CS 2100: Data Structures & Algorithms 1

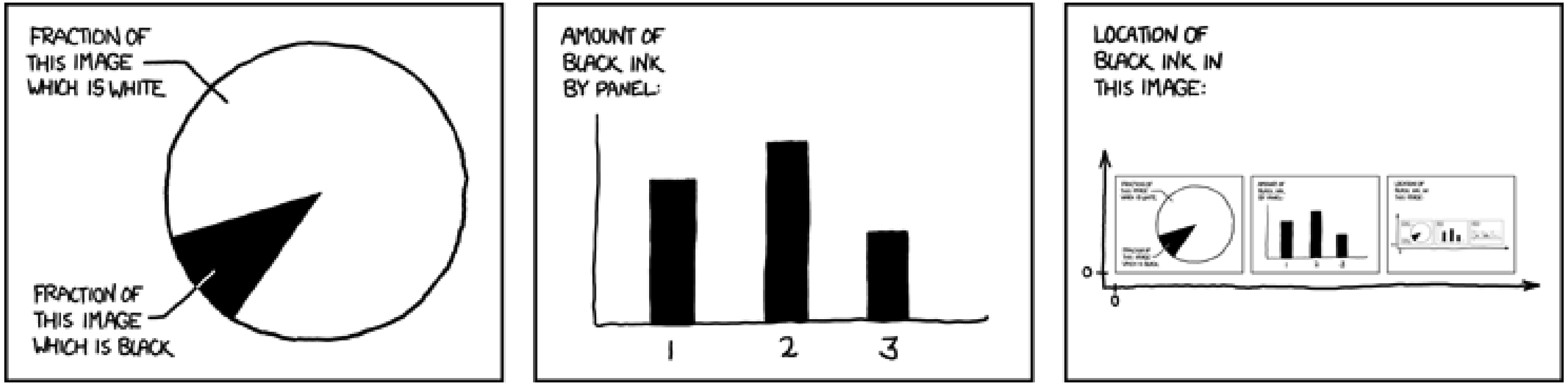
Trees

~Recursion & Examples~

Dr. Nada Basit // basit@virginia.edu

Spring 2022

In Order To Understand Trees…
… We Have To Understand Recursion



https://www.xkcd.com/688/

Friendly Reminders

- Masks are **required** at all times during class (University Policy)
- If you forget your mask (or mask is lost/broken), I have a few available
  - Just come up to me at the start of class and ask!
- No eating or drinking in the classroom, please
- Our lectures will be **recorded** (see Collab) – please allow 24-48 hrs to post
- If you feel **unwell**, or think you are, please stay home
  - *We will work with you!*
  - At home: eye mask instead! Get some rest ☺

Announcements / Reminders

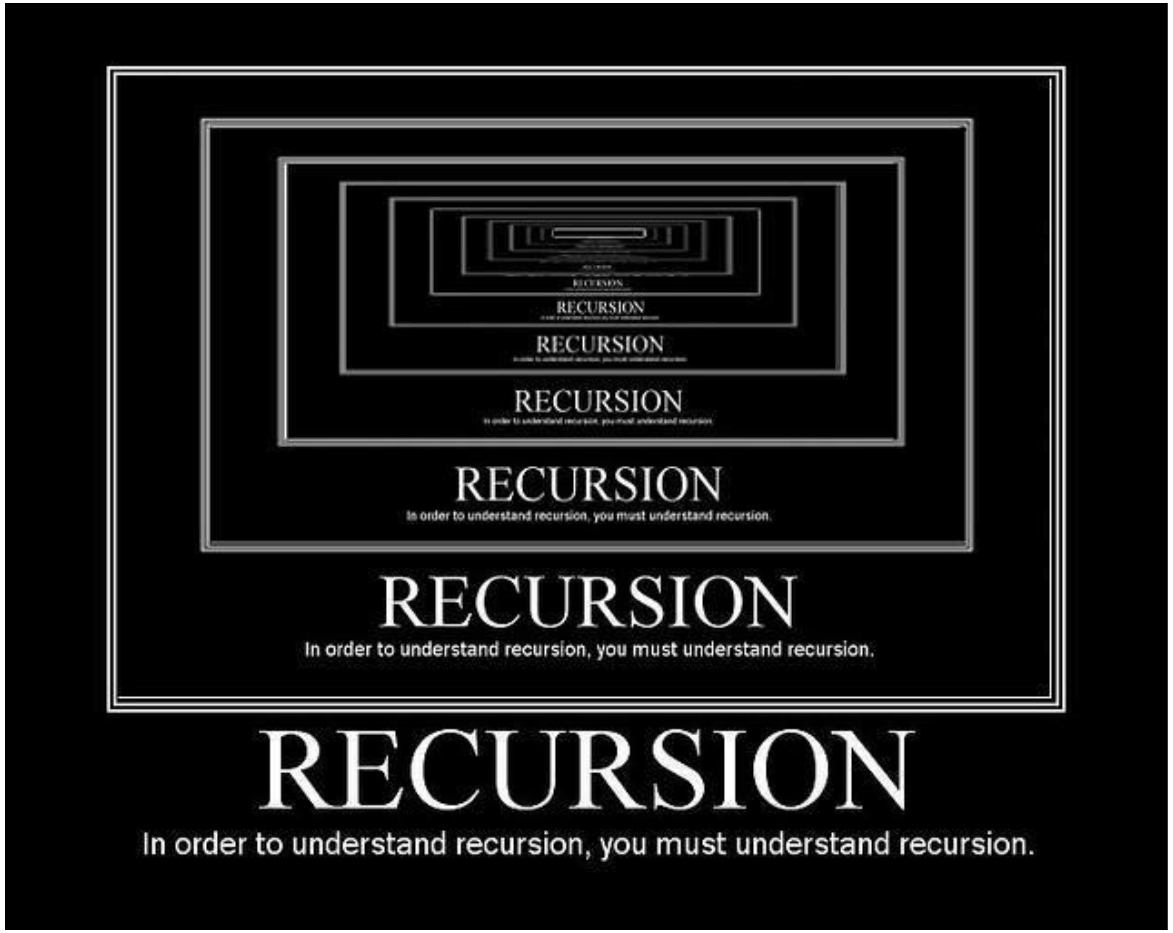
- **Lab tonight (Monday):**
  - Take Quiz 4 for this week – **LL, Stacks, and Queues** (**30 minutes**) – come to lab on time!
  - Once you're done with the quiz, you can work with your cohort on your **Big-Oh** assignments (coding and report)
- **Reminder of Homework Late Policy:** [Announcement sent 02/14/2022]
  - "Homework 1 (coding)" for each module:
    - Official due date: **Wednesday** by 11:59pm ET
    - Late period (with 10% penalty): 1 week; until the following Wednesday by 11:59pm ET
  - "Homework 2 (analysis)" for each module *[if applicable]*:
    - Official due date: **Friday** by 11:59pm ET
    - Late period (with 10% penalty): 3 days; until following Monday by 11:59pm ET
  - Manage your time wisely, seek help (TAs or Profs) when needed, ***use grace period as your extension*** if need be.

Definition (don't write this one down!)

- Recursion

Definition (don't write this one down!)

- Recursion
  - see recursion



What Is Recursion?

- A definition is **recursive** if it is defined in terms of itself
- Recursion is a natural way to express many **algorithms** – in which a method invokes itself to solve a problem.
- For **recursive data-structures**, recursive algorithms are a natural choice
- **Recursive mindset:**
  - *Recursion breaks a difficult problem into one or more simpler (smaller) versions of itself*
- Why do we care? **Trees** use recursion ALL OF THE TIME. So, we need to know it.

Important Recursive Definitions

A recursive Solution contains:

- **BASE CASE**
  - The case for which the solution can be stated **non-recursively** (or solved directly)*. *That is, directly solving the smallest instance of the problem.*
- **RECURSIVE CASE**
  - The case for which the solution is expressed in terms of a **smaller version of itself**. Solve a small chunk manually then **invoke** your method.
  - *You should be making progress towards your base case!*

* [Definition can't be completely self-referential! → need base case]

Extra Examples

Recursion in Algorithms

- Grammar example: What is a noun phrase?
  - a noun
  - an adjective followed by a noun phrase
- List example: Consider the following list of numbers: `24, 77, 18, 47`
  - Such a list can be defined as follows:
    - ```
      A LIST is a: number
               or a: number comma LIST
      ```
  - That is, a LIST is defined to be a single number, or a number followed by a comma followed by a LIST
  - The concept of a LIST is used to define itself
      ```

Extra Examples

Recursion in Algorithms

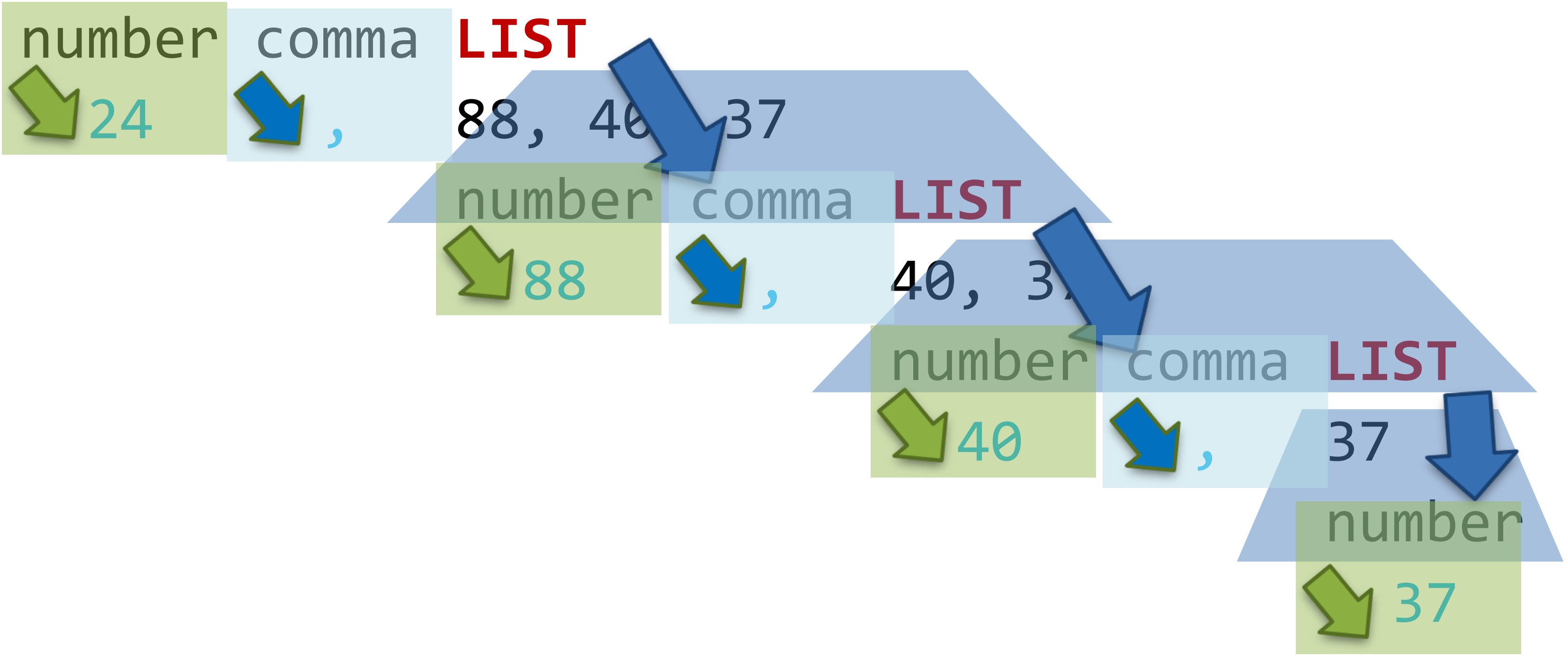
- The recursive part of the LIST definition is used several times, terminating with the non-recursive part:

```
number comma LIST
  24     ,   88, 40, 37
             number comma LIST
               88     ,   40, 37
                          number comma LIST
                            40     ,   37
                                       number
                                         37
```

Extra Examples

Recursion in Algorithms

- The recursive part of the LIST definition is used several times, terminating with the non-recursive part:



Different Views of Recursion

- **Recursive Definition**: **n! = n * (n-1)!**
  (*This example is the definition of* ***factorial****. Non-math examples are common too*)
- **Recursive Procedure**: a procedure that calls itself
- **Recursive Data Structure**: a data structure that contains a pointer to an instance of itself:

```
public class ListNode {
      Object nodeItem;
      ListNode next, previous;
      …
}
```

Questions To Ask Yourself

- How can we reduce the problem to smaller version of the same problem?
- How does each call make the problem smaller?
- What is the **base case**? (Non-recursive part)
- Will you always reach the base case?

Back to Factorial

- **Factorial**: `n! = n x (n-1)!`

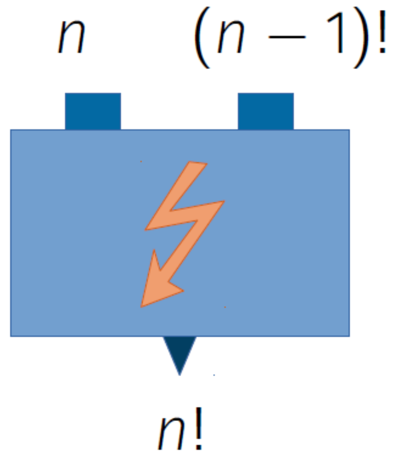
- **Base case:** n = 0: 0! = 1 (solved directly; no recursion)
- **Recursive case:** n > 0: n! = n x (n-1)!

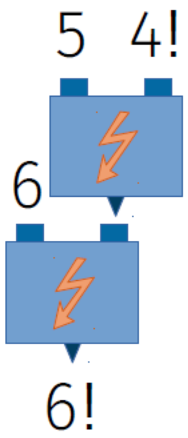
- *Advice: always put the base case first!*

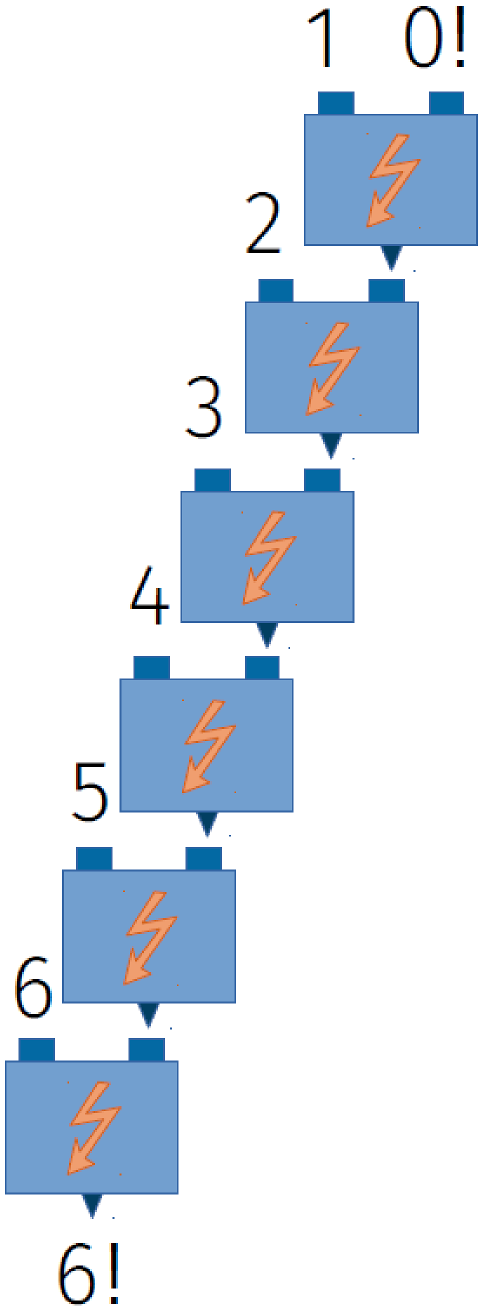
- *Let's convert this into code…*

Recursive Example: Factorial

- Factorial:
  - n! = n x (n-1) x (n-2) x … x 2 x 1
  - n! = n x (n-1)!
    - Solve by multiplying two numbers
    - Note: 0! = 1! = 1







Recursive Example: Factorial (Convert To Code)

```
public int factorial (int n) {
 if (n == 0)     //BASE CASE: n = 0:  0! = 1
          return 1;
 else            //Recursive Case: n! = n x (n-1)!
          return n * factorial(n-1);
}
```

Recursive Example: Factorial (Convert To Code)

```
public int factorial (int n) {
 if (n <= 0)    //BASE CASE: n = 0 → 0! = 1
    return 1;
 else           //Recursive Case: n! = n x (n-1)!
          return n * factorial(n-1);
}
```

- What if someone tries "-1"??
  Recursion can be tricky! *Always* need to stop at a base case!

Trace execution: Recursive Factorial (for n=5)

return 5 * factorial(4)

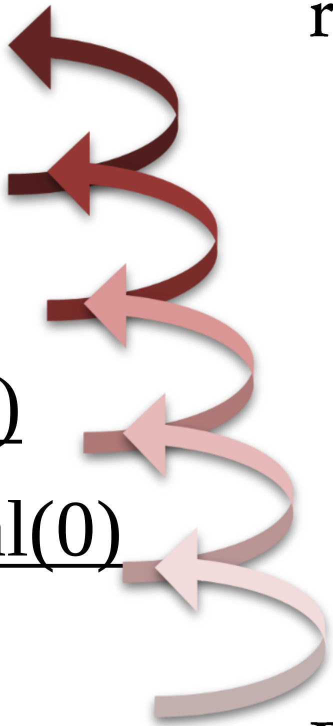
return 4 * factorial(3)

return 3 * factorial(2)

return 2 * factorial(1)

return 1 * factorial(0)

return 1



So… going **bottom to top**:

return 1 * (1)

return 2 * (1)

return 3 * (2 * 1)

return 4 * (3 * 2 * 1)

return 5 * (4 * 3 * 2 * 1)

END

Result: 5*4*3*2*1 = 5!

Why Do Recursive Methods Work?

- **Activation Records** on the **Run-time Stack** are the key:
  - Each time you call a function (any function) you get a new activation record
  - Each activation record contains a copy of all local variables and parameters for that invocation
  - The activation record remains on the stack until the function returns, then it is destroyed
- Try yourself: use your IDE’s debugger and put a breakpoint in the recursive algorithm. Look at the call-stack

Factorial Example, n=4 (Run-time stack)

- New area of memory set aside for function (“**fact**”) and its local variables
- Example showing the run-time stack with activation records
- Begin by calling the method, passing in the value Num=4

Num=4
MAIN **fact(4)**→

Num=4

4*fact(3)→

Num=4

MAIN

fact(4)→

Num=3

3*fact(2)→

Num=4

4*fact(3)→

Num=4

MAIN fact(4)→

Num=2

2*fact(1)→

Num=3

3*fact(2)→

Num=4

4*fact(3)→

Num=4

MAIN

fact(4)→

Num=1

1*fact(0)→

Num=2

2*fact(1)→

Num=3

3*fact(2)→

Num=4

4*fact(3)→

Num=4

MAIN fact(4)→

Num=0

return 1

Num=1

1*fact(0)→

Num=2

2*fact(1)→

Num=3

3*fact(2)→

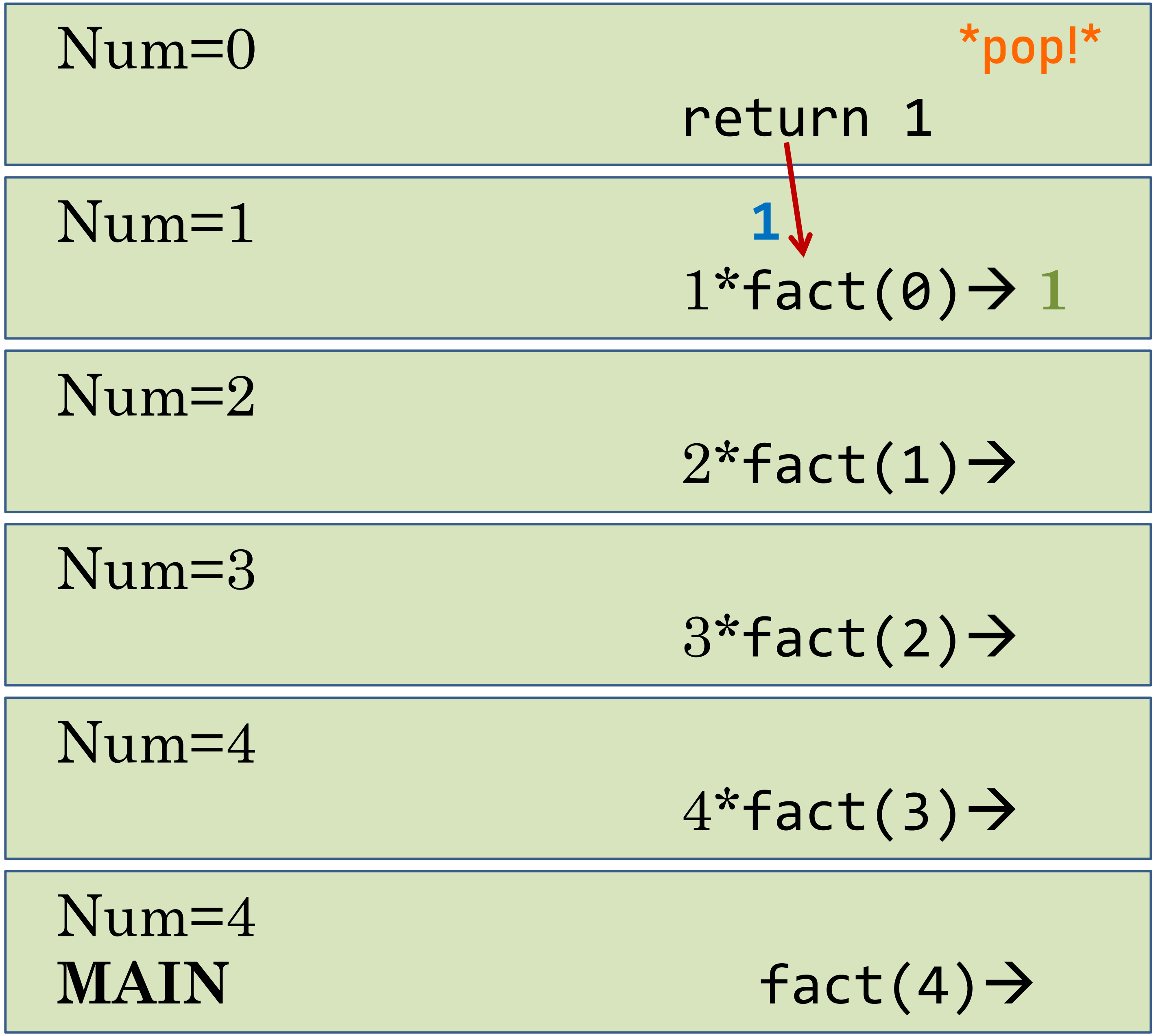
Num=4

4*fact(3)→

Num=4

MAIN

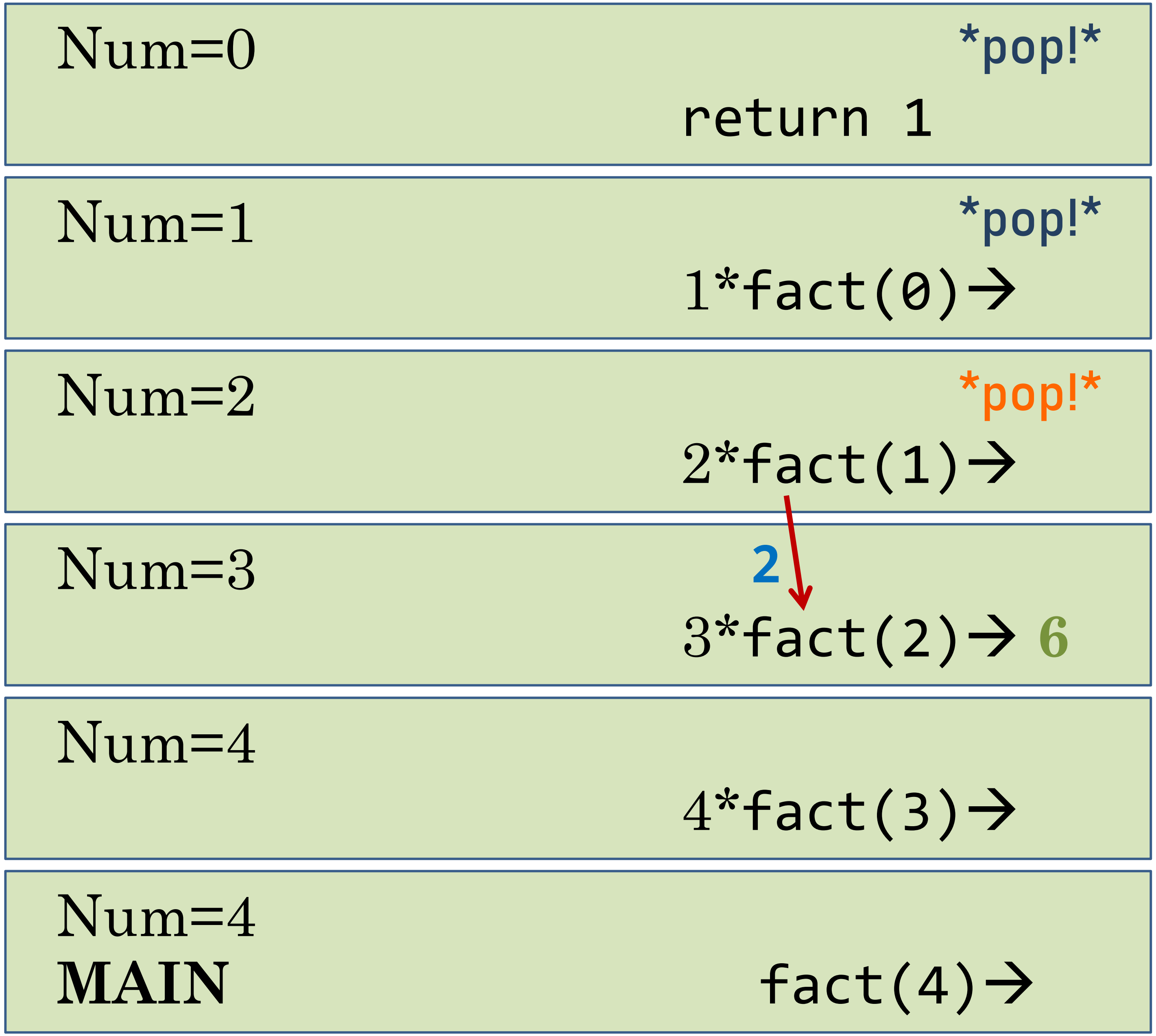
fact(4)→



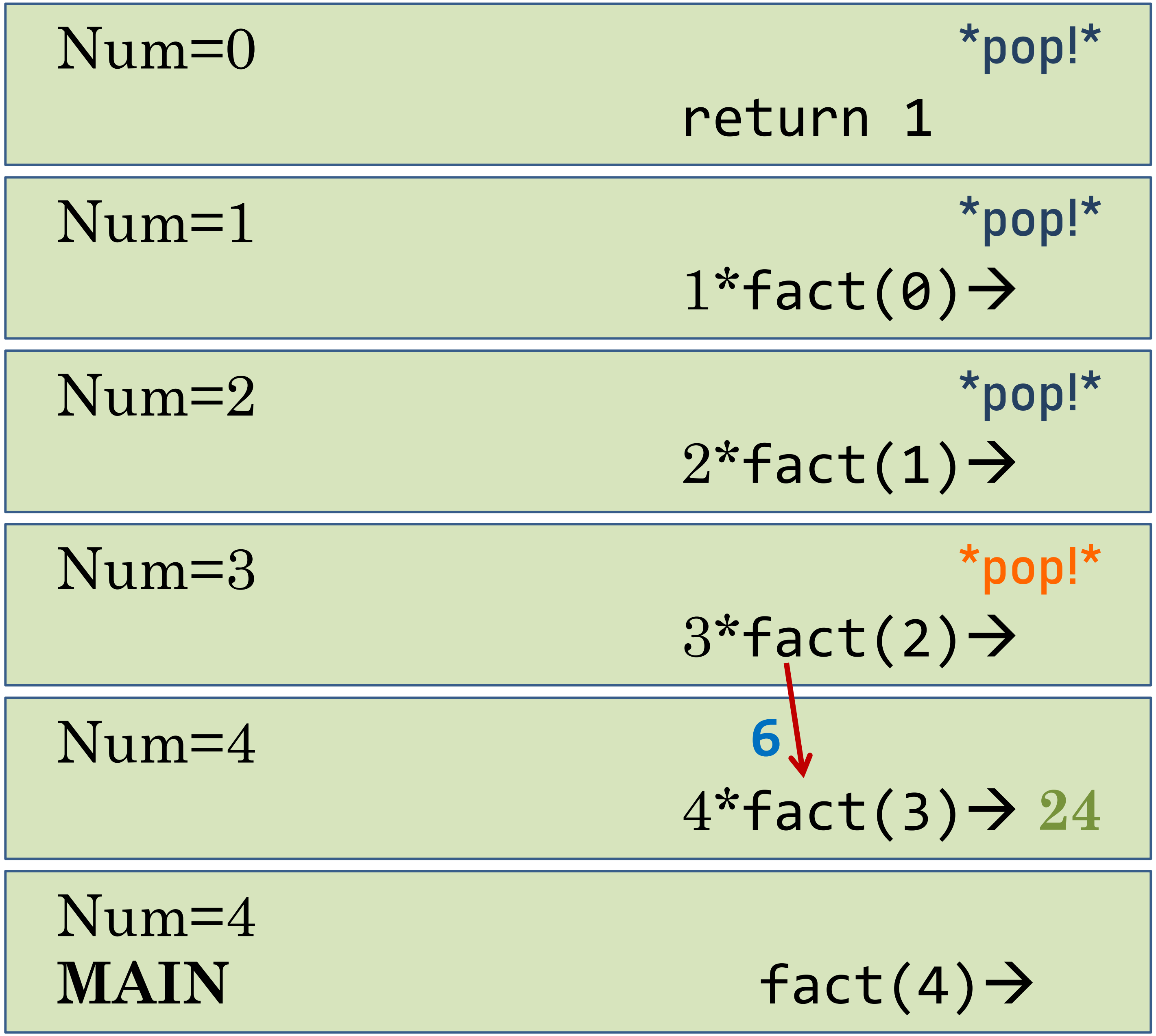
Note: Activation records will be **popped** off the stack (once the method returns) – it is just not shown here in this example (so you can see how it all works)

Num=0	*pop!* `return 1`
Num=1	*pop!* `1*fact(0)→`
Num=2	1 `2*fact(1)→` 2
Num=3	`3*fact(2)→`
Num=4	`4*fact(3)→`
Num=4 **MAIN**	`fact(4)→`

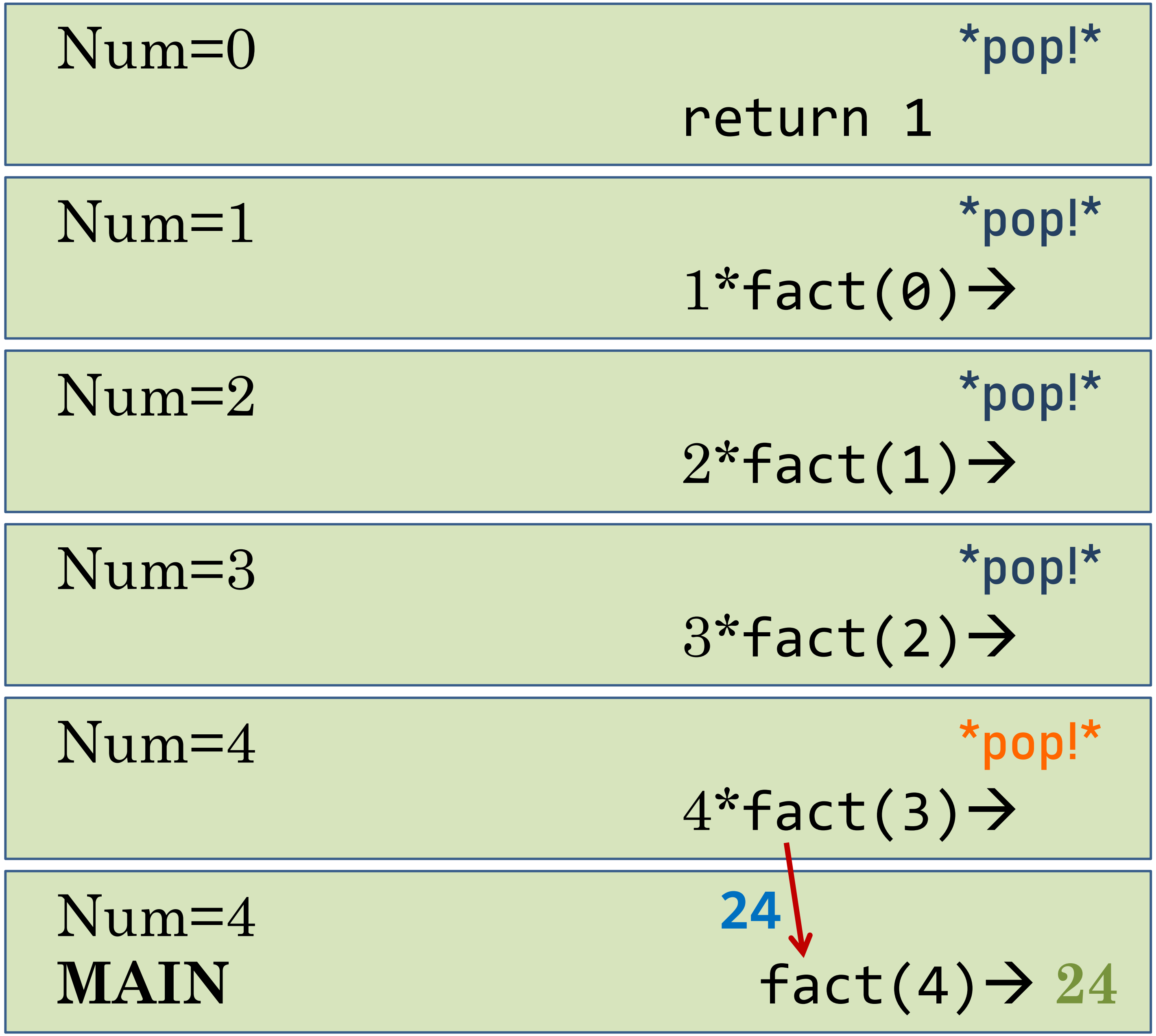
Note: Activation records will be **popped** off the stack (once the method returns) – it is just not shown here in this example (so you can see how it all works)



Note: Activation records will be **popped** off the stack (once the method returns) – it is just not shown here in this example (so you can see how it all works)



Note: Activation records will be **popped** off the stack (once the method returns) – it is just not shown here in this example (so you can see how it all works)



Note: Activation records will be **popped** off the stack (once the method returns) – it is just not shown here in this example (so you can see how it all works)

At the end the stack has popped off all activation records, and execution returns to who called the fact() method → Main

Num=4
MAIN fact(4)→ **24**

Extra Examples

Recursion vs. Iteration

Recursion

```java
public int factorial(int n) {
    // base case
    if (n <= 0)
        return 1;

    // recursive case
    return n * factorial(n-1);
}
```

Build solution from ***top down***

Iteration

```java
public int factorial(int n) {
    int fact_n = 1;

    for (int i = 1; i <= n; i++){
        fact_n = fact_n * i;
    }
    return fact_n;
}
```

Build solution from ***bottom up***

Extra Examples

Recursion vs. Iteration

Recursion

```
public int factorial(int n) {
    // base case
    if (n <= 0)
        return 1;

    // recursive case
    return n * factorial(n-1);
}
```

if stms, no loops

base case(s) before recursive call(s)

recursion: method calls itself

start at n and go down

Build solution from ***top down***

Iteration

```
public int factorial(int n) {
    int fact_n = 1;

    for (int i = 1; i <= n; i++){
        fact_n = fact_n * i;
    }
    return fact_n;
}
```

start at 1 and go up to n

for **loop**

accumulating value

Build solution from ***bottom up***

Broken Recursive Factorial {incorrect code: do NOT use/copy!}

```
public static int Brokenfactorial(int n){
   int x = Brokenfactorial(n-1);
   if (n <= 0)
        return 1;
   else
        return n * x;
}
```

- ***What's wrong here?***
  - Trace calls "by hand"

Broken Recursive Factorial {incorrect code: don't use/copy!}

```
public static int Brokenfactorial(int n){
    int x = Brokenfactorial(n-1);
    if (n <= 0)
          return 1;
    else
          return n * x;
}
```

- ***What's wrong here?*** Trace calls "by hand"
  - BrFact(2) -> BrFact(1) -> BrFact(0) -> BrFact(-1) -> BrFact(-2) -> …
  - Problem: we do the recursive call **first** before checking for the base case
  - **Never stops!** Like an infinite loop!

Recursive Design

- Recursive methods/functions require:
  1. One or more (non-recursive) **base cases** that will cause the recursion to end

     ```
     if (n <= 0) return 1;
     ```

  2. One or more **recursive cases** that operate on smaller problems and get you *closer* to the base case

     ```
     return n * factorial(n-1);
     ```

- Note: The base case(s) should **always** be checked **before** the recursive call(s)

Summary

- Recursive problem can be broken into two parts:
  - Base case: The case for which the solution can be stated non-recursively
  - Recursive case: The case for which the solution is expressed in terms of a smaller version of itself

- Recursion is tricky!
  - Always put the base case first! (If more than one, put all of them first!)
  - Base case should eventually happen given ANY input
  - Recursive call should always get us closer to base case(s)
  - Recursive solution may not always be the best (even though it might look nice!)

37

More Recursive Examples

Seeing many examples will help!

Iterative Example: Printing A List

- Here's a method that prints a simple list iteratively:

```
public void printList(int[] list){
    for(int i = 0; i < list.length; i++){
        System.out.println(list[i] + " ");
    }
}
```

- What about printing recursively?

  Pseudocode:

```
//As long as the list is not empty
//Print one item in list (current position; starting at zero)
//Then print the REST of the list recursively
```

Iterative vs Recursive Example: Printing A List

- Here's a method that prints a simple list iteratively:

```
public void printList(int[] list){
    for(int i = 0; i < list.length; i++){
        System.out.println(list[i] + " ");
    }
}
```

- Here's a method that does the same thing, but recursively:

```
public void printList(int[] list, int curIndex){
    //Base case, if curIndex has run off end of list, do nothing
    if(curIndex >= list.length) return;
    //print one element and then recursively print the rest
    System.out.print(list[curIndex] + " ");
    printList(list, curIndex+1);  }
```

Recursive Example: Printing A List (using a helper method)

- Those who use our code might not know what curIndex is… And might not realize we have to set it at zero. So, we use a helper method!

```
public void printList(int[] list) { // public method
    printList(list, 0); //print starting at index 0 (already set!)
}

//private so nobody can invoke this method directly
private void printList(int[] list, int curIndex){

    //Base case
    if(curIndex >= list.length) return;

    //print one element and then recursively print the rest
    System.out.print(list[curIndex] + " ");
    printList(list, curIndex+1);
}
```

Recursive Example: Binary Search [pseudocode]

- Let's say we're trying to find a particular page in a textbook using Binary Search:

```
find(page_number, book) {
    flip to middle;
    if page == page_number
        return found;
    if page_number is before page
        return find(page_number, first half); // search 1st half
    if page_number is after page
        return find(page_number, second half); // search 2nd half
}
```

Recursive Example: Binary Search [pseudocode]

- More general Binary Search algorithm (pseudocode)

```
public static int binarySearch(int[] list, int value) {
       return binSearch(list, target, 0, list.length -1); //start: entire list is valid
}
private static int binSearch(int[] list, int first, int last, int target) {
    //Base Case: if no where left to look (if low > high) return (-1)
    //Calculate mid (an int)
    //Print mid – the item that is being compared
    //if mid is equal to target, return mid
    //else if mid is less than the target,  first = mid + 1 (target in top half)
    //else (mid is greater than the target), last = mid – 1 (target in bottom half)
    //return [a recursive call to binSearch, passing values list, first, last, target]
}
```

Recursive Example: Binary Search

- This **Binary Search** algorithm has an **int** return type. *What does the returned **int** represent?* It could also be boolean. How would you change it? *[Hint: not many things will change.]*

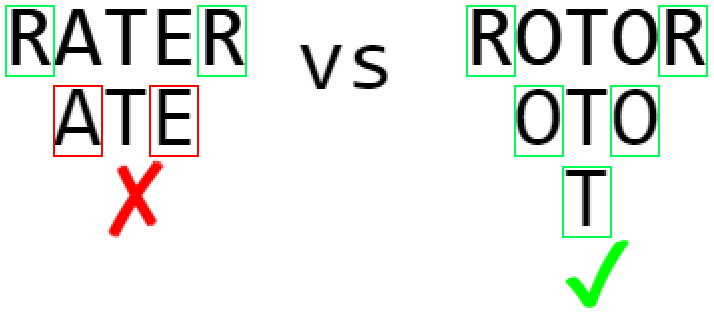
```
int binSearch(int[] array, int first, int last, int target) {
    if (first <= last) {
        int mid = (first + last) / 2;
        if (target == array[mid])
            return mid;
        if (target < array[mid])
            return binSearch(array, first, mid - 1, target);
        else if (target > array[mid]);
            return binSearch(array, mid + 1, last, target);
    }
    return -1;
}
```

Recursive Example: Palindrome

- The word palindrome is derived from the Greek *palíndromos*, meaning running back again (**palín** = AGAIN + **drom**–, **drameîn** = RUN)
- A word that is a palindrome can be read the same in both directions. Some simple examples are:

RACECAR LEVEL CIVIC DEED

- An **empty string** or **a single character** is a palindrome. Larger words: From out to in, characters must match (see next slide)



OVERALL IDEA:

- Test first and last character only
  - If they match AND
  - Everything inside is also a palindrome, then TRUE!

Recursive Example: Palindrome

- Let’s assume the method is called isPalindrome()
- This will test to see if a given string is a palindrome

```
public boolean isPalindrome(String s, int l, int r){
    //Base case
    if(l > r) return true;

    //Recursive call: if outside chars match and inside is Palindrome, then return true
    return (s.charAt(l) == s.charAt(r))
            && isPalindrome(s, l+1, r-1);
}
```

Recursive Example: Palindrome (using a helper method)

```java
public boolean isPalindrome(String s) {
    return isPalindrome(s, 0, s.length()-1);
}

private boolean isPalindrome(String s, int l, int r){
    //Base case
    if(l > r) return true;

    //Recursive call: if outside chars match and inside is Palindrome, then return true
    return (s.charAt(l) == s.charAt(r))
            && isPalindrome(s, l+1, r-1);
}
```

Extra Examples

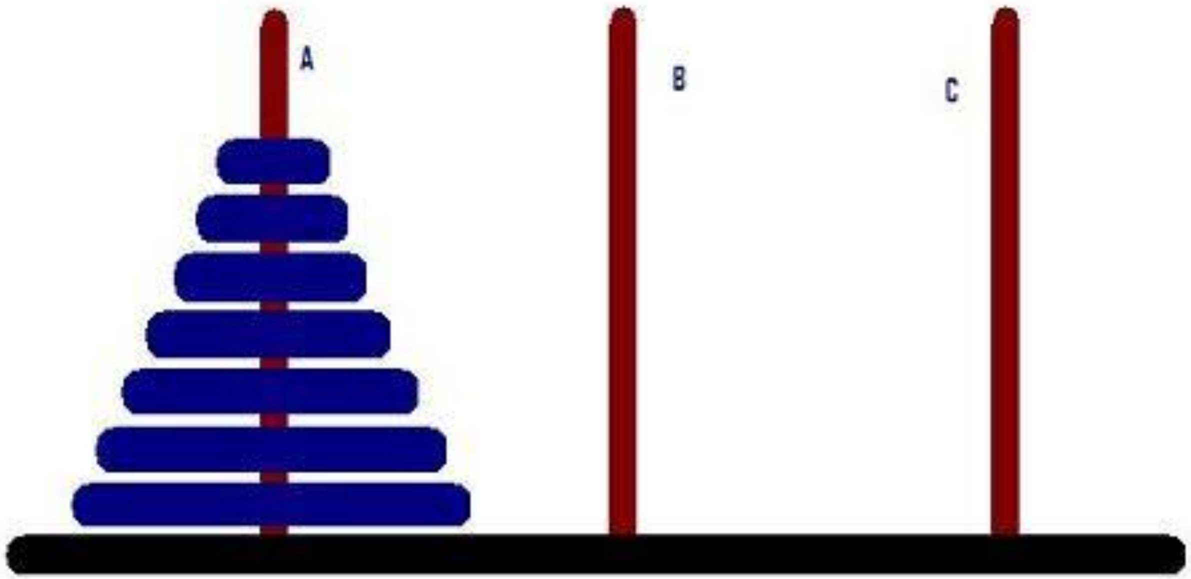
Recursive Example: Palindrome [Another Solution]

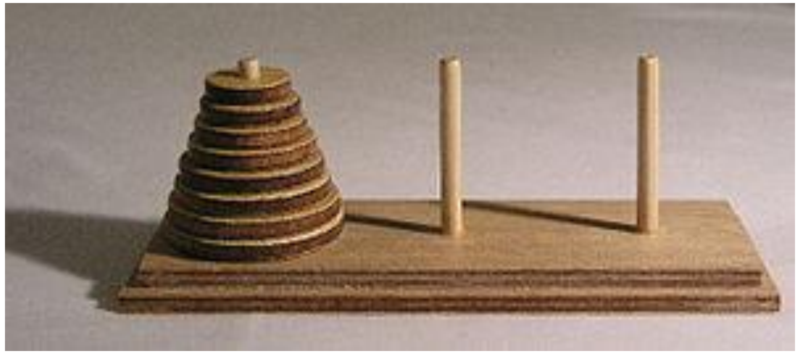
```
public static boolean palindrome (String s) {
      if (s.length() == 0  ||  s.length() == 1) // base cases, length is 0 or 1
          return true; // an empty string or a single character is a Palindrome

      if (s.charAt(0) == s.charAt(s.length()-1)) { // if first == last character
          // Uncomment the next TWO lines to see recursive palindrome() in action!
          System.out.print(s.charAt(0) + " and " + s.charAt(s.length()-1) + " match! ");
          System.out.println("Trying: " + s.substring(1, s.length()-1));
          // recursive call: call palindrome on the rest of the string:
          return palindrome(s.substring(1, s.length()-1));
          // Note: if string length = 5, s.substring goes from indices 1 --> 3
          //       i.e. up to, but NOT including, the second parameter (5-1=4)
          //       (New string sent in recursive call is old string with first and
          //       last characters removed)
      }
      return false; // If the first and last characters don't match, return false
  }
```

Extra Examples

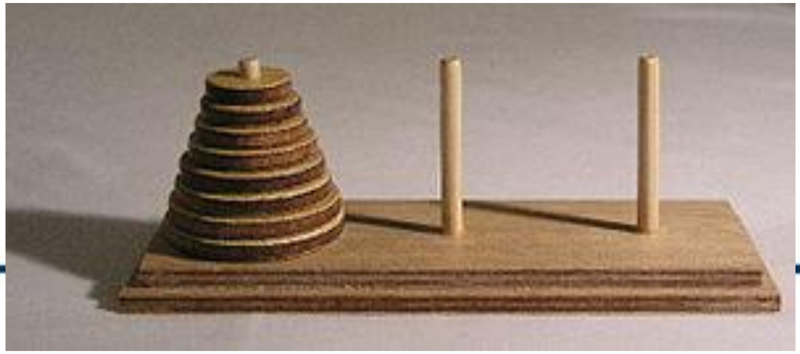
Other Recursive Examples

- Towers of Hanoi
- Euclid's Algorithm
- Fractals
- General activities like
  - Is string a Palindrome?
  - Reverse a String
  - …





Towers of Hanoi



- A game that is old and famous!
- The objective is to transfer entire tower A to the peg B, moving only one disk at a time and never moving a larger one onto a smaller one
  - The algorithm to transfer n disks from A to B in general: We first transfer ***n - 1*** smallest disks to peg C, then move the largest one to the peg B and finally transfer the ***n - 1*** smallest back onto largest (peg B)
  - The number of necessary moves to transfer ***n*** disks can be found by $T(n) = 2^n - 1$

Extra Examples

Euclid's Algorithm

- Calculating the greatest common divisor (gcd) of two positive integers is the largest integer that divides evenly into both of them
  - E.g. greatest common divisor of 102 and 68 is 34 since both 102 and 68 are multiples of 34, but no integer larger than 34 divides evenly into 102 and 68
  - Logic: If p > q, the gcd of p and q is the same as the gcd of q and p % q (where % is the remainder operator)
  - Stop recursion once q becomes zero; at which point return p