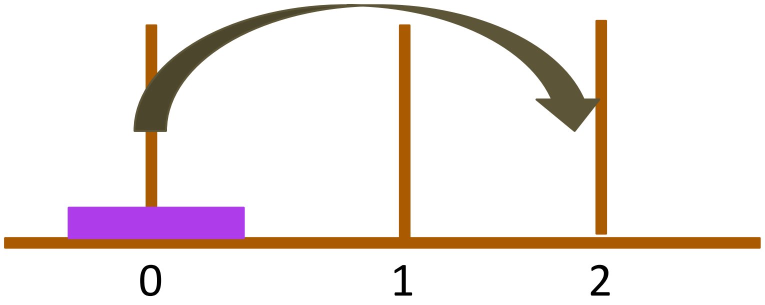
Rules

- 3 pegs and a set of disks of different sizes
- move only one disk at a time
  - take the top disk from a stack
  - put it on the top of another stack
- never place a larger disk on top of a smaller one

# Towers of Hanoi with n disks

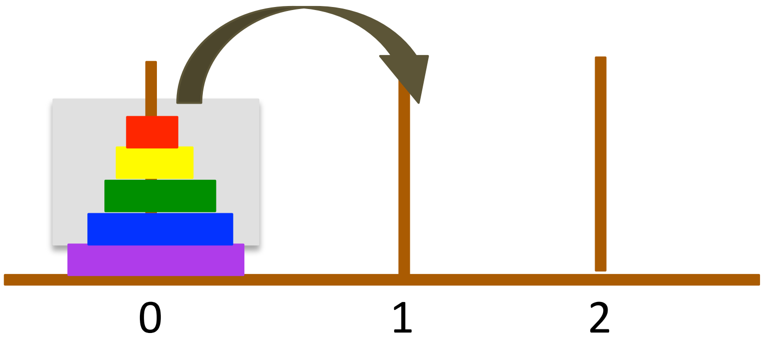
**Base case**

- identifying the stopping point, smallest Tower of Hanoi problem
- move only one disk



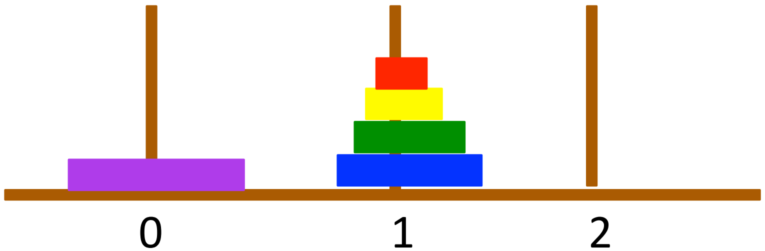

**Recursive case**

- identifying the smaller Towers of Hanoi problem within the Tower of Hanoi problem
- move n-1 disks to spare peg

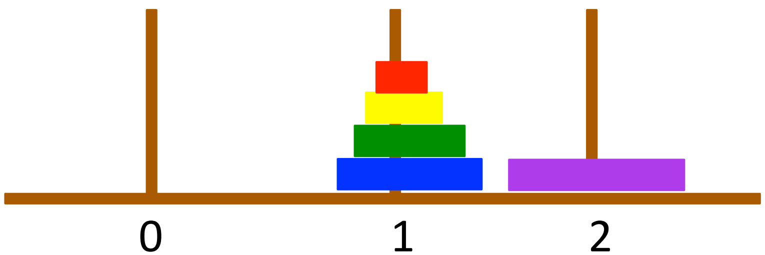


# Towers of Hanoi

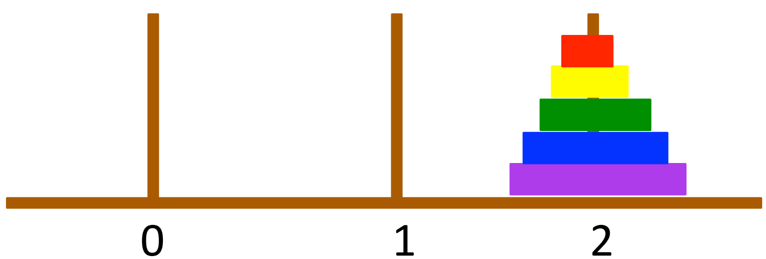
- Move the top n-1 disks to the *spare* peg




- Move the bottom disk to the *to* peg




- Move the n-1 disks from the *spare* to the *to* peg



# Towers of Hanoi

```
static void towersOfHanoi(int disks, int from, int to, int
spare) {
    if (disks == 1) {
        // There is only one disk to be moved. Just move it.
        System.out.printf("Move disk 1 from stack %d to
stack %d%n", from, to);

    } else {
        // Move all but one disk to the spare stack
        towersOfHanoi(disks-1, from, spare, to);
        // move the bottom disk
        System.out.printf("Move disk %d from stack %d to
stack %d%n", disks, from, to);
        //then put all the other disks on top of it.
        towersOfHanoi(disks-1, spare, to, from);
    }
}
```

# Towers of Hanoi: Efficiency

- Let $M(n)$ be the number of moves required to move n disks
- There are 2 recursive calls for n-1 disks and one constant time operation to move a disk
- Therefore:
  - $M(n) = 2M(n-1) + 1$
  - $M(1) = 1$
- Forward substitution
  - $M(2) = 2\ M(1) + 1 = 3$
  - $M(3) = 2\ M(2) + 1 = 7$
  - $M(4) = 2\ M(3) + 1 = 15$

$O(2^n)$

# Binary Search

- Searching an ordered array
  - How to find 12?

| 3 | 5 | 12 | 22 | 56 | 62 | 85 | 94 | 95 | 99 | |
|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | Array index |

| Left | Right | Middle |
|---|---|---|
| 0 | 9 | 4 |
| 0 | 3 | 1 |
| 2 | 3 | 2 |

12 == 56? no 12 < 56? yes

12 == 5? no 12 < 5? no

12 == 12? yes **Found it!**

# Recursive Binary Search

```
int binarySearch(int[] a, int l, int r, int target){
   if (l > r) return -1;
   int middle = (l+r)/2;
   if (a[middle] = target)
      return middle;
   else if (a[middle] > target)
      return binarySearch(a, l, m-1, target);
   else return binarySearch(a, m+1, r, target);
}
```

- The comparisons and the computation of middle take constant time, O(1)
- Each call to binary search has three comparisons and a recursive call on half of the array, so the recurrence relation is:
  $T(n) = T(n/2) + O(1)$
  $T(n) = T(n/2) + 1$

# Recursive Binary Search: Analysis

- Repeat the recurrence

  $= T(n/2) + 1$

  $= T(n/4) + 1 + 1$

  $= T(n/8) + 1 + 1 + 1$

  $\ldots = T(n/2^k) + k$

- Let $n = 2^k$

  $= T(2^k/2^k) + k$

  $= T(1) + k$

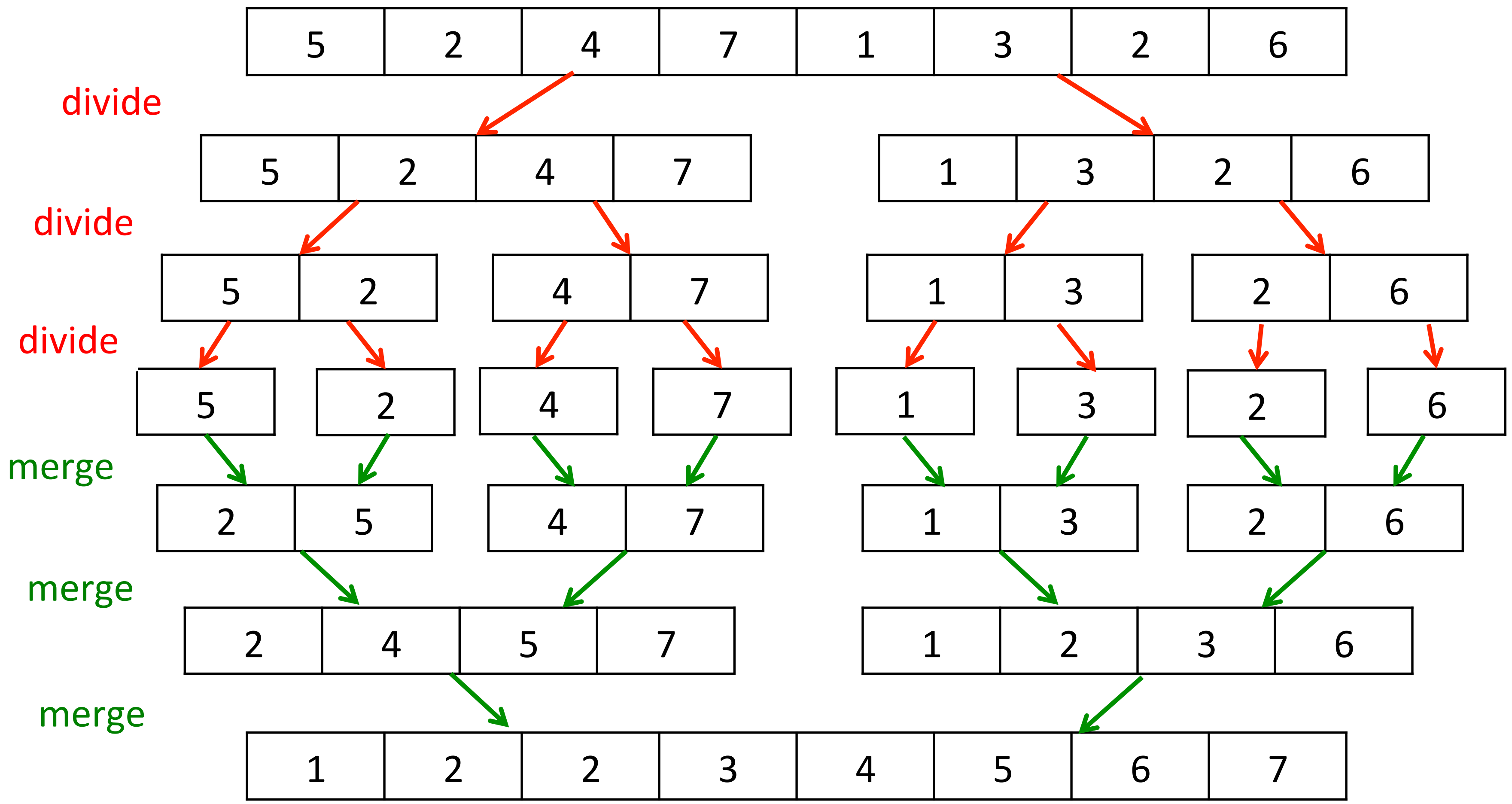
  $= 1 + k$

  if $n = 2^K$ then $k = \log n$ and $T(n) = 1 + \log n = O(\log n)$

# Mergesort

- Follows the divide-and-conquer approach
  1. break the problem into several sub-problems that are similar to the original problem but smaller in size
  2. solve the sub-problems recursively
  3. then combine the sub-problems solution to create a solution to the original problem

# Mergesort



5 2 4 7 1 3 2 6
divide
5 2 4 7
1 3 2 6
divide
5 2
4 7
1 3
2 6
divide
5
2
4
7
1
3
2
6
merge
2 5
4 7
1 3
2 6
merge
2 4 5 7
1 2 3 6
merge
1 2 2 3 4 5 6 7

# Mergesort

Main idea for an array of n elements:

- Divide the unsorted array until there are n arrays, each containing 1 element.
  - Here the one element is considered as sorted.
- Repeatedly merge two sorted arrays until there is only 1 array remaining.
  - This will be the sorted array at the end.

# Mergesort

```
void mergesort (int[] a, int l, int r){
   if (l > r) return;
   int middle = (l+r)/2;
   mergesort(a, l, m);
   mergesort(a, m+1, r);
   merge(a, l, m, r);
}
```

```
void merge (int[] a, int l, int m, int r){
   int[] aux = new int[r-l+1];
   for (int i=l; i<=r; i++) aux[i] = a[i]; //copy
   int i = l, j = m+1;
   for (int k=l; k<=r; k++) {
      if (i >= m) a[k] = aux[j++];
      else if (j >=r) a[k] = aux[i++];
      else if (aux[j] < aux[i]) a[k] = aux[j++];
      else a[k] = aux[i++];
}
```

# Mergesort: Analysis

- Merge: merges two halves of the array
  - making a copy of the array is O(n)
    - constant time for the assignment done n times
  - comparisons O(n)
    - each comparison takes constant time
    - in the worst case there are n-1 comparisons
  - O(n) + O (n) = O (n)

# Mergesort: Analysis

- Mergesort
  - the comparison and the computation of middle is done in constant time O(1)
  - two calls to mergesort: each on half of the array
  - one call to merge O(n)
- Recurrence relation is

$$T(n) = T(n/2) + T(n/2) + O(n) + O(1)$$
$$= 2 * T(n/2) + n$$

# Mergesort: Analysis

- Repeat the recurrence

  $= T(n/2) + n$

  $= T(n/4) + n + n$

  $= T(n/8) + n + n + n$

  $\ldots = T(n/2^k) + k*n$

- Let $n = 2^k$

  $= T(2^k/2^k) + k*n$

  $= T(1) + k*n$

  $= 1 + k*n$

  if $n = 2^K$ then $k = \log n$ and $T(n) = 1 + \log n * n = O(n \log n)$